



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Security in Microservice Decomposition - An architectural approach to support microservice security“

verfasst von / submitted by

Katharina Hahn, B.A.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

UA 066 926

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Masterstudium Wirtschaftsinformatik

Betreut von / Supervisor:

Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I would like to thank Mag. Dr.Georg Simhandl for the support and guidance during the time of my master's thesis. Your expertise and support have been a great motivation during the whole research work.

Appreciation is also extended to Univ.-Prof. Dr.Uwe Zdun for supervising my master's thesis.

I want to extend my gratitude to my friends and family for their support during this period of my life.

Abstract

With the increasing complexity of modern software applications, the importance of microservices is rising. Microservices are gaining popularity as an architectural pattern, offering the ability to break down complex applications into small, independently developed, and deployable services. This is achieved by migrating monolithic architectures into microservices, the so-called microservice decomposition. Microservice decomposition refers to dividing a monolithic application into separate, self-contained services. Because of loose coupling, each microservice is responsible for a specific function and can be developed, tested, and updated independently of other services. During microservice decomposition, the security risks that arise are neglected. Separating the monolith into several microservices increases the attack surface since each microservice is exposed via an entry point to communicate with external clients or other microservices. Therefore, the old security mechanisms are obsolete, since this new microservice structure entails other safety requirements which need new security measures to be implemented.

Therefore, this master's thesis aims to develop a security-oriented tool that accompanies the user on an architectural level from the start to create a secure component model of the microservice architecture. Furthermore, an essential functionality of the tool includes a constant evaluation of the achieved security level during the modeling process.

The developed tool, the so-called DevBot system, supports the developer in implementing appropriate external and internal security patterns and provides a continuous evaluation of the security risks of the model. In addition, the DevBot logs all security design decisions made to enable end-to-end traceability. Finally, the DevBot system evaluates the created microservice architecture against the risks identified by OWASP.

The DevBot system presents a security-oriented tool in microservice decomposition that provides easy and intuitive usage for developers without modeling knowledge and supports the creation of microservice component models and extensibility to new security patterns.

Kurzfassung

Mit der zunehmenden Komplexität moderner Softwareanwendungen wächst die Bedeutung von Microservices. Microservices werden als Architekturmuster immer beliebter, da sie die Möglichkeit bieten, komplexe Anwendungen in kleine, unabhängig entwickelte und einsetzbare Dienste zu zerlegen. Dies wird erreicht, indem monolithische Architekturen in Microservices umgewandelt werden, die sogenannte Microservice Decomposition. Die Microservice Decomposition bezieht sich auf den Prozess der Zerlegung einer monolithischen Anwendung in separate, in sich geschlossene Dienste. Jeder Microservice ist für eine bestimmte Funktion zuständig und kann aufgrund der losen Kopplung unabhängig von anderen Services entwickelt, getestet und modifiziert werden. Bei der Microservice Decomposition werden die entstehenden Sicherheitsrisiken allerdings vernachlässigt. Die Zerlegung des Monolithen in mehrere Microservices vergrößert die Angriffsfläche, da jeder Microservice über einen Einsprungpunkt mit externen Clients oder anderen Microservices kommunizieren kann. Daher sind die alten Sicherheitsmechanismen obsolet, da diese neue Microservice-Struktur andere Sicherheitsanforderungen mit sich bringt, die die Implementierung neuer Sicherheitsmaßnahmen erfordern.

Diese Masterarbeit befasst sich daher mit der Entwicklung eines sicherheitsorientierten Werkzeugs, das den Anwender von Anfang an auf Architekturebene begleitet, um ein sicheres Komponentenmodell der Microservicearchitektur zu erstellen. Eine wesentliche Funktionalität des Werkzeugs ist darüber hinaus die kontinuierliche Evaluierung des erreichten Sicherheitsniveaus während des Erstellungsprozesses.

Das entwickelte Werkzeug, das sogenannte DevBot-System, unterstützt den Entwickler bei der Implementierung geeigneter externer und interner Sicherheitsmuster und bietet eine kontinuierliche Evaluation der Sicherheitsrisiken des Modells. Darüber hinaus protokolliert der DevBot alle getroffenen Entscheidungen zum Sicherheitsdesign, um eine durchgängige Nachvollziehbarkeit zu ermöglichen. Schließlich bewertet das DevBot-System die erstellte Microservicearchitektur anhand der von OWASP identifizierten Risiken.

Durch die Fokussierung auf die Sicherheitsmängel bei der Microservice-Dekomposition ist es das Ziel dieser Arbeit eine sicherheitsunterstützende Anwendung für die Microservice-Dekomposition zu entwickeln, die für Entwickler ohne Modellierungskenntnisse einfach und intuitiv zu bedienen ist. Durch die schnelle Generierung von Komponentenmodellen und die Erweiterbarkeit um neue Sicherheitspatterns stellt das DevBot-System ein erstes Werkzeug für eine sicherheitsorientierte Applikation in der Microservice-Dekomposition dar.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	v
List of Tables	xi
List of Figures	xiii
Listings	xv
1. Introduction	1
1.1. Problem definition	1
1.2. Goals and contribution	2
1.3. Thesis overview	3
2. Background	5
2.1. Service-oriented Architecture	5
2.2. Monolith vs. Microservice architecture	6
2.3. Microservice decomposition	7
2.4. Microservice patterns	8
2.4.1. External security patterns	9
2.4.2. Internal security patterns	12
2.5. Microservice security	14
2.5.1. Identity and Access Management	14
3. State of the Art	17
3.1. Objective	17
3.2. Decomposition approaches	17
3.3. RQA.1: Microservice decomposition approaches in academic literature . .	18
3.3.1. Tool-based approaches of microservice decomposition	18
3.3.2. Model-driven approaches of microservice decomposition	20
3.4. Microservice decomposition approaches in gray literature	21
3.5. RQB: Addressing security aspects in microservice decomposition	22
3.6. Derivation of Research Questions from the Literature Analysis	23

4. Methods	25
4.1. Design Science Research	25
4.1.1. Design Science Research guidelines	26
4.1.2. Relevance Cycle	28
4.1.3. Rigor Cycle	29
4.2. Application design	30
4.2.1. Security risks	30
4.2.2. Decision tree	31
5. Implementation	37
5.1. System design	37
5.1.1. DevBot GUI	38
5.1.2. DevBot Backend	39
5.2. Use case scenarios	41
5.2.1. Use Case 1: Manage Microservices	41
5.2.2. Use Case 2: Manage Endpoints	42
5.2.3. Use Case 3: Manage External Security	43
5.2.4. Use Case 4: Manage Internal Security	45
5.2.5. Use Case 5: Manage Microservice Database	45
5.2.6. Use Case 6: Evaluate Security	47
5.2.7. Use Case 7: Manage Component Model	47
5.3. Requirements	48
5.4. User Interface	51
5.4.1. Development process	51
5.4.2. User instructions	54
5.5. Extensions	61
5.5.1. Benefits of extensibility	61
5.5.2. How to extend external security patterns	62
6. Discussion	65
6.1. Conformance checking of the DevBot system	66
6.1.1. Evaluating microservice security	66
6.1.2. Calculating security risk level	66
6.1.3. Updating microservice security	67
6.2. System validation	68
7. Conclusion	73
7.1. Threats to validity	73
7.2. Learnings and Limitation	74
7.3. Future work	74
Bibliography	75

A. Appendix	85
A.1. Model extension	85
A.2. Conformance checking	86
A.3. Model validation	90

List of Tables

2.1. Selection of possible microservice patterns	9
3.1. Overview of the number of certain keywords in the examined papers . . .	23
5.1. Overview of system requirements	50
6.1. Validation of DevBot system with annotated models	70
6.2. Comparison of DevBot approach and manual approach for microservice architecture	72

List of Figures

2.1. Comparison of a monolithic and microservice architecture	6
2.2. Overview of usage of Service Mesh	10
2.3. Overview of usage of API gateway	11
2.4. Overview of Message Queue Pattern	12
2.5. Overview of Publisher/Subscriber Pattern	14
3.1. Overview of the general steps of microservice decomposition approaches . .	18
4.1. Overview of the overall approach of this thesis	25
4.2. Decision model of the security helper	32
4.3. Submodel 1: Decision model for the general need of external security . . .	33
4.4. Submodel 2: Decision model for specific external security options	34
4.5. Submodel 3: Decision model for the general need of internal security . . .	34
4.6. Submodel 4: Decision model for specific internal security options	35
5.1. Overview of the system design	37
5.2. Bird eye view of the system	38
5.3. Development view of the DevBot application	39
5.4. Logical view of the DevBot application	40
5.5. Overview of all considered scenarios of the DevBot system	41
5.6. Sequence diagram of the microservice management (UC1)	42
5.7. Sequence diagram of the endpoint management (UC2)	43
5.8. Sequence diagram of the external security management (UC3)	44
5.9. Sequence diagram of the internal security management (UC4)	46
5.10. Sequence diagram of the database management (UC5)	46
5.11. Sequence diagram of the security evaluation (UC6)	48
5.12. Sequence diagram of the component model management (UC7)	49
5.13. Overview of "Add endpoints" and "Add internal security" views optimization	52
5.14. Overview of "Evaluate model" view optimization	53
5.15. Overview of "Home" view optimization	54
5.16. Create new model: Start application	55
5.17. Create new model: Add microservice and endpoint components	55
5.18. Create new model: Check current model security	56
5.19. Create new model: Add external security and check model security	56
5.20. Create new model: Add internal security and databases	57
5.21. Create new model: Check model security and save created model	58
5.22. Edit model: Load model and check model security	59

List of Figures

5.23. Edit model: Update external security and check model security	60
5.24. Edit model: Update internal security and check model security	61
5.25. Comparison of ease of extensibility	62
5.26. Overview of the updated GUI with new extension	64
6.1. Evaluation of system	69
A.1. Evaluation of CI1	90
A.2. Evaluation of AC1	91
A.3. Evaluation of CO0	92
A.4. Evaluation of PM2	93
A.5. Evaluation of RS0	94

Listings

5.1. Add new external security component	62
5.2. Function to add external security extension	62
A.1. Function to update or change external security	85
A.2. Modify if statements in supporting function	85
A.3. Modify gui functionality to use extension in DevBot gui	85
A.4. Identification of vulnerable microservices	86
A.5. Identification of vulnerable endpoints	86
A.6. Calculation of overall security level	87
A.7. Update of external security	88
A.8. Update of internal security	88

1. Introduction

In recent years, there has been an increasing interest in microservices from the industrial and academic world [1]. Since software systems are getting more complex and are deployed via cloud services, microservice architecture offers more flexibility by being loosely coupled and independently deployable. In the industry, typically legacy systems can be found, but maintaining these systems is inherently expensive due to poor modularization [2]. Therefore, industries are modernizing these systems by extracting features into microservices [3].

Microservices consist of small services built independently. The goal is that each microservice is responsible for one thing only and can be deployed independently, which leads to more fault tolerance [4]. In addition, it allows polyglot programming, which is a great advantage, as developers can work with the programming language that suits them best. Due to their size, microservices are easier to maintain. All these benefits of microservices lead to a concentration on specialized teams that can perfectly use their know-how for their task area [5].

Besides the advantages of using microservices, microservices also face a lot of challenges, since the complexity of communication and security mechanisms increases. Investing in tools and processes to manage service discovery, load balancing, and Application Programming Interface (API) management is necessary. Ensuring that each service is adequately secured and that data is transmitted securely between services is essential for a successful microservice decomposition [6].

As the usage of microservice architecture increases, tools are developed to support this process. To migrate a monolith to microservices, Evans [7] presents for example the Domain-driven Design (DDD) approach. This approach starts at an architectural level and provides guidelines to revise the monolithic architecture to identify microservices. Besides model-driven approaches, there are also tools, that help to migrate monolith to microservices based on the source code. These tools analyze the code and use algorithms to determine bounded contexts by creating dependency graphs. These dependency graphs form the basis for the possible microservice architecture [8, 9, 10].

1.1. Problem definition

However, during the microservice decomposition, security aspects are often neglected [11, 12]. New external exposed interfaces are not taken into account enough. This is shown by the example of the Uber breach, which occurred in 2016 [13]. Hackers gained access to a database containing the personal information of over 57 million users and drivers. The

1. Introduction

breach was caused by a vulnerability in one of Uber’s microservices, which allowed the hackers to access a database containing sensitive information. One of the challenges that Uber faced in this incident was the complexity of their microservice architecture, which made it difficult to identify and address vulnerabilities [14]. In addition, the distributed nature of the architecture made it challenging to ensure consistent security controls across all services [14, 13].

This example highlights the importance of implementing appropriate security controls when decomposing a monolithic architecture into microservices. Companies must prioritize security and continuously monitor and manage security risks to avoid serious breaches and data loss when their microservice architecture evolves. Securing microservices is crucial, especially when dealing with sensitive data [15].

However, this indicates a need to understand the various perceptions of microservice security that exist among them. The above-mentioned example shows that security aspects are often considered insignificant. Therefore, this master thesis addresses the lack of concern for microservice security tactics and attempts to mitigate these security issues. For this purpose, I want to support the user during the microservice decomposition and raise awareness of security-relevant aspects, which need to be included to secure microservices at an early stage. This should prevent security breaches at an advanced stage by assisting the developer with decomposing a monolith into a microservice structure.

1.2. Goals and contribution

The main challenge faced by many researchers is the implementation of microservice security. Accordingly, the idea of this master thesis is to support the developer in security aspects during the process of migration. The goal of this thesis is to:

- Based on an architectural level, develop an interactive security-oriented tool (DevBot) to create a component model for microservices including all microservice security aspects
- Constantly evaluate the achieved security level during the modeling process.

To achieve the proposed goal, I need to know which tools exist to support security in microservice decomposition and which microservice security patterns need to be considered. Therefore, in the multi-vocal literature review phase, existing decomposition tools and their security support are investigated.

Our security-oriented tool, the DevBot, aims to provide developers with an intuitive tool that allows them to quickly and easily generate a component diagram with a particular focus on microservice security. The intention of the DevBot is not to become another modeling tool, but to support the user in developing a secure microservice architecture, adding security-relevant components automatically and constantly evaluating the security level. The developer can interactively select the appropriate external and internal security

mechanisms on an architectural level, which is logged and incorporated into a Unified Modeling Language (UML) model. The developer does not need any specific knowledge in modeling or a modeling tool to create the component model with the help of the DevBot.

The main contributions of the DevBot are as follows:

1. Security supporting tool for microservice decomposition
2. Easy and intuitive usage for developers without modeling knowledge
3. Quick generation of component model
4. Extensibility to new security patterns

1.3. Thesis overview

In chapter 2, the most important terms and security concepts for microservice architecture are explained. Especially, the security standards for Identity and Access Management are described in detail. In chapter 3 the State of the Art is presented, which aims to identify and classify the existing tools to decompose a monolith system into microservices. The research gap is derived accordingly, and the research questions are outlined. Chapter 4 presents the method used for developing a security-oriented modeling tool, the DevBot application. The reached design decisions for implementation are described, and a security design decision tree is developed, which represents the base for the DevBot application. In chapter 5 the components of the DevBot are described in detail, and the operation flow is presented. Furthermore, a detailed overview of the GUI is given and the extensibility of the DevBot system is demonstrated. Chapter 6 contains the evaluation process of the model and the actual DevBot application. This includes an assessment of the achieved results by executing the DevBot application. Finally, in chapter 7 a summary of the obtained results and the future work that can be done to continue the exploration in this area of research are given.

2. Background

For a better understanding of the field of microservice decomposition, the most important terms are described, and relevant aspects are explained in more detail.

2.1. Service-oriented Architecture

Service-oriented Architecture (SOA) represents significant progress in application development and integration. Before emerging in the late 1990s, connecting applications to external systems required complex point-to-point integrations, which had to be recreated for each project separately [16]. With SOA, developers are enabled to easily reuse existing functionality and connect through the SOA Enterprise Service Bus (ESB) architecture. An ESB provides a centralized software component to handle transformations, messaging, and routing of data models and enables reusability by new applications [16].

The first generation of service-oriented systems relied on monolithic components, configurable at compile time. The current generation of service-oriented systems is built on vertically integrated components, which are customizable and reconfigurable at installation time and, to a certain degree, at runtime [17].

SOA is an approach that enables software components to be reusable and interoperable via service interfaces. Organizations are enabled to create applications by combining various independent services, these services can also be accessed and used by other applications. The process of incorporating existing functionalities is simplified because of the use of common interface standards and an architectural pattern. As a result, application developers are no longer required to duplicate or rebuild existing functionality and can focus on connecting and integrating different services [16]. In a SOA, each service provides a self-contained unit encapsulating the code and data required to perform a particular business function. By providing loose coupling and thereby reducing dependencies between applications, service interfaces can be accessed without detailed knowledge of the underlying implementation. Services can be built from scratch or, by connecting them via SOA, which are implemented in the legacy system, via service interfaces [16, 18].

SOA offers significant progress in the area of application development, from which microservice architecture has evolved. The main difference between SOA and microservice architecture is its scope, while SOA has an enterprise scope, the microservice architecture follows an application scope. Other key differences between these two architectural approaches are communication, interoperability, and data management [18]. In general, the microservice architecture focuses on the independence of microservices, where each microservice uses its own independently deployed communication protocol along with

2. Background

lightweight messaging protocols. In SOA, a shared communications mechanism (ESB) is used to coordinate and manage all services, and rather heterogeneous messaging protocols are used. Regarding data management, in a microservice architecture, each microservice has its database for data storage, while in SOA a common data storage layer is shared with common resources [18].

2.2. Monolith vs. Microservice architecture

To understand the microservice architecture, it is helpful to compare it with the monolithic style. A *monolith* is an application that is built as a single, cohesive executable and interconnected unit which typically consists of three main components (see Figure 2.1): a client-side user interface, a database, and a server-side application [5]. The server-side application handles Hypertext Transfer Protocol (HTTP) requests, executes domain logic, interacts with the database, and generates Hypertext Markup Language (HTML) views for the browser which graphically represents the monolith. Any changes to the system require building and deploying a new version of the entire server-side application. All the request-handling logic resides in a single process, allowing developers to organize the application into classes, functions, and namespaces. Horizontal scaling of the monolith is achieved by running multiple instances behind a load balancer [5].

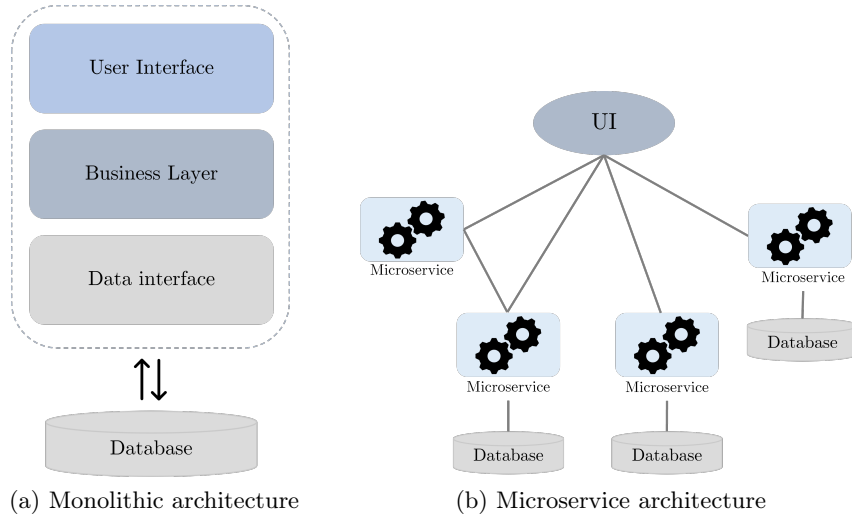


Figure 2.1.: Comparison of a monolithic and microservice architecture in accordance with [19]

While monolithic applications can be successful, the requirements for services are changing, since more applications are deployed in the cloud. Change cycles are tightly coupled, meaning that a modification to a small part of the application necessitates rebuilding and deploying the entire monolith [20]. Over time, with increasing complexity, maintaining a good modular structure becomes challenging, making it difficult to isolate

changes that should only affect specific modules [5].

Because of these disadvantages of a monolith, the microservice architectural style has emerged, which involves constructing applications as collections of services, which is shown in Figure 2.1. Lewis and Fowler [5] describe *microservices* as the following: "[T]he microservice architectural style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often a HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery."

This architecture provides many benefits, including increased agility, scalability, and resilience. Since microservices are deployed independently, they have the advantage of being technology-agnostic. Therefore, each service can use the best-suited technologies and programming language, which can lead to better results by using the full know-how of the developer and can enable organizations to move faster and respond more quickly to changing business requirements. The efficient horizontal scalability around business capabilities simplifies the process to handle variable workloads and manage resources more efficiently [21].

Exposing business capabilities can be done via one or more network endpoints which are also used for communication between services, which ensures that failures in one service do not necessarily affect the entire application. To amplify the independence, each microservice encapsulates its data storage. Hence, databases are hidden inside the microservice boundaries [22]. Database per Service is one possible option to store data. In section 2.4 several data management patterns are presented. Overall, a microservice is only responsible for one thing and can operate independently [22].

2.3. Microservice decomposition

Since microservices offer many advantages, the next step is to examine the development process. Developing microservices can be achieved by different approaches. It can either be developed from scratch, or an existing system can be decomposed into a microservice. The latter is known as microservice decomposition. Microservice decomposition is the process of generating microservices from an existing monolith. During microservice decomposition, the monolith is analyzed and microservices are identified by grouping business logic into suitable sets [6]. There are several approaches to performing this process. There is a tool-based approach and a model-driven approach to decompose a monolith into a microservice. This will be discussed in more detail in chapter 3.2.

Since the single unit is broken up into several microservices, previously used security mechanisms are no longer adequate, which introduces new challenges and complexities that need to be managed. Besides the advantages of microservices, the security aspects are more complex and can become a bigger problem than with monolithic systems. The decomposition increases the attack surface, by communicating with each other through

2. Background

APIs [6].

This can create a complex web of dependencies that can be difficult to manage. The segmentation and decoupling of the microservices lead to several new entry points for each microservice. Application security is more difficult to implement because security is a global property. By developing and deploying each service independently, it can be challenging to ensure that security standards are consistently applied across the entire application. Especially, the communication between microservices is exposed over the network and creates a potential attack surface [6].

Accordingly, *Identity and Access Management* needs to be up-to-date for all microservices. Possible microservice security options will be discussed in the next section.

2.4. Microservice patterns

To secure microservice architectures in the course of microservice decomposition, appropriate security patterns have to be applied. Before security-specific patterns are presented, general microservice patterns are introduced.

To establish a design pattern, the key aspects of a general design structure need to be identified and abstracted, which enables the pattern to be reused in the development of an object-oriented design [23]. In a microservice architecture, Taibi et al. [24] identify three categories for emerging architecture patterns: Data Storage patterns for data management in microservices, Deployment Strategy patterns for managing containers, and Orchestration and Coordination architecture patterns for communication and coordination purposes. On this basis, several patterns are identified and assigned to a category. Richardson [25] subdivides patterns into even smaller categories and establishes several patterns for microservice architecture especially. A selection of these patterns can be seen in Table 2.1.

According to the context, each pattern shows the problem and the forces, while it provides a solution and an example. Application architecture and Migration patterns offer support during the basic design decision process. After choosing a microservice architecture, Data management patterns present possible options to maintain data consistency. The options range from shared databases to one database for each service, which provides possible options for sharing information between services. Deploying a microservice architecture is also an important decision [25]. Therefore, deployment patterns help to find a suitable deployment strategy, whether it is by deploying each service in its host or virtual machine or using a highly automated deployment platform. Observability patterns are used to understand the behavior of a microservice application and detect troubleshooting problems, while user interface patterns provide options for displaying data from multiple services. The Orchestration and Communication patterns offer possible solutions for external and internal microservice communication [25].

Table 2.1.: Selection of possible microservice patterns

Category	Assigned patterns
Application architecture and Migration patterns	<ul style="list-style-type: none"> - Monolithic architecture - Microservice architecture - Decompose by business capability - Decompose by subdomain - Self-contained Service - Service per team
Data management	<ul style="list-style-type: none"> - Database per Service - Shared database
Deployment patterns	<ul style="list-style-type: none"> - Multiple service instances per host - Service instance per host - Service instance per virtual machine - Service instance per container - Serverless deployment
Observability patterns	<ul style="list-style-type: none"> - Log aggregation - Distributed tracing - Exception tracking
User Interface patterns	<ul style="list-style-type: none"> - Server-side page fragment composition - Client-side UI composition
Orchestration and Communication patterns	<ul style="list-style-type: none"> - Messaging - Domain-specific protocol - Remote Procedure Invocation - API Gateway - Backend for Frontends - Client-side discovery - Server-side discovery

Microservice security can be divided into external and internal security. As the name implies, external security handles all external security risks, including securing all points of entry. This security layer handles authorization and authentication operations for external requests and includes network security and endpoint security [26].

Internal security is liable for all data exchange between microservices. Data security is ensured by using communication patterns. These patterns take on authorization operations before sharing data with other microservices [26]. These patterns also include microservice security aspects. In the following, external and internal security patterns are explained in more detail.

2.4.1. External security patterns

When decomposing a monolithic application into microservices, the external attack surface increases. The external attack surface describes all access points of a system. It includes

2. Background

all paths into and out of the application and the code that protects these paths. The external attack surface also incorporates all valuable data, including sensitive business and personal data, and the code that protects this data [27]. Therefore, the external points of entry of each microservice need to be secured. Several external security options can be implemented to enhance the overall security posture of the system. Some of these options are explained in more detail in the following.

Service Mesh

A service mesh or a so-called "Sidecar" provides a dedicated infrastructure layer for managing microservice security within a microservice architecture. All external communication is handled by the sidecar proxy, as shown in Figure 2.2. Sidecars allow microservices to be implemented in a polyglot manner, while security functionalities are handled through external components. Service meshes can provide fast and reliable security validation. Additionally, they can handle security-related tasks such as service discovery, load balancing, traffic management, and encryption [28]. Service mesh simplifies security communication between services in both microservices and containers and improves the process of diagnosing communication errors. Since sidecars can be configured independently, users can change security configurations without changing the source code of the microservice business logic. This leads to an independent security architecture that keeps the system microservices as user-friendly as possible [6]. But it also has disadvantages such as increased runtime instances since each service call runs through the sidecar proxy first, which adds a step to the process [6].

Overall, service meshes are capable of assuming various roles in ensuring the security of communication between services. They can handle authentication, access control, and traffic encryption. Utilizing techniques such as mutual Transport Layer Security (mTLS), service meshes guarantee secure communication channels between services. This reduces the risks of unauthorized access and packet sniffing. Importantly, these security measures are implemented at the platform level, independent of the specific business applications involved [6, 29, 30].

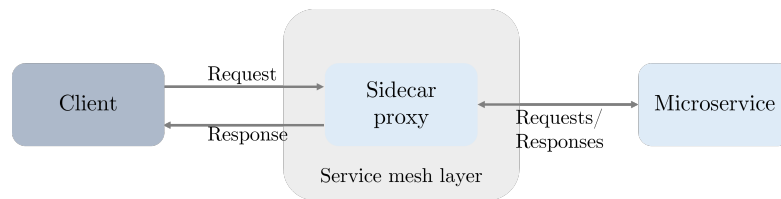


Figure 2.2.: Overview of usage of Service Mesh in accordance with [31]

API Gateway

An API gateway acts as a central entry point for all incoming and outgoing traffic between the client and the microservices (see Figure 2.3). Therefore, it provides a single entry point for all requests. By using a combination of techniques to secure microservices, it can

handle authentication, authorization, rate limiting, monitoring, and service discovery [32]. Adding another layer of security, API gateways can monitor traffic between the client and the microservices using HTTPS to prevent eavesdropping attacks, which occur when a hacker intercepts, deletes, or modifies transmitted data, and man-in-the-middle attacks, where attackers secretly transmit and possibly alter the communications between two parties [33]. They can also enforce authentication and authorization policies to ensure that only authorized clients can access the microservices. This is typically done using standard authentication protocols such as OAuth¹ or OpenID Connect². It also allows for a uniform approach to security across all microservices to protect a microservice-based application from unwanted third-party access. Besides security aspects, API gateways provide additional functionality such as caching, request routing, and load balancing to distribute requests across multiple instances of a microservice [33].

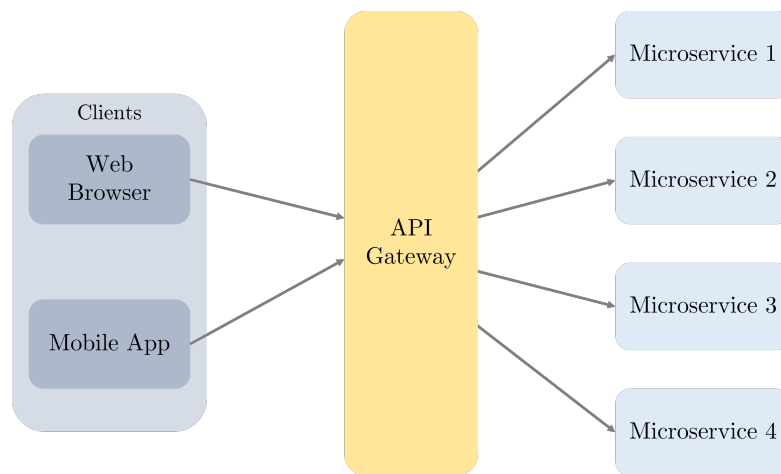


Figure 2.3.: Overview of usage of API gateway in accordance with [34]

Logging and Monitoring

Distributed tracing can help identify and diagnose security-related issues within a microservice architecture. It allows the tracking of requests across multiple services and can be used to identify malicious activity [6]. By logging all events that access information in the system, a continuous record can be created. Monitoring is as important as logging. By monitoring the system, anomalous behavior that may indicate an attack can be detected [6]. It is also important to monitor interdependencies between services to notice downtime of services and record the reasons [35]. This monitoring provides knowledge of exploitation, use of the system, weak points of the system, and a precise overview to study vulnerabilities [15].

¹<https://oauth.net/2/>

²<https://auth0.com/intro-to-iam/what-is-openid-connect-oidc>

2. Background

2.4.2. Internal security patterns

Besides the increased external attack surface, microservice decomposition leads to the fact that inter-service communication is also exposed to the outside. Consequently, service-to-service communication also needs to be secured. To achieve a secure system, internal security needs to be implemented. In the following, the most popular internal security options are explained.

Message Queue Pattern

Message queues play a crucial role in modern software architecture by enabling efficient communication between different services of a system. A message queue is a communication pattern that enables asynchronous and reliable data exchange between software components or distributed systems [36]. It functions as an intermediary or buffer, allowing producers to send messages to a queue and consumers to retrieve those messages. Message Queues assume authentication and authorization tasks before allowing the producer to publish a message and the consumer to access a message [36].

Message queues operate on the principle of decoupling the sender and receiver. Producers are responsible for generating messages and placing them in the queue, while consumers retrieve messages from the queue for processing when they are ready to process them (see Figure 2.4) [36, 37]. This decoupled architecture ensures that producers and consumers can operate independently, improving overall system security. In general, message queues can support fast, high-volume consumption rates since multiple consumers can be added for a topic but still only a single consumer processes each message. This loose coupling and reliable messaging between microservices increased inter-service security in a microservice architecture [36, 37, 38].

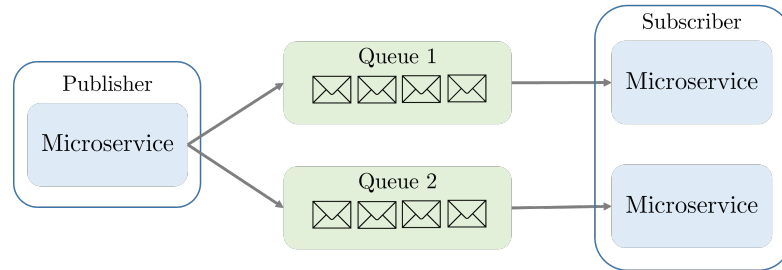


Figure 2.4.: Overview of Message Queue Pattern in accordance with [39]

Asynchronous communication between services allows for improved performance, responsiveness, and utilization of system resources, accepting losing the message order. Message Queues also enable message persistence by keeping messages until they are successfully processed, to avoid message loss. Ensuring reliable message delivery enhances fault tolerance and guarantees message integrity, even in the event of system failures [40, 37, 41]. By separating message generation from processing, message queues simplify horizontal scalability. Multiple consumers can process messages simultaneously,

increasing system throughput and overall performance. Besides scalability, load balancing is supported by allowing multiple consumers to subscribe to a single queue. This enables efficient distribution of workloads and prevents bottlenecks, ensuring optimal system performance [40, 37, 41].

Publisher/Subscriber Pattern

Besides the Message Queue pattern, the Publisher/Subscriber pattern is another useful option for the exchange of information between publishers and subscribers. It facilitates the delivery of messages to multiple consumers within a specific topic, while also preserving the order of message delivery as per their production sequence [38]. A typical publish-subscribe system includes multiple publishers and subscribers engaged in various topics, often spanning multiple queue managers. A microservice can act as both a publisher and a subscriber simultaneously. The publisher, responsible for providing information, delivers messages known as publications without requiring knowledge of the services interested in that information and encompass the specific topic they pertain to [38, 40].

Subscribers are the consuming part of the publications. They consume the provided data by creating subscriptions and specifying the topics they are interested in. Subscriptions determine which messages are forwarded to the respective subscribers. Each subscriber can create multiple subscriptions and receive information from various publishers [42]. All interactions between publishers and subscribers are managed by a queue manager. This queue manager receives messages from publishers and subscriptions from subscribers, which cover a range of topics [42]. Its role is to route published messages to subscribers who have expressed interest in the corresponding topics. This is shown in Figure 2.5.

The existence of topics enables the decoupling of information providers and consumers in publish/subscribe messaging. Unlike point-to-point messaging, there is no requirement to include a specific destination in each message. Publish-subscribe systems are particularly well-suited for stateful applications that rely on the sequential order of message reception. The order in which messages are received directly influences the application's state and the accuracy of its processing logic. Therefore, the preservation of message order is crucial for maintaining the correct functioning of stateful applications [38, 42].

The authorization process is conducted locally on the queue manager using local identities and permissions. This authorization is not dependent on the topic's definition or its location. Therefore, in scenarios involving clustered topics, it is necessary to perform topic authorization on each queue manager within the cluster. This ensures consistent authorization checks across all queue managers in the cluster. The queue manager is responsible for secure communication and assumes the task of authenticating and authorizing each publication and subscription. This increases the security of the microservice because it ensures that each inter-service communication is validated and secured [43].

2. Background

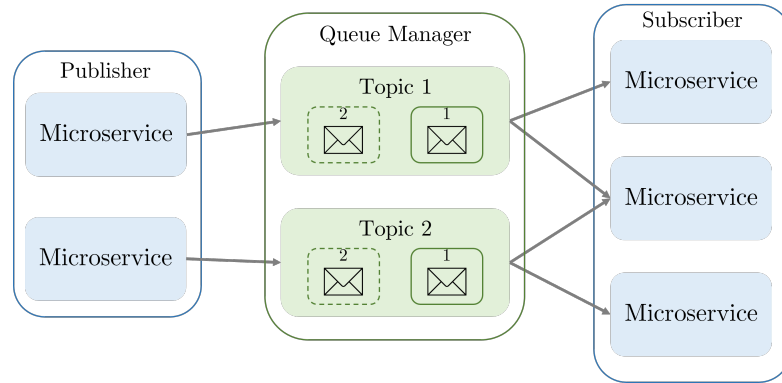


Figure 2.5.: Overview of Publisher/Subscriber Pattern in accordance with [39]

2.5. Microservice security

Underlying every security implementation, Identity and Access Management (IAM) solutions are used to validate each external microservice request. Each of the above-mentioned external security options uses an implemented IAM solution to be able to fulfill their assignment and secure microservice entry points. In the following, IAM is explained in more detail.

2.5.1. Identity and Access Management

IAM solutions can be used to manage user identities and access to microservices. This can include features such as authentication, authorization, and role-based access control [6]. During the authentication, the validity of a claimed identity is established. The system that requests to access a secured component, session, or transaction has to prove its identity by presenting credentials. After authentication, the authorization is checked by determining whether a validated entity is allowed to access a secured resource [44]. This is established based on attributes, predicates, or context. Attributes are name-value pairs describing details about the entity. Based on the environment of the secured asset, predicates are conditions that must be true to gain access to the resource. Context puts the requested transaction into a frame of reference based on time, a history of actions, or the position of a rule within a rule base [44].

Open Authentication protocol

The most popular IAM solution in a microservice architecture is the open standard protocol, Open Authentication (oAuth) which is a protocol for granting access to resources for a user or application [45]. It is commonly used to secure microservices by providing a way for clients to authenticate and authorize their access to protected resources. oAuth enables users to grant third-party access to their resources without sharing their credentials, which enhances security and privacy [45]. It secures server-to-server communication between

API clients and API servers. The OAuth framework reduces the effort and responsibility of developers by eliminating the need and therefore the susceptibility to errors to build their authentication mechanism in each microservice [46]. The OAuth protocol distinguishes on a set of roles and flows that describes how clients can obtain access tokens to access protected resources. The roles in the OAuth protocol include according to [47]:

- Resource Owner: The user who owns the protected resource and grants access to it.
- Client: The application or service that requests access to the protected resource.
- Resource Server: The server that hosts the protected resource.
- Authorization Server: The server that issues access tokens to the client after the user grants authorization.

To define how access tokens are obtained, the OAuth protocol supports different types for granting access. Authorization Code Grant is used by web applications. If the user agrees to the access request, an authorization code is granted to the client application, which communicates with the OAuth service to get an access token. Another grant type is the Proof Key for Code Exchange (PKCE), which is a security-centered OAuth grant type and is conceptualized as proof possession [48]. Before getting an access token, the client applications need to prove that the authorization code is authentic. This grant type has been developed because the older version, Implicit Grant Type, was not secure enough. It was used by JavaScript applications to obtain an access token directly from the authorization server without a server-side component. No authorization code is involved to secure receiving the access token [48]. Client Credentials Grant is used by clients to obtain an access token only using their credentials. By requesting access to the protected resource under its direct control, the client credentials grant is needed [48]. OAuth provides a full validation process of the requesting party before access is granted. This is the standard protocol for security, which is implemented in the external and internal security options. The usage of suitable microservice security is key to a successful microservice decomposition [48].

3. State of the Art

In this chapter, I explore the existing practices related to decomposing a monolith system into microservices, focusing on the goal of identifying decomposition tools supporting microservice security.

3.1. Objective

The goal of the literature review is to identify the existing strategies and approaches for microservice decomposition and arrange the findings into a possible set of groups by common approaches. I also want to know how security aspects are considered in microservice decomposition and when they are included. Therefore, for the conducted literature review, the following questions were defined:

***RQA:** Which strategies are pursued in decomposing monoliths to microservices?*

***RQA.1:** Which techniques and tools are reported in academic literature for microservice decomposition?*

***RQA.2:** How is microservice decomposition performed by industry experts?*

***RQB:** How can security aspects be addressed in decomposing microservices?*

3.2. Decomposition approaches

Answering RQA, I conclude from the gray and academic literature review that microservice decomposition is divided into two strategies (see Figure 3.1): Tool-based and Model-driven approach. For tool-based approaches, three general steps can be identified. First, the source code is analyzed. Depending on the approach, different tools are used in this process. In the next step, a first draft of the microservice is created. For this purpose, a dependency graph is usually created or boundaries are defined. In the last step, a possible microservice is developed.

Model-driven approaches are oriented on the design level and focus more on the architecture of the microservice. Three general steps can also be identified. First, possible services are manually identified from the architecture of the monolith. These are then modeled in bounded contexts using domain-driven design. Finally, patterns are applied to build the microservice. The individual studies are discussed in more detail in chapters 3.3.1 and 3.3.2.

3. State of the Art

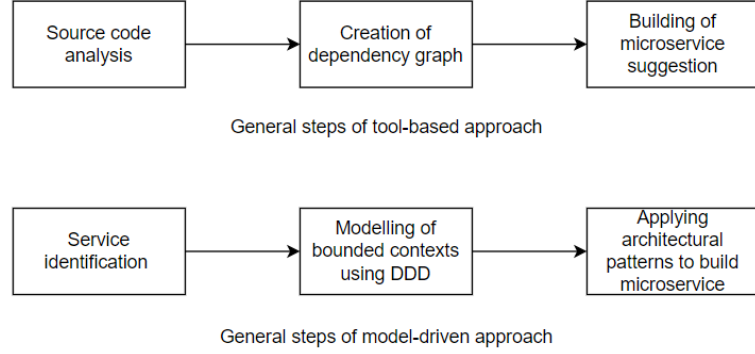


Figure 3.1.: Overview of the general steps of microservice decomposition approaches

3.3. RQA.1: Microservice decomposition approaches in academic literature

3.3.1. Tool-based approaches of microservice decomposition

Mazlami et al. [4] suggest formal coupling strategies and a clustering algorithm to assist with migration. The strategy consists of three steps: the monolith stage, the graph stage, and the microservices stage. A transformation step is performed between each of the individual steps. The first transformation step - the construction step - converts the monolith into the graph representation based on the source code or repository, while the clustering step decomposes this representation of the monolithic architecture into microservice candidates. The authors propose generating a graph and decomposing it into subgraphs that represent suggestions for microservices. It combines a Kruskal-based algorithm with the use of strategies for logical coupling and collaborators [4].

The authors Santos and Paula [8] present a tool that enables users to weigh each dimension to be included and decompose Java-based projects in microservices. Therefore, a combination of the *Monobreak algorithm* developed by Pereira da Rocha [49] and the logical coupling strategy by Mazlami et al. [4] is used. An automatic code analysis, service decomposition, metric calculation, and report generation are performed by the tool. Therefore, static analysis of source code from a git repository of a monolith is performed by the tool and decomposed according to the assigned weights. To identify classes and methods, Java Call Graphs Utilities and JGit Libraries are used. In the next step, application services are identified and grouped into microservices. In the end, microservice suggestions are generated. To evaluate the quality of the decomposition, the silhouette coefficient creates clusters for microservices by comparing the similarity of an object [8].

To use the decomposition tool, the authors Krause et al. [50] start with a domain analysis. Therefore, a familiarization, modeling, and partitioning phase is performed.

3.3. RQA.1: Microservice decomposition approaches in academic literature

As a result, a decomposition at a conceptual domain level is done and a static software structure analysis tool to map source code packages to the previously identified target boundaries can be applied. The static software structure analysis tool *Structure101* is used to analyze Source Code Package (SCP).

As a dynamic analysis tool, the authors use the trace visualization tool *ExplorViz* to enable live monitoring and visualization of large software landscapes [50]. With the help of *ExplorViz*, the bounded contexts can be revised. Besides the analysis of the source code, the data model is also statically analyzed by the database administration tool *DBeaver*. This tool helps to identify the database tables which are used by the use cases. The combination of these tools is a valuable step toward microservice decomposition [50].

Gysel et al. [9] discuss a knowledge management method and supporting tool framework for microservice decomposition. The *Service Cutter* calculates a cluster of nano entities to form microservices. A nano entity consists of data, operations, and artifacts. It extracts coupling information from software artifacts of the software system like use cases and creates an undirected, weighted graph to identify clusters. The tool framework requires a set of specification documents, so-called System Specification Artifacts, from which a set of weighted coupling criteria as input can be extracted. It results in a graph that displays the microservices under consideration as nodes and uses weighted arcs to indicate how cohesive and/or coupled two candidates are. The authors stress the intention of assisting the decision-making process rather than fully automating it. While the framework proposed in [9] was for generic service applications, it can also be a useful support when migrating from legacy monolithic systems to microservices.

Pigazzini et al. [10] introduce an approach to identify candidate microservices in Java Projects by using a tool named *Arcan*. The system architecture, architectural smell detection, and topic detection are performed by static analysis. The tool relies on graph database technology. All the computations are based on the dependency graph. The nodes in this graph represent the system entities, such as Java classes, packages, and methods.

The migration process consists of three steps. The first step is architectural smell detection. In this step, architectural smells that could hinder the identification of services are detected before the migration process is started. As a next step, a dependency graph analysis is performed to visualize a possible decomposition [10]. The connected component detection method is performed to identify connected components in the graph by looking at the undirected edges. The vertical functionality view aims to view each functionality that is contained in the project to assist in the extraction of the parts of code that we are interested in as microservice candidates. The logical layer view separates and assigns each class to its layer. The last step is topic detection, Therefore a document collection is created which is then processed by a Latent Dirichlet Allocation (LDA) algorithm to detect suitable topics [10].

For supporting the identification of microservices in legacy code, Carvalho et al. [51] proposes a many-criteria search-based approach, the so-called *toMicroservices*. The tool

3. State of the Art

identifies microservices by looking at the methods. The tool requires three pieces of information as input: first, the legacy source code, including code elements indicators that will not be parsed, is needed. Secondly a list of features that are related to each execution of the legacy system and lastly the number of microservices to be identified [51]. *toMicroservices* uses a graph-based representation, where the vertices represent the microservice candidates and the edges the communications between them. *toMicroservices* generates a set of candidate solutions as output, where each element is represented as a graph of microservice candidates and the source code in the legacy system is associated with each candidate.

3.3.2. Model-driven approaches of microservice decomposition

Li et al. [52] propose a microservice decomposition approach to provide a complete architecture migration process. Therefore, the *Strangler Fig Pattern* introduced by Fowler is used. To get started, a Domain-driven design is applied to identify microservices. Methods like event storming, domain storytelling, or user story mapping are used to identify requirements. By performing these methods' entity classes and their relationships can be found and abstract domain models for the system can be built. To apply the *Strangler Fig Pattern*, the extraction of services that bring the most benefits must be prioritized. After prioritizing the services, service implementation, integration, and testing can be performed [52]. The new microservice will be implemented around the old monolith. By using the Strangler Fig pattern reconstruction of the system, the monolith simultaneously operates with the strangler application without affecting the client-side users. After completing the development and testing of the new service, the code corresponding to the newly implemented functions can be removed. This migration process can be performed continuously and incrementally until all identified services are decomposed into microservices.

Larrucea et al. [53] propose five guidelines for migrating microservices. To get started, the organization must be prepared for changes. Furthermore, the system to decompose needs to be studied. Identification of the dependencies is performed, and an architecture is defined. These two steps can be supported by the tool but should be performed on an architectural level. After the system is analyzed, a prioritization of the components to migrate is performed. Following the prioritizing of the services, the design, coding, testing, and integration of the specific component can be performed. The authors point out, that security mechanisms need to be considered and implemented according to appropriate encryption techniques [53].

Taibi et al. [24] concentrate on architectural patterns for microservices. Three categories for the emerging architecture patterns are identified: Orchestration and Coordination-oriented architecture patterns, patterns that reflect on physical deployment strategies, and ones that reflect on data management. In the first category, several patterns are identified: The API gateway pattern, the Service Discovery pattern, the Client-side Discovery pattern, the Server-side Discovery pattern, and the Hybrid pattern. All these

3.4. *Microservice decomposition approaches in gray literature*

patterns are used to capture communication and coordinate from a logical perspective [24]. The Multiple Service per Host pattern and the Single Service per Host pattern have emerged in the context of deployment strategies. To perform microservice decomposition on the data storage the Database-per-Service pattern, the Database Cluster pattern, and the Shared Database Server pattern can be performed. These patterns support microservice decomposition from an architectural point of view.

To perform a successful microservice decomposition, Habibullah et al. [54] propose a deep analysis of the system. First, understanding the code and defining the boundaries of the system is important. For this purpose, a class diagram from the existing system is modeled to illustrate the relationships between system classes. This representation of the system helps to determine possible services. Applying domain-driven design supports the deconstruction into smaller functional components and the independent step of the problem which comprises the bounded context is described. In the next step, the concept of data storage needs to be considered to extract tables and place each table in an independent database. These are later used by one of the identified microservices. To perform microservice decomposition, security aspects need to be considered. The authors draw attention to the rising importance of authorization and authentication mechanisms [54].

3.4. **Microservice decomposition approaches in gray literature**

Also, in practice, the general strategies elaborated in Figure 3.1 are followed. Industry experts advise studying the structure of the monolith in detail. To begin implementation, an order of implementation must be established. For this, the most important services must be identified [55, 56, 57]. The handling of data storage is also very important. Before decomposition, it is necessary to think carefully about how the microservice will be built and what data is needed. It must be ensured that each microservice owns and controls access to its data. It must also be defined which API contracts will be used to access the data [56].

Once the structure and data protection have been defined, the core services can be migrated. In the beginning, the most important services should be migrated. Therefore, the migration of access management is often the first step. Authentication and authorization functionalities are implemented for this purpose. Authentication is a complex service. Since most services require this functionality, there is a lot of common logic. For this reason, some of these basic pieces must be extracted so that the primary functions can continue to run without being tied to the monolith [55, 56].

Since microservices are developed and deployed independently, each team needs to include security in the process. To continue the development process, a committee of security experts assesses if the development team can move forward. However, in practice, development teams release new features without an assessment of security experts. This

3. State of the Art

leads to unsecured microservices and increases the attack surface [58].

Besides the problem that development teams often do not feel responsible for security, DevOps tools do not provide proper security. Therefore, developers need to be aware of security issues and regularly challenge themselves to guarantee a stable foundation for implementing proper security [59, 60].

Most gray literature discusses authentication and authorization solutions for developers. This demonstrates that the importance of authentication and authorization is well-known in practice, and best practices are presented [61, 62, 63].

To implement the security aspects, it is useful to use tools such as Spring Security OAuth 2.0¹. These tools can facilitate and support moving access management components. Also, in practice, it is advised to consult security experts for security aspects [58].

3.5. RQB: Addressing security aspects in microservice decomposition

Since access management is one of the most important security mechanisms, its implementation in the decomposition into a microservice is very important. Authorization can be done via flow-oriented authorization frameworks or through standards and frameworks like OAuth 2.0 [6]. However, this topic is not discussed in detail in the academic literature. No precise approach to migrating access management components is presented [53, 24, 54, 64]. However, neither precise step-by-step instructions are given nor tool-based support is provided.

During the gray literature review, it became clear that security topics are known, but not explicitly implemented. This can also be seen in Table 3.1. To emphasize that security is not the primary concern of the tools, keyword matching has been performed. Therefore, keywords of the area of IAM are selected to determine to which extent IAM-relevant security aspects are taken into account.

Tools for microservice decomposition only focus on the identification of microservices in the monolithic system but do not pay attention to including necessary security mechanisms during the decomposition. With such tools, new security mechanisms are only named as an outlook and are not included in the new microservice architecture, although it is known that these are needed. Implementing suitable microservice security is usually postponed to another time when experts are consulted for security aspects. This shows the urgency that the application of security mechanisms must already be taken into account when designing the microservices.

¹<https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html>

3.6. Derivation of Research Questions from the Literature Analysis

Table 3.1.: Overview of the number of certain keywords in the examined papers

Approach	IAM keyword mention						Context usage
	Security	Authentication	Authorization	Secure communication	Unencrypted sensitive data	Input/output control	
[50]	1	0	0	0	0	0	IAM as outlook
[8]	0	1	0	0	0	0	IAM not as relevant concept
[4]	0	0	0	1	1	0	-
[9]	10	0	0	0	0	0	IAM only as general concept
[10]	0	1	0	0	0	0	IAM not as relevant concept
[51]	0	7	0	0	0	0	IAM not as relevant concept
[52]	0	0	2	0	0	0	IAM not as relevant concept
[53]	8	0	3	0	0	0	IAM only as general concept
[24]	2	0	1	0	0	0	IAM not as relevant concept
[54]	6	4	2	0	0	0	IAM only as general concept

3.6. Derivation of Research Questions from the Literature Analysis

To get a perspective on the meaning of support in microservice decomposition in the context of security aspects literature research is conducted, which reveals that (i) in most studies security is no central topic during decomposition, (ii) there are no step-by-step approaches which explicitly contain security aspects during decomposition, and (iii) there is no security-based approach that helps to recognize security mechanisms in a microservice architecture. These results indicate that systematic consideration of security aspects is an unmet need within microservice decomposition. Therefore, the following research questions are derived:

RQ1: *How can security mechanisms be included in microservices?*

RQ2: *How can security tactics be continuously validated in microservices?*

This thesis aims to address the lack of support concerning security aspects by an architectural approach, which is based on a component model-generating tool that assists the developer with decomposing a monolith into a microservice structure. The underlying motivation is that this tool increases the efficiency of decision-making by ensuring a secure

3. *State of the Art*

modeling process.

Within this scope, an interactive bot, the so-called DevBot, is developed, that supports the creation of a component model from the beginning. This bot uses Codeable Models² to formally represent the architectural structure. The bot supports security mechanisms based on a design decision for securing the microservice architecture. Such assisted development of the microservice architecture ensures that all security aspects are included. Furthermore, all design decisions are logged as Y-Statements leading to a development process that is end-to-end traceable.

This semi-automated approach will be evaluated by comparing a created model with predefined security standards. Based on the security level of a component model created with the interactive bot, it is determined if a successful microservice decomposition is achieved. Therefore, the DevBot application is developed to support the user by creating a microservice system. The focus is on providing a security-oriented modeling tool, which ensures that all needed security patterns are included in the modeling process.

²<https://github.com/uzdun/CodeableModels>

4. Methods

To answer the established research questions, the thesis follows several steps to develop the DevBot system (see Figure 4.1). To start the development process, a multivocal literature review is conducted to establish the current State of the Art and identify possible research gaps. After conducting the literature review, I determined a security decision tree to include all possible security risks in the DevBot system. The development of the DevBot system is achieved iterative according to the Design Science Research approach [65]. Since the GUI is a crucial part of the DevBot system, I especially focus on the implementation of the GUI. During the development of the DevBot system, the GUI is developed iteratively in consultation with other developers. As the last step of development, a suitable conformance check and validation of the DevBot system is established.

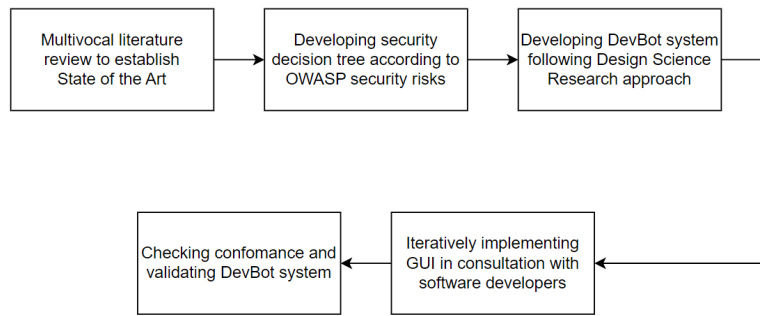


Figure 4.1.: Overview of the overall approach of this thesis

In the following, the applied development methods are discussed in more detail.

4.1. Design Science Research

In the previous literature review, it appeared that security measures are neglected during microservice decomposition. This finding indicates that a solution for dealing with microservice security needs to be found. Following the Design Science Research Framework by Hevner et al. [65], an artifact as a possible solution approach, is developed and iteratively refined until the artifact takes into account both the acquired application knowledge and the required business needs.

The Design Science Research Framework consists of the *Relevance Cycle* and the *Rigor Cycle* to develop the artifact by considering the environment of the artifact and

4. Methods

establishing a knowledge base. In the Relevance Cycle, the environment of the artifact is defined and the involved actors, organizational and technological structures, and problems or opportunities are elaborated. For this purpose, the application context, requirements, and success criteria for the artifact are tracked. The Rigor Cycle uses the generated knowledge from the literature review, which forms the knowledge base. This knowledge base is expanded in each iteration with the new findings, and the artifact is adapted accordingly. The learned knowledge is abstracted and the knowledge is collected to influence the design process [65, 66].

As a possible solution to the problem of the lack of microservice security, the so-called DevBot is being developed as the artifact. For this purpose, the DevBot is created to compensate for the lack of support concerning security aspects by an architectural approach. Based on a component model generating tool, the idea is to assist the developer with implementing security measures during microservice decomposition.

In the following, the Design Science Research guidelines and the cycles are explained in detail.

4.1.1. Design Science Research guidelines

To develop the DevBot successfully, the following seven guidelines defined by Hevner et al. [65] need to be considered:

Guideline 1: Design as an Artifact

To answer the research questions, the DevBot system represents a viable artifact in the form of a prototype to solve the lack of applied microservice security. It provides a rigorous level of security measures by applying security patterns and following security guidelines. The DevBot system is an application to support the user in microservice security-related decisions on an architectural level.

Guideline 2: Problem Relevance

The DevBot system, a technology-based solution that tries to provide a security-oriented tool, is developed to solve the elaborated research gap in tool-supported applications for microservice security. The number of papers in research on microservice security emphasizes its relevance. Especially IAM is ranked highly in microservice security since protecting sensitive data from external clients is important.

Guideline 3: Design Evaluation

The design artifact is validated based on predefined criteria to evaluate the functionality, utility, reliability, and efficacy of the DevBot. Besides the validation, a conformance check is performed to check the quality and accuracy of the DevBot system. The combination of both processes secures a whole evaluation of the artifact, which can be found in chapter 6. It is checked if all requirements are satisfied by following a descriptive evaluation approach.

Based on the established knowledge base, the utility of the DevBot is defined, and scenarios to demonstrate it are constructed. These scenarios are validated via annotated models by Zdun et al. [67]. The validation process checks whether the DevBot can detect and verify the security of existing microservices, and then adds any necessary security measures. It also ensures that the microservice model is correct.

Guideline 4: Research Contributions

Since microservice security is often neglected and there is no tool-based approach to focus on security aspects, the security-oriented design artifact is the main contribution of design-science research. The development of the DevBot is an approach to consider microservice security from the beginning on an architectural level and provides a security supporting tool for microservice decomposition. It enables a simple and intuitive usage for developers without modeling knowledge and provides a quick generation of component models. Furthermore, the DevBot can be easily extended with new security patterns, which allows it to be easily adapted to organizational structures.

Guideline 5: Research Rigor

Since the DevBot is built upon the knowledge generated in the literature review, the effective use of this knowledge is emphasized, and research rigor is derived from this usage. It is based on past research in this area. During the construction of the DevBot, microservice security guidelines and patterns are followed to ensure the usage of appropriate techniques.

Guideline 6: Design as a Search Process

Using codeable models¹ as a basis for generating microservice models build the means to construct a possible solution for the lack of microservice security. The DevBot is developed upon this and extends the functionalities to achieve a good solution. A security-oriented tool is developed that supports the developer on an architectural level with microservice decomposition. After identifying external and internal security tactics, the focus is on the implementation of these security mechanisms. During the development process of the DevBot, the requirements for the DevBot are adapted according to new knowledge to achieve the goals.

Guideline 7: Communication of Research

This thesis presents effectively to both technical and managerial audiences. Since the most important terms and security tactics are explained, the audience does not need to have a security background. The presentation of the development process is primarily technical but also motivates a managerial audience with a detailed introduction to security tactics

¹<https://github.com/uzdun/CodeableModels>

4. Methods

and their benefits. For future work, I plan to present this thesis to industry audiences at special interest group workshops.

4.1.2. Relevance Cycle

The gray literature of the literature review indicates that there is a need to develop further security-oriented tools for practical usage, which forms the application context. During the Relevance Cycle, decisions regarding the environment are made.

Generally, microservice architectures can be found in different business and organizational structures. However, since microservice security is independent of the organization structure, the DevBot does not have to consider specific business logic. On a technical level, the DevBot uses codeable models² as a basis for creating the microservice model and for validation of the previous annotated model by Zdun et al. [67] are used.

To get started with the development of the DevBot, first, I decided on which basis the DevBot would act. Since the literature research shows that most tools migrate microservices based on the source code, the DevBot starts at an architectural level to give the developer an overview of the necessary security components right from the start. Therefore, the DevBot is developed based on a security decision tree, where security risks identified by OWASP [27] are taken into account. In addition, end-to-end traceability has proven to be important as development teams change and continuous logging of the most important design decisions facilitates familiarization and traceability.

After the basic requirements for the DevBot are determined, the next step is to define the microservice decomposition approach. After considering using existing decomposition tools like *Service Cutter* [9] or microservice extraction based on clustering algorithms [4] to generate microservice options from monoliths, the DevBot is implemented to build microservice architectures from scratch or modify existing models created with codeable models. Thereby, microservice security is ensured, since the DevBot constantly evaluates the security level of the current microservice model.

The GUI of the DevBot is an important factor since intuitive and easy usage is essential. Therefore, the GUI has been developed in close cooperation with other developers. The exact iterations are available in section 5.4.

Validation is an essential step in the development process. The first idea of validation of the DevBot is an empirical evaluation. Therefore, the performance of an empirical test with additional software architects and security experts is considered. After unsuccessfully contacting suitable developers, it becomes apparent that an empirical evaluation is too costly and time-consuming for this frame.

Therefore, the DevBot application is tested with existing microservice systems. Preferably, already analyzed microservice models are used for this purpose. To define a ground truth, microservices of different sizes and complexity are searched. Since the focus is rarely on the area of microservice security, only a few datasets are found [67, 68, 69]. The

²<https://github.com/uzdun/CodeableModels/tree/master>

datasets are evaluated and the annotated microservice models by Zdun et al. [67] are selected since these are analyzed in collaboration with software and security experts and are already defined with codeable models. Therefore, these models are optimal for the validation of the DevBot system.

From the annotated models, I select five microservices of different sizes to test as many contingencies as possible. Since the DevBot system should facilitate the implementation of security aspects on an architectural level, the reduction of the steps to be applied and the time of the user are important criteria. Decreasing the costs and effort in developing microservice architectures are also crucial aspects for validating the success of the DevBot system. In addition, the support for security topics should be increased. I compare the coverage of the current security tactics as well as all other criteria with a manual evaluation of the implemented security of each microservice.

The detailed evaluation of the DevBot system can be found in chapter 6.

4.1.3. Rigor Cycle

The knowledge base provides the foundations for the development of the DevBot. In order to start with the development, a literature review is conducted to get an overview of the current approaches in microservice decomposition and the used security tactics. Based on this review, it becomes apparent that microservice security is often neglected. The first requirements for the DevBot are defined based on these results:

- Focusing on a security-oriented tool approach.
- Providing security step by step with no previous security knowledge needed.
- Implementing security on an architectural level to build a basis for further development.

The DevBot is developed according to a security decision tree (see section 4.2.2). The decision tree considers API security risks and is based on OWASP [27]. To define a framework for additional research, recent research papers dealing with microservice security have been examined in the next iteration to extend the knowledge base.

According to the generated knowledge of the literature review, two approaches are followed in the process of microservice decomposition: a tool-driven approach or a model-driven approach. Both approaches do not consider including microservice security tactics. The DevBot follows a security-oriented tool approach and focuses only on microservice security. On an architectural level, the DevBot creates a secure microservice architecture and supports the developer from the beginning.

In further iterations, existing microservice security approaches are examined. Hannousse and Yahiouche [11] support the necessity of further research in the area of microservice security. Although a systematic mapping study shows that most studies address external microservice threats (63%) and 13% address internal threats, only a few studies consider the development of tools that address microservice security. Furthermore, Berardi et al. [12] also discover that there are no well-established techniques that assist developers in

4. Methods

migrating monoliths to microservices, paying particular attention to potential security risks. Security-by-design should be adapted at all stages during the microservice decomposition, but there are no established references or guidelines for practical adaptation.

Some possible tools for identifying microservice security risks include the approach of Chondamrongkul et al. [70] which combines ontology reasoning and model checking technique to automatically identify security threats and offer an insightful result to support the analysis of possible impacts, and an openly available prototype framework by Yarygina and Bagge [71] to secure microservice communication with mTLS, self-hosted Public Key Infrastructure (PKI) and security tokens.

DevBot system is focusing on an architectural approach to include microservice security from the beginning. Since Hannousse and Yahiouche [11] establish that mostly external security is considered, the DevBot system focuses on external and also on internal security. Including inter-service communication achieves a higher overall security level for the microservice architecture.

The Rigor Cycle directly incorporates this knowledge in the development process, and the DevBot system includes internal security mechanisms. This continuously improves the framework and requirements for the DevBot. The requirements and the development process are described in detail in chapter 5.

4.2. Application design

After developing the software development methodology, it is required to identify what security decisions need to be taken into account to implement the identified security functionalities. By developing a method based on the most common security risks in microservices, a security decision tree is developed. Derived from this security decision tree, the DevBot is developed to create a secure component.

4.2.1. Security risks

To create a secure microservice architecture, the main security risks need to be considered. Because of the new architecture, microservices need to be secured due to their distributed nature, sensitive data handling, service isolation, scalability, and integration with third-party services. These characteristics all require a secure API. Open Web Application Security Project (OWASP) [72] identifies the following ten API security risks:

1. *Broken Object Level Authorization*: No proper access controls on objects or resources
2. *Broken Authentication*: Vulnerabilities that allow attackers to compromise user credentials, session tokens, or authentication mechanisms
3. *Broken Object Property Level Authorization*: Combining the risks of Excessive Data Exposure and Mass Assignment which contain improper authorization validation at the object property level and exposure or manipulation of information by unauthorized parties

4.2. Application design

4. *Unrestricted Resource Consumption*: Satisfying API requests are made available by service providers via API integration
5. *Broken Function Level Authorization*: No appropriate authorization checks for different functions or operations
6. *Unrestricted Access to Sensitive Business Flows*: Vulnerability to expose a business flow without compensating for how the functionality could harm the business
7. *Server-Side Request Forgery (SSRF)*: Fetching a remote resource without validating the user-supplied Uniform Resource Identifier (URI)
8. *Security Misconfiguration*: Improperly configured API security settings, such as default or weak configurations
9. *Improper Inventory Management*: No proper and updated documentation of hosts and deployed API versions such as deprecated API versions and exposed debug endpoints.
10. *Unsafe Consumption of APIs*: Adopting weaker security standards to third-party services

4.2.2. Decision tree

According to these risks, a decision tree for microservice security is developed. Figure 4.2 provides an overview of the whole process, while Figures 4.3 - 4.6 depict the detailed submodels.

4. Methods

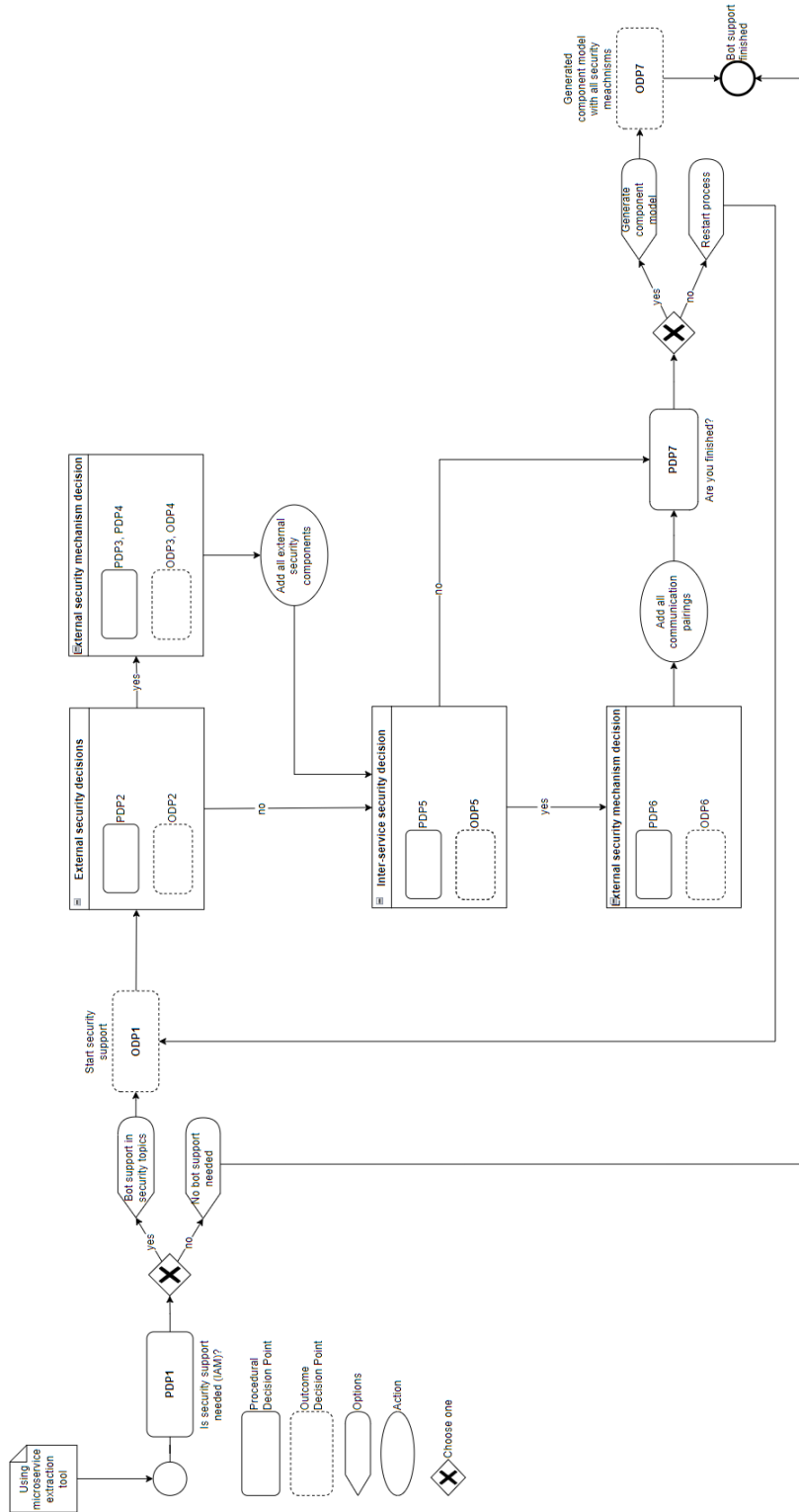


Figure 4.2.: Decision model of the security helper

By considering and addressing these OWASP API risks, organizations can establish a strong security foundation for their microservice architecture. Implementing secure authentication, authorization, input validation, and monitoring measures at the API level helps to protect microservices and the sensitive data they handle [72]. This is where the DevBot comes in and offers the user support for security right from the start, as depicted in Figure 4.2. The first Procedural Decision Point (PDP) is whether general help is needed to include security mechanisms when creating the component diagram for the microservice system. Depending on whether the user wants to accept help, either the process is started in the next step or the user can exit the application. Hence, the next PDP is to decide whether external security is needed to protect the microservices from external clients (see Figure 4.3).

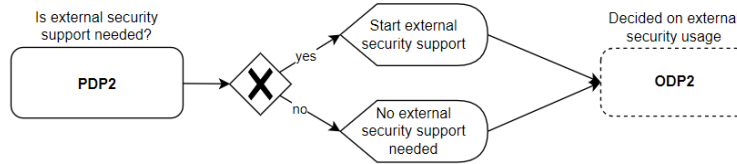


Figure 4.3.: Submodel 1: Decision model for the general need of external security

Based on the requirements for secure microservices, Broken Object Level Authorization, and Broken Function Level Authorization are particularly relevant to microservices architecture due to the decentralized nature of the services. Furthermore, microservices may expose internal object references or identifiers through APIs. Failing to properly secure exposed entry points can enable attackers to access unauthorized data or perform unauthorized actions. Implementing proper access controls like authorization checks at the object and function levels protection of sensitive data within each microservice, and secure object reference handling mitigates this risk [72]. The usage of proper external security mechanisms decreases these risks. To avoid these possible risks, the following security decisions are derived. First, the user can decide which external clients or external endpoints are used before selecting an external security pattern. PDP4 uses the information if sensitive data is handled to establish a suitable security strategy.

Excessive Data Exposure can lead to privacy breaches and compromise the security of the system. Applying data minimization techniques and enforcing strict data exposure policies at the API level helps mitigate this risk [72]. Hence, the DevBot recommends the best security pattern based on the size of the system and the information about the involved data, as shown in Figure 4.4. This can either be an API gateway for smaller systems with sensitive data, a sidecar architecture for bigger systems with sensitive data, or no recommendation if the data is not sensitive. The user can select an option, and the Outcome Decision Point (ODP) 5 logs the recommended security pattern and the selected option.

4. Methods

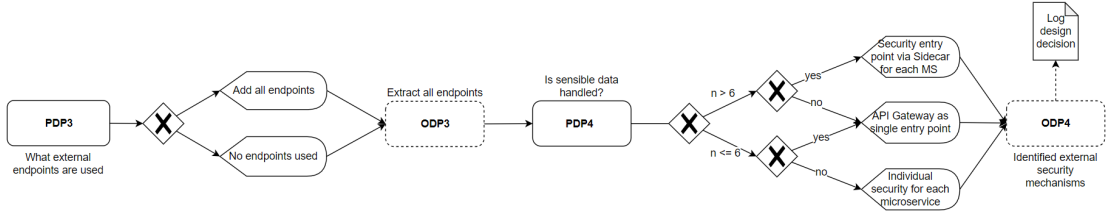


Figure 4.4.: Submodel 2: Decision model for specific external security options

Besides communicating with external clients, microservice architecture involves multiple services developed and maintained by different teams. Inconsistencies in security configurations can lead to security misconfigurations and vulnerabilities. Establishing secure configuration guidelines and regular audits help address this risk [72]. Therefore, an internal inter-service security strategy needs to be implemented. The user can decide in PDP5 if inter-service communication security is needed or not (see Figure 4.5).

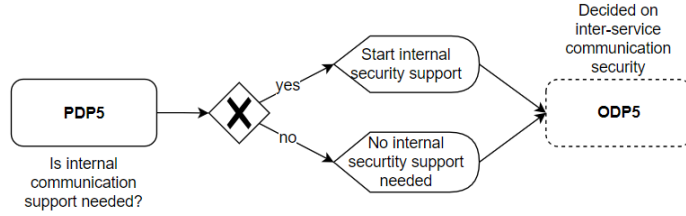


Figure 4.5.: Submodel 3: Decision model for the general need of internal security

If microservices need to interact with each other, the user can choose between the Message Queue pattern or Publisher/Subscriber pattern, as shown in Figure 4.6. In ODP6, the design decision is logged, and all decisions can be seen in the component diagram. Therefore, all involved teams can use the same inter-service security strategy and the risk of security misconfigurations decreases. The overall security of inter-service communication is maintained, and detailed guidelines can be established based on the selected security pattern.

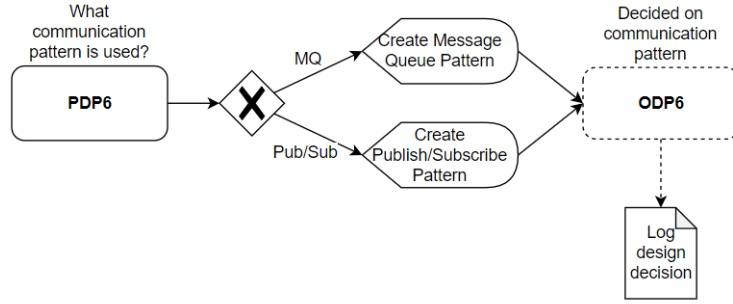


Figure 4.6.: Submodel 4: Decision model for specific internal security options

To finish the component model, the last step to ensure a safe microservice structure is to generate the model where all security components are visualized. The user can restart or finish the decision process by saving the model and exiting the application.

To summarize, the structure of the DevBot was developed based on the identified risks of the APIs by OWASP [27]. The decisions are defined to minimize the security risks and support creating a secure and sound microservice component diagram. The main goal of following the steps of the decision tree is to help the user include security from the beginning of the microservice architecture development.

The security decision tree only focuses on a few security options. To keep the DevBot simple only an excerpt of security mechanisms is implemented during the development process. For external and internal security the most popular patterns are implemented to show the usage and benefits of the DevBot system. The DevBot can be extended for future work to include further security patterns.

5. Implementation

5.1. System design

The DevBot is split up into two main parts, DevBot GUI and DevBot Backend. The majority of the system is contained within the DevBot Backend, with which the DevBot GUI interacts to pass information and perform the tasks requested by the user, who can either be a software developer or a software architect.

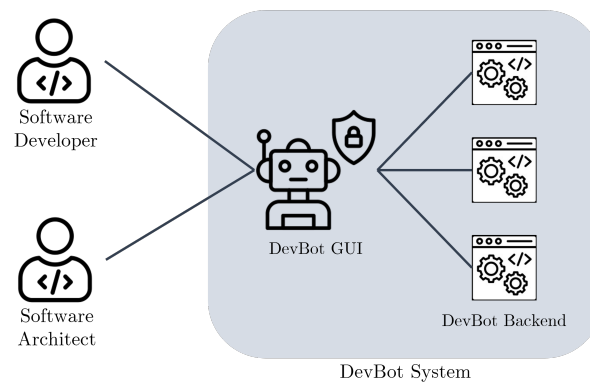


Figure 5.1.: Overview of the system design

Before giving a detailed description of each part of the DevBot system, an overview of the whole system is given (see Figure 5.2). The system consists of a GUI, which is represented in the presentation layer. The implemented application functionalities result in the application layer, and the used codeable models and the corresponding metamodels represent the domain layer.

5. Implementation

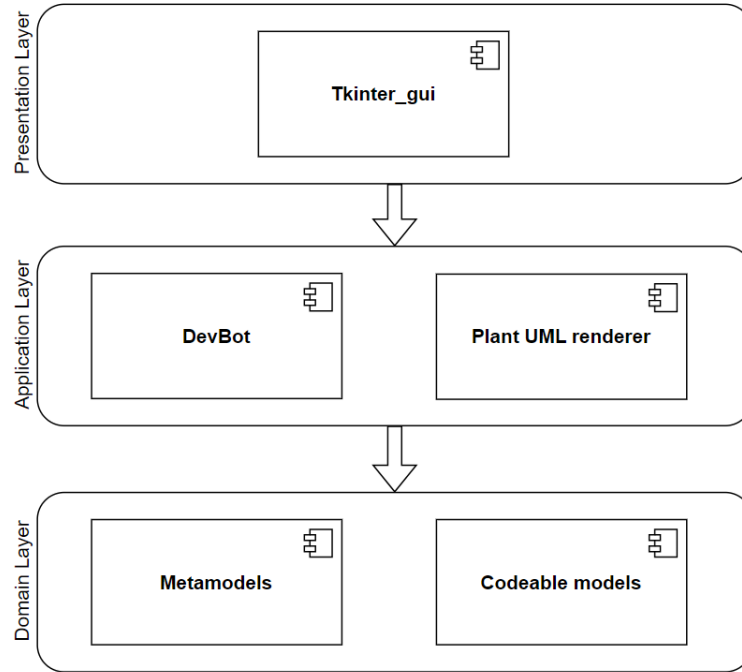


Figure 5.2.: Bird eye view of the system

5.1.1. DevBot GUI

The DevBot GUI is the visualization interface for interacting with the user. It handles user requests and forwards them to the DevBot Backend, where the business logic is implemented. The GUI is easy and intuitive to use by providing convenient visual features to model a secure microservice architecture. It offers an optically appealing interface, which is divided into seven tabs to separate functionalities by context.

The home tab displays a short introduction on how to use the application and includes the functionalities to load or save a model and generate the current component model. The second and third tabs can be used to add or delete microservices or endpoints. Handling external security, the fourth tab is responsible for selecting an appropriate security pattern for external entry points. Besides external security, internal communication also needs to be secured. This task is taken over by the implemented functionalities provided in tab five. Tab six offers the option to add databases to microservices, while the last tab, tab seven, provides an in-tool security evaluation. The functionalities to evaluate the security can be retrieved with a button click and the results are shown. A detailed and visual description will be shown in section 5.4.

5.1.2. DevBot Backend

The DevBot Backend component handles the whole business logic and contains the core functionality of the system. All security mechanisms are implemented in the back-end, divided into several Python files. The application is programmed using Python 3.8. Generally, the DevBot Backend is divided into Python packages which each pursue one functionality. The deployment and the general structure of the DevBot are shown in Figure 5.3. The *DevBot component* contains all the business logic for security aspects, while the *Tkinter component* includes all functionalities for the GUI. The remaining components are needed for generating and creating the component model. Codeable models provide the basic syntax for generating a component model from code. For generating a secure model, the *DevBot component* mainly uses the classes CClass, Clink, and CBundle for creating components and connecting them accordingly.

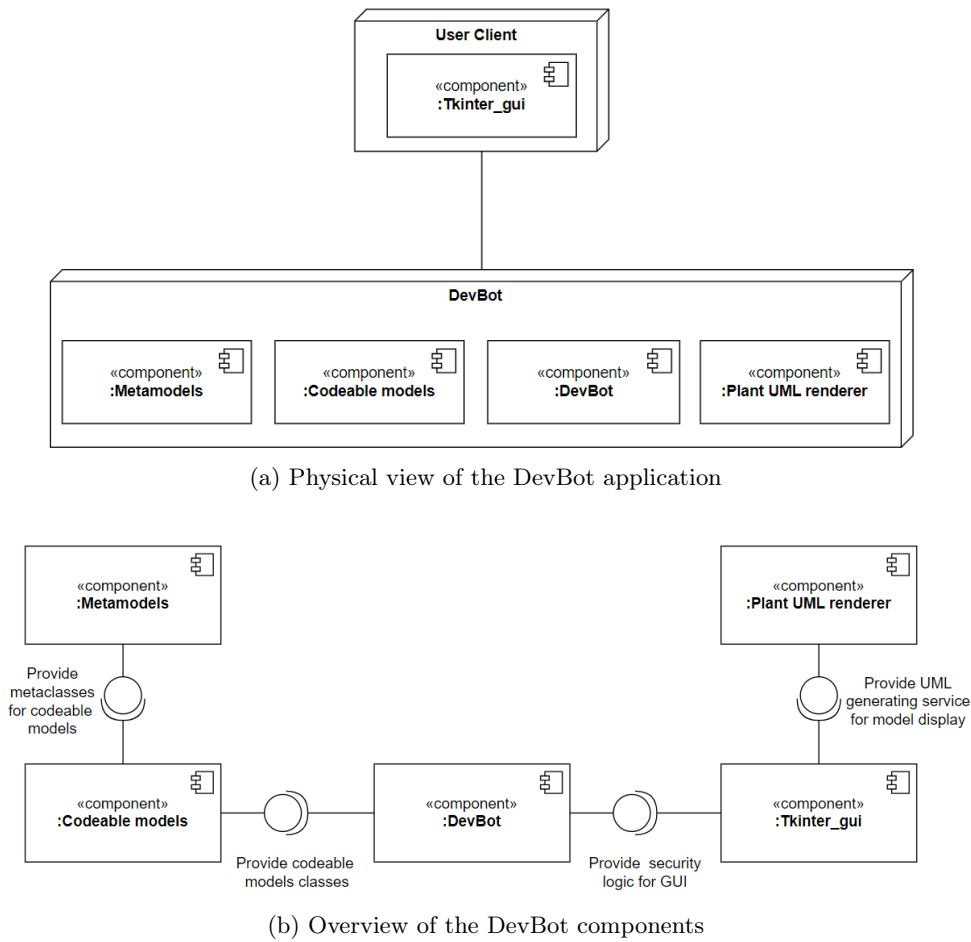


Figure 5.3.: Development view of the DevBot application

As already mentioned, the business logic is implemented in the DevBot component.

5. Implementation

This consists of several services that handle the security functionalities (see Figure 5.4). The *DevBot* service is responsible for basic component services, by handling getting and deleting functionalities. All model-generating functions are collected in the *modelGenerator* service. As the name implies, the *logging* service is responsible for logging all (i) external security and (ii) internal security design decisions, and (iii) saving the created component diagram with all design decisions. The *database* service takes care of the creation of databases and considers the associated microservice. The *externalCommSecurity* service and *internalCommSecurity* service are the two most important services, as they are in charge of the functionality to secure the microservice architecture. They ensure that the user considers both internal and external security when creating the component diagram and that these are taken into account in the model. The *evaluation* service is responsible for the continuous monitoring of the security level. Security risks are constantly checked and reported. The service ensures that the user is informed about the current security status at all times.

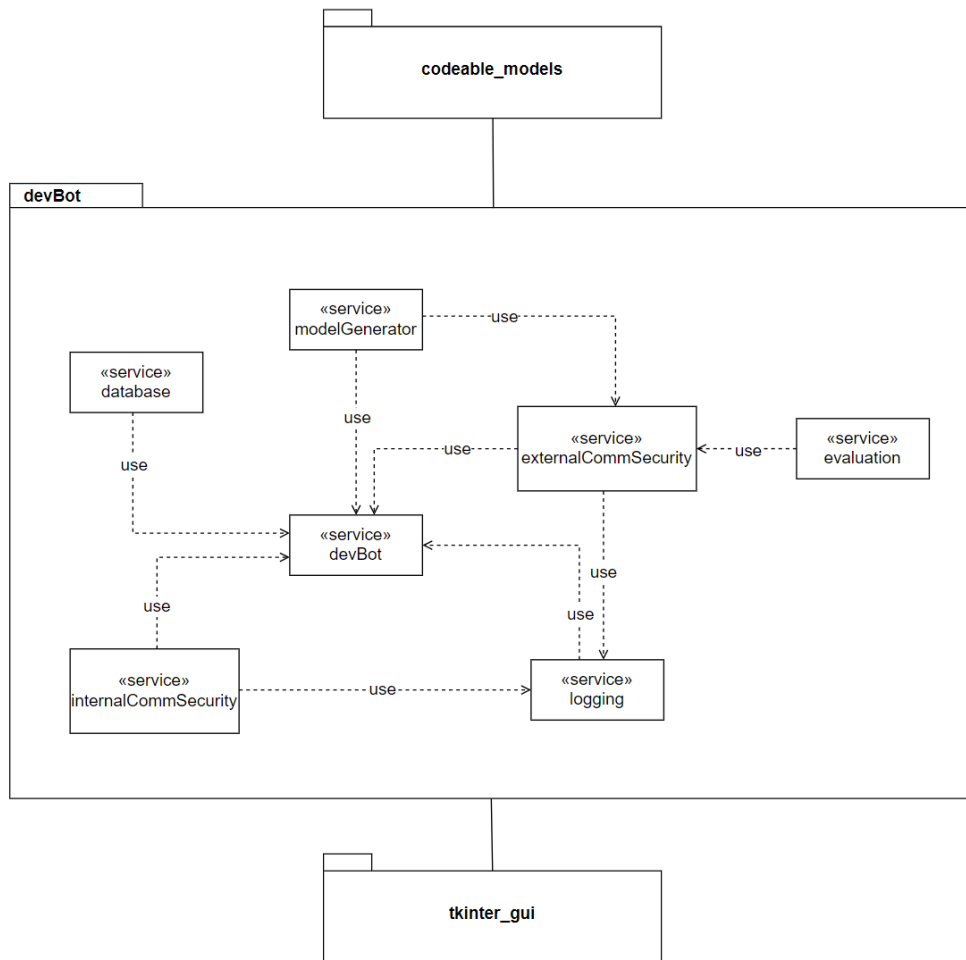


Figure 5.4.: Logical view of the DevBot application

5.2. Use case scenarios

In this section, the considered use cases of the DevBot system are introduced which are depicted in Figure 5.5 with the involved actors. Each use case is described using a uniform template which includes a brief description of the use case itself, the involved actors, and the standard process flow. Moreover, information like preconditions, post-conditions, performance targets, and possible issues of the use cases are given. After the textual description, each use case is modeled as a sequence diagram to show the process flow in more detail.

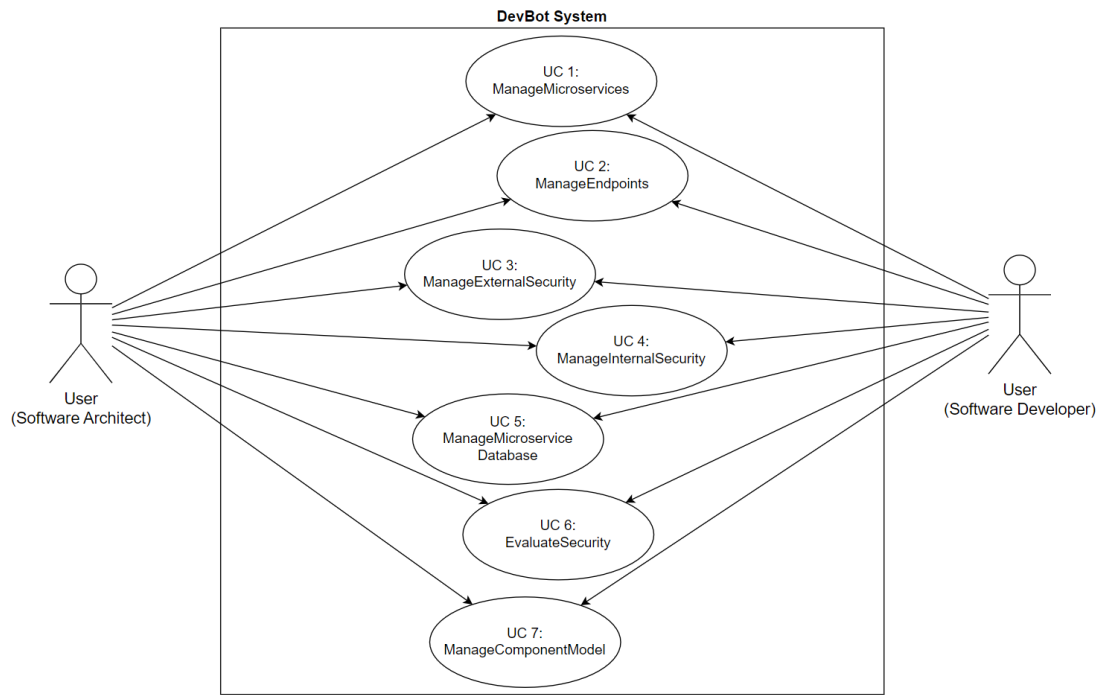


Figure 5.5.: Overview of all considered scenarios of the DevBot system

5.2.1. Use Case 1: Manage Microservices

Brief description. Since it is the main goal of the DevBot system to model a secure component model of a microservice architecture, the first use case handles all microservice-related tasks. The user can add and delete microservices to the model and update external security.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to add or delete microservices. To delete a microservice, the microservice needs to be created first.

Post-Conditions. An updated model is generated and depicted in the model-view window.

Standard process flow. The user opens the "Add microservice" tab in the GUI and enters

5. Implementation

the name of the new microservice. When clicking on the "Add microservice" button, a new microservice component is created in the back-end and the component model is updated in the model-view window. If the user wants to delete a microservice, the name of the microservice has to be inserted and when clicking on the "Delete microservice" button, the system checks if the microservice exists. If it exists, the microservice component and all associated components are deleted, otherwise, the back-end reports back to the GUI that the inserted microservice does not exist. The updated model is generated and depicted in the model-view window.

Performance Target. The DevBot should check automatically if a selected microservice already exists or not. In both cases, the selected action (add or delete) should only be executed if a microservice with the inserted name was found. Otherwise, a message should appear informing the user that the inserted microservice does not exist. This method ensures that no bad input is forwarded.

Possible Issues: Possible issues can be that a microservice exists but is not found during the check. This error is very small because error handling programming is used by checking for invalid input and using error messages.

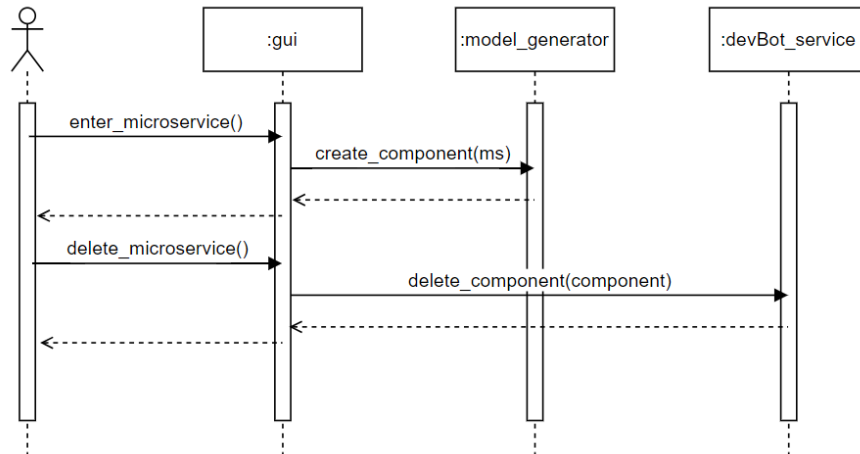


Figure 5.6.: Sequence diagram of the microservice management (UC1)

5.2.2. Use Case 2: Manage Endpoints

Brief description. Besides microservices, communicating with external endpoints is a basic functionality needed in a microservice architecture. UC2 handles all endpoint-related tasks. The user can add and delete endpoints to the model and update external security.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to add or delete endpoints. In order to delete an endpoint, the endpoint needs to be created first.

Post-Conditions. An updated model is generated and depicted in the model-view window.

Standard process flow. The user opens the "Add endpoints" tab in the GUI and selects

the desired endpoint option in a dropdown menu. By clicking on the "Add endpoint" button, the inserted option is verified, and a new endpoint component is created in the back-end, while the component model is updated in the model-view window. If the user wants to delete an endpoint, the name of the endpoint has to be inserted. When clicking on the "Delete endpoint" button, the system checks if the endpoint exists. If it exists, the endpoint component and all connections are deleted, otherwise, the back-end reports back to the GUI that the inserted endpoint does not exist. The updated model is generated and depicted in the model-view window.

Performance Target. The DevBot should check automatically if a selected endpoint already exists or not. In both cases, the selected action (add or delete) should only be executed if an endpoint with the inserted name is found. Otherwise, a message should appear informing the user that the inserted endpoint does not exist. This method ensures that no bad input is forwarded.

Possible Issues: Possible issues can be that an endpoint exists but is not found during the check. This error is very small because error handling programming is used by checking for invalid input and using error messages.

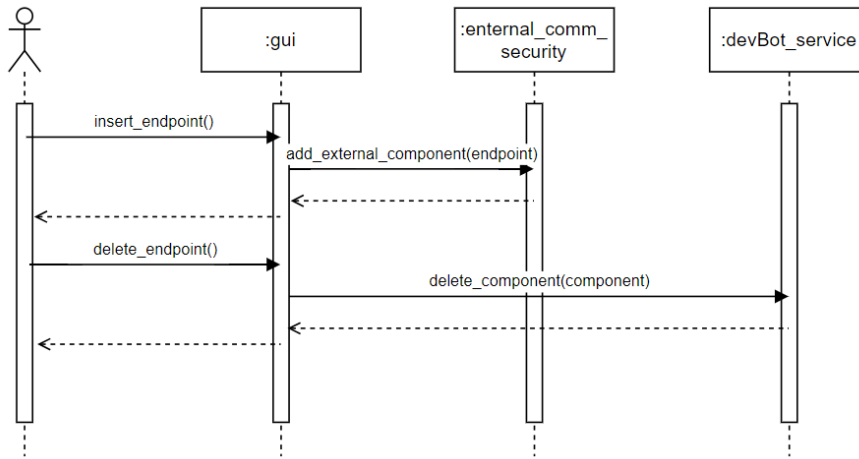


Figure 5.7.: Sequence diagram of the endpoint management (UC2)

5.2.3. Use Case 3: Manage External Security

Brief description. The main task of the DevBot system is to create a secure microservice architecture based on a component model. UC3 takes over the task of external security towards external entry points. Therefore, this use case is very important. All external security mechanisms are controlled in this use case.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to add external security. To handle external security, at least one microservice needs to be created first.

Post-Conditions. An updated model is generated and depicted in the model-view window.

Standard process flow. The user opens the "Add external security" tab in the GUI and

5. Implementation

chooses if the system is handling sensitive data or not.

Option 1: If the system is handling sensitive data, the best suitable security pattern based on the number of microservices is calculated and the result is reported to the user as a recommendation. The user can either accept or deny the recommendation. If the recommendation is accepted, the selected security pattern is created and all existing microservices and endpoints are connected to it to avoid unsecured direct communication. If the recommendation is denied, option 2 is started.

Option 2: If no sensitive data is handled or the user is denied the recommended security pattern, the user can select one of the options via a dropdown menu. By clicking the "Add" button, the selected security pattern is created, and all necessary connections are created. The selected security pattern can be changed during the model development via the "Change" button.

Performance Target. The main performance target is to ensure secure external communication. All endpoints and microservices are only allowed to communicate via a security pattern. The DevBot automatically connects all existing endpoints and microservices with the created security pattern. When the security pattern is changed, all existing external security components are deleted and the new components are created. The connections are immediately updated to ensure a secure microservice architecture.

Possible Issues: Possible issues can be that an endpoint or a microservice is created, but the external security is not updated yet. To avoid this risk, an evaluation of the model can be executed at any time.

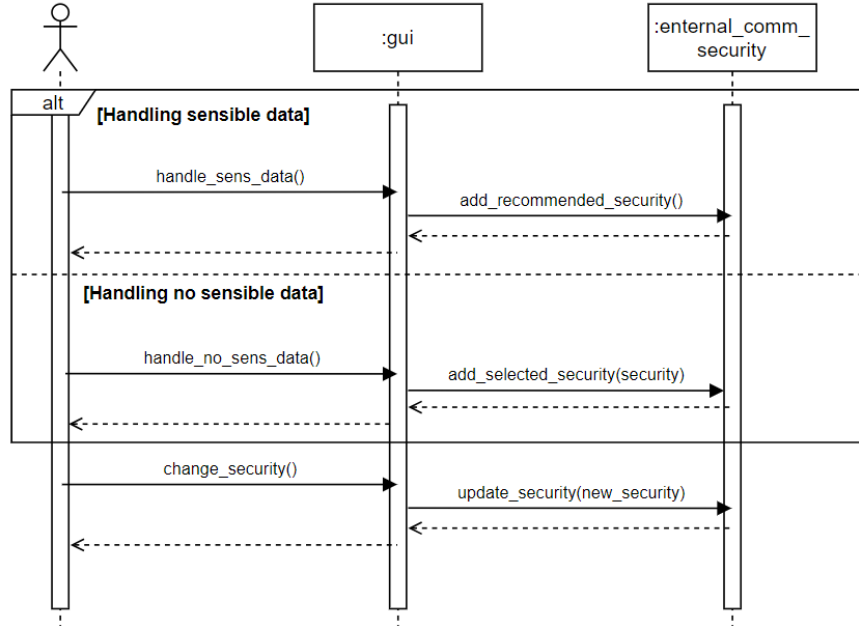


Figure 5.8.: Sequence diagram of the external security management (UC3)

5.2.4. Use Case 4: Manage Internal Security

Brief description. The main task of the DevBot system is to create a secure microservice architecture based on a component model. UC4 takes over the task of internal security between microservices. Therefore, this use case is very important. All internal security mechanisms are controlled in this use case.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to add internal security. To handle internal security, a security pattern needs to be selected, and at least two microservices need to be created first.

Post-Conditions. An updated model is generated and depicted in the model-view window.

Standard process flow. The user opens the "Add internal security" tab in the GUI, wherein a first step the internal security pattern is chosen. By selecting the pattern via a checkbox and submitting the decision with the "Add pattern" button, the component is created in the back-end. After selecting an internal security pattern, the user can add inter-service communication connections between microservices. In order to support the user as best as possible, a list of all microservices will be displayed. The user can now enter all publishing microservice components together with all-consuming microservice pairings and can add the communication by pressing "Submit communication pair". The submitted communication will be added in the back-end and the corresponding microservices will be connected with the pattern.

Performance Target. The main performance target is to ensure secure internal communication. All microservices are only allowed to communicate via a security pattern. The DevBot automatically connects the submitted microservice pairs with the created security pattern and updates the connections if the microservice is deleted.

Possible Issues: Possible issues can be that microservices are communicating directly with each other when an unsecured model is loaded. In order to avoid this risk, an evaluation of the model can be executed at any time.

5.2.5. Use Case 5: Manage Microservice Database

Brief description. To create a realistic microservice architecture model, the user should be able to add databases. Each microservice can have only one database, and the database can not be shared with any other microservice. UC5 is handling all database-related tasks.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to add or delete databases. To add a database, a microservice needs to be created first.

Post-Conditions. An updated model is generated and depicted in the model-view window.

Standard process flow. The user opens the "Add database to microservice" tab in the GUI and selects the desired database option from the dropdown menu. Then the name of the associated microservice also needs to be inserted. By clicking on the "Add database" button, the inserted text is verified by (i) checking if the microservice exists and (ii) validating the selected database option. A new database component is created in the

5. Implementation

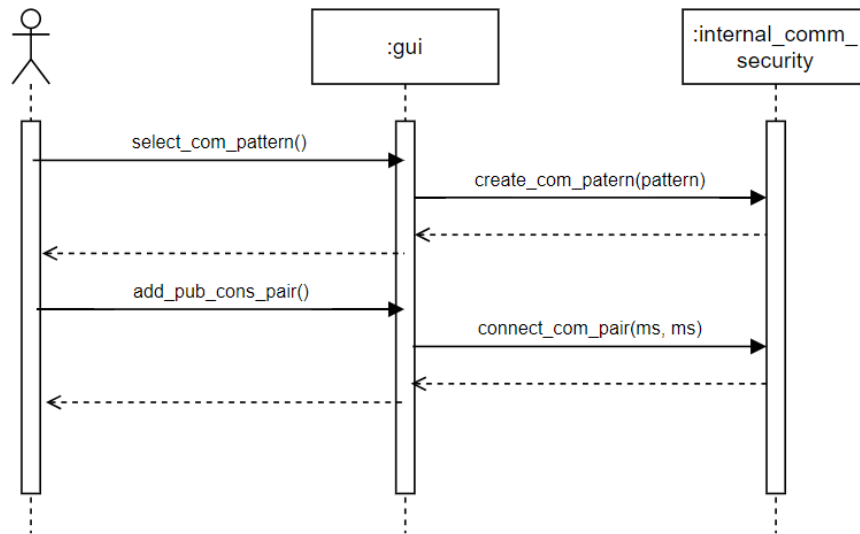


Figure 5.9.: Sequence diagram of the internal security management (UC4)

back-end and the component model is updated in the model-view window.

Performance Target. The DevBot should check automatically if a selected database already exists or not. Creating a database component should only be possible if the selected microservice exists and the selected database is verified. Otherwise, a message should appear informing the user that the associated microservice does not exist, or the input is invalid. This method ensures that no bad input is forwarded.

Possible Issues: Possible issues can be that a database is not deleted if a microservice is deleted. This error is very small because error handling methods are used.

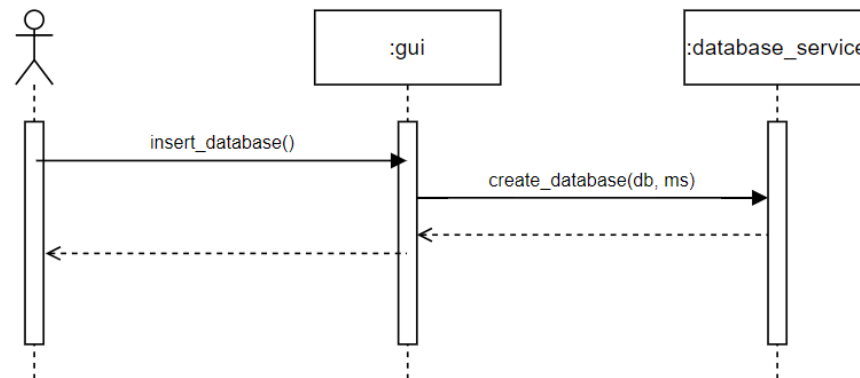


Figure 5.10.: Sequence diagram of the database management (UC5)

5.2.6. Use Case 6: Evaluate Security

Note: The evaluation process is both a use case of the application and an evaluation method for the overall functionality of the developed bot. A detailed description of the system evaluation can be found in chapter 6.

Brief description. The main task of the DevBot is to easily create a secure component diagram. Therefore, use case 6 is very important to evaluate the security risks at any time and to adjust the security. UC6 takes care of all the tasks to evaluate and eliminate security risks.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system needs to be running.

Post-Conditions. The security overview is updated, and the secured model is generated and depicted in the model-view window.

Standard process flow. The user can open the evaluation tab at any time to see the current security level. The security risks are calculated as soon as an action in the bot is executed. The risks are determined constantly, and the safety level is displayed as a visual bar. The user can directly implement measures to increase security. To achieve a higher security level, the user can update both the external security and the internal security by clicking on the provided update buttons. Due to the continuous evaluation of the security level, the component model generally remains very secure, and the user can easily take measures to create a fully secured component diagram.

Performance Target. The DevBot should automatically evaluate the model, after each action the user has taken. The current security level is depicted as a visual bar and can be entered by the user at any time. The calculation of the model security and the corresponding actions that can be taken should be accessible by the user via the update button for external and internal security.

Possible Issues: Possible issues can be that the user is not using the DevBot security measures correctly and is not updating the model correctly, highlighting the potential for improvement.

5.2.7. Use Case 7: Manage Component Model

Brief description. The basic functionality of the system is to support the user by creating a secure component model of the microservice architecture. Therefore, basic functionalities, like saving or loading a component model, need to be available. Use case 7 is handling all modeling-related functionalities.

Involved actors. User (Software Developer), DevBot GUI, DevBot Backend

Preconditions. The DevBot system is ready to work. To load an existing component model based on the codeable models' syntax, the file needs to be in the right directory. If a model is saved, it is a syntactically correct model in the right directory and can be loaded at any time. Otherwise, the model needs to be either transformed into a codeable model-styled model or needs to be moved to the directory `/thesis_ms_security/devBot`.

Post-Conditions. The loaded model is generated and depicted in the model-view window. The saved model can be found in the correct directory.

5. Implementation

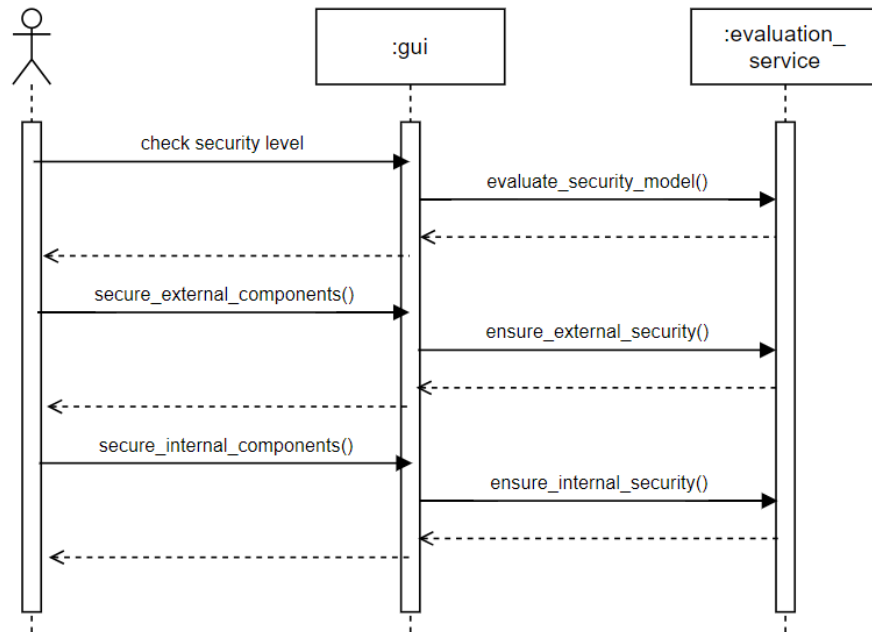


Figure 5.11.: Sequence diagram of the security evaluation (UC6)

Standard process flow. The user starts the application, which opens the "Home" tab in the GUI. Then the user can start working with the DevBot system. Instructions are displayed on the "Home" tab which states the operation of the bot and all provided functionalities. The user has the option to load an existing model into the application, which is implemented with codeable models. Security risks are automatically evaluated, and the model is automatically generated in the component model-view window. In addition, the user can either re-save the extended model that was previously loaded into the app or save the model that was created from scratch.

Performance Target. The DevBot should automatically open the desired model file and generate the component model, by analyzing the uploaded code. Furthermore, the DevBot should be able to save the created model in a syntactically correct way. The DevBot should use the codeable models' syntax and should create an executable Python file of the model containing all created links and components.

Possible Issues: Possible issues can be that the model to be loaded is not in the correct directory or was not created with codeable models. Then the DevBot either can not find the file or the model can not be read and edited by the DevBot.

5.3. Requirements

Besides identifying the use cases of the DevBot system, the functional requirements for the system must first be determined. The bot should be intuitive to use and quickly and

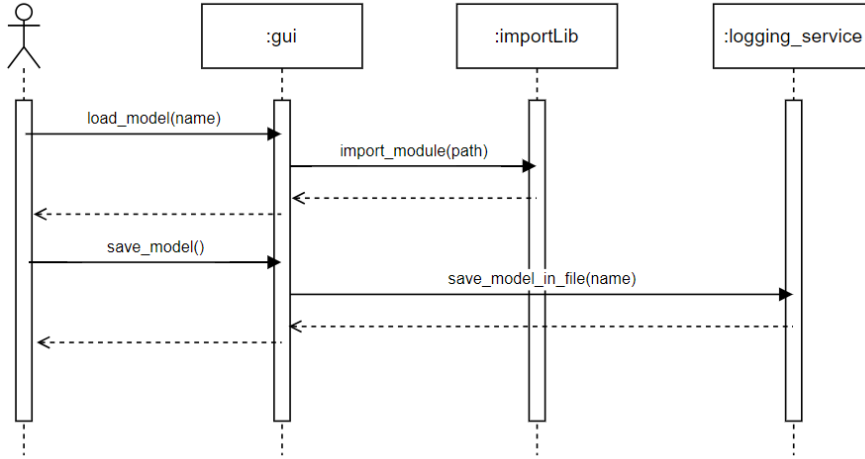


Figure 5.12.: Sequence diagram of the component model management (UC7)

easily create a secure component model for a microservice architecture. The tool should be usable without background knowledge of modeling or microservice security. Based on these assumptions, the following functional requirements for the system are derived:

- R1: Update microservice security when a new microservice is added.
- R2: Update microservice security when a new endpoint is added.
- R3: Update microservice security when microservice is deleted.
- R4: Update microservice security when an endpoint is deleted.
- R5: Change external security pattern when microservice architecture evolves.
- R6: Update inter-service communication between microservices.
- R7: Evaluate the model at any given time to calculate security risks.
- F8: Load and save models at any given time.

A detailed description of each requirement can be found in the following table (see 5.1). The mentioned use cases are described in more detail in section 5.2.

5. Implementation

Table 5.1.: Overview of system requirements

ID	Priority	Brief description	Affected UC	Affected role	Verification
R1	High	Each time a user adds a microservice, the security mechanisms need to be adapted to keep the whole system secure. This can only be reached by connecting the microservice with the selected security pattern.	UC1, UC3	User	Via in-tool evaluation
R2	High	Each time a user adds an endpoint, the security mechanisms need to be adapted to keep the whole system secure. Endpoints should only be able to communicate via the selected security pattern and not be directly connected with the microservice	UC2, UC3	User	Via in-tool evaluation
R3	High	Every time a user deletes a microservice, the security mechanisms need to be adapted to keep the whole system secure. Ongoing API connections need to be deleted and all associated components (e.g. database) need to be deleted.	UC1, UC3, UC5	User	Via in-tool evaluation
R4	High	Every time a user deletes an endpoint, the security mechanisms need to be adapted to keep the whole system secure. Open API connections need to be deleted.	UC2, UC3	User	Via in-tool evaluation
R5	Medium	Since microservice architecture is constantly changing, users need a way to adjust their security. With more complex systems, external security requirements change and new external security is needed.	UC3	User	Via in-tool evaluation
R6	High	When a user loads a component model into the application, it needs to be checked for microservices that communicate directly with each other. The communication needs to be updated to communicate via a security pattern.	UC4	User	Via in-tool evaluation
R7	High	The user should be able to check the security and evaluate the risks at any time.	UC6	User	Via in-tool evaluation
R8	Low	The user should be able to save the model as often as desired and load it when the application is opened again in order to continue working on the model.	UC7	User	Via in-tool evaluation

In addition to fulfilling the use case scenarios, the DevBot is designed to meet all requirements. Each requirement is ranked according to its priority. The higher the priority, the higher the potential security risk for the system. For this reason, special attention is paid to the implementation of the requirements while programming the DevBot. All requirements are included in the DevBot system and are an extension of the predefined use cases.

5.4. User Interface

To create a helpful bot, usability is very important, as the bot develops its strength only when used correctly. The user interface should be kept clean, simple, and intuitive to use. For the implementation of the GUI, the package *customTkinter* was used. In the following, first, the development process is described and then an instruction on how to use the DevBot is given. Figures 5.13 - 5.15 are showing some of the implemented optimizations in comparison to the previous DevBot GUI.

5.4.1. Development process

In general, the user interface was developed in several iterations. After each iteration, the GUI was tested by other software developers and the feedback was incorporated into the next design for the GUI.

In the first iteration, the structure of the individual tabs was standardized to improve clarity and intuitiveness, and a top-down workflow was established. In a second step, feedback messages were added to notify the user of success or failure after a button was pressed, for example, whether a model was successfully loaded or not. Also, a button has been added on each tab to return directly to the home tab. All these changes provide a cleaner and simpler look, and thus for easier and more intuitive use of the bot.

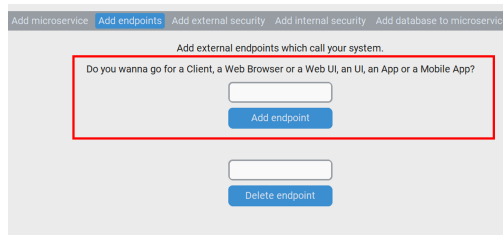
In addition to the visual changes in the first iteration rounds, improvements have also been established on the functional side. The component model is automatically updated in a second model-view window after each action that affects the model. This means that the user automatically has the current visual status of the component model at all times. The automation of model generation makes the system more fail-safe, as the user no longer has to remember to generate the updated model.

In a second major round of iterating the GUI, first optical optimizations are implemented, before functional optimizations are realized. Basically, all tabs are made clearer and headings are added to the individual functionalities so that the user can see at a glance which functionalities are available. In the "Home" tab, the instructions for the use of the bot are also updated to be as comprehensible as possible.

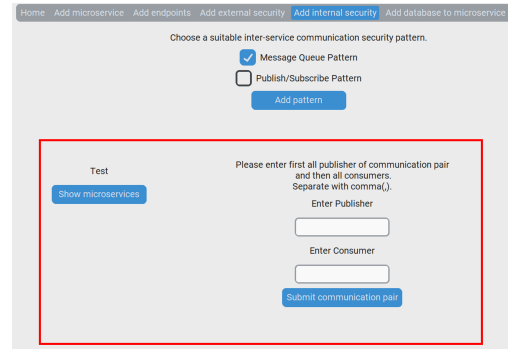
In the "Add microservice", "Add endpoints" and "Add internal security" tabs, the list of all existing microservices and endpoints has been updated so that the user always has an overview of all the available components for the desired actions without having to search in the model. This simplifies, for example, the deletion of microservices or the

5. Implementation

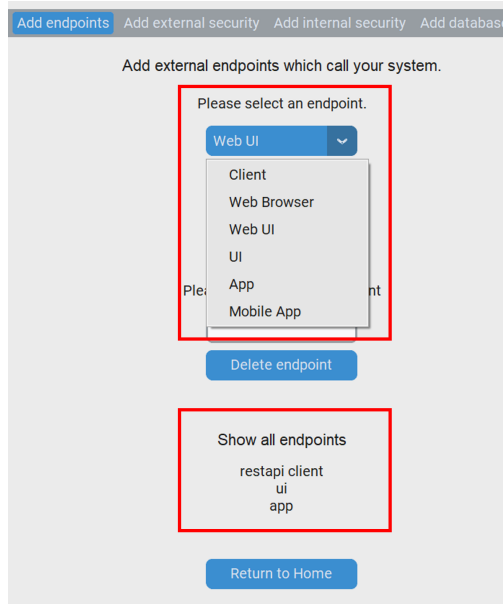
addition of internal communication between two microservices, as the user can search the list for the required microservice at any time. In the tab "Add endpoints" the input for new endpoints has been optimized. Instead of entering the desired endpoint textually, the user can select and add the desired endpoint component using a dropdown menu. This ensures that the bot is less error-prone, due to invalid input.



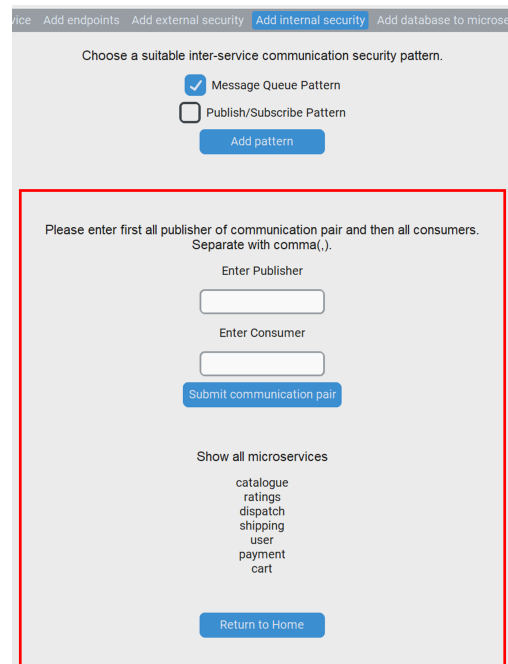
(a) "Add endpoints" view before optimization



(b) "Add internal security" view before optimization



(c) "Add endpoints" view after optimization



(d) "Add internal security" view after optimization

Figure 5.13.: Overview of "Add endpoints" and "Add internal security" views optimization

Adding a database has also been optimized by a dropdown menu. The user can select the desired database and add it to a microservice. This also increases error handling and makes the bot less vulnerable to errors. Initially, the tab for evaluating the safety level of the component diagram has been made easier to read. The overview bar of the current security level is continuously updated to keep track of the risks. The evaluation button has been removed, as this function has been automated to avoid possible misuse by the user. The security level is automatically recalculated and visually displayed after each action is performed. The user can access the "Evaluate model" tab at any time and see the current security level of the component model. The automation of the model evaluation saves time, simplifies usage, and is less error-prone. In addition, a function that continuously checks the size and complexity of the model and gives external security recommendations has been added.

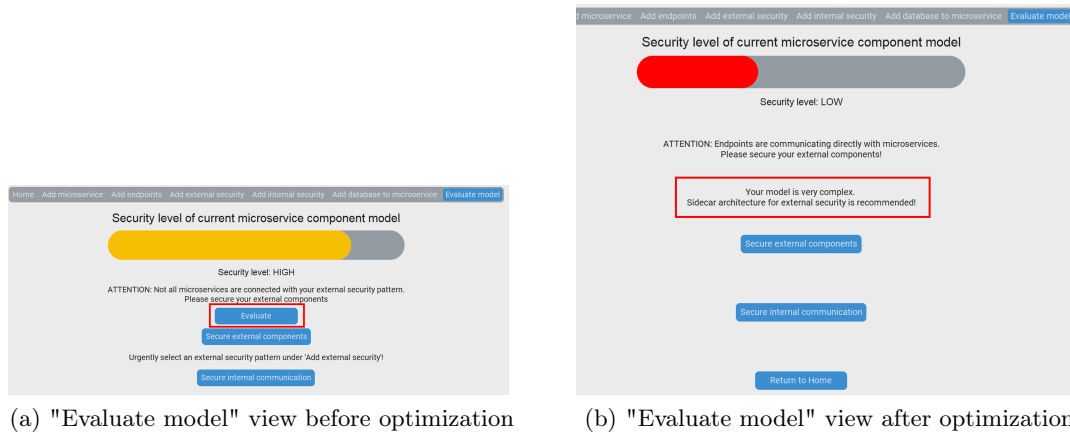


Figure 5.14.: Overview of "Evaluate model" view optimization

In general, a vertical scrollbar has been added to dynamically move the view. A horizontal scrollbar has also been added to the component model-view window. In summary, the feedback showed that a simpler and cleaner look of the DevBot increases usability, as the usage is more intuitive and self-explanatory. Automating important functionalities reduces the bot's vulnerability to errors caused by user misuse.

Future improvements include the use of dropdown menus for deleting operations of microservices and endpoints. In addition, dropdown menus would facilitate the addition of internal communication pairs. These changes would also make the bot less prone to invalid input. This requires the dynamic creation of the dropdown options, which are currently hard-coded. Besides dropdown menus, another future optimization option is not to display the component model in an external window as it is currently but to include it in a tab window as well and use vertical and horizontal scrollbars to go through the model. As an alternative to this option, adding a button in the "Home" tab that pulls the model-view window into the foreground on click is another possibility.

5. Implementation

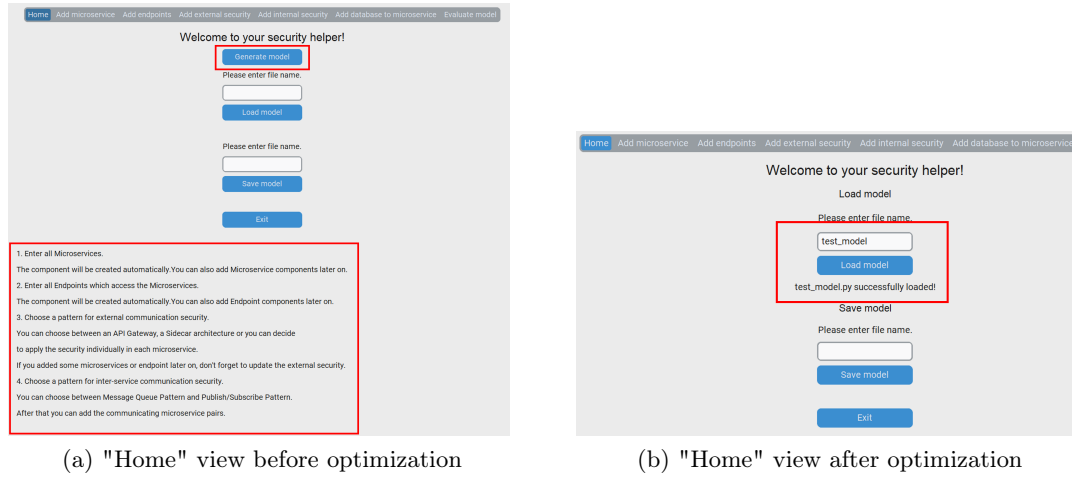


Figure 5.15.: Overview of "Home" view optimization

5.4.2. User instructions

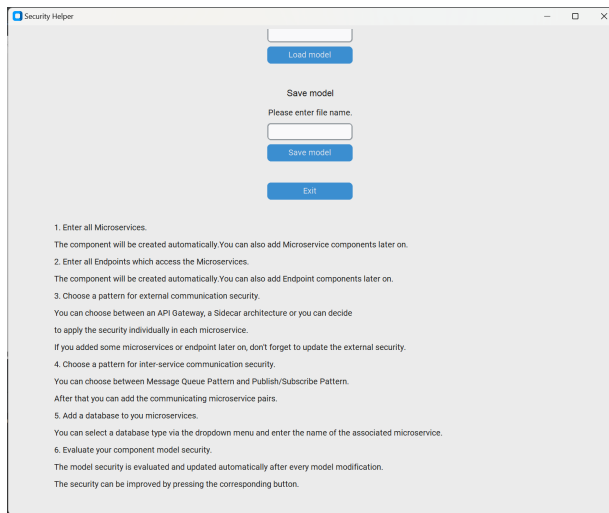
To use the DevBot the user must first start the program. The GUI is generated automatically, and the user can start modeling. Two scenarios are shown below: (i) creating a new microservice architecture from scratch and (ii) editing an existing component model. The example model is created based on the microservice model "robot-shop"¹ and security errors are inserted for demonstration purposes.

Creating a model from scratch

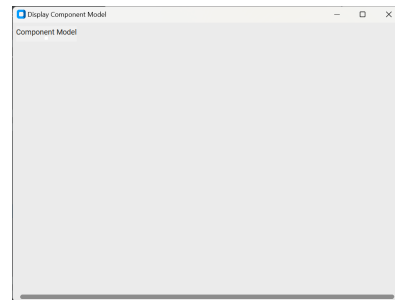
After the user has started the application, the tool opens in the "Home" tab and the component model view window. Instructions for using the tool can be found in the "Home" tab.

The user can now start and add microservice components to the model. The user can now start and add microservice components to the model by opening the "Add microservice" tab and entering the name of the microservice. In this example, the five microservices user, inventory, payment, shipment, and rating have been added. The user can delete a created microservice at any time by entering the name of the microservice to be deleted in the corresponding field and pressing "Delete microservice". All existing microservices are displayed in a list. Next, the user can add endpoints to the model in the "Add endpoints" tab using a dropdown menu. In this example, the endpoints client and web UI were added.

¹<https://github.com/instana/robot-shop>

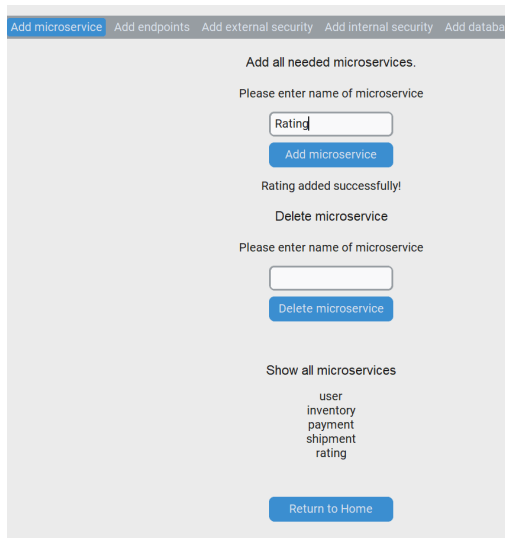


(a) Start application in "Home" tab view

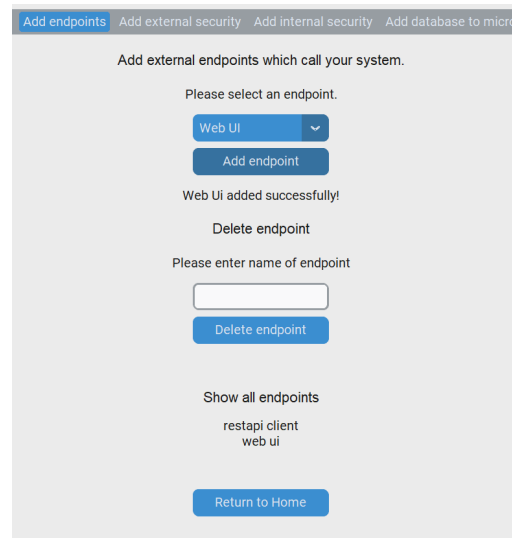


(b) Default component model view

Figure 5.16.: Create new model: Start application



(a) Add microservices



(b) Add endpoints

Figure 5.17.: Create new model: Add microservice and endpoint components

The user can check the current security of the model at any time. In the "Evaluate model" tab, the user can view the current security level.

5. Implementation

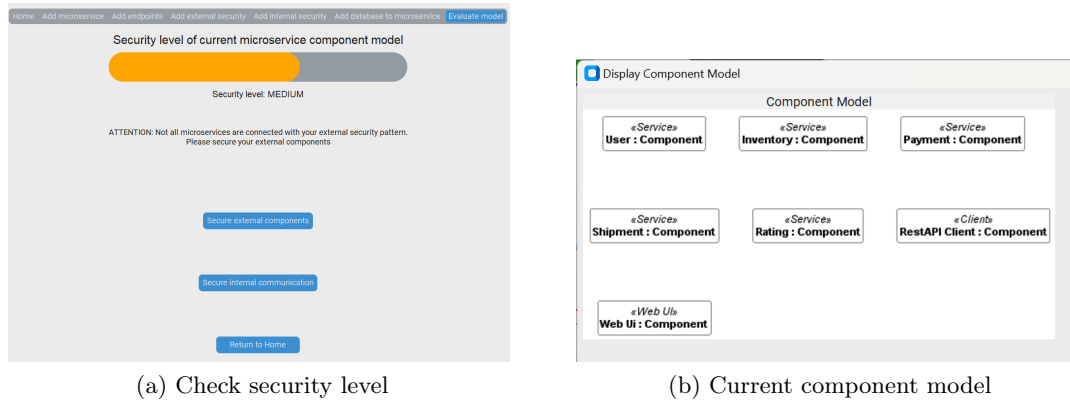


Figure 5.18.: Create new model: Check current model security

To increase the security level, external security patterns are added next. For this purpose, navigate to the tab "Add external security" and select the appropriate external security pattern by answering the questions via the buttons. When the pattern is selected, the user can check the security level again. For this example, the pattern "sidecar" was chosen.



Figure 5.19.: Create new model: Add external security and check model security

After external security aspects have been included, the next step is to manage internal security aspects. The user can select via a checkbox which communication pattern is used. Then he can specify the inter-service communication between microservices. In this

example, the Message Queue pattern was selected and communication between the user and payment, and rating was added.

Finally, the user can attach databases to the microservices. Using a dropdown menu, the desired database can be selected, and the corresponding microservice name can be entered in the field provided. In this example, a MySQL database for the microservice user and a MongoDB database for payment has been added.

(a) Add internal security

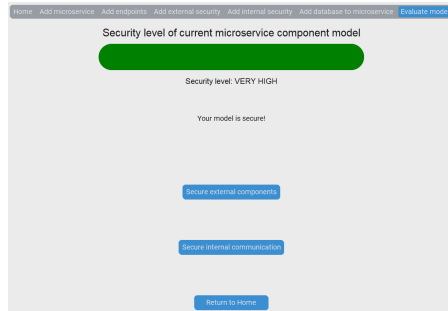
(b) Add database

Figure 5.20.: Create new model: Add internal security and databases

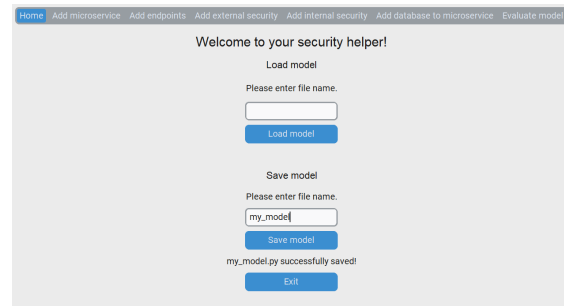
In the end, the security level of the created model can be controlled and the just-created model can be saved in the "Home" tab. In order to save the model, the user must enter a

5. Implementation

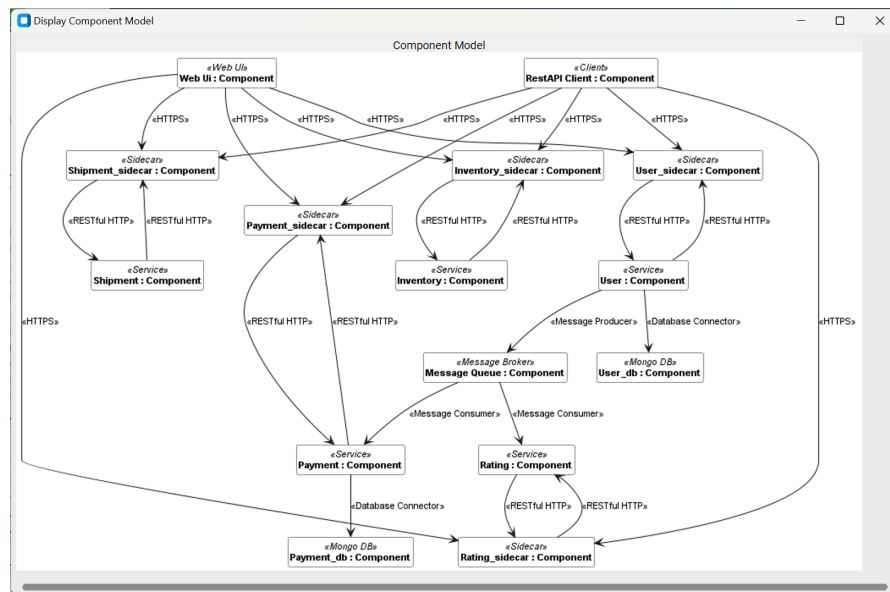
file name and the model will be saved.



(a) Check current model security



(b) Save component model



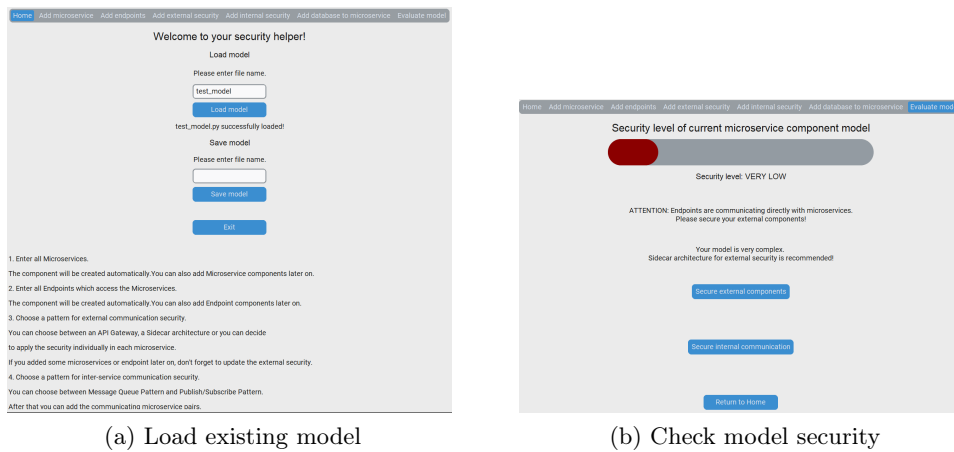
(c) Check current model

Figure 5.21.: Create new model: Check model security and save created model

Once, the model is saved, the user can exit the tool via the "Exit" button.

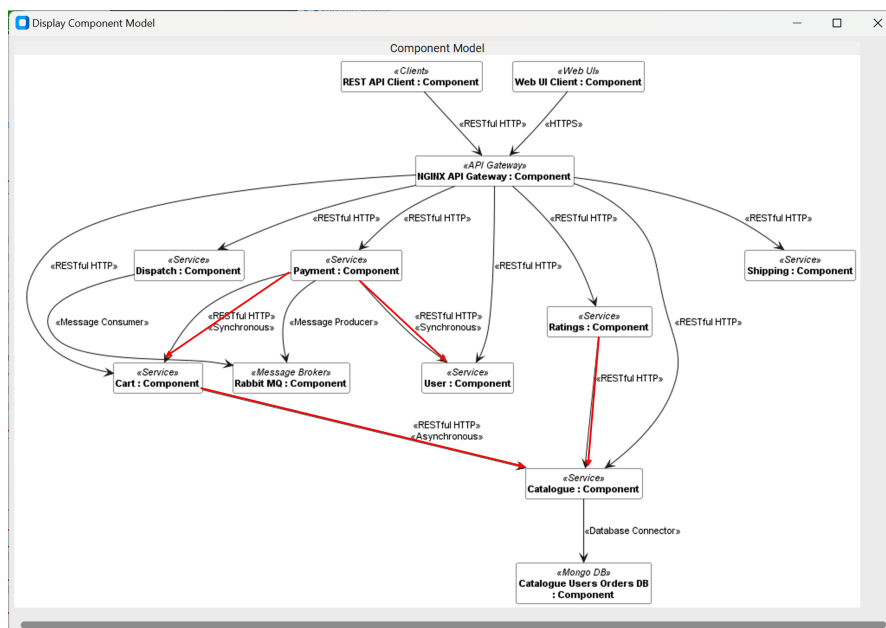
Editing an existing model

To edit an existing component model and check whether all security aspects are fulfilled, the user can load models directly into the application. In order to achieve this, the user must enter the name of the file to be loaded in the field provided, and the model is automatically displayed in the model-view window. In this example, the faulty components are highlighted in red to emphasize the modeling errors. To immediately check how secure the loaded model is, the user can query the current security level under the tab "Evaluate model".



(a) Load existing model

(b) Check model security



(c) Check current model

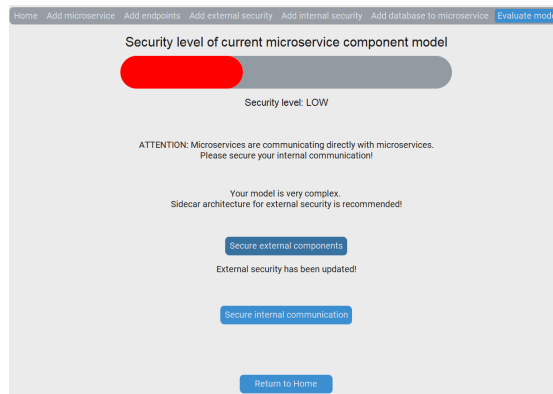
Figure 5.22.: Edit model: Load model and check model security

The evaluation shows that the user has to update the external security. For this purpose, the user can press the button "Secure external components" and the tool automatically secures the model for external entry points. Also, the new security level has been automatically calculated.

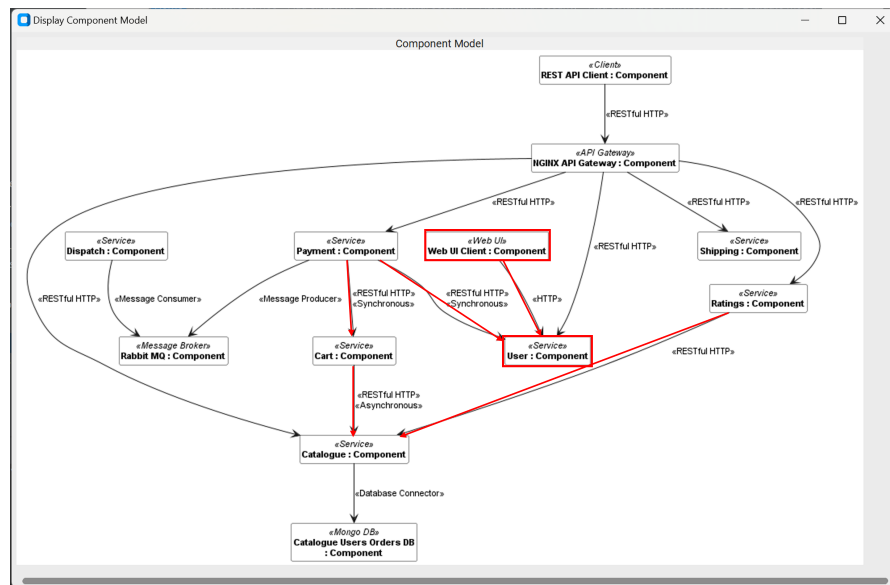
Now the user gets the message that internal security has to be improved as well. With the "Secure internal communication" button, the internal communication is automatically secured and the new security level is calculated.

The user can now continue working on the model or save the updated secure component model.

5. Implementation

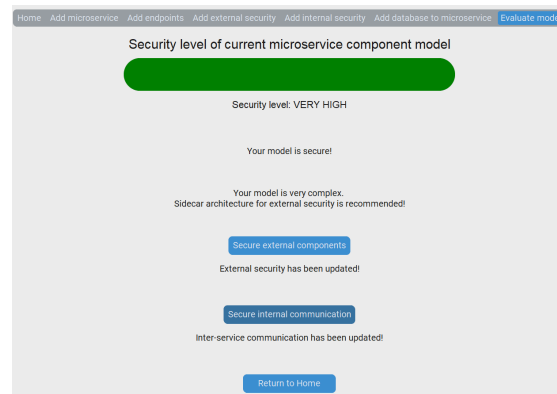


(a) Update external security

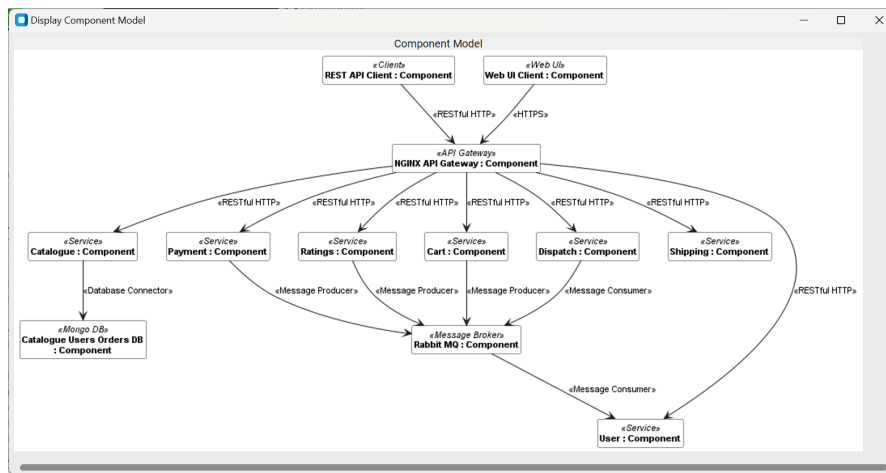


(b) Check current model

Figure 5.23.: Edit model: Update external security and check model security



(a) Update internal security



(b) Check current model

Figure 5.24.: Edit model: Update internal security and check model security

5.5. Extensions

As already mentioned, security in microservice architecture has received little attention so far. Microservice security patterns have to be searched for by software architects for each usage. The DevBot offers a simple solution to include security patterns during microservice decomposition.

5.5.1. Benefits of extensibility

The DevBot is programmed in a way that the application can be easily extended with additional external security patterns without much effort. This allows the DevBot to be kept up to date by software architects in collaboration with security experts and accounts for current security measures. This design decision adds additional value to the application since the DevBot can be adapted to the respective safety regulations.

5. Implementation

Since software developers usually have to search for patterns manually first, then format them into codeable models before adding them to the DevBot and subsequently using them in the modeling process, the extensibility of the DevBot offers great potential. By having security experts ensure that the pattern has been added correctly, the user saves time and no specific security knowledge is needed.

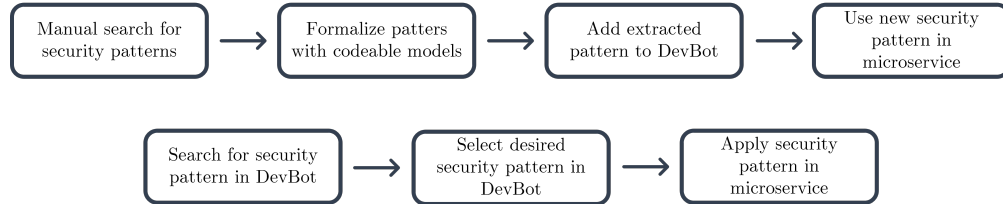


Figure 5.25.: Comparison of ease of extensibility

5.5.2. How to extend external security patterns

The DevBot can be easily extended as previously mentioned. This process is explained step by step, using the Backends for Frontends (BFF) pattern. Before the method can be implemented, the first step is to add the security component to codeable models. This is necessary to be able to create the object at all. Therefore, the file *microservice_components_metamodel.py* in the package "metamodels" needs to be modified, and the component needs to be added as follows before it can be used in further methods:

```
1 backends_for_frontends_gateway = CStereotype("Backends for Frontends  
    Gateway", superclasses=facade)
```

Listing 5.1: Add new external security component

After adding the security component, the next step is to implement the logic of the security pattern. By using the same logic to add external security, "backend for frontends gateway" can easily be added to the DevBot. Since one BFF is focused on a single endpoint component, each endpoint needs its own BFF component. To achieve this, as soon as a new endpoint is added, a new BFF needs to be created, if the user selects the BFF pattern as an external security mechanism. Therefore, the implemented logic needs to include the identification of all existing endpoints, so that each endpoint has its own BFF component. Furthermore, all existing microservices need to be connected via the BFF component to ensure safe communication between the front end and back-end. The following code needs to be implemented in the file *devBot/external_comm_security.py* to add the BFF pattern:

```
1 def add_BFF():  
2     model = get_all_components()  
3     components = get_all_microservices(model)  
4     endpoints = get_all_endpoints(model)  
5     for i in endpoints:  
6         name = str(i.__getattr__("name") + "_bff")  
7         bff = CClass(component, name, stereotype_instances=[  
            backends_for_frontends_gateway])
```

```

8      clink.add_links({i: bff}, role_name="target",
9                      stereotype_instances=[restful_http])
10     for j in components:
11         clink.add_links({bff: j}, role_name="target",
12                         stereotype_instances=[restful_http])

```

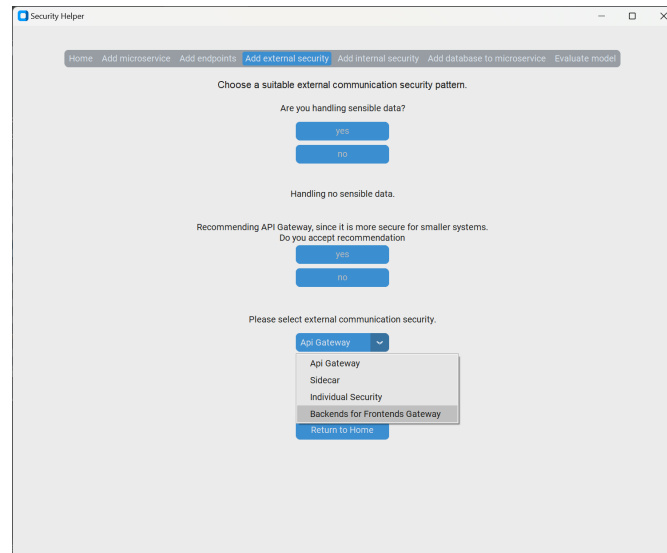
Listing 5.2: Function to add external security extension

This function contains the main functionality for creating the CClass component and all necessary connections. Besides creating the component, modifying the microservice component model is the basic functionality of the DevBot to create pleasant usability. Therefore, updating external security is very important to ensure a secure model and serve these basic functionalities. To achieve this, the `def update_external_security()` needs to be extended. Besides updating the component model when additional endpoints are added, the user has the option to change the external security mechanisms at any point. Therefore, the function `def change_external_security(new_security)` also needs to be extended. Example code snippets can be found in the appendix (see section A.1). The external security pattern operations are following a rule-based logic. According to the selected external security pattern, the associated function is called to create, update, or change the pattern for the microservice architecture.

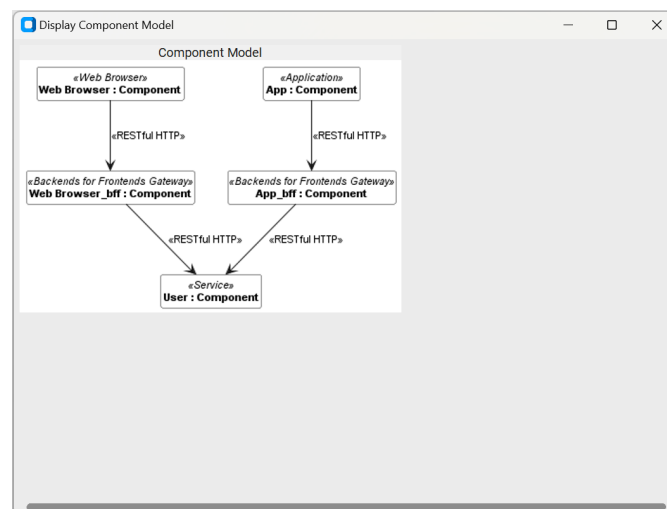
Besides extending the back-end functionalities, the GUI also needs to be updated. The GUI has to be adapted to include the created extension. For this purpose, the dropdown menu has to be extended by the new option and the function `def create_external_security(security_type)` has to be modified. Currently, the pattern is selected via a hard-coded dropdown menu. In future prototypes, the goal is to create a database or file of all possible external security patterns, which contains all necessary operations and properties. The GUI will then directly access this file.

The exemplary extension process shows that the DevBot system is easy to extend and can be adapted and grown as desired. Figure 5.26 depicts the usage of the new extended external security component of the updated DevBot application.

5. Implementation



(a) Updated GUI for added extension



(b) Component model with new extension component

Figure 5.26.: Overview of the updated GUI with new extension

6. Discussion

The validation of the DevBot system is crucial to study the reasonability of the development approach and identify additional challenges. The DevBot system is built using the developed security decision tree, which means that all security risks are taken into account. This leads to a comprehensive security application.

In general, it is important to test the DevBot in two different ways. First, the in-tool evaluation functionality must be tested. For this purpose, a conformance check is performed during which the individual risk calculations are first explained and then the result is manually checked. The component model created is checked for correctness and safety. In addition to the in-tool evaluation functionality, the DevBot system itself is also validated. For this, it is checked whether all requirements are fulfilled and whether the hypothesis is disproved or true. For this purpose, microservices already created with codeable models are used to check whether the DevBot works as expected.

Before discussing the validation process, the DevBot System is compared to normal modeling tools¹²³⁴. While the creation of UML diagrams with normal modeling tools requires time-consuming drag-and-drop operations to create the individual components and then add the names manually, the DevBot generates the component model automatically. The user therefore needs no modeling knowledge and modeling errors like wrong UML syntax are avoided. The user utilizes the DevBot as an interactive security tool that adheres to UML syntax and also provides an integrated security check for the generated component model of the microservice architecture.

The DevBot's focus is not on pure modeling, but more on the inclusion of security-relevant components. While normal modeling tools offer a variety of possible model types (e.g. class diagram, sequence diagram, or use case diagram), the DevBot always creates a component model of the microservice.

In summary, the DevBot is not a modeling tool, as the intention of the DevBot is not the modeling itself, but the application of security mechanisms during the development of a microservice.

¹<https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software>

²<https://staruml.io/>

³<https://www.drawio.com/>

⁴<https://www.visual-paradigm.com/features/uml-tool/>

6.1. Conformance checking of the DevBot system

As reported in section 5.4.2, in-tool conformance checking continuously evaluates the model created by the user. Generally, security risks are divided into two areas: secure external communication and secure internal communication. The in-tool conformance checking is based on a risk calculation using three criteria. The first criterion is the number of microservices that are not yet secured with an external security pattern. The second criterion concerns the number of endpoints that are not connected to an external security pattern. The last criterion is internal security. In this case, all microservices that exchange information with other microservices but do not use an internal security pattern are determined. The individual calculations of the criteria are described in more detail in the following.

6.1.1. Evaluating microservice security

Since unsecured microservices are the biggest security risk in a microservice architecture, microservice security is evaluated first. For this purpose, all microservice components are identified that are either (i) directly connected to an external endpoint, (ii) not connected to an external security pattern or, (iii) directly connected with another microservice. Whereby, the former poses a significantly higher risk potential. The `def check_microservice_connections()` function executes this identification of vulnerable microservices, by checking all connections to other components and extracting relevant linked components according to the identified vulnerable component connections.

In addition to the identification of vulnerable microservices, it is relevant to identify threatening endpoints. Endpoints that communicate directly with microservices are a very high-security risk. Therefore, the next step is to determine all endpoints that are directly connected to a microservice or have no connection to an external security pattern. Since all external endpoints should be secured via a second security layer, this check is essential. This procedure is implemented in function `def check_endpoint_connections()`. This function is structured like the function `def check_microservice_connections()`, it first checks direct connections between endpoints and microservices and stores them in a list before identifying unlinked endpoints.

6.1.2. Calculating security risk level

Once all vulnerable components have been identified, the overall security risk can be calculated.

To calculate the risks a security maturity model approach is used. The security is analyzed based on five levels of maturity [73]. If a model is on level 1, security is unstructured and unorganized, and the development of a security policy is recommended. Level 2 already includes security policies, making security processes repeatable, but there are no cross-functional policies yet. Standardized processes and procedures are introduced in level 3, while security controls are monitored in level 4. In level 5, security processes

6.1. Conformance checking of the DevBot system

are continuously analyzed and improved to ensure optimal security for all systems [73]. Based on these levels, the security risk levels are established.

Since unsecured external endpoints and inter-service communication entail high risks, the security maturity level is low. Therefore, the risks are first sorted according to their relevance. Direct communication between external endpoints and microservices has the highest risk potential. The second-highest risk potential is direct communication between two microservices, followed by microservices without a connection to an external pattern, and finally endpoints without a connection to an external security pattern. The risk assessment examines the different risk potentials in sequence and thus determines the overall security level.

In general, the security risks are derived from the security maturity level. The DevBot offers a standardized process to secure the model and provides monitoring of the current security.

External risks, which pose the greatest risk potential, severely reduce the security level. Unlinked microservices and endpoints also significantly reduce the security level. The more microservices or endpoints that are not linked to an external security pattern, the higher the risk. In addition to external risks, internal communication also plays a major role in risk assessment. Depending on how many microservices communicate directly with each other, the risk increases, and the security level drops.

6.1.3. Updating microservice security

Besides calculating security risks, conformance checking offers the opportunity to eliminate the identified risks. To this end, all unsecured components are revised either with external security measures or with internal security measures. These actions minimize the security risk of the microservice system and ensure a secure microservice architecture.

For this purpose, all previously identified unsecured connections are resolved and replaced by secure ones. This means for external security that the communication passes through the selected external security pattern, and for internal security that the communication follows the selected inter-service communication pattern.

A detailed breakdown of the code can be found in the appendix (see section A.2).

To check if the actual execution of the business logic conforms to the model, the created output is evaluated. To confirm the success of the DevBot implementation, it is examined whether the updated component diagram which can be created with the DevBot considers and minimizes all security risks.

I decided to implement the DevBot according to the established security decision tree because all security guidelines from OWASP [72] are incorporated.

Therefore, the created component diagram of the microservice system is used to confirm whether the DevBot is successful or not. The conformance checking is performed on the same example as already used in chapter 5.4.2. The example model is based on

6. Discussion

the microservice model "robot-shop"⁵ and security errors are inserted for demonstration purposes.

The results of securing the microservice architecture can be seen in Figure 6.1. Figure 6.1 (a) shows the inserted model without proper security, while Figure 6.1 (b) shows the model after the DevBot system evaluated the risks and updated the model to the required security standards. The elements marked in red are external security risks, whereas the blue elements are internal security risks. Elements marked in green are components required for a secure microservice system.

For the DevBot to be successfully implemented, it requires a component diagram that prevents security risks. The secured model shows that an API gateway was used as an external security pattern that takes over the task of authorization to secure the microservice. Broken Object Level Authorization and Broken Functional Level Authorization are prevented since the use of an API gateway ensures that the endpoints of a microservice are not exposed and that the business logic is separated from other functionalities. Furthermore, the use of the external security pattern reduces the risk of Broken Authentication because an extra layer is provided for security aspects. In addition to an external security pattern, an internal security pattern has also been used. After evaluating the risks of the DevBot application, microservices communicate only via a message broker. This minimizes the risk of Unrestricted Access to Sensitive Business Flows, as only selected information is exposed. It also mitigates the risk of Excessive Data Exposure. The evaluation of the DevBot system demonstrates how security mechanisms are included (RQ1) and how security tactics are constantly evaluated (RQ2).

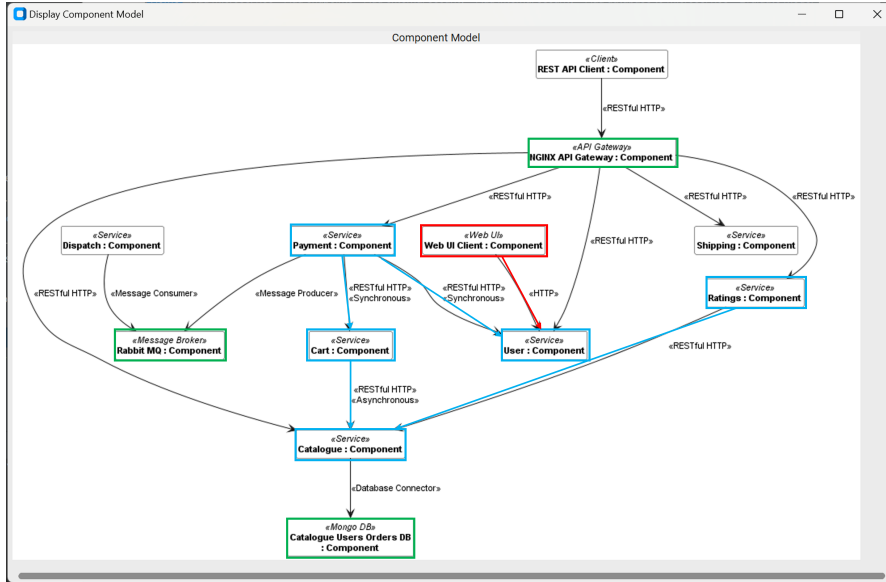
6.2. System validation

Validating the system is essential in the development process. After unsuccessfully contacting suitable developers to perform an empirical evaluation, it appears that it is too costly and time-consuming for this frame. Hence, the DevBot application is tested with existing microservice models. To define a ground truth, I search for microservices of different sizes and complexity to test the DevBot with a broad range of microservices. During the search process, only two studies [69, 67] come into consideration. After the search, the annotated microservice models by Zdun et al. [67] are selected, since the models are already defined with codeable models and are therefore compatible with the DevBot system. Security and software experts define these models, ensuring the identification of all security-relevant aspects. The DevBot is evaluated based on the security steps to be performed, the resulting security effort, the available support for security decisions, the costs for security, and the coverage of the available security tactics.

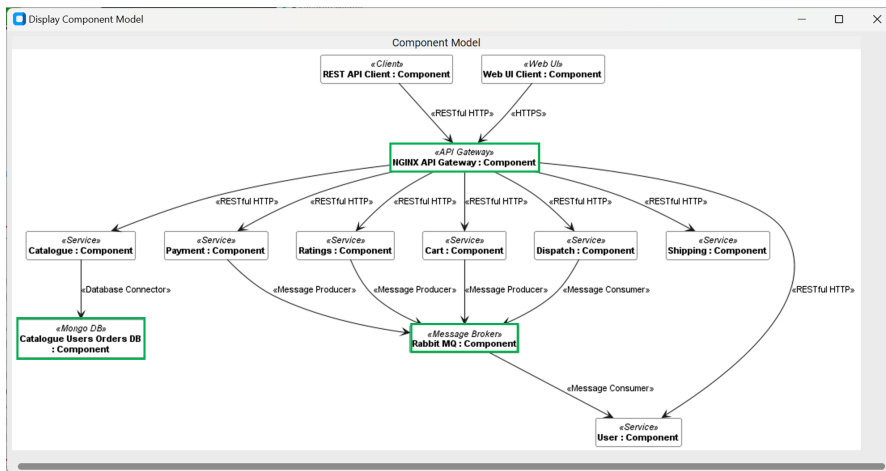
Five microservice systems annotated with codeable models published by Zdun et al. [67] are used for system validation. The microservices are open-source projects from GitHub. They are created manually and have different security properties as well as

⁵<https://github.com/instana/robot-shop>

6.2. System validation



(a) Inserted model before evaluation



(b) Secured model after evaluation

Figure 6.1.: Evaluation of system

6. Discussion

varying complexity. The first microservice (AC1⁶) deals with accounting management and consists of eight components with eleven connectors. The two medium-sized microservices (CI1⁷ and CO0⁸) represent a cinema booking system and a store management system and consist respectively of 12 or 13 components with 21 or 16 connections. The large microservices (PM2⁹ and RS0¹⁰) handle microservice metric collecting and a robot store application and consist of 17 components with 21 connectors and 19 components with 30 connectors.

Since gray literature research has shown that microservice security is often neglected, it is assumed that no security patterns are used in the annotated microservices and that checking applied microservice security manually is time-intensive. To validate this assumption, the sample systems are evaluated with the DevBot system, while also measuring time.

For the validation of the DevBot system, the annotated example systems are loaded into the application and the security risks are checked. Table 6.1 shows the evaluated risks.

It has been shown that all systems except CO0 have implemented an external security pattern, but external endpoints still communicate with microservices directly. The security level is low or very low for all systems except CI1, where all microservices are only accessible via an API gateway. In general, all example systems use an API gateway as a security pattern but do not apply it properly.

Table 6.1.: Validation of DevBot system with annotated models

	AC1	CI1	CO0	PM2	RS0
External security pattern	yes	yes	no	yes	yes
Pattern name	API gateway	API gateway	-	API gateway	API gateway
Security level	low	very high	very low	low	very low
Internal security pattern	no	no	no	no	yes
Pattern name	-	-	-	-	Message Queue
Security level	medium	medium	very high	very high	low

Regarding the internal microservice communication, no security patterns are used. Only RS0 uses a message queue pattern, although not all microservices communicate via this pattern. CO0 and PM2 have a very high level of security despite not using an

⁶<https://github.com/piomin/sample-spring-microservices-new>

⁷<https://github.com/Crizstian/cinema-microservice>

⁸<https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest>

⁹<https://github.com/sqshq/piggymetrics>

¹⁰<https://github.com/instana/robot-shop>

internal security pattern, since the microservices do not communicate with each other. The evaluation steps of the DevBot system for each annotated system can be found in section A.3.

This analysis shows that the assumption that no security patterns are used is only partially true since API gateways as external security patterns are used. However, considering the security level for external security, not all microservices are secured correctly. Regarding internal security, there is still a need for implementing internal security patterns, since only one example system is using a security pattern. However, the lack of internal security is not significant because the microservices in CO0 and PM2 do not communicate with each other, which ensures a high-security level.

In addition to the DevBot system validation with example systems, the DevBot approach has been compared with a manual evaluation approach using the example models. Table 6.2 shows the main comparison criteria.

Considering that the DevBot system is intended to reduce the effort required to apply microservice security, we first compare the development and maintenance time, the security steps, and the resulting time required. Since the development of the DevBot system is time-consuming, the overall spend time for the development is about eight weeks. The DevBot also has to be continuously kept up to date, which requires additional time to be spent. While the development process of the DevBot is time-consuming and costly, a manual approach does not require any development time. However, with a manual approach, the user must invest approximately two days to create a component model of the microservice and to include all security-relevant components. Therefore, the development of the DevBot is worthwhile for more frequent use.

Although the manual approach has the same amount of steps, the overall time spent is significantly longer, since the entire model must be reviewed and individual security patterns identified, while these steps are automated by the DevBot system. This results in the DevBot being much more time-efficient.

Furthermore, the DevBot supports the user in making security decisions, which eliminates the need for the user to acquire additional security knowledge. This also saves costs, since lacking or subsequently added security patterns are expensive [15]. There is no included security support for the manual approach.

Besides these criteria, a cost comparison is performed. The DevBot has a one-time development cost to evaluate all security mechanisms and add them to the DevBot system. There are also maintenance costs to keep the system up to date. With a manual approach, there are security research costs for each project, which increases the costs significantly over time.

Regarding the coverage of security tactics, the DevBot is estimated to cover only 80% of all security mechanisms, either because the system has not yet been updated or because of a researcher bias that does not know all security mechanisms. With a manual approach, the coverage is 100% because the developer can apply the most fitting security pattern when searching for the best microservice security for the project.

6. Discussion

Table 6.2.: Comparison of DevBot approach and manual approach for microservice architecture

Criterion	DevBot system	Manual evaluation
Development and maintenance time	8 weeks + continuous updating for developing the DevBot	No development time needed
Security steps	1. Load model 2. Evaluate and apply security analysis 3. Secure components if needed	1. Review model 2. Identify used security 3. Apply needed security
Security analysis effort	1 minute	2 days
Security support	Security support by decision tree	No support - research necessary
Costs	One-time cost plus continuous maintenance	Cost per research for each project
Security tactics coverage	Only approximately 80% coverage because of researcher bias	100% coverage because of manual research opportunity

Aside from the advantages of the DevBot system compared to a manual approach, the DevBot also has limitations. The DevBot is not always up to date in the area of microservice security, since newly developed security patterns must first be integrated. Additionally, there is a researcher bias, which prevents the DevBot from providing a complete overview of all possible microservice security tactics. As the DevBot focuses on microservice security, it does not take organizational specifications into account. This leads to a lack of support for the user in the implementation of the business logic. From a technical point of view, the DevBot system is currently a single-user tool. As a result, collaboration between different development teams is not possible.

In summary, it is shown that the DevBot system works with any annotated model and can quickly evaluate the model against the OWASP security risks, but there is still a need for improvement to reduce its limitations.

7. Conclusion

This thesis aims to develop an interactive tool for creating a component model for microservices, which accounts for selected microservice security aspects at the architectural level and continuously evaluates the level of security during the creation process. This is achieved by the implementation of the DevBot, which forms a starting point for security-oriented tools during microservice decomposition.

Generally, the application was developed in four steps (see Figure 4.1). First, an overview of existing tools in the area of microservice security was conducted. It was found that security aspects are strongly neglected during microservice decomposition. This is addressed by the DevBot system, which supports the consideration of security aspects during microservice decomposition. For this purpose, the bot starts at an architectural level and supports the user in the creation of a secure microservice component model.

It offers simple and intuitive support for developers without modeling knowledge, which enables a rapid generation of component models. Since the DevBot is based on a security decision tree, it is ensured that security risks are considered. Furthermore, the DevBot application is also easily extensible for new security patterns.

It also enables end-to-end traceability of the design decisions, which makes it easier for other developers to reproduce projects. With the development of the DevBot, security is now brought into the focus of microservice decomposition, as the DevBot system offers a simple guide to take security mechanisms into account right from the start.

7.1. Threats to validity

One of the biggest threats to validity is researcher bias. Researcher bias limits DevBot's ability to provide a complete overview of microservice security tactics. This in turn is a threat to the validation of the system, since existing expectations of the researcher subconsciously select models where the expected outcome is more likely. Random selection by independent groups would counteract this threat.

In addition, the sample with which the DevBot is tested is relatively small and therefore not as meaningful. This can lead to possible constellations of microservices being forgotten to be tested. To avoid this, the sample size must be increased for future tests.

7. Conclusion

7.2. Learnings and Limitation

During the development process of the DevBot system, I experienced several challenges from which I could draw further knowledge. After the multivocal literature research was conducted and the limitations of microservice security emerged, the first idea was to develop a bot involving microservice decomposition tools. However, it turned out that many developed tools are not further maintained in the academic area. Therefore, the concept of the DevBot had to be adapted to the circumstances and new ideas had to be developed. During the development of the DevBot system, I learned to deal with unplanned changes and to adapt the concept continuously. The development of the DevBot was iterative with feedback from developers and continuously adapted. This resulted in an intuitive GUI.

Aside from the learnings during the development of the DevBot system, the DevBot also has limitations. The DevBot may not always have the latest microservice security patterns, due to the need for integration. The DevBot system must be updated regularly to take into account new security patterns. Furthermore, researcher bias limits the ability of the DevBot to present a comprehensive overview of all available microservice security tactics. The DevBot does not consider organizational specifications in microservice security, resulting in insufficient user support for business logic implementation. Since the DevBot system currently only allows for single-user access, collaboration between development teams is prevented.

7.3. Future work

Since this DevBot system is a prototype for a security-oriented modeling tool for microservice architecture, there is still a need for research in this area. Generally, the development of a standardized database for security patterns can be a huge improvement for microservice security. Ideally, such a database is open source so that it can be extended in cooperation with other security experts.

In addition, the legal situation and data protection guidelines of the countries must be taken into account. Implementing a functionality to enable the DevBot to adapt to the respective organization and the legal framework, would be a big improvement for future prototypes.

Regarding the GUI, it can be improved by a web framework such as Flask, Django, or React. In this context, the GUI can be designed to select a security pattern from the open-source database. This includes a general update of the GUI since right now the path for saving and loading models is hard-coded. The option to select the path for saving the model and loading the model from any location can further improve the usability of the application.

In general, conducting a controlled experiment for an empirical evaluation of a manual approach and the tool-based approach for security in microservice decomposition can provide a better insight into missing functionalities of the DevBot system to enhance the prototype.

Bibliography

- [1] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30. IEEE, 2017.
- [2] J. Ransom, I. Somerville, and I. Warren. A method for assessing legacy systems for evolution. In *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 128–134. IEEE Comput. Soc, 1998.
- [3] Leonardo P. Tizzei, Marcelo Nery, Vinícius C. V. B. Segura, and Renato F. G. Cerqueira. Using microservices and software product line engineering to support reuse of evolving multi-tenant SaaS. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, pages 205–214. ACM, 2017.
- [4] Genc Mazlami, Jurgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531. IEEE, 2017.
- [5] Martin Fowler and James Lewis. Microservices. <https://www.martinfowler.com/articles/microservices.html>. last accessed 07-July-2023.
- [6] Anelis Pereira-Vale, Eduardo B. Fernandez, Raúl Monge, Hernán Astudillo, and Gastón Márquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200, 2021.
- [7] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 1 edition, 2003.
- [8] Ana Santos and Hugo Paula. Microservice decomposition and evaluation using dependency graph and silhouette coefficient. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 51–60. ACM, 2021.
- [9] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing*, volume 9846, pages 185–200. Springer International Publishing, 2016. Series Title: Lecture Notes in Computer Science.
- [10] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. *Tool support for the migration to microservice architecture: An industrial case study*, volume 11681 of *Lecture Notes in Computer Science*. Springer International Publishing, 2019.

Bibliography

- [11] Abdelhakim Hannousse and Salima Yahiouche. Securing microservices and microservice architectures: A systematic mapping study. *Computer Science Review*, 41:100415, 2021.
- [12] Davide Berardi, Saverio Giallorenzo, Jacopo Mauro, Andrea Melis, Fabrizio Montesi, and Marco Prandini. Microservice security: a systematic literature review. *PeerJ Computer Science*, 7:e779, 2022.
- [13] Carly Page. Uber investigating cybersecurity incident after hacker breaches its internal network. <https://techcrunch.com/2022/09/16/uber-internal-network-hack/>. last accessed 07-July-2023.
- [14] Dara Khosrowshahi. 2016 data security incident. <https://www.uber.com/newsroom/2016-data-incident/>. last accessed 07-July-2023.
- [15] Nuno Mateus-Coelho, Manuela Cruz-Cunha, and Luis Gonzaga Ferreira. Security in microservices architectures. *Procedia Computer Science*, 181:1225–1236, 2021.
- [16] IBM Cloud Team. What is soa (service-oriented architecture)? <https://www.ibm.com/topics/soa>. last accessed 14-July-2023.
- [17] Grace A. Lewis and Dennis B. Smith. Proceedings of the international workshop on the foundations of service-oriented architecture. In *Proceedings of the International Workshop on the Foundations of Service-Oriented Architecture*, 2008.
- [18] IBM Cloud Team. Soa vs. microservices: What’s the difference? <https://www.ibm.com/cloud/blog/soa-vs-microservices>. last accessed 14-July-2023.
- [19] Stephan Bartholome. microservices vs monolith - sekt oder selters? <https://itbalance.de/mvsm-2020-09-17-10/>, 2020. last accessed 05-october-2023.
- [20] Sam Newman. *Building microservices: designing fine-grained systems*. O’Reilly Media, first edition edition, 2015. OCLC: ocn881657228.
- [21] Tim Bastin. Splitting monolithic systems into microservices. <https://medium.com/13montree-techblog/splitting-monolithic-systems-into-microservices-84f9c9de5f16>. last accessed 07-July-2023.
- [22] Sam Newman and Leandro Guimarães. Migrating monoliths to microservices with decomposition and incremental changes. <https://www.infoq.com/articles/migrating-monoliths-to-microservices-with-decomposition/>. last accessed 07-July-2023.
- [23] Erich Gamma, Richard Helm, Ralph Johanson, and John Vlissides. *Design Patterns Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp, 1. edition edition, 2015.

- [24] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study:. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, pages 221–232. SCITEPRESS - Science and Technology Publications, 2018.
- [25] Chris Richardson. A pattern language for microservices. <https://microservices.io/patterns/index.html>. last accessed 19-July-2023.
- [26] Manhattan Tech Support. The seven layers of it security. <https://www.manhattan-techsupport.com/blog/the-seven-layers-of-it-security/>, 2021-01-21. last accessed 27-August-2023.
- [27] OWASP. Attack surface analysis - owasp cheat sheet series. https://cheatsheetseries.owasp.org/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.html, 2021. last accessed 27-August-2023.
- [28] Ramaswamy Chandramouli and Zack Butcher. Building secure microservices-based applications using service-mesh architecture. *National Institute of Standards and Technology*, 2020.
- [29] Alexander S. Gillis and James Montgomery. Service mesh. <https://www.techtarget.com/searchitoperations/definition/service-mesh>. last accessed 07-July-2023.
- [30] Fernando Chanaka. How to implement security for microservices. <https://medium.com/microservices-learning/how-to-implement-security-for-microservices-89b140d3e555>. last accessed 07-July-2023.
- [31] Joydip Kanjilal. The role of sidecars in microservices architecture. <https://www.techtarget.com/searchapparchitecture/tip/The-role-of-sidecars-in-microservices-architecture>, 2022. last accessed 05-october-2023.
- [32] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and api gateways in microservices. In *Circuit Breakers, Discovery, and API Gateways in Microservices*, 2016.
- [33] Leonid Bugaev. Why do microservices need an API gateway? <https://tyk.io/blog/microservices-api-gateway/>. last accessed 07-July-2023.
- [34] Mehmet Ozkaya. Api gateway pattern. <https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df>, 2021. last accessed 05-october-2023.
- [35] Mary Ann Richardson. Top 10 challenges of using microservices for managing distributed systems. <https://www.spiceworks.com/tech/data-management/articles/top-10-challenges-of-using-microservices-for-managing-distributed-systems/>. last accessed 07-July-2023.

Bibliography

- [36] Moses Bassey. Message queue - what it is, how it works, and when to use it. <https://medium.com/@mosesbasseyekwere/message-queue-what-it-is-how-it-works-and-when-to-use-it-2b7fb5dcef48>. last accessed 07-July-2023.
- [37] Larry Peng Yang. System design - message queues: Concepts and considerations for message queues in system design. <https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22>. last accessed 07-July-2023.
- [38] Korak Bhaduri. Message queue vs. publish-subscribe. <https://blog.iron.io/message-queue-vs-publish-subscribe/>. last accessed 07-July-2023.
- [39] Google Cloud. Ereignisgesteuerte architektur mit pub/sub. <https://cloud.google.com/solutions/event-driven-architecture-pubsub?hl=de>, 2023. last accessed 05-october-2023.
- [40] Işıl Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdogan. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180:111014, 2021.
- [41] Abhinav Vinci. Common problems in message queues. <https://medium.com/@vinciabhinav7/common-problems-in-message-queues-with-solutions-f0703c0bd5af>. last accessed 07-July-2023.
- [42] IBM. Publish/subscribe components. <https://www.ibm.com/docs/en/ibm-mq/9.2?topic=messaging-publishsubscribe-components>. last accessed 07-July-2023.
- [43] IBM. Publish/subscribe security. <https://www.ibm.com/docs/en/ibm-mq/9.2?topic=securing-publishsubscribe-security>. last accessed 07-July-2023.
- [44] Jay Ramachandran. *Designing Security Architecture Solutions*. John Wiley & Sons, Inc., 2020.
- [45] Marco Fioretti. OAuth 2.0: What is it, and how does it work? <https://www.techrepublic.com/article/oauth-2-0-what-is-it/>. last accessed 07-July-2023.
- [46] Frontegg. Microservices: Approaches and techniques. <https://frontegg.com/blog/authentication-in-microservices>. last accessed 07-July-2023.
- [47] Narendra Bandhamneni. Securing microservices with OAuth 2.0. <https://walkintreetech.com/securing-microservices-oauth2/>. last accessed 07-July-2023.
- [48] Aviad Mizrachi. OAuth grant types: Explained. <https://frontegg.com/blog/oauth-grant-types>. last accessed 07-July-2023.
- [49] Diego Pereira da Rocha. Monolise: Uma tecnica para decomposicao de aplicacoes monoliticas em microservicos. *Universidade do Vale do Rio dos Sinos*, page 175, 2018.

- [50] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *Microservice Decomposition via Static and Dynamic Analysis of the Monolith*. arXiv, 2020. Number: arXiv:2003.02603.
- [51] Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, Wesley K. G. Assuncao, Juliana Alves Pereira, Baldoino Fonseca, Marcio Ribeiro, Maria Julia de Lima, and Carlos Lucena. On the performance and adoption of search-based microservice identification with toMicroservices. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 569–580. IEEE, 2020.
- [52] Chia-Yu Li, Shang-Pin Ma, and Tsung-Wen Lu. Microservice migration using strangler fig pattern: A case study on the green button system. In *2020 International Computer Symposium (ICS)*, pages 519–524. IEEE, 2020.
- [53] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.
- [54] Safa Habibullah, Xiaodong Liu, Zhiyuan Tan, Yonghong Zhang, and Qi Liu. Reviving legacy enterprise systems with micro service-based architecture with in cloud environments. In *8th International Conference on Soft Computing, Artificial Intelligence and Applications*, pages 173–186. Aircc Publishing Corporation, 2019.
- [55] David Wobrock. To kill a microservice. <https://medium.com/botify-labs/to-kill-a-microservice-d6c9e7ad444c>. last accessed 07-July-2023.
- [56] Sha Ma and Ben Linders. GitHub’s journey from monolith to microservices. <https://www.infoq.com/articles/github-monolith-microservices/>. last accessed 07-July-2023.
- [57] Sam Jarman. Lessons learned from multiple microservice transitions. <https://www.samjarman.co.nz/blog/microservice-lessons>. last accessed 07-July-2023.
- [58] Maximilian Schoefmann. Security challenges in microservice implementations. <https://blog.container-solutions.com/security-challenges-in-microservice-implementations>. last accessed 07-July-2023.
- [59] Atul Chaturvedi and Limor Kessem. Securing the microservices architecture: Decomposing the monolith without compromising security. <https://securityintelligence.com/securing-the-microservices-architecture-decomposing-the-monolith-without-compromising-information-security/>, 2019-03-21. last accessed 07-August-2023.
- [60] Horacio Duran. Building a solid foundation for microservices security. <https://devops.com/building-a-solid-foundation-for-microservices-security/>, 2018. last accessed 07-August-2023.

Bibliography

- [61] Marco Troisi. 8 best practices for microservices app sec. <https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec>. last accessed 07-August-2023.
- [62] Matt Raible. Security patterns for microservice architectures. <https://developer.okta.com/blog/2020/03/23/microservice-security-patterns>, 2020-03-23. last accessed 07-August-2023.
- [63] Oleksandr Gerasymov. Microservices security: Best practices and expert tips. <https://codeit.us/blog/microservices-security>, 2023-06-23. last accessed 07-August-2023.
- [64] Nichlas Bjørndal, Luiz Jonatã Pires de Araújo, Antonio Bucchiarone, Nicola Dragoni, Manuel Mazzara, and Schahram Dustdar. Benchmarks and performance metrics for assessing the migration to microservice-based architectures. *The Journal of Object Technology*, 2021.
- [65] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *Management Information Systems Research Center*, 28(1), 2004.
- [66] Philip. Design science research methodologie. https://shribe.de/design-science-research-methodologie/#Das_Design_Science_Research_Framework. last accessed 07-August-2023.
- [67] Uwe Zdun, Pierre-Jean Queval, Georg Simhandl, Riccardo Scandariato, Somik Chakravarty, Marjan Jelić, and Aleksandar Jovanović. Detection strategies for microservice security tactics. *IEEE Transactions on Dependable and Secure Computing*, pages 1–17, 2023.
- [68] Anusha Bambhore Tukaram, Simon Schneider, Nicolás E. Díaz Ferreyra, Georg Simhandl, Uwe Zdun, and Riccardo Scandariato. Towards a Security Benchmark for the Architectural Design of Microservice Applications. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–7, Vienna Austria, August 2022. ACM.
- [69] Simon Schneider, Tufan Özen, Michael Chen, and Riccardo Scandariato. micro-SecEnD: A Dataset of Security-Enriched Dataflow Diagrams for Microservice Applications. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 125–129, Melbourne, Australia, May 2023. IEEE.
- [70] Nacha Chondamrongkul, Jing Sun, and Ian Warren. Automated security analysis for microservice architecture. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 79–82. IEEE, 2020.
- [71] Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, 2018.

- [72] Erez Yallon, Inon Shkedy, and Paulo Silva. OWASP API security top ten - 2023. <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>. last accessed 07-July-2023.
- [73] L & Co Staff Auditors. Security maturity models: Levels, assessment, and benefits. <https://linfordco.com/blog/security-maturity-models/>, 2023. last accessed 10-september-2023.

Acronyms

API Application Programming Interface. 1, 7, 8, 15, 21, 30, 31, 33, 35

BFF Backends for Frontends. 62

DDD Domain-driven Design. 1

ESB Enterprise Service Bus. 5, 6

GUI Graphical User Interface. 37–39, 41–43, 45, 51, 54, 63

HTML Hypertext Markup Language. 6

HTTP Hypertext Transfer Protocol. 6, 7, 11

IAM Identity and Access Management. 14, 26

LDA Latent Dirichlet Allocation. 19

mTLS mutual Transport Layer Security. 10

oAuth Open Authentication. 14, 15, 22

ODP Outcome Decision Point. 33

OWASP Open Web Application Security Project. 30, 33, 35

PDP Procedural Decision Point. 33

PKCE Proof Key for Code Exchange. 15

PKI Public Key Infrastructure. 30

SCP Source Code Package. 19

SOA Service-oriented Architecture. 5, 6

SSRF Server-Side Request Forgery. 31

UML Unified Modeling Language. 3

URI Uniform Resource Identifier. 31

A. Appendix

A.1. Model extension

In this section, the code snippets for the extensibility of the DevBot system can be found.

```
1 def update_external_security():
2     .
3     .
4     .
5     elif sec_type == backends_for_frontends_gateway:
6         if len(updated_endpoints) > 0:
7             for i in updated_endpoints:
8                 name = getattr(i, "name") + "_bff"
9                 new_component = CClass(component, name,
10 stereotype_instances=[sec_type])
11                 clink.add_links({i: new_component}, role_name="target",
12                                 stereotype_instances=[restful_http])
13             if len(updated_microservices) > 0:
14                 for i in sec_services:
15                     for j in updated_microservices:
16                         clink.add_links({i: j}, role_name="target",
17                                         stereotype_instances=[https])
18         .
19
20 def change_external_security(new_security):
21     .
22     .
23     .
24     elif new_security == "backends_for_frontends_gateway":
25         add_BFF()
```

Listing A.1: Function to update or change external security

```
1 def get_security_services(model):
2     .
3     .
4     if j == api_gateway or j == sidecar or j ==
5         individual_internal_security or j == backends_for_frontends_gateway:
```

Listing A.2: Modify if statements in supporting function

```
1 options = ["Api Gateway", "Sidecar", "Individual Security", "Backends for
    Frontends Gateway"]
```

A. Appendix

```
2 def create_external_security(security_type):
3     .
4     .
5     elif security_type == "backends_for_frontends_gateway":
6         add_BFF()
7         generate_model_image()
8         evaluate_security_model()
9         external_sec.set("BFFs have been added.")
```

Listing A.3: Modify gui functionality to use extension in DevBot gui

A.2. Conformance checking

In this section, the code snippets for the evaluation and risk calculation of the DevBot system can be found.

```
1 def check_microservice_connections():
2     model = get_all_components()
3     microservices = get_all_microservices(model)
4     endpoints = get_all_endpoints(model)
5     not_connected_microservices = []
6     not_secured_microservices = []
7     not_secured_communication = []
8     for i in microservices:
9         links = i.get_linked()
10        if not links:
11            not_connected_microservices.append(i) # microservice not
connected to security pattern
12        for j in links:
13            if j in endpoints:
14                not_secured_microservices.append(i) # microservice
directly connected to endpoint
15                if j in microservices:
16                    not_secured_communication.append(i) # direct inter-
service communication
17    return not_connected_microservices, not_secured_microservices,
not_secured_communication
```

Listing A.4: Identification of vulnerable microservices

```
1 def check_endpoint_connections():
2     model = get_all_components()
3     endpoints = get_all_endpoints(model)
4     microservices = get_all_microservices(model)
5     not_connected_endpoints = []
6     not_secured_endpoints = []
7     for i in endpoints:
8         links = i.get_linked()
9         if not links:
10            not_connected_endpoints.append(i)
11        for j in links:
```

```

12         if j in microservices:
13             not_secured_endpoints.append(i)
14     return not_connected_endpoints, not_secured_endpoints

```

Listing A.5: Identification of vulnerable endpoints

```

1 def calculate_security_risk(endpoint, microservice, external, internal):
2     if external != 1:
3         risk = external
4         risk_type = "external"
5     elif internal != 1:
6         risk = internal
7         risk_type = "internal"
8     elif microservice != 1:
9         risk = microservice
10        risk_type = "microservice"
11    elif endpoint != 1:
12        risk = endpoint
13        risk_type = "endpoint"
14    else:
15        risk = 1
16        risk_type = ""
17    return risk, risk_type
18
19 def evaluate_model():
20     ...
21     if len(not_connected_endpoints) == 0:
22         endpoints_rating = 1
23     elif len(not_connected_endpoints) == 1 or len(not_connected_endpoints
24 ) == 2:
25         endpoints_rating = 0.9
26     elif len(not_connected_endpoints) == 3:
27         endpoints_rating = 0.8
28     else:
29         endpoints_rating = 0.7
30
31     if len(not_connected_microservices) == 0:
32         microservice_rating = 1
33     elif len(not_connected_microservices) == 1 or len(
34 not_connected_microservices) == 2:
35         microservice_rating = 0.8
36     elif len(not_connected_microservices) == 3:
37         microservice_rating = 0.7
38     else:
39         microservice_rating = 0.6
40
41     if len(not_secured_microservices) == 0:
42         external_security = 1
43     elif len(not_secured_microservices) == 1 or len(
44 not_secured_microservices) == 2:
45         external_security = 0.3
46     elif len(not_secured_microservices) == 3:
47         external_security = 0.2
48     else:

```

A. Appendix

```
46     external_security = 0.1
47
48     if len(not_secured_communication) == 0:
49         internal_security = 1
50     elif len(not_secured_communication)/2 == 1 or len(
not_secured_communication)/2 == 2:
51         internal_security = 0.5
52     elif len(not_secured_communication)/2 == 3:
53         internal_security = 0.4
54     else:
55         internal_security = 0.3
56     security_risk = calculate_security_risk(endpoints_rating,
microservice_rating, external_security, internal_security)
57     return security_risk
```

Listing A.6: Calculation of overall security level

```
1 def ensure_external_security():
2     not_secured_endpoints = check_endpoint_connections()[1]
3     not_secured_microservices = check_microservice_connections()[1]
4     links = codeable_models.clink.get_links(not_secured_microservices)
5     deleted_links = []
6     for i in links:
7         if i.__getattribute__("source") in not_secured_endpoints or i.
__getattribute__("target") in not_secured_endpoints:
8             if i in deleted_links: # check if link already deleted
9                 pass
10            else:
11                endpoint = i.__getattribute__("source")
12                microservice = i.__getattribute__("target")
13                deleted_links.append(i)
14                codeable_models.clink.delete_links({endpoint:
microservice}) # delete direct connection
15    return update_external_security()
```

Listing A.7: Update of external security

```
1 def ensure_inter_service_security():
2     ...
3     not_secured_communication = check_microservice_connections()[2]
4     comm_links = codeable_models.clink
5         .get_links(not_secured_communication)
6     deleted_links = []
7     .
8     .
9     for i in comm_links:
10        # check to connect microservices only once via security pattern
11        if i.__getattribute__("source") in microservices and
12            i.__getattribute__("target") in microservices:
13            if i in deleted_links:
14                pass
15            else:
16                source = i.__getattribute__("source")
17                target = i.__getattribute__("target")
```

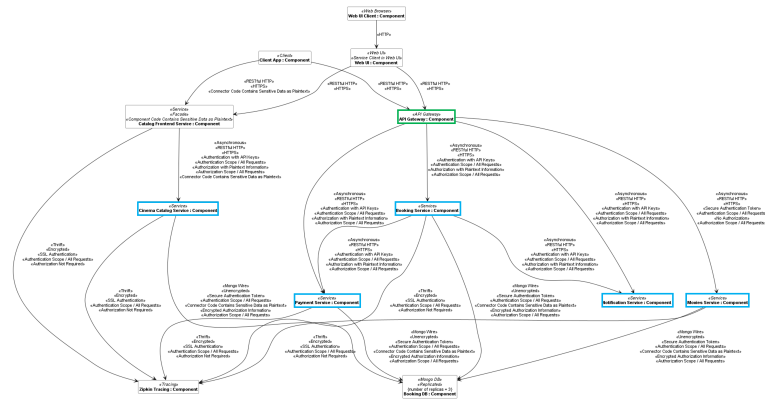
```
18         codeable_models.clink.delete_links({source: target})
19         deleted_links.append(i)
20         target_links = get_target_links(comm_pattern)
21         source_links = get_source_links(comm_pattern)
22         if source not in source_links:
23             codeable_models.clink.add_links({source: comm_pattern
24 }, role_name="target", stereotype_instances=[pub_type])
25             if target not in target_links:
26                 codeable_models.clink.add_links({comm_pattern: target
27 }, role_name="target", stereotype_instances=[sub_type])
28         return True
29     else:
30         return False
```

Listing A.8: Update of internal security

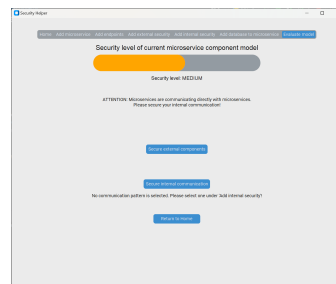
A. Appendix

A.3. Model validation

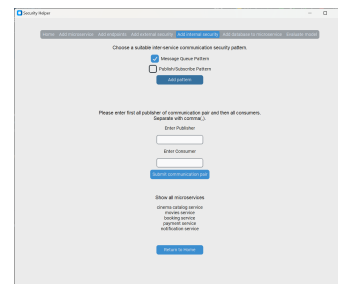
In this section, the evaluation of the annotated systems by Zdun et al. [67] can be found. The first image of the model is always the original annotated model, modified only by color highlighting.



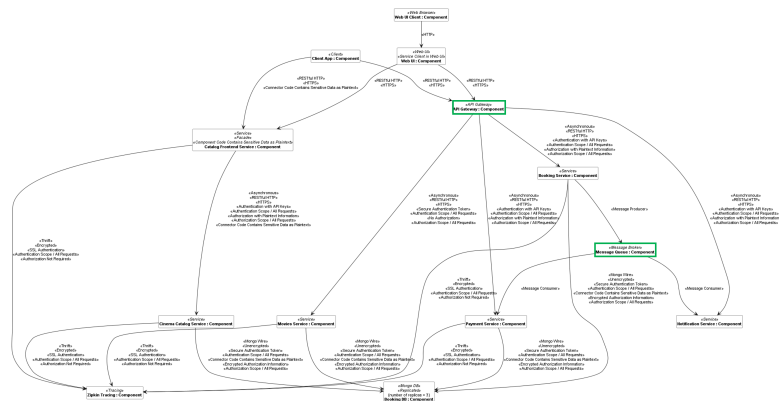
(a) CI1 model without evaluation



(b) Risk evaluation



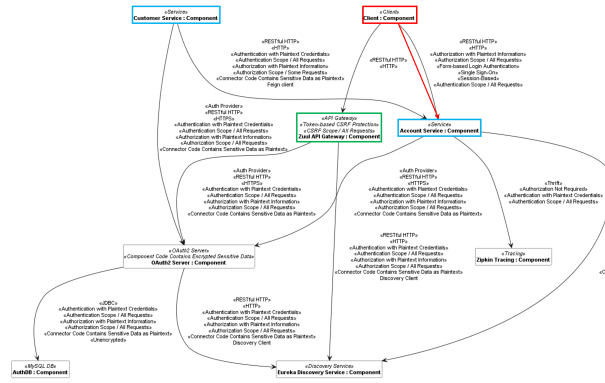
(c) Add security pattern



(d) CI1 model with all security

Figure A.1.: Evaluation of CI1

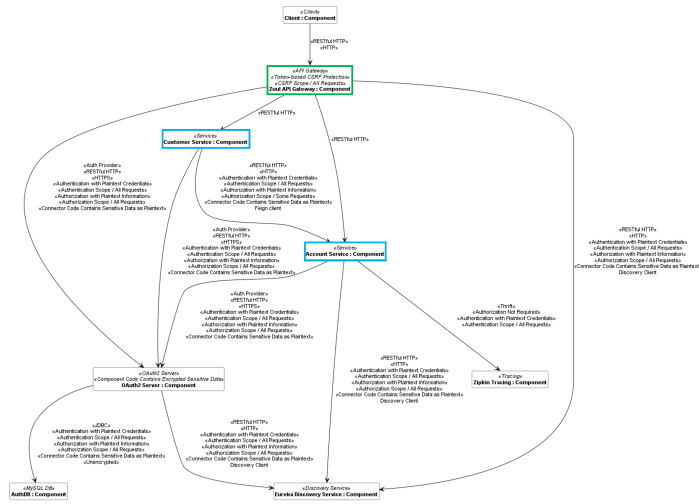
A.3. Model validation



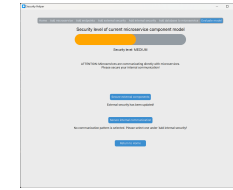
(a) AC1 model without evaluation



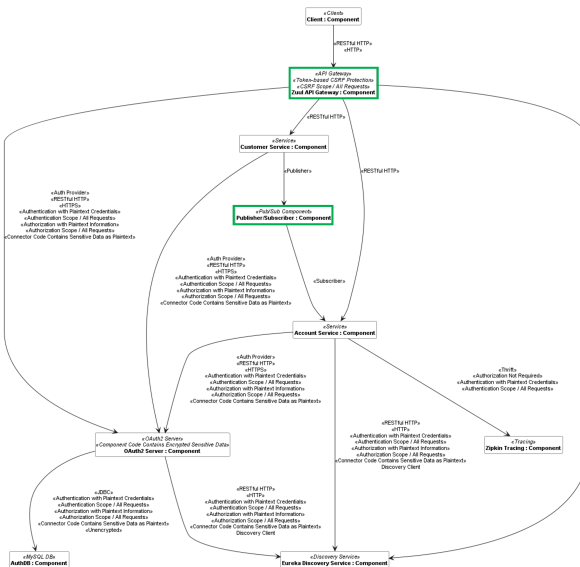
(b) Risk evaluation



(c) AC1 model with external security



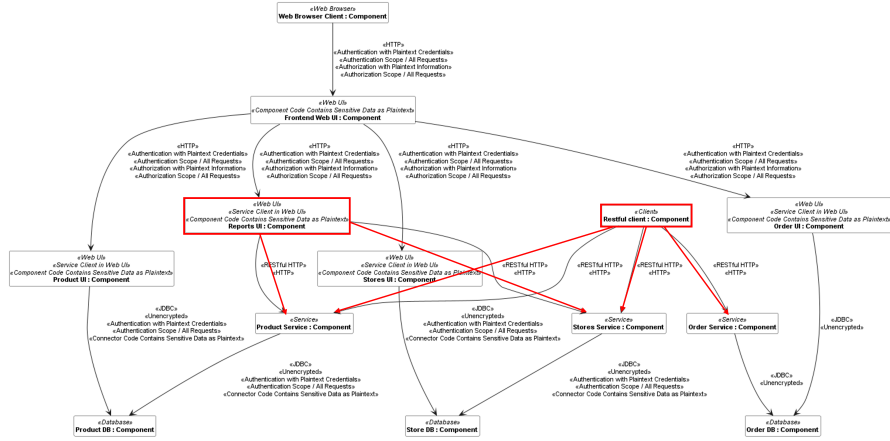
(d) Risk evaluation



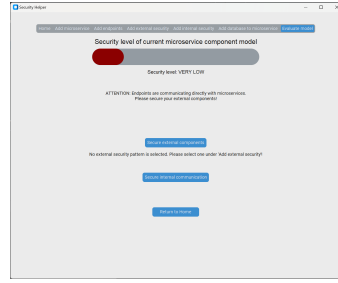
(e) AC1 model with internal security

Figure A.2.: Evaluation of AC1

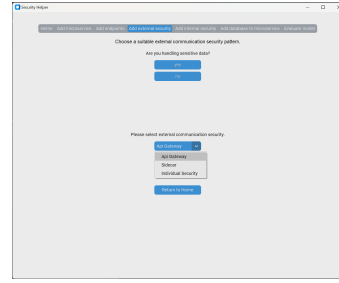
A. Appendix



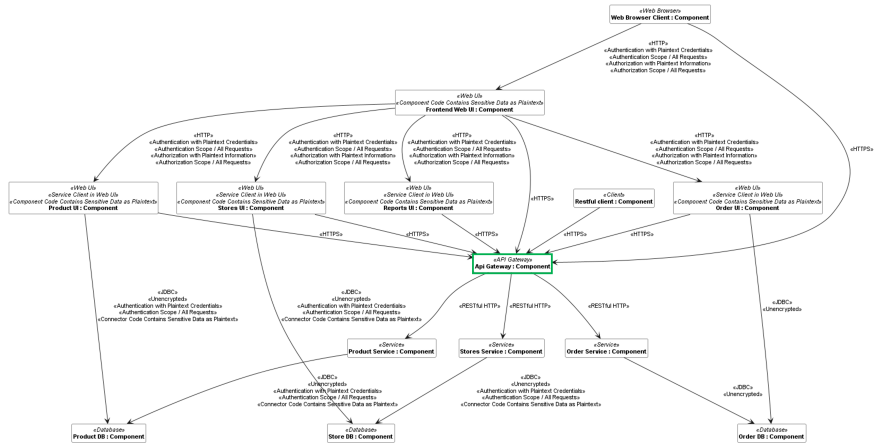
(a) CO0 model without evaluation



(b) Risk evaluation



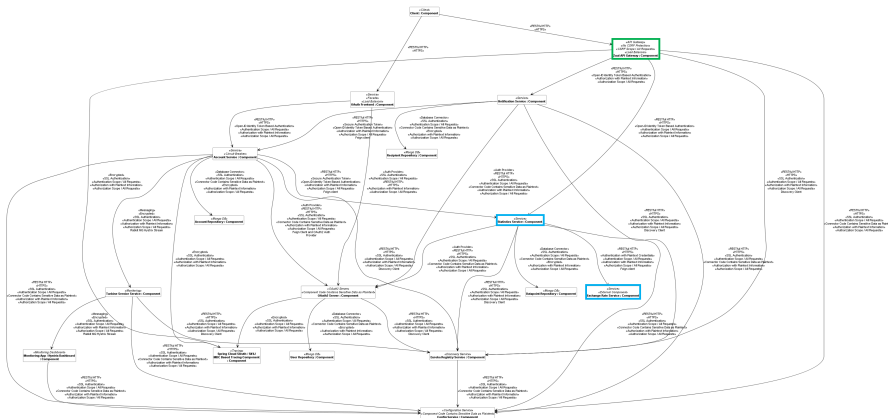
(c) Add security pattern



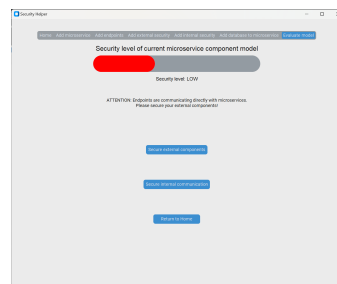
(d) CO0 model with all security

Figure A.3.: Evaluation of CO0

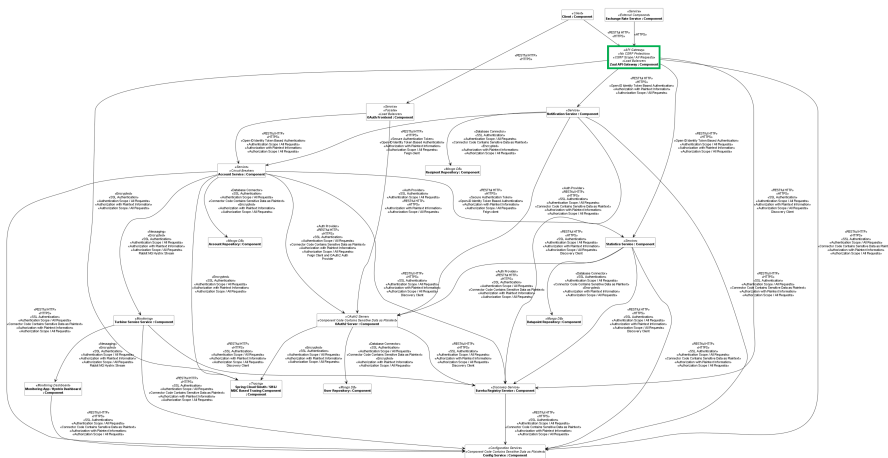
A.3. Model validation



(a) PM2 model without evaluation



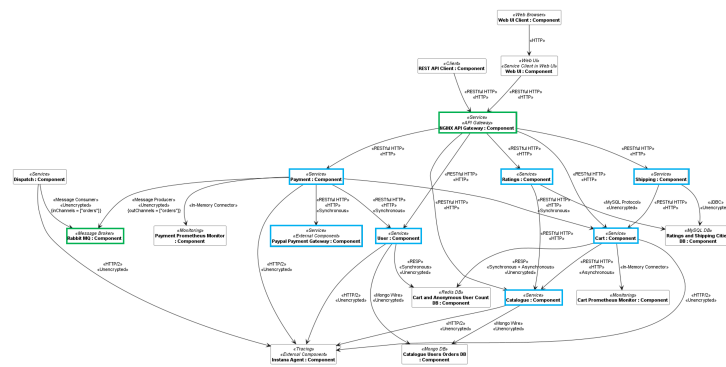
(b) Risk evaluation



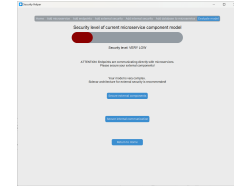
(c) PM2 model with all security

Figure A.4.: Evaluation of PM2

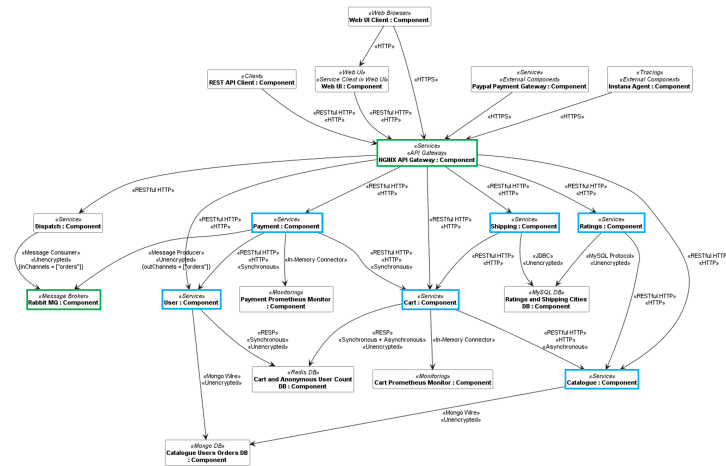
A. Appendix



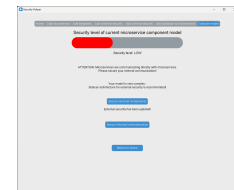
(a) RS0 model without evaluation



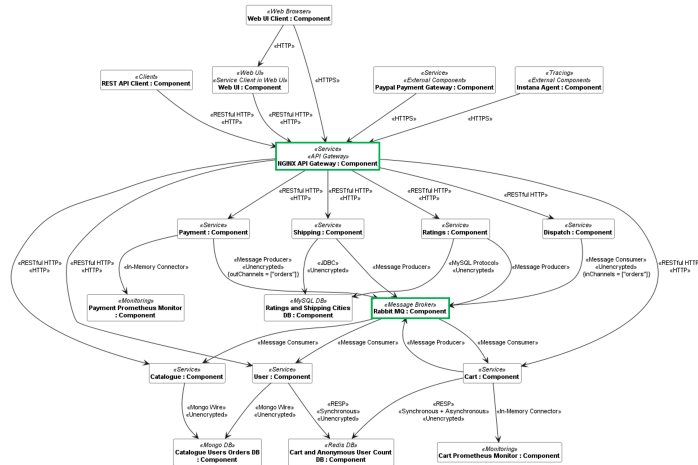
(b) Risk evaluation



(c) RS0 model with external security



(d) Risk evaluation



(e) RS0 model with internal security

Figure A.5.: Evaluation of RS0