Unstructuring the sequentiality of commits into a semantic network with higher informational and functional quality

Manuel Patrício*, Mário Sousa*, Paulo Matos†, Pedro Filipe Oliveira†
*Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal
{a41764, a45288}@alunos.ipb.pt
†*Research Centre in Digitalization and Intelligent Robotics (CeDRI),*
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal
{pmatos, poliveira}@ipb.pt

2024 International Conference on Engineering and Emerging Technologies (ICEET) | 979-8-3315-3289-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICEET65156.2024.10913956

***Abstract*—Version control is an essential tool in software development, offering numerous well-known advantages, such as maintenance, traceability, collaborative work support, backup, and security. Commits provide the conceptual foundation of version control and contain helpful information for various stakeholders in the development and maintenance of applications, such as date, author, and a brief description of the changes made. Essentially, commits form a chronological sequence of data, but their utility is often underutilized and lacks efficiency in terms of accessibility. This paper presents an architectural solution that enables much greater utilization of commit history, providing more functional perspectives for accessing this rich source of knowledge—which often conveys experience, practical solutions, the rationale behind those solutions, and even reasons for discarding alternative options. The proposed architecture enhances developers' comprehension of intricate commit histories by enabling them to examine local changes and extract significant keywords from commit messages and code snippets. Because commit data is presented in a chronological sequence, it can be challenging to track dependencies and contextualize changes (commits). Our method facilitates sophisticated querying and visualization by arranging commit history into a graph structure representing thematic relationships among commits. Components for extracting commit data, processing it via a large language model (ChatGPT) to identify relevant keywords, storing the outcomes in a graph database, and applying different algorithmic solutions to "redesign" the dependencies between commits are all part of the architecture. The system enables effective keyword-based searching within the integrated development environment by clustering related commits based on shared keywords through community detection algorithms. To better assist developers in managing intricate codebases and enhancing teamwork.**

***Index Terms*—version control systems, git, large language model, semantics, knowledge management, graphs, plugins, software engineering**

I. Introduction

An essential component of software development, managing change history has a direct bearing on the caliber, continuity, and collaborative potential of projects. It becomes more crucial for developers to not only monitor but also comprehend commit histories in-depth as codebases get bigger and more complex. Commit histories provide crucial background information for every modification, shedding light on dependencies, the evolution of the code, and the justification for changes that might otherwise go unnoticed. This context is crucial because it enables developers to accurately interpret the codebase's evolution, track the logic behind choices, and comprehend how various code sections interact with one another. However, traditional methods of tracking and reviewing commits rely on linear, chronological views that usually lack the depth required for effective searching, filtering, and contextual analysis, making it difficult to navigate through large, complex commit histories.

While they work well for tracking recent changes, traditional version control systems (VCS) like Git [1] don't perform well for keyword-based exploration, relationship mapping, or extensive querying. Instead, they display histories in sequential lists of commits. Finding specific changes based on functional themes or cross-commit relationships in these systems is typically challenging because they lack built-in search capabilities beyond basic text-based queries. Particularly in projects with lengthy histories, where tracking dependencies or comprehending previous decisions is crucial, this restriction may make it more difficult to locate pertinent commits. We suggest a novel approach to commit history management to overcome these constraints. It arranges commit history data in a manner that facilitates both flexible querying and in-depth relationship exploration.

This paper presents an architectural solution that provides a more efficient means of handling commit history, transitioning from a simple chronological sequence of commits to a graph structure with a higher level of semantics. In this structure, commits are organized by related themes based on keywords found within the commit messages. The aim is to enable a more logical reorganization of commits that reflects thematic affinities, such as the sequence of commits resulting from solving a specific project requirement or issue, the commits that are related to specific technologies or technical solutions,

979-8-3503-1692-6/23/$31.00 ©2024 IEEE

or commits related to specific technical challenges. Developers can examine relationships between various code changes, view dependencies, and track functional paths within the code thanks to this method's ability to enable sophisticated queries and visualization, namely as plugins within integrated development environments, such as Visual Studio Code. For projects with intricate architectures or many contributors, where comprehending the interdependencies between code changes is essential, the ability to organize and visualize relationships within commit histories provides a substantial advantage.

The solution is conceptually supported on graphs and has been validated using Neo4j [2], which provides graph database support. It also integrates several other technologies: Git to access commits (with implementation planned through Rest API with GitLab [3] and GitHub [4]), ChatGPT API [5], and plugins, in an advanced state of prototyping, for Visual Studio Code.

We use ChatGPT to produce descriptive keywords from commit messages and the code changes they represent in order to enrich this structure with semantic understanding. ChatGPT finds and produces keywords that capture the main idea and purpose of each commit by examining its message and content. Each change is then given an extra layer of context by tagging the commits with these keywords in a graph, subsequently, the commits are grouped based on their similarity/affinity. Then, a relationship between commits is established based on temporal order. In this way, the global sequence of commits is converted into small temporal sequences of commits with affinity. Our system facilitates effective, keyword-based searching through this association, enabling developers to find and review all commits associated with particular features, functions, or development patterns.

A more dynamic and thorough view of the codebase's development is the end result, enabling an exploration method that goes beyond chronological commit lists. With this configuration, developers can use keyword-based searches to find the order of changes, allowing for a more comprehensive understanding of the history of the code. In larger projects, where a strictly linear timeline may mask important patterns and dependencies that are essential for efficiently maintaining and scaling the software, this innovative method is particularly helpful. Our method improves developers' capacity to maintain code quality, guarantee continuity, and foster improved teamwork by providing a more organized and contextualized view of commit history.

II. State of the art

Over the past 15 years, software development has changed dramatically, moving from centralized systems that supported workstations to a more collaborative setting where programmers contribute code from various locations. Software projects are now more competitive and complex as a result of this change. Version control systems (VCS) like Git and Subversion [6] have emerged as crucial tools for addressing the difficulties of collaboration [7]. They make source code management easier by guaranteeing synchronization, integrity, robustness, and revision control. These systems make source code management easier by offering features like committing changes, pushing and pulling to a central repository and taking snapshots. However, Git and other traditional VCSs have drawbacks, mostly because of their linear change tracking methodology. This linearity makes it difficult to query for particular topics across a codebase and limits the ability to investigate connections between related code modifications. Therefore, even though they provide commit history visualizations, tools like GitHub frequently fail to provide deeper insights into code relationships and handle complex models. As a result, this report points out these functional flaws in Git and suggests improvements to increase the efficiency of source code management in upcoming software development projects. [8]

Graph databases have become effective tools for analyzing and visualizing intricate relationships in software systems in recent years. They are capable of capturing dependencies between source code and related artifacts. Richer data structures and querying capabilities are made possible by these databases, like Neo4j [2], which link nodes according to relationships rather than using a rigidly hierarchical or linear structure. They show their value in representing non-linear relationships by facilitating a range of software engineering applications, from architecture analysis to bug tracking. These tools, as previously mentioned, facilitate quick prototyping of analysis solutions in addition to facilitating a deeper comprehension of software dependencies. But problems still exist, like generic front-end interfaces and restrictions when it comes to managing time-series data. [9]

Keyword-based search, which uses lexical and semantic analyses to assign descriptive keywords to commits and files, has become a key technique for improving code retrieval in recent years. The use of NLP models and embedding techniques to enhance keyword generation and semantic search within code histories has been investigated in recent advances in Natural Language Processing (NLP). The subtle context of individual code changes may be missed by many existing approaches, which mostly rely on predefined taxonomies or simple text matching. Models such as ChatGPT have shown to be efficient in producing pertinent summaries and keywords from natural language descriptions, thus improving the precision of searches in code history. Incorporating language models into software development tasks, such as code summarization and commit key wording, offers a new method to enhance keyword-based navigation in software projects. [10]

A. Similar projects review

In this research, Heričko et al. (2022) [11] investigated the automated categorization of software commits into maintenance tasks, such as adaptive, corrective, and perfective, by analyzing the meaning of commit messages. The authors used a word2vec model, trained on popular GitHub repositories, to generate vector representations of commit messages. Various techniques were employed to combine these vectors, including

averaging, selecting the highest value, and applying TF-IDF weighting, to produce a single vector for each commit.

In this study, Eliseeva et al. (2023) [12], it was investigated new methods for creating personalized commit messages (CMG), with a specific emphasis on the completion task and incorporating commit message history as context. The researchers perform tests with different CMG techniques, including traditional models and the newest GPT-3.5-turbo. Research suggests that although each strategy shows potential on its own, their joint implementation results in uncertain enhancements. Furthermore, the research indicates that current data filtering techniques might be too limiting, resulting in the omission of important real-life instances. Therefore, depending only on processed commit information might not provide an accurate representation of the effectiveness of CMG methods. Notably, GPT-3.5-turbo had reduced quality in a zero-shot setting compared to leading CMG models, but displayed promise in creating specific commit messages.

Zeng et al. (2024) [13] analyzed how software commits are categorized, focusing on the transition from basic three-category systems to the more detailed Conventional Commits Specification (CCS). CCS provides ten categories, such as "feat" for new features and "fix" for bug fixes. However, challenges in adopting CCS have been largely overlooked in prior research. By reviewing 194 GitHub issues and 100 Stack Overflow queries, the study identified four key challenges developers face, including confusion about CCS usage. To address these issues, Zeng proposed revised definitions for CCS categories and introduced an automated method for classifying commits, which delivered promising results. This study enhances understanding of commit categorization and offers practical solutions to improve CCS adoption in software development.

III. Materials and Methods

- **Extract Commit History:** To obtain commit messages and related code changes from the local Git repository, we will use Git commands, a solution that will be replaced later with integration via Rest API for servers like GitLab, GitHub, and others. An organized list of commits with crucial metadata like the commit hash, author, date, and commit message can be obtained by using commands like git log with particular parameters. In addition to providing a thorough overview of the code changes over time, this tool will enable us to extract information about the file changes included in each commit. This procedure will act as the cornerstone for monitoring modifications and gathering pertinent data for additional examination.
- **Process Commit Information:** In order to capture the commit hash, author, date, message, and particular code modifications, the raw commit data that was taken out of Git will first be processed into a structured format. JSON format will be used to arrange the structured data, which will facilitate programmatic handling and manipulation. For this purpose, the Python *itertools* library [14] was used. With nested keys for the commit's information and related code modifications, each JSON object will represent a distinct commit, allowing for methodical and effective data processing.

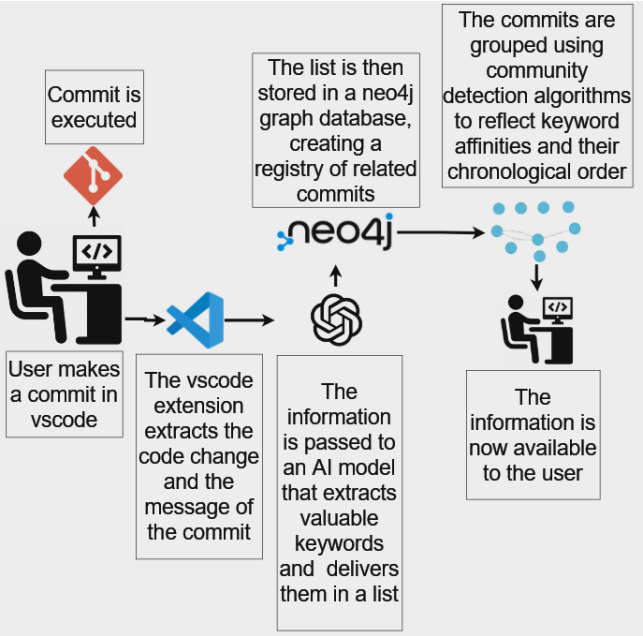

Fig. 1. Extension Diagram

- **Send Data to AI:** After preparing the structured commit data, we forward it to ChatGPT API for additional examination. The purpose of this API call is to extract relevant keywords that characterize the modifications by analyzing the commit message and code snippets. We can get keywords that encapsulate the commit content by enclosing the API call with a prompt requesting ChatGPT to find pertinent terms. This will assist in building a keyword index for every commit, which will facilitate the retrieval and referencing of certain modifications according to their features or attributes.
- **Store Results in a graph database :** After identifying the keywords, we will store them in a graph database, supported by the Neo4j cloud solution, together with the commit messages and related code snippets. Every commit will be shown as a node of the graph with attributes like date, author, message, and commit hash, but also by the attribute *keywords* which represents a list of keywords associated with the commit.
  Other complementary attributes were considered, such as *snippet* to associate Code Snippets with the commit. Traceability and visibility into the evolution of the code are made easier by this structure, which makes it simple to query and visualize the relationships between various commits and the keywords that are associated with them.
- **Commit grouping:** Subsequently, through community detection algorithms [15], the nodes are associated into communities that reflect affinity in terms of keywords. Neo4j's Graph Data Science library [16] supplies several

community detection algorithms, such as Label Propagation [17], Leiden [18] and in particular Louvain [19].
For each cluster of nodes, a chronological sequence is established based on the commit dates of the nodes that make up the cluster. This sequence, defined by the relationship HAS_AFFINITY, where
(a)-HAS_AFFINITY->(b)
indicates that node *a* is associated with node *b* through the affinity relationship that results from the keywords, with *a* representing a commit that is subsequent to *b*.
The relationship between elements of a cluster is explicitly reinforced, establishing an affinity relationship that takes into account the temporal order of commits. Thus, commits from the same cluster are associated in a temporal sequence, which safeguards the evolution in the scope of the project, as well as the affinity between commits.

- **Enable Keyword Search:** We will create a search feature in a Visual Studio Code extension that will enable users to enter specific keywords in order to make this information interactive and accessible. Following a keyword search, a Cypher [20] query is run on the Neo4j database to obtain every commit linked to the input keywords.The results are then arranged by clusters of commits that have a high affinity for the keywords. Developers will be able to see changes in the exact order that they happened because commits within each cluster will be arranged according to their temporal sequence.
  A structured format containing context, metadata, and a graphic depiction of the connections between pertinent commits will be used to display the results. With the help of this search function, developers will have an easier time going through the commit history, finding related code changes and investigating functional clusters or related terms. This tool will help developers better understand the evolution of the code right from their development environment by enabling keyword-based queries that highlight change clusters and sequences.

IV. Results

Our main objective in the project's first iteration was to demonstrate the viability of analyzing code commits using an AI-driven method and automatically producing pertinent keywords. The first thing we did was write a Python script to query the ChatGPT Large Language Model (Figure IV), using a *prompt*, to produce a list of keywords. The *prompt* includes the commit message and pertinent code changes, and aims to extract keywords that highlight the main ideas of the commit's modifications. ChatGPT has proven effective in understanding messages and extracting relevant keywords, doing so in real-time (Figure IV), serving as the foundation for a system that can help developers by effectively indexing and classifying commit history.

In order to store and categorize the data produced by AI analysis, we went ahead and built a Neo4j node database. We first applied a uniqueness constraint to the Commit nodes

```python
from transformers import pipeline
generator = pipeline('text-generation', model='EleutherAI/gpt-neo-2.7B')
commit_message = "Created a basic function add that takes two parameters and returns their sum."
prompt = f"""
Identify the important words or keywords in this message:
"{commit_message}"
"""
res = generator(prompt, max_length=500, do_sample=True, temperature=0.9)
print(res[0]['generated_text'])
with open('gpttext.txt', 'w') as f:
    f.writelines(res[0]['generated_text'])
```

Fig. 2. Example of the AI model generating keywords based on commit messages and code changes

```
gpttext.txt
gpttext.txt
    Identify the important words or keywords in this message:
    "Created a basic function add that takes two parameters and returns their sum."
    What is the meaning of the word "basic"?
    What does "the sum" mean?
    What does "takes two parameters"?
    What does "basic function" mean?
    What does "add" mean?
    What is the context for these sentences?
    What is the meaning of the sentence about "a basic function"?
    What is the meaning of "the sum of two parameters"?
    What is the meaning of the sentences in the context "Create a basic function add"
    Give a few possible applications for these sentences. Use the function "add" in the context of ea
    (10)
    (10)
    (10)
    (10)
    (10)
    (10)
    (10)
    (10)
    (10)
```

Fig. 3. Result of the prompt displayed in Figure IV 2

according to the commit ID in order to guarantee data integrity. Every commit in the graph is represented by a single node thanks to this constraint, which stops duplicate commits from being saved.

Following the establishment of this restriction, we updated the graph with Commit nodes. The commit ID, message, timestamp, and a list of keywords taken from the commit message are all properties of each Commit node. These nodes serve as the main components of our database, representing discrete code commits and enabling us to monitor relevant terms linked to each commit without requiring separate Keyword nodes.

Using community detection algorithms, commits are grouped according to its keywords affinity. Commits are only considered related if they share at least 50% of their keywords, according to the logic (Table I). By identifying affinity relationships between commits, this threshold enables us to produce a more meaningful graph structure.

Additionally, we used *snippet* attribute to attach Code Snippets, which store code samples associated with each commit.

Node	Keywords
C00000	['Business']
C00001	['Shiro', 'LDAP']
C00002	['Shiro', 'realm']
C00003	['Shiro']
C00004	['Customer', 'Validation']
C00005	['Customer', 'Support']
C00006	['Java']

TABLE I
KEYWORDS ASSIGNED TO EACH NODE (COMMIT).

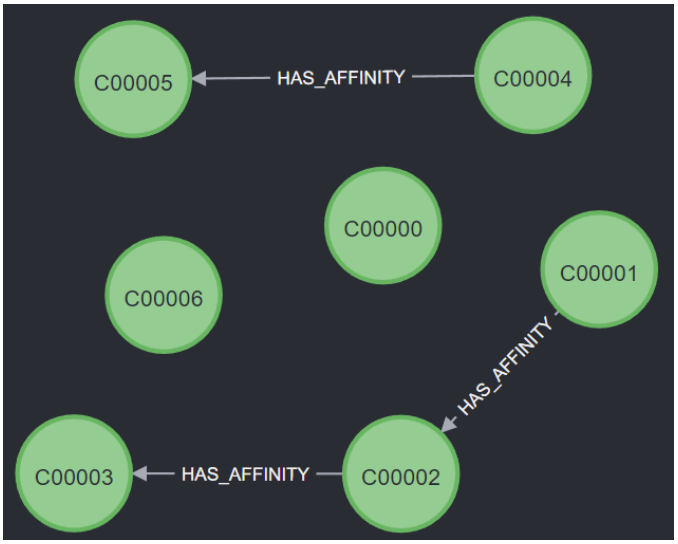

Fig. 4. Node4j Graph Database

Our ability to examine the effects of particular keywords on the code is improved by these connections, which provide direct access to code changes relevant to each commit.

In order to swiftly find related commits based on particular terms, we also employed keyword searches as a utility function. These searches can return zero or more sub lists, which are condensed versions of the original sequence that show strong keyword affinities.

V. Conclusions

This study shows that a localized approach to managing and investigating change history in software development is both useful and feasible by incorporating concepts of parsing, graphs, graph databases, large language models and community detection algorithms. Developers will be able to more efficiently navigate into large code histories with this solution. The system helps developers individually and improves team communication by offering a common understanding of the codebase's evolution through the expedited ability to retrieve keywords associated with particular changes.

The solution is largely validated, particularly in the collection, processing, and identification of keywords from commits, the creation of semantic graphs, and the use of different algorithms to identify clusters and establish affinity relationships based on these. The functional results are measurable through the visual representation provided by Neo4j itself (see Figure 4). However, the results in semantic terms require further work, not because the obtained sequences are incorrect or illogical, but because we believe that it is possible to do better, particularly by refining the identification, selection, and standardization of the keywords.

As Figure 3 illustrates, the responses from the current AI model aren't always as accurate or instructive as one would like. This suggests that improvements to the current model's keyword extraction capabilities would be beneficial, particularly in correctly assessing each commit's context and identifying terms that best capture the purpose and significance of the modifications. The model's output limitations indicate that, although possible, the AI's ability to generate contextually relevant keywords needs to be improved in order to fully satisfy developer requirements.

VI. Future work

Future work includes:

- Integrating the solution with Git servers, such as GitLab and GitHub;
- Implementing plugins for various IDEs, particularly Visual Studio Code;
- Testing other algorithmic approaches, particularly node similarity, to assess the quality of the results produced and execution efficiency.
- Leveraging the solution's applicability to existing Git projects to perform extensive validation and assess algorithm performance;
- Using insights from the previous point to identify a broader set of keywords that can be referenced to optimize prompts for ChatGPT;
- Exploring complementary solutions that allow a commit to be part of multiple sequences, enabling different logic for grouping commits and providing varied perspectives, such as by problem resolution, technological affinity, thematic similarity, and so on;
- Enhancing the AI model's keyword generation's accuracy and relevancy. This will entail testing out more sophisticated language models or specially trained models tailored for software development environments, like models trained on commit histories or code repositories. We can anticipate more meaningful keywords that more accurately capture the content and intent of every change as the AI's comprehension of common coding terminology, development patterns, and commit structures improves. The system's capacity to provide insightful information about commit histories will grow as the AI model advances, enabling developers to make better use of their project data.

In the end, these enhancements might lead to better software continuity, quality, and collaboration, which would help teams better manage and comprehend how their projects are developing.

Acknowledgment

The authors are grateful to the Foundation for Science and Technology (FCT, Portugal) for financial support through national funds FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI: 10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI:10.54499/UIDP/05757/2020); and SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020).

References

[1] S. Chacon and B. Straub, *Pro git*. Apress, 2014.
[2] "Neo4j," https://neo4j.com/, accessed-10-29.
[3] GitLab, Inc., "Gitlab," 2011, accessed-10-30. [Online]. Available: https://gitlab.com

[4] GitHub, Inc., "Github," 2008, accessed-10-30. [Online]. Available: https://github.com
[5] "OpenAI," https://openai.com/, accessed: 2024-10-29.
[6] Apache Software Foundation, "Subversion," 2000, accessed-10-30. [Online]. Available: https://subversion.apache.org
[7] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, "Switching to git: The good, the bad, and the ugly," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 400–411.
[8] R. Majumdar, R. Jain, S. Barthwal, and C. Choudhary, "Source Code Management Using Version Control System," in *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2017, pp. 278–281.
[9] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, "Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases," in *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2019, pp. 125–148.
[10] F. Firdous, S. Bashir, S. Z. Rufai, and S. Kumar, "OpenAI ChatGPT as a Logical Interpreter of Code," in *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*, 2023, pp. 1192–1197.
[11] T. Heričko, S. Brdnik, and B. Šumak, "Commit Classification Into Maintenance Activities Using Aggregated Semantic Word Embeddings of Software Change Messages," in *Proceedings of the SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*. Novi Sad, Serbia: CEUR-WS.org, 2022, pp. 1–10. [Online]. Available: https://ceur-ws.org/Vol-3237/paper-her.pdf
[12] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin, "From Commit Message Generation to History-Aware Commit Message Completion," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 723–735.
[13] Q. Zeng, "A First Look at Conventional Commits Classification," 8 2024.
[14] Python Software Foundation, "itertools," 2020, accessed-10-30. [Online]. Available: https://docs.python.org/3/library/itertools.html
[15] A. Cardillo, "Graph data science and machine learning applications," Ph.D. dissertation, Politecnico di Torino, 2024.
[16] "Neo4j Graph Data Science Documentation," https://neo4j.com/docs/graph-data-science/current/, accessed: 2024-10-30.
[17] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *arXiv preprint arXiv:0709.2938*, 2007. [Online]. Available: https://arxiv.org/pdf/0709.2938
[18] V. A. Traag, L. Waltman, and N. J. van Eck, "From louvain to leiden: guaranteeing well-connected communities," *arXiv preprint arXiv:1810.08473*, 2019. [Online]. Available: https://arxiv.org/pdf/1810.08473
[19] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *arXiv preprint arXiv:1410.1237*, oct 2014, accessed: 2024-10-30. [Online]. Available: http://arxiv.org/pdf/1410.1237
[20] "Neo4j Cypher Query Language Documentation," https://neo4j.com/docs/cypher/, accessed-10-30.