MODELING MICROSERVICE BASED APPLICATIONS: MODEL LIVES INSIDE CODE APPROACH

A Thesis Submitted to
the Graduate School of Engineering and Sciences
of İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Eyüp Fatih Ersoy

July 2024
İZMİR

We approve the thesis of **Eyüp Fatih ERSOY**

Examining Committee Members:

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to those who have provided invaluable support and guidance throughout the journey of completing this thesis. First and foremost, I am profoundly grateful to my supervisor, Prof. Dr. Onur Demirörs. His expertise, wisdom, and unwavering support have been instrumental in guiding me through the complexities of this research. His constructive feedback, insightful suggestions, and encouragement have significantly shaped the direction and quality of my work.

I would also like to thank the faculty and staff at the Graduate School of Engineering and Sciences of Izmir Institute of Technology. Their dedication to providing a conducive learning and research environment has greatly contributed to the successful completion of this thesis.

My heartfelt thanks go to my family and friends, whose love, patience, and encouragement have been my constant source of strength. I am also grateful to my fellow students and colleagues, who have provided a collaborative and stimulating atmosphere.

To everyone who has contributed to this journey, in ways big and small, thank you. Your support has been invaluable, and I am forever grateful.

ABSTRACT

MODELING MICROSERVICE BASED APPLICATIONS: MODEL LIVES INSIDE CODE APPROACH

In today's software development, maintaining consistent documentation is crucial for sharing and preserving team knowledge. As projects grow more complex, developers need to quickly understand and maintain code. However, keeping documentation aligned with business logic without unnecessary technical details is challenging.

Traditional visualization tools like UML, sequence, and activity diagrams focus on object-oriented approaches and often require manual updates, making them less suitable for event-based systems like microservices.

To address these issues, the tool Docupyt was developed using eEPC (Extended Event Process Chains) as the main modeling approach. Docupyt is designed with three key principles: ease of use, simplicity (including only necessary logic), and reactivity (representing event-based systems). eEPC notation helps analyze problems and represent changing logic during development, accommodating fast-changing requirements. It supports both high and low-level process definitions and focuses on business logic without extraneous technical details.

Generated directly from code through simple commenting, this approach simplifies updating documentation as the code changes, reducing maintenance costs. Using the design science research method, Docupyt was validated in a case study, demonstrating it is user-friendly and provides adequate detail without being overly technical. Its main advantage is keeping documentation in sync with code logic, easing updates.

ÖZET

MİKROSERVİS TABANLI UYGULAMALARIN MODELLENMESİ: MODELİN KOD İÇİNDE YAŞADIĞI YAKLAŞIM

Günümüz yazılım geliştirme süreçlerinde, takım içindeki bilginin korunması için dokümantasyona sahip olmak kritik öneme sahiptir. Projeler karmaşıklaştıkça, geliştiricilerin kodu hızla anlaması ve bakımını yapması gerekmektedir. Ancak, belgelerin iş mantığına uygun ve gereksiz teknik detaylar içermeyecek şekilde tutulması zordur.

UML, ardıl etkileşim ve aktivite diyagramları gibi geleneksel görselleştirme araçları, nesne yönelimli yaklaşımlara odaklanır ve genellikle manuel güncellemeler gerektirir, bu da onları mikroservisler gibi olay-tabanlı sistemler için daha az uygun hale getirir.

Bu sorunları çözmek için, ana modelleme yaklaşımı olarak eEPC'yi (Extended Event Process Chains) kullanan Docupyt aracı geliştirilmiştir. Docupyt, kullanım kolaylığı, sadelik (sadece gerekli mantığı içeren) ve reaktiflik (olay tabanlı sistemleri temsil etme) olmak üzere üç temel ilke ile tasarlanmıştır. eEPC notasyonu, sorunları analiz etmeye ve değişen mantığı geliştirme sürecinde temsil etmeye yardımcı olur, hızlı değişen gereksinimlere uyum sağlar.

Dokümantasyonu doğrudan koddan üreten bu yaklaşım, kod değiştikçe belgeleri güncellemeyi kolaylaştırarak bakım maliyetlerini azaltır. Tasarım bilim araştırma yöntemi kullanılarak geliştirilen Docupyt, bir vaka çalışmasında doğrulanmıştır. Ana avantajı, belgeleri kod mantığıyla uyumlu tutarak güncellemeleri kolaylaştırmasıdır.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v

ABSTRACT iv

LIST OF FIGURES viii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. RELATED WORKS & CURRENT TOOLS 5

CHAPTER 3. EVENT-BASED SYSTEM MODELING WITH eEPC 11

3.1. eEpc Notation on Event-Based Systems 11

3.2. Abstract Syntax Tree 13

3.3. Tokenization 14

3.4. Graphviz 15

CHAPTER 4. PROPOSED METHOD 16

4.1. Research Methodology 16

4.2. Research Methodology – Case Study 19

4.2. Development of Software Tool: Docupyt 23

4.2.1. Initial Iteration of Development 23

4.2.2. Development in Iterations 24

4.3. Tokens & Usage 31

4.3.1. Defining Process Components 32

4.3.2. Defining Process Properties 35

4.3.3. Defining Architectural Event Relationship 37

CHAPTER 5. CASE STUDY 39

CHAPTER 6. CONCLUSIONS & FUTURE WORK... 44

REFERENCES... 47

LIST OF FIGURES

Figure **Page**

Figure 1. Example Abstract Syntax Tree 13

Figure 2. Research Methodology 18

Figure 3. Lambda-1 eEPC before development 20

Figure 4. Lambda-1 eEPC after development 21

Figure 5. Lambda-2 eEPC before development 22

Figure 6. Lambda-2 eEPC after development 22

Figure 7. Docupyt Structure on Initial Iteration 23

Figure 8. Docupyt Structure 24

Figure 9. Sample source code block documented using Docupyt tokens 26

Figure 10. Docupyt output at the end of development iteration 1 27

Figure 11. An example output of Docupyt at iteration 2 29

Figure 12. Architectural view of Docupyt 30

Figure 13. Activity Representation 32

Figure 14. Event Representation 33

Figure 15. Sub-diagram Representation 34

Figure 16. Database Connection Representation 35

Figure 17. External Call Representation 36

Figure 18. Architectural Representation 38

Figure 19. System Diagram 40

Figure 20. Order Service Diagram 40

Figure 21. Fraud Service 41

Figure 22. Email Service 42

Figure 23. Payments Service 42

Figure 24. General Architecture Diagram 43

CHAPTER 1

INTRODUCTION

Software documentation is indispensable in software engineering due to several reasons. Primarily, it serves as a repository of essential information pertaining to the system's architecture, design decisions, and implementation details. This comprehensive reference aids developers in understanding the intricacies of the software, facilitating efficient development, debugging, and maintenance processes. Moreover, documentation fosters collaboration among team members by providing clear instructions, guidelines, and standards, thereby ensuring consistency in coding practices and promoting cohesive teamwork. Additionally, it plays a pivotal role in knowledge transfer, enabling seamless onboarding of new team members and preserving institutional knowledge, thus ensuring the long-term sustainability and scalability of software systems.

In the contemporary realm of software development, the necessity for consistent documentation has become paramount. As software projects grow in complexity, developers face the challenge of comprehending and maintaining code efficiently. However, maintaining documentation that accurately represents the business logic without unnecessary technical details presents a dilemma.

Effort is required to manage and maintain documentation. Having balance in representing business logic without overwhelming technical intricacies is challenging. This difficulty extends beyond developers themselves, as extraneous technical details prove unhelpful for any participant involved. Streamlined and concise documentation is essential in modern software development. It not only aids understanding and collaboration within teams but also enhances the overall efficiency and success of software projects.

Documentation and modeling also extends beyond code to encompass various aspects such as user requirements, design specifications and test plans. Managing all these documents in a coherent and organized manner is as tedious as essential. User requirements documentation outlines the functionalities and features expected by stakeholders, serving as a blueprint for development efforts. Design specifications detail the architectural and technical decisions guiding the implementation process. Test plans document the strategies for verifying software quality and functionality, also it mostly includes the paths and possible flows to be tested. User manuals provide end-users with instructions for utilizing the software effectively. Ensuring the accuracy, accessibility, and consistency of these diverse documentation artifacts is essential for facilitating collaboration among team members, minimizing errors, and enhancing overall project clarity and efficiency. All those documents are managed by different teams in companies, using various tools and different notations. In this case, it also increases the difficulty of setting a standard inside the company for creating and maintaining the modeling of the systems.

Apart from setting standards and creating models and notations for the systems, it becomes a mess to maintain those during and after the development. Software systems have always been unsteady and changeful by their nature as it's almost impossible that the systems stay exactly the same as it's planned to be. This also means the change of the models and documents across the teams and departments inside the organization.

Traditional visualization tools, such as those that generate UML diagrams, sequence diagrams and activity diagrams, have been widely used in software development. These tools primarily focus on object-oriented approaches and are designed to support the visualization of static structures and interactions in software systems.

Initial software visualization tools have come out in an object-oriented atmosphere. They were mostly dependent on certain programming languages, mapping source code to object-oriented visuals. Shrimp Views was one of the pioneers, based on Java source code, it was able to demonstrate object-oriented relationships on models, it also included interactivity and a relation to the source code. Unlike SHriMP[1], Imsovision[2] had a different approach to visualization, it aimed to create visuals in a Virtual Reality (VR) environment, mapping C++ code to visual representation. There were other options like Tarantula, SeeSoft[3], etc. The initial source code representations were modeling UML diagrams, and the relationship of database objects, then involved various other schemas like sequence diagrams, and activity diagrams and some tools had flow diagrams to represent call stacks.

As software systems evolved, various approaches have been involved and changed traditional software development. The major change can be addressed as variety of programming languages, which makes visualization tools target to be very limited if tools aim to represent visuals in a certain language. Programming language independence was also nice to have before but in today's challenges it is considered as a must.

Also, the variety of database providers and the variety of the structure of the databases have increased exponentially. Various NoSQL databases and providers and different approaches to storing application data were developed over time. The traditional tools and notations have become outdated and deprecated day by day. Generating a UML diagram on a document-based application wouldn't make sense as much as it did before.

Another change was the architecture and the way software systems handled the challenges. Coming from a synchronized world, the software systems tend towards distributed, event-based systems. The idea of microservices became more and more practical due to the speed of technical advancements. The development of monolithic applicants started to have microservices, and distributed services, and the communication of the services could also be event-based and asynchronous, such as producing a message to an asynchronous queue.

The only change was not in technical details, The requirements were getting taken from the end user, which affects the system design such as database models and service implementations, and results in various branches of testing. Receiving user requirements in a formal way was quite necessary since any misunderstanding or misconception would create a butterfly effect in the whole lifecycle. This urges to have requirement definition documents as well as requirements modeling and visualization. Also, having a block of design phase has its own internal structure other than user requirements, including technical details about the software that is under development. That requires having multiple documentation and visualization tools for different software development phases, even in the same lifecycle phase, different tools might be used due to technical limitations such as varying programming languages between projects.

With today's tools, it's possible to convert source code to visual models while auto-generating the code is much easier and scalable, it limits the external manipulations on the model. Tools like GitUML[4], PlantUML[5], AppMap[6] can be referenced as these kinds of tools. Those tools are highly cohesive with the source code, any source code change is directly reflected in the visual models. On the other hand, those tools suffer from being customized, and mostly represent object-oriented paradigms, visualizing UML diagrams, sequence diagrams, etc. There are also other options like activity diagrams or call stack diagrams, but as said, it's an automatic process that prevents us from hiding unnecessary details, to show only desired logical details. Also, this lacks the ability to represent event-oriented systems like microservice applications.

Tools like Mermaid[7], Code2Flow[8] have more customization, providing the ability to have more control over the visualization. But this does not live inside the source code, the visualization documentation should stay in its own environment and it should be developed in that environment. This requires having different processes for source code development, and software visualization which removes the bond between source code and the visual models. It also requires that any change in the source code should require a change in the environment of visual models which is less sustainable that makes the situation even worse in today's agile, fast-evolving software applications.

In this study, we aimed to fully represent business logic in event-oriented applications with controllable syntax, which lives inside the source code. The visualization are directly be generated source code using comment lines, with a special syntax. Source code is first parsed into code tokens using tokenization tools, and then the syntax is recognized by the tool presented in this study, Docupyt. Docupyt analyses the syntax and builds visualization by landing those into its own data structure. This is accomplished by giving control over the visual models created, hiding and emphasizing the code parts to be represented, and merging the visualization environment and the source code. Since it's controllable it also enables the representation event-relation between microservices directly from the source code.

CHAPTER 2

RELATED WORKS & CURRENT TOOLS

The need for software visualization has been present since the beginning of software development because of its complex and abstract nature. Large-scale development always needs to expose its underlying logic independent of the programming language or the frameworks. For the very first examples of visualization tools a survey[9] was carried out to classify and identify software visualization tools of the time. The study introduces a framework and emphasizes the importance of transforming raw data into a more usable format and highlights the need to consider the nature of the data and the characteristics of the users when designing visual representations. The framework comprises five dimensions: tasks, audience, target, representation, and medium. The document emphasizes that different software engineering tasks require different visualizations and argues that one single software visualization tool can not address all tasks simultaneously. The *tasks* dimension in software visualization defines the specific software engineering tasks supported by a visualization system, such as development, maintenance, testing, and fault detection. The *audience* dimension specifies the attributes of the users of the visualization system, including their roles, experience levels, and information needs. The *target* dimension defines the aspects of the software that are visualized, such as architecture, design, and algorithms. The *representation* dimension determines how the visualization is constructed and presented to the user, including the visual structures and forms used. The *medium* dimension refers to the display medium where the visualization is rendered, such as paper, monitors, immersive virtual reality environments, or large-sized displays.

The study points out that the need for the tool on software visualization might depend on the specific task itself. Even though the tools referenced in the study have evolved and progressed and new tools have been added to the stack, this was important to categorize the utilities of the software visualization. The study aimed to categorize and utilize available software tools for specific tasks but it was important for us to have a target utility of the tool. For example, *ShriMP*[1] was the tool which is addressed to be used for reverse engineering and software maintenance purposes, *Tarantula* was addressed as a testing and defect location and *Imsovision* was supposed to be used best in development, reverse engineering and management purposes.

Another[10] survey was made upon identifying the main values of a software tool. The objective was to identify quality attributes and functional requirements especially research tools targeting visualization in the domains of software maintenance, reengineering, and reverse engineering. The comprehensive literature survey identified seven quality attributes, including rendering scalability, information scalability, interoperability, customizability, interactivity, usability, and adoptability, as well as seven functional requirements, such as views, abstraction, search, filters, code proximity, automatic layouts, and undo/history. *Rendering scalability* refers to the ability of visualization tools to handle large amounts of data efficiently, while *information scalability* addresses the ability of the tools to manage and present large amounts of information in a way that does not overwhelm the user. Essentially, information scalability ensures that the visualization tools can effectively handle and present information in a manner that is easily comprehensible and manageable for the user. *Interoperability* ensures that tools can work together effectively. *Interactivity* refers to the ability of users to manipulate visualizations, usability focuses on ease of use and user interface quality, and adaption to the tool's ability to meet the expectations of users. *Customizability* in software visualization tools refers to the ability to tailor the tools to specific user needs or domain-specific requirements, allowing for adaptability and extension to accommodate diverse visualization needs. *Usability* is a highly desirable requirement for software visualization tools, encompassing ease of use and the quality of the user interface, with a focus on intuitive design, cognitive overhead reduction, and seamless integration into existing processes. *Adoptability* pertains to the factors influencing the likelihood of a tool being adopted by users, emphasizing ease of use, adaptability to specific tasks, and the ability to integrate smoothly into existing processes, tools, and work practices. These quality attributes are essential for the development and evaluation of software visualization tools, as they contribute to their effectiveness and usefulness in related domains.

Functional requirements for software visualization tools were described as specific capabilities and features that the tools must possess to represent the software effectively. These requirements include aspects such as views, abstraction, search, filters, code proximity, automatic layouts, and undo/history. *Views* are the requirement of emphasizing different dimensions of the data, such as the time dimension or level of abstraction. *Abstraction* mechanisms are essential for managing complex software systems, allowing for the creation of higher-level abstractions from low-level program elements and categorizing elements based on specific properties. *Search* capabilities are crucial for software visualization tools, enabling users to locate specific textual or graphical elements within the software system. *Filters* allow for the pruning of nodes or arcs based on specific criteria, helping to reduce information overload and enhance the clarity of visualizations. *Code Proximity* refers to the ability of the visualizer to provide easy and fast access to the original, underlying source code, allowing users to navigate between visual representations and the corresponding source code. *Automatic layout* capabilities are essential for constructing effective visualizations, as they provide a starting point for further manual refinement and help optimize the properties of the graph for improved readability. We will redefine this to be layouts being automatically generated. An *undo* mechanism is crucial for allowing users to revert to previous states and track the history of actions performed during visualization, supporting progressive refinement and exploration.

In this study, we have based on these functional requirements slightly modifying them. We decided to take abstraction, search, filter and code proximity criterias since those were decided to be the most important ones for a microservice-based application. We also have taken the criterias for quality requirements: usability (visualizer to be easily used), adoptability (visualizer to be easily integrated). We decided that those criterias were the most useful ones for a microservice-based application. Including those we also added more criterias for visualizer tools: *event-orientation* which is the ability to represent event-based applications, *architectural* is the ability to represent architectural views, *flowable* is the ability to represent data and event flows and *logical* is the ability to represent business logic in the visualizer, *formatted* is the ability for visualizer to put boundaries and abilities into the tool to specify the tools aim that prevents unconscious usage of the tool without breaking customizability. and we merged all those requirements as "software visualizer requirements".

One of the first software visualization tools in the field was *SHriMP*[1], which is a visualization tool that provides a flexible and customizable environment for exploring software programs. It supports embedding multiple views, both graphical and textual, within a nested graph display of a program's software architecture. SHriMP has been redesigned and reimplemented using Java Bean components, allowing it to be integrated with other software tools. SHriMP uses nested graphs to represent software hierarchies like the package and class structure of a Java program. Relationships like inheritance are also visualized using arcs layered over the nested graph. SHriMP employs a fully zoomable interface to explore the software hierarchy using different zooming approaches. When magnified, nodes can display different views like the children of a package or its Javadoc.

SHriMP was useful for representing object-oriented applications. Also, it was important to be a role model for the next generation of software visualization tools. Even though it was one of the pioneers in the field, together with Imsovision, it was designed to work with Java applications and was dependent on Java, which makes a strong limitation in today's software development, since lacked to show functional parts of the source code.

One of the newer tools in the field is GitUML[4], which is a tool that integrates with Git version control systems and generates UML (Unified Modeling Language) diagrams based on the code stored in the repository. It automatically analyzes the codebase and generates visual representations of classes, relationships, and other structural elements present in the code in the form of UML diagrams. This can be particularly useful for developers and teams to gain a better understanding of the architecture and design of their software projects. By visualizing the codebase through UML diagrams, developers can identify patterns, dependencies, and potential design improvements more easily.

One of the key features that sets GitUML apart from its counterparts is its cross-language compatibility and its integration with Git's version control functionalities. By leveraging its integration with Git, GitUML can analyze codebases written in languages such as Java, C#, Python, JavaScript, and more, allowing developers to generate UML diagrams that provide a comprehensive overview of the project's architecture and design. This unique combination empowers developers to gain insights into projects written in various programming languages by automatically generating UML diagrams directly from the code stored in the Git repository. This not only streamlines the visualization process but also enables teams to track the evolution of code structures over time. With GitUML, developers can effectively navigate complex codebases, identify dependencies, and make informed decisions about software architecture and design, regardless of the languages used in the project.

GitUML adapted more attributes of software development such as version control and programming language independence, which increased *adoptability* but still lacked criterias like *event orientation* and *flowable.*

PlantUML[5] is an open-source tool that allows users to create diagrams using a simple and intuitive textual description language. It supports various types of diagrams, including UML diagrams, sequence diagrams, activity diagrams, and more. Users can write descriptions of these diagrams using plain text, and PlantUML then converts these descriptions into graphical representations automatically. This makes it easy for developers, architects, and other stakeholders to quickly generate and share diagrams without needing to use specialized diagramming software. PlantUML is often used in software development, system design, and documentation to visualize concepts, architectures, and processes clearly and concisely.

Mermaid[7] is another open-source tool designed for creating diagrams, particularly focusing on flowcharts, sequence diagrams, Gantt charts, and more. Similar to PlantUML, Mermaid also utilizes a simple and intuitive syntax for describing diagrams, making it easy for users to create visual representations of their ideas using plain text. One distinguishing feature of Mermaid is its support for creating interactive diagrams directly in Markdown files, which can be rendered in various environments such as web browsers, documentation platforms, and integrated development environments (IDEs). This enables developers and other users to embed dynamic and interactive diagrams directly into their documentation, presentations, or code comments, enhancing the overall clarity and interactivity of their materials. Additionally, Mermaid offers extensive customization options, allowing users to tailor the appearance and behavior of their diagrams to suit their specific needs and preferences.

PlantUML and Mermaid are different than the others by defining their context language, instead of directly reverse engineering from the source code. This increases the effort on documentation creation since documentation is created manually which is considerably a minus on *automatic layouts*, meanwhile increasing the *flowable, filter* attributes.

On the other hand, the studies[11,12] focus on reverse engineering the source code, like GitUML[8] and PlantUML[9], the study[11] does it by transforming the Abstract Syntax Tree (AST) of the code and eventually transforming it into a printable XML-like context language. It is designed for Java programming language, which has drawbacks but differs from other source code reverse engineering tools by being able to create not only structural diagrams but behavioral diagrams like sequence diagrams, activity diagrams, etc. A similar study[13] does it for Python code, which can create call graphs both have the upper hand compared to other source code based software visualization tools (GitUML, PlantUML).

Appmap[6] is one of the latest tools that is designed for automatically generating and visualizing interactive maps of applications' codebases and runtime behavior. It captures data about the interactions within an application, including method calls, HTTP requests, database queries, and more, and then presents this information in a visual format that developers can explore and analyze. These maps, often referred to as "AppMap" provide insights into how different components of an application interact with each other, helping developers understand its structure, dependencies, and behavior. One of the key features of AppMap is its ability to integrate seamlessly into the development workflow. It typically operates as a background process or agent that instruments the application code, capturing relevant data during runtime. Developers can then view and interact with the generated maps using dedicated tools or plugins within their integrated development environments (IDEs) or through web-based interfaces. This allows for real-time visualization and analysis of application behavior, aiding in debugging, performance optimization, and architectural understanding. Appmap is a candidate to be the strongest visualization tool among source code reversing technologies. However, it still lacks *filter* criteria like other source code documentation tools but has the upper hand on *flowable* among the source code-based software visualization tools. It also can show architectural views more than other source code reversing software visualization tools.

Code2Flow[3] has its context language, similar to Mermaid and PlantUML which is capable of showing the logical flows. It's an open-source project easily *usable* and *adoptable*.

Lastly, another to discuss, is Diagrams[14] which differ from others as being able to show architectural details, hence having stronger *architectural* criteria than the other tools. Also, Diagrams has a basic ruleset to start with which can be adapted to specific projects, it is possible to make manipulations on the project level. Which makes it *formatted*. However, it suffers from the ability to represent business logic compared to other tools.

CHAPTER 3

EVENT-BASED SYSTEM MODELING WITH eEPC

3.1. eEpc Notation on Event-Based Systems

The eEPC (Extended Event-Driven Process Chain) notation offers significant advantages for microservice modeling and aligns well with the fundamental principles of microservice architecture. It's discussed in the study[15], primarily, its event-centric approach facilitates the identification and decomposition of microservices based on the events that trigger and conclude their execution cycles. This event-driven perspective not only delineates the bounded contexts of individual microservices but also inherently supports the implementation of asynchronous communication mechanisms, such as message queues, which are essential for enabling loose coupling and autonomy among microservices. Moreover, the eEPC notation provides a comprehensive visual representation of both high-level processes and granular sub-processes, enhancing the clarity and understandability of microservice responsibilities and interactions. By directly capturing the domain knowledge through the explicit modeling of business events and processes, eEPC diagrams encapsulate the application domain's intrinsic logic, fostering a more seamless translation from analysis to design and implementation phases.

In contrast to the traditional Object-Oriented Analysis and Design (OOAD) approaches, which primarily focus on class-based decomposition, the process-centric nature of eEPC aligns more naturally with the architectural principles of microservices. While OOAD excels in data modeling and encapsulation, its class-centric view often struggles to encompass the distributed and event-driven nature of microservice architectures, leading to challenges in identifying and maintaining the appropriate levels of cohesion, coupling, and autonomy among microservices.

eEPC notation leverages a methodology that supports the intrinsic characteristics of microservices, such as high cohesion within individual microservices, loose coupling between them, and the ability to operate autonomously while collaborating through well-defined event-based interfaces. This alignment between the modeling approach and the target architecture not only streamlines the analysis and design phases but also facilitates the subsequent implementation and maintenance of microservice-based systems.

3.2. Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a data structure employed to represent the structure of a programming language. Each element of this structure encompasses the components that constitute the syntax of the program (variables, operators, expressions, etc.) and their relationships. ASTs depict a program as a tree structure, hence the term "abstract." ASTs are commonly utilized in software tools such as compilers or interpreters. These tools employ ASTs to analyze programs for various purposes. For instance, a compiler transforms source code into an AST and then conducts operations such as optimization or error checking on this AST. ASTs also play a significant role in many software tools, including code analysis and automatic transformation tools. In essence, ASTs serve as a powerful tool for representing the structure of a programming language and performing various analyses and operations on programs.

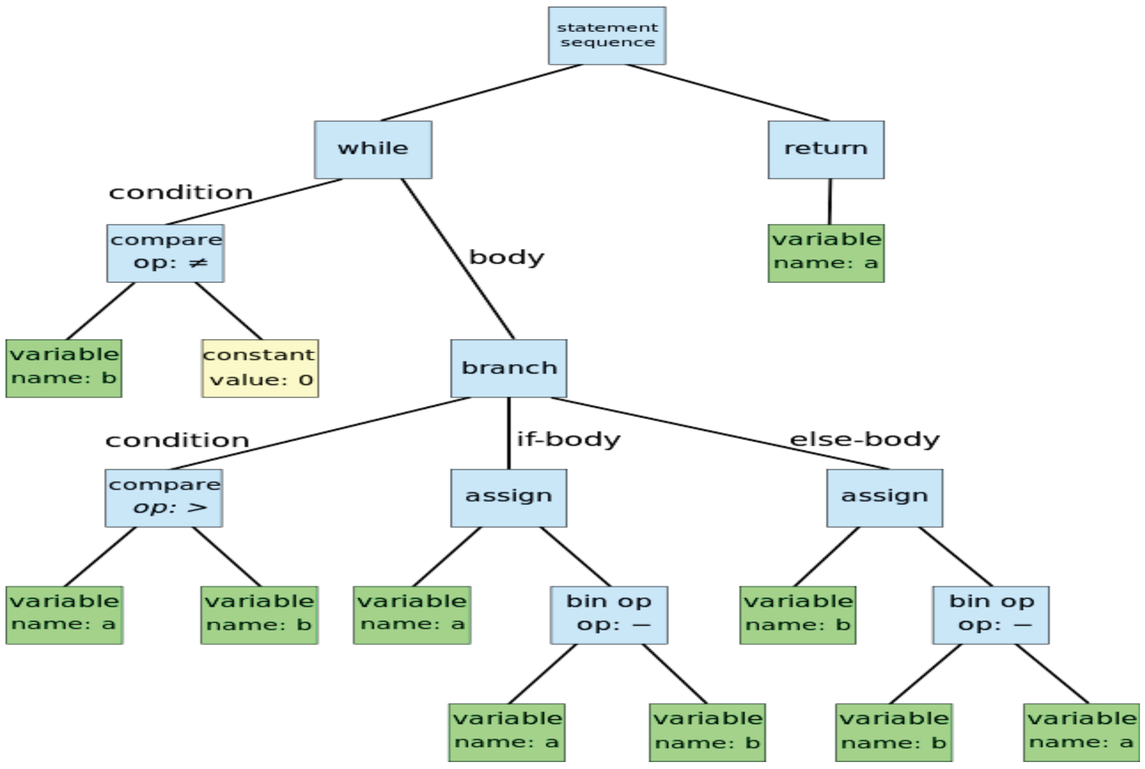

Figure 1. Example Abstract Syntax Tree

3.3. Tokenization

Tokenization is the process of segmenting a continuous stream of text into discrete units, known as tokens, which constitute the basic building blocks for subsequent processing and analysis. These tokens can represent words, phrases, symbols, or other meaningful elements within the given text, depending on the specific requirements of the task at hand. The tokenization process is guided by a set of rules or heuristics that define the criteria for segmentation, often based on patterns such as whitespace characters, punctuation marks, or language-specific conventions.

By breaking down the text into meaningful units, tokenization enables subsequent algorithms and models to effectively process and understand the underlying data, leading to improved performance and accuracy in language-related applications. Additionally, tokenization can be tailored to specific requirements, encompassing various strategies such as word tokenization, sentence tokenization, or subword tokenization (e.g., splitting words into stems or morphemes). Furthermore, tokenization can be adapted to handle language-specific challenges, such as contractions, abbreviations, or special characters, ensuring robust and accurate text processing. Consequently, tokenization serves as a crucial foundation for NLP pipelines, facilitating the extraction of valuable insights and enabling the development of sophisticated language-based applications across diverse domains, which is useful for software visualization tools either.

3.4. Graphviz

Graphviz is an open-source graph visualization software package that provides a powerful set of tools for creating and visualizing graph data structures. It is widely used in various domains, including software engineering, network analysis, database design, and scientific visualization. At its core, Graphviz utilizes a domain-specific language (DOT) to describe the structure and properties of graphs, which can include nodes, edges, and associated attributes. This textual representation allows users to define complex graphs in a concise and human-readable format, making it easier to create, modify, and share graph descriptions.

One of the key strengths of Graphviz lies in its ability to automatically layout and render graphs based on the provided DOT descriptions. It employs sophisticated algorithms and layout engines to calculate the optimal positioning of nodes and edges, ensuring that the resulting visualizations are aesthetically pleasing and easy to comprehend. Graphviz supports various layout styles, such as hierarchical, force-directed, radial, and circular, allowing users to choose the most suitable layout for their specific use case. Additionally, Graphviz offers a wide range of customization options, enabling users to modify the appearance of nodes, edges, labels, and other graph elements through attributes and styling mechanisms. The generated visualizations can be exported in various formats, including PNG, SVG, PDF, and more, facilitating seamless integration with other applications and documentation processes.

CHAPTER 4

PROPOSED METHOD

4.1. Research Methodology

The research methodology for the development of Docupyt, a tool aimed at improving the modeling of microservice-based applications, is a comprehensive and iterative process divided into three main phases: Problem Identification, Solution Design, and Evaluation.

The initial phase of this research methodology is **problem identification** that Docupyt aims to solve. This begins with a literature review. The literature review involves examination of existing research and publications and the currently available software tools related to the modeling of microservice-based applications. During this review, the complexities associated with microservice architectures, such as managing independent services and ensuring efficient inter-service communication, are highlighted. The literature review reveals significant limitations in current modeling tools, which often provide over-detailed representations that are irrelevant to high-level logic essential for understanding and managing microservice architectures.

Following the literature review, the specific problem is identified. The primary issue recognized is the inadequacy of current modeling tools to effectively simplify and visualize the high-level logic of microservice-based applications in an automatic conversion instead of maintaining the documentation manually apart from the codebase. These tools frequently fail to filter out non-essential details, resulting in models that are cumbersome and difficult to interpret. This problem identification is further refined through discussions with target users of the tool, software experts, and developers. These discussions confirm the necessity for a new tool that can bridge the gap between detailed code representations and high-level logical models, thus establishing the foundation for Docupyt.

With a clear understanding of the problem, the research proceeds to the **solution design** phase. The initial step in this phase is the creation of an initial structural template for Docupyt. This template involves the definition of a custom context language, embedded within code comments, which uses specific string tokens to denote high-level logic structures. The tokens, such as *"if:", "end:", "event:",* and *"act:",* are designed to be intuitive and expressive, allowing developers to concisely annotate their code with meaningful logical constructs. This approach aims to overcome the limitations of Abstract Syntax Trees (ASTs), which, while detailed, often include unnecessary syntactic information that complicates the modeling process.

Development of Docupyt is carried out through multiple iterations, each aimed at refining and enhancing the tool. In each iteration, the parsing and visualization pipeline is improved. This pipeline includes stages such as lexical analysis, where code comments are tokenized; parsing, where these tokens are structured according to the defined syntax; semantic analysis, which interprets the parsed tokens; and graph generation, which creates high-level representation graphs from the interpreted tokens. These development iterations are crucial for incorporating feedback and making incremental improvements to the tool's functionality and user experience.

The final phase of the research methodology is the **evaluation** of Docupyt. This phase begins with another round of literature review, this time focusing on comparing Docupyt with existing tools in the field. The literature review supports the claim that Docupyt provides a more manageable and high-level visualization of code logic, particularly suitable for event-based systems and microservice architectures.

Subsequently, case studies are conducted to evaluate Docupyt's performance in actual projects. These case studies demonstrate how Docupyt helps in visualizing and managing complex microservice-based applications, providing empirical evidence of its benefits.

The evaluation phase concludes with a summary of results. This summary synthesizes the findings from the literature review, user feedback, and case studies, presenting a comprehensive assessment of Docupyt's capabilities. The results highlight the tool's success in simplifying the modeling process and enhancing the understanding of microservice architectures. Additionally, potential areas for further improvement are identified, setting the stage for future research and development efforts. In conclusion, the research methodology for developing Docupyt is a structured and iterative process aimed at addressing a critical gap in the modeling of microservice-based applications. Through meticulous problem identification, innovative solution design, and thorough evaluation, Docupyt emerges as a valuable tool that significantly enhances the ability to model and manage complex microservice architectures.

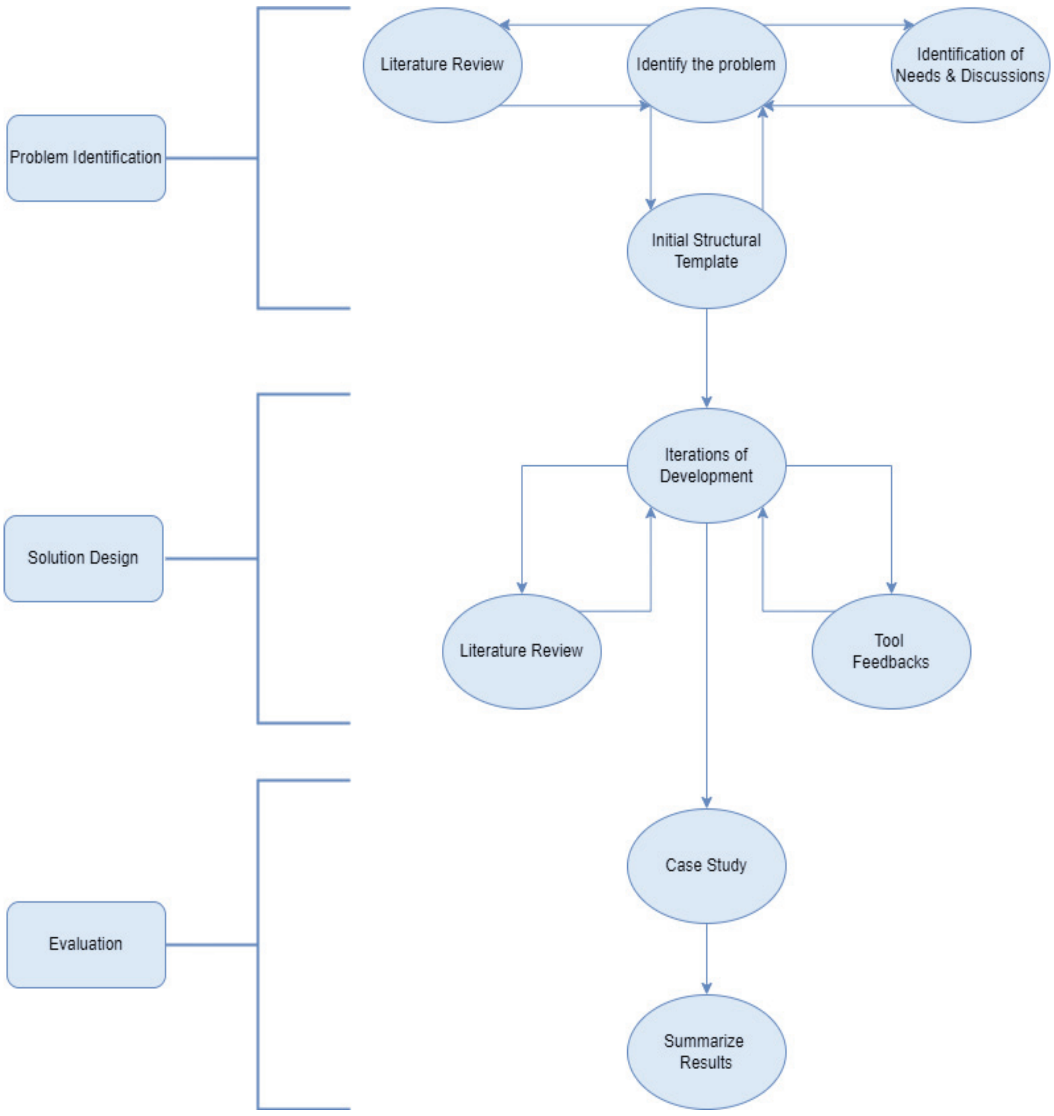

Figure 2. Research Methodology

4.2. Research Methodology – Case Study

As discussed in Section 3.1, we have extended eEPC notation and selected it as the underlying visual representation of the tool to be developed. It's aimed to represent both high-level structural representation as it's already done with eEPC and also low-level application logic, which includes application codes to have representation with it. Docupyt is aimed to have both layers handled with the same notations and context language.

We have conducted a survey starting the development of a microservice-based system in the banking domain on concentrating 16 components (services) which is powered upon Amazon Web Services (AWS) infrastructure. Each component (service) will be referred to as Lambdas in the study. We drew eEPC diagrams of the services before the beginning of the project, after 6 months 9 of them needed critical manipulations on the visual representation (eEPC diagram). Regarding this conceptual analysis, related to our study a microservice-based documentation tool Docupyt's main task would be documentation and maintenance. The user should be a software developer independent of the experience level. The representation will be created by customizing eEPC notation by generating the code itself, using its own tokens inside the code comments. Upon the development of the latest technologies, medium meta would be ignored in conceptual thinking it will be stored in the digital environment.

The project adopted agile principles, which means it should also be taken into account that some services would change more than once during the development, which requires multiple changes on the same service diagram. Docupyt aims to be a two-way bridge between requirements and implementation by using customized eEPC notation.

Some of the diagrams are selected as examples that are shown below.

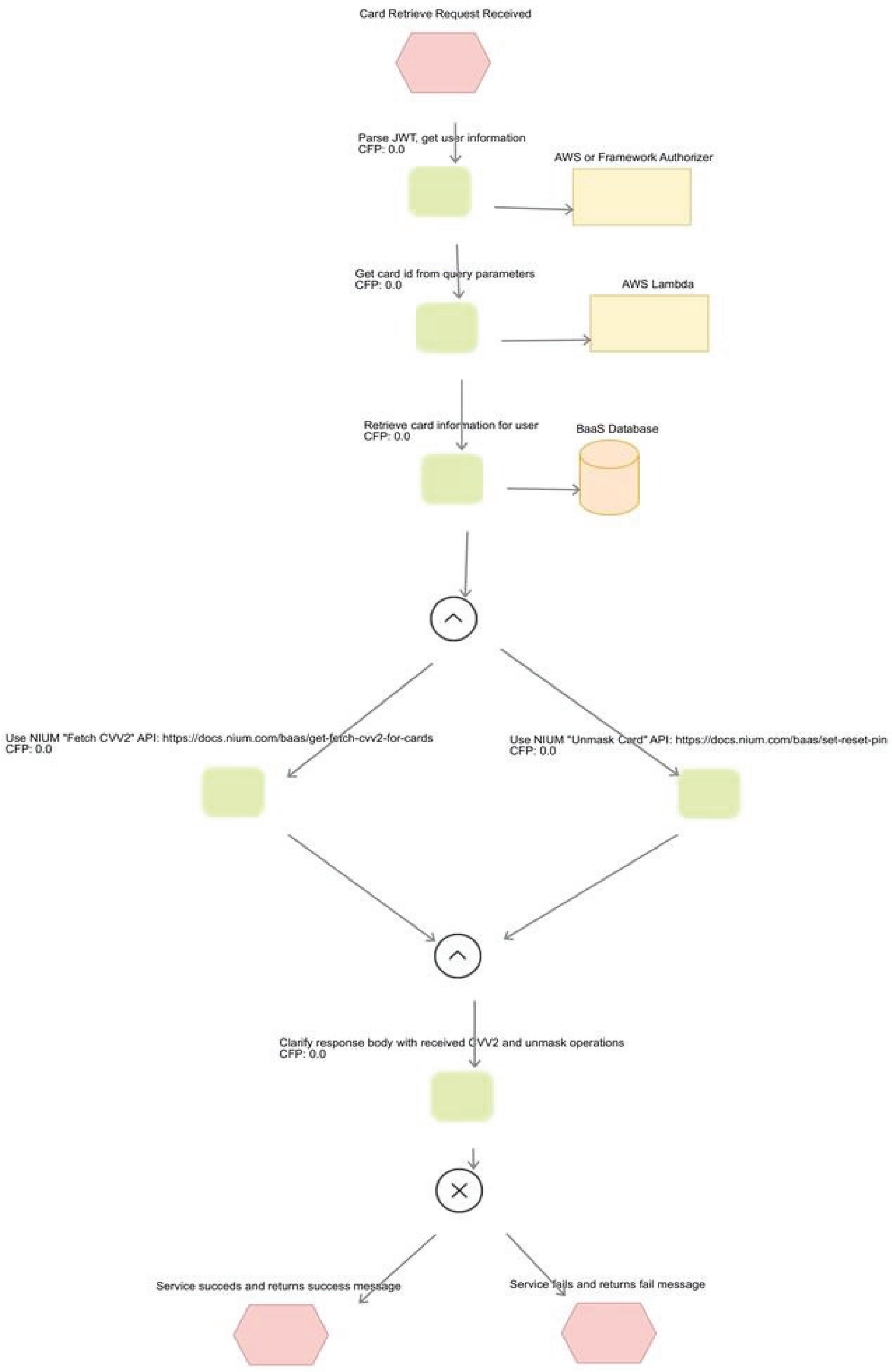
Card Retrieve Request Received
Parse JWT, get user information
CFP: 0.0
AWS or Framework Authorizer
Get card id from query parameters
CFP: 0.0
AWS Lambda
Retrieve card information for user
CFP: 0.0
BaaS Database
Use NIUM "Fetch CVV2" API: https://docs.nium.com/baas/get-fetch-cvv2-for-cards
CFP: 0.0
Use NIUM "Unmask Card" API: https://docs.nium.com/baas/set-reset-pin
CFP: 0.0
Clarify response body with received CVV2 and unmask operations
CFP: 0.0
Service succeds and returns success message
Service fails and returns fail message

Figure 3. Lambda-1 eEPC before development

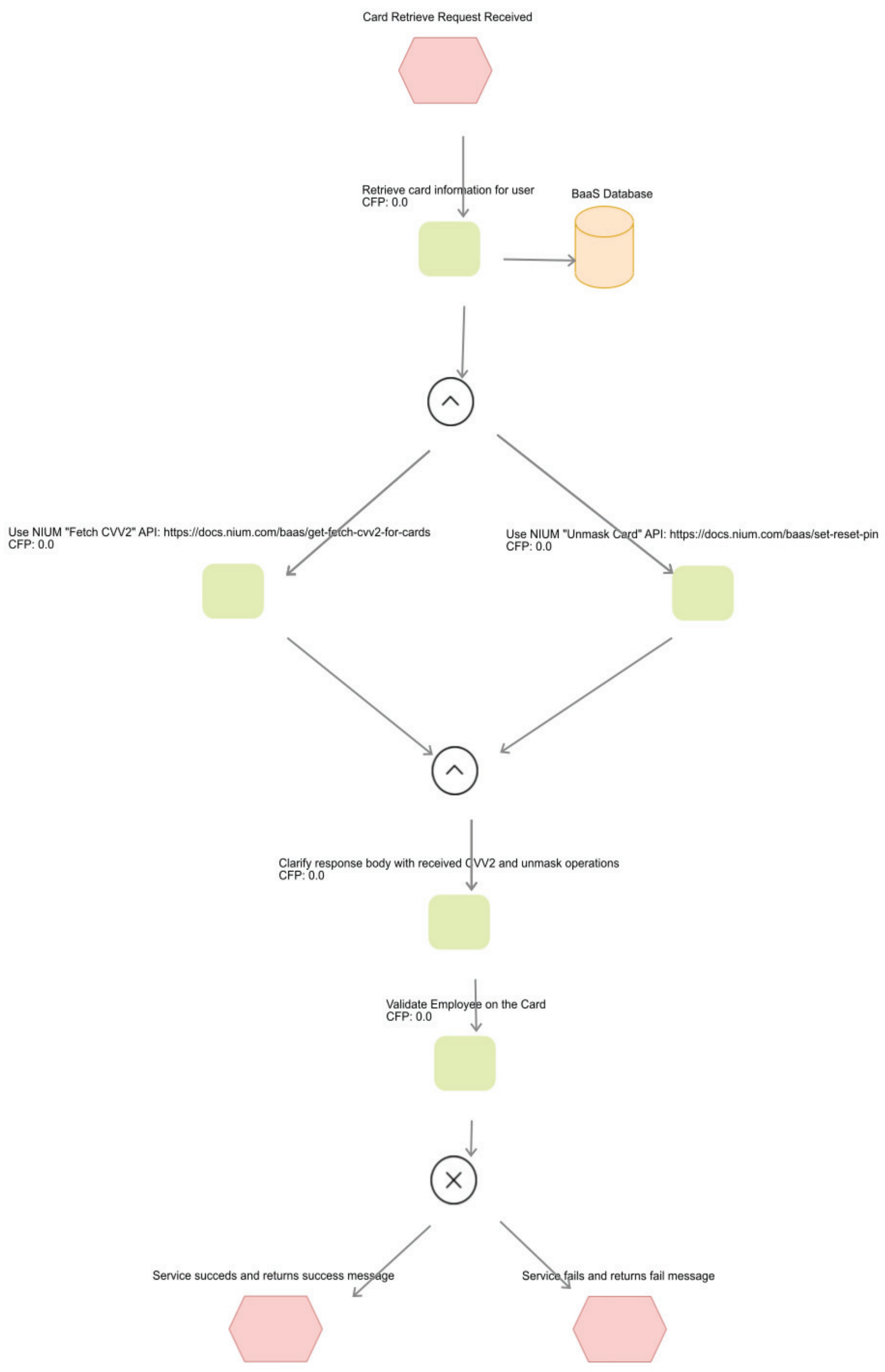
Card Retrieve Request Received
Retrieve card information for user
CFP: 0.0
BaaS Database
Use NIUM "Fetch CVV2" API: https://docs.nium.com/baas/get-fetch-cvv2-for-cards
CFP: 0.0
Use NIUM "Unmask Card" API: https://docs.nium.com/baas/set-reset-pin
CFP: 0.0
Clarify response body with received CVV2 and unmask operations
CFP: 0.0
Validate Employee on the Card
CFP: 0.0
Service succeds and returns success message
Service fails and returns fail message

Figure 4. Lambda-1 eEPC after development

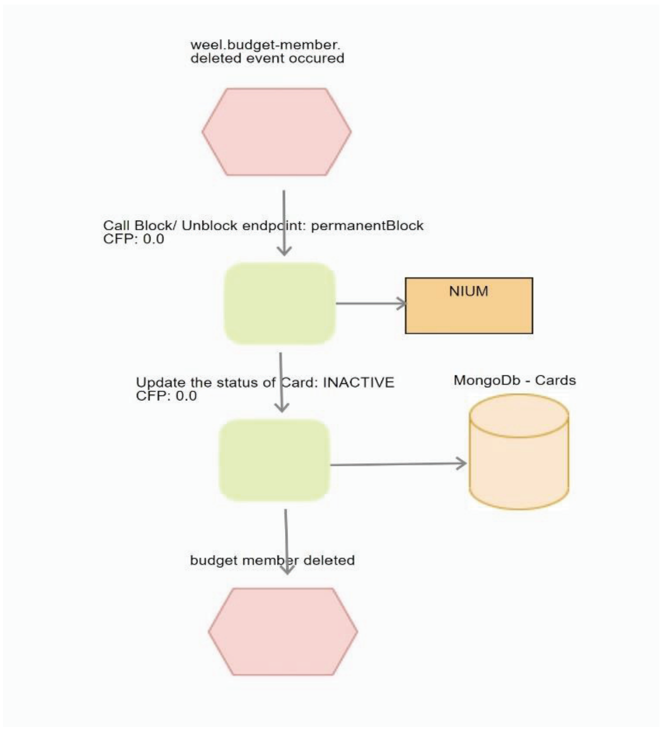

Figure 5. Lambda-2 eEPC before development

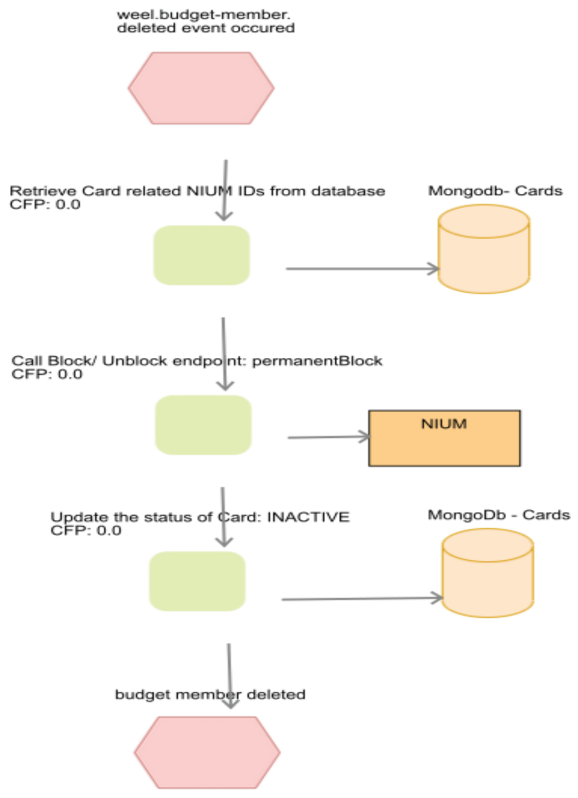
weel.budget-member.
deleted event occured
Retrieve Card related NIUM IDs from database
CFP: 0.0
Mongodb- Cards
Call Block/ Unblock endpoint: permanentBlock
CFP: 0.0
NIUM
Update the status of Card: INACTIVE
CFP: 0.0
MongoDb - Cards
budget member deleted

Figure 6. Lambda-2 eEPC after development

4.2. Development of Software Tool: Docupyt

4.2.1. Initial Iteration of Development

While Abstract Syntax Trees (ASTs) provide a comprehensive representation of code structure, they inherently include granular details that may obscure the higher-level logic developers seek to visualize. When directly visualizing ASTs, developers are confronted with an overwhelming amount of information, including syntactic constructs, variable declarations, and control flow statements. This redundancy of detail not only complicates the visualization process but also diminishes developers' ability to focus on the essential logic patterns within the code.

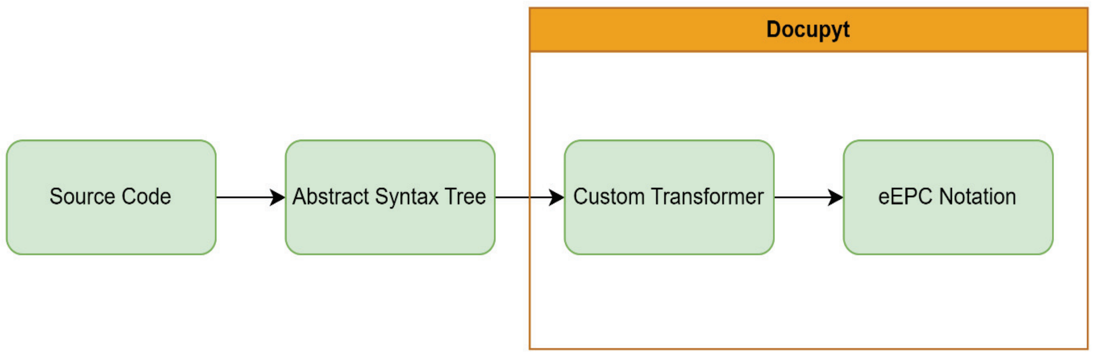

Figure 7. Docupyt Structure on Initial Iteration

Having AST as our domain structure would also cause problems when filtering out unnecessary source code components like variables, classes, functions, etc. As it’s discussed to have limitations of *filter* ability, we would also like to represent “hidden” logic, as a model can have more than the source code. Furthermore, the nested or imported source code parts would also increase the complexity of the model, as a model should be simple to understand.

4.2.2. Development in Iterations

Due to the limitations of using AST, we moved through defining our own tokens which should be recognized inside the source code. Central to our approach is the definition of custom string tokens, which serves as the foundation for expressing high-level logic structures within code comments. The tokens will form the context language which is designed to be intuitive and expressive, allowing developers to describe logic in a concise yet powerful manner. The maintenance is done by writing code-in documentation with specific keywords that represent context language. Additionally, the context language supports nesting and composition, enabling developers to express complex logic patterns effectively.

Once comments written in the context language are identified within the codebase, we employ a parsing and visualization pipeline to transform them into a high-level representation graph. This pipeline consists of several stages, including lexical analysis, parsing, semantic analysis, and graph generation. During lexical analysis, comments are tokenized and parsed according to the syntax of the context language. The parsed comments are then interpreted to construct a representation graph that captures the underlying logical structures.

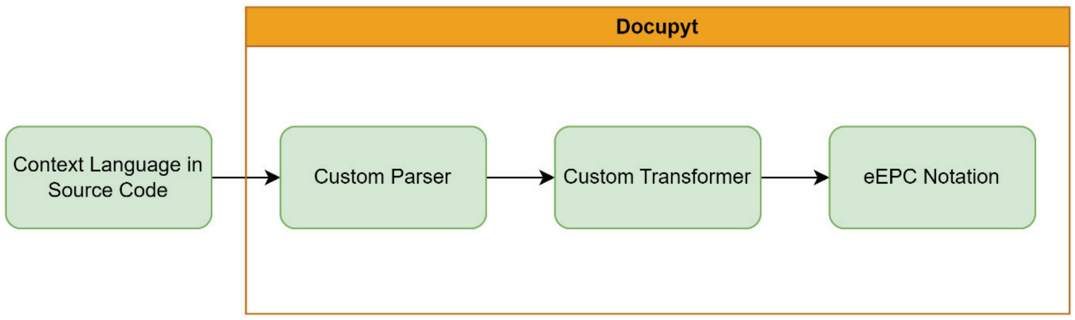

Figure 8. Docupyt Structure

After we moved our contextual language into the source code, even though we had to manage the context language in the source code, we gained more control over the output model. The context model lies inside the code similar to any code-in documentation. The language is embedded inside code comments, which is applicable to any programming language.

On our first iteration we have defined the the tokens as below;

- "*if:*" defines a conditional logic.
- "*end:*" defines end of a branched logic.
- "*event:*" corresponds to an eEPC event.
- "*act:*" corresponds to an eEPC action.
- "*diagram:*" defines a diagram to be represented by Docupyt.
- "*end-diagram:*" is used to limit diagram context.

A Docupyt-adapted real-life example code block is shown in Figure 9.

```
# main-diagram: GET_Cards_Controller
def retrieve_cards(...) -> Tuple[List[Dict[str, Any]], int]:
    # event: Get cards request received
    params = dict(...)

    # if: event: User without permissions  act: Should get cards owned by the
user
    if not has_budget_permission and not has_subscription_permission:
        params.update(dict(employee_id=employee_id))

    # else: event: Only subscription permission is present
    # act: Only subscription cards and cards owned by the user are returned.
    # end:
    if has_subscription_permission and not has_budget_permission:
        params.update(
            employee_id=employee_id,
            only_subscription_cards=True,
        )

    query_result: GetCardsQueryResult = get_cards(
        **params
    )

    cards_count = query_result["total_count"]
    card_list = query_result["documents"]

    # act: query cards
    response_body = [
        validate_model(
            exclude={"cvv", "unmasked_number"}
        )
        for card in card_list
    ]
    url = “business_url”

    # act: create pagination event: service response returned
    return (
        create_get_items_response(
            …
        ),
        HTTPStatus.OK,
    )
# end-main-diagram:
```

Figure 9. Sample source code block documented using Docupyt tokens

Docupyt outputs a representation model of as in Figure 10.

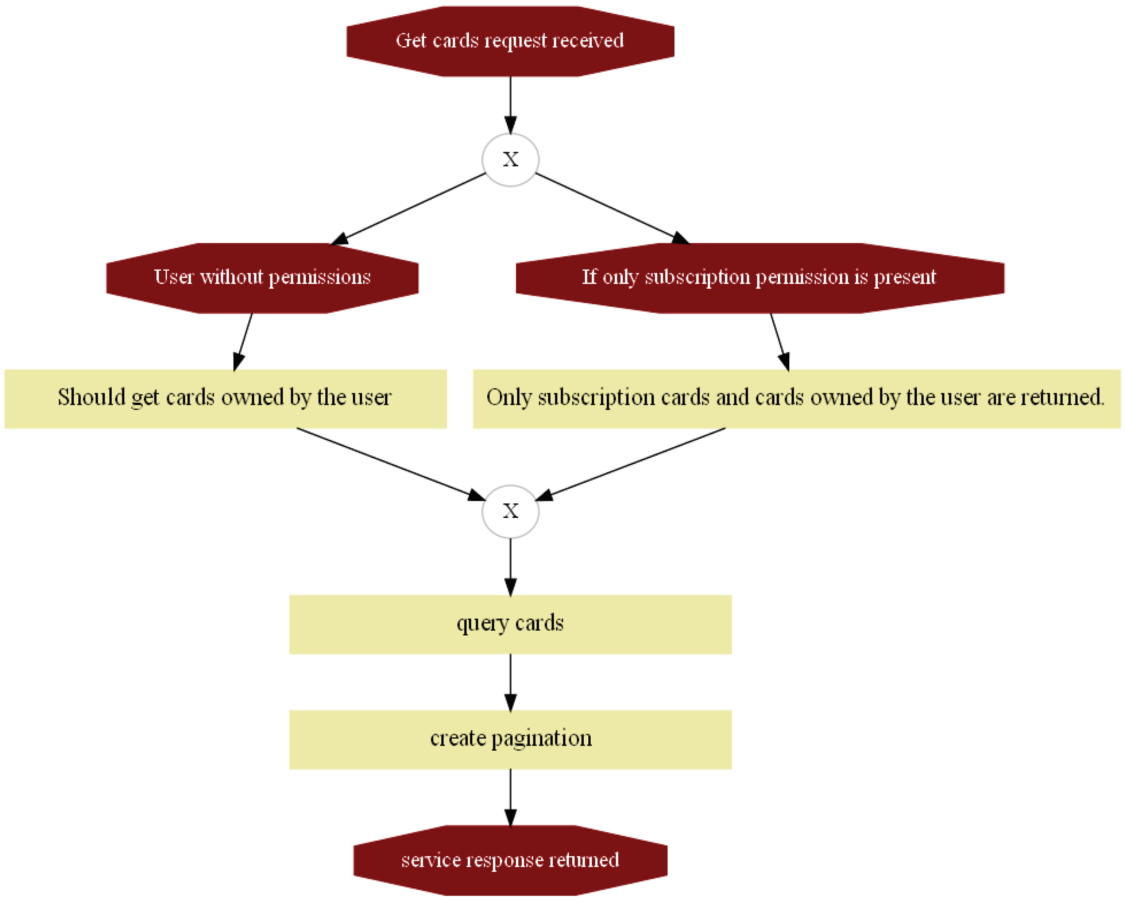
Get cards request received
X
User without permissions
If only subscription permission is present
Should get cards owned by the user
Only subscription cards and cards owned by the user are returned.
X
query cards
create pagination
service response returned

Figure 10. Docupyt output at the end of development iteration 1

Based on a microservice application that had more than 1-year development time, we sampled over 10 services with Docupyt. Based on the feedbacks we have extended our context language to meet the needs of a software representation. The most crucial parts like database connections, external API calls and inner processes are added in the phase of development (iteration 1).

So the context language has become:

- "*if:*" defines a conditional logic.
- "*end:*" defines end of a branched logic.
- "*event:*" corresponds to an eEPC event.
- "*act:*" corresponds to an eEPC action.
- *"[=]"*: corresponds to a database connection.
- *"->":* corresponds to an outgoing information in an API call.
- *"<-":* corresponds to an incoming information in an API call.
- "*main-diagram:*" defines a main diagram to be represented by Docupyt.
- "*end-diagram-main:*" is used to limit the main diagram context.
- *"inner-diagram":* defines an inner diagram.
- *"end-diagram:"* is used to limit the inner diagram context.

We extended eEPC by adding database connections and API calls by respecting its nature, which means they can be supported by eEPC actions. So a database connection can be represented by concatenating it into the action same as API calls. To clarify further an example output is shown in Figure11.

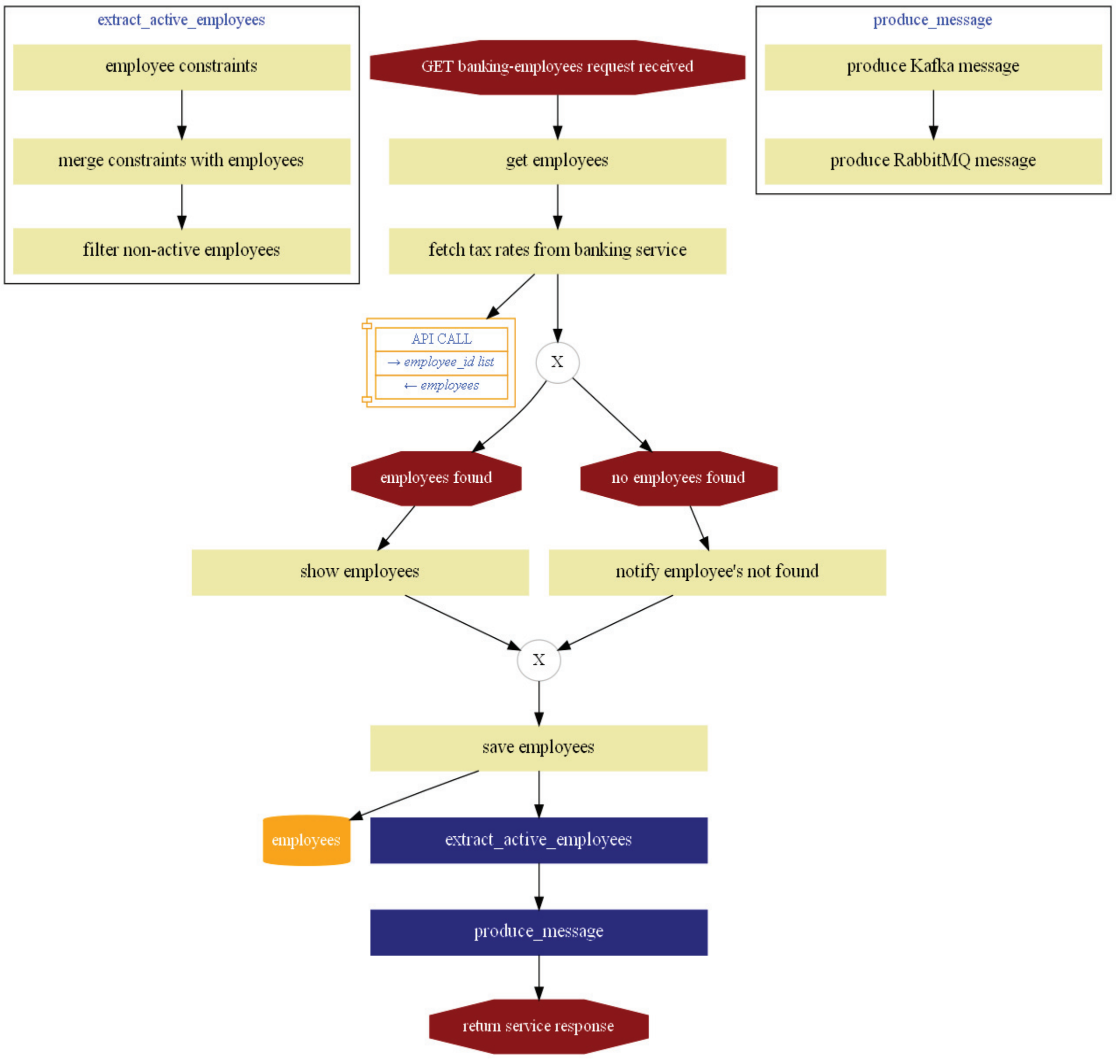

Figure 11. An example output of Docupyt at iteration 2

In our latest iteration, we have added the architectural view of Docupyt. We have extended our context language with architectural notations that represent event-oriented systems. In order to accomplish that we have defined the following keywords:

- "*subscribes:*" defines a triggering point, which is appendible to an event.
- "*publishes:*" defines the consuming point for an event.

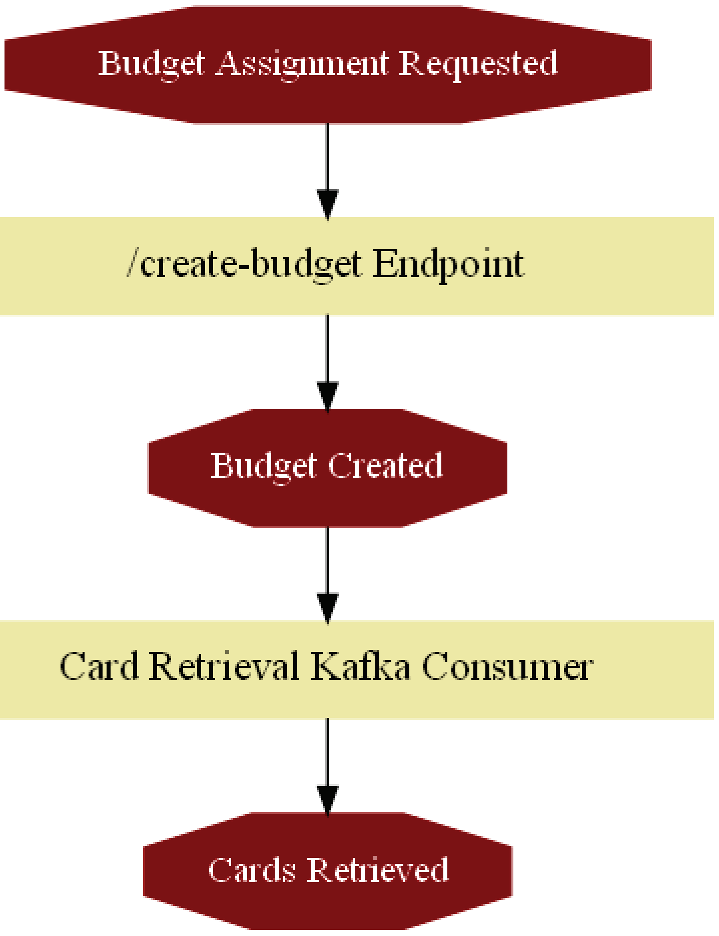

Figure 12. Architectural view of Docupyt

4.3. Tokens & Usage

As discussed in[15], to quickly summarize the benefit of using Extended Event-Driven Process Chains (eEPC) in event-based software visualization is that it provides a clear and structured way to model the high-level process and identify the bounded context of microservices. In the context of Event-Oriented Analysis and Design (EOAD), eEPC allows for the visualization of events, functions, and connectors, which helps in understanding the flow of events and the interactions between different processes. This visualization aids in identifying the boundaries of the context, as events trigger microservices and generate other events upon completion, thus delineating the scope of each microservice. Additionally, eEPC diagrams can be used to model both the AS-IS and TO-BE processes, providing a comprehensive view of the system's behavior and the changes introduced by the automation of processes. Overall, eEPC serves as a valuable tool for event-based software visualization, enabling a clear representation of the event-driven nature of microservice-based architectures and facilitating the analysis and design of MS-based solutions.

While defining Docupyt’s tokens, we have based on eEPC notation and added remarkable new components to it in the needs of a software visual model as shown in Section 4.2.2. This includes having traditional eEPC components as activities and events as well as defining new ones like external call representations, database connections, etc.

4.3.1. Defining Process Components

In the context of the Extended Event-Driven Process Chains (eEPC) diagram, an **activity** represents a specific function or process within the system. In the eEPC diagram, activities are depicted as nodes and are connected by connectors to illustrate the flow of events and functions within the system. Each activity in the eEPC diagram represents a distinct step or action within the process, and the connections between activities indicate the sequence and dependencies of these actions. The eEPC diagram provides a visual representation of the high-level process, including the events, functions, and connectors, and helps in understanding the flow of events and interactions between different processes within the system. An activity represents a small block or part of business logic.

As activities represent actions, properties of actions are limited to be bound to the activity in the context of Docupyt. This implies that an external API call, a database data exchange can only be connected to a Docupyt activity. An activity is defined in Docupyt as below;

act: Example Activity

which is represented as below.

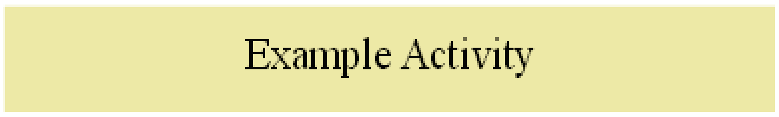

Figure 13. Activity Representation

In an Extended Event-Driven Process Chains (eEPC) diagram, an event represents a specific occurrence or trigger within the system. Events are essential components of the eEPC diagram and play a crucial role in identifying the bounded context of microservices in the context of Event-Oriented Analysis and Design (EOAD). An event triggers the microservice and, upon its completion, generates another event, effectively delineating the boundaries of the context. This event-driven approach is fundamental in the design and analysis of microservice-based systems, as it enables asynchronous communication and the identification of highly cohesive and loosely coupled microservices. Therefore, in the eEPC diagram, events are depicted as nodes and are used to illustrate the flow of events and functions within the system, providing a visual representation of the high-level process and the interactions between different processes.

Events do not represent a block of actions, hence they represent an ending occasion of a small block of process, or triggering/starting flag of a small block of process. Thus, properties of actions can not be bound to an event in the context of Docupyt. An event is defined in Docupyt as below;

event: Example Event

which is represented as below.

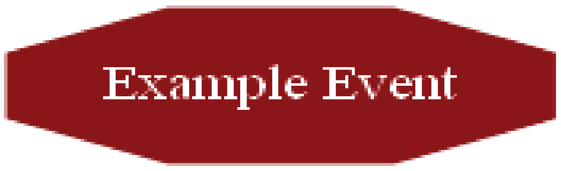

Figure 14. Event Representation

A main diagram in Docupyt is a process that contains blocks of activities and events, and as output, it represents an application service that contains business logic. It starts with a start token and ends with an end token which is defined in Docupyt. Everything in between the main process start token and the main process end token builds up the process. While creating the visual model, Docupyt recognizes those tokens and parses them. How a main diagram starts and ends is shown below.

main-diagram: Example Diagram

end-diagram-main:

In the context of the Extended Event-Driven Process Chains (eEPC), a subprocess refers to a specific, isolated part of a larger process that is managed by an individual microservice in a microservice-based architecture (MSbA). In the eEPC diagram, subprocesses are represented as distinct components that are handled by individual microservices, such as the application, verification, evaluation, and notification subprocesses mentioned in the document. Each subprocess is designed to be autonomous and responsible for a single business capacity, and they can communicate and coordinate with each other to manage the entire process. These subprocesses are designed to be composable, allowing them to be integrated with external microservices to perform other possible use-cases. The nature of eEPC and microservice-based architectures provides fault tolerance, scalability, and autonomy for each subprocess, contributing to the overall resilience and flexibility of the system.

As well as main diagrams Docupyt allows to define sub-diagrams. Sub-diagrams are reusable components inside the main diagrams. Those consist of blocks of events and activities to represent the flow and are reused in several different flows. Those are defined with reused blocks of code. Sub-diagrams are defined below.

inner-diagram: Example Flow

end-diagram:

To use a sub-process in a main process flow annotation is used as below;

inner-flow: Example Flow

A sample output of the combination is shown below.

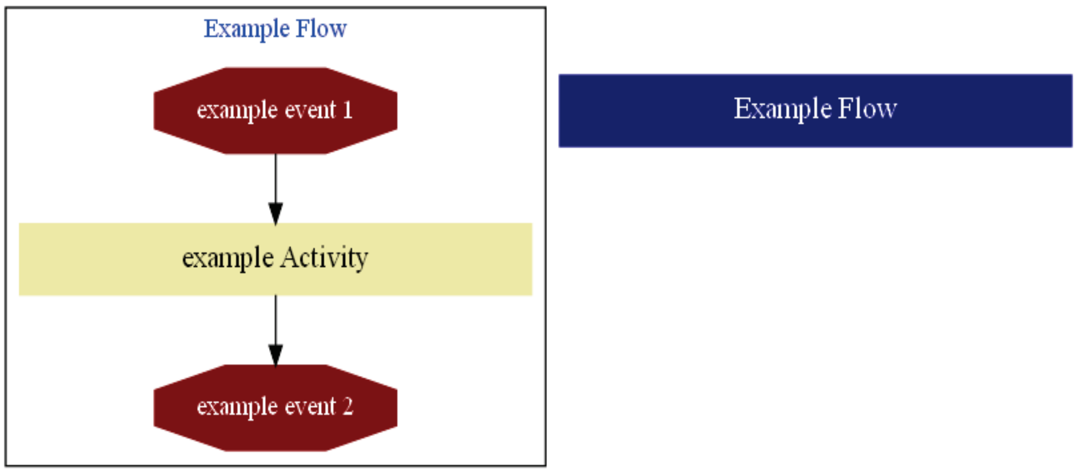

Figure 15. Sub-diagram Representation

4.3.2. Defining Process Properties

Traditional eEPC elements include properties such as organizational units, information objects, application systems. These properties provide comprehensive details on how processes are executed, including the resources required, the responsible organizational units, and any applicable legal constraints when used in industrial requirement specification. In our approach, instead of the conventional eEPC properties like organizational units and performance indicators, we have tailored the tool to have software specific properties instead by adding and removing specific properties. Instead of those properties, we have added database tables, collection links, and external API calls associated with each action. This modification allows for a more relevant and practical representation of the system's architecture, focusing on key technical components that are crucial in a software environment. By highlighting these elements, our tool offers a detailed and precise visualization of the interactions and dependencies within the system, enhancing the analysis and design of modern software architectures.

By integrating database connections, we provide a comprehensive view of how data is ingested, processed, and retrieved, thus highlighting the interaction between various microservices and their data dependencies. A database can be defined in a service flow by connecting it to an activity. A database connection can only be integrated to an eEPC activity due to the methodology of eEPC.

A database connection is defined and integrated with an action as below;

end-diagram:

which outputs as below;

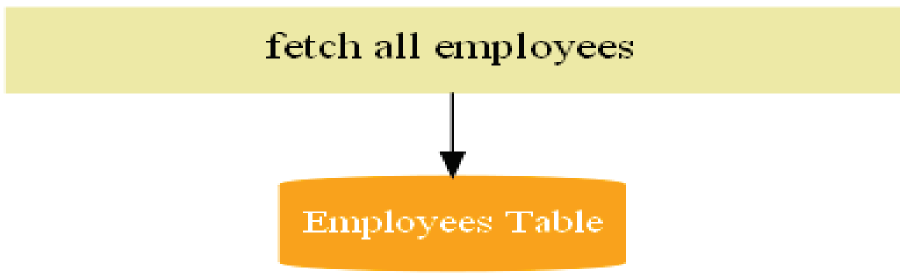

Figure 16. Database Connection Representation

Another property added on the visualization of external API calls associated with each action. This representation includes the identification of the endpoints, the nature of the requests and responses, and the overall interaction patterns with external systems. By mapping out API calls, our tool provides critical insights into how the system communicates with external services, highlighting dependencies and data exchange processes. This detailed visualization captures the flow of data to and from external sources, representing how different microservices interact with third-party applications and services. Emphasizing API calls enhances the understanding of integration points and the architectural design of the system, making it easier to identify potential bottlenecks, optimize data flow, and ensure robust and efficient communication between components. This focus on external API interactions is particularly valuable in the context of microservice-based architectures, where seamless integration with external systems is crucial for maintaining the overall functionality and performance of the application.

An external call is defined and integrated with an action as below;

act: fetch all employees from External Service <- employees -> id_list

which outputs as below;

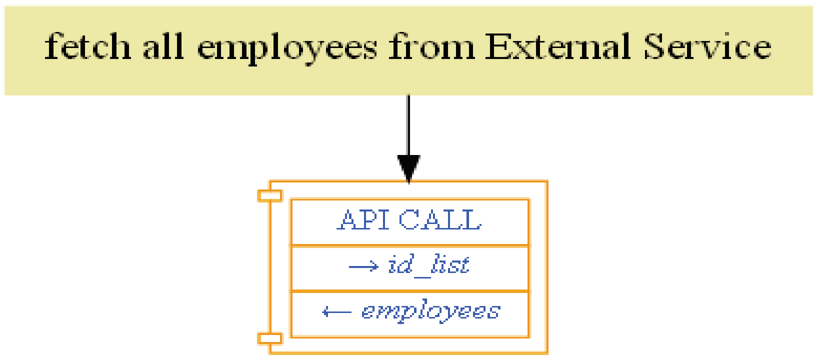

Figure 17. External Call Representation

4.3.3. Defining Architectural Event Relationship

In microservice-based architectures, defining and managing the relationships between microservices is crucial for maintaining a robust and scalable system. These relationships are often mediated through events, which act as the primary means of communication between services. By leveraging an event-driven approach, microservices can achieve loose coupling and high cohesion, enhancing their ability to scale and evolve independently.

Events represent specific occurrences or triggers within the system, such as user actions, system conditions, or other service outputs. Each event can initiate one or more processes, leading to the generation of subsequent events upon the completion of these processes. In an eEPC diagram, a service might publish an event when it completes a some business logic. This event can then be subscribed to by another service that needs to take action depending on the outcome. By clearly defining these relationships, we can map out the flow of data and control across the entire system, ensuring that each microservice reacts appropriately to the changes and events within the environment.

Defining an event in Docupyt also employs an architectural overview. Several main diagrams represent individual microservices. Events defined in those main diagrams belong to those main diagrams, but special tokens attached to the events make those microservices connected. Therefore, an event that triggers a series of actions is defined in a main diagram, other main diagrams (microservices) who subscribe to this event are chained together in architectural output of Docupyt. This means Docupyt creates two kind of outputs:

1. Individual Microservice Diagrams
2. Architectural Diagram

As mentioned above, architectural diagram is created by extracting and recognizing events in main diagrams and connecting those. An example of architectural usage was simplified and demonstrated below.

./service1.py

Diagram1

event: creates order publishes: orderCreated

./service2.py

Diagram2

subscribes: orderCreated

Which creates the output below;

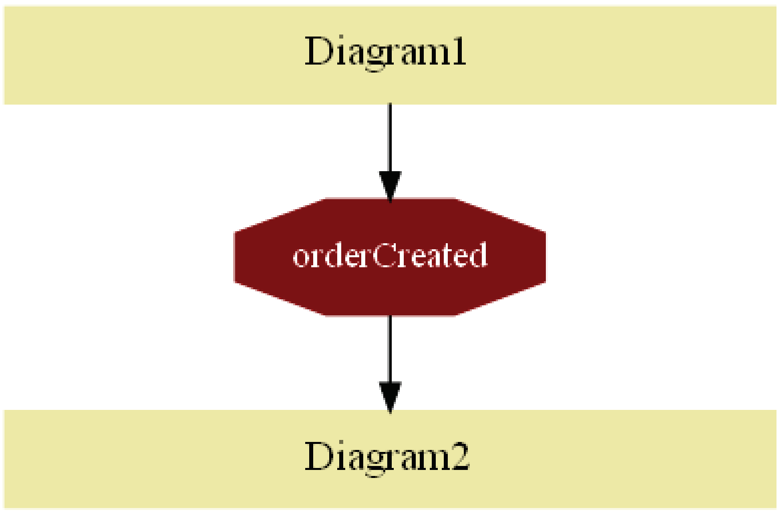

Figure 18. Architectural Representation

CHAPTER 5

CASE STUDY

The Kafka Streams Microservices Example Project by Confluent[16] serves as an exemplary model for constructing microservices using Kafka Streams. This project showcases the development of a robust and scalable architecture, leveraging Kafka's capabilities for real-time stream processing. The microservices in this project include Order Service, Inventory Service, Payment Service, Fraud Service, and Email Service. Each service is independently deployable, ensuring scalability and fault tolerance.

Order Service manages the creation and validation of customer orders, ensuring that each order meets predefined business rules before being processed further. Inventory Service updates inventory levels in real-time, preventing stockouts and ensuring accurate stock management. Payment Service handles the processing and validation of customer payments, interacting with financial institutions to secure transactions. Fraud Service monitors for potential fraudulent activities, utilizing real-time data analysis to detect anomalies. Email Service sends notifications related to orders, payments, and other events, keeping customers informed throughout their interaction with the system.

The Order Service manages the creation of orders. Once order is created it publishes events to the "orderCreated" topic. The Inventory Service updates stock levels in response to these events, while the Fraud Service checks for fraudulent activities subscribing to the same topic. The Order Details Service also subscribes to the “orderCreated” topic and enriches order data, and the Email Service sends notifications based on events from the orders. There’s also another topic named as “Payments”, generated by Payment Service. Whenever a payment is created, after processing the order a “paymentCreated” event occurs. Email Service also listens “paymentCreated” event to inform the related customer about the latest payment status. This setup ensures a scalable, robust system with loosely coupled microservices communicating via Kafka, enabling real-time data processing and efficient service coordination.

We have applied Docupyt to the codebase and modeled the system with Docupyt. Figure 18 represents the diagram drawn by the codebase authors inside the codebase[17].

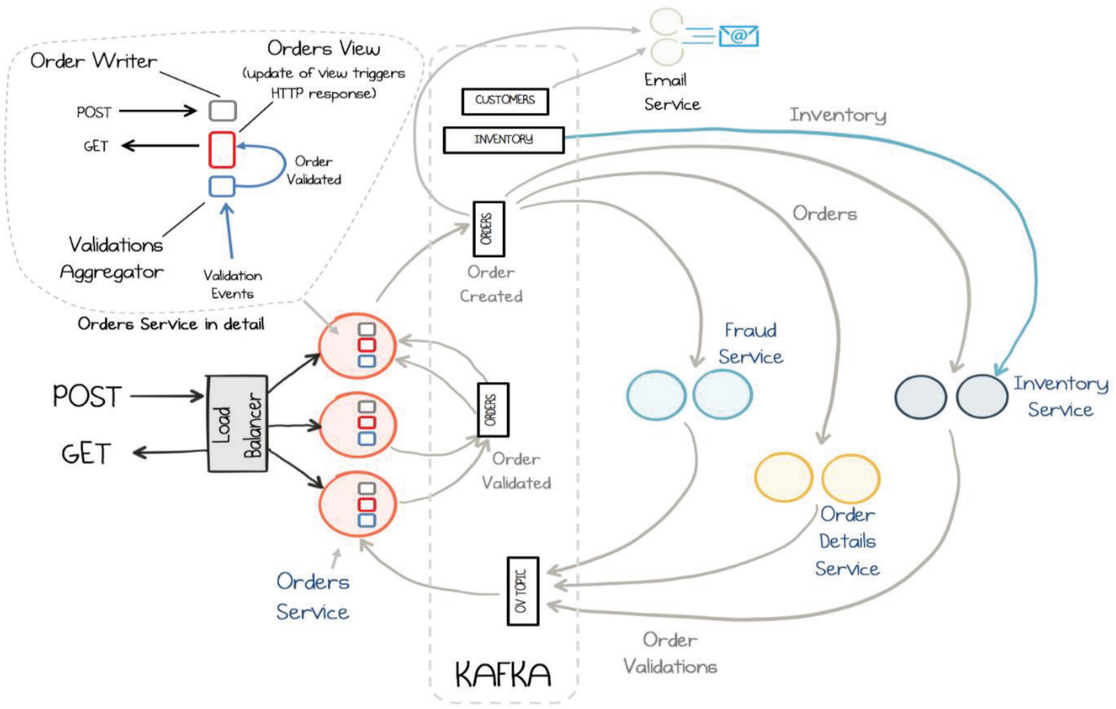

Figure 19. System Diagram

After modeling this microservices with Docupyt, example individual services that are selected to be shown, and general architecture diagram is shared in Figure 19.

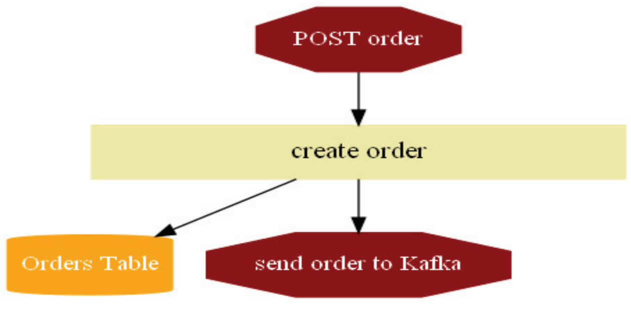

Figure 20. Order Service Diagram

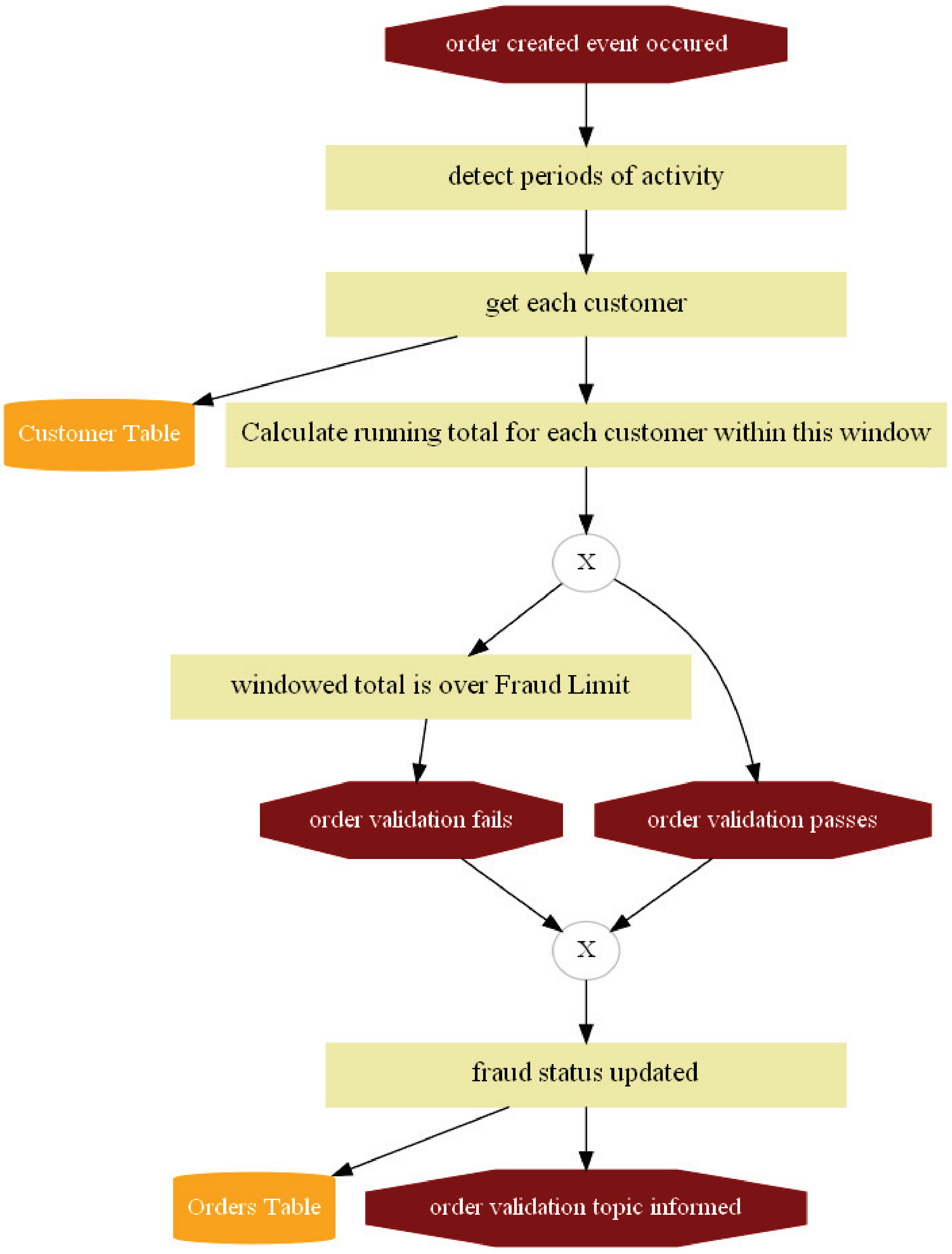
order created event occured
detect periods of activity
get each customer
Customer Table
Calculate running total for each customer within this window
X
windowed total is over Fraud Limit
order validation fails
order validation passes
X
fraud status updated
Orders Table
order validation topic informed

Figure 21. Fraud Service

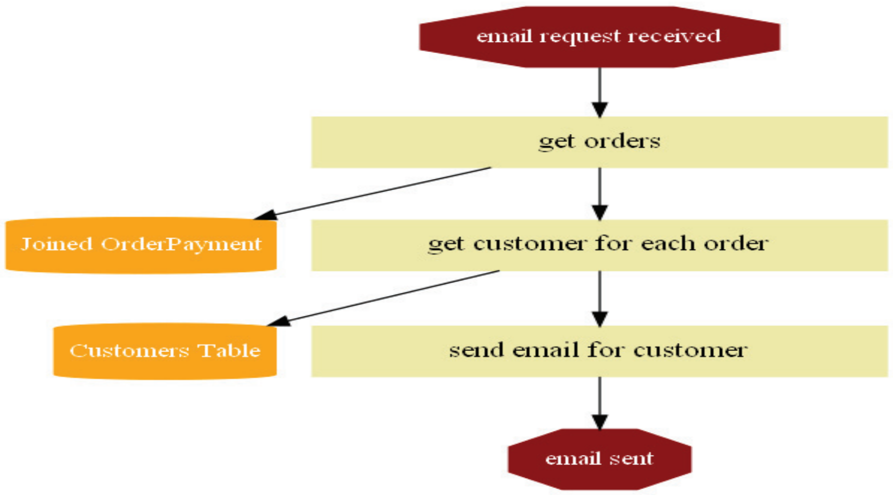

Figure 22. Email Service

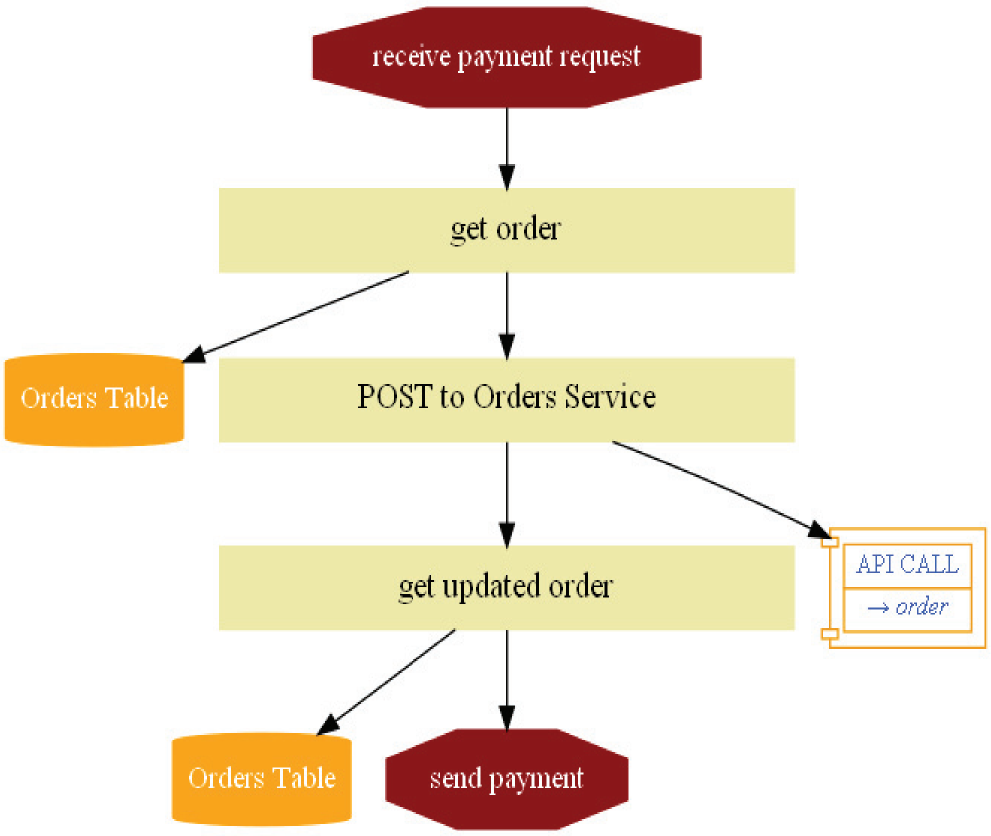

Figure 23. Payments Service

And the general architecture diagram which is comparable with Figure 18 which is drawn by the owners is shown below.

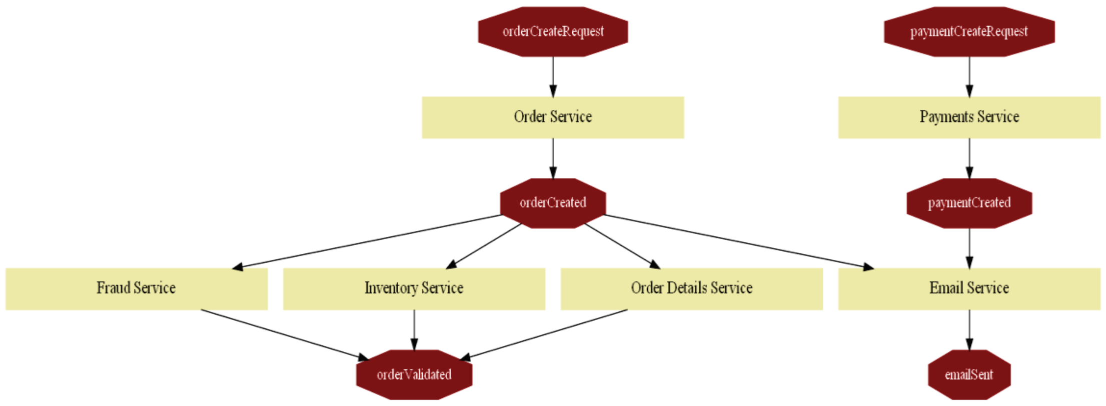

Figure 24. General Architecture Diagram

CHAPTER 6

CONCLUSIONS & FUTURE WORK

The modeling of microservice-based applications using various tools and methodologies provides significant insights into the architecture, behavior, and interaction patterns of microservices. This study highlights the critical aspects of defining and managing the relationship of microservices, emphasizing the importance of an event-driven approach for achieving loose coupling and high cohesion among services.

The integration of tools like Docupyt, which facilitates the creation of both individual microservice diagrams and comprehensive architectural diagrams, emphasizes the necessity for clear visualization in managing complex microservice environments. By tokenizing and parsing events and their relationships, Docupyt ensures that microservices react appropriately to changes within the system.

Additionally, the exploration of various software visualization tools such as AppMap, Code2Flow, Mermaid, and Diagrams reveals the diverse capabilities and limitations of each tool. AppMap's ability to capture real-time interactions within an application and visualize them in an interactive format offers valuable insights for debugging, performance optimization, and architectural understanding. However, it lacks certain filtering criteria which are present in other tools. On the other hand, Code2Flow, Mermaid, and Diagrams provide alternative approaches for visualizing logical flows and architectural details, with Diagrams particularly excelling in representing architectural criteria despite its limitations in business logic representation.

The relevance of eEPC (Extended Event-Driven Process Chain) notation in microservice modeling is another important point. Its event-centric approach aligns well with the principles of microservice architecture, facilitating the identification and decomposition of microservices based on event triggers. This approach not only enhances the clarity of microservice responsibilities and interactions but also supports the implementation of asynchronous communication mechanisms essential for maintaining loose coupling and autonomy among microservices.

Moreover, the comparison between eEPC and traditional Object-Oriented Analysis and Design (OOAD) approaches highlights the advantages of process-centric methodologies in modeling distributed and event-driven microservice architectures. While OOAD excels in data modeling and encapsulation, it often struggles with the distributed nature of microservices, making eEPC a more suitable choice for capturing the dynamic interactions and dependencies within a microservice-based system.

In conclusion, the effective modeling and visualization of microservice architectures are essential for managing their complexity and ensuring their scalability and robustness. Tools like Docupyt and notations like eEPC provide valuable frameworks for documenting and designing these systems.

The exploration of future work in the domain of microservice-based application modeling presents numerous promising avenues for enhancement and innovation. One significant area for further research involves the automation of token insertion in codebases when using tools like Docupyt. Currently, the process of embedding tokens into the code is manually executed, which can be labor-intensive and prone to human error. Developing an AI-based system capable of automatically identifying appropriate locations for token insertion and embedding them within the code would significantly streamline this process. Such a system could leverage machine learning algorithms to analyze code patterns and contexts, ensuring accurate and efficient token placement, thereby reducing the manual effort required and minimizing potential inaccuracies.

Another critical area for future investigation is the incorporation of database operations and API calls within the service boundaries illustrated by Docupyt. By accurately depicting these elements, it becomes feasible to apply measurement frameworks such as COSMIC (Common Software Measurement International Consortium) and Eventpoint for size estimation. COSMIC provides a standardized method for measuring software functional size based on its functional user requirements. Enhancing Docupyt by calculating size estimations, with clearly defined service boundaries, would allow for more precise and meaningful size measurements, aiding in project estimation, resource allocation, and overall project management.

Expanding the variety of tokens used in Docupyt also represents a valuable direction for future work. By introducing a broader range of tokens, encompassing different types of interactions, events, and dependencies, the modeling tool can provide a more detailed and nuanced representation of the microservice architecture. Docupyt can create different types of diagrams for a variety of teams in a corporation. A detailed service diagram can be provided exposing database connections which can be useful for the development team, but this can be unnecessary for the sales team. Docupyt can behave differently and not expose those details and can provide a more high-level requirements-based diagram for the sales team.

While code-in documenting, Docupyt can also produce test cases using possible paths created on the diagram.

In conclusion, future work in the realm of microservice modeling should focus on automating token insertion, incorporating comprehensive measurement frameworks, and expanding the variety of tokens. These advancements will enhance the effectiveness, accuracy, and utility of microservice modeling tools, contributing to the development of robust, scalable, and efficient microservice-based applications.

REFERENCES

1. M. . -A. Storey, C. Best and J. Michand, "SHriMP views: an interactive environment for exploring Java programs," Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, Toronto, ON, Canada, 2001, pp. 111-112, doi: 10.1109/WPC.2001.921719.
2. J. I. Maletic, J. Leigh, A. Marcus and G. Dunlap, "Visualizing object-oriented software in virtual reality," Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, Toronto, ON, Canada, 2001, pp. 26-35, doi: 10.1109/WPC.2001.921711.
3. S. C. Eick, J. L. Steffen and E. E. Sumner, "Seesoft-a tool for visualizing line oriented software statistics," in IEEE Transactions on Software Engineering, vol. 18, no. 11, pp. 957-968, Nov. 1992, doi: 10.1109/32.177365.
4. GitUML. www.gituml.com. (accessed 2024-04-06)
5. PlantUML. www.plantuml.com. (accessed 2024-04-06)
6. AppMap IntelliJ Plugin https://github.com/getappmap/appmap-intellij-plugin (accessed at: 2024-03-21)
7. Mermaid Diagramming and Charting Tool. https://mermaid.js.org/. Last Accessed: (accessed 2024-04-06)
8. Code2Flow. https://code2flow.com/ (accessed 2024-03-21)
9. J. I. Maletic, A. Marcus and M. L. Collard, "A task oriented view of software visualization," Proceedings First International Workshop on Visualizing Software for Understanding and Analysis, Paris, France, 2002, pp. 32-40, doi: 10.1109/VISSOF.2002.1019792
10. H. M. Kienle and H. A. Muller, "Requirements of Software Visualization Tools: A Literature Survey," 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, Banff, AB, Canada, 2007, pp. 2-9, doi: 10.1109/VISSOF.2007.4290693.
11. U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt and A. Amjad, "A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code," in IEEE Access, vol. 7, pp. 158931-158950, 2019, doi: 10.1109/ACCESS.2019.2950884.

12. L. C. Briand, Y. Labiche and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," in IEEE Transactions on Software Engineering, vol. 32, no. 9, pp. 642-663, Sept. 2006, doi: 10.1109/TSE.2006.96.
13. G. Gharibi, R. Tripathi and Y. Lee, "Code2graph: Automatic Generation of Static Call Graphs for Python Source Code," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 2018, pp. 880-883, doi: 10.1145/3238147.3240484.
14. Diagrams - Diagram as Code. diagrams.mingrammer.com. (accessed 2024-04-06)
15. H. Unlu, S. Tenekeci, A. Yıldız and O. Demirors, "Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study," 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Palermo, Italy, 2021, pp. 244-251, doi: 10.1109/SEAA53835.2021.00038.
16. Confluent Inc. (n.d.). Kafka Streams Examples. https://github.com/confluentinc/kafka-streams-examples (accessed 2024-06-16)
17. System Diagram. https://github.com/confluentinc/kafka-streams-examples/blob/master/src/main/java/io/confluent/examples/streams/microservices/system-diag.png. (accessed 2024-06-16)