

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dependency Management in the Development of Complex Component-based Systems

Filipe Rodrigues Fonseca



Master in Informatics and Computing Engineering

Supervisor: Bruno Miguel Carvalhido Lima

Co-supervisor: João Carlos Pascoal Faria

July 15, 2025

Dependency Management in the Development of Complex Component-based Systems

Filipe Rodrigues Fonseca

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Professor Ana Cristina Ramada Paiva

Referee: Professor Miguel António Sousa Abrunhosa Brito

Supervisor: Professor Bruno Miguel Carvalhido Lima

July 15, 2025

Resumo

Cada vez mais, os sistemas de software modernos são modulares e distribuídos, desenvolvidos por equipas distintas que trabalham de forma independente e operam em ambientes complexos. Neste contexto, manter o controlo das dependências entre componentes torna-se fundamental, especialmente no que diz respeito à estabilidade do sistema, à redução da dívida técnica e à capacidade de adaptação rápida a mudanças. Esta dissertação investiga de que forma as bases de dados em grafo, em particular o Neo4j, podem contribuir para a gestão de dependências num cenário real de larga escala, como é o caso do projeto VTV, desenvolvido pela Celfocus.

O principal objetivo do estudo foi aplicar um modelo baseado em grafos para mapear as dependências de software entre três equipas distintas envolvidas no projeto VTV. Após a análise das limitações dos métodos anteriormente utilizados, foi proposta uma solução que combinava uma instância centralizada de Neo4j, para representar as dependências de alto nível entre equipas, com instâncias específicas por equipa para modelações mais detalhadas a nível interno. Foi também desenvolvido um sistema de controlo de versões baseado em Git, com o objetivo de garantir a integridade dos dados do grafo e facilitar a atualização contínua, alinhada com metodologias ágeis.

Para validar o sistema, foi adotada uma abordagem mista, com recurso a questionários e entrevistas informais, centrando-se especialmente na equipa que adotou o Neo4j numa fase inicial. Os resultados demonstraram que a ferramenta melhorou a visibilidade das dependências, incentivou uma comunicação mais proativa e auxiliou na análise de impacto durante o desenvolvimento e o processo de deployment. No entanto, continuaram a existir desafios relacionados com a formação de novos utilizadores e com a definição do nível de detalhe adequado na modelação.

Conclui-se, assim, que a gestão de dependências baseada em grafos, quando integrada nas rotinas diárias de equipas ágeis, pode melhorar significativamente a estabilidade do software e a colaboração entre equipas. A dissertação realça ainda a importância da articulação organizacional e da manutenção de documentação adequada para garantir a sustentabilidade da adoção da ferramenta a longo prazo.

Abstract

More and more, modern software systems are modular and distributed, built by different teams that work independently and run in complicated environments. In this situation, keeping track of dependencies between components is very important, especially when it comes to keeping things stable, lowering technical debt, and allowing for quick changes. This dissertation looks into how graph databases, especially Neo4j, can help with managing dependencies in a real-world setting on a large scale, like the VTV project made by Celfocus.

The study's main goal is to use a graph-based model to keep track of software dependencies between three separate teams working on the VTV project. After looking at the problems with current methods, a solution was suggested that used a centralized Neo4j instance for high-level inter-team dependencies and team-specific instances for more detailed internal modeling. A version control system based on Git was created to keep graph data safe and make it easier to make updates all the time, in line with agile workflows.

We used a mix of questionnaires and informal interviews to test the system, focusing on the team that was the first to use Neo4j. The results showed that the tool made things easier to see, encouraged proactive communication, and helped with impact analysis during development and deployment. But it was still hard to make sure that new users were trained and that modeling was done at the right level of detail.

This dissertation comes to the conclusion that using graph-based dependency management in the daily lives of agile teams can greatly improve the stability of software and the ability of teams to work together. It also shows how important it is for organizations to work together and keep records to keep long-term adoption going.

UN Sustainable Development Goals

This is an analysis of how the research conducted for this dissertation advances the Sustainable Development Goals (SDGs) of the the United Nations. This dissertation’s focus is on enhancing software engineering practices in an organizational setting, but it also touches on a number of SDG targets due to its emphasis on efficiency, innovation, and collaborative development processes.

The primary SDG to which this dissertation contributes is SDG 9 – Industry, Innovation and Infrastructure, with additional alignment to SDG 8 – Decent Work and Economic Growth and SDG 12 – Responsible Consumption and Production.

SDG 9 – Industry, Innovation and Infrastructure This dissertation encourages creativity by presenting Neo4j, a graph-based method for managing software dependencies. The adoption of sustainable, effective engineering practices and the modernization of IT infrastructure are supported by the use of such tools in industrial settings.

SDG 8 – Decent Work and Economic Growth The work reduces time lost to deployment failures and misalignments and helps create more productive working environments by addressing inefficiencies in cross-team communication and dependency resolution.

SDG 12 – Responsible Consumption and Production Reducing the use of redundant resources, such as cloud infrastructure and energy related to system failures and rollbacks, is indirectly aided by encouraging software systems that are effective and maintainable.

To evaluate the extent to which this project contributes to the identified SDG targets, several performance indicators and metrics can be considered:

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.5	Promotes innovation through the use of graph databases for managing software dependencies, modernizing internal infrastructure and increasing visibility across teams.	Number of teams adopting Neo4j. Number of nodes and relationships created and maintained. Reduction in deployment-related incidents caused by unresolved dependencies.
8	8.2	Contributes to more efficient work by reducing friction in deployment and debugging processes, improving team communication, and reducing onboarding time.	Developer feedback on clarity of system architecture. Time saved during impact analysis. Average onboarding duration before and after Neo4j adoption
12	12.6	Encourages structured documentation of system dependencies, traceability via GitHub, and reduction in computational waste due to failed deployments.	Versioned Cypher scripts maintained via GitHub. Number of failed deployments avoided. Frequency of Neo4j update contributions by team members.

While the core contributions of this dissertation are situated within the domain of software engineering and organizational process improvement, their alignment with several SDG targets underlines the broader significance of designing sustainable, collaborative, and transparent systems in technology-driven industries. As dependency management becomes increasingly vital in large-scale software development, the lessons and practices proposed in this work contribute not only to better technical outcomes but also to a more resilient and sustainable digital infrastructure.

Acknowledgements

Five years ago, I entered the faculty for the first time, thinking, "I finally made it". Little did I know all the suffering, successes, failures, and growth that would mark my journey. But once again, here I am, thinking I finally made it. Hopefully, this time, this thought will also mean a new beginning.

To my supervisors, Bruno Lima and João Pascoal Faria, thank you for helping me overcome the last step of my academic journey, for the help given during the last year on multiple occasions, and for showing nothing but support.

To Bárbara Ramos and Diana Marques, thank you for accepting my thesis proposal at Celfocus and for all the support and advice. Without you by my side during the numerous meetings on the project, this thesis would not have gone forward.

To my friends, thank you for always being with me, both in the good and bad times. During all these years, I thought multiple times that this day would not come. Only with the confidence you gave me was I capable of surpassing everything.

Last, but certainly not least, to my parents and sister. Thank you for all the support given to me since the beginning. Thank you for understanding my need to focus on my studies instead of spending as much time as I should with you. Thank you for lifting me up when I was down and doing your best to help me.

I hold all of you close to my heart. Sincerely, thank you.

Filipe Fonseca

“Progress is the realization of Utopias.”

Oscar Wilde

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	1
1.3	Problem	2
1.4	Goals	2
1.5	Document Structure	3
2	Background and State of the art	4
2.1	Theoretical Background	4
2.1.1	Key Concepts	4
2.1.2	Evolution of Dependency Management Practices in Software Engineering	5
2.2	Methodologies and Frameworks for Dependency Management	5
2.2.1	Static Dependency Analysis	5
2.2.2	Dynamic Dependency Management	6
2.2.3	Security and Vulnerability Management	7
2.2.4	Maintainability and Predictive Models	8
2.3	Tools and Platforms for Dependency Visualization and Collaboration	9
2.3.1	Dependency Graphs	9
2.3.2	Dependency Visualization and Conflict Detection	9
2.3.3	Multi-Model Databases	10
2.4	Comparative Analysis	10
2.4.1	Efficiency: Tools and Methods That Improve Runtime Performance	11
2.4.2	Scalability: Approaches That Scale for Large Datasets and Complex Sys- tems	11
2.4.3	Security: Handling vulnerabilities and mitigating risks in dependencies. .	12
2.4.4	Collaboration: Supporting team dynamics and conflict resolution.	12
2.4.5	Maintainability: Metrics and tools for long-term system health.	13
2.4.6	Selecting Neo4j	13
2.5	Overview of Neo4j	14
2.5.1	Neo4j's advantages	15
2.5.2	Neo4j's drawbacks	16
2.5.3	Application in Dependency Management	17
3	Implementing Neo4j for Dependency Management	19
3.1	Project and Team Structure	19
3.2	Requirements Defined by Celfocus	21
3.3	Initial Adoption of Neo4j and Challenges faced	22
3.4	Coordination Among Teams	24

3.4.1	Concerns Raised	25
3.4.2	Discussion and resolution	27
3.4.3	Conclusions of Coordination and Decision to Implement Neo4j	28
3.5	Planning and Technical Implementation	29
4	Validation	38
4.1	Methodology and Participant Characterization	38
4.2	Results from VIPCE	39
4.2.1	Learning Curve and Onboarding Experience	41
4.2.2	Usage Frequency and Interaction Patterns	42
4.2.3	Perceived Utility and Added Value	43
4.2.4	Limitations and Suggestions for Improvement	45
4.2.5	Cultural and Communication Impacts	46
4.3	Feedback from Other Teams	47
4.3.1	Timing Constraints and Limited Exposure	47
4.3.2	Informal Feedback and Early Impressions	48
4.3.3	Influence of Team Attitude on Results	49
4.4	Observed Improvements	50
4.5	Remaining Challenges	51
4.5.1	Balancing Granularity and Maintainability	51
4.5.2	Lack of Centralized and Accessible Documentation	52
4.5.3	Ensuring Long-Term Engagement and Sustainability	53
4.6	Summary of Evaluation	54
5	Conclusions and Future Work	58
5.1	Conclusions	58
5.2	Future Work	59
	References	61

List of Figures

2.1	Example of Neo4j pattern	15
2.2	Example of Neo4j graph	16
3.1	Part of VIPCE's initial neo4j insance	23
3.2	Excel with high level dependencies	26
3.3	Neo4j instance for all teams	30
3.4	General instance of Neo4j	32
3.5	Teams represent different node types	33
3.6	Relationship between nodes	34
3.7	VIPOC's Neo4j instance	35
3.8	VIPOC's node types	35
3.9	VIPOC's instance different relations	36
4.1	Answers to the question "With what frequency do you utilize Neo4j?" in percentage	42
4.2	Answers to the question "What tasks do you utilize Neo4j for?" in percentage . .	43
4.3	Answers to the question "In your opinion, how was the impact of neo4j in depen- dency management?" in percentage	44

List of Tables

2.1	Strengths and Weaknesses of Different Tools	11
2.2	Strengths and Weaknesses of Different Softwares	12

Abbreviations

Abbreviation	Description
CD	Continuous Delivery
CI	Continuous Integration
COG	Collaborative Optimization Graph
DevOps	Development and Operations
ELT	Extract, Load, Transform
Gen AI	Generative Artificial Intelligence
IT	Information Technology
LLVM	Low Level Virtual Machine
MLoC	Millions of Lines of Code
NMI	Normalized Mutual Information
PAIS	Process-Aware Information Systems
PDGraph	Project Dependency Graph
TV	Television
WED	Workflow-Enabled Dependency

Chapter 1

Introduction

1.1 Context

In today's highly competitive market, the continuous evolution of complex software systems made it crucial for organizations to remain agile and responsive to customer demands [24].

Dependency management is essential for ensuring that different components in a complex system work seamlessly together [19]. If not handled properly, dependencies can create system conflicts, decelerate the development process, and generate instability in the production environment [21]. Poorly managed dependencies lead to delays and increase maintenance efforts. By studying and improving dependency management, productivity can be boosted, as the clear tracking of the dependencies reduces troubleshooting and avoids last-minute issues. A well-structured dependency management also impacts team dynamics and onboarding. Speeding up the integration of a new member enhances the overall agility and adaptability of a project.

Optimizing team structures in a Complex-Component based system by shifting from static component-based teams to dynamic and feature-oriented teams leverages cross-functional expertise more effectively. This would allow teams to focus on delivering features end-to-end, improving the speed/quality of the project [23].

Lastly, dependency management systems create stability and sustainable growth, as they minimize conflicts between different versions of components. Simplified maintenance plays a critical role in security, keeping all the systems updated to avoid vulnerabilities, and helps create a more agile and scalable Agile Information Technology (IT) infrastructure.

1.2 Motivation

Large-scale product development is becoming more and more difficult in modern software engineering, especially when several teams are working on parallel workstreams. As a provider of technology and solutions, Celfocus frequently works on the development and design of extremely complex systems that call for close coordination between teams, parts, and delivery channels. Of

these, the VTV project is a prime example of a complex project that captures the complexities of managing dependencies across environments, components, and domains.

The practical challenges of managing technical dependencies in such complex contexts are the driving force behind this dissertation. Misalignments in timing, scope, or interface expectations can lead to deployment failures, regression problems, or compromised functionality downstream because Celfocus teams frequently have autonomy over particular subsystems or features. In agile environments, where knowledge distribution and change velocity present additional threats to system stability and team coherence, these difficulties are exacerbated.

In particular, a persistent issue has been found to be the absence of centralized, visual, and current information regarding the relationships between components. Interdependencies are commonly tracked using team-specific Excel sheets and traditional documentation techniques, but these methods are usually too high-level, disjointed, or out-of-date to offer useful insights. Ineffective debugging, reactive team communication, and the frequent discovery of breaking changes only after they reach staging or production environments are the outcomes of this.

1.3 Problem

Celfocus's use of multiple technologies creates a web of dependencies between them. The current utilization of a high-level dependency in Microsoft Excel is insufficient for managing the dependencies in a detailed and scalable manner. The existing matrix lacks granularity and fails to capture the details of each dependency, creating blind spots in the understanding of how changes in one component can affect others. This can lead to system failures and subsequent delays in project delivery. Microsoft Excel is also prone to human error, making it difficult to scale to the growing complexity of the VTV ecosystem.

Today's agile teams are mostly static and component-based, limiting their ability to dynamically adapt to new project challenges and new features. This structure can slow down the development process and hinder collaboration between members. It is also more difficult to bring new members on board, as the current structure does not provide a detailed view of all the components involved. Rearranging the defined agile teams can increase efficiency, ensure that all tasks are assigned to the most appropriate member, and reduce the number of inconsistencies [13, 2].

1.4 Goals

Considering the context of the previous sections, this research aims to address critical challenges in managing dependencies across complex software systems and optimize team dynamics for a more effective delivery. The following research questions were written to guide the investigation:

RQ1: How can dependencies between different components in complex software systems be managed to simplify maintenance and increase stability?

RQ2: How can a dynamic and feature-oriented team structure improve the efficiency and quality

of software when compared to a traditional, static component-based team development?

RQ3: How can evolving software systems balance the constant need for updates with minimizing security risks, downtime, and cost increases in dependency management?

These research questions are a good place to start when trying to figure out the problems that come up when managing dependencies in big software systems and how to make development teams work together better. The main goal of this dissertation is to look into the main causes of dependency problems in complex environments with multiple teams and to come up with possible ways to lessen their effects.

More specifically, the work aims to investigate the impact of technical dependencies on software stability and maintenance, to understand how team dynamics and coordination practices influence these dependencies, and to identify methods that can improve visibility, traceability, and predictability in such systems. Another important goal is to see how different team structures affect the flow of development, especially when features cross over between systems or components.

The goal of the project was not only to find a workable solution to the problems that had been found, but also to test how well it worked in the real world. This includes checking to see if the suggested approach could help with problems related to dependency, make communication between teams easier, and help people make better decisions during the development and deployment cycles.

1.5 Document Structure

There are five main chapters in this document. The introduction is the first part, where the background, reasons for doing the research, and research questions are all explained. Chapter two goes over the relevant background and related work, giving an overview of the different methods that are used to manage dependencies. Chapter three is all about the implementation. It talks about the project's structure, the requirements that were found, and the solution that was made. The fourth chapter talks about the validation process and looks at the results of the questionnaires and interviews. Finally, the last chapter talks about the main ideas that came out of the work and gives ideas for how to make things better and do more research.

Chapter 2

Background and State of the art

In this chapter, we aim to give an up-to-date survey of the literature on dependency management methods, tools, and practices. By reviewing current research and innovations, this chapter seeks to identify how existing solutions tackle the problem of managing dependencies in complex component-based systems. These challenges comprise the reduction of bloated or redundant dependencies, ensuring of system maintainability by using predictive models, overhead management due to security risks of using vulnerable components, and allowing effective collaboration in the context of distributed development environments through the use of dependency visualization tools.

2.1 Theoretical Background

2.1.1 Key Concepts

Dependency management is the process of identifying, tracking, and maintaining the dependencies between different components in a software system. Good dependency management increases system stability, helps avoid potential conflict risks and provides a smooth update or integration process. When dependencies are poorly documented or maintained, it can also present security vulnerabilities, technical debt, or cause inefficiencies during development or deployment phases [23, 20].

A dependency graph is a directed graph with nodes representing the components and edges between the nodes representing the relationships or dependencies between them. Dependency graphs are used to visualize and analyze the structure of software systems, helping to identify direct and indirect conflicts, security risks, or underutilized dependencies. Graph databases allow for fast querying and analysis of related data, making them a critical part of modern software engineering [8, 4].

You may become familiar with this aspect if you work with Architectures based on components. Components that were short but not simple are now simpler, but the orthogonality is harder. To manage these dependencies effectively, a detailed understanding of the relationships between dependencies, both static and dynamic, is needed [25].

Dynamic environments are software systems running in contexts where dependencies change frequently, such as within DevOps pipelines, agile development workflows, or cloud-based architectures. Such environments require dependencies to be tracked and settled in real-time or they risk build failures, runtime errors, or deployment process inefficiencies [?].

2.1.2 Evolution of Dependency Management Practices in Software Engineering

In the early days of software development, dependency management was often manual and informal, with developers relying on simple include paths or library imports. The lack of automation led to challenges in scaling systems and maintaining consistency across environments [3].

With the growth of software complexity, tools like Maven and Gradle emerged, automating tasks such as dependency resolution and version control. These tools reduced manual effort but struggled with dynamic runtime dependencies and indirect conflicts, especially in large-scale systems.

Recent advances focus on integrating dependency management with workflows, monitoring, and visualization. Tools like Neo4J enable dependency graph analysis, supporting fine-grained conflict detection and real-time collaboration [1]. Security-focused approaches like PDGraph address vulnerabilities in dependency chains, while predictive models analyze maintainability trends, helping developers prioritize refactoring efforts [15].

Dynamic environments have driven the need for hybrid static-dynamic approaches. Workflow-based techniques, such as WED-flow, emphasize explicit representation and management of dependencies in process-aware systems, facilitating incremental system evolution [6]. These methods align with the needs of agile and DevOps practices, enabling rapid adaptation to changes while ensuring system consistency [23].

2.2 Methodologies and Frameworks for Dependency Management

2.2.1 Static Dependency Analysis

Compile-time dependencies detection, or static dependencies analysis at build time, help identify and resolve unused, large or inconsistently defined dependencies early in the software development process. It is imperative to keep software as stable as possible and improve maintainability.

This method is efficient in detecting build dependency errors like missing or redundant dependencies using static analysis tools like VirtualBuild. VirtualBuild does so by producing a "slice" of the original program containing only the important dependencies, rapidly decreasing the time taken for dependency resolution and the total build time (such as bringing down LLVM analysis from 4 hours to 38 mins) [17]. This hybrid strategy maintains soundness and completeness while minimizing overhead during dependency resolution.

Slimming is another innovative technique which helps in identifying bloated dependencies in Java projects using Static reflection analysis. Slimming identifies third-party libraries that are not being utilized, then analyzes the cost-benefit ratio of maintaining such libraries, resolving dynamic

behaviors in frameworks such as Spring Boot and Tomcat. It has shown high precision and recall in data-sets of a largescale, successfully eliminating many security risks as well as elimination of dependency conflicts.

Benefits of Static Analysis:

- **Accuracy** — High accuracy models provided by tools like Slimming and VirtualBuild help in correctly identifying and eliminating unused or redundant dependencies and represent towards reduced technical debt and enhanced system performance. *push*;
- **Scalability** — Static analysis techniques can be applied to codebases of a large scale, evidenced by projects such as Chromium and Maven-based repositories.
- **Security** — Spotting unused dependencies reduces risk from stale or vulnerable libraries.

The limitations of static analysis:

- **Runtime Blind Spot** — Static methods are usually blind to runtime such as dynamically loading dependencies or reflection based invocations. *push*;
- **Difficulties in Modern Architectures** — In systems with microservices or polyglot programming models, static analysis tools might have difficulties with cross-language dependencies or distributed builds.
- **Dependency Conflicts** — Static analysis can identify potential redundant dependencies, but may need more dynamic analysis to resolve these conflicts.

2.2.2 Dynamic Dependency Management

Dynamic dependency management adds more complexity to your build, by analyzing dependencies at runtime or in a workflow-oriented environment, in an attempt to overcome the limitations you just encountered. This allows for more in-depth insights into the behaviors of your systems, enabling the resolution of dependencies in real-time [18].

For example, WED-flow (Workflow, Event, and Data-driven flow) defines interdependencies within workflows by function WED-flow combines declarative process specifications with a database-centric execution model, enabling processes to be tracked and dependencies resolved dynamically in process-aware systems [7]. For example, the SISAUT use case has shown that with WED-flow it is sufficient to model the dependencies between material collection and families interview step once, rather than for each possible combination, because the dependencies between interprocesses are handled correctly throughout various WEDs that are implemented by different flows for individual families needing to be interviewed.

As such, WED-flow's data-centric design minimizes the management of dependencies across systems where such changes are common. This event-based triggers automatically adapt to changes further supporting agile workflows and progressive evolution of systems [16].

In addition to the above, dynamic monitoring may apply information-theoretic approaches such as NMI and clustering based on entropy. These approaches muffle similar features based on error diurnal and use the change in cluster/entropy to understand changes within a cluster. For instance, the rising entropy in a cluster can be a warning for an upcoming problem where preventive measures can be applied (e.g. fault diagnosis in J2EE-based enterprise systems [11]).

Algorithms such as RatioScore, SigScore, and BayesianScore enhance fault diagnosis by considering both contemporaneous components usage and historical component failure rates as statistical distributions. These approaches are significantly stronger than traditional heuristics, particularly when it comes to nonlinear dependencies or hidden interdependencies between components.

Advantages of Dynamic Management:

- **Real-time resolution** — Solves problems as they arise, minimizing downtime and preventing build failures. *push*;
- **Adaptability** — WED-flow and other event-driven strategies relate well to situations, like DevOps pipelines, that fluctuate often.
- **Extensive Error Discovery** — Information-theoretic methods can reveal complex or indirect dependencies that static analysis might overlook.

Limitations of Dynamic Management:

- **Overhead** — Runtime monitoring may incur performance costs, particularly in resource-constrained environments. *push*;
- **Complex Implementation** — Any loosely coupled system, particularly a workflow-oriented one, needs to be carefully designed because, as adaptable as it is, it also creates a complexity within the system.
- **Reliance on Accurate Models** — Dynamic techniques rely on the correctness of the initial dependency graph or clustering, which may need heavy manual supervision for validation.

2.2.3 Security and Vulnerability Management

Unpatched or outdated dependencies can now result in security risks for software systems. To mitigate these risks, strong mechanisms must be in place to discover and manage vulnerabilities in complex dependency trees.

PDGraph is one of the tools CNN leverages using dependency graphs to model and analyze the vulnerability propagation through a software ecosystem. PDGraph creates a project dependency graph, with nodes as projects and edges representing dependency relationships, with vulnerable versions of libraries. This allows to identify insecure edges (dependencies that introduce vulnerability) and give a quantitative risk evaluation of dependencies.

PDGraph analyzed more than 337,000 projects, detected 67,581 insecure edges, and showed that 1.87% of vulnerabilities were patched in an eight-month period, signifying slow uptake of the

fixes [15]. PDGraph underscores the importance of dependency graphs in working towards more resilient systems by uncovering circle dependencies and long dependency paths.

Depending on unpatched dependencies can lead to cascading impacts on system stability, causing:

- **Broader Attack Surface** — Vulnerabilities in popular libraries affect all systems that rely on them, increasing exploitation risk. *push*;
- **Integration Breakdown** — Project dependencies could potentially be insecure and they may not always be backward compatible with newer versions, which could break the development workflow.
- **Compliance Risks** — Projects may inadvertently risk violating laws or industry standards (GDPR, ISO certifications) if vulnerabilities remain unaddressed.

By integrating tools like PDGraph, organizations can proactively address security risks, prioritize patching efforts, and maintain a higher level of system integrity.

2.2.4 Maintainability and Predictive Models

Especially in large-scale, component-based architectures, software maintainability is a key aspect of ensuring the survival of systems over time. Most of the time, they serve as actionable metrics in optimizing code quality and minimizing tech-debt.

For example, Karanikiotis et al propose trend-based approaches that rely on the static analysis metrics to capture weekly changes in the software components [13]. Metrics such as Cyclomatic Complexity, Lack of Cohesion of Methods (LCOM5) and Coupling Between Objects (CBO) are tracked over time, in order to detect components that may become non-maintainable.

These methods leverage machine learning techniques, including weakly supervised learning (WSL), successfully detect faulty components with high accuracy. For instance, models using WSL have achieved an F1 score of 0.89, leading to a rapid detection of maintainability problems addressing issues at a very early stage.

Addressing technical debt is a mix of exact metrics and automation:

- **Code Complexity and Cohesion** — Cyclomatic Complexity, LCOM5 and similar measures help developers to recognize and refactor overly complex or tightly coupled codebases. *push*;
- **Dependency Utilization** — Slimming and Its People are two tools that leverage attachment analysis to detect unused components within dependencies, allowing developers to remove unused components and simplify the build process.
- **Analysis of Trends** — By keeping track of changes in maintainability metrics, teams can proactively prioritize refactoring efforts and tackle emerging problems.

Integrating these approaches in agile and DevOps flow will help maintain software quality, lower maintenance costs, and scale systems.

2.3 Tools and Platforms for Dependency Visualization and Collaboration

Modern software systems have many interdependencies across components, modules, and teams, which is one of the reasons they are inherently very complicated. Visualization and collaboration tools are also key to managing this complexity helping maintain system stability, improving developer coordination, and ultimately enhancing overall efficiency.

2.3.1 Dependency Graphs

Graph databases (such as Neo4J and ArangoDB) have transformed the dependency management problem by providing efficient methods for storing and querying interconnections [5, 9]. Typically, graph databases allow one to identify dependencies as nodes (components; see the example in gray) or edges (relationship or connection; see the example in black) and search them, rather than looking across arbitrary key/values to find those that are connected.

Neo4J empowers graph databases where its Cypher query language enables the ability to query complex dependencies in real-time [14], allowing for the direct and nuanced mode of analyzing relationships in very large systems, as evidenced in this case.

A multi-model database like ArangoDB goes further by combining the document and key-value store models, allowing developers to handle varying types of datasets from a unified platform.

Due to their importance in large-scale, component-based systems, such databases provide support for building a dependency graph for use cases such as impact analysis, conflict resolution, and fault detection.

The feature becomes especially useful in collaborative contexts, such as GitHub, where one team or developer's changes may unintentionally affect the work of others. For instance, tools such as COG (Collaboration Over GitHub) evaluate the interdependencies of events to update the user about direct and indirect conflicts in real-time integrated with graph databases. Direct conflicts happen when two or more developers are working on the same file or module at the same time. Indirect conflicts occur for changes made to one component affecting functioning of the dependent components, whose impact may not be caught without analyzing the dependency graph. They are doing this by using dependency graphs that can assist with collaboration, mitigate integration issues, and provide better visibility of system-wide impacts at the team level.

2.3.2 Dependency Visualization and Conflict Detection

Intuitive, easy-to-use visualization tools like those based on Neo4J make it very easy to see dependency relationships [10]. Through interactive dashboards, developers can also investigate direct

(e.g., out-of-band edits) and indirect (e.g., changes to a given library that may potentially affect deeply nested downstream components) conflicts. With the recently released Neo4J interface, users can dynamically visualize nodes (components) and edges (dependencies) to identify critical paths, redundant dependencies, or vulnerable components. One can easily query the affected components so that one can check the impact thoroughly before deployment

COG uses Neo4J integrated with IDEs such as Eclipse to give developers real-time conflict detection and dependency insights. Utilization of features such as dependency alerts, version tracking, and conflict notifications promotes a proactive approach to dependency management, leading to minimized downtime and enhanced development efficiency.

These tools mitigate experiences that might be less favorable in distributed teams, ensuring that everyone touched by a change understands its effects on the wider system.

2.3.3 Multi-Model Databases

Graph-specific databases such as Neo4J are designed for use cases that heavily rely on relationships, while multi-model databases such as ArangoDB inherit the flexibility of support for multiple data models (graph, document, key-value stores) on a single platform.

Neo4J specializes in graph storage and querying, high-performance relationship analysis, strong community support, but lacks support for non-graph data types and absence of sharding in the free version [14].

ArangoDB has flexibility to deal with various types of datasets, robust support for sharding, and excellent scaling for performance. However, ArangoDB is strictly a graph database, with no global indexes or search (like Elasticsearch) and requires advanced expertise for complex implementations.

Ultimately, deciding between Neo4J and ArangoDB is a matter of the considerations of each individual project:

- **Graph-Centric Projects** — Neo4J also works best for applications that are all about dependency analysis, recommendation systems and fraud detection where you are often executing relationship queries. *push*;
- **Multimodel Requirements** — ArangoDB is more equipped for applications that need to incorporate multiple data types, including hybrid data models combining graph and document models.

2.4 Comparative Analysis

This literature review outlined an evaluative comparative of existing tools and methodologies for accountability and responsibility in dependency management. The efficiency and scalability of the system are key aspects of managing dependencies in a modern large-scale system, so we do an evaluation in this section.

2.4.1 Efficiency: Tools and Methods That Improve Runtime Performance

Static tools such as VirtualBuild and Slimming improve runtime performance by optimizing dependency resolution and detecting unused or redundant dependencies:

To detect build dependency errors quickly, we introduce VirtualBuild, a tool that efficiently generates smaller versions of programs and accelerates opening build failures. Example: it decreased LLVM analysis tooling time from 4 hours to 38 minutes, a 6× performance improvement over state of the art tools [26].

Slimming helps identify bloated dependencies in Java projects allowing you to remove them to improve runtime performance without sacrificing functionality. Its reflection analysis results in a 97% precision rate [25], identifying vulnerabilities on a much lower level.

Dynamic methods focus on resolution and detection of runtime faults like NMIBased clustering and entropy-based clustering. Dependency-related abnormal behavior is detected in real time using these techniques, which minimizes downtime and makes it easier to recover from faults. For instance, entropy-based approaches detected faults in J2EE-based enterprise systems with less than 5% false-positive rate [25].

Approach	Strengths	Weaknesses
VirtualBuild	Optimizes build times effectively.	Limited to static dependencies, lacks runtime insights.
Slimming	Reduces bloat and vulnerability risks.	Requires manual validation to ensure correctness.
NMI/Entropy	Handles runtime anomalies efficiently.	Performance overhead in resource-constrained systems.

Table 2.1: Strengths and Weaknesses of Different Tools

2.4.2 Scalability: Approaches That Scale for Large Datasets and Complex Systems

Graph databases like Neo4J and ArangoDB excel in scaling dependency management to large, complex systems. Neo4J is optimized for deep relationship queries, as it handles large datasets effectively by leveraging its Cypher query language. It supports enterprise-scale applications but lacks built-in sharding in its community version. As a multi-model database, ArangoDB offers superior scalability with robust sharding capabilities, making it ideal for hybrid systems combining graph, document, and key-value models. However, its complexity may pose a challenge for novice user [5].

Workflow-driven methods like WED-flow achieve scalability by formalizing dependencies as database-centric workflows. WED-flow processes dependencies incrementally, enabling scalability in dynamic environments like medical systems and large-scale PAIS (e.g., SISAUT). Tools like

PDGraph demonstrate scalability in dependency risk analysis across vast ecosystems. PDGraph analyzed over 337,000 projects, identifying 67,581 insecure edges while scaling to dependency chains of up to 102 steps, a significant achievement in analyzing complex, interconnected systems [5].

Approach	Strengths	Weaknesses
Neo4J	High-performance graph querying for large systems.	Limited scalability in free version due to no sharding.
ArangoDB	Scalable across diverse data models, robust sharding.	Complexity for novice users in setup and maintenance.
WED-flow	Scales incrementally in dynamic environments.	Best suited for domain-specific applications.
PDGraph	Handles large-scale vulnerability detection.	Focused on security risks, less adaptable to other contexts.

Table 2.2: Strengths and Weaknesses of Different Softwares

2.4.3 Security: Handling vulnerabilities and mitigating risks in dependencies.

It is a major challenge for modern software systems to manage security risks in dependency chains, as library vulnerabilities can pervade entire ecosystems.

Tools such as PDGraph are made for detecting and quantifying dependency risks. PDGraph builds a dependant graph, and marks insecure edges — vulnerable dependencies. This enables developers to focus on getting remediation in place, fixing serious security bugs before they spread further. In a 337,000 project large-scale study, PDGraph identified 67,165 insecure edges [15], demonstrating both its scalability and accuracy in practice.

Tools for dependency management also makes a strong case for updating proactively. Slimming is about removing unused dependencies, reducing the system’s exposure to vulnerabilities present in third party libraries. PDGraph noted a low adoption of patches (1.87% over eight months [15]), highlighting the need for automated systems for patch management.

2.4.4 Collaboration: Supporting team dynamics and conflict resolution.

Collaboration is key, especially in distributed development environments, and better management tools must make it easier to collaborate by reducing conflicts and providing context [1].

Tools like COG (Collaboration Over GitHub) work with platforms like GitHub to offer worries in real time:

- **True Conflicts** — These occur with multiple developers updating the same file or module. *push*;
- **Indirect conflicts** — Familiar to both Developers and the automated tool, through dependency graph analysis, where a change in one component has a domino effect on another related component.

Neo4J based systems allow the developers to have a visual representation of complex dependency relationships which makes it much simple to find conflict paths, and critical dependencies. Using interactive dashboards allows you to resolve conflicts in a proactive manner by having better visibility on both direct and indirect dependencies.

2.4.5 Maintainability: Metrics and tools for long-term system health.

It can take effective tools and metrics to gauge and optimize system health to maintain software quality over the long term.

Static analysis metrics like Cyclomatic Complexity, Coupling Between Objects (CBO) and Lack of Cohesion of Methods (LCOM5) can give you quantitative measures on maintainability. Trend-based approaches monitor these metrics over time to identify and minimize risk. Trend-based prediction models have shown to have high accuracy for detecting such components that are susceptible to technical debt or maintainability problems, with an F1 score of 0.89 [13].

Tools such as Slimming and VirtualBuild can also complement maintainability metrics by:

- Discover and remove unnecessary dependencies and accurate technical debt. *push*;
- Streamlining build processes, eliminating unnecessary tasks, and making systems more scalable.

2.4.6 Selecting Neo4j

We looked at the features, strengths, and weaknesses of several graph-based tools and frameworks, such as Neo4j, ArangoDB, WED-flow, and PDGraph. In the end, Neo4j was the best choice for meeting the specific needs of dependency management in this dissertation.

The fact that Neo4j is mature and widely used in both business and academia is one of the main reasons it was chosen [8]. WED-flow and PDGraph are mostly academic frameworks that focus on certain modeling paradigms or have a limited scope. Neo4j, on the other hand, is a strong and general-purpose graph database that works well at scale. Its extensive documentation, active community support, and long-term maintenance make it even more useful for real-world projects that need to be production-ready.

Neo4j's built-in graph storage engine and declarative query language, Cypher, are also very useful. These features make it easy and natural to model software dependencies, which makes it possible to write complex traversals and queries in a way that is easy to read and understand [8]. ArangoDB, on the other hand, supports multiple models but doesn't have the same level of optimization and community resources for operations that are only based on graphs [5]. Also, when it comes to graph-specific tasks, ArangoDB's query language, AQL, is harder to understand than Cypher.

Neo4j's ability to fit into development workflows easily is another important factor. It comes with built-in visualization tools and works well with external interfaces, making it a great choice for places where updates, validations, and shared access are needed often. The graphical representation of nodes and relationships was especially helpful for team communication, which was not as important in the other tools we looked at [14].

To sum up, Neo4j was chosen because it works well, is easy to use, has a lot of community support, and can model things in a way that is very clear. All of these things fit well with the practical goals and technical limits of this dissertation. Adopting it makes it easier to not only technically implement a dependency graph, but also to keep it up to date and useful for multiple teams working on complicated software systems [4].

2.5 Overview of Neo4j

Neo4j is a very efficient and popular graph database management system that stores data as nodes and relationships that are connected to each other, instead of tables or documents. In Neo4j, nodes are the basic data units that stand for things like users, software components, or other unique things. Key-value pairs that store detailed attributes can be stored in each node. Relationships describe how nodes are connected, such as dependencies, interactions, or associations. They can also have properties that give more context.

Cypher is a special query language that is a key part of Neo4j [14]. Cypher is a declarative and expressive query language made specifically for working with and navigating graph-structured data. It lets users quickly make, get, change, and delete nodes and relationships, which makes complicated queries much easier. As seen in 2.1, Cypher queries look a lot like patterns drawn on a whiteboard because they use simple syntax, which makes them much easier to read and use.

As showcased in 2.2, Neo4j is also great at giving users the ability to visualize data, which lets them explore complex dependency structures in an interactive way. The built-in visualization tools make data easy to read and show how nodes are related to each other, which helps you understand system dependencies better [1]. This graphical representation is especially useful for managing dependencies, where understanding how different parts of a system interact is important for finding problems and making processes run more smoothly.

Neo4j also has a flexible schema, which makes it easy to change to meet new needs without having to completely change the structure. This architecture, which doesn't require a schema or

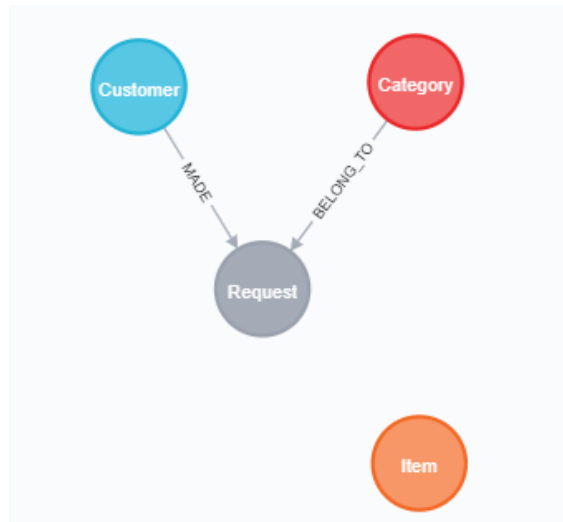


Figure 2.1: Example of Neo4j pattern

allows for one, makes it easy to make changes and develop iteratively. It's great for situations where data relationships are often updated or added to.

Neo4j's powerful indexing system is another important feature [1]. It is made to handle complex queries quickly, even in large datasets with many connections. Neo4j automatically keeps track of indices so that data can be retrieved quickly. This greatly speeds up operations that involve resolving dependencies. Neo4j also follows strict ACID (Atomicity, Consistency, Isolation, Durability) transaction rules [8], which makes sure that data is very reliable and accurate.

Neo4j is a great tool for managing complex dependencies, especially in software development environments where relationships are always changing. This is because it has so many advanced features.

2.5.1 Neo4j's advantages

There are a number of good reasons why Neo4j is considered one of the best graph database solutions. First, the tool handles complicated queries that involve linked datasets very well. Queries that require long relationship traversals often slow down traditional relational databases. Neo4j, on the other hand, is specifically designed to make it easy to quickly navigate relationships, which greatly speeds up query response times and improves the overall performance of the system [5].

The technology's better visualization features are another big plus. These features make it much easier for users to understand and make sense of complex connections between data entities. When teams can see data relationships clearly and easily, they can find important dependencies, fix problems faster, and make smart decisions quickly.

Neo4j also lets you change your schema, which is very useful for projects that are always changing. The schema-less architecture makes it easy to make quick changes and agile iterations. This speeds up development cycles and makes it easy to add new data structures. Traditional

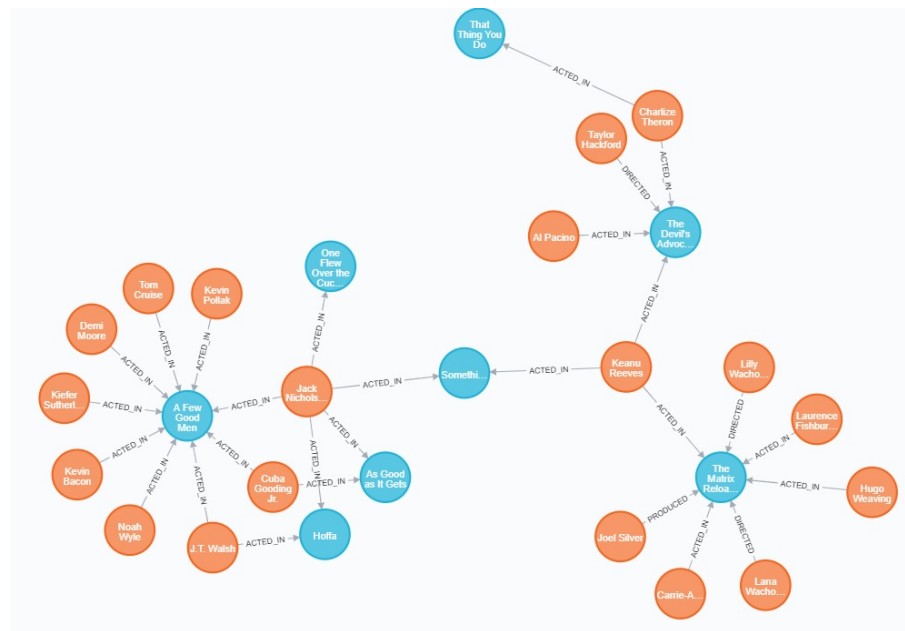


Figure 2.2: Example of Neo4j graph

databases, on the other hand, require strict schema definitions [13]. This flexibility is especially useful in places where the data changes or is updated often.

The graph database also makes sure that data is correct by following the strict rules of ACID transactions. This means that data stays correct, consistent, and reliable even in environments with a lot of transactions and complicated data [18]. It also has strong indexing systems that speed up queries across large, connected datasets.

Neo4j is a powerful tool for managing complex dependencies in modern software systems because it has fast query processing, advanced visualization, schema flexibility, data integrity assurance, and high-performance indexing.

2.5.2 Neo4j's drawbacks

Neo4j has some well-known good points, but it also has some big problems that users should think about carefully. One of the biggest problems is that it's hard to get started with. When teams that are used to working with traditional relational databases switch to Neo4j, they often have a lot of trouble [4]. It takes a lot of training for teams to switch from table-based structures to graph-based data models because they have to learn new concepts like nodes, relationships, and properties. It can also take a lot of time and work to learn the Cypher query language, which can make it harder for the team to get things done during the transition period.

Another big problem with Neo4j is that it doesn't work well with very big datasets. Neo4j works best with data that is connected, but when datasets get bigger than the system memory, performance can drop a lot. This degradation can make queries take longer to respond, cause more latency, or even cause the system to crash. Some of these problems can be solved with

indexing and caching systems, but to manage very large datasets well, you usually need advanced hardware and careful plans for optimizing your database.

Neo4j's versioning system also puts limits on things that could make it harder to grow a project and keep costs down. The Community Edition is free to use, but it doesn't have some important features that are only available in the Enterprise Edition, like better management of concurrency and locking, more advanced scalability options, online backup options, and more. Companies will always have to pay more to get and keep licenses for the Enterprise Edition for important or enterprise-grade software. This can add a lot to the cost of a project [8].

Another important question is how flexible Neo4j's schema is. The schema-less architecture offers considerable agility advantages, especially in dynamic settings characterized by frequent schema modifications; however, it also presents potential risks of data inconsistency. If organizations don't have strict rules and procedures for data governance, they could run into problems like duplicate nodes, inconsistent relationship definitions, and conflicting data attributes [14]. Over time, these inconsistencies can add up, making it harder to keep the data up to date and making the logic for queries more difficult. This can make database administrators and developers work less efficiently and cost them more money.

Adding Neo4j to an existing technology ecosystem can be hard because it is hard to integrate. Many companies already use a mix of technologies, such as NoSQL systems, traditional relational databases, and other software tools. You often have to spend a lot of time and energy on middleware development, configuration, and management to make Neo4j work well with these other systems [14]. This integration is hard and costs a lot of money up front. It also needs ongoing resources to keep data in sync and work with different technologies in the company.

2.5.3 Application in Dependency Management

Neo4j has great tools for managing dependencies, which is especially useful for dealing with the complexity of modern software systems that often have a lot of parts that work together. Dependency management is the process of keeping track of, controlling, and analyzing the relationships between software modules, libraries, services, or individual functions in order to improve the system's stability, maintainability, and overall quality [9].

One big plus for Neo4j in dependency management is that it can naturally and clearly model how software parts are related to each other. To show dependency relationships, traditional databases need a lot of tables and complicated join operations. This makes queries slow, prone to errors, and hard to keep up with [3]. On the other hand, Neo4j shows these dependencies as graph structures, with each software component as a node and each dependency as a relationship. For example, in a big business software environment, each microservice or library can be thought of as a node with information like version number, author, or last change date. Dependencies between these nodes are set up directly, which makes the structure easier to understand and speeds up queries a lot.

Neo4j also makes it much easier to do impact analysis. When you do impact analysis, you try to figure out how changes to software components might affect other parts of the system. With

Neo4j, impact analysis queries can quickly and easily go through many layers of dependencies [23]. For instance, if a certain API or microservice needs to be updated, developers can run a simple Cypher query to get a list of all the software modules and systems that are directly or indirectly affected by the change. This is what such a query might look like:

```
1 MATCH (component:Microservice {name:'AuthService'})<-[:DEPENDS_ON*]-(dependent)
2 RETURN dependent
```

This would quickly return all components that depend on AuthService, making the evaluation process much easier than traditional methods, which might involve keeping track of dependencies by hand in spreadsheets or relational tables.

Neo4j's ability to help with software dependencies is another important benefit. It can be hard to understand and make mistakes when using text or tables to show dependencies in modern systems. Neo4j has built-in visualization tools that let users graphically explore and analyze software dependencies. This makes it easy to find complex connections, groups of components that depend on each other a lot, or possible points of failure. A good example would be being able to see circular dependencies between modules, which often cause problems with integration and maintenance [12]. By seeing these patterns, developers can proactively restructure or decouple parts of the software that are causing problems to make the architecture better.

Neo4j also makes it much easier to keep and improve software. Neo4j makes it easy to figure out which software parts need to be fixed when you're working on updates, upgrades, or refactoring. The dependency graph makes it clear which modules are affected when a software library is no longer supported or is updated. This makes things less confusing and risky when it comes to maintaining software [22]. Think about what would happen if a popular logging library got an upgrade. A simple query in Neo4j could quickly show all the software parts that are affected, which would let developers make changes in a planned and certain way.

Last but not least, Neo4j's flexible schema approach makes it easier to manage dependencies. Agile software development methods depend on making small, repeated changes to software, which often means updating the dependency structure. Neo4j is schema-less or schema-flexible, so it can easily handle these changes. Teams don't have to change the way they store data or the way they organize relational databases every time something changes. For example, when developers add a new third-party API to an existing ecosystem, they can easily add new nodes and relationships without having to change the whole database. This flexibility cuts down on overhead a lot and speeds up the processes of continuous integration and continuous delivery (CI/CD) [6].

In short, Neo4j makes dependency management a lot better by clearly and explicitly modeling dependencies, making it easier to do impact analyses, providing intuitive visualization, supporting accurate maintenance strategies, and giving developers the freedom they need to work in agile environments. These combined features make software systems much more stable, easier to maintain, and more flexible.

Chapter 3

Implementing Neo4j for Dependency Management

As software systems get more complicated, they need more advanced management methods to make sure that development processes are strong, efficient, and easy to maintain. Managing dependencies is an important part of software engineering, but it gets a lot harder as systems become more modular and connected. Graph databases like Neo4j have become powerful solutions for these problems. They show complicated interdependencies in clear and easy-to-understand ways.

This chapter talks about the journey of implementing Neo4j at Celfocus and how it helped them manage dependencies between different software development teams. At first, the chapter gives a thorough introduction to Neo4j, focusing on its main features, benefits, drawbacks, and how it can be useful for managing complex dependencies. It then shows how Neo4j was first used by one team at Celfocus, talking about the results, useful information gained, and any problems that came up.

The next part of the chapter goes into more detail about how different teams in the company work together, including the main issues that were brought up, the work that was done together, and the final agreement. The process of deciding to officially use Neo4j at Celfocus is fully explained, along with the key criteria that led to this strategic choice.

The next parts give a more detailed look at the implementation stage, with in-depth technical methods and code examples and pictures that go along with them. Lastly, the chapter looks at the results of Neo4j's implementation, gives advice on how to keep it up to date and maintain it, and makes suggestions for how to keep improving the company and what new things it could add in the future.

3.1 Project and Team Structure

The use of Neo4j for dependency management was created in the VTV ecosystem, which is a complicated and modular software environment that provides a lot of services to users. There are several development teams in this ecosystem, and each one is in charge of a different part of the

platform's functionality. It was important to know how the organization worked and what each team was responsible for in order to create a shared Neo4j model that accurately and usefully showed how the system's parts depended on each other.

When the VTV program was put into action, three main teams were involved: VIPCE, VIPOC, and Team 4. Each of these teams has a lot of freedom and works on different parts of the VTV platform, from the data infrastructure to the backend integration and operational tools. Even though their areas of focus are different, the teams all face the same problem: dependencies between their parts often cross boundaries, which means that the effects on one team are not always clear right away. The main reason for using a graph-based model was this ability to connect with other models.

The VIPCE team is in charge of making the VIP solution better over time. This solution is the data lake part of the VTV Cloud Video system. VTV's functional teams send them Technical Analysis Documents (TADs) that tell them what to do, and their work is in line with release cycles that happen every so often. Because of this, the VIPCE team often has to deal with getting data into systems, changing it, and sending it out across many analytics and operational pipelines. Because their parts are very data-driven and work with systems run by other teams, it was more important than ever to make sure that everyone understood how data flowed and what depended on what.

The VIPOC team works on a part of the platform that is more service-oriented. Their main job is to make tools that let VTV operations teams manage the TV set-top-box infrastructure from a distance. This includes building and keeping up a web-based operations portal, as well as the backend services that make it possible to take actions from afar in real time. Because of the kind of work they do, the VIPOC team works with a lot of backend services, external APIs, and device firmware integrations. The fact that these dependencies were so complicated, especially how they changed behavior during remote operations, made it hard to keep the deployment stable, especially when teams didn't have full visibility when interdependent functions were changed.

Finally, Team 4 is in charge of building and keeping up the backend services that make the whole VTV infrastructure work. Their job is to make sure that different backend components and third-party services can talk to each other without any problems. Team 4 is in charge of coordinating service calls, data exchange, and performance monitoring. They are the backbone that supports the higher-level functions that other teams implement. Because their parts are so tightly connected, changes to their systems often affect many layers of the VTV platform. This makes it even more clear why they need a shared dependency management strategy.

As a group, these three teams cover a wide range of VTV's technical stack, from data lakes and service orchestrations to tools for frontend operations. Before Neo4j came along, these teams often had to react to problems that arose after deployment, using manual documentation, verbal communication, or issue detection. It was clear that a common, visual, and easy-to-maintain system was needed to keep track of and document dependencies across the ecosystem after carefully looking at their roles and how they were connected.

This understanding of structure was the basis for the Neo4j implementation plan, which guided both the graph schema modeling and the choices about how to integrate it into the organization.

The next subchapter will go into more detail about the requirements and expectations that Celfocus set as part of this project.

3.2 Requirements Defined by Celfocus

Celfocus saw real operational problems with the VTV program that led them to look into a graph-based solution for managing dependencies between teams. Celfocus is a company that makes and delivers complex software systems for telecommunications. They came up with a list of important needs that the solution needed to meet. These requirements helped shape the modeling process, the implementation effort, and the technical and organizational aspects of the work that was done.

One of the main needs was for teams and components to be able to see software dependencies more clearly. Before Neo4j came along, people knew about these dependencies in different ways, such as through spreadsheets, verbal agreements, and undocumented assumptions. This lack of centralized and regularly updated documentation often caused deployments to fail and services to go backwards, especially when changes to one team's component unknowingly affected components owned by another team. Because of this, it was important to find a solution that could give everyone a single, up-to-date picture of how the components worked together without adding too much work for the teams involved.

Celfocus also stressed how important it is for things to be easy to maintain and update. It wasn't enough to just write down dependencies at one point in time; the system had to be able to change over time. Because the teams used agile methods, which involve frequent deployments, changing requirements, and short development cycles, the solution had to be lightweight and easy to add to existing workflows. Developers needed to be able to quickly and easily add, change, or remove nodes and relationships without too much trouble. Because of this, a Git-based workflow was suggested and used to manage changes to the Neo4j schema through versioned scripts, with the added protection of peer-review systems.

Another important requirement was that modeling had to be flexible. This was especially important because each team managed a different set of components. Some teams, like VIPCE, worked mostly with abstract data flows and pipelines. Others, like VIPOC and Team 4, worked with real services, APIs, and deployment artifacts. A strict, one-size-fits-all modeling method would not work. Instead, the chosen solution had to work with different levels of detail, allowing high-level component overviews for everyone to understand, while also letting individual teams model things more deeply when they needed to. This duality was very important and would later show up in the creation of both a shared global Neo4j instance and instances that were specific to each team.

Celfocus also needed the solution to help with onboarding and communication between teams. Because team members changed often and came from a wide range of technical backgrounds, the dependency graph had to be more than just a technical tool; it also had to help people talk to each other. The graph should make it easier for new developers to understand how the platform is put together and how it works with other things. The tool should also help with impact assessments

when planning new features or finding the source of bugs. This will make sprint planning more effective and cut down on the back-and-forth between teams.

Finally, a more strategic need came up for the solution's long-term viability and integration. The graph had to be something that wouldn't get worse over time, like other documentation efforts had. This required both technical reliability, which was achieved by making clear update procedures, and organizational commitment, which was achieved by giving specific roles within the teams clear responsibilities for maintaining the graph. It was hoped that if the solution worked, it could be expanded or changed to work in other parts of the company besides the VTV project.

In short, Celfocus's requirements were not just technical; they were also closely related to how the organization worked and how people worked together. These priorities set the stage for the planning, building, and ongoing improvement of the Neo4j system as a main tool for managing dependencies. The next few subchapters explain how these needs were met during the implementation process.

3.3 Initial Adoption of Neo4j and Challenges faced

Three development teams (VIPOC, VIPCE, and Team 4) were concurrently working on different system components when Neo4j was first adopted at Celfocus as part of the VTV project. Although each of these teams was in charge of distinct technological domains and platform functionalities, a complex web of dependencies made their work inherently interconnected.

The idea behind launching Neo4j came from within VIPCE. Throughout the development and deployment stages of their project, this team encountered persistent operational challenges. Several times, VIPCE-developed code caused unexpected problems with other teams' functionalities. Implicit or undocumented dependencies were the root cause of these problems; these interdependencies were crucial to the stability and behavior of the system even though they were not readily apparent.

In a typical scenario, VIPCE would modify an existing module or implement a new feature, only to find out after deployment that other services that depended on that module and were maintained by Team 4 or VIPOC stopped working as intended. It was very challenging to predict the wider effects of even small changes because it was impossible to see how various parts were connected. Rollouts were delayed, production environments experienced higher error rates, and collaborative development was generally inefficient as a result of this uncertainty.

The team recognized the need for a solution that could methodically map all dependencies between VTV system components in response to these persistent issues. The objective was to reduce the risk of breaking interconnected functionalities by managing and visualizing the dependency landscape in a way that would enable developers to anticipate the possible repercussions of their actions. Neo4j was chosen following internal deliberation and option exploration because of its clear and accurate modeling, representation, and querying of relationships between system entities.

By the end of the year, when I had already joined Celfocus, the first Neo4j implementation got underway. Even though I hadn't yet thoroughly examined graph-based platforms or contrasted Neo4j with competing technologies, I was still able to keep a close eye on the decision-making process and watch the implementation steps as they happened. The internal adoption was spearheaded by VIPCE, who started using Neo4j to model their component landscape and progressively created a graph that represented the interdependencies that were previously limited to individual knowledge or documentation, as shown in 3.1.

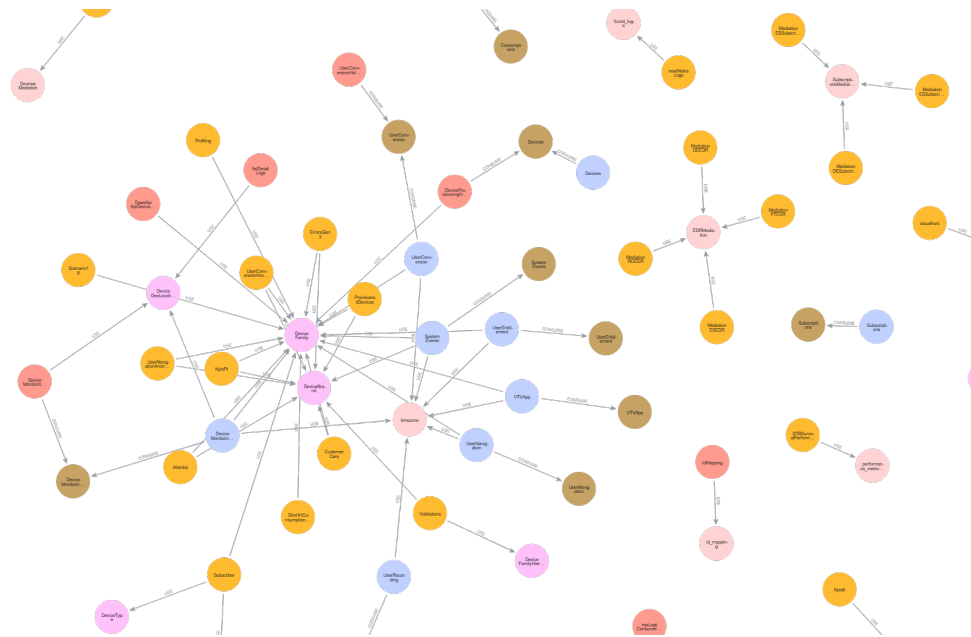


Figure 3.1: Part of VIPCE's initial neo4j instance

Neo4j's early application proved successful right away. Improved coordination with the other teams was made possible by the dependency visualization, particularly when it came to determining which system components could be changed safely and which needed prior coordination and communication. Neo4j began to develop into a technical tool as well as a common language for comprehending and handling the intricacies of cooperative software development within the VTV project as the team improved its use of it.

In addition to being innovative, Celfocus's early Neo4j implementation was filled with difficulties as the team made the switch to graph-based dependency management. Although the tool's adoption had obvious benefits in terms of system visibility and operational effectiveness, the integration process also uncovered a number of organizational and practical challenges.

The effort needed to map the entire scope of their dependencies was one of the VIPCE team's most pressing problems. Before Neo4j, a large portion of the dependency knowledge was either dispersed throughout internal tools and documents or implicit, held informally by developers. It took a lot of time and manual labor to convert this data into a structured graph. The process of identifying, documenting, and modeling each software component separately, along with its associated dependencies, was laborious and cognitively taxing. Considering that there was no

previous automation or infrastructure to support it, this first mapping phase required a significant investment.

The team had to deal with a challenging learning curve in addition to the technical workload. The majority of the team members had no prior knowledge of graph databases or the Cypher query language because Neo4j was a new technology in the company. Developers had to become knowledgeable about the practical aspects of querying, structuring, and maintaining the database in addition to the conceptual distinctions between relational and graph-based data modeling. As the team progressively became proficient with the platform, this learning process necessitated active experimentation, peer-to-peer knowledge sharing, and iterative adjustments.

Furthermore, because of the narrow adoption scope, systemic issues remained even after VIPCE was successful in mapping many of their internal dependencies. Neo4j had not yet been incorporated into the workflows of Team 4 and VIPOC, the other two teams working on the VTV project. Because of this, some dependencies outside of VIPCE's purview were either not mapped or had inadequate documentation. A residual class of dependency-related problems resulted from this partial visibility, and while they were less common than previously, they still showed up. These incidents demonstrated the drawbacks of a segmented adoption strategy, as wider participation was necessary to attain full system awareness.

Granularity was the subject of another important observation during Neo4j's early use. Although the graph representation was successful in capturing high-level dependencies between components and services, it occasionally lacked the resolution required to reveal more specific issues. For instance, dependencies may exist within particular endpoints, data flows, or internal modules in addition to at the level of entire microservices. If this level of information was left out, the graph might not show subtle interactions, which could lead to risk zones going unnoticed or deployment failures going unaccounted for. This insight led to additional discussion regarding the proper level of modeling depth needed to make the tool genuinely effective at preventing conflicts.

In conclusion, although Neo4j's adoption resulted in noticeable structural improvements, the process of putting it into practice revealed significant difficulties with data modeling effort, internal skill acquisition, limited cross-team adoption, and the requirement for exact granularity. In order to scale the solution across teams and guarantee that the tool could deliver consistent, system-wide value, it became imperative to address these problems.

3.4 Coordination Among Teams

Celfocus initiated a structured engagement strategy to make sure that all teams working on the VTV project could use Neo4j in a consistent and scalable way. This technique was made to find the bigger problems in the company that have to do with managing dependencies and working together, as well as to understand the technical details of each team. The goal was to make sure that Neo4j's deployment was tailored to the specific needs and operational realities of each team, rather than utilizing a one-size-fits-all approach.

The first step in this process was a month-long immersion time with each of the three main teams: VIPOC, VIPCE, and Team 4. Because each of these time periods was a development sprint, it was easy to see how they coordinated, deployed, and worked together every day. During these sprints, I worked closely with my team members to learn more about the architecture of their systems, the kinds of dependency problems they had before, and the problems that were still not fixed or continually coming up. These observations provide significant insights into the monitoring or oversight of inter-team interdependence and the persistent discord resulting from misaligned delivery or unforeseen code modifications.

This part of the process not only helped find technical flaws, but it also brought up more general methodological challenges. Some teams had trouble adopting agile principles, especially when it came to managing requirements that crossed functions and working with other teams. Poor communication or sprint goals that weren't aligned often led to late detection of integration problems. Because of this, these characteristics were especially important when it came to managing dependencies.

After the immersion phase, a meeting was set up with people from all three teams. At this meeting, it was shown to everyone what VIPCE had already done and how they had used Neo4j to keep track of their internal dependencies. The two goals of this presentation were to talk about the real benefits and lessons learned from VIPCE's work and to advise that all teams working on the VTV project use Neo4j.

The conversation that followed helped to figure out how open each team was to the concept of using a single platform for managing dependencies. Even though everyone agreed that more collaboration and visibility were needed, each team had its own set of problems, goals, and technology limits. After that, I met with each team one-on-one again to make sure they could all use Neo4j in a consistent and helpful way. The key goals of these follow-up discussions were to learn about each team's codebase structure, the components they made, and the types of nodes and interactions that would need to be shown in the graph. This stage was very important to make sure that the final Neo4j model would be full, reflective of the whole system, and able to handle integrated queries across team boundaries.

After this technical mapping effort, all the teams got together for one more meeting to plan the structure and architecture of the shared Neo4j instance. During this meeting, people had different ideas on the best way to model and how detailed it should be. This will be explored in the next subchapter. But this engagement process was a big change in the way the company handled software dependencies since it set the stage for a cross-team, unified strategy supported by graph technology.

3.4.1 Concerns Raised

A number of issues with the suggested implementation of Neo4j across all teams surfaced after the individual and group meetings with the development teams working on the VTV project.

Concerns about the perceived cost-benefit ratio of deploying Neo4j were among the most commonly expressed. The amount of work needed to completely incorporate the tool into their workflows caused some team members to voice concerns. They believed that the effort required to map dependencies, configure an instance, and maintain the data structure would exceed any possible advantages. These worries were particularly noticeable for teams that had either already implemented partial tracking mechanisms using other tools or had not yet encountered serious dependency-related problems.

One team, for instance, contended that since they had previously recorded high-level dependencies in an Excel file (3.2), a shared instance of Neo4j with general, cross-team data would not significantly improve their daily tasks. The Excel file was not only difficult to read but also demonstrated the necessity of a new tool. They believed that what they needed was a method to make sure that modifications to their codebase could be promptly compared to these well-known high-level dependencies, rather than a complete graph-based visualization. It was recommended that instead of delving into an excessively detailed level of modeling that would require a large initial investment, each team concentrate on mapping their own internal dependencies and connecting them, when required, to a common framework.

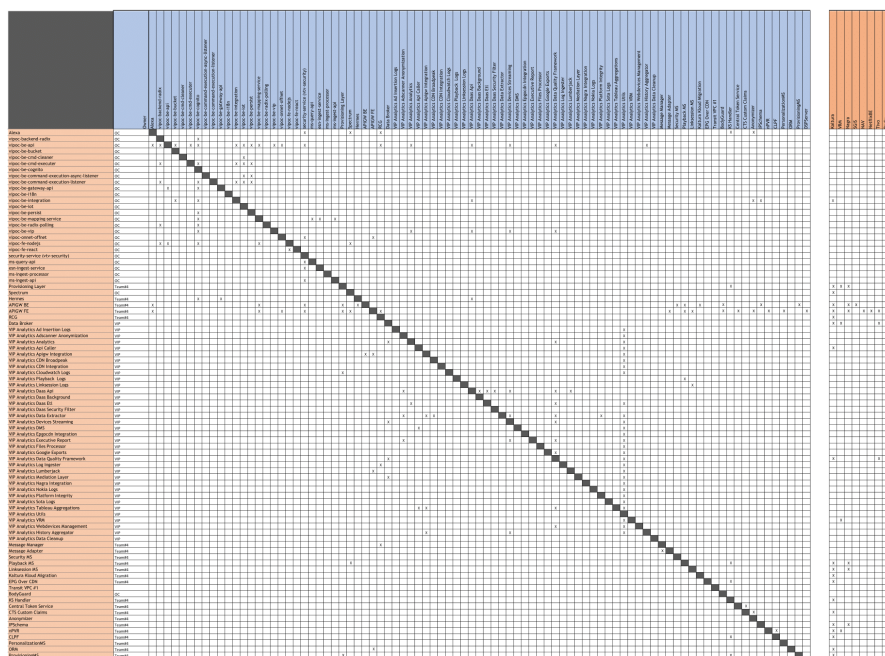


Figure 3.2: Excel with high level dependencies

In the end, a compromise resulted from these divergent opinions. Given the short time frame of this dissertation, it was decided that one of the teams would not deploy their own Neo4j instance because doing so would require an excessive amount of time and resources. On the other hand, a different team did move forward with setting up their own Neo4j instance, customized to meet their own requirements. A global instance with less granularity was created concurrently. Without requiring each team to model their systems in great detail, this shared instance was designed to give developers enough system-wide visibility to help them understand, on a broad level, which parts of the VTV program could be impacted by changes.

The Neo4j learning curve was another significant issue brought up during the meetings. A number of team members brought up the fact that in order to use the tool, developers would need to become familiar with the Cypher query language, comprehend graph structures, and modify their logic to accommodate a new method of modeling dependencies. This extra learning requirement was perceived as a barrier for teams that were already under pressure from sprint deadlines. Some thought it was a waste of time and effort to acquire the training and experimentation required to become proficient in Neo4j, especially for teams with fewer perceived dependency challenges. On the other hand, some believed that learning was a valuable investment that would eventually result in increased control and comprehension.

Long-term maintenance was the last area of concern. Even among those who acknowledged Neo4j's worth, there was doubt about the tool's continued usefulness in the absence of a consistent effort to update it. The usefulness of Neo4j rests on its capacity to represent the system's current state. The graph needs to be updated whenever a dependency changes or a new component is added. There were concerns that the database might soon become outdated in the absence of clearly defined roles and process integration. In order to guarantee that Neo4j would not be merely dropped after the initial implementation phase, a number of conversations focused on the necessity of defining ownership models, integration into the development lifecycle, and automation opportunities.

In conclusion, the interaction with teams uncovered a variety of viewpoints regarding the suggested Neo4j implementation. Others raised serious concerns regarding the workload, learning requirements, maintenance difficulties, and the relative merits of generalization versus customization, even though some saw obvious benefits. The ultimate plan, which balanced localized, team-specific graphs with a shared, global instance that would act as a point of reference for cross-team coordination with manageable granularity, was greatly influenced by these conversations.

3.4.2 Discussion and resolution

After talking with all the teams working on the VTV project, a series of joint meetings were set up to come up with a clear plan for how to use Neo4j that would address the issues that had been raised before, while still being possible within the time and resource limits of the dissertation period.

One of the main issues that came up in these meetings was that it would not be possible for one of the teams to make a separate Neo4j instance in the time they had. This was because their

codebase had a lot of detail, which would take a long time to model correctly in Neo4j. The amount of work needed for this process would be much more than what was planned for this project. This team also kept an internal high-level view of their dependencies through other documentation methods that, while not very detailed, were good enough for what they needed right now. So, everyone agreed that this team would use and add to the shared, global Neo4j instance instead of making a separate, detailed graph of their own parts.

On the other hand, a different team that had never used Neo4j before went ahead and made their own instance. This instance was built with more detail, which accurately shows how their internal software architecture works and lets them keep an eye on and manage their own dependencies in more detail. This detailed approach helped them see more clearly how changes to their code could affect certain functions and possibly cause problems later on.

At the same time, a shared Neo4j instance was created and set up to be used by all teams. The goal of this global graph was not to show every detail of each team's codebase, but to give a high-level picture of how different modules and services were linked together in the VTV project. The purpose of this global instance was to give all developers a single place to look up how their code changes might affect the work of other teams. By keeping this view of the whole team, developers could more easily figure out which parts of the system were likely to have changes spread, which helped avoid problems during deployments.

As part of the agreed-upon solution, it was also agreed that each team would be in charge of keeping their own Neo4j graphs up to date. It was decided that developers would have to look at their user stories and say whether any changes added new dependencies or changed existing ones during each sprint. In these situations, the appropriate Neo4j graph, whether it is local or shared, would need to be updated as needed.

Also, any change that could have an effect on other teams had to be clearly communicated. Developers were told to get in touch with the right teams and let them know about the change in dependency. This way, both sides could check the effect and get ready for any possible outcomes. The goal of this step was to make Neo4j a part of the larger collaborative process, turning it from a passive documentation tool into an active way to find and fix problems early and work together.

By getting everyone on the same page about this decision, the use of Neo4j became more than just a technical fix; it became a group effort. The project made a significant step toward solving long-standing dependency issues in the VTV ecosystem by putting in place a clear maintenance process and a unified strategy for communication between teams.

3.4.3 Conclusions of Coordination and Decision to Implement Neo4j

After the engagement process, the teams working on the VTV project looked at the issues that had been brought up and had long discussions about them. This led to a clear plan for how to implement Neo4j. The decision was based on a consensus that took into account the project's technical goals and the practical limits set by time, team resources, and the dissertation's scope.

The decision was made to use a hybrid structure for Neo4j, which would include a global Neo4j instance and separate instances for certain teams. The global instance was supposed to

be a high-level, central map of how different teams in the VTV project depended on each other. The main goal was to help developers from any team figure out which parts of the system their changes might affect and which other teams might be affected, without having to model each internal service in great detail. This instance would work with less detail, putting accessibility and a system-wide overview ahead of technical detail.

It was also agreed that each team could set up its own Neo4j instance to better keep track of internal dependencies. This was especially helpful for teams that had complicated internal structures or didn't have any tools for visualizing dependencies before. But because some teams' codebases were so big and detailed, it was also decided that not every team would try to make a local implementation during this project. It was determined that the time and effort necessary for one of the teams to create their own detailed Neo4j instance surpassed what was achievable during the dissertation period. In that case, it would be enough for now to rely on the global instance and their current high-level documentation practices.

The decision was based on a number of factors, including whether it could be done within the time limits of the dissertation, whether it would add value to each team's daily work, whether the current methods of tracking internal dependencies were mature enough, and whether the teams were willing and able to keep their graphs up to date over time. One of the most important things that had to happen for the Neo4j-based solution to work was that each team had to be in charge of keeping the data up to date. So, it was decided that from now on, any developer who changed code that affected dependencies would have to make those changes in Neo4j as well. Also, if a change in dependencies could affect other teams, it would be necessary to let those teams know ahead of time about any deployments.

The end of the coordination efforts was this combination of shared and localized graph models, along with clear responsibilities for maintenance and communication between teams. The chosen strategy made sure that Neo4j would be a useful and long-lasting tool for managing dependencies in the VTV program. The next step would be to put this plan into action and make the system work as planned.

3.5 Planning and Technical Implementation

For the VTV project at Celfocus, using Neo4j required a well-organized and planned process. Because there were so many different teams working on the project and the system's architecture was already very complex, it was clear from the start that any attempts to use graph-based dependency management would have to be phased, collaborative, and carefully planned. The planning and preparation phase was the basis for all the work that came after it. It dealt with not only the technical infrastructure but also team engagement, change governance, and long-term maintainability.

The first big choice made during this phase was to build a global Neo4j instance that all teams working on the VTV program would use to show how they were all connected, as shown in 3.3. This instance was made to be the main source of truth for interactions between teams. The strategic idea behind this choice was that by starting with a more general and abstract view

of dependencies, the teams could benefit from better visibility and coordination right away, even before they got into more detailed modeling. The global instance would have nodes for each team's key services, components, or deliverables. The relationships between these nodes would show which components depended on or worked with others. This setup let developers answer the important question, "What other parts of the system might be affected if I make a change here?"

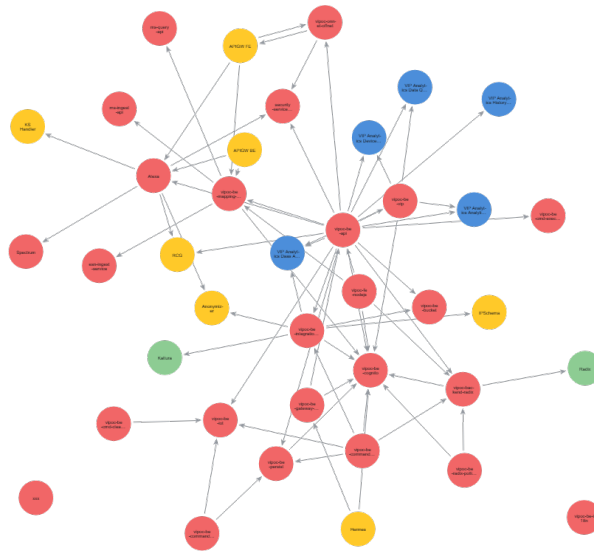


Figure 3.3: Neo4j instance for all teams

After this shared structure was set up, the next step was to make a separate Neo4j instance for the team that had not yet started using the platform. This team had previously expressed doubts about the practicality of deploying a graph database because their codebase was very detailed. There were a lot of small, interdependent modules in their system, which made it very time-consuming to model all of the parts by hand. The planning phase recognized the difficulty and made the design of their instance a priority. They also made support systems a priority by borrowing ideas from the first team to successfully use Neo4j to speed up the process. There were templates for defining nodes and relationships, as well as rules for how to group parts and make useful connections between them. The goal was not to model every detail right away, but to build a structure that could grow over time.

Along with setting up the Neo4j instances, a third important part of the planning process was setting up a GitHub repository that would serve as the main hub for managing all graph data through code. Instead of using the Neo4j browser interface directly, which could lead to accidental deletions, inconsistencies, or changes that weren't tracked, it was decided that all updates to the

graph would be made through Cypher scripts stored and versioned in this repository. These scripts set up the graph's structure, which included all of its nodes, properties, and connections. This method made sure that every change to the database could be audited, reviewed, and traced, which is in line with best practices in software engineering.

A contribution workflow was set up to make data integrity and operational reliability even stronger. If a developer or team member wants to make changes to the Neo4j instance, they must send a pull request to the repository with the changes they want to make. Each team chose at least one reviewer (two in general) to make sure that the changes were correct. The changes could only be combined and put into action after they were approved. This process not only reduced the chance of human error, but it also spread ownership and responsibility among teams, which kept the knowledge base going and stopped any one point of failure.

Another important part of the planning phase was figuring out how this repository would work with the Neo4j instances that were already running. The system was set up so that changes made in GitHub could automatically update the database. This could happen through a CI/CD pipeline or a backend service that was connected to the database. This automated deployment process got rid of the need to enter data by hand again and made the time between planning and execution shorter. It also meant that changes could be made in a developer environment that the teams were already used to, like code, version control, and peer review. This was better than having to learn how to use the Neo4j interface, which was harder.

During this phase, team leaders were very important in making sure that the graph data was correct and useful. I worked closely with each team lead to find the most important parts of their systems and figure out how they were linked to each other and to the outside world. It was very important that they helped make sure that the nodes showed real, working code instead of old or abstract documentation. These talks also helped find undocumented or informal dependencies that were important but had not been tracked in any formal way before.

At the end of the planning and preparation phase, a strong structure was in place: a global instance to capture high-level dependencies (3.4), individual team instances for detailed modeling, a GitHub repository to manage changes, and a governance process to make sure everything is correct, accountable, and easy to maintain in the long run. This foundation made it possible for the project to move forward with clarity, consistency, and agreement among all stakeholders. It also set the stage for the concrete implementation of Neo4j in a way that was both technically sound and operationally sustainable.

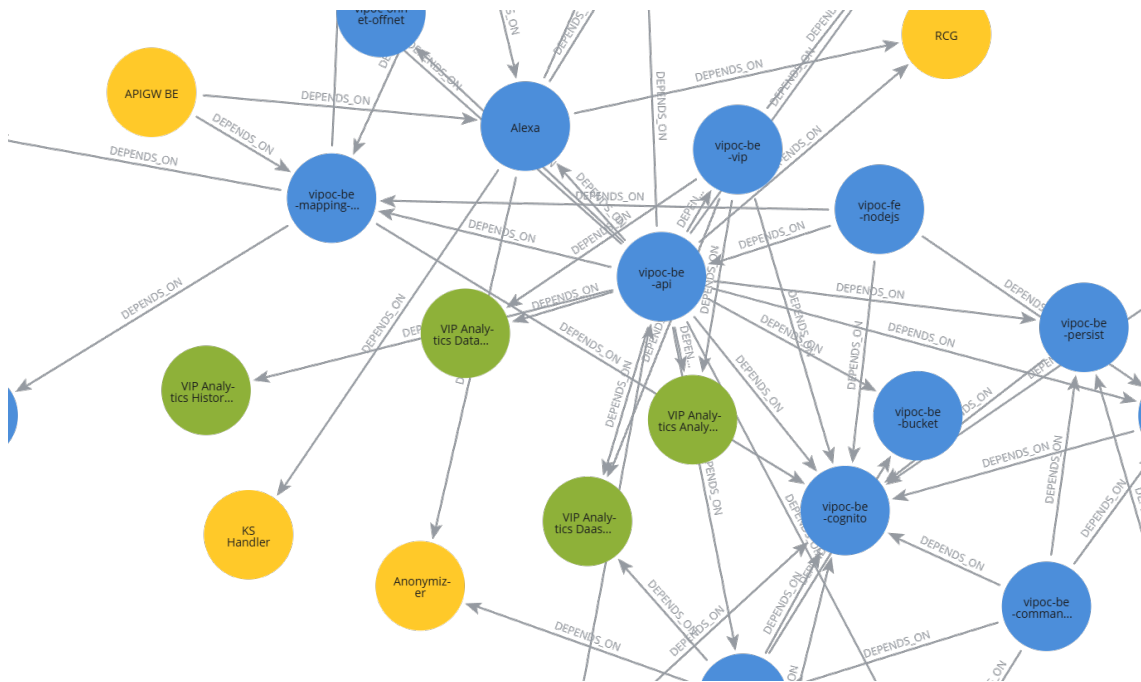


Figure 3.4: General instance of Neo4j

The implementation of Neo4j into the VTV program at Celfocus was done in a multi-layered and carefully planned way. This was done to not only show how the system was built, but also to create a way for multiple development teams to keep track of and understand software dependencies that was easy to maintain and work with. The first step in the implementation was to set up a shared, general Neo4j instance that would serve as the basis for the dependency model. This instance was made to be useful and easy for all of the teams involved. It gave a high-level view of the system's architecture and how different services and modules are connected.

One of the most important choices at the start of the process was to set up the graph with different node labels based on who owned each team. Nodes were given specific labels that showed where they came from instead of using a general label like Component or Service. These labels were External, Team4, OC, and VIP, as seen in 3.5. Using these four groups made it easy to understand the graph right away. Any developer could quickly tell if a node represented a component made by their own team, another internal team, or an outside vendor or partner.

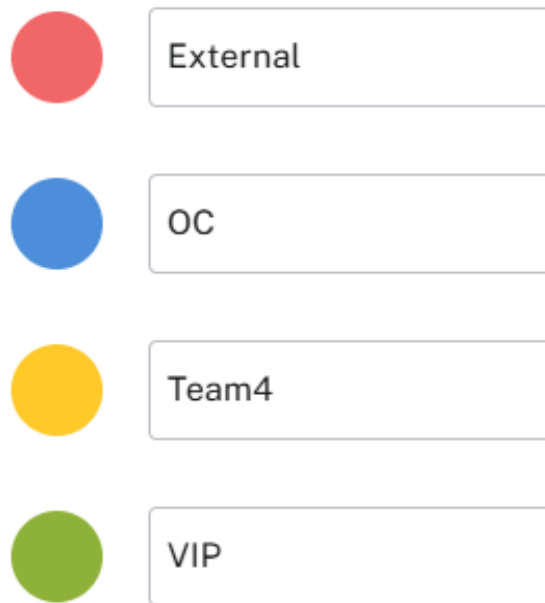


Figure 3.5: Teams represent different node types

The label OC was used to make nodes that were part of VIPOC. This team created vipoc-backend-radix, one of the main backend services, and it was defined in the database using the Cypher query below:

```
1 CREATE (:OC {name: 'vipoc-backend-radix'});
```

Also, components that VIPCE owned were labeled VIP, like the Cognito-based authentication service:

```
1 CREATE (:VIP {name: 'vipoc-be-cognito'});
```

The Team4 label was used to show Team 4's work, which mostly involved frontend and UI services. The team4-ui-core component is one example. It was the main library for many UI features:

```
1 CREATE (:Team4 {name: 'team4-ui-core'});
```

The External label was used to group together third-party services and external technologies. Some of these were important dependencies, like Kaltura, Nagra, Spectrum, and other proprietary systems that were built into the VTV architecture:

```
1 CREATE (:External {name: 'Kaltura'});
```

After the nodes were completely defined and grouped by team or source, the next step was to build connections between them. These relationships showed real operational dependencies, where one service or module needed another to work or to work together at runtime. All of the relationships were labeled with the type `DEPENDS_ON`, which meant that one node's functionality was directly affected by another, as seen in 3.6. If the `vipce-gateway-api` needed the `Cognito` service to log in, the relationship would look like this:

```
1 MATCH (a:VIP {name: 'vipoc-gateway-api'}),  
2     (b:VIP {name: 'vipoc-be-cognito'})  
3 MERGE (a)-[:DEPENDS_ON]->(b);
```



Figure 3.6: Relationship between nodes

What made this implementation even better was the ability to add properties to relationships, which let developers put context into the graph. These properties could include reasons for the dependency, the protocols used for communication, or the parts of the system that would break if the relationship were removed. For instance, if there is a connection between `vipce-gateway-api` and `vipce-be-cognito`, developers could write down what kind of authentication method is being used:

```
1 MATCH (a:VIP {name: 'vipce-gateway-api'}),  
2     (b:VIP {name: 'vipce-be-cognito'})
```

```

3 MERGE (a)-[r:DEPENDS_ON]->(b)
4 SET r.description = 'Handles login and token validation',
5     r.method = 'OAuth2',
6     r.interface = 'REST API';

```

This kind of annotation changed the graph from just a structural overview into a living library of information and reasons for each system dependency.

After the general Neo4j instance was finished and approved by stakeholders, work began on a separate, more detailed Neo4j instance for the VIPOC team, as seen in 3.7. This new instance built on the general graph by adding all of the previously defined OC nodes. It also added a new level of detail by including individual microservices that VIPOC developed in-house, as shown in 3.8. These nodes were given the name *MicroServices*, which let people model things in much more detail than the shared instance did.

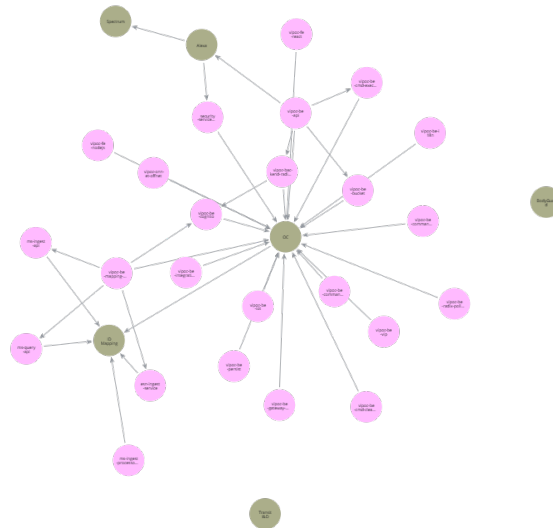


Figure 3.7: VIPOC's Neo4j instance

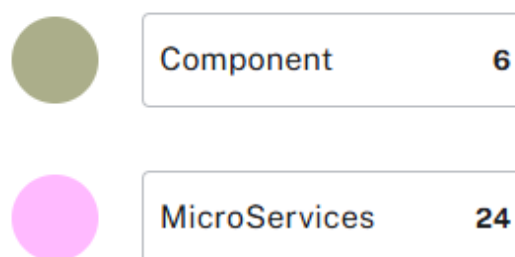


Figure 3.8: VIPOC's node types

Unlike the general instance, VIPOC had two different relation types between nodes as seen in 3.9, one related to an instance belonging to another, and the other about dependency only. This would help when consulting the general instance, as the deeper nodes always belong to a general node, which would then be represented in the general Neo4j instance.



Figure 3.9: VIPOC’s instance different relations

The VIPOC-specific instance would break this down into its parts and show which microservices were in charge of the interaction. For instance, it could say that vipoc-be-command-execution-async-listener was the internal part that used VIPCE’s Cognito service:

```
1 MATCH (a:MicroServices {name: 'vipoc-be-command-execution-async-listener'}),
2     (b:VIP {name: 'vipce-be-cognito'})
3 MERGE (a)-[:DEPENDS_ON]->(b);
```

This higher level of detail was especially useful for debugging, isolating services, and figuring out how risky something was. It helped VIPOC developers figure out not only which external component was causing a problem, but also which internal service was affected by it or caused it.

A script-based automation system was made to make sure the graph stayed accurate over time and could be updated quickly. This system was kept in a private GitHub repository and was meant to run against the Neo4j database whenever an update was needed. The main idea behind the automation was to delete all the nodes and relationships that were already in the database and then rebuild the whole structure using the most recent and verified script. This method reduced the chance of having duplicate entries, relationships that don’t work, or old data that stays in the graph.

The entire database structure, including all of the nodes, their categories, and how they are connected, was written in a .txt file that had the Cypher commands needed. GitHub kept track of different versions of this file, so any team member could see how the graph changed over time. To make changes to the file, developers would edit it, commit their changes, and then run the script to make the graph again. This method made sure that every change was planned, checked, and could be repeated, and that the graph would always show the current structure of the VTV codebase.

In short, the technical implementation of Neo4j at Celfocus started with a global instance that everyone on the team could see and use, based on the labels OC, VIP, Team4, and External.

This made it easy to see how different teams depended on each other. It was expanded with a more detailed, microservice-level model made just for VIPOC, which made it possible to do more detailed analysis and follow the data back to its source. And finally, it was put into action through a GitHub pipeline that was version-controlled, which made it safe and easy to update graphs. Neo4j became a key tool for understanding, documenting, and managing complexity in one of Celfocus's most important projects thanks to this structured implementation.

Chapter 4

Validation

The evaluation is based on two main types of evidence: qualitative insights gained from direct interaction with the teams during and after the implementation process, and structured feedback collected via survey and interviews from the team that had been using Neo4j for a longer period of time.

This chapter will talk about both the good things and the problems that still exist during this early adoption stage. It seeks to give a critical look at Neo4j's contributions to better dependency tracking, how users see the tool, and where it could be improved. The results shown here will help shape the more general conclusions of the last chapter and show where future research could build on what has already been done.

4.1 Methodology and Participant Characterization

A structured questionnaire and a series of informal interviews were used to check how well Neo4j was used in the VTV program and what effect it had. The goal of these validation methods was to get both quantitative and qualitative feedback. This would give a better picture of how useful the tool was seen to be, what problems it caused, and how it was adopted by the teams that used it.

The main way to check was to send a questionnaire to the VIPCE team, which was the group that had already started using Neo4j before it was made available to everyone else. This team was chosen because they were among the first to use the tool and actively used the graph-based dependency management system during its early stages. The purpose of the questionnaire was to get feedback on different aspects of the experience, such as onboarding, ease of use, perceived value, frequency of use, and overall satisfaction. Responses were made anonymous to make people feel more comfortable sharing their thoughts.

At the same time, unofficial interviews were held with people from all teams who had used Neo4j or talked about how to use it. These interviews weren't recorded or transcribed in a formal way, but they added a lot of value to the survey data by giving it context and nuance that helped explain the motivations, expectations, and concerns of different stakeholders. Interviewees included

developers, team leaders, and other people who worked on the project. They talked about how the organization worked when it came to managing dependencies and using new tools.

It's important to talk about the VIPCE team members who answered the survey so that the results make sense in context. Five people from the team filled out the questionnaire, which included people from different roles and responsibilities. Two of the people who answered were developers who work on coding and integrating systems. One was a quality assurance (QA) engineer who tests and makes sure that deployments are reliable. One was a Scrum Master who makes sure that Agile processes are followed and that the team is on the same page. One was a Team Leader who makes sure that the technical direction and project delivery are on track. The different roles made sure that the feedback came from a variety of points of view on how to use Neo4j, from planning to implementation to quality assurance.

This mix of structured and informal validation gave a full picture of the early results of the implementation. It also helped find patterns and differences in how people saw things based on their roles, expectations, and level of technical knowledge. The next sections show and talk about the results of this validation process.

4.2 Results from VIPCE

VIPCE was made up of people in a variety of technical roles by the time Neo4j was released, which resulted in a wide range of viewpoints on how dependency problems could be fixed. Developers, a quality assurance (Q&A) engineer, a team leader, and a Scrum Master all provided input during the assessment process, for a total of six participants. Each participant had a background in information technology, having studied computer science or closely related fields in school. They had a wide range of years of experience at Celfocus. While some had been working on the VTV project for over five years and had a more comprehensive historical understanding of how dependency issues had changed over time, others had only joined within the last 12 months and were still establishing their familiarity with the internal systems.

To finalize the empirical component of this thesis, a structured questionnaire was administered to members of various development teams in order to gather qualitative and quantitative insights into their experience with dependency management and the use of Neo4j. The questions were designed to elicit information about participants' backgrounds, practices, and perceptions, with a particular focus on identifying challenges and evaluating the impact of the introduced tooling. The form included the following questions:

- What is your role/function in the team?
- How long have you been working in the company/team?
- Do you have formal training in Information Technology (IT)?
- What do you understand by "dependency management" in software systems?

- How many years of experience do you have in dependency management?
- Which tools have you used to manage dependencies?
- Did you participate directly in the adoption of Neo4j, or did you start using it afterwards?
- With what frequency do you utilize Neo4j?
- What is your level of autonomy when using Neo4j?
- In what types of tasks do you use Neo4j?
- In your team, who typically updates the data in Neo4j?
- How are changes to dependencies approved?
- Do you consider that the information in Neo4j is generally up to date?
- Before Neo4j, how was dependency management performed?
- Describe at least two cases of intra- or inter-team dependency management issues.
- Do you think Neo4j helped resolve these problems? Why?
- In your opinion, what has been the impact of Neo4j on dependency management?
- Can you provide concrete examples of improvements or difficulties encountered since Neo4j's implementation?
- If you had to quantify it, how much has visibility and control over dependencies improved?
- What suggestions do you have for improving the use of Neo4j in dependency management?
- Is there anything you would like to see implemented that would facilitate your work?
- Any other relevant comments or experiences you would like to share?

The responses to these questions provided a multifaceted view of the current practices, perceived benefits, and limitations of Neo4j in the context of managing complex software dependencies, and directly informed the analysis and conclusions presented in the following sections.

Divergent opinions on the application and utility of Neo4j were a reflection of the differences in tenure and roles. Neo4j had a more hands-on relationship with developers who were more directly involved in the actual implementation of services, especially in terms of their comprehension of how it could assist them in avoiding technical regressions throughout the development cycle. Conversely, more senior positions like the Scrum Master and team leader tended to consider how Neo4j aided in sprint planning, team coordination, and higher-level decision-making. Instead of being a barrier, this difference in viewpoint was crucial in determining how Neo4j could be used for both technical accuracy and cross-role communication.

To comprehend the significance of the feedback given, this team profile is necessary. A detailed assessment of Neo4j's contribution to the team was made possible by the combination of strategic oversight and practical technical knowledge. The comments also aid in placing subsequent findings in context: many of the respondents had used or experimented with other tools like Jira, Maven, and Excel, and they were not unfamiliar with dependency management procedures. Nevertheless, in spite of this experience, Neo4j was seen as offering something novel, especially its ability to visually depict relationships that were previously only implied in code or unofficial documentation.

4.2.1 Learning Curve and Onboarding Experience

Team members had to become acquainted with ideas and tools they had not previously utilized in their daily work because the deployment of Neo4j brought a new method of visualizing and interacting with software dependencies. The VIPCE team did not have a smooth or quick onboarding experience with this graph-based system. Some developers were more involved in forming the Neo4j instance's original structure, while others didn't start using it until its initial iteration was finished and filled with nodes and relationships.

According to comments gathered from the survey and unofficial interviews, Neo4j was the first time that many team members had interacted with graph databases. There was agreement that the early learning stage was characterized by a substantial cognitive shift, even though its visual and relational model was thought to be intuitive once understood. For developers used to working primarily with relational databases or object-oriented paradigms, the logic of traversing nodes and relationships using Cypher was new and needed practice.

According to 80% of respondents, they became familiar with Neo4j through experimentation and exploration. Some reported depending on external documentation and online tutorials, while others said they were first exposed to its fundamental features during internal presentations or impromptu demonstrations. Each team member became accustomed to the tool at a different rate because there was no official training or organized onboarding materials. "Knowing how to add a relationship wasn't the hard part—the difficult thing was understanding how the overall structure worked and what would break if I added or deleted something," one developer said. This quotation exemplifies the larger worry that improper use of the graph may result in false or deceptive depictions of system dependencies, especially in the lack of explicit rules regarding graph modeling conventions.

The topic of confidence was another pertinent element that came to light during the interviews. Many respondents expressed some hesitancy about using Neo4j directly, particularly when it came to updating or changing the graph. This was mostly because people were afraid of inadvertently changing or erasing crucial data. As a result, some team members decided to avoid direct editing entirely or restricted their use of Neo4j to tasks involving navigation and visualization, or they sought advice from more seasoned colleagues before making any changes. This behavior demonstrated how crucial it is to set up secure procedures and role-based access for graph updates. This issue ultimately prompted the development of version-controlled update scripts through GitHub.

Notwithstanding these initial challenges, the majority of team members reported that, although challenging at first, the learning process became tolerable with more exposure. With the exception of one person, everyone reported that they could use the tool with little assistance after a few encounters, especially when it came to querying and examining already-existing nodes and relationships. But the learning curve continued to be a barrier to broader adoption. The time needed to learn and adjust to a new platform was frequently seen as a barrier in the context of rapid sprint cycles, where developers are frequently under pressure to deliver functional features as soon as possible.

4.2.2 Usage Frequency and Interaction Patterns

Patterns about the frequency and context of Neo4j's usage started to show as the tool was incorporated into the VIPCE team's development environment. The frequency of usage was directly related to the size and type of each developer's tasks, according to one of the most obvious conclusions drawn from the survey and the unofficial interviews. As shown in 4.1, some team members used Neo4j on-demand, especially during more complicated deployments or integration tasks that needed cross-team coordination, while others engaged with it on a regular basis.

Responses to the survey revealed a variety of usage patterns. According to some participants (4.1), Neo4j was consulted once a month, especially when a feature or story included modifications that could affect or rely on outside sources. Some used it only when absolutely necessary, usually during planning sessions or retrospectives where the visual map of dependencies provided helpful context, while others interacted with it every few weeks. These differing frequencies were a reflection of the tool's perceived purpose rather than a sign of limited adoption. Neo4j was a contextual support tool used during architectural assessment or decision-making rather than being a part of the daily development loop.

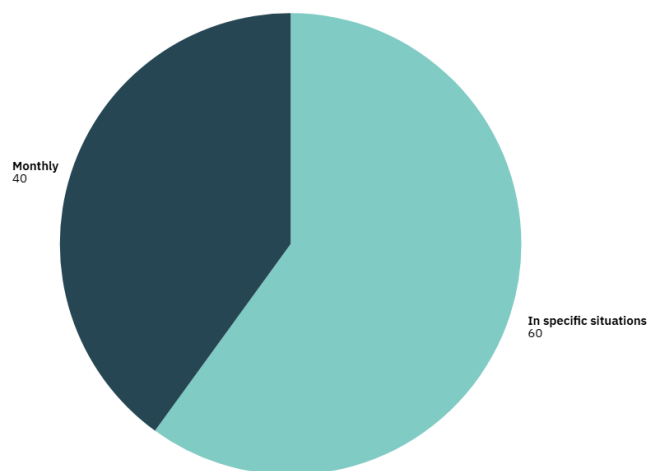


Figure 4.1: Answers to the question "With what frequency do you utilize Neo4j?" in percentage

Neo4j was utilized by most team members for verification and exploration as shown in 4.2. It was common practice to query the graph to find out which other services might be impacted by a particular code change or to confirm if a new feature would necessitate updating the components of other teams. Neo4j was essential to impact analysis in this way, providing developers with a visual and queryable map of the system components that their work might indirectly affect. Developers who participated in interviews emphasized how this visibility was particularly helpful in sprint planning because it allowed them to anticipate possible obstacles or coordination requirements.

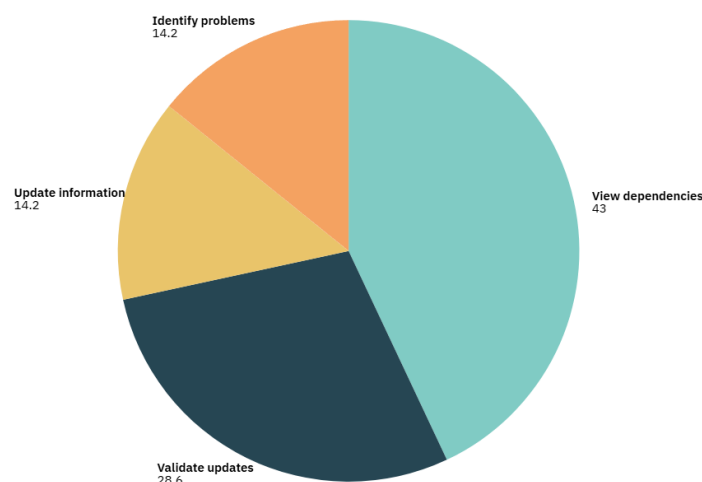


Figure 4.2: Answers to the question "What tasks do you utilize Neo4j for?" in percentage

4.2.3 Perceived Utility and Added Value

The VIPCE team's use of Neo4j added a new level of insight into software dependencies, and its usefulness was assessed not only by how easy it was to use, but also by how much it improved the team's overall development process. It was clear from the survey results and the interviews that Neo4j was seen as a tool that greatly enhanced the team's ability to manage, understand, and respond to system dependencies, despite certain drawbacks.

The ability of Neo4j to centralize and visualize relationships between components that were previously only partially documented or stored in team memory was cited by a developer as its most immediate value. Prior to the graph model's release, the majority of dependency tracking was done by hand or dispersed among Excel sheets, Jira tickets, or casual discussions. This disjointed visibility was replaced by Neo4j's coherent and queryable structure, which was especially helpful when attempting to comprehend how modifications to one area of the system might affect other areas. One team member highlighted Neo4j's role as a predictive safeguard during planning stages by remarking that it "helped prevent surprises at deployment time."

Neo4j's contribution to collaborative awareness was the source of another aspect of perceived utility. Its existence promoted improved cross-role communication, even for users who did not directly manipulate the graph. The same visual references were used by developers, team leads, and

Q&A specialists to evaluate the interdependencies between functionalities. This shared mental model was helpful, particularly when negotiating responsibilities across team boundaries, estimating story complexity, and getting ready for sprint reviews. Using Neo4j interface screenshots, which are frequently shared in team channels or incorporated into planning meetings, has become a common practice in workflow discussions.

According to everyone, the graph database also made it easier to onboard new team members. To better understand system structure, new developers could use Neo4j to visually and conceptually explore relationships rather than depending on long explanations or out-of-date documentation. This decreased reliance on senior developers' availability to repeatedly explain the architecture and expedited their integration.

Neo4j was also acknowledged for helping to detect architectural inconsistencies like unintentionally duplicated relationships, cyclic dependencies, and orphan nodes. In complex systems, these problems frequently appear gradually and are challenging to identify with conventional dependency tracking tools (4.3). The team was able to identify these patterns through targeted Cypher queries using Neo4j, which facilitated speedier diagnosis and correction.

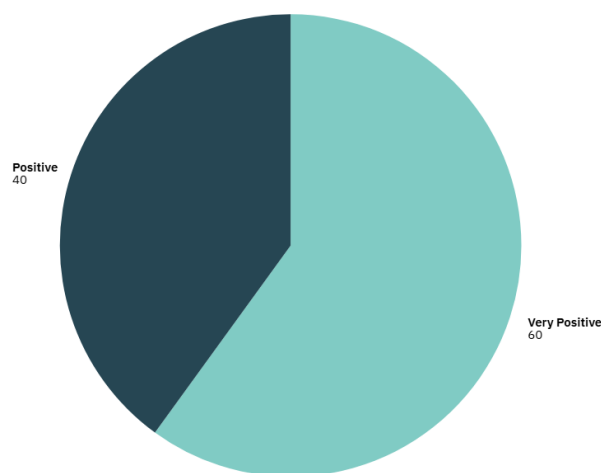


Figure 4.3: Answers to the question "In your opinion, how was the impact of neo4j in dependency management?" in percentage

However, Neo4j's worth extended beyond its technical usefulness. A change in team culture regarding documentation and accountability was attributed to its implementation, according to a number of team members. Some developers adopted the practice of updating the graph as a fundamental component of responsible software delivery. This change represented a noticeable departure from earlier methods, where dependency tracking was more passive and reactive, even though it wasn't always implemented flawlessly.

4.2.4 Limitations and Suggestions for Improvement

Although integrating Neo4j into the VIPCE team's workflow produced significant advantages in terms of project coordination and dependency visualization, its adoption also revealed a number of drawbacks that affected the tool's efficacy and consistency of use. Complementary qualitative interviews and structured survey responses were used to identify these limitations.

One significant drawback that surfaced was the amount of work needed to maintain the graph database's regular updates. Maintaining an up-to-date representation of changing dependencies still required team discipline and manual contributions, even with the establishment of a GitHub-based version control system that allowed database updates to be scripted and reviewed prior to execution. As a result, Neo4j updates were often postponed or deprioritized during times of high development pressure or when user stories required complete focus. This limited the graph's dependability for making decisions by creating circumstances in which it did not appropriately depict the system's current state.

There were additional repercussions for user trust from this problem. When developers had doubts about the completeness of Neo4j's data, they were frequently hesitant to rely entirely on it. Before using the graph to inform planning or architecture choices, they would occasionally cross-reference data with information from other documentation sources, like issue trackers or team discussions. This behavior highlighted a paradox: the graph database's usefulness was directly correlated with the accuracy and regularity of its maintenance, even though its value increased with usage.

The learning curve for Cypher, the query language used in Neo4j, was another issue that was mentioned. Basic graph exploration was made possible by Neo4j's visual interface, but some technical fluency was needed to execute complex queries or make structural changes. Direct interaction with the database was generally avoided by developers who were not familiar with declarative languages or who had little experience with graph-based thinking. As a result, there developed an unofficial division of labor, with some users using the graph mainly for visualization and a smaller, more self-assured group taking on the task of maintaining and expanding it. This kind of dependence on a small number of people limited collaborative ownership and increased the possibility of bottlenecks.

Concerns were also expressed regarding the graph's representations' level of granularity. To emphasize high-level interdependencies, nodes and relationships were purposefully abstracted in the general instance that was shared by all teams. Although this promoted cross-team alignment and expedited planning, some developers felt that this level of abstraction was insufficient to comprehend the nuances of technical integrations. On the other hand, although it required more maintenance, the more intricate VIPOC-specific instance captured relationships at the microservice level. A layered or hierarchical graph model, where users could switch between abstraction levels based on the situation, was required, as indicated by the contrast between these two levels of granularity.

Team members made a number of recommendations for enhancements. The potential integration of Neo4j with automated tools that can extract dependency information from pre-existing source code, build files, or CI/CD pipelines was one of the most often mentioned. This would boost trust in the database's accuracy and lessen the need for manual updates. Making onboarding materials like quick training videos, sample query libraries, or interactive guides was another recommendation. These could help new users adopt the system and lower the barrier to entry for non-experts.

A person suggested extending the use of standardized naming conventions and schema definitions, especially for defining relationships and node types, in order to better support future scaling. This would eliminate ambiguities in the modeling of dependencies and guarantee consistency as more teams use the graph.

4.2.5 Cultural and Communication Impacts

The VIPCE team's adoption of Neo4j resulted in significant changes to team culture, communication styles, and shared accountability for architectural knowledge in addition to its technical capabilities. Although supporting dependency management was the implementation's main objective, during the first phase of use, a number of unexpected but positive cultural effects were noted.

The advent of a more proactive approach to documentation and impact awareness was one of the biggest cultural changes. Teams found it more difficult to ignore how their changes affected other teams or components when there was a central graph representation. The graph required explicit modeling of every relationship, whereas before Neo4j, dependency information was dispersed or informal. This prompted developers to engage in preventive coordination prior to deployment and to think more deeply about the possible consequences of their work. Connection visibility, particularly in the general instance shared across teams, reaffirmed the value of system-wide thinking over the execution of discrete tasks.

The existence of a shared dependency model also had a positive impact on team communication. The Neo4j graph frequently functioned as a shared visual reference during planning meetings or backlog refinements, promoting conversation about possible integration points, roadblocks, or architectural risks. When nodes that were mapped to external teams were impacted by changes or when graph updates exposed new dependencies, developers started reaching out to other teams more frequently. This type of early-stage communication made it possible to address problems upstream rather than at the integration or release point, which helped to lower friction later in the sprint cycle.

Neo4j also helped dismantle knowledge silos in a number of instances. Team members started using the graph to independently investigate service interactions rather than depending entirely on team leads or senior developers to explain cross-functional relationships. This change decreased reliance on particular people and increased team accessibility to architectural knowledge. Updating the graph was becoming more widely recognized as a collaborative effort rather than a solitary duty, even though it was still focused on important contributors.

But this shared accountability also brought up governance issues. In the absence of explicit ownership protocols, it was occasionally unclear who was in charge of updating which graph segment. This was especially true in the general case, where it was more difficult to enforce regular updates due to overlapping responsibilities. The team responded by implementing a lightweight procedure that required each user story to specifically state whether a Neo4j change was required. By acting as a checkpoint during sprint completion and code reviews, this made sure the graph stayed in sync with the underlying system.

The way team members viewed cross-team accountability was one area where the cultural shift was particularly apparent. It was harder to ignore how changes made by one team might impact others after the graph revealed connections. With the addition of new relationships or the modification of existing ones, developers became more intentional in their interactions, frequently sending messages or recording side effects. This practice promoted an open and accountable culture that had previously depended more on impromptu communication or feedback from late-stage testing.

4.3 Feedback from Other Teams

4.3.1 Timing Constraints and Limited Exposure

One major constraint in assessing Neo4j's deployment across all teams working on the VTV project was when it was adopted. The other two teams only started using Neo4j in the last stages of the thesis period, whereas one team had already integrated the tool and offered structured feedback via forms and interviews. The final submission of this document coincided with the end of their first sprint of using the tool. Consequently, there was no chance to record long-term interaction cycles, collect longitudinal data, or evaluate behavioral changes over time.

The type of evidence that could be obtained was limited by the brief period between the initial adoption and the conclusion of the thesis. Although setup progress, early responses, and intended usage strategies could be recorded, these insights were insufficiently detailed for a thorough assessment. Additionally, it meant that some patterns, like consistent updates, real integration into planning processes, or sustained developer engagement, could not be thoroughly examined within the parameters of the thesis.

Furthermore, because the timeframe did not permit appropriate usage reflection, no official forms were given to the recently adopting teams. During the onboarding process, feedback was instead obtained informally through follow-up meetings and one-on-one discussions. These conversations were useful, but they couldn't take the place of the organized analysis made possible by data from longitudinal surveys.

It's also crucial to remember that these teams' initial Neo4j usage was mostly concentrated on filling in the graph and getting to know the interface. The teams were still in the early phases of testing, investigating how to use it, testing update processes, and figuring out how to incorporate it into their own development processes. Therefore, it was too soon to evaluate the Neo4j graph's

long-term usefulness or cultural influence within those teams because it had not yet established itself as a routine tool integrated into daily workflows or decision-making procedures.

These limitations do not imply that Neo4j was unsuitable for these situations; rather, they indicate that a more thorough assessment of its efficacy is necessary following a longer adoption period. The required temporal distance to examine persistent usage patterns, resistance points, or process transformation may be provided by upcoming research or internal evaluations conducted within the organization.

Recognizing this temporal constraint is essential to placing the findings in the larger framework of this chapter. The lack of extensive experience from two of the teams must be taken into account when assessing the complete impact of the Neo4j integration initiative, even though early implementation efforts can yield important lessons.

4.3.2 Informal Feedback and Early Impressions

Several types of informal feedback were gathered through interviews and direct observations during the onboarding and initial usage phases, even though structured evaluation through forms was not practical for the two teams that adopted Neo4j late in the project. These discussions provided insightful qualitative information about the tool's perceived usefulness, accessibility, and compatibility with the teams' workflows and expectations.

The first phases of involvement in both teams focused on comprehending the structure of the Neo4j graph, the data it recorded, and its potential applications in development tasks. The main purposes of developers' interactions with the graph were to examine component relationships, evaluate the precision of mapped dependencies, and test its applicability in daily tasks. The practicality of the interface, the clarity of the relationships shown, and the graph's initial utility in elucidating interdependencies among teams were the main points of feedback in this context.

One team reported that the onboarding process was simple and beneficial. The shared instance, according to the developers, provided an instant improvement over their prior procedures, which mainly depended on institutional memory or disjointed documentation. For onboarding newer developers with less prior knowledge of the system architecture, the ability to visually navigate between nodes and comprehend the relationships between components across team boundaries was considered a significant advantage. Despite the brief exposure period, this team showed a willingness to integrate Neo4j into regular tasks by using it as a reference point during backlog refinement and code review discussions.

The second team, on the other hand, showed a more circumspect disposition. While recognizing Neo4j's theoretical worth, some developers expressed doubts about how feasible it would be to update the graph on a regular basis and whether the long-term advantages would outweigh the extra work. The right amount of granularity was also questioned; some participants were concerned that if the general instance was made too detailed, it would either oversimplify meaningful relationships or require too much manual maintenance. Although they are early, these observations imply that adoption was viewed more as a technical experiment than a complete cultural change.

However, both teams reported that the onboarding procedure was straightforward and that high-level architectural decisions were simpler to comprehend thanks to Neo4j's visual structure. Neo4j's ability to lessen team misalignment and enhance pre-deployment coordination was also acknowledged by all. Developers frequently referred to the tool as a "map" or "system overview" in discussions, emphasizing its usefulness as a diagnostic and navigational tool.

4.3.3 Influence of Team Attitude on Results

One of the most significant findings from the early stages of Neo4j implementation was the obvious impact that team mindset and adaptability had on the tool's efficacy and perceived worth. Although technical expertise and onboarding materials were similar for all teams, teams' initial inclination toward Neo4j adoption seemed to be closely linked to how much they used and benefited from it.

Different opinions about Neo4j were immediately apparent among the teams that used it in the thesis's last sprint. One group welcomed the platform as a chance to increase openness, make it easier to understand the system, and simplify communication between teams. Developers independently investigated the graph, suggested changes to the structure, and started bringing it up in daily stand-ups and planning sessions as a result of this proactive attitude. The team was able to use the graph as a tool for collaborative planning, architectural awareness, and visual reasoning despite the short exposure period.

The other team, however, adopted a more tentative stance. Despite the completion of the onboarding procedure and the availability of the graph, little effort was made to incorporate the tool into routine development processes. Discussions uncovered skepticism about the visual model's long-term utility, concerns about the perceived overhead of maintaining the graph, and uncertainties about the return on investment. Because of this, the tool was only occasionally used, and during the observation period, its full potential was not fully realized.

This discrepancy reveals a larger trend that has been seen throughout the project: organizational and cultural preparedness to embrace new procedures is just as important to the success of graph-based dependency management as the tool's capabilities. Even in a short period of time, teams that showed flexibility and understood the strategic importance of architectural clarity were able to reap significant Neo4j benefits. On the other hand, teams that saw the tool more as an operational hassle than a structural benefit were more likely to put it off or use it inactively.

Crucially, these perceptual variations also affected the kind of feedback that was given. The more involved group identified particular situations where the tool had already aided in decision-making, offered specific recommendations for enhancing the graph structure, and indicated interest in extending its application to more granular levels. In contrast, the more doubtful group concentrated their remarks on issues related to maintenance work and uncertainty regarding the proper degree of abstraction. These varied viewpoints highlight how team mentality influences adoption and feedback trajectory in addition to usage patterns.

4.4 Observed Improvements

Even though Neo4j was only implemented for a brief time before this thesis was finished, the teams that used the tool saw a number of noticeable improvements. The extent and magnitude of these benefits vary based on the level of adoption, but they offer preliminary evidence of the usefulness of managing software dependencies in a complex, multi-team setting with a graph-based approach.

The improved visibility of cross-team dependencies was one of the most noticeable and frequent improvements that teams reported. Neo4j enabled developers to view system architecture as an explorable network rather than an abstract concept by converting the relationships between components, services, and external systems into a graphical structure. Because it can be challenging to determine how planned changes will affect external modules, this visibility was especially helpful during sprint planning. Teams reported that having the ability to query or navigate the graph helped them feel less uncertain about how their work might impact or rely on the services of other teams.

Early risk identification was also aided by this visualization. Several developers reported that they were able to foresee possible integration problems or conflicting updates that would have otherwise only come to light during testing or after deployment by looking at the graph before adding new features. In certain instances, this resulted in teams getting in touch early to talk about and reduce these risks before they became incidents. Despite their rarity, these instances demonstrated a change in the way teams handled coordination, from reactive solutions to proactive alignment.

By providing a centralized, queryable representation of the system, the graph structure made knowledge sharing easier. The Neo4j instance developed into a useful reference tool for comprehending past decisions and the reasoning behind component connections in situations where documentation was either absent or out-of-date. Newer team members found the graph particularly helpful as it allowed them to quickly understand the dependencies that were already in place without having to rely on sporadic technical notes or peer explanations. The Neo4j interface was specifically brought up during onboarding sessions in one of the teams, highlighting its function as a teaching and navigational tool.

Neo4j also enhanced shared language and collaboration in teams where the tool was utilized more frequently. During inter-team meetings, visual representations offered a common frame of reference and assisted in organizing technical discussions. Teams could identify specific contact points, define ownership, and point to nodes and relationships in place of ambiguous or abstract descriptions of "interfaces" or "shared flows." This effect, which lessens the friction that frequently results from misaligned assumptions, was particularly noticeable when working across teams with different domains or responsibilities.

Furthermore, the decision to connect Neo4j to a version-controlled repository, via a GitHub-based script that reconstructed the database from structured commands, introduced a governance model that balanced openness and control. This model decreased the possibility of unintentional data loss and established confidence in the graph's integrity by requiring code-level changes to

be examined and approved before being pushed to the live instance. Teams were assured of the system's dependability and the traceability and reviewability of any changes.

While not all teams saw these improvements equally, they showed how Neo4j could facilitate more thoughtful, open, and knowledgeable development processes. The advent of a graph database marked a fundamental change in the way dependencies were perceived and handled in an environment marked by frequent handovers, dynamic codebases, and interdependent modules.

These initial advantages imply that the graph's value may eventually be increased through deeper integration into Agile practices and DevOps pipelines. There is a growing alignment between dependency management and data-driven engineering practices, as evidenced by the interest shown by several teams in investigating automation for graph updates, integration with CI/CD triggers, and dynamic impact analysis as next steps.

4.5 Remaining Challenges

4.5.1 Balancing Granularity and Maintainability

The difficulty of choosing and sustaining a suitable degree of granularity in the modeling of software components and their dependencies was one of the main issues faced during Celfocus's Neo4j implementation. Although Neo4j's queryable nature and visual clarity offered significant advantages, the degree to which those advantages could be realized depended on how intricate the nodes and relationships were intended to be.

Nodes were broadly classified into four labels (OC, VIP, Team 4, and External) based on their team of origin in the general Neo4j instance, which comprised all teams. Teams were able to visualize the VTV project's overall architecture and represent key component blocks with the help of this high-level structure. But there were drawbacks to this coarse-grained strategy as well. The graph was unable to accurately depict interactions between smaller functional units or within components when dependencies developed that were not included in the model. Occasionally, developers who wanted to know how a modification to one module might impact another were left without enough resolution to derive useful conclusions.

The VIPOC team, on the other hand, created a more detailed Neo4j instance that added individual microservices as separate nodes within the OC components, extending the general model. Developers were able to determine precisely which internal service of one component was reliant on or being used by another thanks to this increased level of detail, which also made it possible to comprehend interdependencies more precisely. This method greatly increased the graph's utility for tasks like change impact analysis, testing coordination, and debugging.

But there were costs associated with this increased granularity. It took a lot of work to map microservices, keep nodes and relationships consistent, and update the graph whenever there was a logical or architectural change. When their work had an impact on the system structure, developers were supposed to update the Neo4j instance and incorporate graph maintenance into their sprint routines. This manual update procedure frequently clashed with other priorities due to the quick

development and frequent rollout of new features, resulting in missed updates or discrepancies between the live system and its graph representation.

Furthermore, there was no widely accepted cutoff point for how much information was "worth" showing in the graph. While some developers maintained that in order to minimize clutter, only shared APIs or externally exposed interfaces should be represented, others thought that internal flows and logic-level dependencies were crucial for onboarding and transparency. It was occasionally challenging to establish a consistent modeling approach across teams due to these differing viewpoints.

The lack of automation made the granularity problem even more difficult. The procedure remained completely manual because Neo4j updates depended on a script hosted on GitHub, requiring each node and relationship to be explicitly defined in Cypher and versioned. This scripting approach increased the cognitive and procedural load on developers, many of whom were still learning the Cypher language and the database structure, even though it offered version control and governance.

The difficulty of striking a balance between maintainability and granularity ultimately emerged as a major conflict in the Neo4j integration. Although accurate dependency tracking and impact forecasting were made possible by detailed modeling, teams ran the risk of becoming overburdened by the constant manual updates. The experience brought to light the necessity of explicit modeling guidelines, common inclusion criteria, and possibly future semi-automated node creation and dependency inference processes. It might become impossible to sustain granularity at scale without such mechanisms.

4.5.2 Lack of Centralized and Accessible Documentation

The lack of thorough and centralized documentation somewhat hampered the success of Neo4j's implementation at Celfocus, despite the fact that it brought about promising improvements in managing and visualizing software dependencies. This restriction was especially noticeable when it came to cross-team coordination, onboarding, and graph model maintenance. Many users, particularly those new to the company or not familiar with graph databases, found it difficult to fully comprehend the structure and intended use of the Neo4j implementation in the absence of clear guidelines or learning materials.

Despite providing a structured script to define all nodes and relationships, the GitHub repository used to administer the Neo4j instance lacked contextual explanations of the modeling conventions employed. There was no reference manual outlining the standard naming conventions, the meaning of each node label, or the proper structure and classification of relationships. This discrepancy resulted in variations in the additions or interpretations of graph elements by various contributors. Additionally, it made it hard for new developers to confidently contribute, particularly when they weren't sure if certain connections were already represented or if they needed to be made in a different format.

In teams with more detailed graph models, this problem was particularly noticeable. For example, developers frequently voiced doubts about whether their updates complied with current

standards or practices in the VIPOC team's instance, where microservices were represented separately. Teams were left to make their own decisions, which occasionally deviated from the graph's intended logic, in the absence of a common document specifying what should or shouldn't be modeled or how to handle particular architectural patterns (like optional dependencies or conditional logic flows).

Furthermore, the absence of onboarding resources meant that informal assistance was needed to comprehend Neo4j's contribution to the project. Newcomers were forced to rely on direct help from individuals who had spearheaded the implementation or verbal explanations from more seasoned colleagues. When scaling usage across multiple teams or when developers joined mid-project, this method became a bottleneck, even though it had worked in smaller settings or within teams that had championed the tool. In these situations, consistent engagement was hampered by the learning curve for comprehending both the domain-specific structure of the graph and the Cypher language.

Feedback from the interviews also showed that a number of developers were uneasy about dealing directly with the Neo4j instance, not because of the technology but rather because they were unclear about its expectations and limitations. For instance, frequently unanswered questions included "What type of change warrants an update in the graph?" "What if a service is renamed or split?" and "Who is responsible for verifying relationship accuracy?"

Several improvement paths were identified in order to address this challenge. One was the establishment of a living documentation hub that would contain the following and could be incorporated into the company wiki or GitHub repository:

- A visual schema of the Neo4j model
- Definitions and use cases for each label and relationship type
- Versioning practices and rollback procedures
- Example Cypher queries for both reading and writing data
- Update responsibilities and approval workflow for each team

In addition to lessening the need for impromptu justifications, this kind of resource would encourage uniformity, responsibility, and usability for all participating teams. Furthermore, incorporating documentation into sprint planning procedures and onboarding checklists would support the notion that Neo4j is a collaborative platform that needs shared maintenance and comprehension rather than being a standalone visualization tool.

4.5.3 Ensuring Long-Term Engagement and Sustainability

Although Neo4j's initial deployment at Celfocus produced encouraging results and sparked interest from a number of developers, the tool's long-term success hinges on both its technical feasibility and the ongoing involvement of the teams that use it. There are organizational and procedural challenges in maintaining the Neo4j instance's accuracy, timeliness, and seamless integration into development processes as it grows and changes.

The possibility of gradual disengagement was one of the main issues noted in the observations and interviews. A number of team members voiced doubts about whether the work needed to keep

the graph up to date would still seem justified in the long run, particularly if team compositions changed or other operational priorities took precedence. Neo4j graph updates run the risk of becoming erratic or deprioritized in the absence of explicit incentives or enforcement mechanisms, gradually undermining the system's accuracy and credibility.

The manual nature of the update procedure adds to this worry. The Neo4j instance was updated by altering a script that was kept in a GitHub repository, as was covered in earlier chapters. This approach decreased the possibility of unintentional data loss and guaranteed version control, but it also came with a procedural overhead that some developers felt was excessive. Some people felt that writing Cypher code, submitting a pull request, and waiting for approval—even for minor changes—was a hindrance to regular updates, especially during hectic development cycles.

Furthermore, there is still an unequal distribution of ownership for the graph's maintenance. While it was anticipated that each team would assign one or more people to review and approve updates, there were instances in which this duty was not clearly defined or carried out consistently. The graph runs the risk of being abandoned or becoming out of date in the absence of clearly defined roles and monitoring procedures, especially if ownership is not codified or incorporated into sprint procedures. Therefore, Neo4j's viability depends on it being viewed as an essential component of the development process, comparable to testing or documentation, rather than as a side project.

Additionally, a number of developers noted that Neo4j's usefulness was most evident when working across teams or solving problems, like figuring out the cause of a regression or the consequences of a refactor. However, fewer people looked at the graph during more solitary or regular sprints. This suggests that unless there are serious dependency problems, usage may naturally decrease in the absence of deliberate reinforcement or automation.

It was suggested that Neo4j be more directly integrated into current workflows in order to break this trend. These included incorporating Neo4j queries into CI/CD pipelines for impact analysis, automating alerts for out-of-date nodes or missing relationships, and specifying update requirements as part of each user story's "definition of done." By taking these steps, Neo4j would be seen as an essential component of the engineering process rather than an auxiliary tool.

The cultural aspect of sustainability comes last. As previously mentioned, teams were more likely to incorporate Neo4j into their daily operations and reap the benefits of its presence if they saw it as a chance to enhance architectural awareness and coordination. It will be crucial to make sure that this way of thinking is more widely accepted. Strengthening engagement and promoting long-term investment in the tool could be achieved by promoting success stories, facilitating frequent graph reviews, and involving team leads in modeling decisions.

4.6 Summary of Evaluation

The assessment of Neo4j's deployment at Celfocus shows a complex result influenced by practical limitations, cultural dynamics both within and between development teams, and technical advancements. Although it was still in its infancy, the mapping and management of dependencies

using a graph-based database created new ways of visibility and coordination and, in some teams, changed the way dependencies were perceived and expected in routine tasks.

The outcomes were notably favorable from the standpoint of the team that initially implemented Neo4j. The tool reduced deployment risks, promoted more thoughtful change planning, and made it possible to visualize internal and external service interactions more clearly. The importance of having a structured view of system interconnections was highlighted by feedback gathered through forms and interviews, especially in situations where documentation was either out-of-date or lacking. In addition to troubleshooting, developers used the graph to streamline knowledge transfer and onboarding, two processes that were previously characterized by uncertainty and lost time.

However, each team's level of success differed. The benefits derived from the implementation were significantly impacted by varying degrees of engagement, as the team that showed more flexibility and dedicated more time to using and updating the graph also saw more noticeable gains in alignment and problem-solving.

However, there were some restrictions on the rollout. The most significant of these was the difficulty of striking a balance between model granularity and maintenance work. More intricate graphs produced more accurate insights, but they also needed constant attention and manual maintenance. In a similar vein, wider adoption was hindered by the absence of centralized documentation, particularly for developers who were not familiar with Cypher or best practices. Team-specific interpretations started to appear in the absence of well-defined modeling standards, which occasionally made cross-team consistency more difficult.

Developing habits around graph updates and guaranteeing sustained engagement were additional difficulties. Neo4j's applicability grew more dependent on its incorporation into standard development procedures as its novelty wore off. Feedback from interviews revealed a common worry: the graph's utility would gradually decline in the absence of structured reinforcement.

Notwithstanding these challenges, the implementation produced a technical and organizational proof-of-concept that worked. The global instance allowed for systemic visibility, the GitHub-based update mechanism brought governance, and the dedicated team-specific instance showed off the capabilities of high-resolution modeling. When combined, these components establish a standard for the organization's more data-driven, scalable dependency management procedures.

To sum up, Celfocus's Neo4j project was successful in starting the transition to graph-based dependency management. Although not yet fully developed, the foundation for wider adoption has been established, and the knowledge gained—from process integration to structural design—provides a useful road map for further development. The following chapter will examine the implications of these findings for dependency management in complex software systems as well as how they relate to the thesis's larger research questions.

Based on the evidence gathered, we can go back to the original research questions and give informed answers based on the results of the implementation and the feedback we got during the project.

RQ1: How can dependencies between different components in complex software systems be managed to simplify maintenance and increase stability?

Using a graph-based model to manage dependencies, as shown with Neo4j, worked to improve system stability and make ongoing maintenance tasks easier. The model made it easier for developers to see how services and components are connected, both within and between teams. This lowered the risk of unexpected effects during deployments and made it easier to plan changes more carefully. The ability to query the graph for dependency chains made it possible to do impact analysis ahead of time, which is especially helpful when working with teams that are spread out and systems that are not tightly coupled. This visibility replaced random communication and made it less necessary to use old documentation. This directly helped make development cycles more stable and predictable.

RQ2: How can a dynamic and feature-oriented team structure improve the efficiency and quality of software when compared to a traditional, static component-based team development?

The results show that teams that were willing to change the way they worked and be more open to new ideas—especially when it came to how they shared responsibility for updating dependencies—did better. The team that used Neo4j the most was able to solve problems between teams faster, plan feature work more accurately, and onboard new developers more effectively. These changes show that dynamic structures, where teams know how their work affects others and actively contribute to shared knowledge bases like the graph, can make code quality and efficiency better than when teams work alone on specific components. But for this change to last, it needs to be culturally aligned and carefully built into daily tasks.

RQ3: How can evolving software systems balance the constant need for updates with minimizing security risks, downtime, and cost increases in dependency management?

The graph-based model's ability to support controlled evolution in software systems was one of its most important contributions. The GitHub-driven update system, along with versioned scripts and centralized validation, made it possible to make changes to the dependency model often but in a way that could be tracked without losing consistency or reliability. This method lowered the chance of regressions or incompatibilities happening during updates and encouraged the cost-effective use of existing knowledge. The graph's visibility also helped teams better assess risks before deployments, which helped them avoid unnecessary downtime and security holes that could have been caused by untracked dependencies. This method will only work if you stick to your update schedule and make sure the modeling standards are clear so that they don't diverge.

Overall, these answers show how using a well-organized, visual, and collaborative method for managing dependencies can make a real difference in how complex systems are built and kept up

to date. The results not only support the original hypothesis, but they also give us a framework that can be used in other similar situations that are facing similar problems.

Chapter 5

Conclusions and Future Work

This last chapter is a summary of the work done in the dissertation, with a focus on the study of ways to manage dependencies in complex software environments. The research used the real-world example of the VTV project at Celfocus to try to understand and solve the problems that come up when multiple independent teams work on interdependent systems. Here, we look back at the results of the implementation and the feedback we got during validation to see how well the proposed approach worked and what it means for software stability, team collaboration, and development efficiency in general. At the end of the chapter, there is a discussion of future work that could build on what this study has already done.

5.1 Conclusions

The reason for writing this dissertation came from the fact that large software systems are becoming more complicated, especially those that are built and maintained by several independent teams. When coordination, communication, and visibility are low, it becomes harder to manage the dependencies between components in these kinds of situations. These problems were clear in the VTV project at Celfocus. Different teams worked on different parts of a platform that had to change all the time without losing stability or ease of maintenance. This dissertation set out to find structured, long-term solutions to the problems caused by poorly managed dependencies, such as unplanned downtime, integration conflicts, and rising technical debt.

Three main research goals were set to help with this exploration. The first goal was to figure out how to best handle the dependencies between different parts of complicated software systems so that maintenance is easier and the system is more stable. The second goal was to find out if a more flexible and feature-focused team structure, as opposed to a strict, component-based one, could improve the quality and speed of development. Finally, the third goal was to find out how software systems that are always changing could better balance the need for regular updates with the need to keep security risks, operational disruptions, and cost increases that come with bad dependency management to a minimum.

The study met these goals by suggesting and putting into action a graph-based dependency management system that used the Neo4j database. This system made it easy to see and keep track of dependencies at both a high level (across teams) and a lower level (within individual teams). One of the most important things that came out of the study was that using a graph to model dependencies helped teams find possible problems earlier in the development cycle and work together better across organizational boundaries. This made it much easier to plan changes and avoid problems that weren't planned for, which directly led to systems that were more stable and easier to maintain.

At the same time, the study looked at how the structure of a team affects how aware people are of their dependencies and how well they work together. The study didn't require a full switch to feature-oriented teams, but it did show that teams that were more open to thinking across functions and seeing other teams got more out of the solution. This backs up the idea that being able to change structures and having everyone understand the system can lead to better results in complicated situations.

The dissertation also added a lightweight way to govern things by combining graph updates with GitHub version control. This solution made sure that things could be traced and held accountable while still being in line with agile workflows. Developers could suggest changes to the dependency model, and reviewers who were chosen could check those changes before they went live in the system. This method spread the responsibility for keeping the model up to date among the teams, which lowered the risk of information loss.

The dissertation's most important contributions include making a general instance of Neo4j to capture dependencies across the system, making a high-resolution instance for one team's internal use, and adding update mechanisms that work with agile delivery processes. These changes are more than just technical—they show a new way for Celfocus to handle documentation, impact analysis, and team coordination that is both scalable and long-lasting.

In short, this dissertation showed that graph-based modeling can be a good way to handle dependencies in big software systems with many teams. It showed how this kind of solution can work with what is already in place and also push for changes in culture and organization that lead to better software quality and lower operational risk.

5.2 Future Work

This dissertation set the stage for graph-based dependency management, but it needs a few more improvements to make it work better and have a bigger effect over time. Documentation is one area that needs to be improved right away. Making onboarding guides that are easy to find, clear modeling standards, and an up-to-date knowledge base would help both new and current team members better understand and contribute to the model.

Automation is also a great chance. The graph model needs to be updated by hand right now, but it could be better if it could work with static code analysis tools and version control systems to

automatically find and suggest changes to dependencies. This would cut down on mistakes made by people and make sure the model stays up to date with changes to the system.

More work needs to be done to make scalable granularity possible. A system that can show both high-level and detailed views, depending on the situation, would be easier to use without putting too much maintenance work on teams. Layered or modular graph representations could help with this.

Finally, formally adding graph updates to agile workflows—by including them in the definition of done or setting up review points—could make them more popular. Long-term studies on how the model affects deployment success, onboarding, and communication efficiency would also help us understand how useful and long-lasting it is.

By focusing on these areas, future efforts can turn the initial implementation into a strong and automated practice that meets the changing needs of large-scale software development.

References

- [1] Ritu Arora, Sanjay Goel, and R. K. Mittal. Using dependency graphs to support collaboration over GitHub: The neo4j graph database approach. In *2016 Ninth International Conference on Contemporary Computing (IC3)*, pages 1–7. IEEE, aug 2016.
- [2] Meriem Belguidoum and Fabien Dagnat. Dependency management in software component deployment. *Electronic Notes in Theoretical Computer Science*, 182:17–32, 2007.
- [3] Hojin Cho, Kwanwoo Lee, and Kyo C. Kang. Feature relation and dependency management: An aspect-oriented approach. In *2008 12th International Software Product Line Conference*, pages 3–11. IEEE, sep 2008.
- [4] Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *Queue*, 16(4):44–65, aug 2018.
- [5] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: allegrograph, arangodb, infinitedb, neo4j, and orientdb. In *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, pages 373–380. SCITEPRESS - Science and Technology Publications, jul 2018.
- [6] Joao Eduardo Ferreira, Pedro Losco Takecian, Leonardo Tadashi Kamaura, Bruno Padilha, and Calton Pu. Dependency management with WED-flow techniques and tools: A case study. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 379–388. IEEE, oct 2017.
- [7] João E. Ferreira, Osvaldo K. Takai, Simon Malkowski, and Calton Pu. Reducing exception handling complexity in business process modeling and implementation: The WED-flow approach. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *On the move to meaningful internet systems: OTM 2010*, volume 6426 of *Lecture notes in computer science*, pages 150–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [8] José Guia, Valéria Gonçalves Soares, and Jorge Bernardino. Graph databases: neo4j analysis. pages 351–356. SCITEPRESS - Science and Technology Publications, apr 2017.
- [9] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 101–104, New York, NY, USA, may 2018. ACM.
- [10] Chi-Yu Huang, Alan Holt, John Monk, and Kai Cheng. The application of dependency management in an integrated manufacturing network framework. *The International Journal of Advanced Manufacturing Technology*, 33(3-4):354–364, jul 2007.

- [11] John Hunt and Chris Loftus. The fault tracker J2EE case study. In *Guide to J2EE: enterprise java*, pages 641–657. Springer London, London, 2003.
- [12] John Jacobson, Mike Bentley, Cayden Lund, Ganesh Gopalakrishnan, Ignacio Laguna, and Gregory L. Lee. Float: framework for workflow analysis, visualization and transformation. In *2022 First International Workshop on Visualization in Testing of Hardware, Software, and Manufacturing (TestVis)*, pages 22–27. IEEE, oct 2022.
- [13] Thomas Karanikiotis, Michail D. Papamichail, and Andreas L. Symeonidis. Analyzing static analysis metric trends towards early identification of non-maintainable software components. *Sustainability*, 13(22):12848, nov 2021.
- [14] Chris Kemper. Querying data in neo4j with cypher. In *Beginning Neo4j*, pages 83–101. Apress, Berkeley, CA, 2015.
- [15] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. PDGraph: A large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173. IEEE, jun 2021.
- [16] Rodrigo Alves Lima, Calton Pu, Bruno Padilha, Pedro L. Takecian, Leonardo T. Kamaura, and João E. Ferreira. Modeling time-critical processes with WED-flow. In Hai Jin, Qingyang Wang, and Liang-Jie Zhang, editors, *Web services – ICWS 2018*, volume 10966 of *Lecture notes in computer science*, pages 34–48. Springer International Publishing, Cham, 2018.
- [17] Qian Ma, Ying Li, Kewei Sun, and Liang Liu. Model-based dependency management for migrating service hosting environment. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 356–363. IEEE, jul 2007.
- [18] Yannis Marketakis and Yannis Tzitzikas. Dependency management for digital preservation using semantic web technologies. *International Journal on Digital Libraries*, 10(4):159–177, dec 2009.
- [19] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531, New York, NY, USA, oct 2020. ACM.
- [20] Elke Pulvermüller, Andreas Speck, and James O. Coplien. A version model for aspect dependency management. In Jan Bosch, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, editors, *Generative and Component-Based Software Engineering*, volume 2186 of *Lecture notes in computer science*, pages 70–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
- [21] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *ACM SIGPLAN Notices*, 40(10):167–176, oct 2005.
- [22] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. Efficiently trimming the fat: Streamlining software dependencies with java reflection and dependency analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, New York, NY, USA, apr 2024. ACM.

- [23] Viktoria Stray, Nils Brede Moe, and Andreas Aasheim. Dependency management in large-scale agile: A case study of DevOps teams. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, Proceedings of the annual hawaii international conference on system sciences. Hawaii International Conference on System Sciences, 2019.
- [24] Richard Vidgen and Xiaofeng Wang. Coevolving systems and the organization of agile software development. *Information Systems Research*, 20(3):355–376, sep 2009.
- [25] Pei Wang, Jingiu Yang, Lin Tan, Robert Kroeger, and J. David Morgenthaler. Generating precise dependencies for large software. In *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 47–50. IEEE, may 2013.
- [26] Rongxin Wu, Minglei Chen, Chengpeng Wang, Gang Fan, Jiguang Qiu, and Charles Zhang. Accelerating build dependency error detection via virtual build. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, New York, NY, USA, oct 2022. ACM.