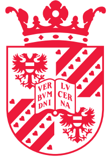

university of groningen

faculty of science and engineering

# NeoRepro: A Tool For Creating Replication Packages for Mining Software Repository Research using a Graph Database

Andrew Rutherfoord

university of groningen

faculty of science and engineering

University of Groningen

# NeoRepro: A Tool For Creating Replication Packages for Mining Software Repository Research using a Graph Database

**Bachelor's Thesis**

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at the University of Groningen under the supervision of
Prof. Vasilios Andrikopoulos (University of Groningen)
and
Dr. Daniel Feitosa (University of Groningen)

**Andrew Rutherfoord (s4667646)**

August 2, 2024

**Abstract**

In this thesis we present NeoRepro, an application which can be used to conduct a Mining Software Repositories (MSR) study and create a replication package for the study, which can easily be distributed alongside the research paper for replication of the results. These packages encompass all source data and scripts used in the study. The NeoRepro application is centered around the Neo4j graph database which is used to store all gathered software repository data and facilitate querying of the collected data. Using a centralized data storage solution enables the data to be queried quickly with a standardized query language without revisiting the original data sources. To enhance our tool, we conducted a thorough analysis of existing tools in this domain, identifying key areas for improvement which NeoRepro addresses. We demonstrate the application's utility by creating and distributing a replication package for an existing MSR study. Our evaluation confirms that NeoRepro is highly effective in producing replication packages that facilitate the accurate replication of research results.

# Contents

**1 Introduction** **3**

**2 Related Work** **5**
2.1 Code Libraries for MSR . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Toolsets for MSR . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Database Systems for MSR . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Design decisions for MSR tools . . . . . . . . . . . . . . . . . . . 9

**3 Requirements** **10**
3.1 Personas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.2 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

**4 Architecture** **13**
4.1 Drilling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.2 Analyzing the mined Data . . . . . . . . . . . . . . . . . . . . . . 19
4.3 Replication Package . . . . . . . . . . . . . . . . . . . . . . . . . 21

**5 Evaluation** **22**
5.1 Case Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.2 Comparative Analysis . . . . . . . . . . . . . . . . . . . . . . . . 28

**6 Conclusion** **30**

**A Additional Data** **37**

# List of Figures

3.1 Pohl et al Requirements template outline [12] . . . . . . . . . . . 10

4.1 Architecture of NeoRepro. . . . . . . . . . . . . . . . . . . . . . . . 14
4.2 NeoRepro frontend home page displaying README and menu. 15
4.3 The drill configuration editor page of the NeoRepro frontend. 16
4.4 UML Class Diagram of Driller Worker. . . . . . . . . . . . . . . . 17
4.5 Frontend interface where the user can see the current status of each drill job. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.6 Structure of the Graph Database storing Git repositories. Inspired by GraphRepo [20] . . . . . . . . . . . . . . . . . . . . . . 19
4.7 NeoRepro query interface . . . . . . . . . . . . . . . . . . . . . . 20
4.8 NeoRepro database management and backup interface. . . . . . 20

5.1 Steps of data collection and analysis that were used in Mining Cost Awareness Study [8] . . . . . . . . . . . . . . . . . . . . . . 23
5.2 Structure of the Graph Database for Mining Cost Awareness replication. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

# 1 | Introduction

Mining Software Repositories (MSR) is a process of extracting data from version control code repositories in order to "uncover actionable information about software systems" [24]. MSR is a growing field of research within computing science as it allows data to be gathered about the code that was written under the constraints of the real world, such as time constraints and technical debt. This allows for research to be conducted into areas such as technical debt [2], software risk assessment [4], and cost awareness [8]. Research in this field is often done by extracting data from code repositories stored in the Git format. In 2023, there were 284 million open source repositories on GitHub [22] alone, which provides a vast number of software projects in a variety of domains of computing science which can be analysed using MSR.

During an MSR study, data is collected and analyzed to answer a specific research question. Upon completion of the study and the writing of a research paper, a replication package is often released alongside the paper in order to abide by the ACM Artifact Review standards. According the the ACM, a replication package (or artifact) should contain anything "used as part of the study or generated by the experiment itself" [1]. Releasing a replication package that contains all the source data and scripts used to gather the data, enables the independent replication of the results of a study. Ideally it should facilitate the replication of a study from start to finish without needing any additional information. Unfortunately, creating a replication package can be challenging, since the research process typically focuses on obtaining data to support the research question rather than on creating well-documented datasets and scripts for external use.

To address this challenge, a system that requires minimal configuration and saves all data during the study can significantly reduce the effort needed to create a replication package. NeoRepro [1] is designed with this in mind, allowing researchers to conduct an entire MSR research project within it, including repository drilling and data analysis. By consolidating all these functionalities into a single system and enabling researchers to save their work as they progress, NeoRepro simplifies the creation of comprehensive and accessible replication packages.

MSR studies are usually conducted with a research question and hypothe-

[1]NeoRepro: https://github.com/AndrewRutherfoord/NeoRepro-MSR-tool

sis predefined before data is found, which will be analysed to answer the predefined question(s). In previous studies [3, 8], custom scripts were built which extracted the data needed to answer their pre-defined research question. In order to make new observations on a mined dataset the scripts have to be modified and then re-run on the dataset. This leaves little room for exploratory studies since updating the scripts and re-processing the data from a large set of software repositories is a time consuming task. Therefore NeoRepro provides a simple configuration interface which can extract information from a large number of repositories quickly with a large amount of flexibility, but without the need to write custom scripts. All of the extracted data is loaded into a Neo4j [2] graph database where it can be efficiently queried using Neo4j's Cypher query language [3] without needing to traverse the source repository data again.

Once the entire study is complete, the drill configuration, dataset queries, and multiple dataset images (from different stages in the study) are all contained within a single Git repository alongside all of the NeoRepro tooling. This comprehensive repository can be easily distributed, enabling other researchers to access and replicate the study with minimal effort. By embedding query scripts and detailed instructions within the replication package, NeoRepro improves the reproducibility and transparency of research findings, facilitating peer validation and further research based on the original study. Ultimately, NeoRepro allows researchers to focus on research while ensuring that their work is easily reproducible and verifiable.

In the our research and creation of this tool we aimed to answer two research questions:

- How does NeoRepro improve the accuracy and ease of replicating an MSR study's results?
- What are the key advantages of using the Neo4j graph database for data storage and retrieval in the context of MSR studies?

To answer these research questions this paper will be in the following structure. In section 2 we will present a number of related tools and how NeoRepro compares to each. In section 3 we will outline the requirements that NeoRepro fulfills. Section 4 we outlines the architecture and technologies used in the creation of NeoRepro and exhibits all of the features that NeoRepro offers. In section 5 we evaluate the effectiveness of our tool by performing a case study in which we create a replication package for an existing study, as well as addressing the shortcomings and advantages of NeoRepro. Finally, in section 6 we conclude our analysis with some final thoughts and future work.

[2]Neo4j Graph Database: https://neo4j.com/

[3]Neo4j Cypher Query Language: https://neo4j.com/docs/cypher-manual/current/introduction/

# 2 | Related Work

To facilitate MSR research, a number of tools have been created which allow researchers to easily retrieve information from a large number of Git repositories. MSR research projects are often done with bespoke scripts that use code libraries such as GitPython[1] and JGit[2]. These libraries provide an interface that facilitates interaction with a Git repository with code, allowing for the construction of tools which can extract data from repositories for a given study. However, using libraries to create bespoke scripts to mine data from version control repositories for each new study can be time-consuming.

## 2.1 Code Libraries for MSR

Since the process of gathering data from repositories is often very similar across different MSR research, software libraries like PyDriller[3] have been created specifically for MSR research and facilitate the extraction of repository information, alongside a number of common metrics which are used in MSR. PyDriller is an abstraction ontop of GitPython but provides a simpler interface for extracting data from Git repositories. In the research conducted by the PyDriller team, their tool was found to require fewer lines of code to achieve the same outcome when compared to a tool created using GitPython alone [21]. Since pydriller is a Python library, it provides a researchers with a lot of flexibility in what information is extracted from a repository. However, PyDriller "does not include a CLI for batch mode processing," [15] so for each study a script needs to be written that can extract the necessary data. Since PyDriller is widely used for MSR research and already provides the repository drilling functionality, we decided to use it in the implementation of our driller workers for NeoRepro.

## 2.2 Toolsets for MSR

Although PyDriller provides significant flexibility due to it's implementation as a Python library, it necessitates writing a custom script for data extraction. This outlines the need for unified tools that provide functionality out of the

[1]GitPython: https://github.com/gitpython-developers/GitPython
[2]JGit: https://www.eclipse.org/jgit/
[3]PyDriller: https://github.com/ishepard/pydriller

box. Multiple toolsets which provide data extraction and analysis functionality from software development repositories have been created. GrimoireLab [6] is one such toolset. GrimoireLab provides an open source set of tools to "retrieve, organize, analyze and visualize software development data"[6]. When compared to other similar tools for analysing software development data, the breadth of data sources is what differentiates GrimoireLab. It can collect data from 30 different kinds of repositories and aggregate it onto a single system.

In order to achieve their flexibility and breadth of tooling, GrimoireLab has adopted a modular design in which each module provides capability to complete one type of task and communicates with the other components of the system. The modular design system has facilitated the creation of an extensive set of first and third party modules which can be used to extract and analyze repository data. Furthermore, this extendability allows researchers to create their own modules for a specific study if none of the existing modules are suitable. However, there are a large number of first-party modules that GrimoireLab has created. One such module that has seen extensive use is the SortingHat [4], a module which tries to merge identities from across and merges them together automatically and also provides an interface for manually merging identities [6]. The modules can also be used as stand alone libraries in Python, which has facilitated wide use of modules like Perceval [5] and SortingHat.

Although the GrimoireLab toolset can provide a wide variety of insights and enables customisation due to it's modular design, in our testing we found that setting up and using the GrimoireLab stack with the relevant modules was quite difficult and might not be worthwhile for a small MSR task. Furthermore, with the scale of the GrimoireLab toolset, understanding the tool enough to be able to write custom modules with specific functionality would be quite difficult. Although we opted not to use GrimoireLab toolset for the creation of replication packages, the concept of an extendable, modular, unified tool is followed by NeoRepro.

When conducting MSR studies, a large number of repositories need to be sampled in order to ensure that a reliable result is reached. The sheer volume of data involved necessitates an efficient way to traverse it and retrieve relevant information. There have been a number of attempts to improve this process. One such approach is to create a Domain Specific Language (DSL) which can be used to query data gathered from mining. For example, QORAL [13] uses a DSL in order to streamline the process of gathering relevant data. A DSL is a software language tailored to a specific domain or task [9, p.1]. DSLs are often compiled into lower level languages so they can be executed. By using a DSL, which is compiled to an executable in a lower level language, they were able to reduce the number of man hours

[4] GrimoireLab SortingHat: https://github.com/chaoss/grimoirelab-sortinghat

[5] GrimoireLab Perceval: https://github.com/chaoss/grimoirelab-perceval

needed to process data for an MSR study [13]. Although using a DSL significantly improves the process for researchers to formulating queries across their datasets, the speed of traversing the dataset to respond to that query depends on the underlying implementation of the DSL. Furthermore, the usage of a DSL rather than a standardized language provides an additional barrier to entry for new users of the tool since they will need to learn the DSL to use the tool, that is only needed for using that particular tool.

Although a DSL such as QORAL can allow data to be easily extracted from a large number of repositories, they do not improve the time taken to traverse all of the repositories and gather the information. To address this, platforms such as Boa [7] provide datasets and infrastructure as a service so that researchers can access pre-created datasets and easily query them. Similar to QORAL, Boa uses a custom DSL which facilitates querying their pre-created dataset on their infrastructure. Accessing Boa is done through a web interface [6] where users can formulate and execute queries on pre-made datasets of open source repositories. Once a query is written, it is executed on Boa's infrastructure. This is convenient for MSR research as it removes the need to "develop expertise in programmatically accessing version control systems" [7] and there is also no requirement to "establish an infrastructure for downloading and storing the repository data" [7]. Additionally, since the datasets are already publicly available, producing a replication package for a new study only requires publishing the Boa queries that were used. These queries can be used by third parties to replicate their results by accessing Boa, thus removing the need to set up a way to distribute datasets.

Despite the convenience of platforms which provide datasets and infrastructure as a service, they cause the replication of studies which used them to be inherently reliant on that service staying online. If a service that provides datasets online were to stop working, it would leave any studies that used the datasets without a method for replication, or access to the original data. For example, SeCold was a "linked data source code dataset for software engineering researchers" [11], but the dataset was shutdown in 2012 due to "a licensing issue" [19]. As a result, a distributed, open source approach might be a better alternative for some, ensuring the longevity and accessibility of datasets and replication packages. This approach reduces the dependency on a single point of failure, and could allow for community maintenance in case the original maintainer is no longer maintaining their tool or dataset. As a result, NeoRepro follows an open source approach to it's data. All source code for the tool and all data for a replication package is published freely for anyone to run on their own computer.

## 2.3 Database Systems for MSR

One way to create a local dataset which is easy to manage and fast to query is to load repository data onto a standard data storage system where it is

[6] Boa Web Interface: https://boa.cs.iastate.edu/

accessed using industry standard query language. One such approach is to load the mined data onto a relational database where SQL can be used be used for queries. With decades of development and widespread usage, relational database systems and SQL have become the standard for data storage. Consequently, anyone with prior experience using a relational database will be able to efficiently access and query the repository data without needing to learn any MSR-specific DSLs or tools. However, since Git repositories have many relationships between items (for example, each commit is related to a previous commit, and is also related to an author, etc) it can be difficult to formulate queries that traverse distant relations. Furthermore, databases with a strict schema may be limiting for an MSR study as adding relevant data is difficult and require changes to the schema of the database. This makes these types of databases difficult to adapt to the needs of an individual study. As a result, GrimoireLab opted to use a no-SQL database in order to provide flexibility in which data is stored in their database.

An alternative data system that can be used is a graph database, which stores data as a collection of nodes and edges, where the nodes (or entities) represent an attribute, and the edges represent the relationship between them [16]. Each node or edge can also contain additional information, much like a row in a relational database system. A graph database is ideal for storing data from Git repositories since a Git repository is, by it's nature, formatted like a graph [20]. For example, commits and authors are treated as entities, and the relationships between the authors and the commits are represented by an edges. Graph databases are ideal in a situation like this, where the data is "highly connected" [16] since they treat relationships as "first-class citizens of the graph data model" [18, p. 6].

The GraphRepo team [20] used a graph database in the creation of creating their tool. They utilised the Graph Database Neo4j [7] to build a platform which data from mined repositories can be loaded onto in order to facilitate queries against the mined data. GraphRepo is a Python library that is built using PyDriller and provides a variety of drillers and miners for MSR. Their drillers are used to extract data from Git repositories and insert it into a Neo4j database. The drillers receive a configuration dictionary for a single repository and drill the repository based on the configuration. Once the information is inserted into a Neo4j database, the miners are used to access a Neo4j database and programmatically extract information for the drilled data. GraphRepo's approach to extracting repository data was the main inspiration for the data drilling and data storage used by our tool NeoRepro. Much like our tool, GraphRepo used PyDriller to perform the drilling. However, there were a few aspects that resulted in us building our tool's driller from scratch. At the time of writing, the most recent commit to GraphRepo is 4 years ago. As a result, a large number of the dependencies that are used have changed significantly. Furthermore, we found that GraphRepo's implementation did not provide sufficient access to PyDriller configurations. This

[7] Neo4j: https://neo4j.com/why-graph-databases/

resulted in drills that searched for specific information drilling a lot of additional information. Finally, there were a large number of additional features and abstractions that were not necessary for the aims of our tool. As a result the drilling we opted to not extend GraphRepo's implementation and instead used our own.

## 2.4 Design decisions for MSR tools

The Kaiaulu team [15] conducted an extensive analysis of MSR tools and their design decisions in order to create their tool. The decisions made by Kaiaulu inspired a number of the design decisions used in NeoRepro. In their analysis they found that there are two main ways of configuring MSR, "tool configuration files" and "project configuration files" [15]. We decided to use the "project configuration file" approach to configuring NeoRepro since it supports "replication through the storage of data in a single harmonized schema". This approach suited our goals of creating a replication package application. Similar to Kaiaulu, NeoRepro uses YAML for it's project configuration files since it is easily human-readable.

The Kaiaulu team also noted different processing methods that were used throughout different MSR tools. They noted that existing tools either use batch mode or an interactive mode [15]. For NeoRepro we opted to use a batch processing mode for the drilling of the repositories since this worked best with the configuration file approach that was mentioned earlier. However for querying the data NeoRepro uses an interactive mode, as the user has direct access to the database and can directly query the data. The Kaiaulu team also noted that "existing tools choose either APIs or Command Line Interface (CLI)s" [15]. We opted to only provide an API for extending the tool. NeoRepro is designed to be primarily accessed through it's frontend.

Finally, the Kaiaulu teams recommends that tools have a minimal path to data [15] in order to allow users to easily start using the tool. The tool should require minimal upfront learning in order for the user to be able to use the tool.

Both the Kaiaulu tool and the NeoRepro tool provide extensive functionality for performing an entire MSR study, however our approach varies slightly from that of Kaiaulu. NeoRepro provides a comprehensive graphical user interface which allows a user to easily interact with the system. Kaiaulu instead provides a CLI and API for interacting with their tool. Both approaches provide access to the extensive features of each tool, but in different ways.

Both Kaiaulu and NeoRepro make use of a Graph model for representing the data retrieved, however where NeoRepro uses the off the shelf Neo4 Graph Database, Kaiaulu uses their own system. We decided to use the off the shelf Neo4j database in order to leverage the extensive query functionality that it has.

# 3 | Requirements

Based on our analysis of existing tools for MSR, we will now outline the requirements that the NeoRepro aims to satisfy. To ensure that our requirements are clear we followed the template outline by Pohl et al [12, p.53], shown in Figure 3.1. This template has the following components:

1. System: The relevant tool or software that this requirement is for
2. Modal Verb: Conveys the necessity of achieving a given requirements. “shall” indicates legal necessity, “should” indicates that it is highly recommended, “will” indicates a future requirement, and “may” indicates a future desired requirement.
3. Process Verb: The core functionality that the requirement specifies.
4. Object: Some process verbs need to be considered with along with a specific object. For example, if the process verb is “print”, the object would be what is printed [12].
5. Additional information: Details that improve clarity of requirement.

## 3.1 Personas

In order to outline the requirements of this system we must first outline the potential user of the system. The following personas represent different types of users who will interact with the tool and how they are likely to use it. The goals and abilities of each persona may vary significantly and thus the

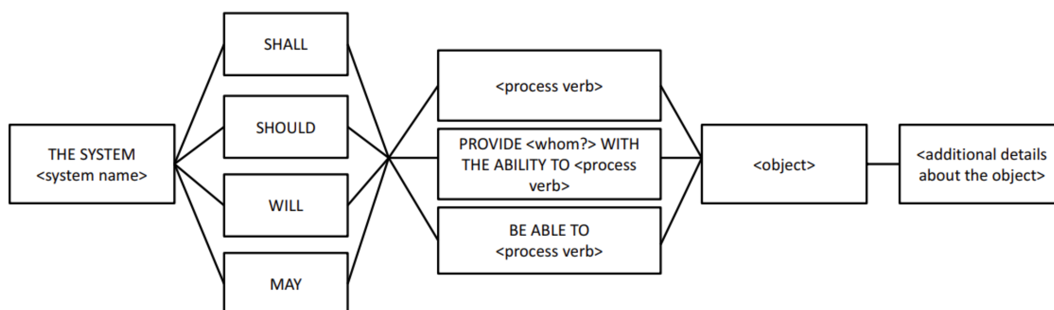


Figure 3.1: Pohl et al Requirements template outline [12]

requirements must be tailored to suite the needs of each.

### MSR Researcher

This persona is using the tool to perform a MSR study and may wish to create a replication package for their study. During the new study this persona may need to extend the functionality of the tool to fit their specific needs since all MSR studies are different.

### Data Consumer

This persona is accessing a replication package created by a MSR Researcher to use the dataset, queries and results from a replication package.

### Study Replicator

This user is using the tool to replicate the results of a previous study that provided a replication package using our tool. This persona has similar goals to the 'Data Consumer' but may wish to replicate the entire process of the study rather than just accessing the dataset.

## 3.2 Requirements

### Functional Requirements

#### Mining the Repositories

The application:

- Will reduce the scripting effort, when compared to writing a bespoke script with PyDriller , required to create a dataset from a list software repositories provided by the user.
- Will have flexibility for the user to define what data is mined from a given repository on per commit basis.
- Will provide functionality to filter a commit based on a string in the commit message.
- Will consolidate mined data from multiple software repositories onto one storage system where it can all be queried simultaneously.
- Will provide the ability to set default drilling parameters which are applied when drilling each individual repository.
- Will allow default mining parameters to be overridden by a configuration for specific repositories to be drilled (i.e. repository specific mining parameters)

- Will provide the user with feedback to indicate the drilling progress of each repository.
- Will allow drill configurations to be saved in the system so that it will be included in the replication package.

#### Querying Mined Repository Data

- Will provide an interface for researchers to query the mined data
- Will utilize a standardized query language to retrieve information from the mined repository data
- Will allow queries to be saved and re-run again later.
- Will allow query results to be saved.
- Will provide example queries so that a new user can get examples of how to use the system.

#### Replication Packages

- Will be able to save the dataset at different stages in the study so that data consumers and study replicators can access datasets from different stages of the study.
- Will be able to package the dataset, queries and query tooling used in a study for distribution as a replication package

### Non-Functional Requirements

- Queries on mined data should be processed faster when using Neo-Repro than if they were run with scripts that traverse the repositories directly
- The user interface should be intuitive and provide access to all system functionality.
- The code for the application will be documented and easily extendable to allow an MSR Researcher to extend the tool for the specific needs of their study
- Using a replication package should require minimal setup and no assistance other than the replication package documentation.
- The system should use standard MSR packages for extracting the data to ensure that MSR Researchers can easily adapt the functionality.

# 4 | Architecture

In this section we will discuss the architecture and technologies used to create NeoRepro. In order to reduce the difficulties during installation and setup of NeoRepro, all components are run within Docker [1] containers and orchestrated using a Docker Compose. "Containers are standardized, executable components that combine application source code with the operating system libraries and dependencies required to run that code in any environment." [26] By using Docker, the user will only need to install Docker on their host system. No additional applications or dependencies need to be installed on the root operating system. NeoRepro can be executed using a single Docker Compose command and can be run on any system with the Docker Engine installed. An overview of the system architecture, broken down into Docker containers is outlined in Figure 4.1.

We decided to structure the core of NeoRepro around a graph database, since the structure of a Git repository is similar in structure to a graph. For this purpose we are using Neo4j, which is particular well suited for modelling and querying graph data. In a study by Jouili et al. [10] Neo4j was found have superior performance when benchmarked against other similar graph databases when doing read intensive operations. However, in the same study, Neo4j was not the best in read-write intensive benchmarks, but since the dataset will be built once but then queried many times, this trade off is acceptable. Furthermore, Neo4j has a query language called Cypher which "is like SQL for graphs" [17] and follows a syntax that is very similar to SQL. By making use of this query language, we will be able to give researchers an interface to query and visualise the graph data which will feel familiar to anyone with experience using SQL. Moreover, Neo4j is a schemaless database, meaning that there is no predefined schema that the data must abide by. This provides flexibility for researchers to add any additional data without the constraints of a predefined schema.

The architecture of the system is broken down into two distinct stages: the drilling stage and the query stage. Drilling is the process of extracting data from a repository. The drilling is performed based on a drilling configuration which describes the repositories that should be drilled and what should be drilled from each. The drilled repository data is then inserted into the Neo4j

[1] Docker: https://www.docker.com/

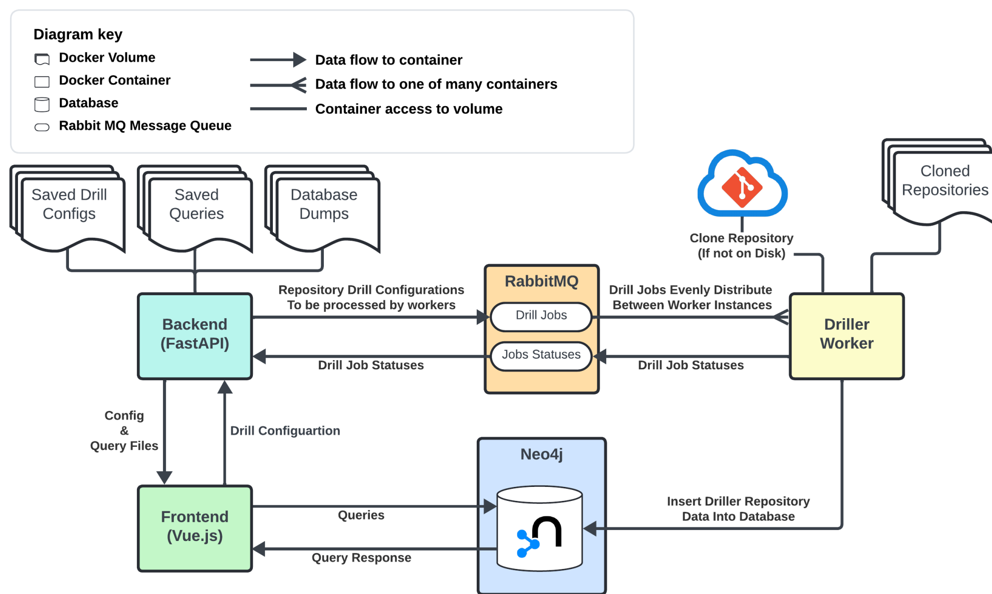


Figure 4.1: Architecture of NeoRepro.

database. In the second stage, the user can utilise the Cypher query language to extract, mutate and reduce the information in the Neo4j database. Backups of the Neo4j database can taken at any time during this stage to allow the database to be reverted. Once the study is complete, all of the drill configurations, Cypher queries and database backups can easily be added to a replication package along with the entire tool that was used to create them.

The frontend of NeoRepro provides a user-friendly interface that simplifies interaction with the various components of the system. The frontend has a corresponding backend through which interaction with the rest of the system is conducted. All standard features of the NeoRepro can be accessed through the frontend, and all drill configurations, Cypher Queries and Neo4j backups can be saved to the backend where they are stored in Docker volumes. These Docker volumes are located directly in the cloned NeoRepro repository on the user's system. This makes it easy for a researcher to commit their files to a repository and leverage the robust version control of Git, whilst also enabling easy distribution of the replication through any of the Git storage platform, such as GitHub.

Once the Docker containers for the application are running, the user can access the frontend at `http://localhost:5173/`. Figure 4.2 shows the home page of the frontend where the README file of the tool's repository is displayed. This allows the user to easily access instructions for using the tool from directly within the frontend. On the left side of Figure 4.2 is the menu which provides access to the other features of the application. In order from top to bottom they are:

1. Home page displaying README.

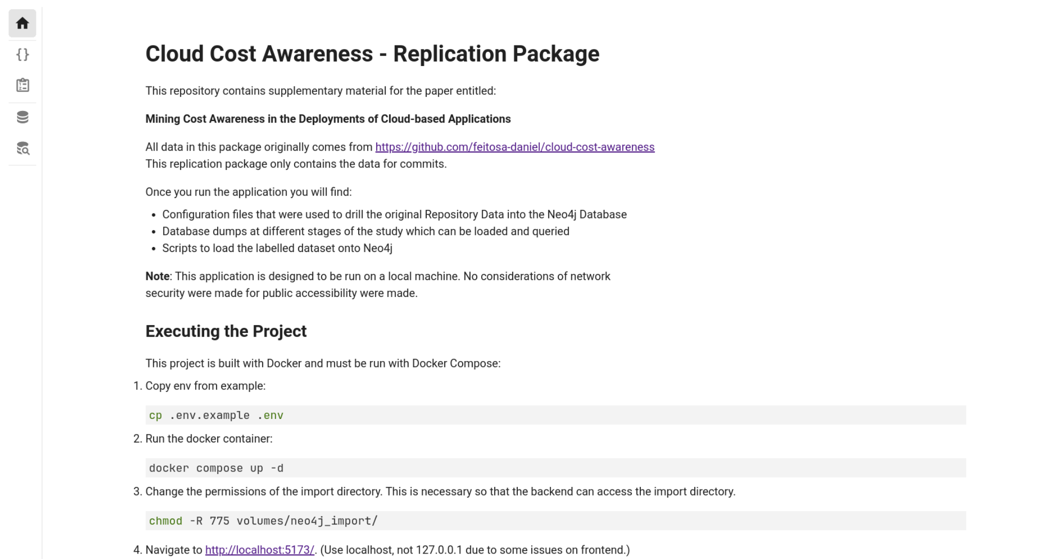


Figure 4.2: NeoRepro frontend home page displaying README and menu.

2. Drill configuration editor. Shown in Figure 4.3
3. Job status view. Shown in Figure 4.5
4. Manage Database. Shown in Figure
5. Query Neo4j. Shown in Figure 4.7

The NeoRepro tool is designed to be run locally on a researchers machine, and thus there are no considerations about access control or network security in the current version of NeoRepro.

## 4.1 Drilling

The first stage of using NeoRepro is the drilling stage. During the drilling stage, data is extracted from the repositories which are being studied and inserted into the Neo4j graph database, where it can be analyzed at a later stage. Rather than creating our own repository drilling functionality from scratch, we opted to utilise the state of the art tool PyDriller [21] to drill the repository data. This provides an MSR researcher with metrics and functionality that they are familiar with.

Instead of requiring the user to create their own script which can extract data from the repositories and insert the data into Neo4j, we to use a YAML configuration. We constructed our own YAML configuration schema which the user can follow to provide a list of repositories and additional configurations to limit which data is drilled from a given repository. This provides the researcher with flexibility in what information is extracted from the repositories rather than extracting all of the data, which can be time consuming for large repositories. We chose YAML as the configuration format for it's

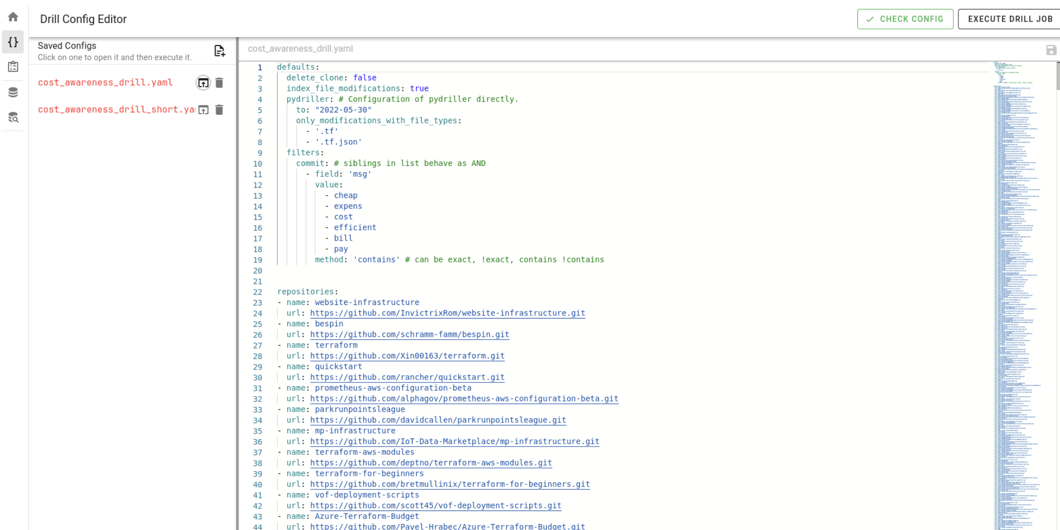


Figure 4.3: The drill configuration editor page of the NeoRepro frontend.

readability and ease of use, which is as a result of the reduced syntactic complexity when compared to other configuration languages, such as JSON.

The configuration file contains two main sections; the defaults and the repositories sections. The repositories section contains a list of all repositories that will be drilled along with drill configurations specific to that repository. The defaults section contains configurations that will be applied to all repositories. If a configuration parameter is present in the defaults and in an individual repository's configuration, the default is overridden for that particular repository. The defaults or an individual repository configuration can contain any of the following sections:

- `pydriller`: Configurations passed directly to the PyDriller `Repository` class [2].
- `filters`: Filters that reduce the amount of drilled data with string comparisons.

Permitting direct access to the parameters of PyDriller from the drill configuration provides flexibility in how the repository is drilled and allows researchers that are proficient in the use of PyDriller to access the parameters they are familiar with, whilst removing the necessity of writing their own drilling script.

For an example configuration, see Listing 5.1 in the Case Study section.

The NeoRepro frontend, shown in Figure 4.3, provides an editor in which the user can compose the drill configuration and check if it is correct using the built in schema check. This editor interface uses the Monaco editor [3], which

[2] Pydriller Repository configurations: https://pydriller.readthedocs.io/en/latest/repository.html
[3] Monaco Editor: https://microsoft.github.io/monaco-editor/

 4.1. Drilling

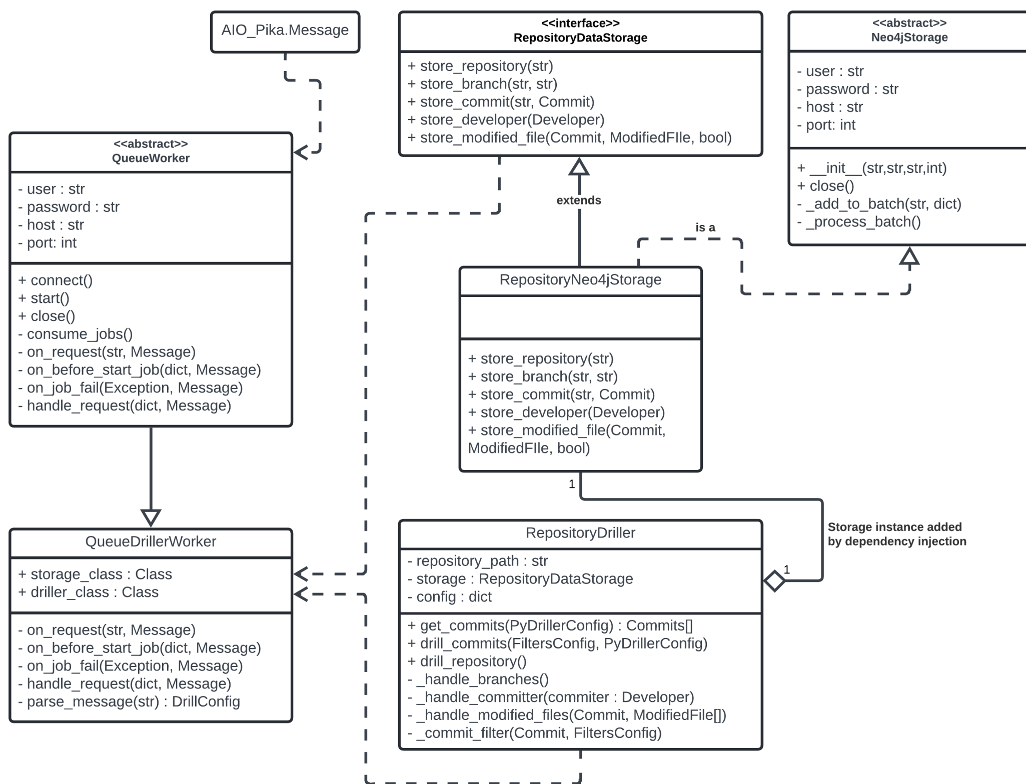


Figure 4.4: UML Class Diagram of Driller Worker.

is the same base editor that is used by Visual Studio Code, so will provide a familiar interface to a large number of developers. Through this interface, the user can save drill configurations or load existing configurations. If the user saves the configuration through the frontend then it will be stored correctly so that it can be distributed in the replication package later on.

NeoRepro uses Driller Workers, which are Docker containers dedicated to executing the drilling of repositories. The Driller Workers are designed such that multiple instances can run simultaneously and drill repositories in parallel. The user can adjust the number of driller nodes that they want to run based on their computer's hardware can handle. To ensure efficient workload distribution, communication between the Driller Workers and the backend is conducted via RabbitMQ, an open source message queueing system [4]. The workers are setup in a Remote Procedure Call (RPC) configuration which enables the backend to invoke drill jobs on the workers as needed and expects a status return value from the workers to indicate the drill job's current status. As a result, when the user executes a drill configuration from the frontend, it is split into one job per repository that is to be drilled, and added to the message queue. The jobs are then evenly distributed among the running driller workers, which execute repository drilling using PyDriller. As the data is extracted from the repository, it is inserted into the Neo4j graph database. The driller workers are implemented in Python in order to make

[4]RabbitMQ: https://www.rabbitmq.com/

Figure 4.5: Frontend interface where the user can see the current status of each drill job.

use of PyDriller, and communication via RabbitMQ is conducted using the package AIO Pika [5]. As shown in Figure 4.4, the driller workers are constructed using Python's Object Oriented syntax to create modular components which can be adapted to the needs of an individual study if our configuration is not suitable.

During the drilling process, the user can access the Job Status page, shown in Figure 4.5, on the frontend to see the current status of the drill jobs in real time. The FastAPI backend receives status messages from the driller workers via RabbitMQ that indicate the current state of the jobs. These messages are sent to the frontend over a WebSocket, which "makes it possible to open a two-way interactive communication session" [23] so that the frontend can receive realtime job status messages.

The repository data is inserted into the graph database in a structure similar to that of the original Git repository. This structure is inspired by that used by the GraphRepo team [20] in their tool. As shown in Figure 4.6 each commit is represented by a node in the database and is linked to the developer who made the commit, the files that were modified and the branches which the commit belongs to. The nodes in Neo4j can hold additional data that was retrieved from PyDriller during the drilling stage, such as the Delta maintainability score. Delta maintainability is a widely used metric in MSR which "is the proportion of low-risk change in a commit" [5]. During the drilling stage, the user can also opt to have file changes (in the Git diff format) recorded in the database too by setting `index_file_modifications` to true in the configuration. If this is enabled then the file changes are added to the `MODIFIED` relationship between the relevant commit and the file nodes.

[5] AIO Pika: https://aio-pika.readthedocs.io/en/latest/

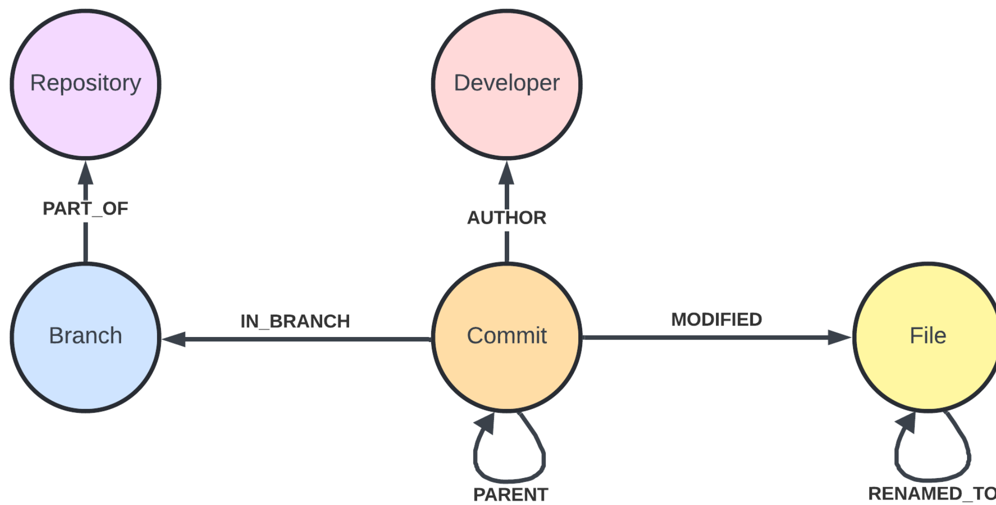


Figure 4.6: Structure of the Graph Database storing Git repositories. Inspired by GraphRepo [20]

## 4.2 Analyzing the mined Data

After the repository data has been inserted into the Neo4j database during the Drilling phase, the data can be queried using Cypher. The use of Cypher makes it easy to query the deeply related data that comes from a Git repository. There are two places where Cypher queries can be executed; Either in the NeoRepro frontend, or in the Neo4j frontend. The NeoRepro frontend, shown in Figure 4.7, provides only basic functionality with regards to autocompletion and data display when compared to the Neo4j interface. It simply displays the data as a table on the frontend. However the main advantage of using the NeoRepro frontend is the ability to save and load queries. Furthermore, the queries that are saved are placed in the correct location for being added to the replication package. Finally, out of the box NeoRepro comes with some example queries that are relevant to querying Git repositories, which could be useful when a user is trying to learn the new system. Alternatively, the Neo4j frontend comes built in to the Neo4j Docker container that we are using for NeoRepro. This frontend provides a more feature rich interface with functionality to display the repository data in a graph diagram, and also has better autocompletion. However, in our testing we found that the rendering of the graph was only useful on small scale queries to visualize data but since it has a render limit of 300 nodes, it is not useful for large datasets and queries. Although the Neo4j interface is not used by NeoRepro, it is still accessible to users if they wish to access it's visualization features.

During a study, additional information may need to be added to the database. For example, in the case study they had to add labels to cost related commits for topic modelling. For small changes to the data, this can be done through a Cypher query interface, but for large changes with very specific functionality

 4.2. Analyzing the mined Data

Figure 4.7: NeoRepro query interface

Figure 4.8: NeoRepro database management and backup interface.

the need arises for writing short scripts which can insert additional data into the database. This is where the object oriented, modular design outlined in Figure 4.4 proves useful. Rather than writing a script from scratch that needs to interact with the Neo4j database, the `Neo4jStorage` class, seen in Figure 4.4, can be extended which provides the functionality for interacting with the database. This makes it easy for custom behavior to be added to NeoRepro. This process was done for the case study in order to insert the labels.

During the analysis stage, snapshots of the current database state can be taken in order to backup the state of the database. These snapshots of the database can be reloaded into the database if a mistake was made during an operation. These database images are also stored in the replication package and thus a third party could access the data at different stages of the study. The interface where database backups are taken is shown in Figure 4.8. In this interface some additional metrics about the current state of the database

are also accessible.

## 4.3 Replication Package

The primary purpose of NeoRepro is to make the creation of a replication package for an MSR study simple. In order to do this, the MSR researcher must clone the NeoRepro repository and use it as a template at the start of a study. This ensures that all necessary components and configurations are available. Once the template is cloned, the researcher can conduct their MSR study using the tool as explained in the previous 2 sections.

The NeoRepro repository contains predefined directories where files used in the study are located. These locations come with some pre-made examples to help guide the user. The files in these directories are passed into NeoRepro's Docker containers using volumes, which allow directories in the host file system to be mapped to a particular location within a given Docker container. Using volumes makes the tool adaptable in case the user would like to change the location where these files are stored.

By leveraging Git for version control, the tool enables researchers to manage their replication package easily and track changes that they make using a tool that is familiar to most. Furthermore, this replication ability can also be used during the study. At any stage of a line of research, if the dataset or configurations need to be shared with a co-worker for replication or validation, this can be done using the replication abilities of NeoRepro by adding it to the Git repository. There are a plethora of platforms, such as GitHub, which can be used to distribute Git repositories, thus making the distribution of the replication package simple.

Once a study is complete and the replication package has been published as a Git Repository, the data and queries used in the study can be easily accessed and interacted with by a third party. By cloning the replication package and executing it with the Docker compose command, the user has access to the entire tool set that was using during the original research. Therefore, regardless of whether the user want to replicate the study in it's entirety, or just access the datasets that were generated during the study, they can do so.

# 5 | Evaluation

To assess the performance and utility of NeoRepro, this section presents a detailed evaluation through a targeted case study. The objective is to illustrate the practical utility of NeoRepro in performing an MSR studies and in generating replication packages. We selected a well-documented MSR study as our case, aiming to replicate the results using NeoRepro. This not only tests the effectiveness of our tool in a real-world scenario but also provides a benchmark against which to measure improvements over existing tools.

In addition to demonstrating NeoRepro's capabilities, we also conduct a comparative analysis with other MSR tools in the field. This comparison focuses on several key metrics, including ease of use, efficiency in data handling, and the overall quality of the replication packages generated. Through this comparative approach, we aim to highlight the distinct advantages that NeoRepro offers over it's counterparts and discuss any aspects that NeoRepro fails to improve over existing tools. The results from this case study will offer valuable insights into the enhancements that NeoRepro brings to the MSR community, highlighting it's potential as an advancement in creation of replication packages.

## 5.1 Case Study

To evaluate the effectiveness of our novel tool, we created the replication package from the "Mining for Cost Awareness in the Infrastructure as Code Artifacts" [8] study in NeoRepro [1]. The replication package is a Git repository which contains all of the drilling configurations, Cypher queries and database snapshots that were used throught the following case study. The original study aimed "to examine to what extent software developers are aware of the cost of deploying and operating cloud-based software, and what kind of concerns and action initiatives they are having about it" [8]. To answer this question, they searched for open source repositories which used Terraform Infrastructure as Code (IaC) to define the resources that software is deployed on. This allowed them to examine whether developers where aware of the influences that their modifications to the code had on the cost of deploying the application. This research serves as an ideal case study as

[1] Case Study Replication Package GitHub Repository: https://github.com/AndrewRutherfoord/cloud-cost-awareness-NeoRepro-reproduction

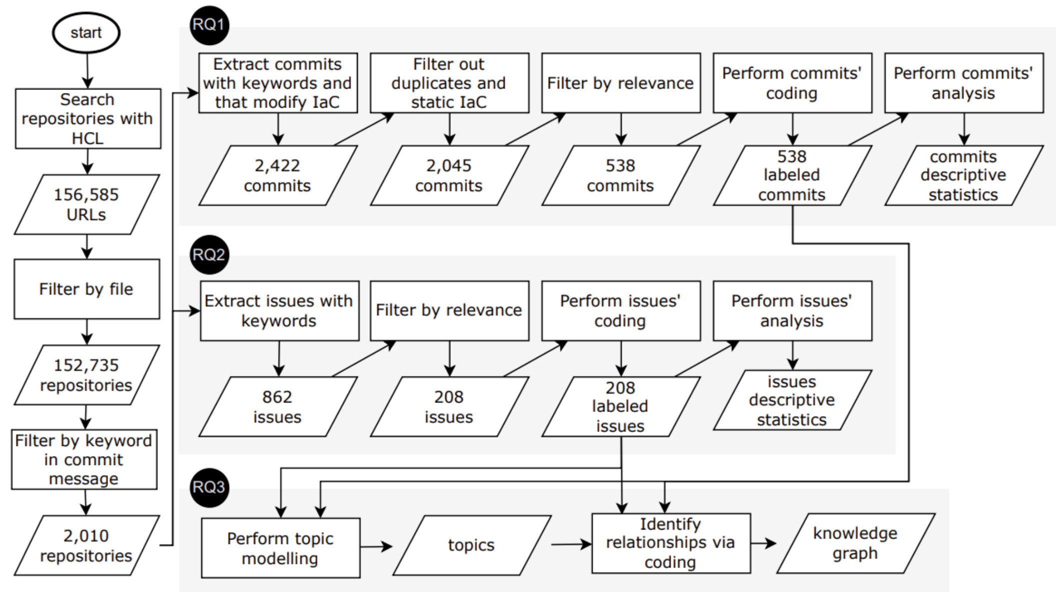


Figure 5.1: Steps of data collection and analysis that were used in Mining Cost Awareness Study [8]

it used a number of bespoke scripts to extract and analyze information from open source repositories. Furthermore, all of the scripts used in the production of the original study have been provided by the researchers in an extensive replication package [2]. The steps that were used to collect the data in the study can be seen in Figure 5.1. In the original study they answered 3 research questions, but in this case study we will only focus research question 1: "What kind of relevant information can we extract from commits on IaC artifacts?". As a result we will not replicate any of the data or scripts that were used in the analysis or research questions 2 and 3. As shown in figure 5.1, the analysis for research question 1 followed 3 steps:

1. Searching for repositories which used Terraform (156585 Repositories were found) [8]

2. Filter out repositories that don't contain Terraform artifact files (Reduced to 152,735) [8]

3. Filter by keyword in commit messages (2010 repositories) [8]

In our replication of this study we will only focus on steps 2 and 3 of the study since the first step, curating a set of relevant repositories to be drilled, differs significantly depending on the source of the data that needs to be analyzed. Since our tool is designed to be a general tool that can be adapted to a study, we didn't see this step as relevant to our aims. In this case they had to query the GitHub API in order to find repositories which were suitable for their study. When doing the case study we will use the curated list of 488

[2]Replication Package for Mining Cost Awareness Study: https://github.com/feitosa-daniel/cloud-cost-awareness/

repositories that was provided in the replication package and perform the second and third steps using our NeoRepro. Unfortunately during our replication we found that 32 of the repositories that were analyzed in the original study were no longer available on GitHub, so we were unable to extract this information. However, we were still left with 456 repositories which could be drilled for our replication. The missing repositories highlights the need to be able to distribute the dataset that was used during the study since the source data may no longer be accessible in the future.

### Drilling the Repositories

The process of replicating the data from the Cost Awareness Study started by converting their curated list of 488 repositories into the configuration file that is used by the drilling component of NeoRepro. Once the all of the repositories were added to the configuration file, we set up the default drill configurations which were applied to each repository. By putting the configurations in the default section, they are applied to the drilling of each repository. Using the `pydriller` section we are able to directly access Pydriller's configurations and the commits that are drilled. In this case we limited the data to which we driller commits to 30th May 2022 [8, p.4], which is the date that the original study ended their data collection. Furthermore, we only drilled commits which contained changes to Terraform files, since these are the only files that the original study was concerned with. We also used the commit string filtering to limit the drilled commits to onces that contained cost related keywords in the commit messages. A snippet of the configuration used in the drilling for the case study can be seen in Listing 5.1.

In this case study we used the configuration file to perform a targeted drill, however by reducing or removing these targeted configurations, we could increase the scope of the data drilling process. This broader approach can be particularly useful in exploratory studies or during the initial stages of research to uncover new insights. This flexibility demonstrates the versatility of our tool, allowing it to facilitate highly focused data extraction when needed, while also being capable of comprehensive data drilling from the repository. This adaptability ensures that researchers can tailor the data collection process to suit the specific needs of their study, whether they require precise, targeted data or a wide-ranging exploratory dataset.

Once the configuration was composed, it was executed from the frontend, which sent the configuration to the FastAPI backend which added each individual repository drill job to the RabbitMQ. The message queue distributed the repository drill jobs evenly among the driller worker nodes, ensuring that the drilling is completed in parallel. When performing this case study, we had 3 driller workers instances running (on a single machine), so 3 repository drill jobs could be executed simultaneously. The number of driller workers can be changed by the user based on the capabilities of their computer. Whilst the drilling was underway, the frontend provided live feedback that showed which repositories were being drilled currently, which had failed,

| Node Type | Number of Nodes |
|---|---|
| Repository | 387 |
| Branch | 335 |
| Developer | 338 |
| Commit | 528 |
| File | 1873 |

Number of each type of node in the database after the initial case study drilling.

and which had already finished. This allows the user to see a high level overview of the drilling process without needing to look at the logs from the driller worker Docker containers. Once all of the repositories had finished drilling with the configurations shown in listing 5.1 we were left with 3475 nodes in the database. The repository data at this point was in the structure shown in 4.6.

```
defaults:
  delete_clone: false
  index_file_modifications: true
  pydriller: # Configuration of pydriller directly.
    to: "2022-05-30"
    only_modifications_with_file_types:
      - '.tf'
      - '.tf.json'
  filters:
    commit:
      - field: 'msg'
        value:
          - cheap
          - expens
          - cost
          - efficient
          - bill
          - pay
        method: 'contains'

repositories:
  - name: website-infrastructure
    url: https://github.com/InvictrixRom/website-
       infrastructure.git
  - name: bespin
    url: https://github.com/schramm-famm/bespin.git
    ...
```

Listing 5.1: Snippet of drill configuration used during “Mining for Cost Awareness” case study

Once the data had been inserted into the database, a database backup was taken in order to create an backup of the original source data. This way if a mistake is made during a query or a script that removes or modifies some data, the database can be rolled back to that version. This backup is also part of the replication package that was created, so therefore a third party will be able to access and use the original data if they need to. Unfortunately the backup taken at this stage is quite large at 1.4 MB and this means that loading the data from the backup can take quite a lot of time.

## Storage of Additional Data

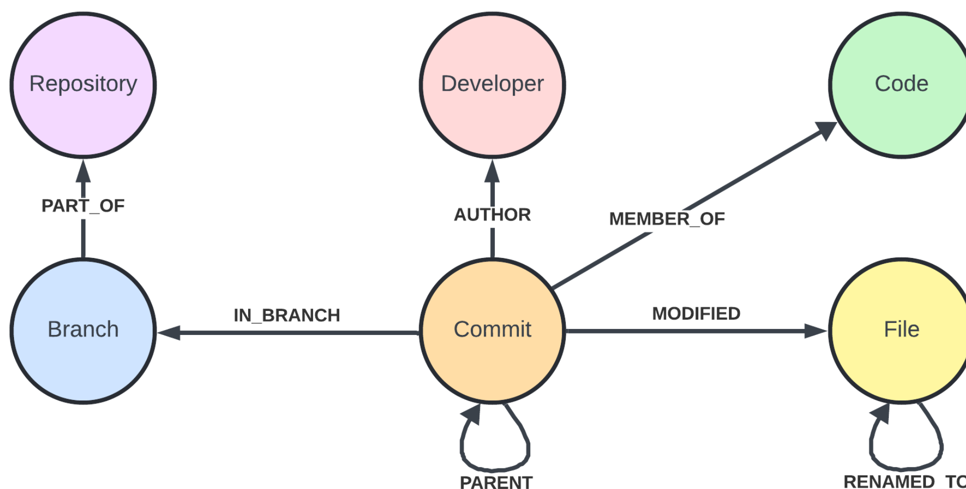


Figure 5.2: Structure of the Graph Database for Mining Cost Awareness replication.

As it stands, NeoRepro is designed to only extract information from Git repositories and insert it into the Neo4j. This means that other datasources often used for MSR research, such as issues or pull requests, cannot be easily mined. However, the schemaless nature of the Neo4j database makes it simple to add additional data, or even adapt the driller worker to extract data from whichever datasource is needed. This extendability of the driller worker is possible due to the object oriented design of the driller worker, shown in figure 4.4. By extending the `Neo4jStorage` class, a script can easily be written that can write data into the Neo4j database that links to the existing repository data.

In the original cost awareness study, they manually decided on codes for each commit that "refer to central ideas in the discussion and their characteristics" [8] during the commit message. For this case study we needed to insert the code data and link it to the commit. We extended the `Neo4jStorage` class, from figure 4.4, and used the Neo4j access functionality it provides to easily write a short script that added the code data. The code data was provided in the replication package from the original study so our script parses the

JSON file, creates a node for each Code and then creates a MEMBER_OF relationship between the code and each relevant commit. The structure of the resulting data is shown in figure 5.2. This resulted in 718 labelled commits. The number of commits per code word can be seen in Appendix A.

The same process can be followed for data from different sources to add them and link it to the repository data. For example, in to answer research question 2 of the Mining for Cost Awareness study, GitHub issues were also studied for information relating to cost awareness. In our study we did not use the Issue data that they gathered, however it could inserted into the Neo4j database easily using this method. Furthermore, the issues could be linked to the repository that it belongs to if it existed in in the database. By unifying all of this data onto a single database where relationships can be created between any two nodes, it facilitates a number of different queries that might not have been feasible with other methods of data storage.

## Analysis of Case Study Data

Now that all of the repository and code data has been inserted into the Neo4j database, the analysis of the data can be performed using Cypher. We wrote queries that replicate the results that the original study performed on this data.

```
MATCH (l:Code)--(c:Commit) RETURN l.name as Name, count(c) as
    Num_Commits ORDER BY Num_Commits DESC
```

Listing 5.2: Cypher query to count commits per code

With a simple query we were able to count the number of commits per label and we also found that saving was the most popular code, at 255 commits, and awareness was the second most popular, at 139 commits. This is the same as they had in the original research paper [8, p.7] The query used is shown in Listing 5.2.

```
MATCH (f:File)-[m:MODIFIED]-(c:Commit)--(b:Branch)--(r:
   Repository), (code:Code)--(commit)
WHERE f.name CONTAINS ".tf"
  AND r.name = "terraform-google-vault"
WITH r, f, COUNT(m) AS modifications, code
ORDER BY modifications DESC
RETURN r.name AS repository, f.name AS file, modifications,
   code.name
```

Listing 5.3: Cypher query most modified terraform file per code

Although we are easily able to query commits per label data using Cypher, this functionality could have easily been achieved though simpler means. Where using Cypher provides significant benefits over other means is when querying data across many relationships. As an example, we decided to write a query to find the most modified Terraform file per code word in a particular repository in the case study. This involves finding the file that is

modified in the most commits in a particular repository and and grouping it by the file code word. The query that retrieves this information can be seen in listing 5.3.

### Releasing the replication package

Since the entire process to recreate the replication package for the Cost Awareness Study was done on NeoRepro, the process of releasing the replication package was simple.

Through the creation of the case study replication package we used the Git repository for version control. This was used repeatedly when rollbacks were required due to a mistaken change to a file. By using Git version control throughout, and publishing the repository on GitHub, the replication package was ready to be released from the start.

In order to create documentation for the replication package, we wrote a `README.md` file, which contains documentation for how to use the replication package. This README file is also displayed on the home page of the frontend so that a third party can easily reference the README whilst using the replication package frontend.

Once the README file was written, the replication package was complete. It contains the drilling configuration, additional scripts, queries and database backups that were used or created throughout the course of the case study.

This shows that using NeoRepro significantly improves the process of creating a replication package for a MSR.

## 5.2 Comparative Analysis

Based on the insights gained through performing the case study, we will now analyze how NeoRepro improves on the state for tools in MSR and replication packages. The case study shows that NeoRepro is capable of performing an MSR study from start to finish.

As shown the Case Study section, NeoRepro possesses the capabilites to perform an entire MSR study from start to finish. The use of a configuration file for drilling repositories is significantly simpler than writing a custom script each time. The configuration file that NeoRepro uses provides more precise configurations for drilling the repository data when compared to other MSR tools which also use a configuration file, such as GraphRepo. Providing direct access to PyDriller's parameters through our YAML configuration allows researchers who are familiar with PyDriller to access the features the are accustomed to without writing a bespoke script.

The ability for NeoRepro to drill multiple software repositories concurrently through the use of the Driller Worker Docker containers significantly speeds

up the drilling process. None of the widely used tools that were surveyed in our analysis of existing tools possessed concurrent drilling abilities.

The original replication package from the Cost Awareness in Infrastructure as Code study had data from each stage of their study, however all of the commit data was just stored in a JSON file as a list, which loses all of the related data such as the developer, the repository information and file change information. This is the primary advantage of the NeoRepro replication method, it provides a simple storage method so that data can be distributed along with all of the context.

As addressed previously, NeoRepro is not able to process other data sources easily. Processing data that is not Git Repository data requires writing a script to load it. To some extend this defeats the purpose of NeoRepro as we aimed to reduce the need for custom scripts. However we feel we have found a fair middle ground by creating a framework that can be extended to create a script to extract additional data.

Another aspect that reduces the effectiveness of NeoRepro as a tool for MSR studies is for a large dataset the backups that are taken from the Neo4j graph database are very large and the process of loading them into Neo4j takes a long time. For the full dataset image from the case study, it took over an hour to load the dataset. However a reduced dataset was able to be loaded quickly. This is unfortunately a shortcoming of the Neo4j database as this is their method for backing up the database. A potential method for improving the loading time of the database it to just take an image of the Neo4j docker volume and add that to the replication package. This method could allow the different states to be swapped our with minimal changeover time. However we are unsure of the size implications of this method. For an initial version of NeoRepro, we feel that the current backup method is sufficient, but could definitely be improved in future iterations of the tool.

# 6 | Conclusion

In this paper, though a comprehensive analysis of existing tools for MSR we have found a need for a system to easily create a replication package for an MSR study. As a result we created NeoRepro, a system for performing a MSR study from start to finish and then allows for the creation of a replication package. The use of NeoRepro for repository mining eliminates the need to write bespoke drilling scripts while providing the flexibility to configure repository data extraction as required by the study. From our analysis of existing tools we found that a graph database would be the most effective storage method for this tool. By leveraging a Neo4j graph database for data storage, NeoRepro efficiently handles complex relationships within the repository data, enabling powerful queries on repository data using Cypher queries.

To evaluate the utility of NeoRepro we conducted a case study by creating a replication package for the "Mining for Cost Awareness in the Infrastructure as Code Artifacts" [8] study. By reproducing the steps of their study on NeoRepro, we were successful in creating a replication package which can be used to replicate the results of their research and can easily be distributed though GitHub.

Although NeoRepro provides powerful functionality for extracting data from repositories, it lacks functionality for extracting data from other sources, such as GitHub Issues or Pull Request, both of which are commonly used in MSR analysis. Although a researcher can extend the object oriented source code of NeoRepro's drillers to extract data from other sources, in future, it would be worthwhile to develop drilling functionality for other data. Additionally, this drilling extended functionality would need to be incorporated into the drilling frontend.

# Acknowledgements

I would like to thank my supervisors Prof. Vasilios Andrikopoulos and Dr. Daniel Feitosa for suggesting this project and providing invaluable guidance and feedback throughout the process of creating the NeoRepro tool and thesis.

# Bibliography


[1] *Artifact Review and Badging - Current*. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 08/02/2024).

[2] Markus Borg et al. "Increasing, Not Diminishing: Investigating the Returns of Highly Maintainable Code". In: *Proceedings of the 7th ACM/IEEE International Conference on Technical Debt*. TechDebt '24. New York, NY, USA: Association for Computing Machinery, June 2024, pp. 21–30. ISBN: 9798400705908. DOI: 10.1145/3644384.3644471. URL: https://dl.acm.org/doi/10.1145/3644384.3644471 (visited on 08/02/2024).

[3] Enzo Camuto et al. "A Suite of Process Metrics to Capture the Effort of Developers". In: *Proceedings of the 2021 10th International Conference on Software and Computer Applications*. ICSCA '21. New York, NY, USA: Association for Computing Machinery, July 2021, pp. 131–136. ISBN: 978-1-4503-8882-5. DOI: 10.1145/3457784.3457805. URL: https://dl.acm.org/doi/10.1145/3457784.3457805 (visited on 04/01/2024).

[4] Morakot Choetkiertikul et al. "Characterization and Prediction of Issue-Related Risks in Software Projects". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. May 2015, pp. 280–291. DOI: 10.1109/MSR.2015.33. URL: https://ieeexplore-ieee-org.proxy-ub.rug.nl/document/7180087 (visited on 08/02/2024).

[5] Marco Di Biase et al. "The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. Montreal, QC, Canada: IEEE, May 2019, pp. 113–122. ISBN: 978-1-72813-371-3. DOI: 10.1109/TechDebt.2019.00030. URL: https://ieeexplore.ieee.org/document/8785997/ (visited on 07/02/2024).

[6] S. Dueñas et al. "GrimoireLab: A Toolset for Software Development Analytics". In: *PeerJ Computer Science* 7 (2021), pp. 1–53. ISSN: 2376-5992. DOI: 10.7717/PEERJ-CS.601.

[7] Robert Dyer et al. "Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories". In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 422–431. ISBN: 978-1-4673-3076-3 978-1-4673-3073-2. DOI:

10.1109/ICSE.2013.6606588. URL: http://ieeexplore.ieee.org/document/6606588/ (visited on 06/22/2024).
[8] Daniel Feitosa et al. *Mining for Cost Awareness in the Infrastructure as Code Artifacts of Cloud-based Applications: An Exploratory Study*. Dec. 2023. arXiv: 2304.07531 [cs]. URL: http://arxiv.org/abs/2304.07531 (visited on 02/27/2024).
[9] Paul Hudak. "Domain Specific Languages". In: (Dec. 1997). URL: https://cs448h.stanford.edu/DSEL-Little.pdf (visited on 12/02/2024).
[10] Salim Jouili and Valentin Vansteenberghe. "An Empirical Comparison of Graph Databases". In: *2013 International Conference on Social Computing*. Sept. 2013, pp. 708–715. DOI: 10.1109/SocialCom.2013.106. URL: https://ieeexplore.ieee.org/abstract/document/6693403 (visited on 03/30/2024).
[11] I. Keivanloo and J. Rilling. "Software Trustworthiness 2.0 - A Semantic Web Enabled Global Source Code Analysis Approach". In: *Journal of Systems and Software* 89.1 (2014), pp. 33–50. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.08.030.
[12] Chris Rupp Klaus Pohl. *Requirements Engineering Fundamentals, 2nd Edition: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB Compliant*. 2nd Edition. Rocky Nook, Inc., Apr. 2016. ISBN: 978-1-937538-77-4.
[13] Hiroki Nakamura et al. "QORAL: An External Domain-Specific Language for Mining Software Repositories". In: *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. Oct. 2012, pp. 23–29. DOI: 10.1109/IWESEP.2012.20. URL: https://ieeexplore.ieee.org/abstract/document/6363292 (visited on 03/13/2024).
[14] YAML Org. *The Official YAML Web Site*. URL: https://yaml.org/ (visited on 06/29/2024).
[15] Carlos Paradis and Rick Kazman. "Building the MSR Tool Kaiaulu: Design Principles and Experiences". In: vol. 13365. 2022, pp. 107–129. DOI: 10.1007/978-3-031-15116-3_6. arXiv: 2304.14570 [cs]. URL: http://arxiv.org/abs/2304.14570 (visited on 07/04/2024).
[16] Jaroslav Pokorný. "Graph Databases: Their Power and Limitations". In: *Computer Information Systems and Industrial Management*. Ed. by Khalid Saeed and Wladyslaw Homenda. Cham: Springer International Publishing, 2015, pp. 58–69. ISBN: 978-3-319-24369-6. DOI: 10.1007/978-3-319-24369-6_5.
[17] *Query a Neo4j Database Using Cypher - Getting Started*. URL: https://neo4j.com/docs/getting-started/cypher-intro/ (visited on 07/01/2024).
[18] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. "O'Reilly Media, Inc.", June 2015. ISBN: 978-1-4919-3086-1.
[19] secold. *Some Research Domain Shutdown OpenData Project Due to Licensing Issues http://bit.ly/OqshMK but We Keep Our Hope Alive http://bit.ly/ROFmPO*. Nov. 2012. URL: https://x.com/secold/status/245295898840678400.

[20] Alex Serban, Magiel Bruntink, and Joost Visser. *GraphRepo: Fast Exploration in Software Repository Mining*. Aug. 2020. DOI: 10.48550/arXiv.2008.04884. arXiv: 2008.04884 [cs]. URL: http://arxiv.org/abs/2008.04884 (visited on 02/20/2024).

[21] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "PyDriller: Python Framework for Mining Software Repositories". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 908–911. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3264598. URL: https://dl.acm.org/doi/10.1145/3236024.3264598 (visited on 02/20/2024).

[22] Kyle Daigle Staff GitHub. *Octoverse: The State of Open Source and Rise of AI in 2023*. Nov. 2023. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/ (visited on 03/13/2024).

[23] *The WebSocket API (WebSockets) - Web APIs | MDN*. Mar. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 07/17/2024).

[24] M. Vidoni. "A Systematic Process for Mining Software Repositories: Results from a Systematic Literature Review". In: *Information and Software Technology* 144 (Apr. 2022), p. 106791. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106791. URL: https://www.sciencedirect.com/science/article/pii/S0950584921002317 (visited on 02/20/2024).

[25] *What Is an API (Application Programming Interface)? | IBM*. Apr. 2024. URL: https://www.ibm.com/topics/api (visited on 08/01/2024).

[26] *What Is Docker? | IBM*. June 2024. URL: https://www.ibm.com/topics/docker (visited on 06/30/2024).

# Glossary

**API** "a set of rules or protocols that enables software applications to communicate with each other to exchange data"[25]. 9

**git** A version control system that is widely used within software development. A project that is managed by git is called a repository. 2–5, 8, 13, 14, 18, 19, 21, 22, 26, 28, 29

**JSON** A standard file format storage of data. . 16, 27, 29

**SQL** Structured Query Language. A standardised language that is used to make queries in relational database systems.. 8, 13

**YAML** YAML Ain't Markup Language™: YAML is a human-friendly data serialization language for all programming languages. [14]. 15, 28

# Acronyms

**CLI** Command Line Interface. 9

**DSL** Domain Specific Language. 6–8

**IaC** Infrastructure as Code. 22, 23

**MSR** Mining Software Repositories. 3–12, 15, 18, 21, 22, 26, 28–30

**RPC** Remote Procedure Call. 17

# A | Additional Data

## Drilling Configuration Example

Listing A.1: Snippet drill confugration used during "Mining for Cost Awareness" case study:

```
defaults:
  delete_clone: false
  index_file_modifications: true
  pydriller: # Configuration of pydriller directly.
    to: "2022-05-30"
    only_modifications_with_file_types:
      - '.tf'
      - '.tf.json'
  filters:
    commit: # siblings in list behave as AND
      - field: 'msg'
        value:
          - cheap
          - expens
          - cost
          - efficient
          - bill
          - pay
        method: 'contains'

repositories:
- name: website-infrastructure
  url: https://github.com/InvictrixRom/website-infrastructure
     .git
- name: bespin
  url: https://github.com/schramm-famm/bespin.git
```

## Number of Commits Per Code

| Name | Number of Commits |
|---|---|
| alert | 28 |
| area | 9 |
| awareness | 139 |
| billing_mode | 19 |
| cluster | 11 |
| domain | 5 |
| feature | 39 |
| increase | 8 |
| instance | 91 |
| networking | 39 |
| policy | 6 |
| provider | 17 |
| saving | 255 |
| storage | 52 |