Universität Bremen

Development of a Server for Code Smell Data

Bachelor Thesis

University of Bremen

Author:

Felix Frederick Becker

Enrolment number 4227648

Handed in on:

February 17, 2020

Primary reviewer:	Prof. Dr. rer. nat. Rainer Koschke
Secondary reviewer:	Dr.-Ing. Hui Shi

Declaration of Academic Honesty

Hereby I declare that my bachelor thesis was written without external support and that I did not use any other sources and auxiliary means than those quoted. All statements which are literally or analogously taken from other publications have been identified as quotations.

Date	Signature

Table of Contents

Table of Figures

List of Tables

Table of Listings

1 Introduction

1.1 Motivation

Modern software is more complex than ever and needs to be maintained for ever longer periods, as users expect software to be continuously updated or even be provided as service and subscription models. At the same time, we are seeing a purge of innovation in new software products in a highly competitive market that demands rapid release cycles. This demand can often push developers to take technical debt to ship features faster, intentionally or unintentionally sacrificing the maintainability of their software, often leading to the introduction of “code smells.”

This makes the emergence and evolution of code smells an interesting target for research, one that is actively explored in the software technology working group at the University of Bremen [1] and other research institutions [2] [3]. Martin Fowler defined code smells as “symptoms of poor design and implementation choices” [4]. Examples of interest include how these code smells emerge, what negative effects they have, how they can be detected, avoided, and removed automatically.

To research these questions, research projects usually *require* a significant set of data to analyze. Presently, this is raw source code of projects to analyze, optionally with versioning information to research the *evolution* of code smells. As a result of the research they also *produce* data such as detected code smells, optionally tracked across revisions of the code, revealing at which point in time the code smell was *introduced* and *removed* again, e.g. through refactoring. This again can be the input to higher-level research that combines the knowledge about the evolved code smells with other data, such as the authors that worked on the file [1], the number of authors, the stage in the lifecycle of the software [2], and many more.

Especially the higher-level research currently incurs a high overhead, as researchers first need to start from scratch to develop a significantly large data set of code smells by raw source code analysis. Such analysis takes a long time (especially if it needs to be applied to many revisions for the purpose of capturing their evolution) and incurs additional work to bring the output of different tools into the same format. The actual research part is non-trivial either, as such a data set can be massive in storage size.

This makes efficient analysis a challenge, especially when code smell data needs to be correlated back to other data such as the file content or versioning information.

1.2 Structure

This thesis is structured into 8 further sections. Chapter 2 describes the goal of the project and defines the scope. Chapter 3 lays out prior research in this area that provide context and inspiration. Based on the defined goals, Chapter 4 explains the choices in technology that were made to rely on for the implementation of the project. Following that, chapter 5 goes into detail of the chosen data models and chapter 6 into the implementation of business logic, public API and data access. Chapter 7 walks through the usage of the server and demonstrates its capabilities and flexibility. To verify the project meets the goals outlined in chapter 2, the project is evaluated in chapter 8 by reproducing the research of select research questions in a recent thesis. Lastly, chapter 9 contains a conclusion and outlook.

2 Goal

The challenges described in 1.1 should not need to be solved for every new research project. Instead, a new project should be able to rely on a pre-existing, publicly available, sizable set of code smell data in a consistent format. This data should be easily queryable such that it is easy to correlate with a diverse range of metadata. Furthermore, it should be possible to easily contribute the results of research projects back into the data store, for future research to build on top of.

The goal of this project is to find out whether a data store can be built that solves the aforementioned use cases.

2.1 Features

The server should allow a variety of queries over the data from different views.

F1. Basic requirements are querying the available repositories and their version control history. For each revision in the version history, it should be possible to query information about the state of the repository at that point in time, e.g. what files existed and their contents.

F2. In addition, the information about the *changes* from revision to revision need to be exposed, e.g. what files changed, the commit message, date, and time. Revisions should be addressable by different criteria, e.g. the commit ID, a range of dates, or a filter on the commit message.

F3. Code smells should be queryable in particular from two views, referred to as the *horizontal* view and the *vertical* view. *Horizontal* shall mean inspecting the evolution of a code smell throughout the repository's history.

F4. The *vertical view* refers to inspecting all the code smells in a specific revision.

F5. In every view it should be possible to directly include related version control information mentioned above with code smells, e.g. the contents of the affected range, or metadata about the commit it appeared in.

F6. Code smell queries should also be filterable, by criteria such as the code smell kind (e.g. "God Class"), the affected folder, file, or file type.

F7. All lists that are exposed through the API must support pagination, as the server will have to deal with copious amounts of items. This means to support limiting the queried list to an arbitrary number of items and continuing from the end of the list in a follow-up query. Analyses that research projects must run over code smells may take multiple weeks to complete. With these timeframes, it is desirable to iterate over code smells in pages and it is possible that data (code smells, version history, …) gets added or removed on the server in between requests for the next page. As such, it is important that pagination is implemented such that this scenario under no circumstances would cause elements to be iterated multiple times, or elements to be skipped.

F8. Since a research project that contributes analyzed code smells will likely want to be able to evaluate their own contributed code smells in isolation, the server needs to have the ability to tie code smells to the analyses they were discovered in. This should also allow deleting all the code smells of an analysis in case of a mistake.

2.2 Non-Goals

While there are a lot of related problems in this space worth solving, there are some that are out of scope for this thesis or better left to other tools.

The server itself is not intended to do any analysis on the code itself, but only store code smell data that was produced out-of-band. Building analyzers into the server would make the server not analyzer- and language-agnostic.

Further, it is not the goal of the code smell store to become a general-purpose store for arbitrary metadata on source code. The data model is specifically optimized for the domain of code smell research. While other metadata can be interesting in certain circumstances in the context of code smell research, this project only includes directly inferable information from the stored data, that is mostly language-agnostic (e.g. line count of files).

Research over large bodies and histories of source code is an inherently lengthy process. As such, the server does not need to be highly optimized to return responses in under a second. A primary goal is flexibility in query capabilities, which is prioritized over query performance.

The primary goal of the code smell store is to *expose* data in a flexible way with a rich data model. It is expected that the consumer needs to apply further client-side processing (such as aggregation, grouping, counting) to the data to answer their research questions. Moving such aggregation into the server would require predicting every conceivable way the data may be used, which is not feasible.

The initial version presented in this thesis does not have any authorization system. HTTP basic authentication can be configured, but access is “all or nothing.” The server could be extended later with a more complex permission system, but that is out of scope for this thesis.

3 Prior Work

Code smells in software systems and their evolution were the target of a significant body of existing research.

M. Steinbeck [5] created a library that can track code smells across the version history of Git, Mercurial and SVN repositories and is therefore suitable to research the evolution of code smells. It walks a given range of revisions in a repository and tries to map every code smell to a successor and predecessor in the previous and next revision, respectively. This library was used in a recent bachelor thesis on code smell evolution by D. Schulz, who indicated in his outlook that a REST API for code smells would make it easier to process results [6, p. 55].

Landfill [7] by Palomba et al. is a public dataset of code smells with a web UI. Code smells can be uploaded as CSV files and repositories as archives. However, it is not built to research the *evolution* of code smells. The repository upload only contains the specific version that was analyzed. Code smells note the affected part of the codebase through its fully qualified class name. It also features an API, which gives access to the file contents of the analyzed repositories at the uploaded version. The project has some interesting social features, such as voting for the accuracy of a detected code smell. Unfortunately, the public instance linked in the paper does not appear to be online anymore.

Palomba et al. did utilize version history in another paper [8] to better *detect* code smells, but not to research the *evolution* of them over time. Lozana et al. made use of version history for the same goal [9]. This usage is orthogonal to the planned code smell store — the code analyzer that produces the code smells is not precluded from making use of version history for detection, but the server's goal is to expose version history for the sake of *evolution* research.

There are online platforms for storage of generic research datasets such as *Mendeley Datasets*, that host some code smell related datasets [10]. However, since the platform is not focused on code smells specifically, datasets are not easily queryable, but only available for download as files in the format they were uploaded in (e.g. .zip, .xlsx, .txt).

General data models of code smells can be found in tooling and editor infrastructure. For example, the Language Server Protocol [11], an editor-agnostic RPC protocol for code intelligence functionality based on JSON, has the concept of a “code diagnostic” that expresses a general problem about a part of code. This can be either something like a code smell, or something more severe like a compile error. Diagnostics in the protocol have a message, a single location, code, severity, source (e.g. name of the generating tool), zero to many tags, and “related” locations.

4 Technology Stack

The code smell server is composed of a storage component (database) and a public-facing API that exposes query and upload functionality specific to the domain of code smell analysis. As every project, the code smell server needs to build on top of a variety of fundamental, existing technologies that are explained in this chapter.

To allow easy consumption, the code smell store is exposed as a web service over HTTP. While HTTP allows a variety of data exchange formats, JSON is the most common format for modern web APIs and easily integrable into most client languages.

Two architectural styles have emerged as the most popular for web APIs: REST [12, pp. 76-106] and GraphQL[1]. The architectural style REST (representational state transfer) relies mostly on the core functionalities of the HTTP protocol. Every resource or collection is addressed by its URL, and different actions are indicated through HTTP verbs. APIs following REST are easily consumed from any environment and can build on decades of compatible tooling. A big disadvantage of REST is that resources expose mostly fixed views. This means views often either return more data than needed, or not enough and require subsequent requests to correlate data. GraphQL's main objective is to solve these problems, by allowing the client to query all the data it needs over a strongly typed graph data model. The disadvantage of GraphQL is a higher implementation complexity on the server to serve these arbitrary views, and to reimplement many of the features HTTP supplies for URLs through GraphQL. Because of its younger age, it is also less known and may require a small learning curve.

For the code smell server, query functionality must be flexible enough to be tailorable to any code smell research project and not limited to a handful of views. At the same time, it must handle requests for large amounts of data, where it becomes important to not send unnecessary bytes over the wire. These attributes are better provided by GraphQL than REST. Because GraphQL allows the client to freely select any needed subset of the available fields, the server can expose a lot more fields to query without having to worry about bloating the payload with fields not asked for. This especially supports feature F5 to correlate various data with code smells from different views, as the consumer can construct a query resulting in a graph structure suiting their needs.

[1] https://graphql.org/

Another advantage is the integrated query explorer GraphQL comes with. This GUI is accessible through the web browser and provides autocompletion and documentation while writing a query. This adds discoverability to the API without requiring engineering effort spent on a GUI. In opposite to a web UI specifically built for the server, the query explorer allows to trivially translate any query into an automated script for more advanced use cases.

Ideally, we can expose both GraphQL for advanced queries and REST endpoints for simpler use cases or more constrained environments. This adds additional implementation complexity, but this complexity can be minimized by having the REST endpoints execute GraphQL queries internally. The REST endpoints are therefore merely small wrappers that expose a handful of fixed views. They still support the same filter capabilities (F6) through query parameters and pagination (F7) through Link headers as specified in RFC8288 [13].

A desired feature is the correlation of code smells with the file content they affect, as well as with revision metadata like message, author, date, or other files changed in the revision (F5). This versioning data comes from a version control system such as Git, Subversion or Mercurial, and must be uploaded together with the code smell data. Most open source projects, especially since the advent of GitHub, use the version control system Git nowadays. Git is *distributed*, which makes it easy to push a copy of all versioning information of a project to a different server (and keeping that copy up to date). Supporting multiple version control systems is possible, but would drastically increase complexity, as it would require designing an abstract data model over multiple VCS. This would exclude the opportunity to directly rely on Git's unique features. For the less common cases of analyzing a code base using SVN, Mercurial or Perforce, Git provides features to easily convert these to Git repositories [14, pp. 344-356]. Because of this, the code smell store will support only Git as source of version control system data.

The most efficient way to transfer the version history is the `git push` command, as Git's protocol is highly optimized. This is enabled by allowing the code smell server to be added to Git as a *remote*. It also allows updating the version history incrementally later. Since Git supports transfer over HTTP, the endpoint can be integrated into the same HTTP API, which also allows it to share any future features added like authentication.

For cases where the server is not directly reachable by the repository owner, `git push` may not be possible. In these cases, the owner can pack the repository into a single file with the `git bundle` command. This bundle can be transferred through other means (e.g. email) and be directly uploaded to the server with a POST request. This method is still superior to uploading a file archive of the repository, as Git bundles can be created from a subset of the history and support incremental updates as well.

Non-Git data will need to be stored in an external database. Relational databases are the most mature in this space and because of that a good default choice. A different database system *may* yield a better query performance, but the evaluation of multiple alternative database systems would be out of scope for this project and is left for future projects. The open source relational database system PostgreSQL[2] is the option among the available RDBMS that I have the most experience with.

The backend could be implemented in a variety of languages. The reference implementation of GraphQL is written for NodeJS[3], which makes it a good option. It is also the one I have the most personal experience with. Node generally allows to quickly implement HTTP-based APIs and offers a great ecosystem of open-source modules. On top of NodeJS, TypeScript[4] adds type safety to the codebase, which prevents a lot of issues and accelerates development.

The server should be easy to deploy and self-host, as well as have a reproducible build process. To ensure this, the server will be designed to run in an isolated container. The image will be defined in a Dockerfile and will be publicly available from Docker Hub[5].

[2] https://www.postgresql.org/
[3] https://nodejs.org/
[4] https://www.typescriptlang.org/
[5] https://hub.docker.com/r/felixfbecker/olfaction

5 Data Model

This chapter explains the data model of the code smell server. The most important side of this is the GraphQL schema, which is the rich domain-specific data model the API consumer will interact with. It is described in 5.1, while the internal database model is explained 5.2.

5.1 API

The structure of the data to be stored and queried affords modelling it as an object graph, which will be exposed through GraphQL. This graph is strongly typed in a GraphQL schema, which is visualized as a UML class diagram in Figure 1. A GraphQL query then allows to arbitrarily follow edges of this graph, starting at the root `Query` type and returning them as a deeply nested JSON object structure.

The `Query` type exposes as a top-level entity the available analyses. An analysis is the outer-most container for detected code smells. Its purpose is to allow to only look at data from a specific analysis without mixing it with prior analyses (F8). In addition, it exists to record which commits of which repositories were analyzed. In particular, this allows the important distinction between commits that *have no code smells* and commits that *were not analyzed*. This is especially important when only analyzing a subset of the history in a research project, such as a date range or only every nth commit.

Every other piece of data is contained within a `Repository`. Repositories have a unique *name* that they can be addressed by from the root of the API. This name is also used to address it when uploading Git history. They can also be queried from the root (F1).

From a repository, revisions are exposed through the `Commit` type. Commits are always queried from a start revision, which can be a commit, a branch, or a tag, and defaults to the repository HEAD. The returned list is the result of walking all commits by following their parent commits. This list may be filtered through parameters, to only query between a start and end date or by matching the commit message with a regular expression.

A repository also exposes a connection to the *lifespans* of code smells throughout the history, enabling the *horizontal view* (F3). From a lifespan, its *instances* throughout the commit history can be acquired, which is a connection to the `CodeSmell` type, containing a linear list. Each of those instances then links to the commit it was detected in. The `Lifespan` type hosts the field for the code smell kind, because the kind must be consistent across all instances of the code smell lifespan.

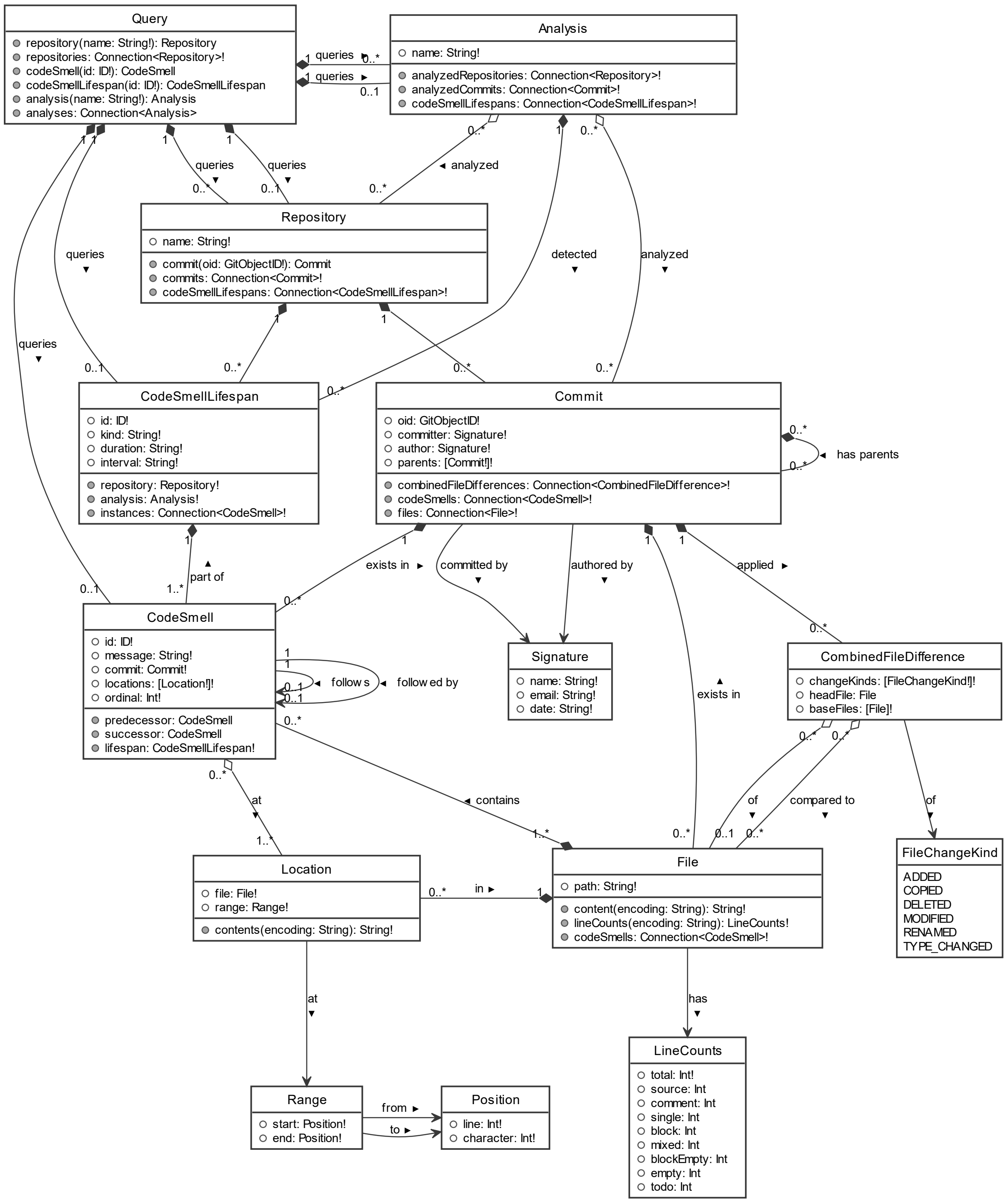


Figure 1: GraphQL schema.

Besides that, lifespans should expose their total age in a format that is easy to consume. This can be calculated on demand from the commit metadata of the first and last instance of the code smell lifespan. The ISO6801 standard defines string formats for durations (e.g. `P12H30M5S`) and intervals (e.g. `2007-03-01T13:00:00Z/2008-05-11T15:30:00Z`). In opposite to alternative means like exposing the duration in seconds, the ISO6801 standard allows to represent even large intervals and durations in a way that is both still human-readable and at the same time parsable by most programming languages through their standard library. This format is used to expose the age of a lifespan in the `duration` and `interval` fields. An application of the `duration` field can be found in 8.1.

To be noted is that the largest unit that the ISO duration string can contain is hours. To calculate days, months and years, the time zone of the commits would need to be known, as the length of a day can vary throughout the year depending on the time zone (e.g. because of daylight saving time shifts). However, Git only stores the time zone *offset* from UTC for each commit, which is not enough to allow date arithmetic.

To enable the *vertical view* (F4), a commit (that can be queried from a repository) exposes a connection to all the code smells detected in that commit.

Fields exposed on the code smell include its ID, message, and its locations in the source code. Every code smell also links to its lifespan. The position within the lifespan is given through the `ordinal` field. The code smell also directly references its own predecessor and successor, given they exist, otherwise these fields are `null` respectively. This matches the data model of LibVCS4J [5], which also maps every code smell to a single predecessor and successor, given one was found.

A location is defined as a range in a file, where a range is a pair of a start and end position. Positions in a file could be represented in different ways. A single integer representing a byte offset can be enough to identify a position but is not intuitive to work with. Better suited for human-readable presentation is a pair of line and character position. Git also generally operates with lines in diffs, so line numbers are important when calculating how the line of a code smell moved through the changes in a commit.

The `character` field is intentionally not a "column" as presented by a file viewer, as that number is a lot more complex to calculate and therefore error-prone to work with. For example, the width of a tab character depends on the tab size preference.

Unicode also contains many characters that are less than one column wide (e.g. zero-width space) or multiple columns wide (e.g. emojis).

Both line and character also need to be either zero- or one-based. The open Language Server Protocol [11], which also features an otherwise equal type definition for Positions, uses zero-based numbers, therefore the same was chosen for this schema.

For each location, the content of only this range can directly be accessed through its content field, which is sliced from the file content on demand when the field is queried.

Besides code smells, commits expose the known metadata about the commit, such as the commit message, committer, and author information. Committer and author information are each represented as a `Signature`, with name, email, date, time, and time zone offset (F2). The last three items are all exposed as a standard ISO6801 date string. Every commit also links to its parent commits, which contains multiple in the case of a merge commit or none in the case of the root commit.

The fact that commits can have multiple parents also needs to be accurately reflected in the way file changes are exposed from the commit. Git itself has two ways to present these, either compared individually to each parent (the `-m` flag to every command that produces diffs) or in the "Combined Diff Format" [14, pp. 282-284]. The latter only includes file changes that were changed when compared to all parents and for each changed file, lists the detected kind of change, and the corresponding parent file for each parent commit. The combined format is the default in Git and generally easier to work with. Because of this, an object structure representing this format is what is exposed in the API.

The file paths of the parent files may be different in the case of a file rename or copy, as indicated by the `FileChangeKind` enum. If the file did not exist in the parent, or no longer exists in the new commit (as is the case with a change kind of `ADDITION` or `DELETION` respectively) `headFile` may be `null` or `baseFiles` may contain `null` at the position of the respective parent. The `CombinedFileDifferences` type is currently only used for commits, but general enough to e.g. also be used for comparisons between arbitrary revisions (which is why it uses the term "baseFiles" and not "parentFiles").

Commits also expose a direct connection to files, which are the files that are present in the repository at that commit. These can optionally be filtered by a regular expression applied to the file path, which enables filtering by a subdirectory, as well as filtering by a file extension (F6).

Files most importantly expose their file path (relative to the repository root) and their content as a string. Since the encoding of the file is not known, it can be provided as a parameter. If not given, the server will try to use common heuristics to guess the encoding, otherwise fall back to UTF8.

Knowing the content, every file also exposes a field `lineCounts` with metrics related to number of lines. Metrics include the physical line count, lines of code, lines with comments, empty lines, and lines with "TODO" comments. This is computed on demand by the `sloc` library only if the field is queried and supported for 52 of the most common languages (otherwise only physical line count is available).

Files also expose a connection to the code smells that existed in exactly this file at this commit. A use case of this field can be found in 8.3.

5.1.1 Pagination

A horizontal design concern across the entire API is the design of the pagination feature (F7). The established industry standard for pagination in GraphQL is the Relay specification [15]. Every connection to a type with a high cardinality is represented not as a plain list but using the pseudo-generic `Connection` type, on a field accepting pagination parameters. It allows querying for only the first n results with the `first` parameter. The Connection object then exposes *cursors* in the form of opaque strings on the `pageInfo` field and on each edge, which can be passed to the `after` parameter to query only edges after that cursor in a subsequent query. The UML diagram in Figure 2 shows the structure of the Connection type. GraphQL does not actually support generic types, so a type is generated dynamically for every instance of `Connection`. For brevity, the concrete Connection types and lines to the Connection type are omitted in Figure 1.

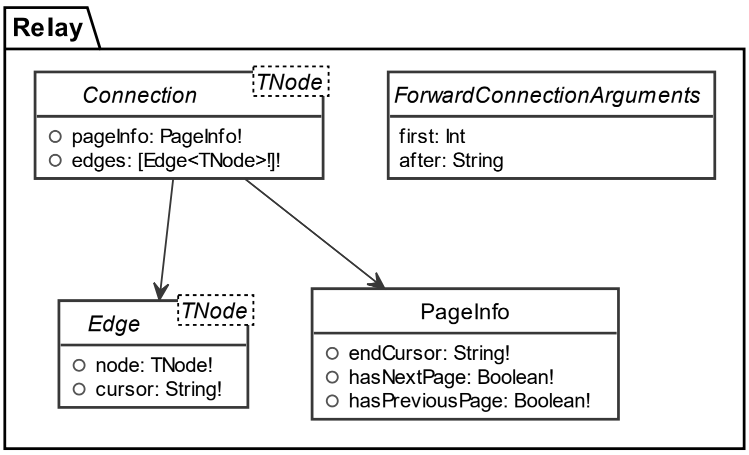


Figure 2: Conceptual GraphQL schema of the Relay specification.

The benefit of cursor-based pagination compared to limit/offset is that it is guaranteed that no element will appear twice, even if the data is changed in the background in between paginated requests. In addition, starting a table scan at a given ID encoded in the cursor can efficiently utilize the primary key index to find the start point, while starting at a numeric offset requires rescanning the table from the start and is thus much more expensive for the database.

5.2 Storage

While the GraphQL data schema is the most important for usability of the server, the data schema of the storage backend needs to be considered too.

We can distinguish between two principal areas of data: The basic repository data including version history and file contents, and the code smell data for these repositories and their versions. One possibility would be to store all this information in a database, which would allow atomic transactions over all data and ensure referential integrity on the database level. However, the version history and file contents are already tracked very efficiently by Git and an attempt to replicate this in a database would take a lot of effort, with the likely result of slower queries and larger storage size. A better alternative is to store version information and file contents as bare Git repositories in a known folder on disk and ensuring data integrity on the application level. With this approach, listing repositories is possible with a `readdir` operation from the file system. Querying information from a repository is done by executing the appropriate Git command with the repository as a working directory and parsing its output.

Code smell data on the other hand needs to be stored in a separate database. The main entities needing storage are code smells (in a commit) and code smell lifespans (across commits). In addition, we need to bundle these into *analyses* (to support F7) and store which commits were analyzed in an analysis. Figure 3 visualizes a normalized database schema for these.

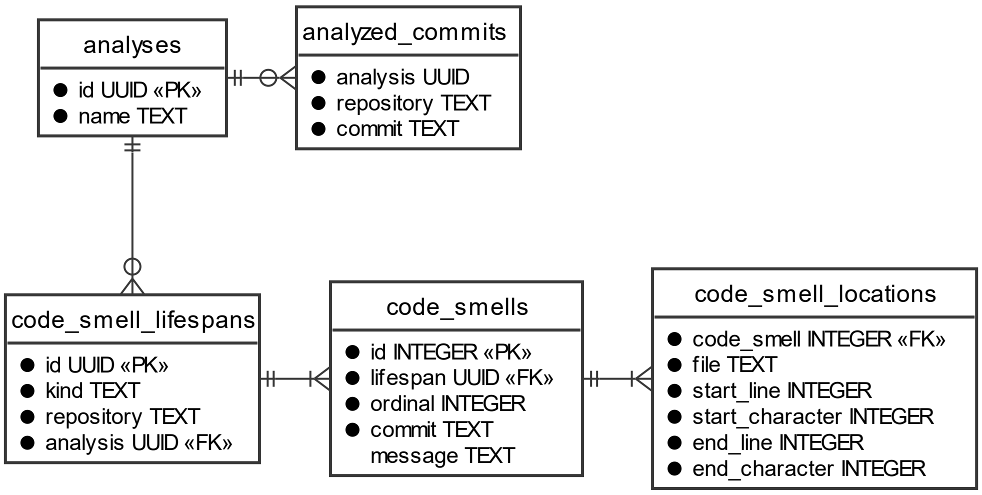


Figure 3: Normalized database schema.

In this schema, a code smell has exactly one lifespan and a code smell lifespan has $0..n$ code smells. Code smell lifespans store columns for information that is the same for all its code smell instances: The kind of the code smell, the repository it was detected in and the analysis that detected it. The ID of a lifespan is a universally unique identifier (UUID). This allows the client to pick IDs before uploading, which simplifies the API for associating code smells with their lifespans. Each code smell instance links to its lifespan through a foreign key. The order within the lifespan is stored as an integer in the column “ordinal” of the code smell table. Code smell instances are also associated to a commit through the commit ID, where each tuple of (code smell ID, commit ID) is unique within the code smell table. In opposite to the code smell kind, the *message* can (but does not have to be) different for each code smell.

Each code smell can have one to many locations in the code, which are stored in the `code_smell_locations` table following this schema. Each location is linked to its code smell through a foreign key.

There are no tables for commits and repositories, because these are stored outside of the database and referenced by strings. Code smell kinds are also not considered their own entity, but an attribute of a code smell. The server does not expose a way to query all available kinds.

The application will have to query the most by repository, commit, analysis, and kind. To prevent a full table scan for these queries, there are binary tree indices in the code smells table on the lifespan foreign key and the commit column. In the code smell lifespans table, there are three binary tree indices on the repository, analysis and kind columns. The analysis name, analyzed commit and analyzed repository are also indexed.

While this database schema ensures data integrity well, it was suspected that it will not perform well because of the number of needed JOINs. A common query like asking for code smells together with their kind and locations requires two JOINs. For illustration, Figure 4 contains a visualization of a query plan produced by PostgreSQL for batch-loading code smells and their kinds for 1,000 commits over a data set of 343,014 code smells. This query took 5.29 seconds to execute. The plan shows that the most expensive nodes are joining the lifespan and location tables, as well as grouping the locations for each code smell. The aggregation node at the very top is to group results for each commit for batch loading purposes (explained more in 6.3).

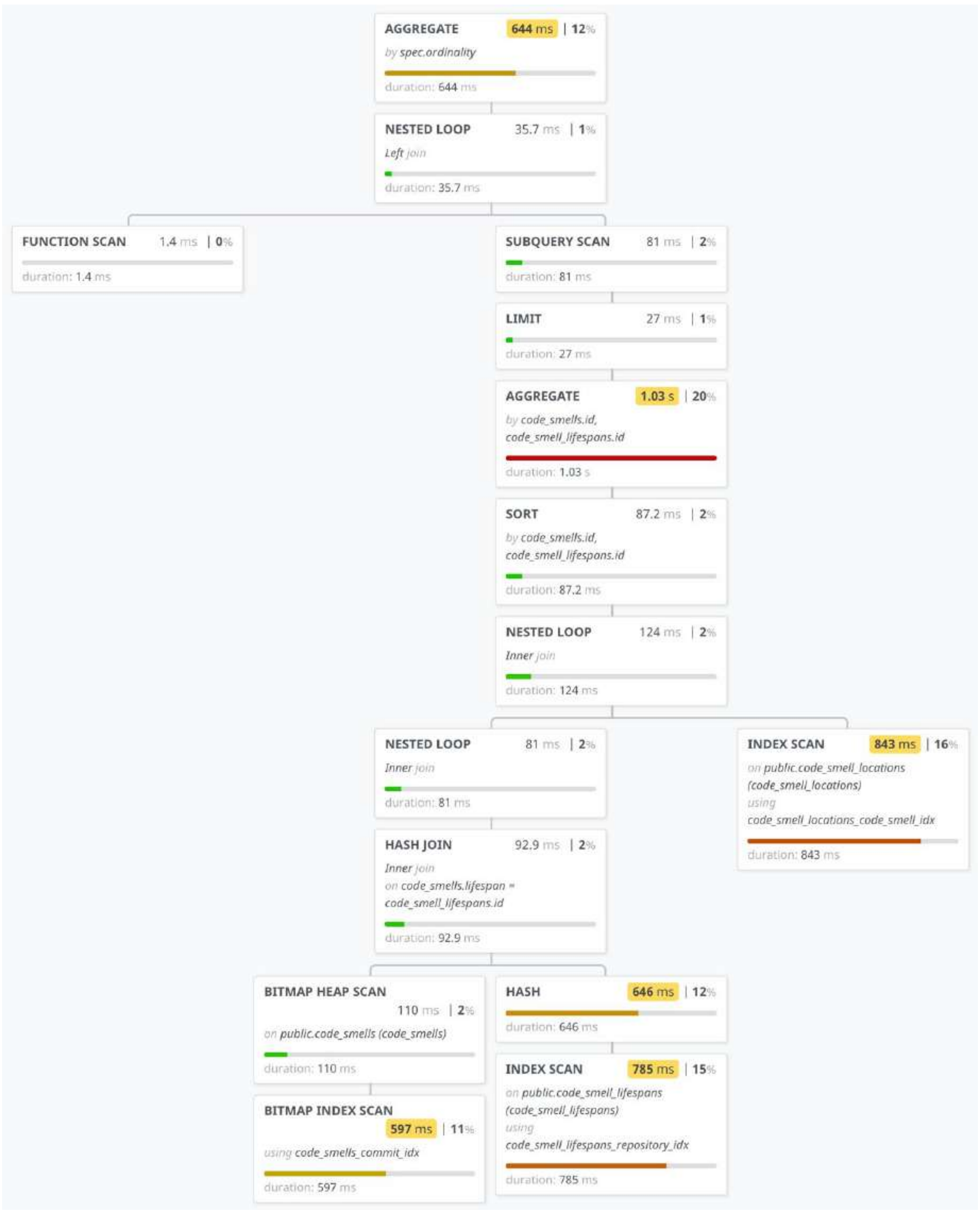


Figure 4: PostgreSQL query plan for a common code smell query in normalized schema.

Since the server later needs to scale to millions of code smells, the cost of these JOINs is problematic. To solve this while still maintaining data integrity guarantees, the tables could be merged into a materialized view. Alternatively, the tables themselves could be denormalized.

With a materialized view, the schema described in Figure 3 would still act as the always-consistent source of truth, but the materialized view would need to be refreshed after every insert. Refreshing a materialized view means truncating the outdated view contents, then recopying all data from the source tables into the materialized view. Unfortunately, this process would take magnitudes more time than the insert itself and unacceptably slow down the process of adding code smells to the server.

The alternative is to denormalize the tables themselves by inlining joined data into the respective tables. Figure 5 shows the updated denormalized schema using this approach.

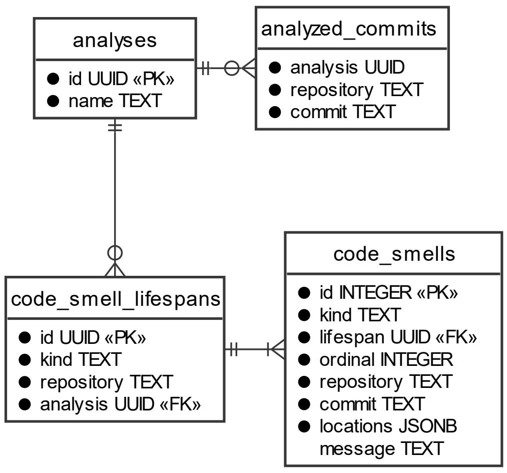


Figure 5: Denormalized database schema.

The kind, repository and analysis of code smells is now stored on both the lifespan and the code smell, to make sure both the horizontal view (F3) and the vertical view (F4) can be queried without JOINs. The obvious disadvantage of this approach is slightly increased storage size. In addition, the kind between the two tables needs to be kept in sync. Since the server has no requirement to expose a way to update existing code smells, this is not a concern and was deemed an acceptable trade-off.

Instead of a separate table, locations are stored as a JSONB column of the code smells table holding an array of location objects, including the file path and the range composed of the start and end position. The disadvantage of this would be a slower query when filtering by line and column in a file, as it is not possible to index properties of objects contained in a JSONB array. However, no such capability is exposed by the server, so it does not make sense to optimize for this scenario while sacrificing general query performance. Queries for code smells in specific files can still be kept fast by using a GIN index on the column.

Figure 6 shows a query plan for the equivalent query with the denormalized schema. There are no JOINs needed anymore, instead the necessary index scan and aggregation for batch loading the input commits are now the slowest nodes. This query finished in 1.89 seconds, about a third of the previous execution time.

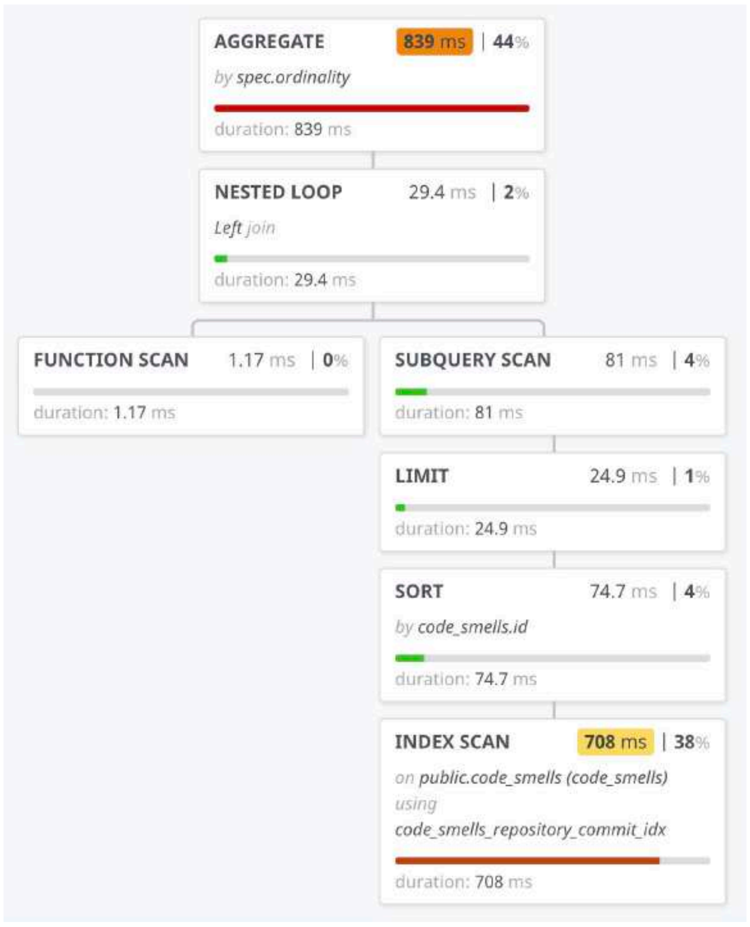


Figure 6: PostgreSQL query plan for common query with denormalized schema.

6 Implementation

This chapter addresses the implementation of the code smell server. As explained in chapter 4, the HTTP API server is implemented as a NodeJS application. Figure 7 gives an overview over the components of the system as a UML component diagram. The handlers for the different URLs are registered using the `express` library[6] in the server entry point. The exposed endpoints are divided into three routers: The GraphQL endpoint, the REST endpoints, and the Git upload handlers. These are implemented as separate routers in the `routes/` directory. The following sections go into more detail for each component and the components they depend on.

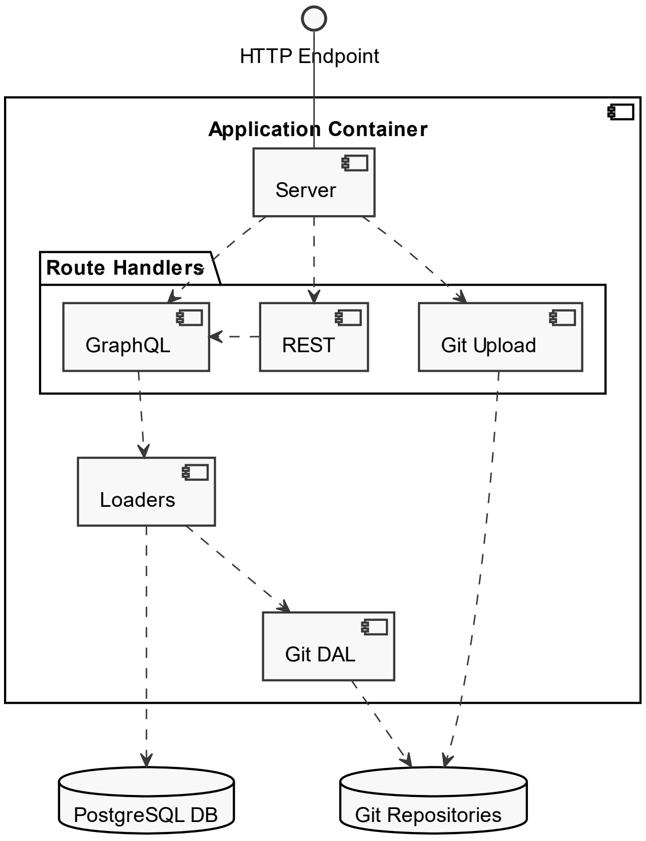


Figure 7: Implementation components.

6.1 Git Repository Upload

The Git upload endpoint executes Git's native CGI module `git http-backend`[7] to handle requests. This allows uploads through a simple Git push over HTTP after adding the

[6] https://expressjs.com/
[7] https://git-scm.com/docs/git-http-backend

server as a Git remote (example in chapter 7). The URL for the Git remote is the server address, with the path `/repositories/` followed by a chosen repository name and the suffix "`.git`". If the repository does not exist on a push or bundle upload, it is initialized ad-hoc.

The alternative upload method through Git bundles uses the same URL, except that the suffix is "`.bundle`". The bundle can be uploaded with a POST request, which stores it in a temporary folder, then uses `git fetch` to merge the Git objects contained in the bundle into the actual repository stored on disk.

Repositories are stored as *bare* repositories in directories on disk, meaning they only have versioning information and have no working directory. The root directory for the repositories can be customized through the `REPO_ROOT` environment variable.

6.2 GraphQL Endpoint

Two major libraries exist for writing GraphQL backends in JavaScript: The official reference implementation `graphql-js`, and Apollo. No relevant differences were found between the two. For the code smell server, the official reference implementation was chosen. The GraphQL server library is used to define the schema on startup. On a request, it takes care of parsing the GraphQL query and invoking *resolvers* for each field. Each type in the GraphQL API is implemented as a class with methods that serve as resolvers for each field. Each class holds a private data object of a domain-specific type that was returned by the `loaders` module.

6.3 Data Loading

Since every resolver is executed in isolation for each field and has no knowledge about the context it is queried in, a resolver for a type in a list may get executed n times for each item in the list. If the resolver does a database query for each item to query related data, that would lead to executing $n + 1$ database queries (with n being the number of items in the list), which would be detrimental to query performance. This is commonly referred to as the $n + 1$ query problem.

One approach to work around this is to eagerly load related data. While it comes at the obvious cost of sometimes loading data that is not used in the end, it is also not feasible with the data storage chosen for the code smell server that is split between the database and Git repositories on disk.

A better solution is using the `dataloader` library[8]. It buffers calls to load data within the same GraphQL request and dispatches all queries in batches. With `dataloader`, the data access layer of the code smell server is implemented as a collection of functions to batch-load multiple items, each wrapped in a `Dataloader` instance which takes care of the batching.

These functions have no requirement on how they acquire the data, which presents multiple options. For example, for database queries, the load function could use an ORM, or perform a raw SQL query. Using an ORM can save some code but can sometimes lack the flexibility to express complex queries. Additionally, they can make the actual SQL queries harder to predict and optimize. In the code smell server implementation, the load functions instead execute plain SQL statements on the server as an ORM does not provide enough of a benefit. The load function passes the specifiers of what is to be loaded into the parameterized SQL query as an array parameter, which performs a `LEFT JOIN` on that array to correlate it with results. These results are grouped back into result arrays using `GROUP BY` with the input index as the grouping criterion and `json_agg()` to aggregate the list for each input into an array of objects.

In the case of querying for paginated lists, the `LEFT JOIN` is marked to be a `LATERAL JOIN`, and the joined table is a subquery. Thanks to the `LATERAL` qualifier, the subquery has access to the input to project it onto the correct results. The subquery can then apply `LIMIT`, `ORDER BY` and a `WHERE` to limit the result to the desired page.

Listing 1 shows an example of a query that batch loads code smells for a life span while supporting pagination for each list of code smells. The input is passed in as a query parameter and contains an array of specification objects that specify for which lifespan to load code smells for, how many and which page.

[8] https://github.com/graphql/dataloader

```sql
SELECT json_agg(c ORDER BY c.ordinal) AS instances
FROM (
  SELECT
    ordinality,
    (input->>'lifespan')::uuid AS lifespan,
    (input->'first')::int AS first,
    (input->'after')::int AS after
  FROM ROWS FROM (unnest($1::jsonb[])) WITH ORDINALITY AS input
) AS input
  LEFT JOIN LATERAL (
    SELECT code_smells.*
    FROM code_smells
    WHERE input.lifespan = code_smells.lifespan
      AND code_smells.id > input.after
    ORDER BY id
    LIMIT input.first + 1
  ) c ON true
GROUP BY input.ordinality
ORDER BY input.ordinality
```

Listing 1: Example SQL query to batch-load code smells of a lifespan, paginated.

Cursor-based pagination is supported in this query by sorting by the IDs of the code smells. The `after` input field, if set, contains the ID of the last code smell of the previous page. The page is continued after it through the `WHERE` clause `code_smells.id > input.after`. The page size is set to the `first` value through the `LIMIT` clause. To note is that one additional code smell is requested than asked for by the client. This last code smell is stripped out in the application logic and used to determine the value of `PageInfo.hasNextPage`, a field required to be returned by the Relay specification (see Figure 2).

For data loaded from Git, loading is performed using Git commands that accept multiple inputs and parses the output to correlate it back to the input. For example, to get the details of multiple commits, the batched commit IDs are passed as command line arguments to `git show`, which accepts multiple revisions. The output is controlled through a custom format through `--format`, which the application then parses from the command output. The same command is used to get the difference between a commit and its parents.

6.4 REST Endpoints

For simpler use cases, there is also a fixed set of views exposed as REST HTTP endpoints each under their own URL. This can save on verbosity of having to write a GraphQL query, for example in contexts where encoding a multi-line GraphQL query into a JSON object to submit it to the server is not easily possible. Table 1 lists the available REST endpoints.

Route	Description
`/analyses`	List analyses
`/analyses/:name/analyzed-commits`	List analyzed commits of analysis
`/analyses/:name/analyzed-repositories`	List analyzed repositories of analysis
`/analyses/:name/code-smell-lifespans`	List code smell lifespans of analysis
`/repositories`	List repositories
`/repositories/:name/commits`	List commits of repository
`/repositories/:name/commits/:oid`	Get commit of repository
`/repositories/:name/commits/:oid/code-smells`	List code smells in commit (F4)
`/repositories/:name/code-smell-lifespans`	List code smell lifespans in repository (F3)
`/code-smell-lifespans/:id`	Get code smell lifespan by ID
`/code-smell-lifespans/:id/instances`	Get instances of code smell lifespan
`/code-smells/:id`	Get code smell by ID

Table 1: Supported REST endpoints.

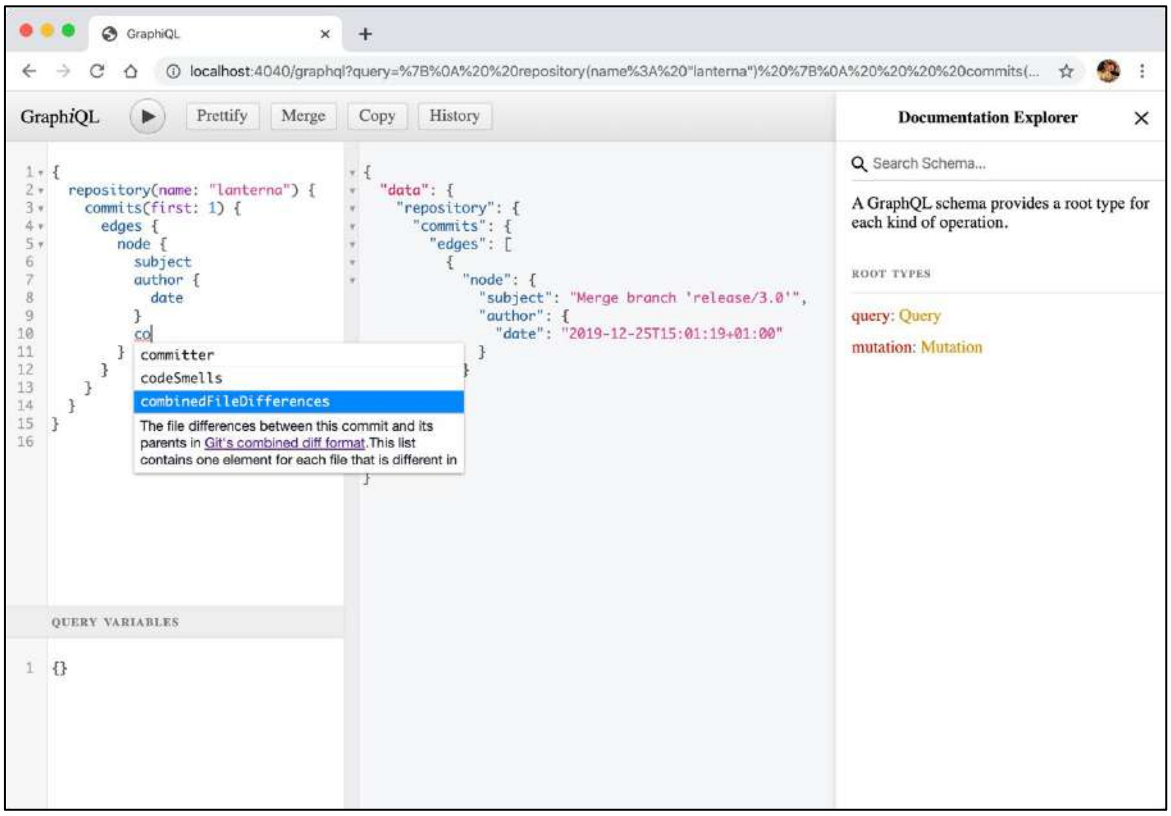
The REST endpoints are implemented by executing an internal GraphQL query and therefore come with a low implementation complexity cost. They otherwise go through the same data loading logic.

Pagination is implemented for these endpoints through standard Link headers as defined in RFC8288 [13]. Every paginated route supports the URL query parameters `first` and `after`. If there is a next page, a Link header will be set with the `next` relation pointing to the URL of the next page. This URL is identical to the current URL, except that it has the `after` parameter set to the cursor of last element of the current page. This allows supporting HTTP clients to automatically iterate all pages in a streaming fashion and moves pagination concerns out of the JSON response body.

7 Usage

This chapter walks through some example usage patterns of the code smell server, covering exploration, addition of data and querying of inserted data.

The first step to use the code smell server is to navigate a web browser to the URL `/graphql` of the server endpoint. Accessed in a web browser, this shows an interactive query environment. The editor on the left side allows to edit the query with autocompletion and help texts. Query variables can be set in the bottom left. The right side shows the result as formatted, syntax highlighted and collapsible JSON. In addition to autocompletion, clicking the "Docs" button in the top-right expands a panel that lists the documentation of every API object, including a search box. This enables a fast feedback cycle to find the right query that returns the data a user is looking for.



```
{
  repository(name: "lanterna") {
    commits(first: 1) {
      edges {
        node {
          subject
          author {
            date
          }
          co
        }
      }
    }
  }
}
```

committer
codeSmells
combinedFileDifferences
The file differences between this commit and its parents in Git's combined diff format.This list contains one element for each file that is different in

```
{
}
```

```
{
  "data": {
    "repository": {
      "commits": {
        "edges": [
          {
            "node": {
              "subject": "Merge branch 'release/3.0'",
              "author": {
                "date": "2019-12-25T15:01:19+01:00"
              }
```

A GraphQL schema provides a root type for each kind of operation.

Figure 8: Interactive query explorer.

Figure 8 shows a screenshot of the query editor. The query displayed asks for the subject and date of the latest commit in repository "lanterna." The result panel in the middle displays the JSON response, which precisely contains the fields that were requested.

The first query a researcher is likely to run when starting a research project is investigate whether the server already has code smell datasets applicable for the project. Listing 2 shows an example of a query for all analyses, with a small sample of analyzed repositories and detected code smell lifespans for each analysis.

```
{
  analyses {
    edges {
      node {
        name
        analyzedRepositories(first: 10) {
          edges {
            node {
              name
            }
          }
        }
        codeSmellLifespans(first: 10) {
          edges {
            node {
              kind
            }
          }
        }
      }
    }
  }
}
```

Listing 2: Query for analyses with samples of repositories and lifespans.

If the available data is sufficient, the user can proceed to query it for the sake of answering specific research questions as outlined in 7.2. If the existing data is not sufficient, they will have to produce a new dataset by running their own code analysis and adding the data to the server (for example with LibVCS4J).

7.1 Adding Code Smell Data

After producing a dataset through analysis, the researcher would first want to upload the analyzed repository to the server. The easiest way for this is to *push* the repository. The example in Listing 3 clones the repository “lanterna” from GitHub and pushes it to an example server instance under http://example-endpoint.org.

```
git clone https://github.com/mabe02/lanterna
cd lanterna
git remote add olfaction https://example-endpoint.org/git/repositories/lanterna.git
git push olfaction --all
```

Listing 3: Example command to push a repository.

After this, the repository is immediately available from the GraphQL API and is ready to have code smells referencing it.

Code smell data is added through GraphQL *mutations*, which are like regular queries, but cause side effects. Before being able to add code smell data, the researcher first must create a new analysis (if not appending to an existing one). This is accomplished through the `createAnalysis` mutation as shown in Listing 4 and can be easily done in the query explorer as well.

```
mutation {
  createAnalysis(input: { name: "my-analysis" }) {
    analysis {
      name
    }
  }
}
```

Listing 4: Example mutation for creating a new analysis.

Adding the code smell dataset is more likely to be done through a script, which could invoke the `addCodeSmells` mutation for every commit that was analyzed (even if no code smells were found, to record that fact). The script would pass the input containing the analysis name, repository name, commit and code smells as a typed *GraphQL variable*, which are attached to the JSON request body as a JSON object under the `variables` field.

```
mutation($input: AddCodeSmellsInput!) {
  addCodeSmells(input: $input) {
    codeSmells {
      id
    }
  }
}
```

Listing 5: Parameterized GraphQL mutation to add code smells.

Each code smell is associated to the lifespan it belongs to through a client-provided UUID. If a lifespan with the given UUID does not exist yet, it is created by the server with the information about the kind, repository and analysis. The order within the lifespan is given through the `ordinal` field. The created code smells are returned in the result of the mutation.

7.2 Querying Code Smell Data

After ensuring the server is filled with the necessary data, the researcher has the freedom to query any needed view on it to answer their research questions, through the query explorer or through a script.

To get a *vertical view* (F4) of the code smells throughout the repository history, a consumer can query code smell lifespans as displayed in Listing 6. For each lifespan of the kind "EmptyCatchBlock," this query asks for the instances of the lifespan throughout history, and in which commit that instance was detected.

```
{
  repository(name: "lanterna") {
    codeSmellLifespans(kind: "EmptyCatchBlock") {
      edges {
        node {
          duration
          instances {
            edges {
              node {
                commit {
                  oid
                  subject
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 6: Example GraphQL query for a vertical view.

A *horizontal view* (F3) can be achieved by querying the code smells in each commit as shown in Listing 7. As in Listing 6, the code smells are restricted to only the kind

“EmptyCatchBlock.” For each code smell, the query then asks for the commit of the code smell’s predecessor and successor, as well as the duration of its associated lifespan.

```
{
  repository(name: "lanterna") {
    commits {
      edges {
        node {
          codeSmells(kind: "EmptyCatchBlock") {
            edges {
              node {
                lifespan {
                  duration
                }
                predecessor {
                  commit {
                    oid
                  }
                }
                successor {
                  commit {
                    oid
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 7: Example GraphQL query for a horizontal view.

The returned data may then be aggregated in any way by a client-side script. Thanks to pagination resilient to mutations, this can also happen in a streamed fashion over a longer period of time. How the data should be aggregated heavily depends on the research question posed. Multiple such aggregations are presented as part of the evaluation with specific research questions in chapter 8.

8 Evaluation

The goal of this section is to verify whether the server is capable of aiding in answering real research questions. The focus of this evaluation is to assess the *functionality* of the code smell server. To accomplish this, a handful of research questions from the recent bachelor thesis “Evolution of Code Smells” by D. Schulz [6] were answered through the help of the code smell server, in the form of scripts that call the API:

RQ 1. How long is the lifespan of specific kinds of code smells?

RQ 2. How scattered are code smells in each commit over time?

RQ 3. Are files that contain at least one code smell changed more often than files without code smells?

The scripts could be written in any programming language that can make HTTP requests and parse JSON, such as Python, Groovy, Bash, JavaScript, PowerShell or others. For the evaluation, PowerShell was chosen, because its object-oriented pipeline makes it particularly easy to interact with web APIs returning JSON. It is open source and can run on the three major operating systems. Each script is described in detail in the following sections for each research question.

To test the server, it first needed to be filled with real code smell data. The most important aspect is to generate as much code smell data as possible in an acceptable amount of time. The *quality* of code smell detection matters less for the sake of the evaluation, as it does not influence whether the *server’s* capabilities are sufficient for the research questions.

For this reason, the analyzer PMD[9] was chosen to detect code smells. As it only performs syntactic checks (no type or data flow analysis), PMD is less precise than other analyzers, but due to its simpler nature it is also much faster than more advanced alternatives like the analyzers used by Schulz. Another benefit is that LibVCS4J [5], a Java library for repository mining, offers an integration for PMD. Together, LibVCS4J was used with the PMD plugin in a small Java program that walks all revisions of the selected repositories, analyzes them, and maps any findings to successors and predecessors across commits. This analysis was done in parallel over multiple physical machines, in batches of the commits to analyze. The resulting code smells and code smell

[9] https://pmd.github.io/

lifespans were then merged and uploaded sequentially into the code smell server (a truly parallel insert is not possible because of database locks).

The code smells that are checked for by PMD can be configured in a *ruleset*. The ruleset used for the analysis was inspired by the code smells researched by Schulz, as shown by Table 2 below.

Code smell kind in Schulz's thesis	Rule in PMD
God Class	`GodClass`
Long Method	`ExcessiveMethodLength`
Long Parameter List	`ExcessiveParameterLength`
—	`ExcessiveClassLength`
Cycle	–
Comments	–
Data Class	–
Switch Statements	–
Unused Code	`UnusedPrivateField`
Temporary Field	`SingularField`
Method Chain	–

Table 2: PMD ruleset.

Not all code smells researched by Schulz have an equivalent in PMD, and vice versa. The code smell "Data Class" is included in PMD v6, but LibVCS4J unfortunately only integrates with PMD v5 at the time of writing. Initially, the rule "LawOfDemeter" was also included, which aims to detect a similar anti pattern as the "Method Chain" detector. However, PMDs (likely simplistic) implementation of the rule caused the number of violations for it to be magnitudes larger than any other rule, presumably with many false positives. It was therefore excluded. "ExcessiveClassLength" was not researched by Schulz but is similar to "Long Method" and "Long Parameter List". "UnusedPrivateField" only covers a small subset of "Unused Code," as detecting the entire range of possible unused code properly requires analysis far beyond the syntactic checks PMD performs. For the same reason, there is no equivalent to the "Cycle" detector. Lastly, "Comments" and "Switch Statements" happen to not be implemented by PMD.

The repositories analyzed are a sample of 20 repositories of the repositories analyzed by Schulz, featuring a variety in number of files and commits. Table 3 lists the sample with additional metadata of each repository and number of findings. The amounts of detected code smells also show considerable differences, beyond the expected correlation to the number of files and commits. Especially the repository “checkstyle” stands out. With a high number of commits and files, as well as one of the highest numbers of code smells per file, it amounts to over 3.7 million code smells, which is more than all other repositories combined. “lombok” has the second-highest number of code smells per file and consequently total number of code smells. Meanwhile, “jna” also reaches a high code smell count despite having a small number of code smells per file, due to a long history and large number of files. In total, circa 5.1 million code smells were inserted in the database, from analyzing circa 24.6 million files.

Repository Name	Commits	Files in HEAD	Files across all commits	Code smells across all commits
arangodb-java-driver	1,438	366	322,993	29,704
awaitility	675	142	67,626	506
burstcoin	163	514	77,734	26,601
bytecode-viewer	294	245	94,635	25,495
checkstyle	9,010	2,837	13,191,256	3,716,252
Cleanstone	856	857	427,156	58,241
feign	806	283	127,048	6,567
flexy-pool	333	293	57,732	1,464
jackson-datatype-money	457	33	18,913	1,200
jacoco	1,667	867	938,394	19,210
jeromq	1,075	410	291,184	58,978
jna	3,724	1,114	3,755,382	224,099
jnr-ffi	966	333	219,405	32,048
jolt	392	398	88,798	784
junit-dataprovider	562	238	57,934	2,311
lombok	2,857	1,654	2,403,193	749,091
MutabilityDetector	1113	311	793,561	52,188
one-nio	220	291	46,961	7,399
randomizedtesting	912	512	314,641	13,526
rest-assured	1926	671	1,292,346	31,351
Total	29,446	N/A	24,586,892	5,057,015

Table 3: Analyzed repositories and found code smells.

Because it is of interest to see how the server performs in relation to the amount of data, all three evaluation scripts were run after inserting the analysis result for each repository, with their runtime measured. The measurements and the results of each script are described in the following sections.

The server and scripts were run on a dedicated server with the following specifications:

- **OS**: Debian GNU/Linux 10 (buster)
- **CPU**: Intel® Core™ i5-2400 CPU @ 3.10GHz
- **Memory**: 16 GB
- **Storage**: 256GB SSD

All three scripts are using pagination to not reach the memory limits of the server in the high numbers of code smells, explained in detail in the following sections.

8.1 Research Question 1

How long is the lifespan of specific kinds of code smells?

This question can be answered by querying the server for all code smell lifespans, specifically their duration field. To make the lifespans comparable across repositories, the durations are recorded relative to the total age of the project. The total age of the project is calculated by querying the first and last commit with their authoring date and calculating the duration between them. This is slightly different than Schulz's approach of counting the number of commits [6, p. 39], because it gives weight to the variable time between commits. The script calculates the relative age for each code smell lifespan as a number between 0 to 1. Like in Schulz's thesis, the fractions for each code smell lifespans are grouped by their kind.

Because all lifespan duration data would not fit into memory (neither of the client nor the server), the script makes use of pagination (F7) on the `codeSmellLifespans` connection. Since this connection is exposed on each repository and the script additionally needs to query the first and last commit, it first queries only the names of all repositories (unpaginated). For each repository, it queries the `commits` connection twice under two separate field aliases: one fetching only the first commit by paginating with a page size of 1, the other fetching only the last commit by paginating backwards with the same page size.

It should be noted here that fetching the last commit (paginating backwards), in opposite to fetching only the first commit, requires the server to iterate all commits. This is because Git's commit graph only stores *parent* references and a pointer to the *tip* of the history (HEAD), which means there is no way to determine the initial commit without iterating all commits. The backwards pagination is still useful, as it saves sending all the interim commits over the network.

Once the script retrieved the commit dates and calculated the total project age, it requests the code smells lifespans in pages, before moving on to the next repository. The page size can be varied depending on the available system resources on the server and client. There is no straightforward way to determine the ideal page size. In general, a higher page size is expected to perform better as it reduces the number of requests needed, but a page size too high can exceed memory constraints on client and server. For this evaluation, the page size was set to 700 lifespans. In total, 46 HTTP requests were issued to gather all data.

Like all scripts, this script was run repeatedly while filling the server after inserting each repository. The runtime of the script was measured for each run. Figure 1 shows the runtime at each run in relation to the number of code smell lifespans present in the database after inserting the respective repository.

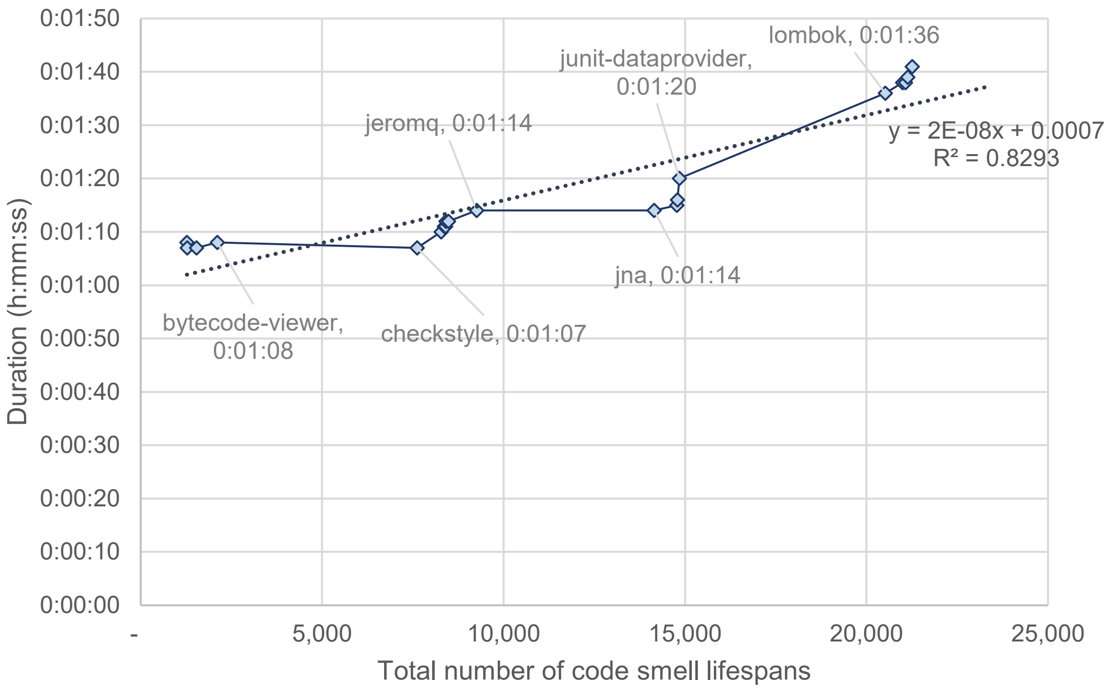


Figure 9: RQ1 script runtime by number of inserted lifespans.

We can generally observe that the runtime increases with the number of code smell lifespans, but in *stages*, at certain breakpoints. This leads to the suspicion that the runtime is not directly tied to the number of lifespans, but mostly influenced by the number of HTTP requests made as part of a run, as the number of requests has to increase every time the last page of lifespans overflows and requires a request for another page. This suspicion is confirmed by Figure 10, which demonstrates a much better correlation between number of HTTP requests and total script runtime.

This is a good property to observe, because it means that analyses can optimize runtime by parallelizing HTTP requests. The API server can be scaled horizontally, i.e. run with multiple replicas and a load balancer, to handle more concurrent requests.

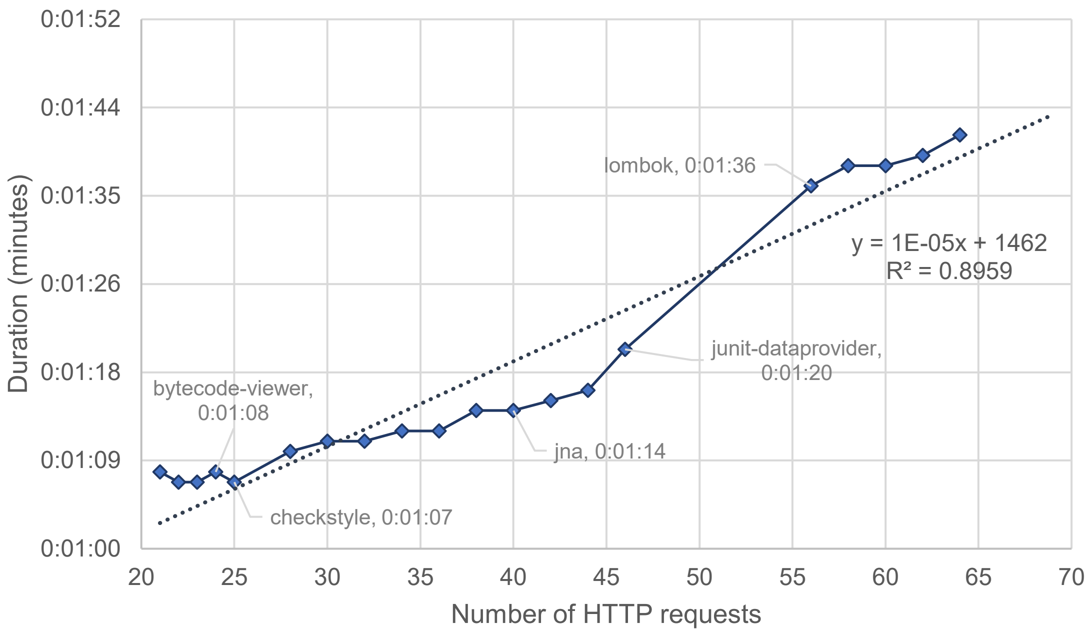


Figure 10: RQ1 script runtime by number of HTTP requests.

Deviations from the linear approximation can be caused by various reasons. Besides the pagination, PostgreSQL’s data storage and query execution are highly dynamic. Completely different execution plans may be chosen depending on the amount of data under operation. This could explain why the evaluations at the very beginning did not follow the linear progression as closely. There are also optimizations to the data storage that run in the background periodically (VACUUM) and can have positive impacts on query performance. Since there is a lot of network and file system IO involved, each runtime also contains an element of randomness that can also contribute to somewhat surprising deviations. An example of this is the run after the insertion of “checkstyle” at 25 requests/1:07, where the evaluation took slightly less time than at the previous run despite more data having been added.

The aggregated results of the scripts can be found in Table 4. This includes 29,446 code smell lifespans with a total of 5,057,015 individual code smell instances. Compared to Schulz’s results, all code smell kinds equally appear to have shorter lifespans. This could be attributed to a variety of differences in the measurement. The biggest one is the different approach to calculating the total project age, which means these percentages represent the fraction of *time* the code smell was present in the history, as opposed to the fraction of *commits*. Another potentially relevant implementation detail is that the server counts the lifespan duration as the time between the *first* and *last* known occurrence of the code smell, as opposed to the alternative of counting between the first occurrence and the commit that *removed* the code smell. This explains why all kinds in this evaluation have a minimum duration of 0%, which in this measurement corresponds to a code smell that only lived for one commit. With the alternative measurement, a lifespan of 0% would not be possible.

Kind	Count	Avg.	St. Dev.	Min.	Max.
ExcessiveClassLength	1,067	1.97%	13.90%	0.00%	100.00%
ExcessiveMethodLength	2,099	8.19%	27.43%	0.00%	100.00%
ExcessiveParameterList	1,373	1.02%	10.05%	0.00%	100.00%
GodClass	1,911	7.48%	26.32%	0.00%	100.00%
SingularField	2,540	3.43%	18.19%	0.00%	100.00%
UnusedPrivateField	12,130	3.57%	18.55%	0.00%	100.00%

Table 4: Lifespan of code smells by kind.

Equal to Schulz’s measurements, all kinds feature at least one code smell that spans the entire project history and generally a lot of outliers, as visualized in Figure 11. *God class* and *excessive method length* are also the two most persistent code smells from the researched set, but in Schulz’s results *long methods* live longer than *god classes*. This could be caused by a different configuration of the analysis – if the threshold for long methods is higher, a refactoring that removes *some* parameters may still not be enough to end the smell’s lifespan in his analysis, but may already have fallen below the threshold by PMD’s metrics. At the same time, with a lower threshold more instances would get detected (and not long after be detected as removed again), causing the code smell kind to appear more volatile and consequently show shorter lifespans. The same applies to *excessive method lengths*.

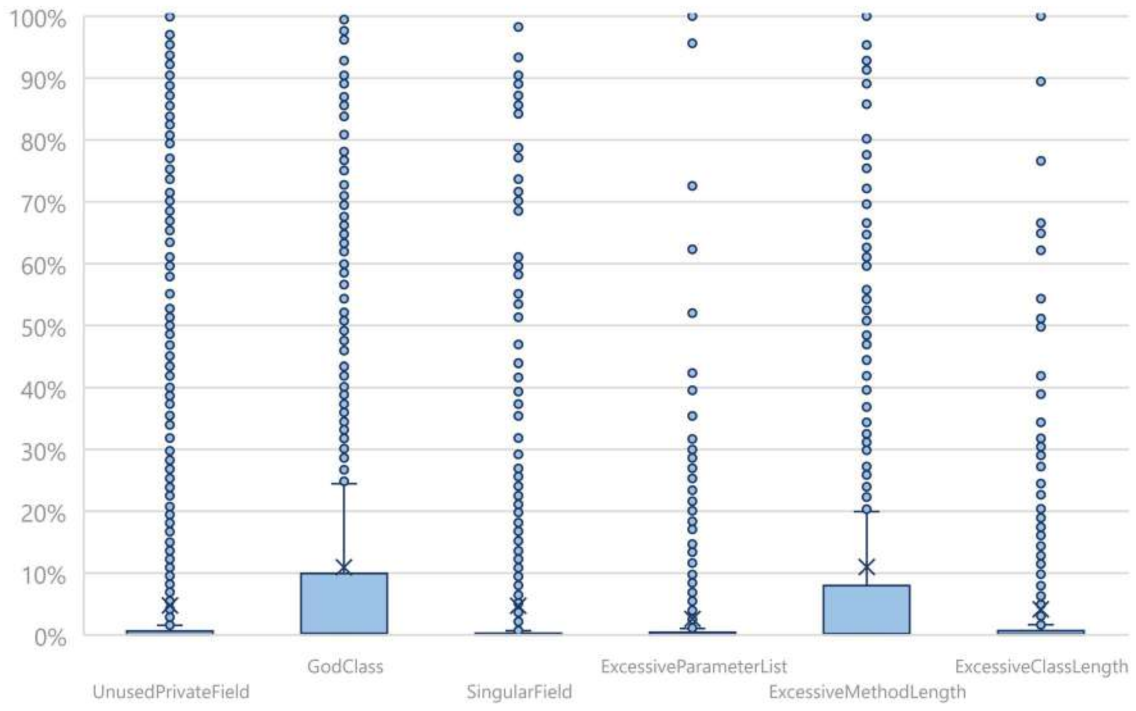


Figure 11: Lifespan of code smells by kind as box plot.

8.2 Research Question 2

How scattered are code smells in each commit over time?

Schulz defined the “scatter” of code smells as the pairwise *distance* between the file locations of all code smells in a given commit, where the distance is defined as the number of directory changes one has to walk to get from one file to the other [6, p. 3]. Further, he defined the “breadth” of a code base at a given commit as the maximum pairwise directory distance between all files that exist in the commit. For each code smell location (grouped by code smell kind) in each commit he calculated the pairwise directory distances and divided them by the maximum breadth of the code base at that commit. This fraction was defined as the “scatter index.” These scatter indices were then aggregated per code smell kind.

The data that needs to be queried for this analysis are all the commits of all repositories. To calculate the breadth, the query needs to include the paths of all the files at each commit. Furthermore, it needs the code smells at that commit with the file paths of their locations. This makes RQ2 the most data- and compute-intensive research question to answer. To handle this, it was split into two scripts: RQ2a gathers all the necessary data and stores it in temporary files, and RQ2b does the client-side computation. In opposite to other scripts, only the first part was run for each repository during insertion, as the client-side computation is independent of the server performance and would increase the evaluation time by magnitudes.

Figure 12 displays the runtime after each repository by the number of HTTP requests needed, with a particularly good linear correlation ($R^2 = 0.98$). In this evaluation, the script paginates over commits (i.e. always fetches all files and code smells). The page size for commits is initially set to 1 (a single commit with all its files and code smells), but adjusted dynamically depending on how many files and code smells were found in the previous page. This is a heuristics-based optimization to reduce the number of requests for ranges of commits with only few files or code smells.

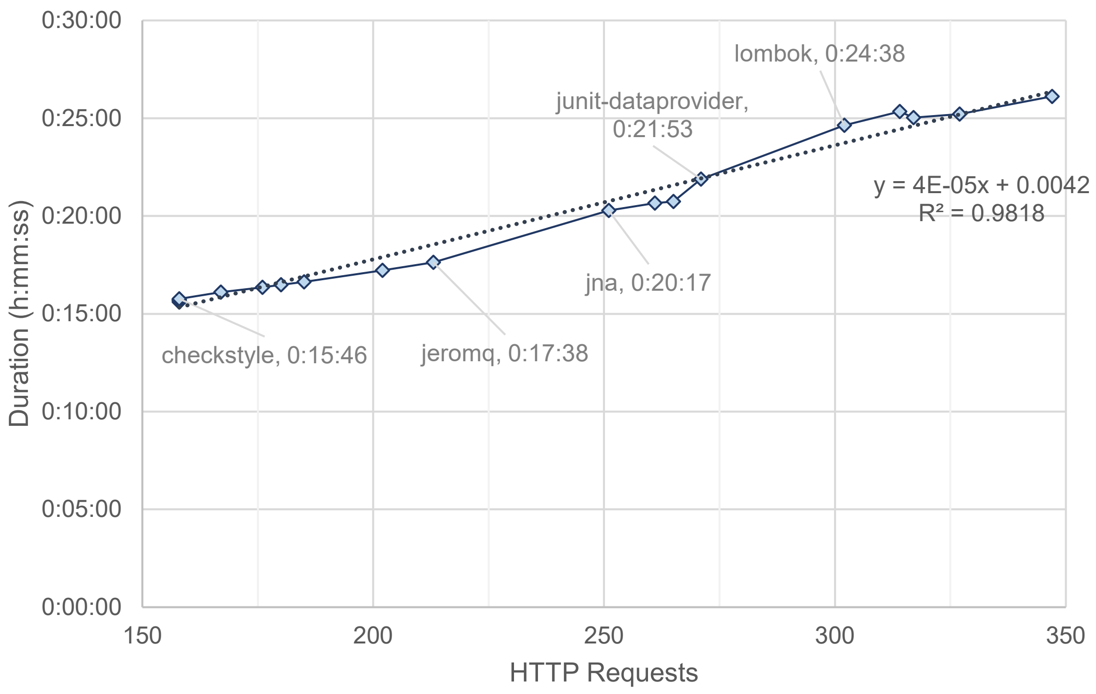


Figure 12: RQ2 script runtime by number of HTTP requests.

After saving all the results from the first script, the second script iterates over all commits and calculates the pairwise path distances between each file and each code smell location. This second part is very computationally expensive due to the complexity of the pairwise comparison of the file paths, but it is parallelizable. Similar to the seed script, this part was split into 297 jobs of 100 commits each that were distributed over multiple physical machines. Instead of PowerShell, this script was translated to

JavaScript, which showed a better performance in the CPU intensive task. All jobs were finished after 3h 21min and produced a total of 352,536,602 data points encoded in 11.6GB of CSV. The results of the final run can be found in Table 5 below.

Kind	Avg.	St. Dev.	Min.	Q1	Med.	Q3	Max
ExcessiveClassLength	0.44	0.28	0.00	0.12	0.61	0.63	0.96
ExcessiveMethodLength	0.46	0.27	0.00	0.23	0.46	0.71	1.00
ExcessiveParameterList	0.44	0.30	0.00	0.13	0.57	0.70	0.96
GodClass	0.45	0.26	0.00	0.21	0.38	0.69	1.00
SingularField	0.35	0.27	0.00	0.12	0.25	0.68	1.00
UnusedPrivateField	0.37	0.26	0.00	0.21	0.26	0.30	1.00

Table 5: Scatter of code smells by kind.

We can see that the detected code smells were scattered across a median 25% to 61% of their project. All kinds are generally less scattered than in Schulz's analysis [6, p. 46]. The differences are most likely due to the differences in PMD's detection approaches. Figure 13 visualizes the result as a box plot.

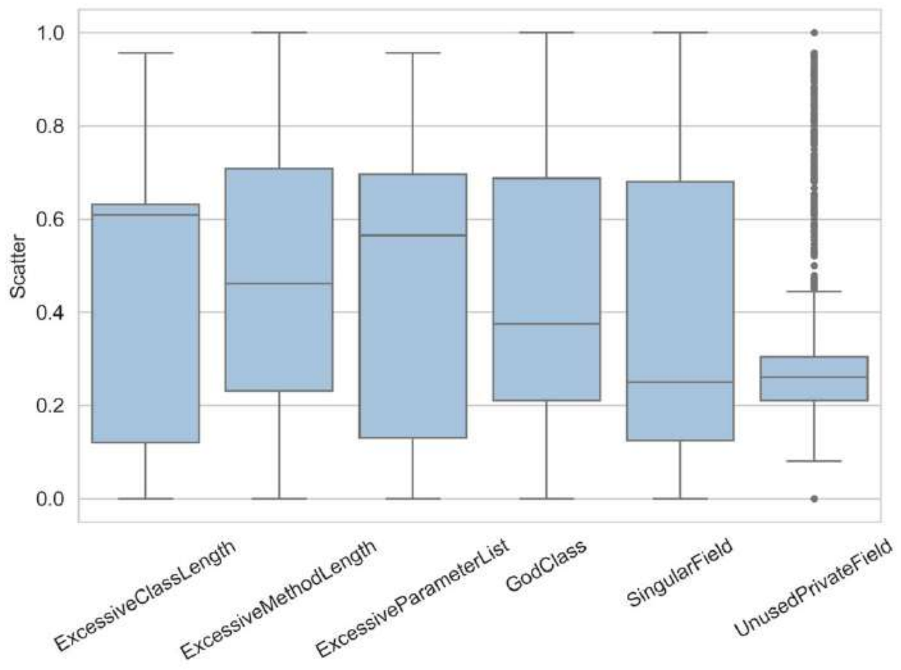


Figure 13: Scatter of code smells as box plot.

8.3 Research Question 3

Are files that contain at least one code smell changed more often than files without code smells?

To answer this question, the script needs to inspect the file changes of the commits in each repository, which is paginated with a page size of 1. Schulz constrained the list of commits to those referencing an issue number [6, p. 50]. This is possible with a regular expression passed to the `messagePattern` parameter of the `commits` resolver, which is queried by the script in pages of 1,000 commits. File changes are available from the `Commit` type in the `combinedFileDifferences` field, which contains the difference between the commit and its parents in a structure representing Git's default combined diff format [14, pp. 282-284]. We are interested in checking whether the file had any known code smells *prior* to each commit. The corresponding file in the parent commit is exposed on the `CombinedFileDifference` type through `baseFiles`. This is a list, since a commit can have multiple parent commits and a file therefore multiple parent files. For each parent file, using pagination, we query a single code smell to see if at least one code smell exists. For each commit, the script partitions the file differences by whether at least one parent file had at least one code smell.

Figure 14 again shows the script runtime after each inserted repository in relation to the number of HTTP requests, which again shows a good linear correlation ($R^2 = 0.98$). Since only repositories and commits are being paginated, this means the runtime scales linearly (but in stages due to page sizes) with the number of repositories and commits added.

The final run over all repositories took 11 seconds. This low runtime is mostly thanks to constraining the commits to only those that reference an issue, which reduces all 28,958 commits to only 468. However, given the liner scaling we can infer that this analysis would still perform well even for large data sets.

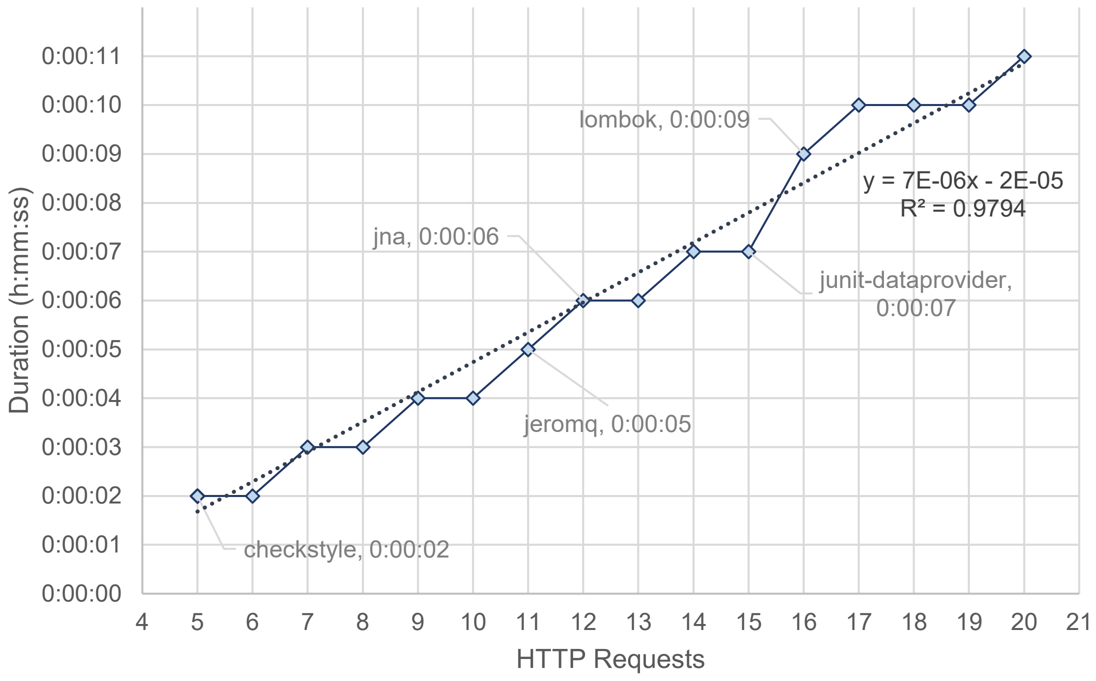


Figure 14: RQ3 script runtime by number of HTTP requests.

Table 6 shows the aggregated output of the script. The column "Sum" shows us the total number of file changes in each category. The average column tells us how many files of each category are changed per commit on average. Finally, minimum and maximum show the minimum and maximum number of files any commit changed with and without code smells.

File changes	Sum	Avg.	St. Dev.	Min.	Med.	Max.
With code smell	427	1.044	3.151	0	0	49
Without code smell	1,522	3.721	8.173	0	2	117

Table 6: Number of changes to files with and without code smell.

We can see that file changes to files without code smells are more than three times as common in our dataset. This leads us to the opposite conclusion of Schulz's result, which had 1.2x as many changes to files *with* code smells.

An important aspect that may explain this is that fewer code smell kinds were searched for in our analysis. This means overall, there is a smaller chance of a code smell being detected in a file, and therefore a smaller chance for a commit to change a file with a code smell. Especially the "comments" code smell Schulz analyzed is a code smell that is likely observed in almost every file, as indicated by Schulz's results for RQ2 [6, pp. 46-48]. This code smell was not analyzed as part of this evaluation, as

PMD does not include an equivalent rule. Additionally, Schulz partitioned file changes by whether the file contained a code smell at the respective commit, which means commits that *introduced* code smells are also counted. Besides these aspects, there is also a plausible explanation for this result: Files with code smells can be significantly harder to change than files without code smells and may receive less changes because of that.

Figure 15 visualizes the results as a box plot. Outliers were omitted for brevity. The lower quartile of changes with code smells is equal to the minimum and median of 0, which means that a quarter of all commits contained zero changes to a file with code smells. The third quartile is at 1, meaning that three quarters of all commits changed zero or one file(s) with code smells.

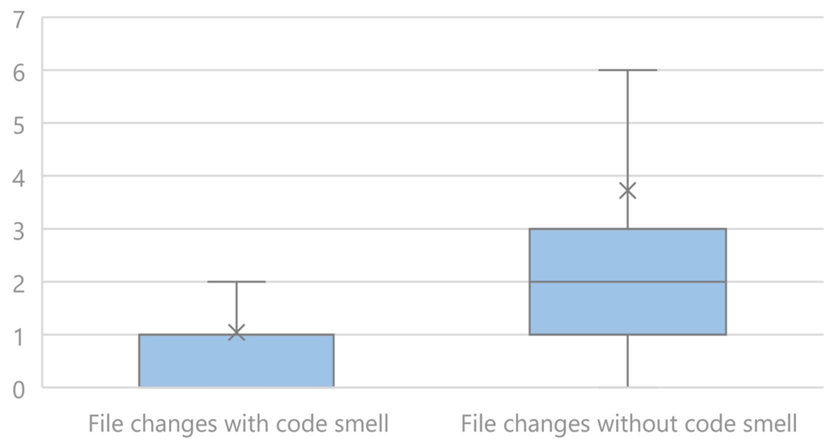


Figure 15: Box plot of changed files by code smell presence. Outliers omitted.

Figure 16 includes outliers, which shows that the number of files changed in a commit can have extreme outliers in commits that touch a sizable portion of the repository. The highest outlier in both categories is a commit[10] in “lombok.” It automatically refactored 117 files without code smells and 49 files with code smells to solve a referenced issue.

[10] https://github.com/rzwitserloot/lombok/commit/889c935ec9f0e45bba1e88b0f256e1f29a734f39

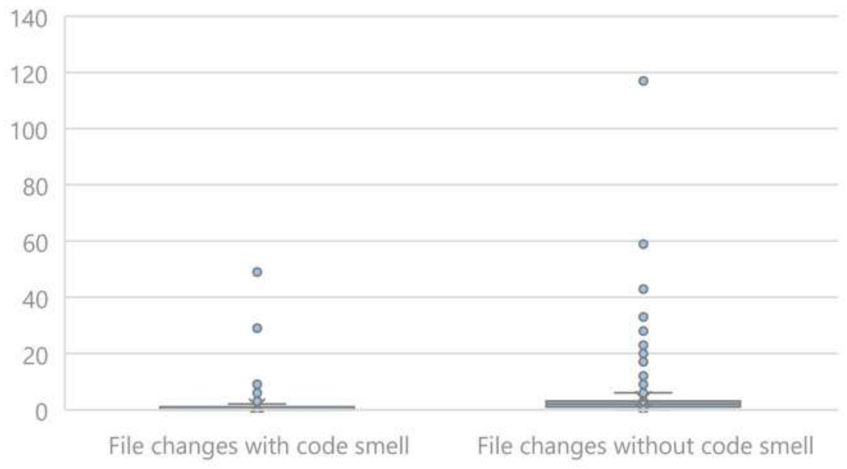


Figure 16: Box plot of changed files by code smell presence. Outliers included.

9 Conclusion

The goal of this thesis was to research whether a server for code smell data can be built to aid in research on code smells and their evolution. Making code smells data more easily accessible can accelerate research projects in the space, lower the bar for new research and enable evaluations of larger data sets than before.

A rich graph data model for code smells, their lifespans, and version control information was conceptualized for the server. A web API was implemented that exposes this rich data model, allowing users to perform flexible queries that suit their needs. This API was backed by a storage backend, which was tailored to the domain and expected access patterns.

To verify the built server can assist research in a meaningful way, an instance was deployed and filled with over 5 million real code smells and their lifespans. This data was gathered by running an analysis with PMD over more than 24 million files in circa 29 thousand commits of 20 distinct repositories.

The server was then used to answer three example research questions from a recent bachelor thesis [6], applying the same aggregations and visualizations to demonstrate the feasibility. By incrementally adding more data to the server and repeating the evaluation after each incremental addition, it was shown that the time required to run such an evaluation scales linearly.

The results of the evaluation in part showed significant differences to prior analysis by Schulz, for which potential causes were identified, including differences in the analysis tools used. This indicates that the server can also be used to compare different analysis tools with each other, which is another interesting application.

9.1 Outlook

As mentioned throughout the thesis, the built server provides a solid foundation for many potential future improvements, that can ease its use or open it up to more use cases.

Performance is an area that the server could improve, although it needs to be weighed against the cost in flexibility an improvement might entail. The most interesting avenue of exploration here would be the experimentation with alternative databases, that can store and query data in a way that matches the domain model more closely (e.g. graph databases) and that are built to handle enormous amounts of data.

The data model could also be modified to model a more complex relationship between code smell and version history, if tools arise that can make use of this. For example, a code smell could have $0..n$ predecessors and successors instead of $0..1$, which would make code smell lifespans a graph rather than a linear history. This would be closer to real data model of the Git version history, however tools like LibVCS4J [5] are currently not equipped to make use of this freedom. Changing this aspect would likely have severe impact on query complexity and, in turn, performance, which may necessitate the mentioned exploration of a different storage backend that is more suited to store this graph-like data.

It also makes other concepts harder to define, e.g. the duration of a lifespan is not clearly defined if code smells can “branch off.” A code smell may then be considered “removed” in one branch, but still existent in another. In reality, the analysis would then want to treat one of the branches as the “main” branch, which means from the perspective of the analysis, lifespans *are* linear again. Since the existent code smell evolution research usually considers only a linear history [1] [2] [6] (although sometimes constrained by the limitations of SVN), the current model can be seen as sufficient for most research use cases.

Feature-wise, one can imagine a variety of additional query capabilities, e.g. the ability to query code smells affecting a specific line in a file.

To facilitate more users and to open up an instance to the general public, the server would eventually need an access control system that regulates write access to the stored data.

10 Works Cited

[1] J. Harder, "How Multiple Developers Affect the Evolution of Code Clones," in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, 2013.

[2] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells using Software Repository Mining," in *2012 16th European Conference on Software Maintenance and Reengineering*, Szeged, Hungary, 2012.

[3] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort," in *IEEE Transactions on Software Engineering*, 2012.

[4] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[5] M. Steinbeck, "LibVCS4j: A Java Library for Repository Mining," in *20. Workshop Software-Reengineering und -Evolution*, Bad-Honnef, 2018.

[6] D. Schulz, „Evolution von Code Smells,“ University of Bremen, Bremen, 2019.

[7] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk and A. De Lucia, "Landfill: an Open Dataset of Code Smells with Public Evaluation," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy, 2015.

[8] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk and A. De Lucia, "Mining Version Histories for Detecting Code Smells," in *IEEE Transactions on Software Engineering*, 2015.

[9] A. Lozano, M. Wermelinger and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, New York, NY, USA, 2007.

[10] V. Lenarduzzi, D. Taibi and A. Janes, "How Developers Perceive Code Smells and Antipatterns in Source Code: a Replicated Study - Raw Data," 01 06 2017. [Online]. Available: https://data.mendeley.com/datasets/8n6k8dfw2f/1. [Accessed 27 01 2019].

[11] Microsoft, "Language Server Protocol Specification - 3.15," 24 07 2019. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/specification-current/. [Accessed 27 01 2019].

[12] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," University of California, Irvine, Irvine, CA, 2000.

[13] M. Nottingham, "Web Linking," RFC Editor, 2017.

[14] S. Chacon and B. Straub, Pro Git, vol. 2, New York: Apress, 2014.

[15] Facebook Inc., "GraphQL Cursor Connections Specification," [Online]. Available: https://facebook.github.io/relay/graphql/connections.htm. [Accessed 15 02 2020].