

Historisierung von konzeptionellen Modellen unter Verwendung von Wissensgraphen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Mustafa Mohammad, BSc

Matrikelnummer 01528066

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Mitwirkung: Associate Prof. Dr. rer. nat. Simon Hacks, M.Sc., B.Sc.

Wien, 11. August 2024

Mustafa Mohammad

Dominik Bork

Historization of Conceptual Models Using Knowledge Graphs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Mustafa Mohammad, BSc

Registration Number 01528066

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Assistance: Associate Prof. Dr. rer. nat. Simon Hacks, M.Sc., B.Sc.

Vienna, 11th August, 2024

Mustafa Mohammad

Dominik Bork

Erklärung zur Verfassung der Arbeit

Mustafa Mohammad, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. August 2024

Mustafa Mohammad

Acknowledgements

I would like to give my thanks from the bottom of my heart to my supervisors Dominik Bork and Simon Hacks. Thank you for guiding me through this thesis with your help and support. I wouldn't have succeeded in it without the knowledge you shared and the direction you showed.

I further thank my parents and siblings whose love has been limitless as well as their encouragement that has always acted as a safe haven for me during this difficult time. Your trust in what I can do has been unwavering.

I would like to express my deepest thanks to my wife. Your patience, understanding and love have given me the strength not to give up. Thank you for being there for me, and never doubting in me.

Lastly, I am grateful to my friends too, who have been with me in this academic journey through the ups and downs. It has been fun and memorable because of your company and consistent support.

Thank you very much for your guidance, encouragement as well as dedication. I could not have completed this thesis successfully without you; moreover, it has also fired me up towards realizing my educational and career goals with great zeal.

Abstract

The goal of this thesis is to examine how Conceptual Models (CMs) can be historized through the use of Knowledge Graphs (KGs) to manage their evolution more effectively. The study develops and evaluates a prototype which enhances semantic representation and querying for model changes in a dynamic nature unlike traditional version control systems. Through systematic testing, the prototype demonstrated enhanced traceability and manageability of model versions. These findings reveal that KGs are good at dealing with complex evolution of models, although there are still some difficulties, such as feature integration and transformation complexity. This contribution advances the field of Version Control Systems (VCs) by demonstrating where KGs might be useful in the historical representation of CMs and suggesting directions for future enhancements.

Contents

Abstract	ix
Contents	xi
1 Introduction	1
1.1 Motivation	1
1.2 Aim of the Work	2
1.3 Research Methodology	4
1.4 Structure of the Work	5
2 Background	7
2.1 Information Technology Architecture ITA	7
2.2 Model Driven Engineering MDE	8
2.3 Conceptual Models CMs	9
2.4 knowledge graph KG	12
2.5 Version Control System VC	16
2.6 Software Architecture and Technology Stack	17
2.7 Software Process Model	25
3 Related Work	27
3.1 A General Theory for Evolving Application Models	27
3.2 Bridging the Gap between OpenMBEE and Git	29
3.3 Reflecting on the past and the present with temporal graph-based models.	30
3.4 Calculation and Propagation of Model Changes Based on User-level Edit Operations	31
3.5 A new versioning approach for collaboration in blended modeling . . .	32
3.6 Historization of Enterprise Architecture Models Via Enterprise Architec- ture Knowledge Graphs	34
3.7 From Conceptual Models to Knowledge Graphs: A Generic Model Trans- formation Platform	35
4 Historizing Generic Models	37
4.1 Requirements Engineering	37
4.2 Solution Design	42
	xi

5	Prototype Implementation	49
5.1	Software Development	49
5.2	User Manual	55
6	Evaluation	69
6.1	User Acceptance Testing	71
6.2	Use Case Scenario	77
6.3	Performance Evaluation	80
7	Conclusion and Future Work	83
7.1	Discussion	83
7.2	Future Work	84
7.3	Conclusion	84
	List of Figures	87
	Glossary	89
	Acronyms	91
	Bibliography	93



Introduction

This chapter outlines the research's motivation and clarifies the problem area. Next, the research's objectives are defined, followed by a structuring outline for subsequent work.

1.1 Motivation

In the fast-changing world of Information Technology (IT), how a company manages its Information Technology Architecture (ITA) is crucial to its success. A key aspect of this management is ensuring that the technological efforts are directed effectively, which often involves dealing with complex systems and models [Lan18]. As software development processes evolve, there has been a notable shift towards Model Driven Engineering (MDE) methodologies. MDE represents an innovative approach that uses models as fundamental artifacts throughout the entire software development lifecycle [FS04a].

The representation of heterogeneity in architectural models is facilitated through CMs, which are important in both ITA and MDE. CMs are important tools for designing complex domains and illustrating their basic structures, relationships, and behaviors. These models are widely used in various areas such as information systems, software engineering or knowledge management as they allow gaining deep understanding of the complex domain and helping to communicate about its underlying complexity effectively [Oli07].

CMs are like source code, which have many changes during their development life cycle and are managed by multiple users. Therefore, it becomes necessary to integrate them into VCs, allowing parallel modifications by multiple users and ensuring detailed tracking of their evolving history. Traditional VCs such as [Git] can be a solution. However, the competence of traditional VCs developed for text files is limited in this context.

Traditional VCs function mainly at the file level and are based on line-focused text comparison. However, when interacting with the textual representation of visual models, this approach becomes insufficient and impractical for effectively visualizing and managing the models, especially as they grow in size and complexity. Furthermore, alternative advanced VCs often force specific modeling formats, limiting their applicability. However, given the diverse types of models available, enforcing a standardized formatting approach across different domains and modeling languages becomes challenging [ABK⁺09, BKL⁺12].

Below is a list of limitations that current VCs have when applied to conceptual models:

1. Large model files stand a challenge in VCs due to their size. Textual VCs are not made to manage such large files efficiently. Thus, persisting and handling these model files within a VC repository can become difficult and slow.
2. Textual VC are not well designed for efficiently comparing models. These systems are developed to track textual modifications, such as documentation or code, rather than numeric differences in model parameters. Therefor, text based VCs struggle to provide meaningful information or comparisons between different versions of models.
3. Textual VCs have limitations when it comes to visually representing complex model architectures. Models consist of cascading layers, connections, and parameters that cannot be appropriately visualized in text based systems. As a result, understanding and visualizing the evolution of a model's architecture over time becomes challenging within these systems.
4. Parallel modification in textual VCs are less efficient when it comes to models. The large size of model files and the absence of visual model diffing capabilities make modification more challenging.
5. The heterogeneity among different types of models poses challenges towards their integration into VCs thereby making it hard to have uniformity on how versions should be handled consistently and efficiently throughout different stages of development life cycle for any given software system.

Addressing these limitations requires a re-evaluation of the suitability of VCs for CMs, highlighting the need for solutions that can meet the unique requirements of managing, storing and tracking the evolution of CM in a way that aligns with the complexities and diversity of ITA and MDE.

1.2 Aim of the Work

Knowledge graphs have become very popular as a way of representing and organizing massive amounts of information. These data structures represent knowledge in the form of a graph with nodes, edges, and properties. Knowledge graphs are a structured

and semantic method for organizing and linking information that enables meaningful representation, reasoning over, and retrieval of knowledge [CJX20].

Utilizing knowledge graphs in the GraphML format is in accordance with international standards recognized for their broad acceptance across systems. This format is selected to effectively manage the integration of diverse model types, a challenge documented in existing research. Smajevic et al.'s work provides a way how to convert conceptual models into knowledge graphs [SB21a] sharing with us a building framework which gives step-by-step guidelines for transforming them into our required format while ensuring consistency across different elements of design solution.

Bratfors et. al. work [BHB22], seeks to give a history of Enterprise Architecture (EA) models with KGs which record changes' timing and nature. However, it fails to consider important functional features. Instead, he adds history metadata during transformation process only and requires pre-configuration before model commits.

In this regard, we suggest an automated system for recording any conceptual model that can be expressed as a knowledge graph. Our system makes sure that every committed model is integrated into the historical context immediately after it has been made in a stateless manner, without the need of manual configurations thus saving time and enhancing functionality.

To enhance user understanding of modifications made on models, our goal is to offer text level differences at the XML level in Graph Markup Language (GraphML) as well as visual representation at the node level. Such an approach will make the historization system more usable.

Additionally, we emphasize the need to efficiently handle GraphML files. To achieve this, we will employ graph databases that are capable of speeding up historical querying and data management in our case. Graph databases are designed for dealing with complicated relationships among large datasets thereby supporting dynamic querying plus strong tracking of history [Liu14]. Rather than storing information about entities or objects alone like relational DBMSs do [Cha76]; these systems also store their interconnections (edges) alongside properties (vertices). A query language such as one supported by graph databases can then be used for accessing various paths within such a graph structure.

We propose the integration of knowledge graphs into the version control system to enhance the versioning and management of conceptual models. By leveraging knowledge graphs, we aim to capture the complex semantics and relationships inherent in these models, thereby enabling more effective tracking and controlling of model artifacts. Importantly, the expected result of this research is **a prototype that supports the generic historization of any arbitrary conceptual model represented as a knowledge graph**. This prototype will offer the flexibility to accommodate a wide range of modeling languages and formats, as long as these models can be transformed into knowledge graph structures.

1.3 Research Methodology

The following is an overview of the proposed methodology for the master thesis, which incorporates a series of essential steps aimed at accomplishing the project's objectives.

1. Literature Review: The investigation should gather an comprehensive literature review of past studies on this subject. By doing so, it will establish a firm foundation for theoretical background knowledge.
2. Research Design: A qualitative grounded theory [Kha14] approach will be adopted for this study. Grounded theory is appropriate because it enables one to explore and build theories from the data hence fits well with historizing conceptual models with the aim of gaining new understanding.
3. Sampling: In this research, convenient sampling [Eme15] is going to be applied in selecting conceptual models. This will involve two sources namely: Conceptual models which can be translated into knowledge graphs through CM2KG tool and preexisting knowledge graphs that already represent conceptual models. The reason for using convenient sampling is because it is an easy way of choosing what one needs, so the most suitable conceptual models for historization study will not miss out.
4. Requirements Engineering [Mac12]: is focused on determining the necessary features of the version control system, such as history tracking, change monitoring, history's metadata. We will also document the conditions for recording and managing the historical context of conceptual models in a knowledge graph representation. These specifics should be written as user stories which can then be plugged directly into our tasks in git. Please note that this is an iterative process where different requirements get integrated at different times throughout the development. Initially we start with historization and later on other features must be brought in step-by-step until they build together the promised effective prototype.
5. Architecture Design [EK03]: The main objective of this step is to come up with a design for the prototype. This means integrating knowledge graphs based on collected needs. Identifying the major parts of the system and how they interact with each other, as well as selecting appropriate software, tools and programming languages used to develop it are also included in this phase. These steps provide together strong grounds for prototype development which acts as starting point towards realization of research goals stated within this thesis statement.
6. Development: the version control system will be implemented as per the architectural design and requirements. It entails coding for functionalities that are needed while ensuring compatibility with selected technologies and programming languages. Further to this, tests shall be done together with debugging so that any arising problems or bugs can be detected and fixed. Adoption of an iterative approach to

development like agile method (Scrum) [Sch97] allows for continuous improvement on product features as well as adaptability at different stages in the software life cycle.

7. Evaluation and Discussion: Involves testing the developed prototype and provide a use case scenario to cover all functional requirements. The level of generality in accommodating arbitrary conceptual models will be discussed, and functional requirements of the prototype will be evaluated. Limitations and potential areas for future improvement will be identified based on the insights gained from the evaluation of the developed prototype.

1.4 Structure of the Work

The thesis is structured as follows:

Chapter 2 Background:

Presents the theoretical foundation and fundamental information that supports the proposed thesis. This chapter gives an outline of key ideas, including ITAs, MDEs, CMs, as well as KGs, and model historization.

Chapter 3 Related Work:

Provides an overview of previous CM historization research and practices. It investigates existing techniques and contextual analysis that catch the historical evolution of these models. This chapter provides useful context for the subsequent chapters' creation of a new historization prototype, highlighting the advantages and disadvantages of previous methods.

Chapter 4 Historizing Generic Models:

In this chapter, we describe how we present the solution and ensure that generic models can track their changes over time. We also detail the construction of the solution to meet the primary objectives of the thesis. This critical step forms the basis for the development of the prototype.

Chapter 5 Prototype Implementation:

This chapter explains how the prototype for historizing CM with KG was developed. Covering the engineering requirements, the software process model, the design and technology choices. The chapter also details the prototype's interface including its API and web UI.

Chapter 6 Evaluation:

This chapter presents User Acceptance Testing and a use case scenario to check the functional ability of the prototype to historize CM. It starts with User Acceptance Testing

to verify that the prototype meets the requirements of the stakeholders. Then, it explains how a use case scenario can be used to show practical functioning of the prototype in historizing CM, thereby indicating operational efficiency and user interface design aspects.

Chapter 7 Conclusion and Future Work:

This chapter summarizes the research findings, highlighting the prototype's capability to historize CM and its implications for MDE. It discusses the prototype's strengths and limitations and proposes directions for future research to enhance its functionality. This section aims to provide a clear roadmap for ongoing advancements in CM historization.

CHAPTER 2

Background

2.1 Information Technology Architecture ITA

In managing the complexity of any large organization or system and achieving success, a master plan is essential, and this can be realized through architecture. But what precisely does the term “architecture” entail? As per the standard ISO/IEC/IEEE 42010:2011, the definition of architecture [Lan18] is:

Theorem 2.1. *Architecture: fundamental concepts or properties of a system in its environment, embodied in its elements, relationships, and the principles of its design and evolution.*

The definition has a broad scope that covers both the initial design of the system and how it can be changed with time. This makes it helpful in various fields, thus ensuring that systems are understood and developed in complete wholes. It underscores the importance of taking into account both internal components and external influences, which is crucial for the sustainable management and evolution of a system.

Under the information technology field, there are various types of architectures, each of which has a specific purpose and addressing various aspects of business and technology requirements. Here are some of the main types...

- EA [SR07]: provides a holistic design of an organization’s strategy, processes, information, and IT assets. EA’s goal is to align the vision of the organization with technology solutions to improve efficiency and effectiveness.
- Infrastructure Architecture [Laa17]: provides the view of hardware and software components, such as servers, storage, network devices, and data centers, that support an organization’s IT environment. The aim is to ensure that the IT infrastructure can support current and future business needs.

- Application Architecture [WBFG03]: gives the design of individual applications. Including the definition of processes and solutions that meet all of the technical and operational needs. It also focuses on optimizing quality metrics like performance, security, and maintainability.
- Security Architecture [EE05]: the purpose is to design, build, and manage a safe application environment and infrastructure. This covers operations, technologies, and policies designed to protect data from threats and vulnerabilities.
- Network Architecture [McF76]: provides the topology of the network, including hardware, software, communication protocols, and transmission modes of data.
- Cloud Architecture [Var08]: indicates the components required for cloud computing, these are made of frontend platforms, backend platforms, network platforms infrastructure, etc.
- Data Architecture [SK77]: provides the design, structure, and manages an organization's data. Providing a view of how the data are stored and integrated.
- Software Architecture: it is the top-level structure of software systems. It involves setting up the processes and standards utilized in software development.
- Service Oriented Architecture (SOA) [LL09]: A software design paradigm in which application components utilize a communication protocol over a network to provide services to other components.
- System Architecture [Man93]: is a general term for a range of architectural styles that describe the overall design of the system. It describes how different IT procedures and components interact in an ecosystem to fulfill business requirements.

Each of these architectures plays a vital role in an organization's overall IT strategy and helps to achieve business goals by utilizing technology effectively and efficiently.

2.2 Model Driven Engineering MDE

Software engineering has undergone a significant evolution over the past four decades, empowering developers to build more intricate and dependable software-intensive systems [FS04b, Fav04]. This progress can be attributed to three primary categories of techniques: explicit development process outlines, abstraction enhancement techniques, and testing and software verification. MDE has emerged as a framework for precisely defining techniques, constructing systems at various abstraction levels, and automating testing and validation. MDE emphasizes the use of intelligible models that can exist at any abstraction level, offering tools for model refactoring, refinement, and code generation. MDE processes involve iteratively refining abstract models into concrete ones, binding them in the automatic generation and deployment of complete code.

Computer Aided Software Engineering (CASE) tools have been developed to simplify software development and maintenance but still face challenges in tool selection, abstraction level maintenance, platform support, and feature diversity [Fis88]. Long-term system life cycles and the need for collaboration among various CASE tools necessitate standardized compliance standards across software development life cycles. The Object Management Group (OMG) initiated the Model Driven Architecture (MDA) initiative to address some of these challenges, which is a specific framework within MDE focusing on separating concerns related to platform independence and specific implementations. However, MDA is still evolving, with certain requirements lacking clarity or completeness.

MDE can benefit developers across different domains, but it cannot accommodate all system development scenarios. This introduces the concept of reusable MDE components for defining MDE processes and summarizes MDE activities by methodologists and developers [MD08].

Model-driven activities involve a series of steps for defining and implementing an MDE process. MDE developers primarily design models, adhering to a specified methodology that defines each step and the relationship between the models at different abstraction levels. The flexibility of MDE allows localized changes, but challenges persist in applying MDE-inspired methods to large projects involving multiple developers and tools, defining an MDE methodology requires significant effort and expertise in various domains and platforms. A clear understanding of MDE components, including the sequence of abstraction levels, modeling languages, refinements, and mappings, is essential for a successful MDE process definition.

2.3 Conceptual Models CMs

Over the past decade, "conceptual modeling" has gained popularity across various domains. This is evidenced by the number of papers and the increasing research activity on this subject [Oli07, Par15]. Consequently, there are multiple approaches and angles to conceptual modeling which explains why there is such a wide interpretation spectrum. On one hand, this diversity can stimulate constructive debate and enhance the profession; however, it can also lead to confusion.

This greater interest in scientific communities is shown through the increased literature, including books, journal papers, and conference presentations. Alternatively, differences in perceptions about what conceptual modeling entails have been identified. These disparities are common within forward-moving fields or well-established ones. Therefore, the aim of this dialogue is to clarify these conflicting opinions.

The benefits of having a conceptual model rather than not having them become apparent when compared with situations where they are not present at all [RAB⁺15]. Conceptual models give an organized framework for understanding complex systems or problems thereby facilitating clearer communication and more effective problem solving.

Various software systems include stakeholders from diverse backgrounds and fields [RW11]. It is typical to find different levels of understanding and expectation among these stakeholders. In the absence of a conceptual model as a common reference point, there may be a fundamental divide between what the project is about and how the stakeholders conceive it or contribute to its realization.

Lack of clear objectives and consensus among stakeholders leading to possible misalignments of project goals. Inappropriate problem descriptions prevent informed decision-making.

Software developers or programmers may have to take crucial decisions without enough input from the stakeholder, resulting in less than perfect results. The lack of an underpinning conceptual framework often means that there are divergent opinions on key outputs produced by projects. The project results cannot be validated or verified because they do not adequately integrate the insights and expectations of the stakeholders within the project framework.

A concept model is an abstract depiction that people use to understand, communicate or solve a problem [RAB⁺15]. By emphasizing key links between subsystems, it shows where the system boundaries are and allows for more detailed learning of complex systems by learners. In addition, this serves as an important documentation. It is the initial model that will be refined later on after discussions among software development teams, end users and managers who also seek to analyze possible scenarios and lay out some assumptions. Consequently, this keeps objectives of the project throughout its process of realization. The aim of conceptual modeling is to effectively manage complexity when implementing complex systems, which helps to understand and predict how people, technologies, and processes interact within a system.

A conceptual framework is a top-level representation of a concept, system or challenge not tied to any particular software or implementation tool. It should inform the methodologies and approach to the project, rather than being restricted by certain tools and technologies. The relationship between a conceptual model and architecture, as possibly discussed in David Powell et al.'s article [PSA⁺01] likely indicates that the conceptual model provides a high-level overview of the system's architecture. The conceptual model may then be taken into consideration when outlining the key principles and objectives of the architecture, transforming them further into detailed technical designs. This is important because it establishes that how theoretical issues are worked out through practical means resulting in corresponding developments made on systems being designed. In summary, the conceptual model shapes and ensures that architectural approaches fully integrate both theoretical knowledge and system design requirements.

Unified Modeling Language UML

Unified Modeling Language (UML) diagrams take CM abstractions and develop them into detailed, technical blueprints for implementation. This relationship ensures a smooth transition from business requirements to technical specifications.

UML is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers specify, visualize, construct, and document the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems [BRJ99].

UML is used in various phases of software development and system design. Requirement gathering, where UML's use case diagrams are particularly useful in understanding and communicating system requirements and user interactions. System and software design, where UML provides a wide range of diagrams to capture static (class diagrams, object diagrams) and dynamic aspects (sequence diagrams, state diagrams) of software design. Implementation, UML helps in defining artifacts (components, nodes) that are directly mapped to languages like Java, C++, etc. Testing and deployment, UML diagrams can represent deployment and component diagrams which help in understanding the deployment environment and system configuration [Fow03].

The class diagram in Figure 2.1 describes an example where students are enrolled in a subject [McG01]. The relationship 'enrolled in' is a counter of how many persons are in the list and has cardinality of 1..* each student being able to be enrolled in one or more subjects. Students can either be Honours or Pass and every Honours student is expected to work on one Honours project represented by the 'works on' relationship with cardinality 1.

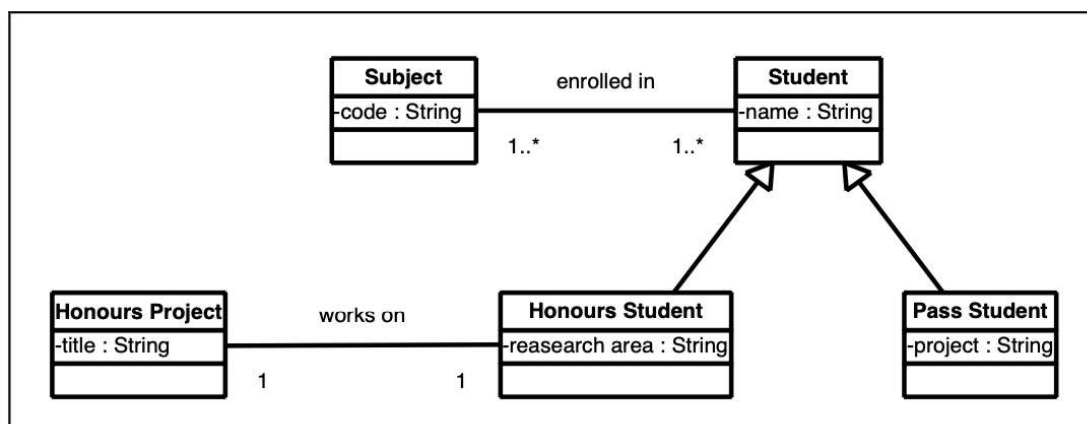


Figure 2.1: UML Class diagram example.

UML is a language that may be used to create conceptual models, which is how UML and conceptual models are related. Although conceptual models concentrate on the system's high-level structure without going into implementation specifics, UML offers the standard notations and semantics needed to represent these models. UML diagrams, such as class diagrams, are frequently used in conceptual models to represent entities and their relationships or to show system interactions from the point of view of the user.

2.4 knowledge graph KG

In the paper by Ping Zhang and Gary Chartrand [ZC06], a graph is described as a type of mathematical structure that explains how any two items within a group relate. Graphs consist of vertices (or nodes) and edges (or arcs) which link pairs of vertices. A graph is used to describe relationships between elements in various fields such as physical, biological, social or informational. Whereby cities, atoms or people are represented by vertices while the links between them can be directed or undirected to depict one-way or bidirectional connections respectively. This discipline categorizes many types of graphs such as simple graphs, multigraphs, weighted graphs and directed graphs for various purposes in order to explore structures of any kind of insights and applications to identify patterns, optimize processes, and solve computational problems.

Over the past years, knowledge graphs, or KGs, have quickly become a significant subject in various information technology domains. A directed, labeled, multirelational graph with some kind of semantics can be the simplest definition of a KG [Zaf21]. This has been aided in part by the structured information available online and the well-reported achievements of massive initiatives like the Amazon Product Graph and the Google Knowledge Graph. On the other hand, the cross-disciplinary nature of academic KG research is another element that has been equally important to the success of KGs but is less talked about.

Knowledge Graph is a flexible, reusable data layer used for answering complex queries across data silos. Using contextualized data that is represented and arranged in the form of graphs, achieving the highest level of connectivity. Designed to accommodate the dynamic nature of knowledge and effortlessly assimilate information, clarifications, and specifications.

When it comes to knowledge graphs, a “graph” refers to a specific way of organizing data that emphasizes the connections between data points through visual representations of nodes and links. In contrast, relational databases like Oracle or MySQL organize information using fixed table schemas, where data is stored in structured tables with predefined columns and rows. Relational databases are often used in scenarios where a strict and well-defined schema is needed to manage and retrieve data. The key difference lies in how each system handles relationships: knowledge graphs offer flexibility in representing complex and interconnected data, while relational databases rely on predefined structures [LSL19].

Relational databases are typically for stable enterprise systems with relatively simple relationships among data and with infrequently changing data structures. Conversely, graph databases (or graph representation within knowledge graphs) have more adaptability enabling complex relationship representations and dynamic data models that change with time. This distinction explains why knowledge graphs use graph-based models rather than traditional table-based ones especially where there is a need for changing relationships and linkages among the data points in the application area.

A knowledge graph is the best tool for enterprise data integration, since it can make context from the real world machine-understandable. The power of graphs to continually link concepts unifies data without altering the underlying data, as opposed to merging tables to integrate data. Data unification then creates a flexible data layer for the company by connecting data silos.

Graph Markup Language GraphML

One of the ways to represent knowledge graphs is by using GraphML, Graph Markup Language or GraphML is a feature-rich and easy file format made specifically for graphs called GraphML. It can describe the structure and contents of graphs because of its foundation in Extensible Markup Language (XML). GraphML supports directed, undirected, and mixed graphs in addition to hypergraphs, hierarchical graphs, and graphic representations. This format aims to simplify the exchange of graph data between programs and the storing of graphs [BELP10].

GraphML is mostly utilized in the following areas: Data analysis and visualization, where import and export of graph data is a common feature of network analysis tools and graph visualization software. Software engineering, the process of creating graphs that can be used to analyze software dependencies, architecture, and complexity for testing, documentation, and other reasons. Research areas, in which relationships and structures are best represented as graphs, such as in bio-informatics, social sciences, and physics. Web development areas, where GraphML is used to create flowcharts for navigation and model the connections between various online entities.

GraphML offers several main advantages such as: Interoperability, the interoperability of the system, software and platform is greatly enhanced by its foundation XML. Flexibility, graphs with complicated structures and properties can be represented by GraphML, along with other forms of graphs. Extension, GraphML's structure enables user-defined data to be included, which enables it to be tailored to the unique requirements of a broad range of applications. Tool support, GraphML is easily used in many tools for processing, analysis, and visualization because it is supported by a wide variety of graph-related software.

The GraphML snippet in Figure 2.2 from the paper [BELP10] represents a simple undirected graph with six nodes and seven edges enclosed within the root element '`<graphml>`'.

The GraphML schema provides definitions for keys related to both nodes and edges. In particular, there is a key with '`id="d0"`' for nodes, which enables the saving of the string attribute called '`color`', which takes the default value of '`yellow`' when not specified. Furthermore, another key identified by '`id="d1"`' has been defined for edges; it will save real number attribute named "`weight`" representing an edge weight.

In this graph, '`id="G"`' indicates that all edges are bidirectional by default. The graph has six nodes that are marked with different identifiers ('`n0`' – '`n5`') and also colors. These

2. BACKGROUND

colors include, node n0 is green while n2 is blue, n3 is red, n1 and n4 are by default yellow, and lastly node n5 is turquoise.

There are seven edges between these nodes whose unique identifiers and specific weights are indicated. The edge e0 in the graph connects node n0 to node n2 with a weight of 1.0; similarly, the edge e1 links node n0 to node n1 with a weight of 1.0. Furthermore edge e2 links node n1 to node n3 with a weighting of 2.0. Edge e3, e4, e5 connects nodes n3 to n2, n2 to n4, and n3 to n5. Finally edge e6 links n5 to n4 with a weight of 1.1.

This GraphML example effectively demonstrates how the detailed data structure of a graph can be represented including additional attributes on its nodes and edges. The approach is especially useful for processing or visualization in graph-manipulating software, as it provides standardized formats that improve accessibility and usability of graph data during computational analysis.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <key id="d0" for="node"
    attr.name="color" attr.type="string">
    <default>yellow</default>
  </key>
  <key id="d1" for="edge"
    attr.name="weight" attr.type="double"/>
  <graph id="G" edgedefault="undirected">
    <node id="n0">
      <data key="d0">green</data>
    </node>
    <node id="n1"/>
    <node id="n2">
      <data key="d0">blue</data>
    </node>
    <node id="n3">
      <data key="d0">red</data>
    </node>
    <node id="n4"/>
    <node id="n5">
      <data key="d0">turquoise</data>
    </node>
    <edge id="e0" source="n0" target="n2">
      <data key="d1">1.0</data>
    </edge>
    <edge id="e1" source="n0" target="n1">
      <data key="d1">1.0</data>
    </edge>
    <edge id="e2" source="n1" target="n3">
      <data key="d1">2.0</data>
    </edge>
    <edge id="e3" source="n3" target="n2"/>
    <edge id="e4" source="n2" target="n4"/>
    <edge id="e5" source="n3" target="n5"/>
    <edge id="e6" source="n5" target="n4">
      <data key="d1">1.1</data>
    </edge>
  </graph>
</graphml>
```

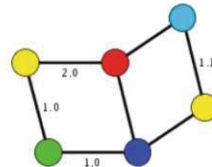


Figure 2.2: GraphML Example.

Neo4j

Neo4j is a database built on graphs that are highly efficient in accessing and managing interconnected data. To those applications which need to represent complex relationships and query them efficiently, it offers an excellent solution for managing a KG.

KGs mimic the way human beings store and recall knowledge by having a network of entities and interrelationships between them. It naturally conforms to KG structure because Neo4j is designed to handle nodes (entities), edges (relationships) and attributes (properties). This agreement implies that there is one-to-one correspondence between the abstract KG and how practical data is stored in the database thereby minimizing impedance mismatch that relational databases suffer from when dealing with complicated related information.

Given Below are some of the reasons why prototyping a knowledge graph system through Neo4j and Cypher language would be ideal:

Intuitive Data Modeling: Neo4j enables one to model your knowledge domain in a way that is intuitive. The graph model of the tool is whiteboard-friendly such that you store it in the database as you draw it on a whiteboard.

Performance: With Neo4j, you will experience efficient storage and retrieval of graph structures which may be much faster than relational databases, especially when it comes to associative datasets and larger volumes of networked data as well.

Flexibility: As knowledge evolves, so does the structure of KG. Therefore, it is ideal for prototypes requiring frequent changes and evolution because Neo4j's schema-optional approach allows for the easy addition and modification of nodes and relationships.

Cypher: Is a declarative graph query language that supports expressive and efficient querying of graph data. The major productivity boost while prototyping is achieved by its understandable syntax. In short, Cypher facilitates matching patterns of nodes and relationships, subgraph extraction, complex filtering through mutations as well as altering graph structures.

Neo4j's Ecosystem and Tooling: A majority of the tools available for Neo4j are used in the importation of data, management of databases as well as visualization of graphs. Neo4j comes with drivers that can be easily integrated into the prototype technology stack.

The Community and Support: There are online resources like documentation, tutorials, and forums, among other support options provided by commercial vendors with whom you can consult if you have inquiries or want to know more about something. When developing a prototype, this is very important because at times you need assistance or answers to queries urgently.

Historizing Changes: It is very important for historical reasons that what has been called KG is historized. For instance, there are built-in features that can be deployed in order to keep track of how the graph changed over time using Neo4j.

2.5 Version Control System VC

By using VC, developers can store earlier versions of project files as well as source code, which can be recovered later [Rup10]. It keeps the version information in a repository that encompasses the individual files and the entire project structure.

The concept of branches is at the core of version control systems and is crucial in both Subversion (SVN) and Git [PCSF08, BDW16]. A branch in simpler terms is a separate copy of your project at a particular point in time that you can work on without affecting the main codebase. This permits various developers to make multiple features, fixes or experiments without interfering with the others work. When you create a branch, it serves as creating a parallel version of the project that can grow independently.

Branches are so useful because they permit for instance, developers who simultaneously work on new feature or bug fix development. These branches allow for unsteady or experimental changes to be made without touching the stable principal codebase. Branches however help manage varying versions of software products. An example could be having one stable release branch for ongoing bug fixes and another separate development branch where new features would be added. In this way, production version remains robust while the development continues. Branches also serve to isolate changes which must be fully tested and reviewed before being merged back into the main line of development. The point here is to ensure incorporation of stable codes into the main project line.

The branching in Git is flexible and lightweight [BDW16]. For instance, branches can be easily created and merged which is important especially when dealing with collaborative projects that have a number of contributors. This flexibility makes it possible to develop open source projects or other environments where multiple people could be working on different parts of the project simultaneously.

Multiple branches may exist within the repository. This feature is useful when managing different releases of a project, for example, one stable release while continuing development. Branching also makes it possible to work on experiments separately from the main project referred to as sandboxing where test versions are not linked with the final ones. Additionally, within a branch, labels called tags can assist in pointing out certain releases or even future access to the latest development version. Besides this, there is an option for adding comments along with author name or ID signing off on them when committing changes. Thus, detailed change logging including comprehensive version tracking provides a rich history of modifications made over time, providing evidence of how the project evolved.

Various contributors can modify a file's detailed record at different times. Such records include deletions, insertions, and changes in files. However, the ability to track the renaming and moving of files differs from one VCs tool to another. Every historical reference should be accompanied with authorship details like their names, reason for change and date of change. The complete history is also very helpful when it comes to resolving problems in earlier versions of software packages and helps in determining the

cause of mistakes. Within the context of active development, every piece of source code will be seen as an 'older version' at some point or another.

Reliability is increased by annotating each change with a note that explains why it was made and the goal it was aimed to achieve. This helps in making well-informed changes as well as cohesive modifications that are consistent with the main design principles of the system; this is a detailed history to facilitate such changes. For instance, when developers are working on legacy code, they need a deep understanding of the existing codebase so as to make accurate future estimates for their work. A lot of care needs to be taken during software development to maintain precision.

The latter determination has been brought about due to the realization that developing software without version control could lead to major risks which no professional team would ever want exposed their projects into. Therefore, the question is not whether or not version control should be used but rather which implementation of this technology should be chosen out of many available options. At the same time, one might visit an option of learning other types of version control software too so that one may have a better knowledge about them all together.

2.6 Software Architecture and Technology Stack

This section highlights the importance of planning the architectural design and the technology choices.

2.6.1 The Significance of Software Architecture

Software architecture must be understood in order to build software systems. It supports the system's ability to adapt to changes and acts as a guide for the original design. It significantly affects the general quality, maintainability, development speed, and flexibility of the system in response to changing user requirements and technical breakthroughs [Has18].

A system succeeds or fails on the basis of how it is built. Picking up the right architecture will make the system works well from day one. Poor selection results in major problems. It's a simple equation; architecture choices significantly affect performance, reliability, and safety.

Software updates are smoother with solid architectural plans. Such plans simplify integrating many projects at once. That is because they define how software elements interact. Clear architecture also helps future updates happen easily.

Software architecture is very important for software development. It helps to ensure that the system works well. It also makes it easier to build the system. The architecture can be updated to meet changing needs. Software architecture influences project management. It is the driving force behind the technology used. To advance software engineering, we

need to enhance the process of creating architectural frameworks. This will help software projects succeed.

2.6.2 Design Solution

Architectural design patterns help making sense of complex systems. Examples such as multi-layer, client-server, event-driven patterns aid in understanding these systems. They provide templates for common problems. This makes maintaining and extending code simpler. Developers, architects, stakeholders all share a language, which makes patterns boost communication significantly [KMLS18, BR10].

Client-server architecture is important for networked computing. It allows applications like web browsers and databases to work. The client makes requests, and the server fulfills them. Clients are devices or programs that ask for services or resources. Servers are powerful computers that manage things such as files, printers, or networks. They wait for requests from clients and then provide what was requested [Kum19].

The architecture separates an application into two parts: clients and servers. Clients request services or resources from servers. Servers are dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers). They receive requests from clients and fulfill them. Devices or programs that request services or data are called clients. Some clients are Web browsers, email programs, or online games. Servers are hardware or software made to provide services or data to clients. Web servers, databases and file servers are some types of server.

Computers use various client-server architecture patterns. The one-tier architecture as shown in Figure 2.3 has the program, data, and user interface on one device. This is used for small standalone applications, such as personal databases. The two-tier architecture as shown in Figure 2.4 divides tasks between the client and the server. Typically, a client program talks directly to a database server. The three-tier architecture as shown in Figure 2.5 adds an intermediary layer, often called the application layer, which handles business logic and communicates to the database for the client. The N-Tier architecture as shown in Figure 2.6 extends the three-tier model with more layers, allowing better scalability, flexibility, and management. Each layer focuses on specific tasks, such as displaying content, processing application logic, or managing data.

The client-server architecture has many benefits. Centralized control keeps data storage and management focused, this boosts data integrity and security. Another key advantage is scalability, as this model easily expands for more clients. Plus, flexibility lets different devices access server services simultaneously. Also, server maintenance happens without affecting clients. This simplifies system updates. When building client-server systems, you may face several issues. Centralized resources are vulnerable to attacks. Although scalable servers have finite client capacity limits. The performance of the system depends greatly on the stability and speed of the network. The client-server architecture separates system requirements into parts. Clients request data or resources. The servers process requests.

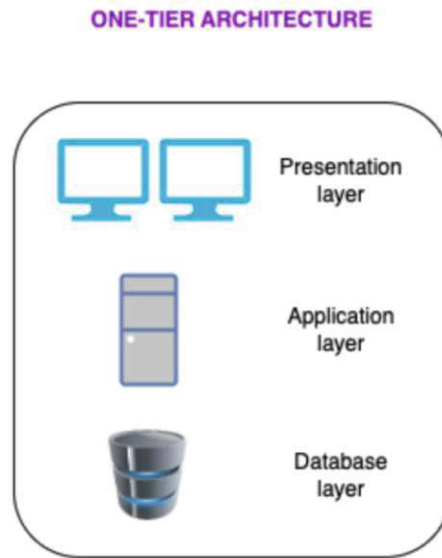


Figure 2.3: One tier client-server architecture

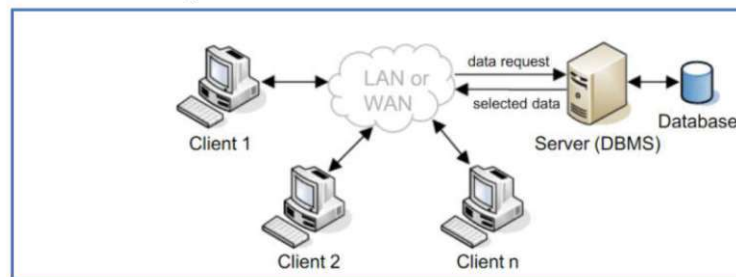


Figure 2.4: Two tier client-server architecture [Kum19]

This model works well for many apps. It offers a robust framework. Understanding client-server relationship is a key for developers, network designers, and IT planners.

The essential building block of Web communication is Hyper Text Transfer Protocol (HTTP). This protocol makes it possible to transport documents that are hypermedia, such as HTML. It has various uses in addition to interacting with servers and web browsers. HTTP has a traditional client-server architecture in which a client connects to a particular server via Transmission Control Protocol (TCP) in order to initiate a request. After responding to the client's request, the server cuts off communication.

A collection of methods defined by HTTP indicates the intended set of actions to be followed on a certain resource. Among these methods are: GET: This asks for the resource's representation to be provided. All that should be done with GET requests is

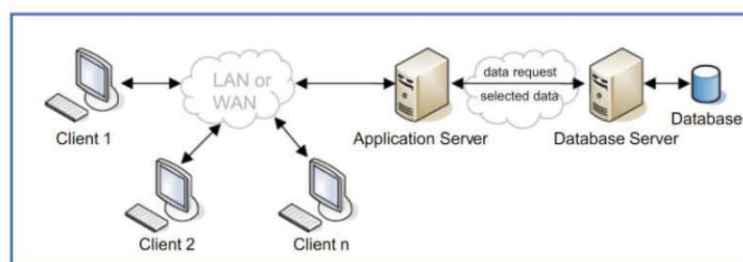


Figure 2.5: Three tier client-server architecture [Kum19]

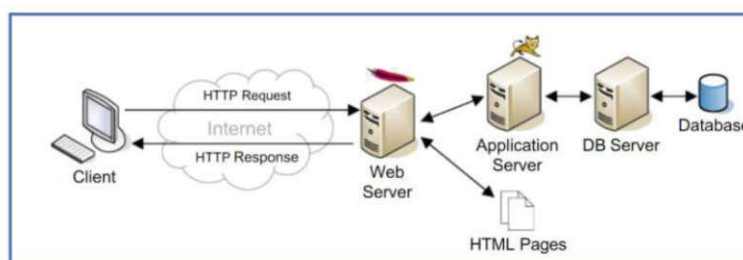


Figure 2.6: N tier client-server architecture [Kum19]

retrieve data; nothing more. HEAD: This is comparable to GET, but instead of requesting the response body, it requests the exact same response without a body. POST: When an entity is sent to the designated resource, the server may experience side effects or a state change. PUT: This updates the request data into place of all existing representations of the target resource. DELETE: This eliminates the given resource. PATCH: This makes some minor changes to a resource. TRACE: This runs a test along the path to the target resource, a message loop-back. CONNECT: This creates a tunnel to the server that the target resource has pointed to.

The following status codes show the success or failure of a HTTP request. They are separated into five groups: 1xx (Informational) - The request was submitted, and work is still being done on it. 2xx (Successful): The request was effectively received, accepted, and approved. 3xx (Redirection) - More action needed for the request to be completed in full. 4xx (Client Error): The request is not fulfillable or has incorrect syntax. 5xx (Server Error): The server did not fulfill a valid request.

Additional information can be shared between the client and the server with a HTTP request or response thanks to headers. Different kinds of headers exist: Additional details about the resource being retrieved or the client who submitted the request are provided in the request headers. Additional information about the server or the response is provided in the response headers. Although they are applicable to both requests and responses, general headers have no direct connection to the information in the body. Entity headers include details about the body of the resource, such as the MIME type and content length.

2.6.3 Hyper Text Transfer Protocol

HTTP is a protocol that does not store information about previous requests. Each request from the client to the server must contain all the necessary details to understand and complete the request. There is no open connection between successive requests carried out on the same connection. Although HTTP is stateless, web applications can maintain state using cookies, which are sent with requests and responses. Sessions are used to store information across multiple requests.

The secure variant of the standard HTTP protocol is called HTTPS. Using Transport Layer Security (TLS) or the more outdated Secure Sockets Layer (SSL), it encrypts the transmission. This additional security measure protects against man-in-the-middle and eavesdropping attacks. Over time, the HTTP protocol has experienced multiple enhancements. Starting with HTTP/1.0, 1.1, 2, and 3. These upgrades increase the speed and efficiency of the HTTP protocol.

To describe, it is a practice to code web services in using the Representational State Transfer (RESTful API) architecture due to its being simple, scalable and stateless. It makes use of standard HTTP methods and revolves around the idea of exposing network resources through simple and predictable Uniform Resource Locators (URLs). Furthermore, it has other principles and components apart from its being client-server architecture and http protocol.

Each API call runs in an isolated thread which contains all necessary information enhancing scalability and visibility. Responses are either cacheable or not ascertained. If response is replicable, then later client request might retrieve again the same response data with no any interaction with server. In this way, RESTful API uniformly interfaces simplify and decouple architectures allowing each component evolve on its own. These guiding constraints are:

Resource Identification in Requests, resources are identified in requests using Uniform Resource Identifiers (URIs). The resources themselves are conceptually separate from the representations that are returned to the client.

Resource Manipulation Through Representations, when a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

Self-descriptive Messages, each message includes enough information to describe how to process the message. Responses also explicitly indicate their cacheability.

Hypermedia as the Engine of Application State (HATEOAS), clients interact with a RESTful API service entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

The client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. Intermediary servers can improve system scalability by

enabling load balancing and providing shared caches. Moreover, resource naming best practices are usually used within RESTful API. Use nouns to represent resources (e.g., `/users`, `/books`). Use HTTP methods for operations on resources. Use plural forms to denote collections. Keep URLs as simple and intuitive as possible. Adherence to these principles allows for the development of flexible, scalable, and easy-to-use APIs that can evolve over time without breaking clients' implementations.

2.6.4 Backend

The Java programming language has a rich feature set that makes it an attractive option for backend development, especially when paired with the Spring Boot framework and Maven for library management.

There are several reasons why Java was selected as the backend programming language. It is especially helpful for systems deployed over multiple hardware because it can function on different computers without requiring modifications. Furthermore, Java boasts a well-developed ecosystem with a large set of useful libraries, frameworks, and tools. For many backend applications, Java performs almost as well as lower-level languages like C thanks to its compiler and JVM optimizations. Java's ability to manage scaling workloads is essential for backend systems. Java type safety reduces runtime problems and enhances code quality by assuring the early detection of errors.

A strong technology that makes Java more capable and speed up the development process is Spring Boot. Developers do not have to create as much complex code because it takes a "easy over complex" approach. Microservices are supported by Spring Boot, enabling programmers to create scalable and independent services. It is adaptable and appropriate for a range of backend applications, including complex transactional systems and basic web apps. Its integrated server, which facilitates deployment and minimizes the need for external setups. Furthermore, Spring Boot easily integrates with other Spring projects as well as external libraries, offering pre-made solution for a variety of backend requirements.

Maven is a useful project management tool. By offering a common repository that effectively manages project dependencies, compilation, packaging, and deployment tasks. Library dependencies are handled automatically by Maven's dependency management system, which guarantees that projects have all the libraries they need without any conflicts or missing parts. It streamlines and speed up library management by storing and retrieving Java libraries and third-party dependencies from a central repository. Maven additionally minimizes the "it works on my machine" issue by ensuring consistent builds across various development environments. Because of its plugin-based architecture.

There are numerous advantages to using Java, Spring Boot, and Maven in backend development. This technological stack creates a basis for the application that is stable, scalable, and sustainable while also speeding up the development process. Many developers prefer these technologies over others because they provide effective library management, robust performance, and easy development. This combination also produces reliable and scalable applications while speeding up the development process.

An industry standard for backend software development is layered architecture, particularly in systems that use Java. Software is arranged into layers, each with a specific function. The Presentation Layer, sometimes referred to as the controller, manages the user interface and the way the program communicates with the browser. The rules and business logic are handled by the business layer. Then there is the Persistence Layer, which is frequently mentioned when talking about CRUD processes and is in charge of finding and storing data. Sometimes the business and persistence functions are taken care of by the same layer in simpler applications. Even more layers may be included in more complex systems.

Every layer is focusing on a specific set of tasks. For example, the Presentation layer covers providing the user with information, while the Business layer covers data processing. Building and maintaining the program is made easier by this organizational structure, as each layer only needs to communicate with the layers directly above or below it.

Layers are supposed to be "closed," which implies that a user's request proceeds through each layer in order. Because it keeps changes in one layer from interfering with those in another, this helps ensure the stability of the software. However, in many cases, the layers are "open," allowing requests to bypass certain layers. This might make the process faster, but it also makes it more difficult to handle changes. Let us say a user requests access to information about consumers. To obtain the data, their request starts at the Presentation Layer, proceeds through the Business Layer, Persistence Layer, and Database.

Depending on what software is required, different techniques and patterns are used to bring these layers together. While very reliable, this tiered method is not flawless. It may result in instances where the software just rearranges requests without providing any additional value, which is inefficient. It may also lead to large, expensive software that is challenging to evolve and scale.

In general, layered architecture is a good option for beginning tasks, such as creating prototypes. It must be used carefully to prevent becoming too big or limiting. When used correctly, it offers a transparent framework to manage software updates over time.

2.6.5 Frontend

Due to its rich feature set and compatibility with modern design patterns, Angular is a great frontend development framework to build web applications, especially prototypes. To address the dynamic nature of web development, Google has developed Angular as a complete frontend solution that handles user experience related issues. For instance, it has an architecture that is predominantly component-based. This means that every element of a program including its logic and UI templates is built as an independent unit. These are the key components of any application that facilitate a clean, testable, and maintainable codebase in software development. The concept behind this is that angular is modules-based, where each module serves different purposes, so developers make scalable and organized code structures by using them correctly according to their needs.

To use Angular for development, TypeScript must be employed. It is an improved version of JavaScript that includes static typing. Static typing allows for early error discovery and fixing during development hence making the code more manageable and reliable. It is a superset of JavaScript with features such as generics, interfaces, and others present in statically typed languages. This addition makes it easy to apply cross-cutting development techniques and follow object-oriented programming principles.

It is a well-known fact that Angular has ‘two-way’ data binding. This enables the model and the view to be in sync automatically. This can enable developers to have more interactive UIs simply by writing less JavaScript code as they will not need a hand-written boilerplate code which aligns the model with the view on their own. Additionally, Angular supports dependency injection (DI) framework among other things which helps create decoupled components by injecting only needed components when their need arises thereby resulting in a more flexible, reusable and testable codebase.

Angular comes with a very powerful command-line interface (CLI) that enhances its basic architecture. Angular CLI takes care of regular programming activities such as creating new components and streamlining the final build. Besides speeding up the entire development process, this ensures consistency and adherence to industry best practices throughout all stages of project development. In order to provide large-scale network effect for sharing common knowledge, debugging assistance with huge amount of reusable codes required especially during inception stage where rapidity is crucial there are a lot of other tools and libraries behind angular’s ecosystem supported by vast community plus third parties ones.

The incredible innovation made by Angular in the MVC pattern can be seen through its application architecture. The three components that make up an application structure according to MVC are controller, model and view. Angular has improved this even further using its services design and two-way data binding feature. The Angular model encompasses services and classes for defining application data structures as well as business logic. These are important since they reflect the state of the program at any given time and dictate how it should handle data. In Angular, views are built using templates which are made up of HTML combined with Angular’s template syntax. An Angular component system replaces the conventional controller system altogether. In between are some subcategories of views controlling the flow from a higher level view down towards lower levels where information is rendered or user actions controlled; these are angular components. For instance, event handlers for UI controls such as buttons, drop-downs etc., rendering fetched data through angular interpolation directive or having input forms which users can interact with it.

Angular adds services that contain shared business logic, data services, and other code not directly related to the view are added by Angular to the standard MVC architecture. Given the complicated workflows found in modern web applications, this advanced architecture makes a codebase easier to maintain and organize. Prototyping can benefit from the use of Angular and the MVC pattern. The foundation for future scalability and rapid development is established by the framework’s built-in structure along with the

MVC architecture. Prototype complexity can increase while keeping an organized and flexible code structure thanks to Angular design principles. The quality of the code is improved and errors are avoided by using TypeScript, which strengthens the development process.

For frontend developers, Angular is a compelling option, especially for prototypes that need to be developed and demonstrated quickly. Because of its conformity with the MVC pattern, it offers a strong foundation for developing applications that can grow from a conceptual prototype to a finished, production-ready product.

2.7 Software Process Model

This section describes how the agile software development process is integrated with the Scrum framework to develop the prototype. Scrum is well known for emphasizing incremental progress, flexibility, and teamwork within the framework of well-defined goals [HH18].

The Scrum process was applied methodically through the creation of the prototype. Tasks were divided into time-boxed sprints that required completion of tasks and preparation for review. Every sprint started with a planning meeting with the master thesis supervisors to establish the objectives, and ended with a review to evaluate the results of these objectives.

Scrum is generally a collaborative framework, yet it was designed to operate well for a one-man activity. By self-assigning duties that are normally divided among a team, this novel adaption ensured a disciplined approach to the project management parts of development.

The sprint and product backlogs, among other Scrum artifacts, were maintained by a single person. The effective implementation of the traditional Scrum ceremonies like daily stand-ups, sprint planning, reviews, and retrospectives—made self-evaluation and agile decision-making possible.

GitHub, through its issues feature, provides a way to manage the Scrum process. Issues can be turned into sprint backlog items on the platform, functioning as a virtual Scrum board. Labels help define ticket classification and prioritization, ensuring that the status of the project is always on track. GitHub's task management features together with agile Scrum principles help keeping the development process focused, organized, and flexible to the changing requirements of a project. This autonomous yet systematic approach highlights the flexibility of Scrum and allows developers to use agile methodologies to achieve excellent results.

Related Work

A sophisticated concept that spreads across various software engineering disciplines, including collaborative software development practices, version control, and model-driven development is the management of heterogeneity in conceptual models and their historization within VCs. Although this may be limited due to the specificity and depth of the topic, some basic ideas as well as other relevant research works provide useful insights and frameworks for addressing these issues.

3.1 A General Theory for Evolving Application Models

The design theory for object model evolutions, as researched by Proper et al. and van der Weide et al. [PvdW95], provides the necessary theoretical foundation for this study.

According to [PvdW95], a snapshot is a state or representation of the information structure at a particular point in time. A snapshot in evolving information systems captures the underlying conceptual level of the information structure like its present configuration and data population. This distinction, fundamental to theory, separates the unchanging character of the data from its evolutionary flow through time. By examining snapshots, one can study and follow changes made so that the system can evolve according to well-formedness axioms and other principles, such as those supported by object-oriented methods and object-role modeling.

The paper [PvdW95] have proposed modeling approaches of evolution that have several uses in the comprehension and handling of evolutions in application models. Each approach has its distinct advantages that contribute to a better understanding of model evolution.

The first approach is based on snapshots see Figure 3.1a. Evolution modeled by snapshots entails keeping full copies of the evolving design at different time points. This method ensures that each state of the system is fully documented, providing a clear and

3. RELATED WORK

comprehensive view at any given point. The main advantage of this approach is historical traceability, as one can follow up the model's history from early stages to understand why changes were made and what they entailed. Furthermore, having complete snapshots helps in maintaining data integrity since every snapshot represents consistent state of the system. This method also allows for easy rollback processes which enables recovery to previous states when errors or mistakes occur during implementation.

In the second approach, evolution modeled over time by functions (see Figure 3.1b), the emphasis is placed on individual element changes that occur in a given moment. In this way, a more elaborate picture of how each part of the model grows can be outlined. One advantage of this approach is that it makes querying easier. That would better facilitate the search for any change, which can also be made with delta expression having been assigned to all states following one another so as to find out whether this or that piece was included in some state or not. Such techniques enhance identification and understanding of changes. This also eliminates redundancy as well as saves space in storage since we only track changes without duplicating unaltered parts of the system.

In the third approach called derive snapshots from element evolutions (as shown in Figure 3.1c) that combines the advantages of both snapshots and functions approaches. This gives a complete position of each state as seen in snapshots though one is allowed to track changes on a minute level; like evolution modeled over time by functions approach, the connection between full snapshots and the previous ones guarantees uniformity and logical continuity in every situation. This method maintains a comprehensive state representations even with some redundancy caused by copying unchanged models while retaining detailed change tracking.

All these modeling approaches make it possible to effectively manage, understand and use evolving application models. They provide a combination of historical responsibility, data consistency, effective querying and balanced representation that is useful in tracking all changes within evolving conceptual models.

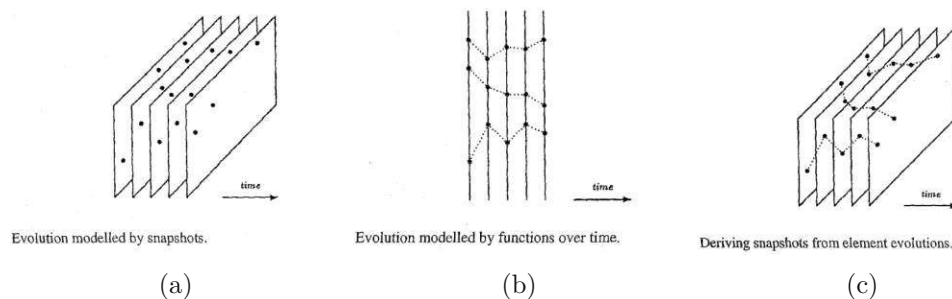


Figure 3.1: Theoretical design approaches for modeling evolution [PvdW95].

The research commences by noting that there is an increasing need for information systems that can be changeable with no expensive data conversions and reprogramming regarding their information base (conceptual schema) and structural nature (dynamic

characteristics). According to the theory suggested by the authors, it is possible to update all aspects of the application model: constraints, population, operations on it,tc. This theory aims at being independent from any specific modeling approach used. It builds upon a weak assumption about how models are constructed beneath it, thereby making it usable within a wide range of data modeling techniques including generalized object role data modeling, extended Entity Relationship (ER) or ER data modeling.

The authors define the domain for application model histories and include time as one of the essentials for evolution. They propose element evolutions, which explain the changes that have occurred in certain elements of application models over time. The study highlights how important it is to track application model versions to effectively manage their evolution. As an example, this change illustrates how the information structure changes, as illustrated by the rental store moving from a state of LPs to that of CDs. This example clarifies the practical implications of changing information systems.

The paper [PvdW95] provides a theoretical framework that underpins evolving information systems and addresses issues related to adaptation to changes in the basis and structure of information. The authors' theory provides a strong foundation for comprehending and managing the evolution of application models, thus making an important contribution towards data engineering and development of information systems. In the future, the paper's work proposes that research trends will include implementing changing information systems, developing modeling techniques that accurately represent changes, and studying how evolving internal representations of information structures impact upon them.

In conclusion, this work provides a foundational theory for developing a prototype that derives snapshots from the evolution of elements.

3.2 Bridging the Gap between OpenMBEE and Git

The Siegl et. al article [Sie21] argues for more advanced configuration management in Model-Based Systems Engineering (MBSE) with agile methodologies. The paper brings in Lemon Tree, a 3-Way Diff and Merge Tool for system models to address the lack of Git support in any commercial modeling solutions. Lemon Tree optimistically versions models, making changes clear visually. Moreover, this article investigates integration of Lemon Tree with Model Management System (MMS) web services as an alternative to ad hoc integrations. Also, the authors have exemplified how they can use Model Development Kit (MDK) and MMS together in the Open Model-Based Systems Engineering Environment (OpenMBEE) area by integrating Cameo, MDK and Lemon Tree as a diffing and merging tool for Cameo models. The technique is intended to provide graphical support for model merging while maintaining synchronization.

The aim of [Sie21] was to demonstrate that it is feasible and advantageous to combine Git; a popular version control system; with an open MBSE environment called OpenMBEE. In MBSE it is expected that Git's powerful version control features including branches and merges will be used for dealing with complexity and versioning of system models.

The steps involved to accelerate the integration process included: Preparation of the model: Development of system models was undertaken using OpenMBEE-compliant System Modeling Language Models (SysML) tools. These models formed the basis for subsequent version control processes. Incorporation of Git: Upon merging the models, activities such as push, pull and commit could be run on the repository. For this purpose, it was mandatory to establish a Git environment that effectively managed model files. LemonTree Merging: To show how updates to the model are diffed and merged, Lemontree was employed. The study illustrated how alterations on a model can come from different branches exhibiting how intricate models can be compared using this tool. OpenMBEE Synchronization: Finally, the study demonstrated that changes made through git may be synchronized with OpenMBEE so as to have the most up-to-date versions of their systems' designs in both worlds.

Git was demonstrated to be an effective tool for documenting changes, branching and merging that is used in managing versions of system models. LemonTree and Git can be combined together so as to accommodate complicated model merging thus making it easy for several engineers to work together. Any System Model Changes done on LemonTree, which are then merged with the help of Git are synchronized with OpenMBEE Platform for consistency in the entire MBSE ecosystem.

The paper [Sie21] demonstrates how combining git version control with OpenMBEE's MBSE capabilities enhances systems engineering processes. It solves problems associated with model versioning and collaboration, thereby enabling more flexible, efficient and scalable workflows for MBSE.

3.3 Reflecting on the past and the present with temporal graph-based models.

Using temporal graph databases, the paper [GDBG18] proposes a unique method for improving the self-explanation abilities of Self Adaptive Systems (SelfAS). This technique addresses the necessity for these systems to comprehend and rationalize their past and present behavior in order to enhance decision-making. The goal of the proposed framework is to give the system a basis for storing execution traces in a way that makes it easier to analyze them both forward and backward, allowing the system to respond to inquiries about the rationale behind particular choices and observable behaviors.

The foundation of the suggested methodology is the development of a generic meta-model intended to organize SelfAS execution traces. This meta-model ensures that significant information is recorded methodically by providing a framework for arranging data about the system's observations, choices, and actions throughout time. Concepts like metrics, decisions, actions, and Non-Functional Requirement (NFR)s are all included in the meta-model. Because of its extendable architecture, it may be tailored to different SelfAS types and the data they must track.

The study recommends temporal graph databases as a useful tool for storing and expressing the execution traces that the generic meta-model defines. With the use of temporal graphs, one may simulate how a system's state changes over time by identifying the entities (decisions, actions, etc.) and the temporal links that connect them. This form is especially useful for examining how the behavior of the system varies over time and in response to various circumstances. The authors demonstrate how to create a temporal graph model from a series of traces and show how this type of graph can handle intricate inquiries about the system's historical behavior.

The article presents a query language designed specifically for this purpose, which should make interacting with the temporal graph models easier. By adding features for navigating across time, such as retrieving all iterations of a model element, querying states within a certain time period, and accessing the historical sequence of changes, this language expands on previous model querying languages. The authors show how the query language can be used for interactive diagnosis at runtime, forensic analysis, and self-explanation through a case study.

The paper demonstrates the usefulness of the suggested framework with a case study of a self-adaptive system using Remote Data Mirroring techniques. The case study demonstrates how queries may be created to look at the system's decision-making process and how the system creates and maintains execution traces in a temporal graph. The authors demonstrate how the framework can be used to address inquiries concerning the system's adaptation activities, the justification for certain choices, and the effects of those choices on the goals of the system.

Finally, the study provides a thorough methodology for using temporal graph databases and a customized query language to improve SelfAS's self-explanatory capabilities. To facilitate better decision making and increase transparency for users and operators, the framework addresses the crucial requirement for SelfAS to reflect on their behavior over time.

3.4 Calculation and Propagation of Model Changes Based on User-level Edit Operations

The aim of the paper [Keh15] is to give an overview of models with respect to the management of versions, evolution, and variability in software development. Specifically, it focuses on the domain languages that are used in the building of complex systems using advanced software configuration management tools. Alongside pointing out how versioning technologies fall short, it provides some user-friendly alternatives. The topics covered within the document involve feature-oriented domain analysis, model difference calculation and visualization, and change propagation. It also stresses the importance of maintaining consistency and the difficulties encountered when addressing errors or conflicts during model modifications. Ultimately, the paper recommends tools and

techniques for working with models in a more natural manner that enables consistent collaborative development efforts.

The paper [Keh15] focuses on SiLift, a framework based on the widely used Eclipse modeling project, that enables user-specific model change mining and applying tools based on the concepts developed in this thesis. This framework is provided as a flexible solution for adaptability to different modeling languages and situations, with a major objective of reducing laboriousness usually involved in developing model version control tools. SiLift is designed for variability among modeling problems therefore enabling mapping of user visible features into distinct pieces of software for easy composition into separate tool configurations.

It also provides details about the core components and their configurations for differencing tools, thereby explaining how different parts such as a model editor, difference calculator, matcher, etc., could be combined to form the tool. Moreover, the framework seeks to exploit existing components from MDE tools so as to avoid redundancy and improve performance.

The systematic approach to configuring patching tools that apply patches interactively or in a batch mode is provided by the SiLift tool. This framework provides for different core components, such as validators, argument managers, and patch engines, which maintain the consistency and integrity of the model after applying patches on it. The paper [Keh15] also focuses on how SiLift can be adapted to various modeling languages and situations, demonstrating its adaptability and versatility. The scope ranges from comparing Ecore models' versions to managing variant models in factory automation systems and logging feature models changes. It illustrates how the framework handles complex versioning requirements that span across domains and involve multiple modeling languages.

To summarize, the author of this paper has introduced new methods and approaches into structural model versioning that utilize some level of abstraction to turn these changes into user actions. More so, the article also suggests how consistent patching of models can be achieved and models in workspace can be updated in a different way. It outlines a general configuration tool component framework that makes use of reusable components for creating tools with ease. They are evaluated through application on several projects in order to understand their functioning well. This research highlights the significance of raising differences between models to semantic levels above. In essence, this finding is quite significant for areas related to model change management given its utilization of edit operations which are rich in semantics.

3.5 A new versioning approach for collaboration in blended modeling

The study [EPR⁺23] makes it clear that in the development of modern software-intensive systems, they have become more complex, thus requiring developers to be flexible

with their choice of languages and toolkits. Consequently, the authors suggest blended modeling as a possible avenue for addressing these problems where an individual may edit one model through several computer programs. For instance, the authors found that some of these new models still lack certain functionalities such as bidirectional change propagation between Concrete Syntaxes (CS) and Abstract Syntaxes (AS), layout continuity, and deferred conflict resolution, among others. This gap motivates the need for an alternative versioning approach.

The core of the paper introduces a novel operation-based versioning system that is dependency-aware. It was made with complexity blending in mind by reducing model operations to graph operations. In order to facilitate merging and branching of model versions, conflicts and dependencies between operations are dealt with efficiently. Structured versioning in collaborative environments can be achieved by representing models using graphs that show changes over time. Details about the implementation of this versioning system are provided including how primitive and composite deltas (changes) were handled, constructing a history graph for edit operations tracking and mechanisms for branching and merging changes.

The paper dedicates a large part of itself to an example that shows how the proposed versioning can be applied. An example demonstrates minimal state graphs which act as a proof of concept to handle visual CS and textual changes in synchronization with the underlying AS. The authors employ this example in their work to demonstrate how practical and adaptable their approach is when used in practice. In their implementation of the versioning system, the authors also provide further details about setting up web-based demonstrator. This section presents the system's technical architecture, utilization of TypeScript library to handle deltas, versions and graph states, return users back into these states; as well as creation of user interface that allows users to use it and try the versioning system out within a hybrid modeling environment.

Decentralized conflict resolution is the topic of this paper, which provides users with alternatives in merging so that they can determine how conflicts should be addressed. With this method, users have the ability to postpone solving conflicts, and they are able to take part in collaborative negotiations about how to merge changes from different versions to make informed decisions. This article presents a comprehensive and innovative approach to versioning in blended modeling environments that addresses the challenges of collaboration, change propagation, and conflict resolution. By illustrating a detailed methodology, a practical running example, and a demonstrator implementation, the authors convincingly argue for the benefits of their dependency-aware operation-based versioning system, which sets a foundation for future advancements within the field.

3.6 Historization of Enterprise Architecture Models Via Enterprise Architecture Knowledge Graphs

The paper [BHB22] examines transforming enterprise architecture (EA) models into EA Knowledge Graphs for managing their complexity and supporting automated visual analysis. A major weakness of the existing methods is that they treat EA as a static representation, which means it is represented at one single point in time only. In order to be able to trace its history and process EAs, the authors introduce a historization procedure which allows analyzing how EAs have changed throughout time. Therefore, it entails making a historical record of EA models which makes it possible to investigate the changes and the overall evolution of EA itself.

This method supports graph-based analysis by converting nodes and edges into graphs from EA components and relationships for different types of analysis, including graph-based algorithms for detecting flaws or identifying improvement opportunities. Prior research [SB21a] has shown how EA models can be transformed into graphs to allow for analysis, but none has been capable of tracking their development over time. The CM2KG platform [SB21a] is indicated as being capable of turning conceptual models into Knowledge Graphs, thereby facilitating the measurement of quality attributes such as Enterprise Architecture Smells (EA Smells) [SHB21]. Nevertheless, it does not cover all phases in the evolution lifecycle of the enterprise architecture, thus revealing only one state of it. Consequently, this article suggests expanding this platform by incorporating changes and evolutions into the graph structure so that historical analysis becomes part of this system.

Historization consists of adding properties to the elements of a graph so that it is possible to detect the smells and insights from EA evolution based on history. Also, the GraphML format has been used in this implementation to integrate with the CM2KG platform and store historical data in an efficient way. In order for it to be practically usable, the paper assesses the performance of the platform in terms of transformation times, fetching times, and querying times.

However, there are some gaps for exploration in a thesis about historizing EA models utilizing knowledge graphs. Creating more enhanced query facilities will facilitate complex historical analysis, while exploring their integration with other EA modeling tools will promote interoperability as well as adoption. Techniques for scalability and optimization can be researched to better manage larger models and more frequent updates. Better visualization and interaction with historical data through improving the UI could make it easier for users to navigate and analyze EA evolution intuitively. Those areas would help significantly develop knowledge graph-based historization of conceptual models, which are practical since they can contribute significantly to research literature in this area.

3.7 From Conceptual Models to Knowledge Graphs: A Generic Model Transformation Platform

The authors Smajevic and Bork discuss in [SB21a, SAB24] a generic and extensible platform for converting conceptual models into structured knowledge graphs. This research is aimed at automating the analysis of such models, which is particularly important for medium to large-scale models. CM2KG extends prior work on graph-based analysis [SB21b] that is able to support multiple modeling languages and tools, thereby ensuring scalability and flexibility.

Examples of such platforms include Eclipse Modeling Framework (EMF), ADOxx, or Papyrus, while formats like Graph Markup Language (GraphML), Resource Description Framework (RDF), or Web Ontology Language (OWL) serve as standard knowledge graphs. In this regard, the platform can be viewed as language-independent; specializing in transforming various kinds of object-oriented models developed using Unified Modeling Language (UML) or Systems Modeling Language (SysML) notation as well as enterprise architecture frameworks like ArchiMate among others. The three main components are 'Model Import', 'CM2KG Cloud', and 'Third Party Tools'. The latter includes the Neo4j database where transformed knowledge graphs can be stored and analyzed.

The central functionalities of the CM2KG platform are model transformation, knowledge base initialization, visualization and analysis processes. In this regard, the platform enables users to upload conceptual models in XML format for transformation into KGs. The transformed graph is used in the initialization of a knowledge base in Neo4j that can be visually inspected and analyzed further. Additionally, the platform provides a rich set of graph visualizations and analysis features including predefined queries for analysis, code smell analysis, and user-defined queries.

This is demonstrated through a case study in which 5000 UML diagrams were transformed into KGs. From there, the authors detected Smells automatically within these UML diagrams. The study was done to show that this platform can handle big data sets and perform complex analyses at an acceptable level of efficiency within reasonable time limits.

The evaluation revealed that a large part of UML models could be transformed and analyzed through the CM2KG platform with the detection of various code smells like Message Chain, Deep Hierarchy, Cyclic Dependency. There was efficiency in the performance of transformation and analysis processes, as evidenced by the average transformation time of 16 ms and query execution time ranging from 0.007 to 0.43 s.

The CM2KG platform stands out compared to other tools because it is language neutral, easily adaptable to new languages, and scalable to big models for analysis purposes. Although some tools provide code refactoring facilities, CM2KG focuses on providing comprehensive web-based functionality that mitigates installation and compatibility problems.

3. RELATED WORK

In summary, the CM2KG Cloud platform enables conversion of conceptual models into knowledge graphs that facilitate advanced analysis and reasoning. By doing so, this platform becomes an important addition to the model-driven software engineering community, calling for more research in knowledge graph-based analysis development too.

Historizing Generic Models

Our approach begins with the requirements engineering phases, setting the stage for this chapter. We begin with identifying and analyzing necessary requirements which formulate a solution design. On that basis, we move on to designing a robust solution that implements a prototype of historizing knowledge graphs.

4.1 Requirements Engineering

A systematic requirement engineering process was used to develop the prototype for historizing CM with KG, ensuring a consistent and structured way to manage the requirements. This approach effectively addressed all needs while minimizing the risk of errors. The process started with defining large goals known as epics, which were then broken down into smaller and manageable tasks that together make up the epic. To handle these needs as issues, GitHub was implemented at this phase starting with definition of large, overarching objectives referred to as 'epic'.

Several tasks were involved in each epic aimed at addressing various aspects of project scope. This method became possible through GitHub's labels feature that enhanced job prioritization and classification. The following labels were used:

- Bug: These tickets required bug fixes.
- Epic: This represented a complete story that fulfilled a high-level goal.
- Documentation: Tasks such as outlining the high-level architecture, detailing the design solution or documenting the solution within this master thesis had to be extensively documented.
- NiceToHave: These tickets were not necessarily critical to the thesis but included improvements or additional features that would enhance the end product.

- Spike: These tickets involved unexpected research and exploration arising during execution of other tickets.
- Task: Completion criteria for newly introduced features were defined using this label.

Every epic has several tasks in it. Once all the linked tasks with an epic were closed, only then could it be considered complete.

The phase of requirements engineering was very important in determining course of the project as well as ensuring all goals and functions are satisfied. Each epic became a focus point that accelerated development by employing GitHub features. In this system, each documentation was a full enumeration including high-level architecture and design solutions.

The characteristics of 'nice to have' were considered, but were not given preference for the sake of not undermining main points of the thesis. Also, "Spike" tickets helped in investigating unknown issues and introducing innovative ideas that met the need for agility during development process. As every issue stated what exactly should come out to achieve more general goals outlined by epics.

Figure 4.1 shows the distribution for different types of labels used throughout prototype development. All created and closed issues can be found at GitHub, please note that currently the repository is private and a read authority is required to be able to access it.

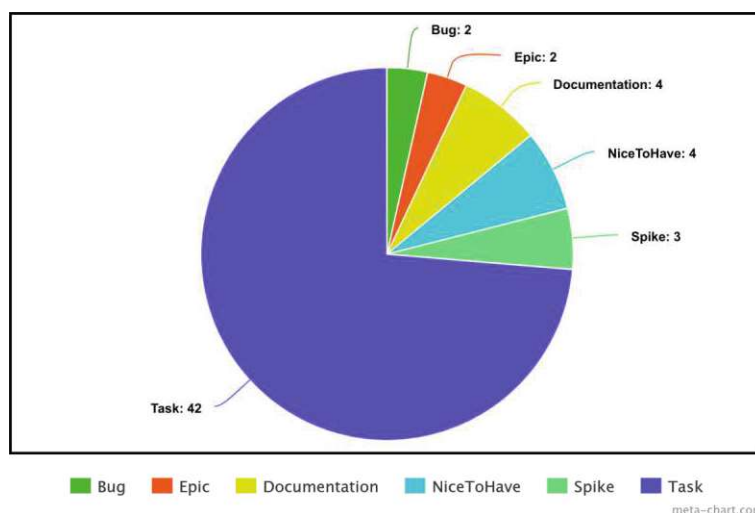


Figure 4.1: Github labels types statistics

The project was steered from start to finish by the requirements engineering phase. The development process was facilitated through GitHub issues that served as a platform for requirements management.

The requirements of historizing knowledge graphs arose from researching the need to build a version control system for conceptual models. These requirements were derived from stakeholders, including myself (the developer) and supervisors (the users). Below is an outline of the requirements and tasks that have been identified through this process according to best practices in requirements engineering [LK22].

Requirements engineering for this project starts by defining the basic needs which are required for developing a strong version control system specific to conceptual models. This involves capturing requirements that enable one to compare, traverse, and track changes in knowledge graphs overtime.

Stakeholder Requirements

As a developer, I want to historize conceptual models using knowledge graphs in a manner that allows for storing snapshots and versions of their graphml files.

As a developer, I want to historize conceptual models using knowledge graphs in a manner that allows for comparison and traversal of their history.

As a developer, I want to historize conceptual models using knowledge graphs in a manner that allows for filtering based on metadata attributes.

As a user, I want to historize conceptual models using knowledge graphs using a web application to be able to upload new versions of graphml files.

As a user, I want to historize conceptual models using knowledge graphs using a web application so I can track their changes.

As a user, I want to historize conceptual models using knowledge graphs using a web application so I can compare their versions.

As a user, I want to historize conceptual models using knowledge graphs using a web application so I can filter their versions based on metadata such as, authors, type of change, date and etc.

As a user, I want to make sure that the functionality of historizing conceptual models using knowledge graphs in the web application is feasible.

The following functional and non functional requirements are derived from stakeholder requirements and added as Issues in GitHub:

Functional Requirements

- Implement auto-complete functionality for nodes and set limits for queries to enhance usability and performance.
- Integrate different libraries to support the comparison and differentiation of graph versions.
- Improve the user interface to provide a more intuitive and efficient experience for users interacting with conceptual models.

- Develop comprehensive unit tests to ensure code quality and functionality across the system.
- Integrate the Conceptual Model to Knowledge Graph (CM2KG) framework to enhance system capabilities.
- Add filtering features to allow users to sort and view data based on specific criteria.
- Implement a snapshot approach to track the state of conceptual models at different points in time, ensuring accurate historical records.
- Ensure that updates propagate throughout the entire model to maintain consistency.
- Develop mechanisms to handle API exceptions gracefully, ensuring system stability.
- Ensure that only one instance of the XML file is saved in the database per commit request to avoid duplication.
- Implement functionality to save the graph ID for tracking and referencing purposes.
- Enhance the frontend by adding more attributes per model to provide detailed information.
- Improve the commits user interface by adding more features for better usability.
- Add the ability to filter commits based on various parameters for better management of changes.
- Replace the Neovis library with Vis Network for better visualization performance.
- Design the system so that each node represents a version of the conceptual model.
- Support the creation of types in the commit relation for all approaches to maintain consistency.
- Enhance the representation of nodes and edges for clarity and usability.
- Develop a delete solution applicable to all approaches to manage deletions effectively.
- Implement a timeline view to visualize the history of conceptual models' modification, displaying changes chronologically.
- Implement functionality to handle the deletion of nodes or relations.
- Create an endpoint to construct the GraphML file from a specific version for accurate version tracking.
- Implement the ability to export and import the history of a conceptual model to/from a file for backup and migration purposes.

- Enhance the user interface for viewing the history of conceptual models, making it more user-friendly.
- Implement the ability to restore a conceptual model to a previous version, ensuring data integrity.
- Add the ability to filter the history of a conceptual model by date range, user, or type of change for detailed analysis.
- Implement the ability to view the history of the conceptual model, providing a comprehensive overview of changes.
- Add support for reading XML files in the post endpoint to enhance data input capabilities.
- Implement features to handle conflicting changes to maintain data integrity.
- Add hover titles on edges instead of nodes for better user experience.
- Implement a timeline view to visualize changes over time, aiding in historical analysis.

Non Functional Requirements

- Make sure that the functionality of historizing conceptual models using a web application is feasible.
- Optimize storage by moving redundant metadata to the location where the GraphML file is saved.
- Configure Docker for the application to ensure consistent deployment environments.
- Implement the new design of the application to improve functionality and user experience.
- Integrate all created components to ensure seamless functionality and interoperability.
- Develop orchestration for different components to manage their interactions effectively.
- Dockerize all components for easy deployment and management.
- Document the snapshot approach and its configuration to provide clear guidance for future development.
- Implement and configure a hybrid (mix of Snapshot and Delta) approach for conceptual models to enhance flexibility.

- Set up the application source code for development and collaboration.
- Document the application architecture to provide a clear overview of the system design.

The project is designed to build a strong base for historizing conceptual models by concentrating on former requirements. In which, user interface enhancements and library integration, the introduction of new features and settings can all be implemented. As such, this broad approach guarantees that stakeholders are able to contrast, browse compare, traverse, track, follow and update the history of conceptual models effectively.

4.2 Solution Design

CMs vary across different domains and even within a single domain. CMs types are Entity-Relationship (ER) models, Unified Modeling Language (UML) diagrams, Business Process Model and Notation (BPMN), Data Flow Diagrams (DFD), among others. Each of these models has different structures, semantics as well as syntaxes which makes it highly complex to integrate them into one representation or framework. For example ER model is used mainly in database design to describe the data structure in terms of entities and relationships [Che76]. On the other hand UML ensures the general-purpose modeling language used in software engineering to depict the architecture of a system [JB21]. BPMN represents business processes by means of graphical notation [DHK11]. DFD are typically illustrated by showing how data flows through a system, bringing out what each process takes in and gives out [DeM11]. State charts indicate the states that a system can obtain as well as how it undergoes changes from one state to another [Har87]. As such, these different models with their own various purposes and relevance for different facets of systems and processes designing add-up to complexity while integrating them into coherent scheme.

Since there are many CMs, integrating them into a single representation or framework is very difficult, especially when involving historized models. This problem is particularly pronounced when considering the generic historization of CMs because various CMs have distinct structures and semantics, requiring a flexible and reliable method to capture them effectively. However it can be simplified by focusing on the KGs rather than the CMs, since they offer flexibility and standardization in the representation of different data structures and relationships. KGs have a common framework that can accommodate dissimilar CM's structures, semantics, and syntaxes unlike the complicated operation of directly integrating several models like ER diagrams, UML diagrams, BPMN, DFDs and state charts which have unique formats as well as purposes.

KGs are easily adaptable to changes over time, making them very flexible in capturing the historical evolution of CMs. Furthermore, they integrate data from various sources and formats to enhance interoperability thus enabling more cohesive representation of information across multiple domains and applications. They are also designed to

effectively handle large complex datasets; a feature that is useful when monitoring multiple changes or versions over time in CMs. In addition, advanced graph-based analysis techniques can be supported by KGs which are crucial for finding patterns, relationships as well as changes within data. This ability is very important when it comes to analyzing historical data related to CM's because one can identify trends or anomalies [HBC⁺21].

To make it through these difficulties, a general graph translation methodology aims at systematic translations of different CMs into an identical KGs format. This transformation enables the semantic richness and connectedness of a knowledge graph to be used in order to represent the information encapsulated within CMs more dynamically. Consequently, this simplifies merging different models and facilitates better analysis, querying, understanding data underlying, and relationships involved.

Utilization of the build approach outlined by Smajevic and Bork et al. [SB21a], can enable the conversion of conceptual models into knowledge graphs. Adopting knowledge graphs, specifically in the form of the GraphML standard, offers a widely accepted approach that the prototype can leverage in a generic manner. This choice helps to address the challenges that arise from the heterogeneity of the model types; see Figure 4.2.

The achievement of historization in a KG involves tracking changes over time within the graph, allowing users to understand how the information has evolved. Historization is crucial for models where it is important to see how relationships, attributes, or entities have changed. Moreover, it is also important to track other metadata, such as the author, who made the changes, the GraphML file before and after the changes, the time of the changes and the reason for the change.

The approach we used for versioning the KG is inspired by the strategy mentioned in the paper [PvdW95], namely the "Deriving snapshots from element evolutions" technique. What this means is that each element from one version of a KG should be interconnected with its corresponding element within another one thus generating a comprehensive graph comprising all versions. In this section, we discuss reasons behind our design choices for such an approach. Our approach to versioning keeps track of every item (node or edge) in the knowledge graph across versions. We have ensured that individual elements are tracked through different versions by linking each element from a particular instance to another same element in subsequent ones. With this kind of fine-grained versioning, we can follow every bit of change taking place across any part of the graph.

As we analyze individual parts, snapshots are made in order to represent the state of the whole KG at specific times. These are produced from changes in elements that have been accumulated, giving an entire picture of KG at various stages. This combination is necessary for achieving both fine-grained and all-inclusive versioning. Maintaining links between entities across different versions maintains continuity and consistency in the KG. Tracking back linking elements enable us to follow the history and evolution of particular element, making sure that changes are logical to maintain them over time. It

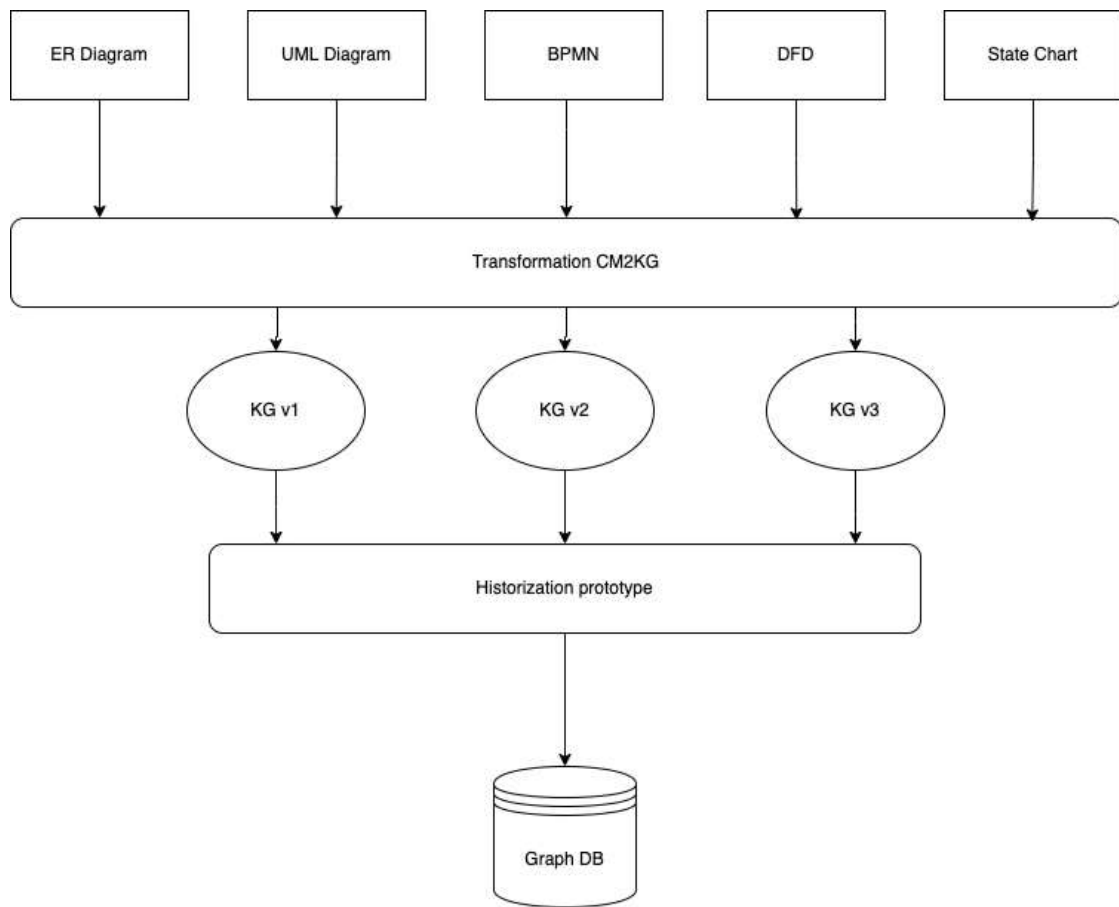


Figure 4.2: Conceptual idea of generic historization.

also allows the history chain to grow with scalability because it supports incremental updates whereby a new version does not require recreation of the whole KG. The design is efficient as it avoids redundancy with focusing on differences among versions only resulting in better performance when storing or querying historical data.

The approach is flexible and can accommodate different types of change including additions, deletions, and modifications of elements. By concentrating on the evolution of elements, we are able to adapt better to differing kinds of updates and maintain a robust versioning system that supports diverse use cases as well as data types. The design allows for sophisticated historical analysis and query capabilities. Users can make complex queries by keeping detailed records of how each element changes in order to track trends, patterns, or anomalies in the data over time. This is essential for applications that depend on temporal data analytics and insights. We integrate our approach with existing KG tools and frameworks KG such as Neo4j and Cypher, so that it could be practical and usable. This improves interoperability while enabling users to take advantage of advanced versioning features of familiar tools.

The implementation involves capturing the initial state of the KG as a baseline snapshot. For each update, changes are tracked at the element level, linking new versions of elements to their predecessors. Periodically, new snapshots are created to represent the current state of the KG, derived from the changes of the accumulated element. Querying capabilities are implemented to allow users to retrieve historical data and analyze the evolution of the KG. Our design decisions for versioning the KG, inspired by the "Deriving snapshots from element evolutions" method, focus on ensuring detailed, consistent, and scalable tracking of changes. By linking elements across versions and maintaining comprehensive snapshots, we provide a robust framework for managing the historical evolution of knowledge graphs, enabling advanced analysis and efficient data management. As an example see Figure 4.3, where node one from snapshot one is connected with node one from snapshot two. Their relationship in snapshot two are updated. Node two is deleted in snapshot two, and node three is created in snapshot two.

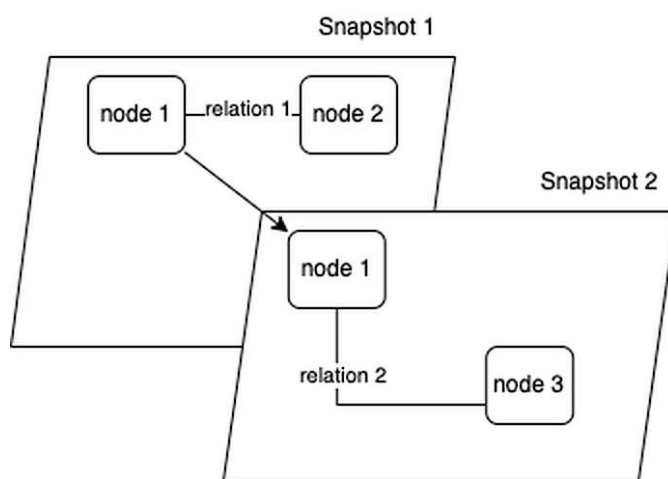


Figure 4.3: Conceptual idea of KG historization base on deriving snapshots from element evolutions.

We achieve historization by maintaining the following capabilities. Versioning: Implemented to track changes over time. In our case, implemented on the level of the relationship. Temporal Properties: Give relationships and nodes temporal features. This entails establishing the starting and ending dates for the validity of each node or relationship. Temporal properties enable querying the graph as it existed at any point in time. Change Logs: Maintain the change state of each node and relationship. Record the modification history including CREATE, UPDATE, DELETE, UNTOUCHED. They are implemented as a property of the Commit relationship. Querying: Support query capabilities of historical data, this includes queries that can navigate through the graph elements and snapshot or use temporal properties to extract historical data at any given time.

Selecting a graph database such as neo4j to operate as the model historization repository

involves considering several crucial aspects. The most noteworthy of them is the ability of graph databases to handle intricate relationship management. One of their greatest characteristics is that they naturally recognize how to manage and traverse the complex links between data sets, which is particularly helpful when it comes to historization. Because the relationships in the data are subject to change over time, graph databases are an advanced technique to display historical links. They provide sophisticated query capabilities that cover an entity's current state as well as its past interactions with other entities.

Furthermore, graph databases provide significant support for dynamic schemas. Their schema-agile design is critical to the model historization development. It allows for the addition of new entities and relationships without requiring substantial modifications to the database's present architecture. This specific feature of graph databases ensures a seamless and continuous integration of historical data tracking capabilities, which is particularly helpful in situations where the data model is dynamic.

Lastly, the need for graph databases' efficient historical data retrieval is unquestionable. Structured for optimal traversal of relationships, they are particularly potent for querying historical data. This is especially useful for historization, as it is typical to have to navigate the complex interconnected nets of previous relationships and entities. The faster extraction of past states and variations made possible by graph databases' superior design facilitates the timely and informed decision-making process [RN12, AG08, RWE15].

Figure 4.4 illustrates a historization solution utilizing GraphML within a graph database. Here, 'n1', 'n2', and 'n3' denote nodes labeled as Entity. The term Relation describes the model's relationships. Furthermore, nodes designated as Metadata are indicated by 'G'. The Commit relationship, alongside these, holds the temporal properties and logs changes for each model element. Moreover, observing the nodes, i.e., 'n1', we note that it is connected to other 'n1' nodes via the Commit relation, signifying the historical changes of the node 'n1'. The initial version has a Commit relation that points to itself; subsequent versions are navigated through a chain of Commit relations until the end version is identified, analog also for 'n2' and 'n3'. Each Commit relation holds the following properties: DateTime indicates the timestamp when this version was committed to the system, 'MetadataId' serves as the identifier linking to a Metadata entity, and this association between Metadata entities and Commit relations is characterized by a 1:N relationship, with N being on the side of the Commits. The property Type captures the operation represented in the change log, such as CREATE for creation, UPDATE for update, DELETE for deletion, or UNTOUCHED for unchanged. It is important to note that multiple commits within a single versioning event may exhibit different Type values, contingent upon their state and the nature of the incoming change. Moreover, with each new version or file upload related metadata including the commit message, author, modelId, and graphMlBase64 are stored in a single graph database entity labeled as Metadata. This method is chosen to ensure data consistency and to prevent duplication across all Commits relations.

Preserving the inherent properties of entities and relationships from the original KGs is

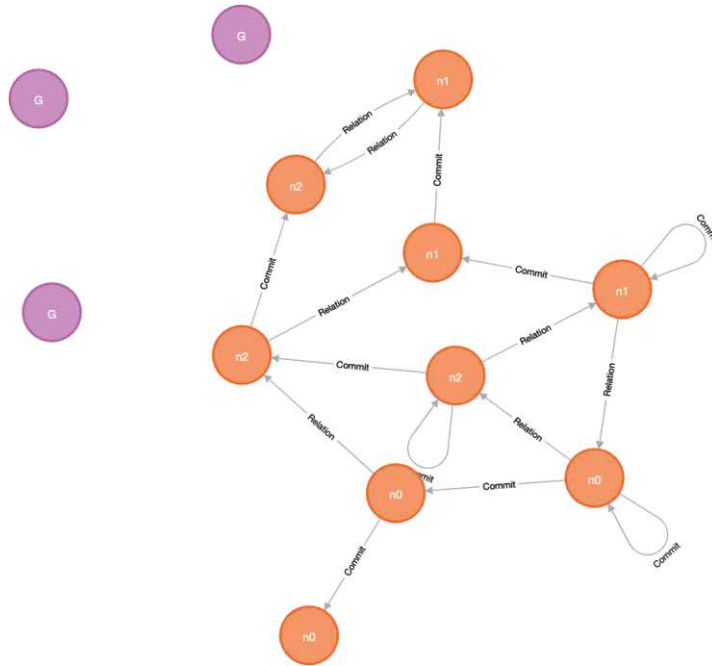


Figure 4.4: Neo4j Historization Example.

crucial. Such properties are persisted within a designated sub-map associated with each node and edge. For instance, an edge's properties denoted as 'attribute1, attribute2, ..., attributeN' are encapsulated within the graph database relationship `Relation`, specifically as 'properties.attribute1, properties.attribute2, ..., properties.attributeN'. An analogous approach is employed for nodes, where their original properties are retained within the graph database entity `Entity`. Additionally, each element in the KG should have a unique identifier (ID) in its original graph model. This is important to keep elements distinct and also plays a role in recording history. When the same element's ID goes through changes, we mark it as an `UPDATE`. If there is a new ID, we record it as a `CREATE`, and if an ID is no longer present in the latest version, we note it as a `DELETE`. Note that the decision is made based on KG's node changes, an edge creation or deletion is handled as an update to the bidirectional connected node. Otherwise, if nothing changes a node is marked as `UNTOUCHED`. When a node is reintroduced after being deleted, it is labeled as `CREATE`. This node won't have a commit relation linking to itself. Instead, it will connect from the previously deleted node, which is tagged as `DELETE` in the graph.

Additionally, if there are no changes in the entire file compared to its previous version, the uploaded file will be disregarded, and no snapshots will be created. A semantic XML comparison library will be employed to detect differences. Further details about this library will be provided in Chapter 5.

The former structure simplifies querying especially with the use of declarative graph

query language, and facilitates traversal through history records. It enables users to focus on what needs to be retrieved from the graph rather than the retrieval process itself, similar to SQL. Thus, graph query supports efficient and expressive queries that reveal previously hidden data connections and clusters, allowing users to maximize the potential of their property graph databases.

The creation of new entities and relations in the graph database is linked to how many nodes and edges the KG has. Because the strategy 'Deriving snapshots from element evolutions' requires adding new nodes and edges with each file upload, any new KG version will add a similar number of nodes and edges plus n Commits edges. This might seem like it could slow down performance. However, [Pok15] indicates that graph databases efficiently manage such setups. This efficiency is seen in many advanced applications that use even more connections. They can still query and navigate the graph quickly, especially when reading data. For instance, as shown in Figure 4.4, if there is an update with a node change, the graph will include three additional nodes linked to the existing ones. These new nodes will have the same kind of connections as before see the use case in Figure 4.5.

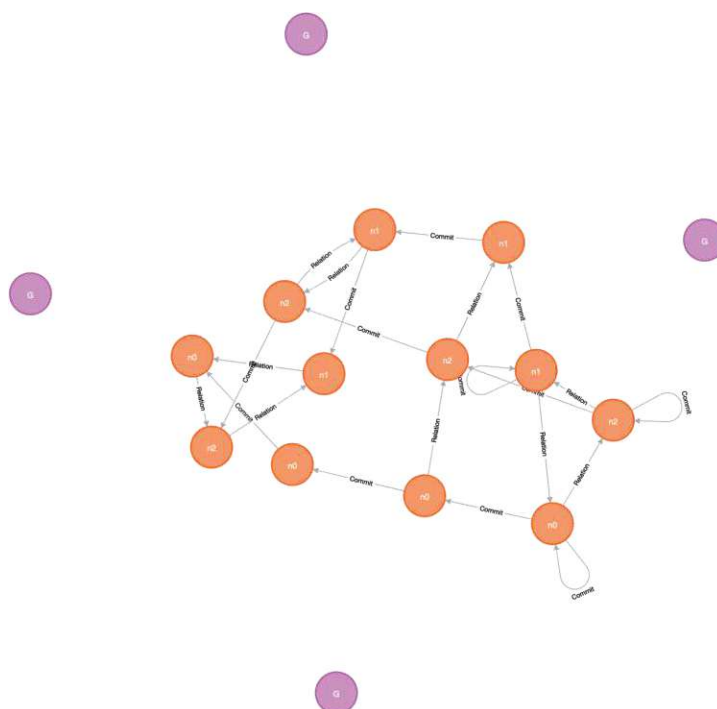


Figure 4.5: Neo4j Historization Example Post Update.

Prototype Implementation

When developing a prototype for historizing CMs, Neo4j is both practical and reliable enough to fit into the needs of its requirements. It does this by cutting down on complex joins and queries which would otherwise be required by other types of databases, thus providing less complicated way of working with data. This can be complemented by Cypher, as it efficiently queries and modifies graphs that evolve in their knowledge.

This is followed by Figure 5.1, which shows a visual presentation of the earlier design solution. It presents a clear picture of how the different components that made up this system were connected to each other. The aim of this image is to give more information about what really took place when the structure was built and its stages at that moment.

5.1 Software Development

In this section, we will explain how software engineering best practices can be used in developing a prototype for historizing CMs through KGs. This section outlines the various strategies of implementation, tool usage, and sticking to boundaries that are important when designing the prototype so that it functions well and meets stakeholder expectations.

Test-Driven Development (TDD) is an approach integrated into our development process. We use JUnit for writing tests ahead of service implementation to ensure that TDD pattern has been followed religiously. Every service starts with its test cases that direct the development and confirm that the code produced is within predetermined standards [Kos07].

Through Postman collections, we further commit ourselves to TDD, which complements such Junit tests by providing integrated testing. These collections are deployed before development for end-to-end testing to validate functionality within the application ecosystem and also serve as a proof of the methodology TDD.

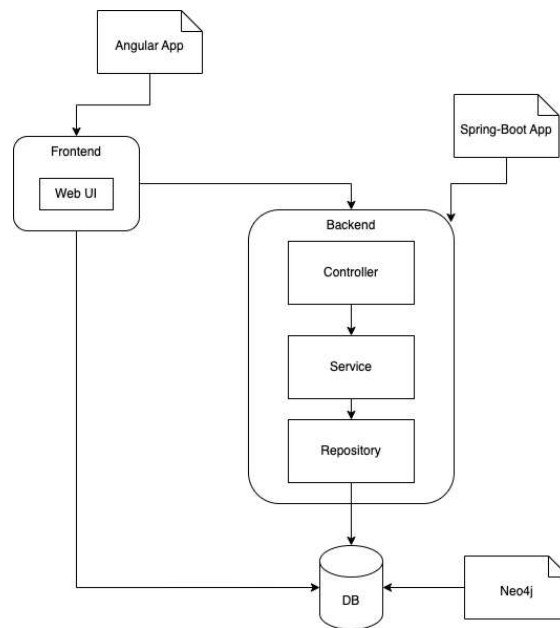


Figure 5.1: Full stack application architecture

We have chosen IntelliJ IDEA to be our development environment as Integrated Development Environment (IDE) with its strong features to smooth coding, debugging, and testing. In addition, the Neo4j Browser is an essential tool for checking the structures of the data model so that it can represent the intricate web of relationships inherent in the knowledge graph.

The backend of the prototype uses a range of Spring Boot dependencies as a suite. The suite consists of:

- ‘spring-boot-starter-actuator’ for monitoring and management purposes.
- ‘spring-boot-starter-cache’ which provides better performance through caching.
- ‘spring-boot-starter-data-neo4j’ needed for integration with Neo4j graph database.
- ‘spring-boot-starter-validation’ that ensures that data adheres to predefined constraints.
- ‘spring-boot-starter-web’ which helps in building web applications and RESTful services.
- ‘springdoc-openapi-starter-webmvc-ui’, which makes the documentation of the REST API possible.
- Other useful tools such as spring-boot-devtools, lombok and spring-boot-configuration-processor greatly enhance developer productivity, making their work more effective while speeding up development cycle.

- Handling JSON data was made possible by ‘sprint-boot-starter-json’. While our tests are built upon TDD using ‘sprint-boot starter-test’.
- More dependencies including ‘tinkergraph-gremlin’, ‘xmlunit-core’ and ‘xmlunit-matchers’ facilitate graph operations as well as XML manipulation and inspection.

The unit test coverage of over 90% in our service layer is a testament to the fact that rigorous testing and quality assurance are key components of the development process. The careful selection of Spring Boot dependencies ensures not just functional but also highly standards compliant prototype, leading to a software that is robust, testable, and maintainable. The prototype attempts through these purposeful strategic decisions to effectively historize CMs and therefore make a reliable reference for future knowledge graph development. Please note that 90% was set as a lower bound for covering the unit tests of the service layer, and most services even have 100% coverage. Only few services contain code that is not testable via unit testing due to involving database connections and objects that are too heavy to mock. However, key functional requirements must be covered within the unit testing.

A critical service method, the `persistCommit` is responsible for persisting changes in a graph to the repository. The method starts by fetching the last nodes and their commits in order to determine the old metadata ID if it exists. A map is then created to trace the types of commits; all vertices from provided graph are retrieved with current state being compared to the previous one in order to identify any differences. It then decides whether each node/edge that requires creating, updating, or deleting based on those differences. If there were no previous metadata, all vertices are marked as new creations. The resulting metadata of the new graph is saved before each vertex is processed, and if changed, updates or creates these commits in what they exist in repository. Lastly, deletions are handled that result in relationship persisted and ensure that the repository contains an accurate reflection of state relative to its graph see Algorithm 1.

A historization system prototype with various data structures is defined using the Spring data Neo4j. The main entities used in this case are `Commit`, `Entity`, `Metadata`, and `Relation`.

The class for `Commit` represents a commit within the graph database that holds changes on the entity over time. It contains these following fields `id` which is a unique identifier of the commit, `dateTime` which is a timestamp of this commit formatted and serialized using Jackson annotations, `type` which can be either `CREATE`, `UPDATE`, `DELETE`, `UNTOUCHED` stored within an enum `CommitType`, `metadataId` being an ID of metadata associated with the `Commit` and finally, `targetEntity` referring to the affected entity by this commit. This class uses the following annotations: `@RelationshipProperties` indicates that it holds properties of relationship and `@JsonIgnoreProperties` configures Jackson to ignore unknown JSON properties during serialization/deserialization. Moreover, Lombok annotations are employed for generating getters, setters as well as `equals`/`hashCode` methods.

Algorithm 1 persistCommit

Require: Graph *graph*, String *msg*, String *author*, String *modelId***Ensure:** ServiceException may be thrown

```
1: Fetch latest nodes and their commits
2: Get old metadata ID if available
3: Create a map for commit types
4: Retrieve vertices from the graph
5: if old metadata exists then
6:   Compare current graph with old graph to identify differences
7:   for each difference do
8:     Determine change type (create, update, delete)
9:   end for
10: else
11:   Mark all vertices as new creations
12: end if
13: Persist new graph's metadata
14: for each vertex do
15:   Create or update commit
16:   Save commit to the repository
17: end for
18: for each entry in commit type map do
19:   if entry is marked for deletion then
20:     Handle deletion commit
21:   end if
22: end for
23: Persist graph relations
```

CREATE, UPDATE, DELETE, and UNTOUCHED are the four types of commits that can occur as defined by the `CommitType` enum. It has a static method called `getFrom(String value)` that takes in a string argument and returns its corresponding instance of `CommitType`.

An instance of the class `Entity` represents a node within the graph database. Its fields include `id`, which is a unique identifier for the entity; `labels`, a dynamic list of these labels for this entity; `properties`, which map key-value properties associated with the entity; `relations`, an arraylist contains relations to other entities; and finally, there are various commits lists representing a historical view of changes effected on this entity. The class also has methods such as `addProperties()`, `addCommits()`, `addRelations()` and etc., annotated with Lombok annotations to facilitate generation of setters, getters, equals/hashcode methods, etc. The class is annotated using `@Node` to indicate it's a node entity in Neo4j and `@JsonIgnoreProperties` so that Jackson will ignore unknown JSON properties.

The `Metadata` class serves as a representation of graph metadata. This includes the following fields `id`, that is the unique identifier for this metadata; `modelId`, which is the ID of the model corresponding to the graphML file; `graphMlBase64` a string with base64 encoded xml content, `message` the associated message, and `author` the name of its creator. The class contains annotation `@Node` that makes it a node entity in Neo4j and another annotation `@JsonIgnoreProperties` that allows Jackson to ignore not recognized JSON properties. Lombok annotations are used to create getter methods, setter methods, and `equal/hashcode`.

The `Relation` class represents a relationship between two entities on a graph. It contains such fields like `id`, the property responsible for unique identification of this relation; `targetEntity` which is an end point of connection and properties that may contain key-value pairs related to this specific link between elements. There is also a method called `getRelations()` for acquiring all relationships from graph along with their properties. It is annotated with `RelationshipProperties` showing that it has relationship properties and `JsonIgnoreProperties` indicating that Jackson should omit unrecognized JSON properties as well. Lombok annotations enable automatic generation of getter, setter, equals, and hash code functions.

These data structures in general, are meant to simulate a graph database with entities `Entity`, relationships `Relation` and the commit history `Commit`. Each of these classes has annotations for Neo4j graph mapping and JSON serialization/deserialization that simplifies the management of the graph state and its evolution over time. Enums `CommitType` and composite properties `properties` ensure adaptable and comprehensive representation of data.

Our prototype's frontend framework is Angular because it can produce interactive and responsive web applications. The main reason why Angular is preferred over others is that it enables quickening of development pace in addition to increasing the user experience with component-based architecture. This method assists app developers in creating applications which include reusable UI aspects and as a result, make development and maintenance easier. Here is a list of angular dependencies:

- Foundational libraries like '@angular/core', '@angular/common' and '@angular/forms' that allow for creation of scalable web apps.
- The Angular Material Components package provides a full suite of component-based modules with Material Design implementations throughout its components.
- We went for visualization tools 'neovis.js' and 'vis-network' due to their effective representation of complex knowledge graph relationships that can be easily understood at first sight.
- Reactive programming capabilities provided by 'rxjs' help manage asynchronous data streams, improving application reactivity.

The fact that Angular Material has been included is because it makes the UI development easier and also ensures an accessible and consistent design language across the prototype. For instance, this library permits developers to use many HTML components directly from a comprehensive selection of Angular Material, thus simplifying development while concentrating on unique features of the prototype.

In addition, Angular services are critical in the frontend architecture since they function as go betweens for reaching out to the backend. Services are responsible for retrieving and submitting data, resulting in a smooth connection between the frontend user interface and the backend logic.

What is more important, is that our choice of Angular as well as its ecosystem of dependencies reflects our aspiration to develop a functional and efficient frontend that is also capable and easy to navigate. Therefore, this framework serves as a solid basis for constructing a conceptual model historization tool that has all necessary features required by its users, such as visualizing, filtering, searching, diff tracking, etc.

Crucially, Docker comes into play when we design the full stack of our prototype ranging from the frontend and backend to databases. Docker is a platform that enables us to encapsulate our application together with its environment into containers. This approach drastically simplifies deployment, maintains consistency across different environments, enhances scalability, and provides isolation.

The management of this multi-container setup for our application that has angular frontend, spring boot backend, and Neo4j database is done using Docker images. Docker allows us an easy way to define and run multi-container applications with Docker. With a YAML file configuring the services, networks, and volumes of this application, launch it as just a single command in our entire stack through Docker compose. It makes managing each component separately less complicated than ever before.

That is what explains why we use Docker:

- Simple development process, it becomes easier to build and test the app locally when using Docker Compose since developers can clearly define the application stack as well as its dependencies.
- Unified environment, Docker helps in making the run-time environment standardized across different machines by containerizing the application. Therefore, it eliminates the issue of 'it works on my machine', providing a reliable way to test and deploy.
- Fast deployment, with ready-made configurations, it's possible to have a prototype up and running in no time through Docker compose orchestration. Starting from installing databases to setting up services required.
- Containerization security/isolation, isolating containers improves security processes, hash conflicts between service endpoints.

- Expandability, to make scaling easy, Docker and Docker compose are used; more instances of a service can be added to handle more load, hence improving how well it can handle additional users or data volumes.

Docker compose in our project is responsible for coordinating communication between the Angular frontend that provides the UI and the Spring Boot backend that takes care of business logic as well as data processing, and the Neo4j database, which is the home of a KG. As such, all parts of this system work together to produce a reliable solution to historize CM.

The Docker and Docker compose indicate our interest in the latest technology in order to enhance development life cycle from coding to deployment. This way we are able to ensure that our application becomes resilient, scalable and easy to deploy.

What we do in this project is to make use of GitHub for code management and version control, which serves as a secure collaborative tool that supports our software development procedure. Through the use of GitHub, team members can work together effectively regardless of their geographical locations. It has an excellent feature set for code versioning that lets us keep track of every kind of alteration made to the codebase. This is necessary when debugging, figuring out how the project has evolved, and attributing changes to someone or another. Hosting it on GitHub guarantees that all our codes are located in a secured place where other team members can access them anytime improving continuity and recoverability of the project. Github also helps centralize project management by serving as a platform used to document projects, issues tracking, and feature requests making simpler organization tasks. This goes beyond just coding since not only does it enable the team to concentrate on development priorities, but it also ensures quality coders put together our prototypical implementation for historizing conceptual models using a knowledge graph.

5.2 User Manual

In order to speed up the launch and management of a full-stack application for historization of CM on KG, Makefile is used. This approach makes it more convenient for users with Docker installed in their system when it is time to launch a particular application. The official documentation is available here [Docker's official documentation](#).

The use of Makefile simplifies the setup, boot sequence, and supports the accessibility and manageability of the prototype. In this setup there are also Docker, Swagger for API documentation and user-friendly frontend that make this system comprehensible by end-users; from submitting concept models to be historized up to watching them evolve over time with Neo4j Browser.

5.2.1 Access Points

The frontend of the application may be accessed at 'http://localhost:80' upon going live. The web interface is a friendly way for users to access and interact with the application.

The Backend and API documentation services can be reached through 'http://localhost:8080/swagger-ui.html'. This also serves as a place for Swagger powered API documentation. It allows to understand what endpoints are accessible and how they work. Swagger enables users to send HTTP requests from their browsers, without requiring any additional tools or plug-ins, that would trigger historization on the frontend UI, see more information in 5.2.2.

Neo4j Database Interface: The database API is located at 'http://localhost:7687,' while the Neo4j Browser interface allows direct interactions with Neo4j database through http://localhost:7474. To access the Neo4j Browser, use username 'neo4j' and password 'password'.

5.2.2 Rest API

When it comes to developing web services using Spring Boot, Swagger is a tool that simplifies the documentation process by allowing developers to do it with ease. The updated version of Swagger, which is referred to as Swagger 3 in the context of this document generation, can still generate interactive API docs. This particular feature facilitates an easy and accurate description of RESTful APIs including all available endpoints, parameters, and expected responses.

Swagger plays an important role in providing documentation for back-end services through our prototype. This documentation is not only for internal developers but also meant for external users who interact with the API. For instance, the implementation of Swagger on Spring Boot 3 is aided by libraries such as Springfox or newer ones like OpenAPI-based Springdoc which configures automatically Swagger UI according to the code.

Here are some of the key Features and Benefits we think of when we swagger in our prototype.

- **Interactive Documentation:** Swagger produces a user interface that presents information about all API endpoints. It allows users to know what functionalities are available on the backend services without having to spend time on the code base.
- **API Testing:** The good thing about Swagger is that it allows you to test API endpoints directly from your browser. Just one click on your computer and you can call an API and see how it responds in real-time – no separate tools required. This feature is very nice especially when demonstrating how an API handles conceptual models like handling and historizing them.

- **Clarity and Accessibility:** Using swagger, we ensure that our API is understood by everyone involved, from developers to stakeholders. This transparency plays a crucial role in successful collaboration and integration efforts.
- **Error Reduction:** Early detection during the development or testing phases of applications is very important for identifying inaccuracies/errors concerning inputs/outputs associated with APIs due to their interactive nature of swagger.

For our case, the Swagger UI is used in the Spring Boot 3 backend to enable detailed documentation of each endpoint related to historization of conceptual models. The Swagger UI is accessible at a fixed URL (usually 'http://localhost:8080/swagger-ui.html' on local development environments) and allows users to explore different API operations such as POST, GET, DELETE, and PUT. These operations allow for submission of new KGs, retrieval of stored KGs, modification of present KGs among others.

In addition, the prototype exploits Swagger's capacity to connect with the API directly from the documentation page. This implies that end users or developers can actually test the endpoints as they are updated by themselves, and these tests provide instant feedback about how the API behaves and also about what is produced during historization process. Users can for example use the swagger UI to upload a KG and initiate a POST request so that they can see how it gets processed and historized within that knowledge graph.

We will list all existing endpoints, their parameters, body, and responses for our Swagger API documentation to describe thoroughly. This kind of organized documentation is helpful for API users and even developers.

5.2.3 API Endpoints

Base URL

The base URL for all API requests is:

`http://localhost:8080`

Fetch All Commits

Method: GET

URL: `/v1/commit`

Description: Retrieves all commits based on provided datetime or filter criteria.

Parameters: • `dateTime` (Optional): A `LocalDateTime` string in the format `YYYY-MM-DDThh:mm:ss`.

- **filter** (Optional): A string in the format `key==value`, where the key includes options such as `{from, to, author, create, update, delete}`. The value for `from` and `to` should be a `LocalDateTime` formatted as `YYYY-MM-DDThh:mm:ss`. The value for `author` can be any string, and the values for `create`, `update`, `delete` are boolean, either `true` or `false`, with a default of `false` if not specified. Note that all key-value pairs are optional, and if the filter is left empty, all commits will be returned.

Responses:

- 200 OK: Returns an array of commit objects.
- 400 Bad Request: Occurs when input format is wrong.
- 404 Not Found: if no commits match the given parameters.

```
[
  {
    "id": 1,
    "dateTime": "2023-04-12T15:20:30",
    "type": "CREATE",
    "metadataId": 101,
    "targetEntity": {
      "id": 5001
    }
  }
]
```

Post a Commit

Method: POST

URL: `/v1/commit`

Description: Submits a file as a commit with all related metadata information using multipart/form data. Please note that the file must be KG in graphML XML format.

Parameters:

Example Success Response: `message: Required. Descriptive message about the commit.`

- **author:** Required. Name or identifier of the commit author.

Request Body:

Content-Type: `multipart/form-data`

Required: `file` (binary)

Responses:

200 OK: Commit successfully created.
 400 Bad Request: Input validation failed.
 404 Not Found: No specified resource was found.

Example Success Response:

```
{
  "message": "Commit created successfully."
}
```

Fetch Metadata by ID

Method: GET

URL: /v1/metadata/{id}

Description: Retrieves metadata by the specified ID, including the model file in Base64 format.

Parameters:

- **id:** Required. The integer ID of the metadata.

Responses:

200 OK: Successfully retrieved metadata.
 400 Bad Request: Invalid ID format.
 404 Not Found: No metadata was found with the given ID.

Example Success Response:

```
{
  "id": 101,
  "modelId": "Model123",
  "graphMlBase64": ["data:base64..."],
  "message": "Initial Commit",
  "author": "DeveloperA"
}
```

Fetch Diff between Metadata IDs

Method: GET

URL: /v1/diff/{id1}/{id2}

Description: Fetches the difference between models associated with id1 and id2.

Parameters:

- **id1:** Required. The integer ID of the first metadata.

- **id2:** Required. The integer ID of the second metadata.

Responses:

200 OK: Successfully retrieved the differences.

400 Bad Request: One of the IDs is invalid.

404 Not Found: One or both IDs do not exist.

Example Success Response:

```
{
  "diffs": {
    "propertyChanges": [
      {
        "property": "name",
        "oldValue": "Old Name",
        "newValue": "New Name"
      }
    ]
  },
  "desc": "Differences detailed here."
}
```

Fetch Distinct Node IDs

Method: GET

URL: /v1/node

Description: Retrieves all distinct entity IDs from the system without duplication. This endpoint is particularly useful for obtaining a comprehensive list of all unique entities currently managed within the knowledge graph.

Operation ID: getNodes

Responses:

- **200 OK:** Successfully retrieves an array of unique entity IDs. The response includes a list of all entity IDs present in the system, ensuring that each ID is distinct and no duplicates are provided.
- **Any Failure:** Returns an error response with details about the failure.

```
{
  "description": "failure operation",
  "content": {
    "*/*": {
      "schema": {
        "$ref": "#/components/schemas/ErrorResponse"
      }
    }
  }
}
```



```

    }
  }
}

```

- **200 Successful Operation:**

```

{
  "description": "successful operation",
  "content": {
    "*/*": {
      "schema": {
        "type": "array",
        "items": {
          "type": "string"
        }
      }
    }
  }
}

```

It is important that we provide all the information and instructions on how to interact with the API, and then demonstrate how a user can perform operations involving version control or metadata management of our prototype. As such, the documentation here provides insight into the way in which users can deploy concepts embodied in the version control and metadata administration sections of their content while using knowledge graphs in our prototype. In this light, this is a major reason why this guide appears to be very easy for anyone intending to apply it effectively, as they can still have an effective interaction with the system.

Optionally, you can use the swagger provided web ui to send RESTful API requests see Figure 5.2, and most important functionality is committing files, which can for now only be performed using RESTful API and can be also performed using swagger web ui, where input field and file picker can be filled based on the user need and once execute is clicked a http request will be sent to the backend using RESTful API see Figure 5.3

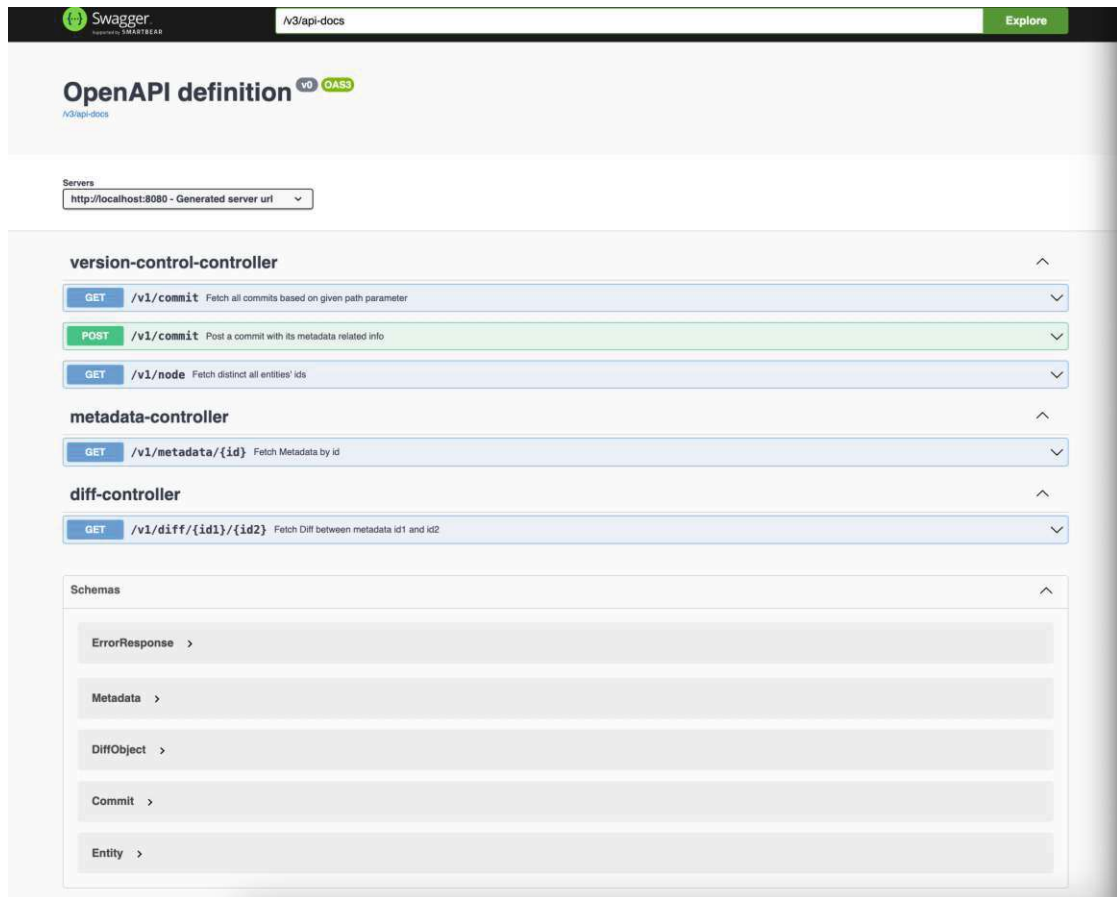


Figure 5.2: Swagger web ui page

5.2.4 Web UI

Two key navigation features are supported by the prototype. The first one, model search, serves as the default home page (see Figure 5.4). Here, the entire model elements are presented as a single node in the user interface to maximize readability and to facilitate useful feedback from changes. Each node is connected to another node via a Commit relation in the UI. Please note that the structure in the graph database is as previously described in earlier sections. Additionally, hovering over the Commit relation will display the Commit's metadata, including counts of created, updated, deleted, or untouched nodes, with updates in yellow, creations in blue, deletions in red, and untouched nodes without color to enhance readability (see Figure 5.7).

In the visual area where nodes are displayed, the history chain is sorted by date in ascending order. Clicking on any node (since each node represents a model) on the page will display the graphml.xml file of the clicked model in a text area below the visual area of the nodes (see Figure 5.5).

The screenshot shows the Swagger UI for the `POST /v1/commit` endpoint. The title bar indicates the method and path, followed by a brief description: "Post a commit with its metadata related info". Below this, a sub-description states: "Post a file as a commit with all related metadata info as multipart/form-data".

The **Parameters** section contains two required query parameters:

Name	Description
message * required string (query)	message
author * required string (query)	author

Buttons for **Cancel** and **Reset** are located to the right of the parameters table.

The **Request body** section shows a dropdown menu set to **multipart/form-data**.

Below the request body, there is a **file** * required field of type `string(binary)`. It includes a **Choose File** button and the text "No file chosen".

A large blue **Execute** button is at the bottom of the form.

Figure 5.3: Swagger web ui commit functionality

The model search page also contains three vertical forms. The first form allows us to compare two existing models from the history. In this form, the ID shown in the frontend is entered in the first model input field and another ID in the second model input field. Clicking the "Compare Diff" button reveals a text area for model differences (see Figure 5.6), where messages detailing unmatched elements from the graphml files for both models are shown side-by-side; the left one corresponds to the first model input, and the right one to the second input, the inner text of the files highlighted in green for addition and red for deletion, please note that an update is a combination of a deletion and an addition. In case of invalid input, such as a negative number (-1) or a nonexistent ID (e.g., 100), a feedback error message "Model's id needs to be picked from below and have different values" appears as a snackbar with a close button or a timeout of 5 seconds. Clicking the "Clear" button resets all input fields in the first vertical form and the displayed text areas.

The second vertical form pertains to filtering Commit functionality, where a date range for the history and a specific author who committed the model can be specified, along with the type of change on the model history (i.e., created, updated, or deleted node). By default, if no filter is provided, then filtering for that specific group of filters is disabled, where three groups exist: date range, author, and create/update/delete options. For instance, if all filter groups are specified with the date range 3/3/2024–3/9/2024, author mustafa, and only the Commit type Create ticked, the result will return all Commits between these dates and all Commits published by Mustafa (case-insensitive), and only Commits of the type 'Create' (see Figure 5.8), please note that the operation between filter groups is a logical AND operation. Clicking "Clear" will reset all groups in the form.

The users can upload a commit to the backend server through the third vertical form as a

POST request. This involves adding a new node model to update the chain in the model nodes view area. There are three fields required by this form namely: author, message and file. The author field is for indicating who made the commit, the commit description should be filled into message field and the file picker should be filled with the conceptual model file. It should be noted that only GraphML format files are allowed here and so the picker has been set up in such a way that it accepts anything with .graphml extension. The upload action will be triggered once the button commit is clicked, if the fields are empty a validation error message will be shown and required fields will be marked in red. In case of clicking cancel, the all fields will be cleared and the whole form will be reset, see Figure 5.9.

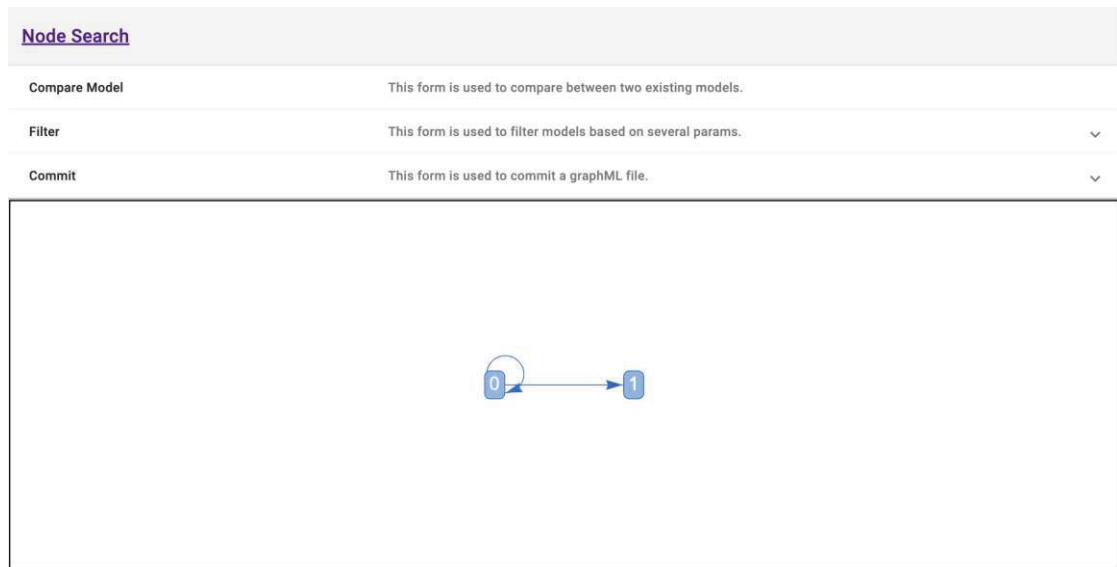


Figure 5.4: Model Search View

The second navigation feature is the node search page, which you can access by clicking the link back from the model search page at the top section. The node search page includes a form with two input fields; “Node ID” and “Query Limit,” located next to each other, and two buttons; one labeled “Reset” and another with a label “Search”, positioned on the right. This page shows each node of the model as an independent node connecting them through a history chain referred to as Commit. Moreover, it displays relations that are part of the actual node relationships in the model itself.

By default, the first node is selected with a limit of 50. The “Node ID” field offers a dropdown with auto-completion having all existing nodes on it. Any selection from this drop-down menu will show the history and all its relationships with other nodes and history chain relation Commit. For example, in Figure 5.10, if we consider n_0 with a limit of 50 as the default selected node, only 50 nodes are displayed which include n_1 and n_2 as they have historical relation links with the selected n_0 . However, assuming that there exists a node n_3 , where there is no relationship between n_3 node in the

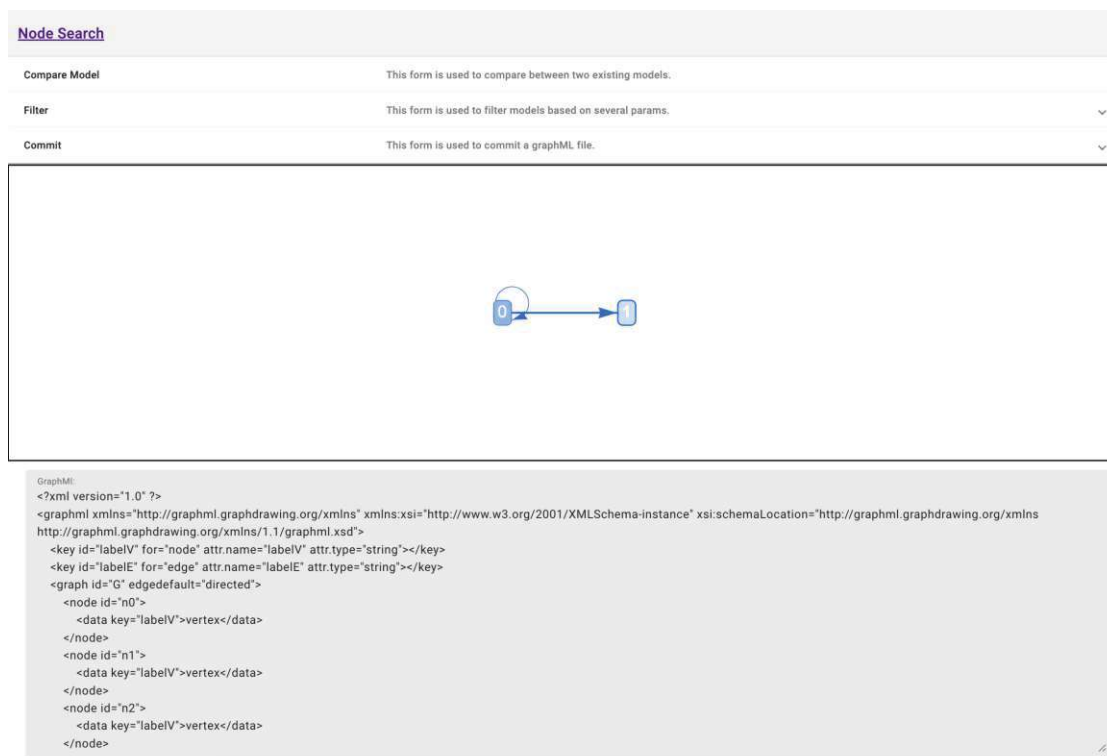


Figure 5.5: Model Search View, node clicked

model and `n_0`, then node `n_3` will not be displayed.

This page keeps track of how model nodes change over time, so it shows which nodes are related to each other in a model at different times. In the history view, the first element is said to have a relation `Commit` to itself. If this element is connected to other nodes, then these connections and corresponding nodes will be shown too. A node that only has `Commit` relations representing it means that there are no other relevant model's node relationships for that node.

In the next `Commit`, if something changes, then also all relationships with end nodes will be duplicated from the previous `Commit`. If a relationship exists in the preceding node but not in the current one, it is considered deleted in the uploaded version of a model. Similarly, adding new relationships is tracked too.

If an updated end node of a relationship is reached, then details about this updated end node appear on the following node connected by `Commit` relation(s). As your model grows bigger you might need to increase query limit to prevent un-rendered nodes and relationships due to Node Query Limit while expanding models.

5. PROTOTYPE IMPLEMENTATION

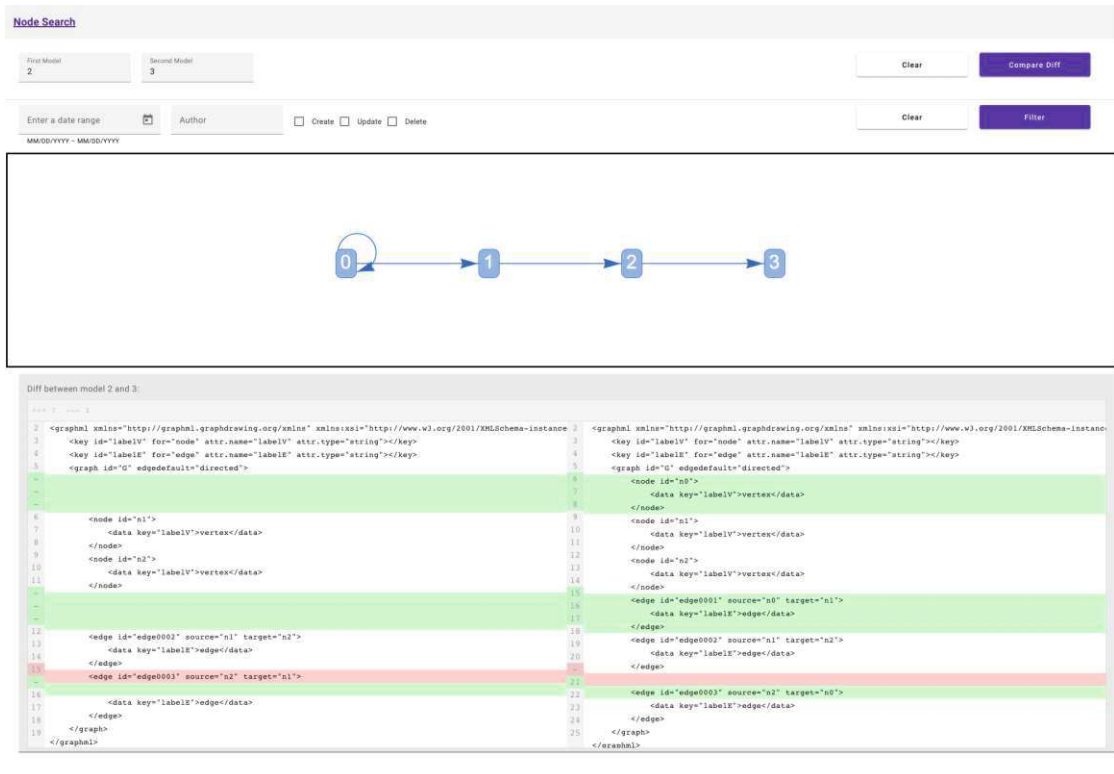


Figure 5.6: Model Search View, compare two models

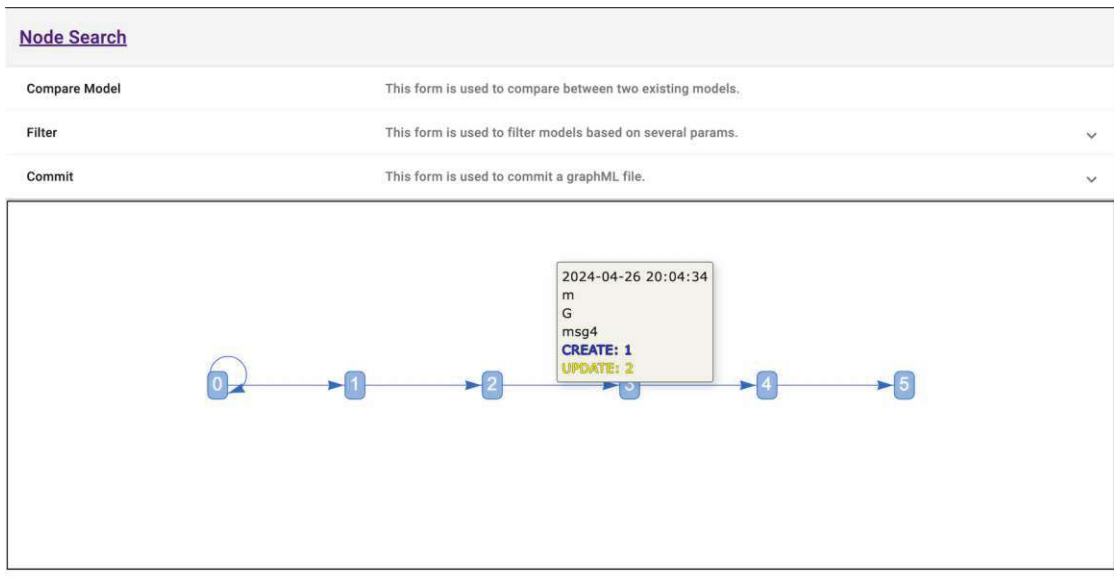


Figure 5.7: Model Search View hover on a commit

Node Search

Compare Model

This form is used to compare between two existing models.

Filter

This form is used to filter models based on several params.

Enter a date range

MM/DD/YYYY – MM/DD/YYYY

Author

☒ Create ☐ Update ☐ Delete

Clear

Filter

Commit

This form is used to commit a graphML file.




Figure 5.8: model Search View, filter example

5. PROTOTYPE IMPLEMENTATION

Node Search

Compare Model This form is used to compare between two existing models.

Filter This form is used to filter models based on several params. ▾

Commit This form is used to commit a graphML file. ^

Author* Mustafa

Message* first commit

Choose File | 1 model-driv..._v2.graphml

Cancel Commit




Figure 5.9: model Search View, commit example

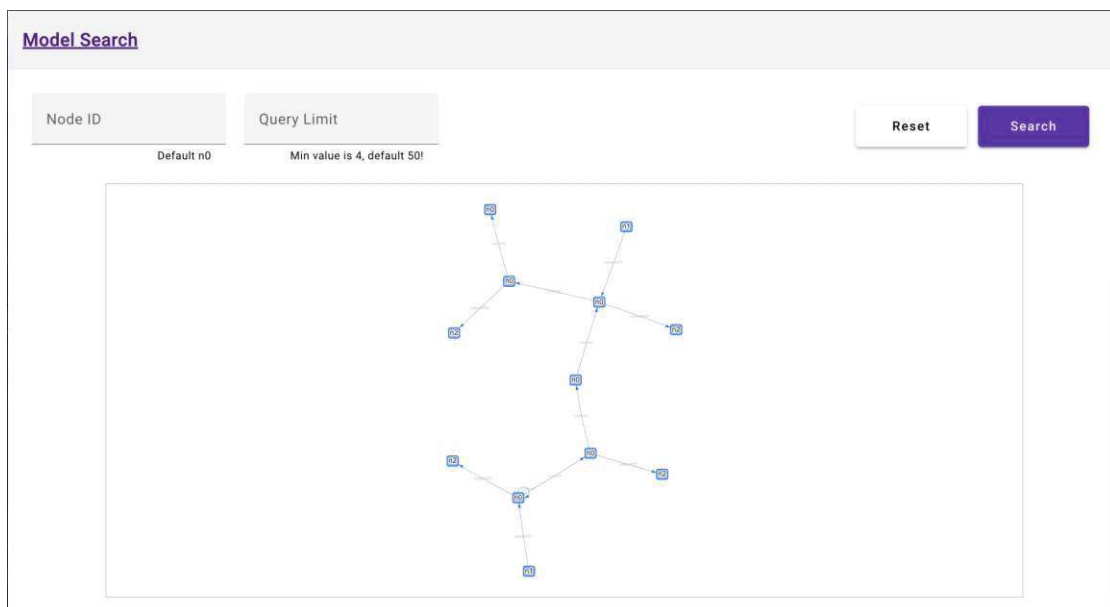


Figure 5.10: Node Search View

CHAPTER 6

Evaluation

To ensure that the web prototype meets all the specified functional, non-functional requirements and properly historizes any arbitrary CM (represented as a KG in GraphML), we have to start with a clean prototype deployment and delete file 'data' from where docker-compose.yml is located. It will not be there if this is the first time it is running. In Linux/Unix, use the shell command; in Windows, use the batch command for application deployment according to Section 5.2. Let us begin with these acceptance test structures in the web browser under `http://localhost:80` with the following test data, which cover most possible test cases to prove that the provided prototype fulfills the goal of the thesis.

The test data [1..5].graphml, which can be found in the appendix of the transformed directory, were used to evaluate the implemented prototype for historizing conceptual models using knowledge graphs. These datasets were designed to cover a range of scenarios, ensuring that the prototype meets the specified stakeholder requirements.

Test Data One: Initial Simple Graph, the first test data starts with a basic graph consisting of two nodes connected by one edge with a weight of 2.5. This simple configuration serves as a foundational scenario to verify that the prototype can correctly handle basic graph structures and attribute management.

Test Data Two: Adding a Node and an Edge, the second dataset extends the initial graph by adding a new node and a new edge. This test ensures that the prototype can handle incremental changes to the graph structure, maintaining the integrity and consistency of the graph.

Test Data Three: Updating an Edge and Adding Another Edge, the third dataset adds a third edge connecting the last node (n2) with the first node (n0) and updates the weight of the first edge (e0) to 10. This scenario tests the prototype's ability to handle updates and more complex changes within the graph, ensuring correct versioning and consistency of the graph.

Test Data Four: Deleting a Node, the fourth dataset deletes the first node (n0) and consequently removes its connecting edges to maintain a valid XML structure. This test evaluates the prototype's capability to handle deletions and their cascading effects on the graph structure.

Test Data Five: Adding a Node Attribute, the fifth dataset adds a color attribute to a node, introducing additional complexity by incorporating node attributes. This test ensures the prototype can handle and correctly store node attributes within the knowledge graph.

The test data sets were selected to reflect the stakeholder requirements for historizing knowledge graphs. Here's how each requirement is addressed:

- **Storing Snapshots and Versions:** The test data allow for creating and managing multiple versions of graph structures, ensuring the prototype can store and retrieve historical snapshots.
- **Comparison and Traversal of History:** By incrementally modifying the graph and updating attributes, the test data provide a basis for comparing different versions and traversing the graph's history.
- **Web Application Upload and Tracking:** The datasets simulate real-world scenarios where users upload new versions, track changes, and compare different versions of the graph.
- **Filtering Versions Based on Metadata:** The metadata attributes and incremental changes in the test data allow users to filter versions by criteria such as authors, type of change, and date.
- **Feasibility of Historizing Functionality:** The diverse scenarios covered by the test data ensure that the prototype's functionality is robust and feasible, even without focusing heavily on performance optimization.

By addressing these scenarios, the test data provide a comprehensive evaluation of the prototype, ensuring it meets the specified requirements and is capable of handling real-world applications effectively.

We need to perform six Commits using the form Commit in the Web browser under `http://localhost:80`. The first Commit with test date one, the second Commit with test date two, the third Commit with test date three, the forth Commit with test date four, and the fifth Commit with test date one and the sixth Commit with test date five. Once all of these Commits are posted and with the UI Subsection 5.2.4 details, asserting that the following section features with acceptance criteria can be fulfilled. Moreover, it is necessary to use also to go through the use case scenario in the next section.

6.1 User Acceptance Testing

Feature: GraphML File Upload and Commit Logging

Given the user is on the 'Model' page.

When the user uploads a GraphML file and provides an author name and commit message in the Commit form on the web UI.

Then the following acceptance criteria should be met:

- Success in uploading: No errors should be shown when users upload the files of GraphML.
 - *Functional Requirements*: Ensure that only one instance of the XML file is saved in the database per commit request to avoid duplication.
 - *Non-Functional Requirement*: Ensure the functionality is feasible without focusing much on performance.
- Author and Message: An author name and a commit message must be included with every upload; both should not allow null or empty values.
 - *Functional Requirements*: Enhance the commits user interface by adding more features for better usability.
 - *Non-Functional Requirement*: Implement the new design of the application to improve functionality and user experience.
- Logging of Commit: Once uploaded, the system records a commit using given author and message.
 - *Functional Requirements*: Ensure that updates propagate throughout the entire model to maintain consistency. Implement functionality to save the graph ID for tracking and referencing purposes.
- Validation of file: It should only accept valid GraphML files. Any file that does not meet the requirements of GraphML schema must be rejected.
 - *Functional Requirements*: Develop mechanisms to handle API exceptions gracefully, ensuring system stability. Implement features to handle conflicting changes to maintain data integrity.

Feature: Advanced Filtering Options

Given the user wants to search through the model's commit history.

When the user applies filters such as date range, author name, or commit type in the filter form on the web UI.

Then the following acceptance criteria should be met:

- **Filtering Function:** The history of commits should be filterable by date, author, and commit type.
 - *Functional Requirements:* Add filtering features to allow users to sort and view data based on specific criteria.
- **Proper Filtering:** The filtering has to match commits that meet the given filter form.
 - *Functional Requirements:* Add the ability to filter commits based on various parameters for better management of changes.
- **UI Design:** It must be intuitive and clearly labeled so users can easily understand how filters work.
 - *Functional Requirements:* Improve the user interface to provide a more intuitive and efficient experience for users interacting with knowledge graphs.
 - *Non-Functional Requirement:* Implement the new design of the application to improve functionality and user experience.
- **Validation and Feedback:** The filter will not allow invalid input, and fields will be marked in red if invalid.
 - *Functional Requirements:* Develop comprehensive unit tests to ensure code quality and functionality across the system.

Feature: Model Comparison and Visualization of Differences

Given multiple model versions are available.

When the user selects two models and initiates a comparison in the Compare Model form on the web UI.

Then the following acceptance criteria should be met:

- **Selection Capability:** Users should be able to select two different model versions for comparison.
 - *Functional Requirements:* Integrate different libraries to support the comparison and differentiation of graph versions.
- **Diff Accuracy:** The system should accurately display the differences in GraphML format between the selected models.
 - *Functional Requirements:* Enhance the representation of nodes and edges for clarity and usability.

- **Highlight Changes:** Changes should be highlighted clearly, with deletions, additions, and modifications easily distinguishable.
 - *Functional Requirements:* Implement a timeline view to visualize the history of knowledge graphs' modifications, displaying changes chronologically.
- **Validation and Feedback:** If models' IDs are invalid, useful feedback will be shown.
 - *Functional Requirements:* Develop mechanisms to handle API exceptions gracefully, ensuring system stability.

Feature: Visualization of Model History as a Connected Node Graph

Given the user is reviewing the model history.

When the user observes the visual representation of the model's evolution.

Then the following acceptance criteria should be met:

- **Complete Display:** The entire history of a model should be visible as a chain of nodes, each representing a commit of the whole GraphML file.
 - *Functional Requirements:* Implement a timeline view to visualize the history of knowledge graphs' modifications, displaying changes chronologically. Design the system so that each node represents a model, enhancing modularity.
- **Chronological Order:** Nodes must be displayed in chronological order of their commits.
 - *Functional Requirements:* Implement a timeline view to visualize changes over time, aiding in historical analysis.
- **Interactivity:** Users should be able to interact with the chain, possibly clicking on individual nodes to get more details.
 - *Functional Requirements:* Enhance the user interface for viewing the history of knowledge graphs, making it more user-friendly.

Feature: Display of Metadata on Graph Interactions

Given the graph of the history of the model is presented.

When the user hovers over or interacts with nodes or edges within the graph.

Then the following acceptance criteria should be met:

- **Display Metadata on Hover:** When a user hovers over a commit relation, metadata such as the commit date, message, and number of changes (updates, deletes, creates, untouched) should be displayed.

- *Functional Requirements*: Add hover titles on edges instead of nodes for better user experience.
- Readability: The displayed information should be clear and easy to read.
 - *Functional Requirements*: Enhance the frontend by adding more attributes per model to provide detailed information.

Feature: Detailed View of GraphML Content on Node Selection

Given the user selects a node within the model's history graph.

When the user clicks on a node to explore its details.

Then the following acceptance criteria should be met:

- Read GraphML: Clicking on one node in the view shows the corresponding GraphML file as text in the WebUI.
 - *Functional Requirements*: Add support for reading XML files in the post endpoint to enhance data input capabilities.
- Clear GraphML: Clicking on any location on the view clears GraphML from the WebUI.
 - *Functional Requirements*: Enhance the user interface for viewing the history of knowledge graphs, making it more user-friendly.
- Data Integrity: The displayed GraphML should accurately reflect the file as committed at that node.
 - *Functional Requirements*: Develop comprehensive unit tests to ensure code quality and functionality across the system.
- User Interface: The GraphML display area should be well-integrated into the UI, providing a seamless user experience.
 - *Non-Functional Requirement*: Implement the new design of the application to improve functionality and user experience.

Feature: Detailed History Visualization

Given that the user has a GraphML file that represents the knowledge graph.

When the user navigates to the 'Node' page.

Then the following acceptance criteria should be met:

- **Graph Visualization:** The system should display the graph with all included nodes and their relationships, allowing the user to see a full vision of the model's history.
 - *Functional Requirements:* Implement a timeline view to visualize changes over time, aiding in historical analysis.
- **Default Node Selection:** The system should pre-select the default node ID corresponding to the first node upon file upload
 - *Functional Requirements:* Implement functionality to save the graph ID for tracking and referencing purposes.
- **Node Tracking:** When a user selects a node ID from the form, the system should display all related nodes connected to the picked node.
 - *Functional Requirements:* Implement the ability to view the history of the knowledge graph, providing a comprehensive overview of changes.
- **Limit Setting:** The user should be able to set a limit on the number of related nodes to display, with a system-enforced minimum limit of 4 and a default limit of 50.
 - *Functional Requirements:* Add the ability to filter the history of a knowledge graph by date range, user, or type of change for detailed analysis.

Feature: Node-specific History Tracking

Given that the user wants to track changes to a specific node within the graph.

When the user inputs a node ID in the provided form.

Then the following acceptance criteria should be met:

- **Node Detail Display:** The system should display the detailed history and relations of the selected node.
 - *Functional Requirements:* Implement functionality to handle the deletion of nodes or relations.
- **Connected Node Visualization:** The system should visualize all nodes related to the selected node up to the specified limit.
 - *Functional Requirements:* Create an endpoint to construct the GraphML file from a specific version for accurate version tracking.
- **Commit Chain:** The system should show how the nodes are connected using the commit relation, in a way it creates a chain history out of these commits.

- *Functional Requirements*: Implement the ability to export and import the history of a knowledge graph to/from a file for backup and migration purposes.

In the software development process, it is important to have an acceptance testing phase which assesses whether a system meets the specifications and is ready for use. The Table 6.1 shows several tests being associated with their fulfillment status, giving a wide view about how much readiness the system is and points out certain areas that might be addressed further if need be.

Table 6.1: Test Cases and Fulfillment Status

Test Case	Description	Fulfillment Status
Success in Uploading	No errors should be shown when users upload the files of GraphML.	Met
Author and Message	An author name and a commit message must be included with every upload; both should not allow null or empty values.	Met
Logging of Commit	Once uploaded, the system records a commit using given author and message.	Met
Validation of File	It should only accept valid GraphML files. Any file that does not meet the requirements of GraphML schema must be rejected.	Met
Filtering Function	The history of commits should be filterable by date, author, and commit type.	Met
Proper Filtering	The filtering has to match commits that meet the given filter form.	Met
UI Design	It must be intuitive and clearly labeled so users can easily understand how filters work.	Met
Validation and Feedback	The filter will not allow invalid input, and fields will be marked in red if invalid.	Met
Selection Capability	Users should be able to select two different model versions for comparison.	Met
Diff Accuracy	The system should accurately display the differences in GraphML format between the selected models.	Met
Highlight Changes	Changes should be highlighted clearly, with deletions, additions, and modifications easily distinguishable.	Met
Complete Display	The entire history of a model should be visible as a chain of nodes, each representing a commit of the whole GraphML file.	Met
Chronological Order	Nodes must be displayed in chronological order of their commits.	Met

Continued on next page

Table 6.1 – continued from previous page

Test Case	Description	Fulfillment Status
Interactivity	Users should be able to interact with the chain, possibly clicking on individual nodes to get more details.	Met
Display Metadata on Hover	When a user hovers over a commit relation, metadata such as the commit date, message, and number of changes (updates, deletes, creates, untouched) should be displayed.	Met
Readability	The displayed information should be clear and easy to read.	Met
Read GraphML	Clicking on one node in the view shows the corresponding GraphML file as text in the WebUI.	Met
Clear GraphML	Clicking on any location on the view clears GraphML from the WebUI.	Met
Data Integrity	The displayed GraphML should accurately reflect the file as committed at that node.	Met
User Interface	The GraphML display area should be well-integrated into the UI, providing a seamless user experience.	Met
Graph Visualization	The system should display the graph with all included nodes and their relationships, allowing the user to see a full vision of the model's history.	Met
Default Node Selection	The system should pre-select the default node ID corresponding to the first node upon file upload.	Met
Node Tracking	When a user selects a node ID from the form, the system should display all related nodes connected to the picked node.	Met
Limit Setting	The user should be able to set a limit on the number of related nodes to display, with a system-enforced minimum limit of 4 and a default limit of 50.	Met
Node Detail Display	The system should display the detailed history and relations of the selected node.	Met
Connected Node Visualization	The system should visualize all nodes related to the selected node up to the specified limit.	Met
Commit Chain	The system should show how the nodes are connected using the commit relation, in a way it creates a chain history out of these commits.	Met

6.2 Use Case Scenario

With the former test data provided, once all six Commits are posted, we can now go deep into our application to check exactly what has happened. In the Web UI, the 'Model'

page under `http://localhost:80/model` will display six nodes shown in the visual area of the page. These nodes are enumerated with IDs from zero to four. Hover on the first Commit, which is also recognized by a commit relation to itself; then the commit should show in blue CREATE and next to it number two, referring to the number of newly created nodes from uploading the first GraphML file. Note that the UPDATE here does not exist as all nodes are new; in the backend application, the CREATE type always overwrites the UPDATE type.

Now, hovering on the commit arrow between node zero and node one will show three types, namely CREATE, UPDATE, and UNTOUCHED, with number one indicating that the model encountered by the second CREATE an UPDATE, a CREATE, and an UNTOUCHED node within the same model in history. Because checking the test data two, we can mark an addition of node 'n2', triggering a CREATE state and adding a new edge 'e1', which will cause node 'n1' to have an UPDATE state, while node 'n0' did not encounter any changes, thus triggering the UNTOUCHED state within the model history.

Hovering on the third Commit, we can see that there are three UPDATES noted. This is because the test data three have added a new edge 'e2', making the node 'n2' and 'n0' marked as UPDATE, since both share the same edge. Edge 'e0' is also updated with weight 10, resulting in an UPDATE in nodes 'n0' and 'n1'. In conclusion, all three nodes are marked as UPDATE.

Now when we hover on the fourth Commit, we can see two UPDATES and one DELETE. This is because checking the test data four, we can note that node 'n0' is deleted, and as an effect, edges 'e0' and 'e2' are also removed from the test data; otherwise, the schema of the GraphML file will not be valid. Since edge 'e0' is shared with node 'n1', the node 'n1' is marked as UPDATE due to the deletion of edge 'e0'. Analogously for node 'n2' and edge 'e2' deletion, resulting in one DELETE due to 'n0' and two UPDATES because of 'n1' and 'n2'.

Hovering on the fifth Commit, we can note that all three states are there except UNTOUCHED, since all three nodes are touched. Analyzing the test data one, since this was the fifth Commit post, we can see the return of 'n0' as a CREATE state, an update of 'n1' as an UPDATE state, due to the creation of edge 'e0'. The deletion of node 'n2' as a DELETE state. Finally, hovering on the sixth Commit shows one update and one untouched state, the update is caused due to adding an attribute to the node 'n0'.

Working with the initial test data and six Commit posts in the Neo4j database, shown in Figure 6.1, each node keeps a series of Commits representing its evolution over time. If we run the query `MATCH (n) WHERE n.properties.id = 'n0' RETURN n LIMIT 25`, we will see node *n0*, depicted in Figure 6.2. This node begins with a Commit relation to itself that indicates the start of its creation. The last link in this chain is a node with no outgoing Commit relationships.

Once you click on Commit, it will show the properties with `metadataId: 11, type: CREATE`. In this case, `metadataId` refers to properties that have been published by other

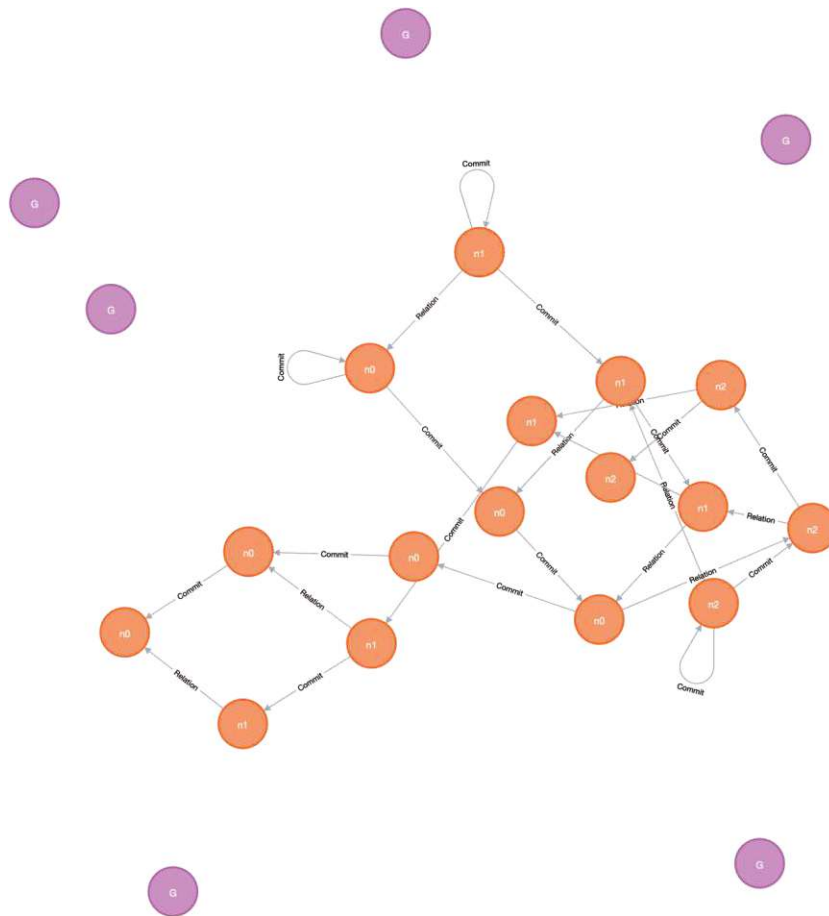


Figure 6.1: Neo4j graph look for the use case scenario six commit posts

commits and are common across the model so that duplication is avoided in related nodes by moving metadata to an external node. The *type* property records what state of the node this Commit has resulted to; in our situation it is CREATE. If we fetch the Metadata node using `id=11` which was fetched before from selected commit's property, we will find below properties:

```
<id>: 11
author: Alice
graphMlBase64: [object Object]
message: Alice's first commit message
modelId: G
```

These properties define the metadata: *id* refers to the metadata identifier, *author* to the person who committed, *message* to the commit message, and *modelId* to the identifier of the GraphML file, ensuring different GraphML files do not get mixed up.

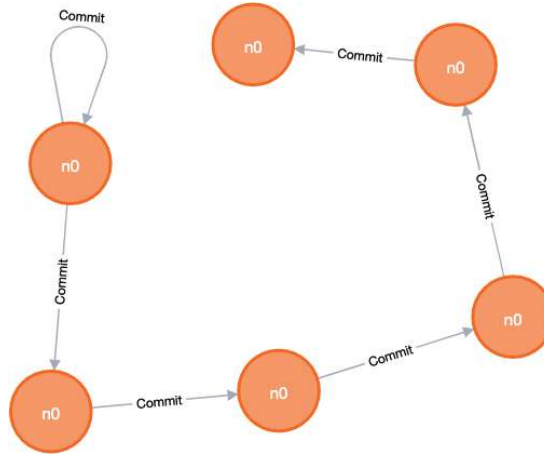


Figure 6.2: Neo4j graph look for the use case scenario six commit posts and filtering node $n0$

Returning to Figure 6.1, we note that between Commits, there are different types of relationships labeled as *Relation*. This custom label is used by the backend server to identify model-related relationships for tracking their evolution. A similar setup is provided in the Web UI Node page at <http://localhost:80/node>, where the node id can be selected. For example, selecting $n0$ displays the graph shown in Figure 6.3.

The view is as follows: the starting node $n0$ begins with a Commit relation to itself. At this point, $n0$ has a relationship $e0$ to $n1$. The next Commit leaves $n0$ unchanged, as the second set of test data does not affect node $n0$. A subsequent Commit of type *Update* adds a new edge connection to $n2$ with $e2$. Hovering over the following Commit, marked as *Delete*, shows that there are no further relations, consistent with the removal of node $n0$ to maintain compliance with the GraphML schema. The final Commit, marked as *Create*, indicates that $n0$ is added again, also displaying its relation to node $n1$ with $e0$. Finally hovering the next commit, will show an *Update* state, which refers to the updated node $n0$ with a new attribute color, which can be seen by hovering on the last node $n0$. Similarly for edges, when hovering on model related relations the attribute of the edges will be shown, i.e. in case of $e0$ the weight will be displayed.

6.3 Performance Evaluation

This section examines how well the prototype performs uploading and querying GraphML files of different versions, file sizes, and complexities. The performance metrics collected include loading time measured in milliseconds (ms) and memory usage measured in megabytes (MB). The used files for testing the performance are the following: **Small** (2 kilobytes (KB)), **Medium** (68 KB), and **Large** 253 (KB) can be found under the performance testing graphml files directory in the appendix.

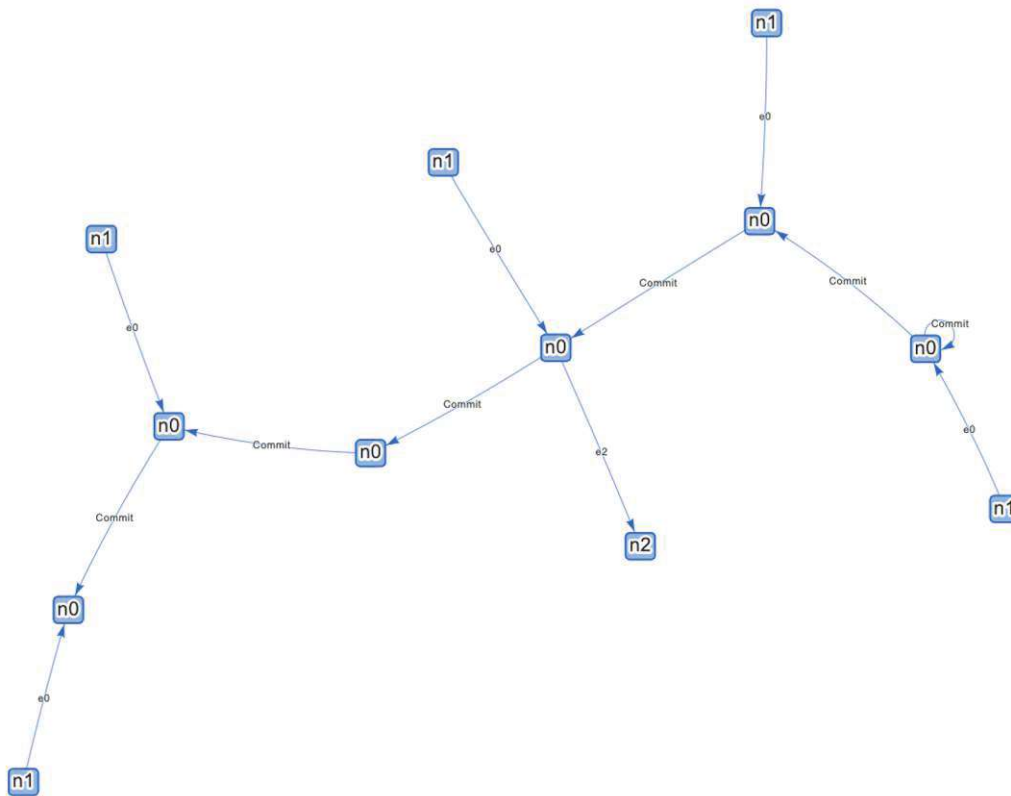


Figure 6.3: Node page view section

File Size	Version	Upload Time (ms)	Upload Memory (MB)	Query Time (ms)	Query Memory (MB)
2 KB	v1	308	76	0	76
2 KB	v2	1492	64	0	64
2 KB	v3	733	73	0	74
68 KB	v1	12055	64	0	64
68 KB	v2	6185	73	0	73
68 KB	v3	5748	72	0	72
253 KB	v1	11285	59	0	60
253 KB	v2	12595	80	0	80
253 KB	v3	11558	59	0	59

Table 6.2: Performance Metrics for GraphML Files

Analysis

From the performance metrics captured in Table 6.2 the following analysis can be concluded:

Upload Performance

Upload time significantly differs across versions for the small file (2 KB). Version 1 has an upload time of 308 ms, Version 2 takes longer at 1492 ms and Version 3 is at 733 ms.

Memory usage fluctuates a little but hovers around 64-76 MB. Often, the increase in upload time for Version 2 indicates such an anomaly or additional processing that could be associated with this specific version.

Regarding the medium file (68 KB), newer versions lead to a decrease in upload times. While Version 1 uses up to 12055 ms, in Version 2 it falls down to 6185 ms and then further reduces to just about half of that value in version 3 it takes 5748 ms. The memory usage remains pretty much consistent between about 64 and 73 MB. The drop off from Versions One through Two and Three, signifies optimizations or improvements relating on how medium sized files are handled.

As for the large file (253 KB), upload times slightly vary but remain high. For example, Version 1 takes as much as 11285 ms while version 2 has an additional 10 ms and finally version 3 needs only 11558 ms. Memory usage varies with the highest being observed for version 2 which recorded approximately 80 MB. The constant high upload times imply that large files usually need more processing power inherently.

Query Performance

In all versions and file sizes, query times are constant at around zero milliseconds (note that zero is around by the measured program, the actual time was slightly above zero in nano seconds). Memory usage is static for every file size; the small file takes around 64-76 MB, medium-sized occupies 64-73 MB while large-size ranges from 59 to 80 MB. This implies that regardless of the size of the GraphML file, efficient handling of queries occurs over it because their execution time remains constant. During queries, memory consumption is consistent which means that the prototype can handle changing data sizes without significant performance degradation.

Conclusion and Future Work

7.1 Discussion

The prototype's development and testing show that KGs can serve as a historiographical tool for conceptual models. This means they provide the structures necessary for capturing how these models change over time in a dynamic way. Using KGs enables users to have a richer semantic representation of changes. This not only preserves history about modified items but also makes it easier for people to access and understand old information. Therefore, this is particularly useful in the software engineering domain where understanding the path of changes is crucial to ongoing maintenance or development of systems.

One major advantage of using KGs for model historization is their ability to link data correctly reflecting the complicated relationships found within any given CMs. Therefore, we can consider each change event as being multi-faceted through this approach; thus supporting complex queries that may cut across versions while tracing back on how the model has evolved over different stages. Additionally, KGs are flexible enough to accommodate various forms of alteration without breaking the original context or structure integrity much, so that there still remains a strong version control mechanism.

Another limitation is that currently, the prototype only concentrates on some features and functions while ignoring many possible versions' features such as branching, stashing, merging, etc., or conflict resolution requirements that may arise in a real-world environment.

For use in practical settings, one needs to consider whether the scalability of this prototype will enable it to perform well under real-world conditions as expected by users. That means being able to tackle huge datasets together with complicated model structures alongside integration into existing systems as well as workflows. Also, making sure that the system is user-friendly while still accommodating different individuals' specific

requirements within organizations should be taken care of since without these factors success might not be achieved neither would adoption happen easily.

7.2 Future Work

Any future work needs to look at the following for making the prototype more functional and useful:

Include sophisticated version control mechanisms, conflict detection techniques, branching, stashing, merging, forking and rebasing is very important. This will allow many revisions of the system during development which can be done simultaneously by different developers working on it.

There is a big opportunity here for us to build a whole system off the back of that success. It should cover CM2KG and all that we have done so far. This system will enable people to post their CMs as they are, still being able to historize them without following KG's format due to the mechanism of transforming the CMs to KGs provided by the tool CM2KG [SB21a, SAB24].

One challenge with model versioning is how to deal with conflicts when they occur between two different modifications made concurrently. The prototype could be modified by adding conflict resolution capabilities since KGs are structured systems that can handle data consistency and integrity in case of conflicting changes made by various people at once.

The flexibility and user-friendliness of the system can be greatly increased by adding de-transformation functionality i.e., KG2CM so that users can view CMs in their original format. This will allow people to work with historical data in a way they understand best which is very important if you want users adoption and actual utility.

7.3 Conclusion

The primary goal of the thesis was to develop a prototype that can historize CMs using (KGs), and this has been achieved. KGs have shown themselves to be capable of managing and representing how a model changes over time, which is important for many applications where versioning or historical changes need to be considered. The proposed approach has been implemented successfully, thus providing a concrete basis for further investigations in this area.

This study discussed the challenges faced when historizing CMs, highlighting the complex steps involved in maintaining semantic consistency and data coherence. Despite these difficulties, the prototype was successful in demonstrating its capability to provide a structured, queryable format for historical model data accessibility and usability. This not only enhances technical understanding in modeling historization but also contributes towards software engineering practice and data management systems.

Traditional version control systems are far behind this prototype, which integrates KGs because they can only handle CMs with less detail, semantics, and context awareness about the histories of changes made to them. This thesis lays down a firm basis for further investigation into model historization, identifying various benefits brought about by this development as well as possible improvements aimed at making the effective management of evolving models easier than before.

The performance testing showed that the uploading time differed from one version to another and depending on the size of the files, while query times remained consistently optimal. Moreover, troubleshooting those abnormalities and optimizing such functions as big file handling can contribute towards boosting the prototype's efficiency.

List of Figures

2.1	UML Class diagram example.	11
2.2	GraphML example having edges with weights and nodes with colors . . .	14
2.3	One tier client-server architecture	19
2.4	Two tier client-server architecture [Kum19]	19
2.5	Three tier client-server architecture [Kum19]	20
2.6	N tier client-server architecture [Kum19]	20
3.1	Specifying graphically the three used theoretical design approaches for model- ing evolution [PvdW95].zoi8z7uj	28
4.1	Github's labels types statistics as Pie Chart	38
4.2	Conceptual idea of generic historization.	44
4.3	Conceptual idea of KG historization base on deriving snapshots from element evolutions.	45
4.4	Graphml historization example of three versions in a neo4j database . . .	47
4.5	Graphml historization example of four versions in a neo4j database	48
5.1	High Level overview of the full stack application architecture	50
5.2	Swagger web ui page	62
5.3	Swagger web ui commit functionality	63
5.4	Model Search view from the UI of the implemented prototype for historizing KG	64
5.5	Model Search View, node clicked	65
5.6	Model Search View, compare two models	66
5.7	Model Search view from the UI of the implemented prototype for historizing KG	66
5.8	model Search View, filter example	67
5.9	model Search View, commit example	68
5.10	Node Search view from the UI of the implemented prototype for historizing KG	68
6.1	Neo4j graph look for the use case scenario six commit posts	79
6.2	Neo4j graph look for the use case scenario six commit posts and filtering node n0	80
6.3	Node page view section	81
		87

Glossary

Commit Special type of graph database relation label used to define a new element within a new snapshot.. 45, 46, 48, 62–65, 70, 77–80, 89

CREATE State property included in the relationship Commit, which refers to a node creation. 45–47, 78, 79

Cypher Neo4j query language. 15, 49

DELETE State property included in the relationship Commit, which refers to a node deletion. 45–47, 78

Entity Special type of graph database node label used to identify models' nodes.. 46, 47

Metadata Special type of graph database node label used to store common versions' metadata i.e. file, author, etc.. 46

Relation Special type of graph database relation label used to store all types of model relation generically. 46, 47

UNTOUCHED State property included in the relationship Commit, which refers to a node holding its previous state. 45–47, 78

UPDATE State property included in the relationship Commit, which refers to a node update or its relationships update. 45–47, 78

Acronyms

AS	Abstract Syntaxes.	33
CASE	Computer Aided Software Engineering.	9
CLI	command-line interface.	24
CM	Conceptual Model.	ix, 1, 2, 5, 6, 10, 37, 42, 43, 49, 51, 55, 69, 83–85
CS	Concrete Syntaxes.	33
EA	Enterprise Architecture.	3, 7
ER	Entity Relationship.	29
GraphML	Graph Markup Language.	3, 13, 43, 46, 64, 69, 71–74, 76–80
HATEOAS	Hypermedia as the Engine of Application State.	21
HTTP	Hyper Text Transfer Protocol.	19–22
ID	identifier.	47
IDE	Integrated Development Environment.	50
IT	Information Technology.	1, 7, 8
ITA	Information Technology Architecture.	1, 2, 5
KG	Knowledge Graph.	ix, 3, 5, 12, 15, 37, 42–49, 55, 57, 58, 69, 83–85, 87
MBSE	Model-Based Systems Engineering.	29, 30
MDA	Model Driven Architecture.	9
MDE	Model Driven Engineering.	1, 2, 5, 6, 8, 9
MDK	Model Development Kit.	29

MMS Model Management System. 29

NFR Non-Functional Requirement. 30

OMG Object Management Group. 9

OpenMBEE Open Model-Based Systems Engineering Environment. 29, 30

RESTful API Representational State Transfer. 21, 22, 61

SelfAS Self Adaptive Systems. 30, 31

SOA Service Oriented Architecture. 8

SSL Secure Sockets Layer. 21

SysML System Modeling Language Models. 30

TCP Transmission Control Protocol. 19

TDD Test-Driven Development. 49

TLS Transport Layer Security. 21

UML Unified Modeling Language. 10, 11

URI Uniform Resource Identifier. 21

URL Uniform Resource Locator. 21

VC Version Control System. ix, 1, 2, 16, 27

XML Extensible Markup Language. 13, 47

Bibliography

- [ABK⁺09] Klaus Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why model versioning research is needed!? an experience report. In *Proceedings of the MoDSE-MCCM 2009 Workshop @ MoDELS 2009*, 2009.
- [AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), feb 2008.
- [BDW16] John D Blischak, Emily R Davenport, and Greg Wilson. A quick introduction to version control with git and github. *PLoS computational biology*, 12(1):e1004668, 2016.
- [BELP10] Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. Graph markup language (graphml). In Roberto Tamassia, editor, *Handbook of graph drawing and visualization*, pages 517–541. Chapman & Hall, London, 2010.
- [BHB22] Robin Bråtfors, Simon Hacks, and Dominik Bork. Historization of enterprise architecture models via enterprise architecture knowledge graphs. In Balbir S. Barn and Kurt Sandkuhl, editors, *The Practice of Enterprise Modeling*, pages 51–65, Cham, 2022. Springer International Publishing.
- [BKL⁺12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. *An Introduction to Model Versioning*, pages 336–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [BR10] Stephan Bode and Matthias Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture*, pages 182–197, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series). *J. Database Manag.*, 10, 01 1999.
- [Cha76] Donald D Chamberlin. Relational data-base management systems. *ACM Computing Surveys (CSUR)*, 8(1):43–66, 1976.

- [Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36, 1976.
- [CJX20] Xiaojun Chen, Shengbin Jia, and Yang Xiang. A review: Knowledge reasoning over knowledge graph. *Expert Systems with Applications*, 141:112948, 2020.
- [DeM11] Tom DeMarco. Structured analysis and system specification. In *Software pioneers: contributions to software engineering*, pages 529–560. Springer, 2011.
- [DHK11] Remco Dijkman, Jorg Hofstetter, and Jana Koehler. *Business Process Model and Notation*, volume 89. Springer, 2011.
- [EE05] JHP Eloff and MM Eloff. Information security architecture. *Computer Fraud & Security*, 2005(11):10–16, 2005.
- [EK03] Amnon H Eden and Rick Kazman. Architecture, design, implementation. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 149–159. IEEE, 2003.
- [Eme15] Robert Wall Emerson. Convenience sampling, random sampling, and snowball sampling: How does sampling affect the validity of research? *Journal of visual impairment & blindness*, 109(2):164–168, 2015.
- [EPR⁺23] Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, and Matthias Tichy. A new versioning approach for collaboration in blended modeling. *Journal of Computer Languages*, 76:101221, 2023.
- [Fav04] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd workshop in software model engineering, wisme*, pages 262–271. Citeseer, 2004.
- [Fis88] Alan S Fisher. *CASE (Computer Aided Software Engineering): using software development tools*. John Wiley & Sons, Inc., 1988.
- [Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., USA, 3 edition, 2003.
- [FS04a] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*, 2004.
- [FS04b] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. 01 2004.

- [GDBG18] Antonio García-Domínguez, Nelly Bencomo, and Luis H. Garcia Paucar. Reflecting on the past and the present with temporal graph-based models. *CEUR Workshop Proceedings*, 2245:46–55, November 2018.
- [Git] Git VCs. <https://git-scm.com>. Accessed on [10. Jan 2024].
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
- [Has18] Wilhelm Hasselbring. *Software Architecture: Past, Present, Future*, pages 169–184. Springer International Publishing, Cham, 2018.
- [HBC⁺21] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d’Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *ACM Computing Surveys (Csur)*, 54(4):1–37, 2021.
- [HH18] Ramzi Haraty and Gongzhu Hu. Software process models: A review and analysis. *International Journal of Engineering and Technology(UAE)*, 7:325–331, 05 2018.
- [JB21] Lvar Jacobson and James Rumbaugh Grady Booch. The unified modeling language reference manual. 2021.
- [Keh15] Timo Kehrler. *Calculation and propagation of model changes based on user-level edit operations : a foundation for version and variant management in model-driven engineering*. PhD thesis, Universität Siegen, 2015.
- [Kha14] Shahid N Khan. Qualitative research method: Grounded theory. *International journal of business and management*, 9(11):224–233, 2014.
- [KMLS18] Mohamad Kassab, Manuel Mazzara, JooYoung Lee, and Giancarlo Succi. Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, 14(4):263–271, December 2018.
- [Kos07] Lasse Koskela. *Test driven: practical tdd and acceptance tdd for java developers*. Simon and Schuster, 2007.
- [Kum19] Santosh Kumar. A review on client-server based applications and research opportunity. *International Journal of Scientific Research*, 10:33857–33862, 08 2019.
- [Laa17] Sjaak Laan. *IT Infrastructure Architecture-Infrastructure Building Blocks and Concepts Third Edition*. Lulu. com, 2017.
- [Lan18] Marc Lankhorst. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Springer, 2018.

- [Liu14] Yufan Liu. A survey of persistent graph databases. Master's thesis, Kent State University, 2014.
- [LK22] Phillip A Laplante and Mohamad Kassab. *Requirements engineering for software and systems*. Auerbach Publications, 2022.
- [LL09] Kathryn B Laskey and Kenneth Laskey. Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(1):101–105, 2009.
- [LSL19] Malgorzata Lazarska and Olga Siedlecka-Lamch. Comparative study of relational and graph databases. In *2019 IEEE 15th International Scientific Conference on Informatics*, pages 000363–000370. IEEE, 2019.
- [Mac12] Linda A Macaulay. *Requirements engineering*. Springer Science & Business Media, 2012.
- [Man93] M Morris Mano. *Computer system architecture*. Prentice-Hall, Inc., 1993.
- [McF76] John H. McFadyen. Systems network architecture: An overview. *IBM Systems Journal*, 15(1):4–23, 1976.
- [McG01] Matthew John McGill. Uml class diagram syntax: An empirical study of comprehension. 2001.
- [MD08] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof?-a review of experiences from applying mde in industry. In *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings 4*, pages 432–443. Springer, 2008.
- [Oli07] Antoni Olivé. *Conceptual Modeling of Information Systems*, pages 1–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [Par15] Avi Parush. Chapter 8 - summary of the components of the conceptual model according to the layered framework. Morgan Kaufmann, Boston, 2015.
- [PCSF08] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion: next generation open source version control*. " O'Reilly Media, Inc.", 2008.
- [Pok15] Jaroslav Pokorný. Graph databases: Their power and limitations. In Khalid Saeed and Wladyslaw Homenda, editors, *Computer Information Systems and Industrial Management*, pages 58–69, Cham, 2015. Springer International Publishing.
- [PSA⁺01] David Powell, RJe Stroud, A Adelsbach, D Alessandri, C Cachin, S Creese, M Dacier, Y Deswarte, K Kursawe, JC Laprie, et al. Conceptual model and architecture. *Department of Computing Science Technical Report Series*, 2001.

- [PvdW95] Henderik Alex Proper and Theo P van der Weide. A general theory for evolving application models. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):984–996, 1995.
- [RAB⁺15] Stewart Robinson, Gilbert Arbez, Louis G Birta, Andreas Tolk, and Gerd Wagner. Conceptual modeling: definition, purpose and benefits. In *2015 winter simulation conference (wsc)*, pages 2812–2826. IEEE, 2015.
- [RN12] Marko A Rodriguez and Peter Neubauer. The graph traversal pattern. In *Graph data management: Techniques and applications*, pages 29–46. IGI global, 2012.
- [Rup10] Nayan B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, jan 2010.
- [RW11] Nick Rozanski and Eoin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2011.
- [RWE15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015.
- [SAB24] Muhamed Smajevic, Syed Juned Ali, and Dominik Bork. Cm2kg^{cloud} - an open web-based platform to transform conceptual models into knowledge graphs. *Sci. Comput. Program.*, 231:103007, 2024.
- [SB21a] Muhamed Smajevic and Dominik Bork. From conceptual models to knowledge graphs: A generic model transformation platform. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*, pages 610–614. IEEE, 2021.
- [SB21b] Muhamed Smajevic and Dominik Bork. Towards graph-based analysis of enterprise architecture models. In Aditya K. Ghose, Jennifer Horkoff, Vítor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings*, volume 13011 of *Lecture Notes in Computer Science*, pages 199–209. Springer, 2021.
- [Sch97] Ken Schwaber. Scrum development process. In *Business Object Design and Implementation: OOPSLA'95 Workshop Proceedings 16 October 1995, Austin, Texas*, pages 117–134. Springer, 1997.
- [SHB21] Muhamed Smajevic, Simon Hacks, and Dominik Bork. Using knowledge graphs to detect enterprise architecture smells. In *Proceedings of the 14th IFIP Working Conference, PoEM 2021, Riga, Latvia, November 24-26, 2021*, pages 48–63. Springer International Publishing, 2021.

- [Sie21] Daniel Siegl. Bridging the gap between openmbee and git. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 465–466, 2021.
- [SK77] Edgar H Sibley and Larry Kerschberg. Data architecture and data model considerations. In *Proceedings of the June 13-16, 1977, national computer conference*, pages 85–96, 1977.
- [SR07] Carolyn Strano and Qamar Rehmani. The role of the enterprise architect. *Information Systems and e-Business Management*, 5(4):379–396, 2007.
- [Var08] Jinesh Varia. Cloud architectures. *White Paper of Amazon, jineshvaria. s3. amazonaws. com/public/cloudarchitectures-varia. pdf*, 16, 2008.
- [WBFG03] Roel J. Wieringa, H. M. Blanken, M. M. Fokkinga, and P. W. P. J. Grefen. Aligning application architecture to the business context. In Johann Eder and Michele Missikoff, editors, *Advanced Information Systems Engineering*, pages 209–225, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [Zaf21] Christina Zafeiriou. Literature review and example implementation on knowledge graphs. 2021. Accessed: February 4, 2024.
- [ZC06] Ping Zhang and Gary Chartrand. Introduction to graph theory. *Tata McGraw-Hill*, 2:2–1, 2006.