Code Panorama

Bachelor Thesis

Department of Computer Science
University of Applied Sciences Rapperswil

Fall Term 2019

Authors:	Patrick Bächli, Marc Etter
Advisor:	Prof. Dr. Farhad D. Mehta
Project Partner:	IFS (Institute for Software, HSR)
External Co-Examiner:	Tom Sydney Kerckhove
Internal Co-Examiner:	Prof. Frank Koch

Abstract

Objective

CodePanorama is a tool for software developers, reviewers, and consultants. Its goal is to assist in identifying points of interest within a code-base to review. A software developer might join a new project and want to quickly find the most interesting parts of the code to get started. A supervisor must review the results of a project but does not have the time to look at the entire code-base. Instead, they look to CodePanorama to make an educated guess as to where their efforts should be focused.

Procedure / Result

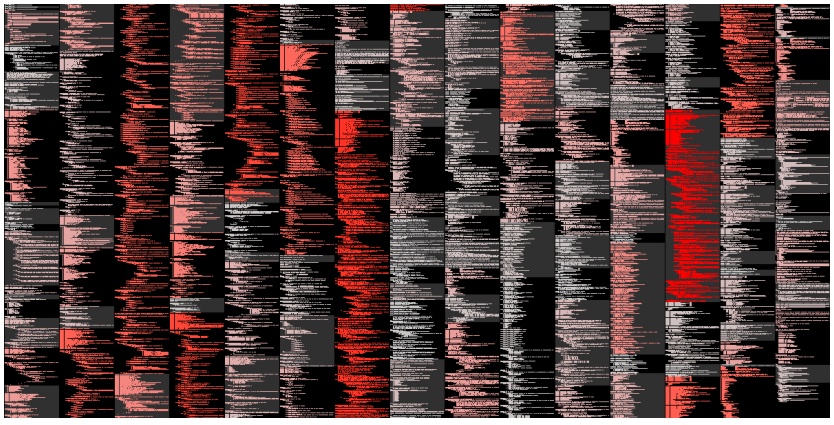
In contrast to other code metric tools, CodePanorama is designed to provide the user with a non-reductionist, "zoomed-out" overview of the entire code-base. It is up to the reviewer to find interesting patterns and curious anomalies based on indentation, spacing, line lengths, and color overlays, instead of the usual metrics.

After entering the URL to any git repository, CodePanorama will clone the repository in the background, and generate the panorama view. The result is basically a collage of all files in the repository, glued together.

Often, this new perspective on a code-base can find patterns such as duplicated code, excessive indentation, or any other feature the human eye might recognize. With the addition of color overlays, a reviewer might find intriguing correlations between git statistics, such as change frequency, and code layout.

Once such a feature has been identified, CodePanorama offers the functionality to simply click on a section of the panorama image. This allows the user to directly dive into the actual code at that location. From there, they can review the code in place, or just take a peek before switching to their tool of choice.

Figure 0: Code panorama of CodePanorama with change frequency (~9k lines)



Management Summary

Problem statement

Determining the quality of software code is hard. Several tools exist to assist in doing so, by providing mathematical formulas and metrics. This is useful to get an overall idea of the quality of a project, but provides very limited help in getting a sense of exactly which parts of the code have problems. CodePanorama implements a new approach utilizing the human ability of pattern recognition in images to detect repetitions, correlations and anomalies inside code.

Target audience

CodePanorama is mainly targeted at users tasked with reviewing medium-to-large amounts of unknown code on a regular basis. Such users include project supervisors, examiners, and consultants.

Approach

CodePanorama generates a panorama view of any project's code, by "gluing" together all files into a single large "poster". By "zooming out" of the code to a distance where only the silhouette of each file remains recognizable, new patterns start to emerge. With a bit of practice and programming experience, a user can quickly identify unexpected patterns. To further assist the user, color overlays can be selected, showing for example which parts of the code are being changed most often.

Results

CodePanorama is publicly accessible[1] for anyone to analyze their project of choice. For more sensitive environments, we provide instructions on how to run CodePanorama on your own infrastructure for internal usage. Currently, CodePanorama includes some filters for which files to include in the panorama view and a selection of color overlays.

Perspectives

We are looking forward to using CodePanorama in business environments and receive user feedback. There are an almost unlimited number of additional features, ideas and improvements for further development. Apart from adding new types of filters and overlays, we would like to further improve the panorama image to be more comprehensible. Finally, we would like to add a user management system. There, users could see all projects they previously analyzed in a user-specific dashboard view.

[1] `https://codepanorama.io`

Lay Summary

Whenever a piece of software is created, both the developers and the stakeholders are interested in how "good" (or "bad") the produced software code is. Naturally, there has been a lot of research on how to measure the "goodness" or "badness" of code. The most established methods use mathematical formulas and other procedures to assign numbers to various aspects being measured. This allows for easy comparison between different parts of the code, or even with code of other software products.

CodePanorama suggests a different approach: By generating images from the program's code, we create images specific to the inspected software product. Using patterns and colors in the image, information about the code's structure can be gained. More importantly, outliers can easily be identified, as they visually do not fit in with the rest of the image. Such outliers could point to problems in the code, or simply to bad discipline by the authors.

We do not intend to replace or obsolete already established methods of analyzing code. Rather, we aim to provide a new angle on an existing problem, in an attempt to make a hard task easier.

Contents

1 Introduction

1.1 Project description

Several software quality metrics (lines of code per class/method, cyclomatic complexity, etc.) exist to estimate the quality and volume of large code bases. All metrics are reductionistic: They compute a number, or a set of numbers from a much larger amount of code. Although such metrics offer a quick overview of some aspects of the code, a more thorough code review is often needed in order to arrive at a more accurate estimation of code quality. For large code bases, such a code review can only be done on (often random) samples of the code.

As part of a study project in the fall of 2018, a prototype was built to explore the following idea: In case it was possible to visualize the entire (or a large part of the) code-base on one large surface (screen, poster, etc.), it could be possible to identify patterns or areas of the code as candidates for further inspection. In case version control is used, such a visualization could also provide historical and programmer-based information that could be relevant to such a further inspection.

As part of this bachelor thesis, this idea is to be pursued further and the prototype is to be developed and refined into a full-fledged and mature software tool.

1.2 Goals

The main aim of this project is to build upon the work already done for `code panorama.io`. A mature and widely usable software tool is to be designed and developed. This tool shall visualize the code contained in a git repository in varous ways in order for a developer to review it more effectively. In particular, the application must satisfy the initial goals set out for the study project (reproduced verbatim below) [BE19].

The application must:

1. Be able to filter the files visualised on the basis of file type, path, length, age, or other yet to be discovered quality relevant properties.
2. Be able to highlight lines of code on the basis of author, number of changes, age, or other yet to be discovered quality relevant properties.
3. Be easy and intuitive to use for a software developer or reviewer.
4. Have an attractive user interface.
5. Support user and repository management (possibly with encryption) for working with private or confidential repositories.
6. Suggest an appropriate size such that all the code that is required to be visualized can be shown, possibly using multiple pages, scrollbars, or another appropriate mechanism.

7. Offer the possibility to be easily (i.e. installation time $<$ 10 mins) run locally (for instance using readily downloadable docker containers), or deployable on a local server.
8. Be maintainable and easily extensible (programatically), in particular for new types of filters or highlighting.
9. Use a CI/CD pipeline for development and deployment.
10. Use a Haskell-based toolchain as far as possible.
11. Be able to be effectively used for reviewing project code at the HSR.

The following tasks summarize some of the next areas in which the application should be developed during the bachelor thesis project in order to satisfy the above goals:

1. Incorporate user-study feedback into its look-and-feel.
2. Implement the most frequently requested features by users.
3. Provide a complete user manual.
4. Scale properly to various desktop display resolutions.
5. Provide a way of handling codebases that cannot fit on one screen.
6. Be able to filter the parts of the code to be included in the panorama in a more fine-grained manner.
7. Be able to highlight code based on file type, user, number of changes, age, or other yet to be discovered quality relevant properties.
8. Be maintainable and easily extensible (e.g. new types of filters or highlighting, etc.).
9. Provide thorough technical documentation, so the project could be open-sourced or transferred to other developers with no prior knowledge.
10. Allow users to save and resume previous analyses with an integrated user management.

The students are also required to perform a literature review on the work already done in this area at the beginning of this project in order to determine the most effective strategies of filtering, highlighting and visualization in this domain, as well as judging the utility and other possible applications of such a tool.

1.3 Motivation

The original inspiration for this project came from a talk held by Laura Kovás at an ETH workshop on 13 October 2017.[1] There, a partial "panorama" of the *Vampire Theorem Prover* [Kov17] was shown. While the image already looked quite good, the process to generate it was rather tedious. It involved splitting the source-code into chunks small enough to not crash the LaTeX-compiler, which could only handle 180 pages at a time. Subsequently, all the generated PDFs had

[1] `https://www.sri.inf.ethz.ch/workshop2017`

to be merged together into a single page with yet another script.[2] This project aims to automate these steps as much as possible and provide an easy-to-use application everyone can run to generate panoramas of their code-base(s).

[2] F. Mehta (personal communication, 26 September 2018)

2 Related Work

2.1 Code-map metaphor

SeeSoft was a pioneering work by Eick et al. from the 1990s [ESS92]. It was one of the first tools that tried to find a solution to the problem of software visualization. The original publication has since been cited over 800 times and remains one of the fundamental works in this area.

Figure 2.1 shows a screenshot of *SeeSoft*, visualizing several files containing over five thousand lines of code. The different colors indicate different code age, where red depicts recently modified code and blue depicts code that has been left unchanged for some time.

Figure 2.1: SeeSoft — Visualizing program code changes

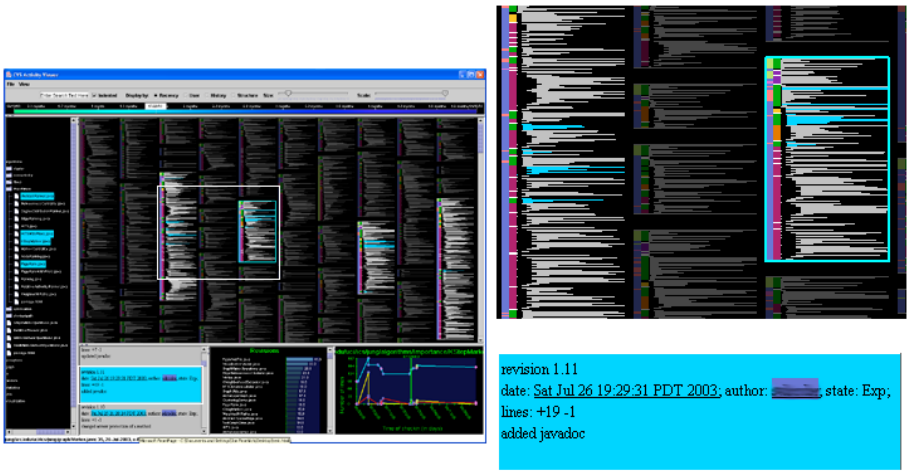
This kind of visualization has been named “code-map metaphor”, which is defined as follows: “The mapping of source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled down representation of text, in order to allow stakeholders to comprehend various statistics collected at the level of detail of individual lines of code.” [BNK17].

Besides *SeeSoft*, Bacher et al. reviewed 21 different implementations of this metaphor and concluded that it is generally a good way of visualizing source

code, since the zoomed out representation is a direct mapping to it, which in turn seems to yield high levels of trust on behalf of the users. However, they also admit that "[...] to date, little to no quantitative data exists in the literature that supports the claim that the use of the metaphor can faciliate the process of software development.". This is partly owed to the fact that previous works either did not carry out their own evaluations, or did not provide the data on which the presented findings were based.

One of the more interesting implementations is *Augur* by J. Froehlich and P. Dourish [FD04]. *Augur* explored the idea of combining information about both artifacts and activities in one visualization. Figure 2.2 shows a concrete example of this concept. The left side shows multiple displays containing various information about the structure of the whole system and the individual files as well as graphs and details about the activities. On the right side is a zoomed-in view of several files, which are annotated with details about authors, code structure and check-in dates. This information is made visible by coloring the individual lines, but also through the use of additional columns next to the files.

Figure 2.2: Augur — Multi-panel interface and detail view



Several case studies had been conducted to assess the usefulness of this visualization. While the results were deemed far from conclusive, they were still generally supportive. Froehlich and Dourish further concluded that using the already existing spatial structure of the source code and annotating it as seen above, is an effective way of "stitching together" activity and artifact information in a single view.

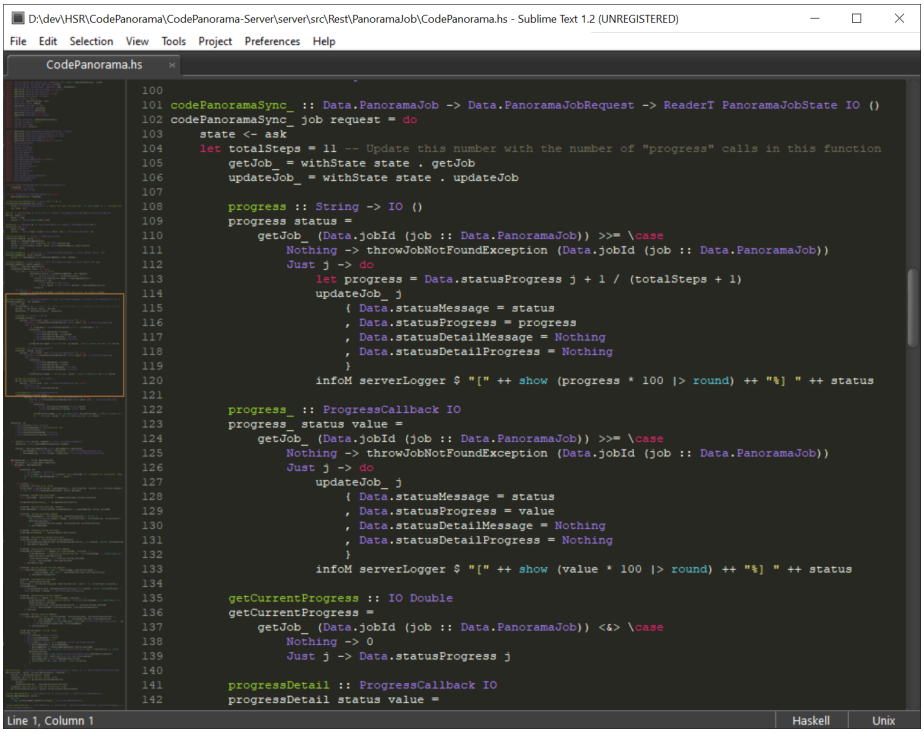
2.2 Code mini-map

The code-map metaphor was originally used for visualizing an entire codebase [ESS92]. Meanwhile, a related visualization arose around 2009: the "code mini-map". Bacher et al. define the code mini-map as follows: "The code mini-map visualization is based on the code-map metaphor and acts as an overview

component mapping source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled down representation of text, presenting viewers with a zoomed out view of the currently open source code document showing the layout of the code." [BMK18].

According to Owen Searls [Sea18] (and other community websites), the code mini-map was first introduced in *Sublime Text*[1]. The earliest version of *Sublime Text* we could find available for download is 1.2, released in June 2009 [Ski09]. This version already included the code mini-map on the left side of the window as a means of navigating through an opened file (see figure 2.3).

Figure 2.3: Sublime Text 1.2 — Code mini-map



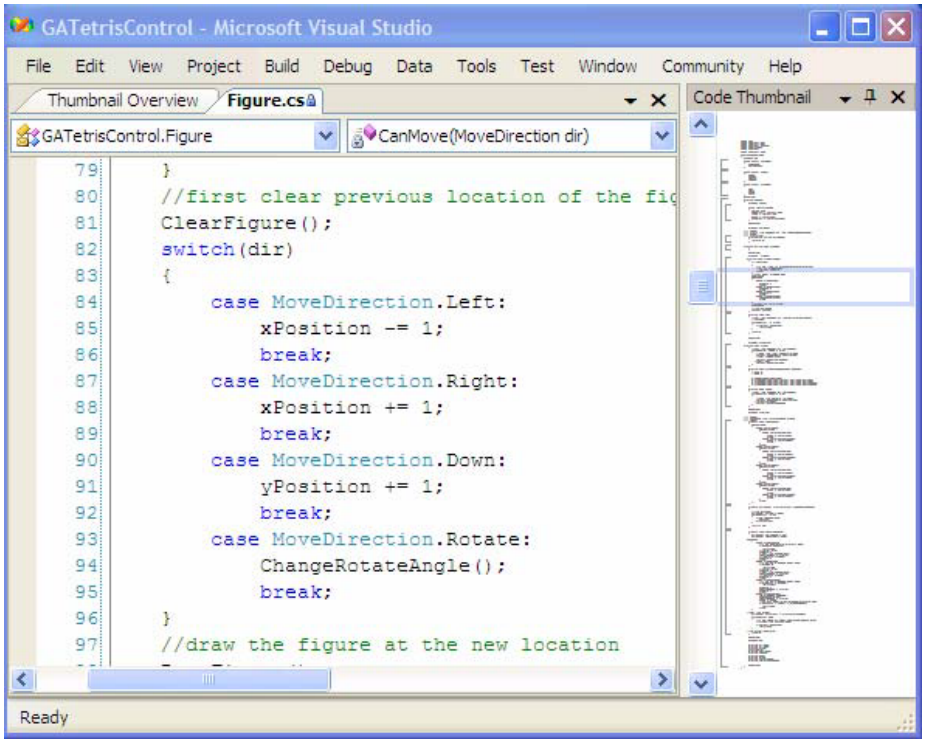
However, a paper published in 2006 by Deline et al. [Del+06] already mentions the idea of a "Code Thumbnail" in *Microsoft Visual Studio*[2]. This Code Thumbnail is precisely what was later named a code mini-map (see figure 2.4).

Not only does the design proposal by Deline et al. include a code mini-map, it also includes the "Code Thumbnail Desktop". The Code Thumbnail Desktop shows thumbnails of all files, allowing the user to easily navigate between files based on spatial memory and visual cues, rather than cognitive memory (i.e. remembering file- and symbol-names). This view was heavily inspired by the original work on *SeeSoft* [ESS92] and *Data Mountain* [Rob+98].

Unfortunately, the study could not conclusively prove a meaningful increase in productivity. However, Deline et al. were able to show "[...] that users are

[1] `https://www.sublimetext.com`
[2] `https://visualstudio.microsoft.com`

Figure 2.4: Microsoft Visual Studio 2005 — Code Thumbnail



quickly able to learn to navigate using thumbnail images of the code" and that "They enjoy this style of navigation both by their navigation choices during programming and search tasks and by their subjective ratings." [Del+06].

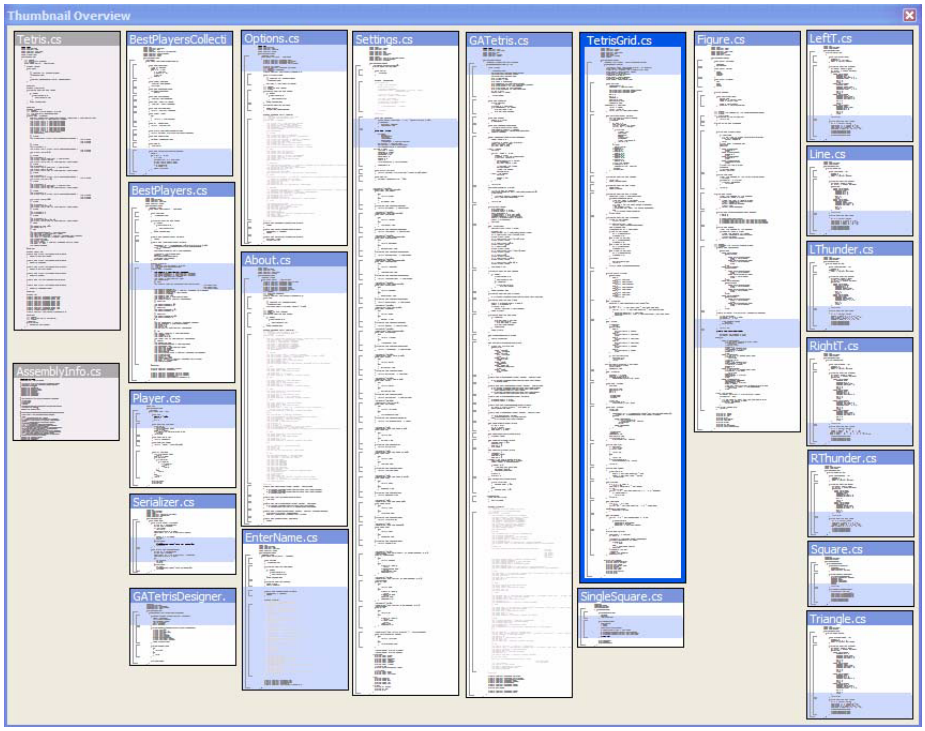
2.3 Comparison

Both the code-map metaphor and the code mini-map use a "zoomed-out" view of the code as the primary means of visualization. In both cases, information about physical proximity, indentation, and spacing are preserved. This provides the user with a simple overview of the code.

The main difference between the code mini-map and the code-map metaphor visualizations are the intended use-cases: The code-map metaphor in general was originally conceived as a tool to help in visualizing a large amount of code in a well comprehensible manner for reviews and overlaying software [ESS92] or organizational metrics [FD04].

The code mini-map, on the other hand, is tailored to real-time development assistance and code navigation from within the text editor. Symbols and colors are used to annotate lines and sections with information relevant in understanding the code [BMK18], rather than assessing the code's quality.

Figure 2.5: Microsoft Visual Studio 2005 — Code Thumbnail Desktop



2.4 Static code analysis

Traditionally, code is assessed using code metrics. Such metrics usually parse the code in question statically (i.e. without executing it) and use tailored algorithms to compute a single numerical value. The code is then deemed "good" or "bad", depending on whether the metric exceeds the configured thresholds. Examples of such metrics include cyclomatic complexity [McC76], coupling between objects [CK94], and comment-to-code ratio [OH92].

Popular tools to perform static code analysis include *SonarQube*[3], *Microsoft Visual Studio*[4], and *Codacy*[5].

These code metrics are (intentionally) inherently reductionistic. They help in quickly assessing and comparing code, but provide little value in identifying the origin of the problems. With CodePanorama we want to provide a completely language-agnostic tool without any reductionistic metrics, which is why we do not make use of code metrics. If a user desires to visualize their favorite code metrics, this is still possible through the use of custom overlays, as described later in subsection *Custom overlay* of section 3.3.

[3] `https://www.sonarqube.org`
[4] `https://docs.microsoft.com/visualstudio/code-quality/code-metrics-values`
[5] `https://www.codacy.com`

3 Methods

This chapter details the conceptual, organizational, and technological approaches used in creating the CodePanorama application.

3.1 Technologies

Many of the technologies we used during this project are based on personal preference. We do not claim that these tools and frameworks are the best at accomplishing the task, but we found success in using them. Wherever more detailed evaluations of alternatives were conducted, this will be described in the corresponding sections.

Development environment

As our IDE of choice, we decided on *Visual Studio Code*[1]. It is a lean, but highly extensible editor with a modern UI. During development, we used the following extensions, available from the official marketplace[2]:

- EditorConfig for VS Code
- Elm
- haskell-linter
- Haskero
- hoogle-vscode
- indent-rainbow
- Rainbow Brackets
- stylish-haskell

Git

As is currently the industry standard, we used Git[3] as version-control system. On top of that, we followed the Git Flow[4]. New features and bug fixes are developed on specific branches, based on the `develop` branch. Same as in our precursory term project, we did not split responsibilities between ourselves, instead opting for maximum exchange of knowledge.

Following software engineering best practices, any change developed by one of us had to be reviewed by the other using *Gitlab*'s merge request feature. Usually, a merge request would be reviewed and merged into `develop` within a day. On every milestone, the current state of the `develop` branch was integrated into `master`, automatically triggering a deployment to production.

Haskell

As requested in the task description set by our supervisor, the server-side part of the application is written using Haskell[5]. Haskell is a functional programming language with a very extensive and safe type-system.

[1] `https://code.visualstudio.com`
[2] `https://marketplace.visualstudio.com`
[3] `https://git-scm.com/`
[4] `https://github.com/nvie/gitflow`
[5] `https://www.haskell.org`

This project uses the Glasgow Haskell Compiler[6] version `8.6.5`. For dependency resolution, Stack[7] is employed on the resolver `lts-14.6`.

CodePanorama makes use of over 30 different Haskell packages. Many of those only provide simple utilities, while others offer core functionality to CodePanorama's implementation. A listing of notable packages can be found in table 3.1.

Table 3.1: Notable Haskell packages

Package name	Description
aeson	JSON serialization and deserialization
cache	In-memory key-value store with expiration support
git†	Haskell-native implementation of git operations
hslogger	Logging framework for Haskell
JuicyPixels	Generate images from pixel buffers
juicy-draw†	Create images from primitive 2D-shapes
hspec	(Unit-)Testing framework
hxt	XML serialization and deserialization
parallel-io	Sequence IO actions onto a thread pool
prizm†	Convert colors between color spaces
process	Execute external processes via CLI
QuickCheck	Property-based testing framework
regex-tdfa	RegEx implementation in Haskell
servant-server	Serve REST-APIs
servant-swagger	Generate OpenAPI specification from Haskell code
servant-swagger-ui	Serve the API specification using Swagger-UI
warp	Haskell web server

† Not available from the stackage resolver

Elm

For the client part of the application, we decided to use Elm[8]. We evaluated a couple alternatives, but the only satisfactory language turned out to be Elm.

Elm provides a built-in package manager, making dependency management very simple. Some of the noteworthy packages used by the CodePanorama client can be found in table 3.2.

[6] `https://www.haskell.org/ghc`
[7] `https://docs.haskellstack.org`
[8] `https://elm-lang.org`

Table 3.2: Notable client dependencies

Dependency name	Description
pzp1997/assoc-list	Extension to dictionaries, allowing arbitrary keys
rundis/elm-bootstrap	Type-safe bootstrap styles and widgets for Elm
jxxcarlson/elm-tar	Create tar-archives in Elm
dosarf/elm-tree-view	Provides the tree-widget to display the directory filter
elm-explorations/test	Testing framework for Elm
Bootstrap†	Simple and modern style library
FontAwesome†	Library of uniform icons for websites
highlight.js‡	Syntax highlighting for displaying code
showdown‡	Convert markdown files to HTML

† A CSS library, rather than an Elm package
‡ A JavaScript library, rather than an Elm package

Alternatives

The following alternatives were evaluated during the term project and not re-evaluated as part of this bachelor thesis. The results of the evaluation in the term project are reproduced verbatim below [BE19]:

Miso Although, at first, Miso[9] sounded very promising, the installation process quickly proved to be an immense pain (as we later found out with any GHCjs-based front end library): Initial installation took between two and four hours per developer machine, after which it simply decided to exit with a cryptic exception, requiring downgrades of GHC, another couple hours of installation and the last of our patience. Additionally, running the build command on a project with no changes, took the build process about one full minute, until it decided there was nothing to build, which we found completely unacceptable.

Purescript PureScript[10] actually worked pretty well, but relies on more external libraries than Elm and had slightly slower build times, which is why we decided on Elm.

Fay Again, in theory, Fay[11] sounded very good, but after some testing, we found that (at least the current version, as of October 8th, 2018) did not work under Windows whatsoever.

Reflex After short evaluation of the Reflex[12] library, we again found it doesn't work under Windows (would work using the Linux Subsystem), and the initial build time takes a very long time.

[9] `https://github.com/dmjio/miso`
[10] `http://www.purescript.org`
[11] `https://github.com/faylang/fay`
[12] `https://github.com/reflex-frp/reflex`

GHCjs All the problems we had with Miso, we also encountered with pure GHCjs[13], which wasn't too surprising, as Miso is built on-top of it.

NPM

As a means of creating shortcut commands for commonly used tasks, we decided to use the Node Package Manager (NPM)[14]. Since Elm is already installed via NPM, there are no additional dependencies required to use NPM as an orchestration tool, such as make[15].

```
"build-client": "cd CodePanorama-Client && elm make src/Main.elm --
    output=public/out/elm.js --debug && lessc less/style.less
    public/out/style.css && npm run showdown"
```

Listing 3.1: Example of an NPM shortcut command

Swagger

To specify the REST-API of the CodePanorama server, we decided to use the popular standard *Swagger 2.0*[16]. By adhering to an established standard, we could make use of several tools in our development tool-chain: `servant` has a module to generate the specification from the Haskell types, and the *OpenAPI Generator*[17] can use this specification to generate Elm code. This way, code duplication between the server and the client is minimized.

The current specification of CodePanorama can be found at `https://swagger.codepanorama.io`.

Gitlab

As already done during the prototype study, we used Gitlab[18] to manage our code repository. Within Gitlab we were able to configure a CI/CD pipeline to automatically build, test, lint, and even deploy our source code.

[13] `https://github.com/ghcjs/ghcjs`
[14] `https://www.npmjs.com`
[15] `https://www.gnu.org/software/make`
[16] `https://swagger.io/docs/specification/2-0`
[17] `https://github.com/OpenAPITools/openapi-generator`
[18] `https://gitlab.com`

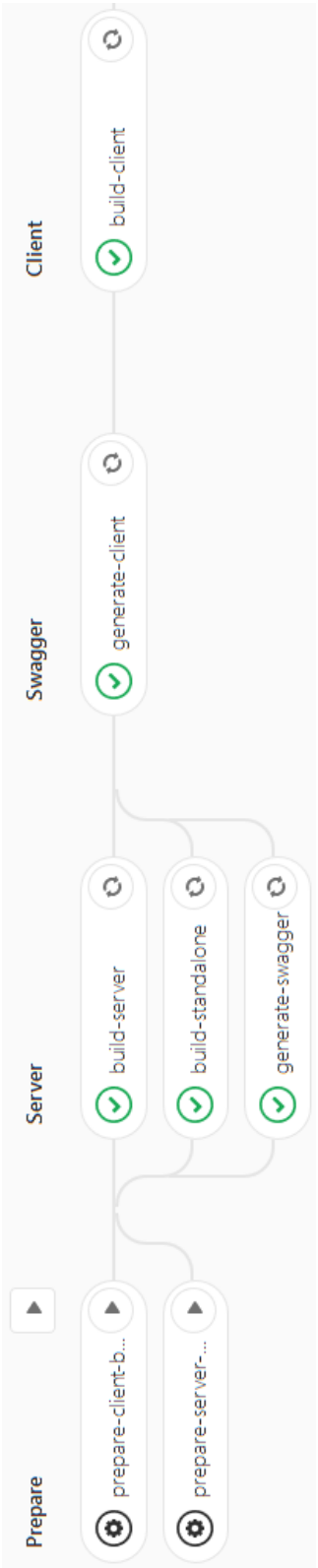


Figure 3.1: Gitlab pipeline of CodePanorama (part 1)

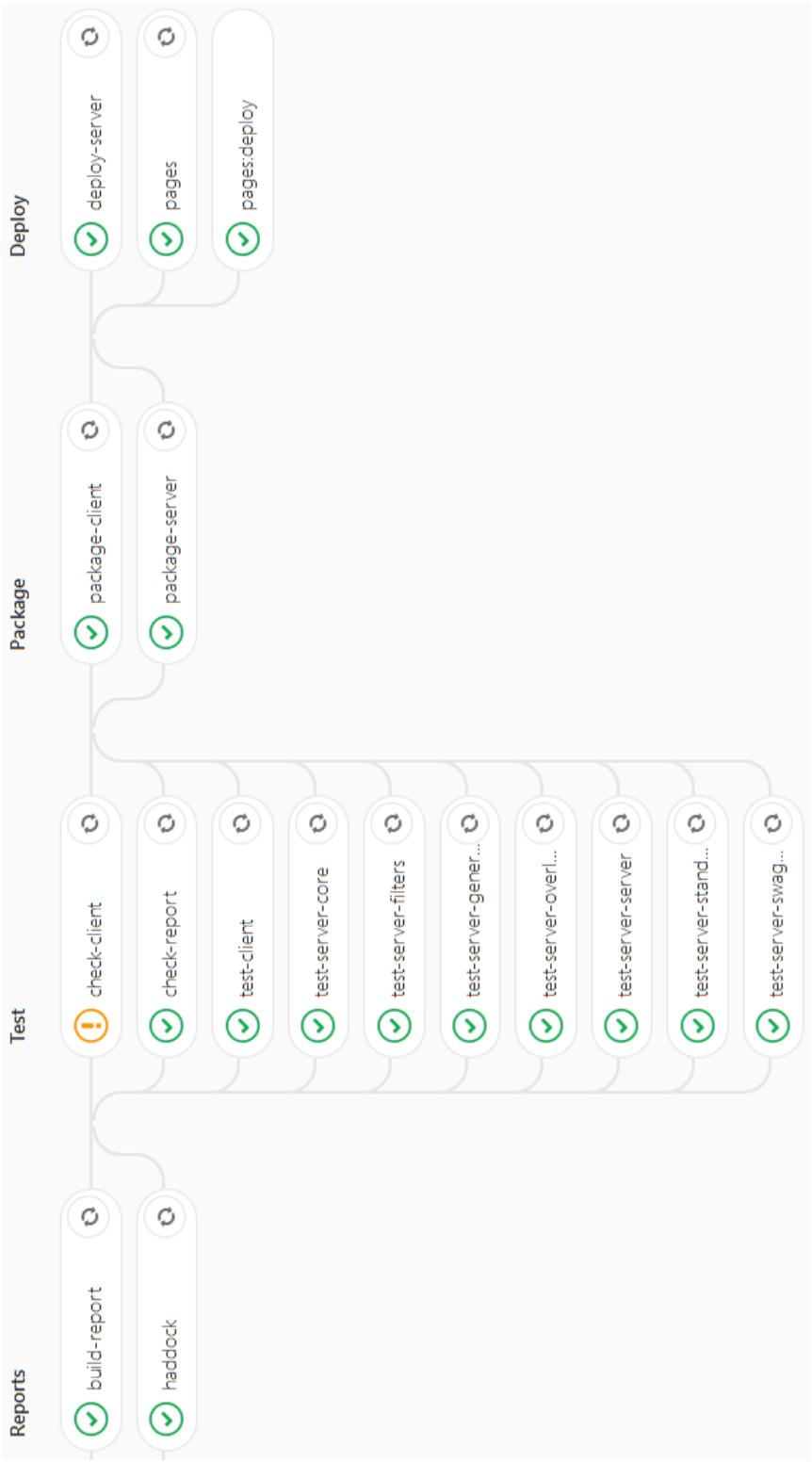


Figure 3.2: Gitlab pipeline of CodePanorama (part 2)

Jira

Different from our precursory work, we decided to use Jira[19] as our project management tool. In our opinion, Jira provides a better organized view of issues, and easier management of the backlog and releases. However, the most important point of why we preferred Jira over Gitlab's built-in issue management, is Jira's time tracking ability. Although Gitlab has the ability to log work on issues, it is very hard to extract meaningful reports. Using the *Timetracker*[20] app, generating reports of logged work per milestone becomes very easy.

Docker

Docker[21] is a virtualization technology integrating deeply with the operating system's kernel, providing a leaner footprint than virtual machines. Docker is extremely useful in encapsulating applications and tools in a controlled environment. With CodePanorama we made heavy use of docker images for our build pipeline. Every step in the pipeline runs inside a docker container defined in the build script. By using custom docker containers, we can use almost any tool we need, without the need of installing any software on the Gitlab host server.

We also use docker as our deployment platform of choice. More on this can be found in section 4.3. Artifacts.

[19] `https://www.atlassian.com/software/jira`

[20] `https://marketplace.atlassian.com/apps/1211243/timetracker-time-tracking-reporting`

[21] `https://www.docker.com`

3.2 Filters

One of the important features of CodePanorama are filters. By allowing the user to decide what will be included in a code panorama, it is very easy to create panorama images that only show the parts of a repository a user is really interested in.

Extensibility

Although we were able to implement all currently existing filters based on the regular expression filter (see regular expression implementation), it is likely that some other type of filter will be needed at some point. To simplify future development, we designed the filter package with extensibility in mind.

For this to be possible, the `filters` package exposes generic mechanisms, which allow for great flexibility when implementing a filter.

One of the key parts is the implementation surrounding the `ConfigurationMap` type. This type is nothing more than a simple string-based key-value map.

```
type ConfigurationMap = Map.Map String String
```

However, with the help of the `Serializable` and `MapSerializable` types, this map will be transformed into a record, which is not limited to strings only, but can contain any type that defines an instance of `Serializable`. By default, every type deriving the `Read` and `Show` classes can be used.

```
-- Generically makes all (Read, Show) types Serializable
instance {-# OVERLAPPABLE #-} (Read a, Show a) => Serializable a

-- Special case for String not using quotes
instance {-# OVERLAPPING #-} Serializable String where
    serialize = id
    deserialize = Right
```

Listing 3.2: Predefined `Serializable` instances

The `ConfiguredFilter` type is important, as well. This is a predicate, which decides for every file in a repository whether it passes a filter or not.

```
type ConfiguredFilter = FilePath -> Bool
```

Lastly, there is the `Registry` module. When a user requests filters to be applied (e.g. by requesting a preview), the names of all the files in a repository along with the user's configuration will be passed to the filters registered here.

```
data FilterName
    = RegularExpression

filterRegistry :: FilterName -> Filter
filterRegistry name = case name of
    RegularExpression -> regularExpressionFilter
```

Details on how to implement a new filter module can be found in the documentation of the `filters` package in the source code.

Example: Regular expression implementation

To begin with, the regular expression filter defines a `Configuration` record and `Scope` as a supporting type.

```
data Configuration = Configuration
    { scope :: Scope
    , regex :: String
    }

data Scope
    = Extension
    | FileName
    | FullPath
    deriving (Read, Show)
```

As outlined above, an instance of this record will be created based on a key-value map. This map is based on the user's configuration and might look like the following (simplified) example.

```
{
    "key": "scope",
    "value": "FullPath"
},
{
    "key": "regex",
    "value": ".*Test\.java"
}
```

It can be clearly seen that the keys of this map directly correspond to the field names in the `Configuration` record above. Furthermore, the parallels between the maps and the record's values are easily recognizable. The correct mapping of the values will be handled by the internal logic of the `Serializable` and `MapSerializable` types.

Now that the configuration is defined, all that is left to do, is implementing the actual filter. This is done based on the `Configuration` and `ConfiguredFilter` types and will simply return whether a certain file name matches the given regular expression or not.

```
regularExpressionFilter :: Configuration -> ConfiguredFilter
regularExpressionFilter config file =
    let
        filePart = case scope config of
            Extension -> takeExtension file
            FileName  -> takeFileName file
            FullPath  -> file
    in
    case regex config of
        "" -> True
        r  -> filePart =~~ r
```

3.3 Overlays

A highly requested feature and a natural extension to plain code panoramas, are color overlays. By enriching the panorama images with color information, not only the spatial structures of the code can be learned, but also the correlation of said structures with whatever metric is chosen for the overlay.

Both *SeeSoft* [ESS92] and *Augur* [FD04] have made heavy use of colors to supplement the basic information provided by the code-map metaphors. In an attempt to make CodePanorama as user-friendly as possible, we evaluated multiple strategies on how, where, and how many colors to use.

We are convinced that it does not make much sense to display multiple metrics encoded as colors simultaneously. Therefore, we have decided to limit the number of displayed overlays to at most one. Furthermore, since we color the actual lines of the code, combining overlays would require mixing colors which is exceedingly hard to encode in a meaningful way and even harder for the human brain to parse.

Overlay types

CodePanorama tries to encode two fundamental types of data into overlays: quantitative and nominal. The current implementation can display information according to table 3.3. A detailed description of each metric can be found in Appendix D.2 Overlays.

Table 3.3: Categorization of overlay information

Quantitative	Nominal
Change frequency	File type
Author participation	Search
	Blame

Since quantitative and nominal data have different relations between data points, it is only natural to select different visualization techniques.

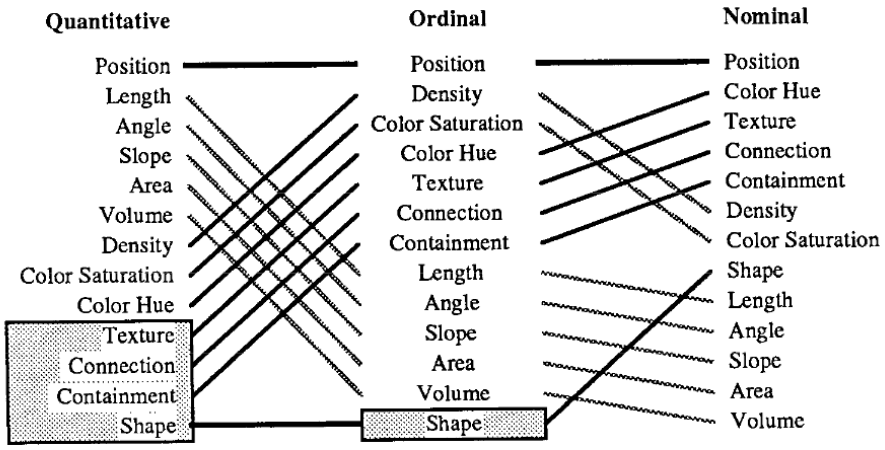
As we have already decided to only use colors to visualize the additional information, the best methods in each category turn out to be color saturation for quantitative data and color hue for nominal data (see figure 3.3). In practice, we implemented both visualizations using a color gradient.

For quantitative data, we used a scale between white and red, where the saturation is calculated according to equation 3.1.

$$S(l)_{l \in L} = \frac{f(l)}{\max_{i \in L} f(i)} \tag{3.1}$$

where L is the set of all lines included in the current code panorama configuration, and f is a metric assigning a real value to every line. The color red was selected

Figure 3.3: Ranking of perceptual tasks. [Mac86]



arbitrarily as a familiar color scheme for heatmaps.

Concerning nominal data, the primary objective is to make the different hues as distinguishable as possible. To accomplish that, we opted to use the entire color spectrum and evenly space the data points across the available hues. Hues are commonly defined as degrees on a circle [PF07]. Therefore, it is natural to calculate the hue to be displayed as in equation 3.2.

$$H(l)_{l \in L} = \frac{i_C(f(l))}{|C|} \times 360\,\text{deg} \tag{3.2}$$

where C is the set of all possible classes for the selected metric, f assigns a class c_i from C to every line, and i_C maps any class c_i to its index i (starting at zero).

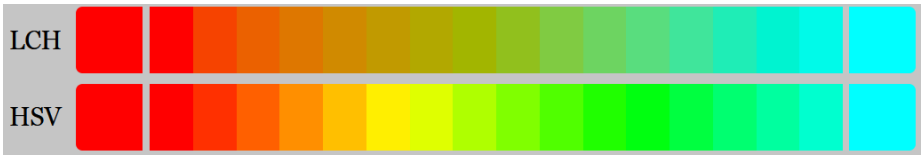
Color space

Now that we have formulas for the saturation and hue for quantitative and nominal data, respectively, the question remains how to map these values to actual colors. Having calculated values for saturation and hue, it would seem obvious to use the HSV color space [22].

We want to pay special attention to the perception of the step-distances in quantitative overlays (heatmaps) and easy distinguishability between hues for nominal overlays. Therefore, we decided to use a more involved color space: the CIE 1976 L*u*v* color space (also known as simply CIELUV or LCH amongst others), where "Numerical values representing approximately the relative magnitude of colour differences can be described by simple Euclidean distances in the spaces [...]" [ISO11664-5]. In other words, the same numerical change in hue or saturation is perceived equally, regardless of which color the change is applied to. A comparison of a color gradient using HSV and CIELUV can be seen in figure 3.4.

[22] `https://en.wikipedia.org/wiki/HSL_and_HSV`

Figure 3.4: Comparison of a CIE 1976 L*u*v* (here LCH) and HSV gradient.



It becomes clear that the perceived brightness of the colors in HSV vary greatly to the point where some steps around red and green can hardly be distinguished at all. Meanwhile, the steps on the LCH gradient are much more (perceptually) equidistant.

Fortunately, we did not have to implement the conversion to and from CIELUV ourselves. Instead, we were able to make use of the pre-existing Haskell library `prizm`[23].

Implementation

In terms of technical implementation, we generate a separate PNG image for every overlay metric. These images follow the same layout as the code panorama, but contain colored lines where applicable and a transparent background. To enrich a code panorama with an overlay, the overlay image can literally be overlaid on top of the base panorama image, superimposing its colors on the otherwise plain line-rectangles.

For example, the overlay image of a search overlay would only contain colored rectangles of the lines where the search expression was found. All other lines are simply transparent.

Custom overlay

To allow for maximum customizability at runtime, we decided to implement a custom overlay. Using this custom overlay, any user can supply their own metrics and display them in the CodePanorama application.

To use the custom overlay functionality, the user must provide a JSON file containing the specification of their desired overlay. This file must then be placed in the repository to be analyzed. The application will then recognize the custom overlay, render it, and offer the overlay as a selection in the UI.

In essence, the custom overlay allows to specify a numerical value per line for whatever metric the user desires and how to map these values to a color. Additionally, the color can be overridden on a per-line basis.

An example usage of a custom overlay would be test coverage. Since test coverage can only be calculated language-specifically, it is easier to leave this to the user. As a proof-of-concept, we developed a CLI tool to generate a custom overlay specification from a JaCoCo[24] coverage report.

[23] `https://hackage.haskell.org/package/prizm`
[24] `https://www.jacoco.org`

For details on how exactly to create a custom overlay specification, please consult the corresponding manual in Appendix C.5 Using custom overlays.

Extensibility

Although, in theory, the custom overlay covers all possible future overlays, the burden of creating such an overlay lies with the user. To simplify future development on CodePanorama, we designed the overlay package with extensibility in mind.

Re-using the generic design of the filter package, overlays can be added in a modular manner, with commonly used functionality available in helper modules. When the user requests overlays to be generated, the user's configuration and the complete internal representation of the code panorama are passed to every registered overlay module. An overlay implementation must therefore only specify how to calculate the relevant metric, and which color to select for the calculated values. In most cases, the overlay framework and pre-defined generators then take care of actually rendering the overlay, sending it to the client, and properly displaying it.

Details on how to implement a new overlay module can be found in the documentation of the `overlays` package in the source code.

Example: Change frequency implementation

Firstly, the change frequency implementation defines the `ValueSupplier` type as a function which maps any combination of file and line number to maybe a number of changes this line has been subjected to. The `Maybe` type handles the case where a given combination of file name and line number might not actually exist.

```
type ValueSupplier = FilePath -> Int -> Maybe Int
```

Next, the (simplified) `ValueSupplier`-implementation for a per-line change frequency looks like this:

```
lineBasedValueSupplier :: IO (ValueSupplier, Int)
lineBasedValueSupplier = do
    (changeNumbers, maxChanges) <- getLineChangeNumbers repoDir
    return
        ( \file lineNumber ->
            Map.lookup (file, lineNumber) changeNumbers
        , maxChanges
        )
```

This function returns a `ValueSupplier` and the maximum number of changes across all files and lines. The `getLineChangeNumbers`-function encapsulates the logic for walking the entire directory tree and the various git commands to retrieve the number of changes per line. The actual `ValueSupplier` is a lambda, where the value we want to assign to a line is simply the result of a lookup into the (curried) map returned by `getLineChangeNumbers`.

This `ValueSupplier` is then used with the helper function `lineOverlayGenerator`, which generates an overlay image based on providing a color for every panorama line:

```
valueSupplier <- lineBasedValueSupplier
return $ lineOverlayGenerator (\file lineNumber ->
    valueSupplier file lineNumber
        <&> ((/ maxChanges)
            .> lerpColorLCH gray red
            )
)
```

The code above passes a single lambda to the `lineOverlayGenerator`, namely a function which is expected to return a color for a given combination of file name and line number. We simply pass these values to our `ValueSupplier`, providing us with an integer of the absolute number of changes on this line. Dividing this number by the maximum number of changes across all lines, we get a value between zero and one. This value can now be used for a Linear Interpolation (lerp) between gray and red in the CIELUV color space, resulting in the desired color for this line.

3.4 Rendering-performance

One of the bigger problems of Code Panorama is that loading the generated images in a browser can take anything from a few seconds to several minutes, and it is even possible for a browser to become stuck and eventually crash. This is due to the fact that in the prototype version the images are saved in SVG-format, that contains more and more elements the more detailed the Code Panorama becomes, which in turn increases loading times for the browser, since all elements have to be loaded individually.

To find alternative solutions, several tests, which explore different ways of generating images, have been conducted. Common to all approaches is that what would eventually be displayed by a browser are now PNG-files instead of SVG-files. Since images in PNG-format use much less space and can be loaded by a browser as one element, we expected loading times to improve significantly.

Test results

As a point of reference, the average time necessary to generate a single SVG-file at medium size was measured: 0.6 seconds. This also includes the additional time needed to read the files from the repository, which is included in the following measurements as well. Furthermore, the medium-sized panorama SVG is around 2.2MB in size.

The actual tests were run on three different implementations. In the first two, the PNG is generated from the already existing SVG, which is either read from the disk or the memory. And in the last implementation the PNG is generated directly, without pre-generating an SVG-file.

Table 3.4: Image-generation runtimes and file-sizes

	SVG	PNG (memory)	PNG (disk)	PNG (directly)
Time (s)	0.6	2.9	0.9	1.0
Size (MB)	2.3	0.185	0.185	0.081

Conclusion

The general observation is that directly generating PNG-files takes a bit more time than generating SVG-files (1.0 and 0.6 seconds respectively) and is also slightly slower than generating a PNG-file from the SVG loaded from memory (0.9 seconds). However, the files generated by the former approach are about 56% smaller than those produced by the latter. This difference becomes smaller — as low as 27% — the less detailed an image is, but also increases — as high as 80% — the more detailed an image is.

Based on this, we decided that the best way to improve the performance of image-loading is, to switch to generating PNG-files directly, since this yields much smaller files at the cost of minimally increased run times for generating the images.

3.5 Asynchronous processing

The process of generating a code panorama, including all overlays, requires reading every file in the repository at least once. Additionally, many git commands must be executed and interpreted to gather all the necessary data and statistics for the overlays.

Depending on the size of the repository, all of these tasks together can be very time- and resource-consuming. To provide a satisfying user experience when waiting for the images to be generated, we implemented an asynchronous processing model: Whenever a user wants to generate new images, the client sends a POST request to the server to create a `PanoramaJob`. The server will create the job and start executing it, but immediately respond with the job's ID. The client may then use this job-ID to query the progress of the requested job and ultimately retrieve the result, once the job is finished.

Polling

There are many technical approaches to checking the progress on an asynchronous job [Abr17]. Although Server-Sent Events would fit the task more accurately, we decided to implement the simplest solution: short polling on a 500ms interval.

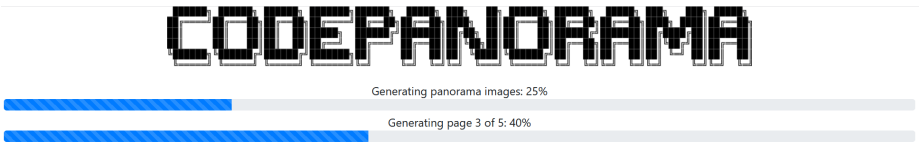
Performance

We did not conduct any well-founded benchmarks on the entire panorama generation process. However, we gathered some experience from working on our local machines and the production instance provided by HSR. Generating five code panorama pages on a medium-sized repository (100k–1M lines), usually takes around 10–15 seconds. Generating a "per-line" statistic, such as when selected on the change frequency, or when using the blame overlay, can easily require up to 5 minutes on small-to-medium repositories (10k–50k lines), and even 15 or more minutes on larger repositories.

Progress reporting

With asynchronous processing in place, the server can now report detailed progress on its jobs' executions. To accomplish that, any long-running function must be able to report its progress back to the caller. By nesting progress-updates, an entire progress-hierarchy can be built. To avoid unnecessarily flooding the user with progress information, we decided to limit the level of detail to two progress indicators: The upper progress bar showing the overall progress of the job, and the lower bar showing the progress of the currently running top-level step. Such a multi-level progress bar is well-known from e.g. software installers.

Figure 3.5: CodePanorama UI displaying the progress of a running `PanoramaJob`.



To allow functions to report their progress, we opted for a solution inspired by the `progress-reporting`[25] package. Unfortunately, this package only allows for reporting progress as numerical values, whereas we needed updates to include a status message to display to the user. Therefore, we defined the following callback-type:

```
type ProgressCallback m = String -> Double -> m ()
```

By passing this callback to long-running functions, the caller can specify what happens when the called function reports a status update, while the called function has control over when and what updates are reported.

The (simplified) function for calculating the number of changes per line, looks as follows:

```
map (\file -> do
    -- increment of counter omitted
    progress ("Calculating ChangeFrequency for " ++ file) $
        fromIntegral counter / genericLength files
    (file,) <$> getFileChangeNumbersForFile file
) files
```

Here, we map over all files and report a progress update for every file we process. The percentage of the progress is simply the index of the current file, divided by the number of files to process. The function `getFileChangeNumbersForFile` then contains the logic for obtaining the actual numbers used in the overlay metric.

Getting the number of changes per line on a long file could be very time-intensive, as well. However, since the time required to get the number of changes for a single line is very short, the overhead of reporting progress on the line-level would significantly degrade the overall performance. As we make heavy use of multi-threading to speed up this process, the progress reporting must be synchronized between threads. If every thread was reporting progress on a line-basis (effectively every couple milliseconds), the additional time required for keeping the progress updates synchronized would be enormous.

Now that progress is properly reported by long-running functions, we can use this information in the `PanoramaJobServer` to update the corresponding job status, by defining the `ProgressCallback` passed to its sub-functions:

```
progress_ :: ProgressCallback IO
progress_ status value =
    getJob_ (Data.jobId job) >>= \case
        Nothing -> throwJobNotFoundException (Data.jobId job)
        Just j -> do
            updateJob_ j
                { Data.statusMessage = status
                , Data.statusProgress = value
                , Data.statusDetailMessage = Nothing
                , Data.statusDetailProgress = Nothing
                }
```

These status fields are then displayed in the client as seen in figure 3.5.

[25] `http://hackage.haskell.org/package/progress-reporting`

3.6 Concurrency handling

Another issue we encountered in the development of CodePanorama are concurrent users. Although Haskell claims to easily support proper concurrency, the issue arises from the fact that CodePanorama makes heavy use of the file system both for storage and as main input. Making sure these directory structures and storage files are handled in a thread-safe manner is therefore not trivial.

Affected resources

Before attempting to implement proper concurrency handling, the relevant resources need to be identified, which must be protected from race conditions. In the case of the initial CodePanorama implementation, these were the following:

Repository store

The repository store is a registry of all cloned repositories in the server's workspace. Since server-side repositories are identified through UUIDs, this registry maps the repository URLs to the corresponding UUIDs. This enables re-using existing repository clones, whenever a user requests a repository which has previously been cloned (either by another user, or by the same user). This registry also stores hashed credentials for private repositories, so a private repository will only be re-used if the same credentials are supplied again.

Since every clone operation (`RepositoryJobRequest`) needs to read and potentially write to the repository store, this resource is a potential source of race conditions.

Repository information

Whenever a repository is analyzed, the CodePanorama server computes metadata about the given repository, such as total number of lines, proportions of file extensions, list of contributors, etc. This information can easily be cached to speed up performance on subsequent requests. On the flip side, such a caching mechanism introduces potential concurrency risks, which must be addressed.

Panorama and overlay images

The final results of the CodePanorama process are the actual PNG images generated by the server. Since these images are based upon specific configurations by the client such as filters, panorama size and image dimensions, they can not easily be re-used across clients. Therefore, the application must guarantee that images generated by one user will never be displayed to another user.

Solution

In general, concurrency can be solved by not sharing data across concurrent access, or by ensuring proper synchronization between concurrent access of the same resource. Depending on the resource in question, we implemented a custom-tailored solution for that resouce.

Repository store

Since the repository store is a global registry for the entire application, the registry must be shared across all users. The "correct" approach would be to use an established database, such as sqlite[26], which already makes all the concurrency guarantees we would need.

We had already implemented the repository store with a simple JSON-file, and accesses to the registry happen rather unfrequently. Therefore, we decided to simply secure all reads and writes into the repository store with a mutex. This has the drawback of always only allowing a single thread access to the store, even if concurrent access would be unproblematic. However, it only required minimal modification to the existing code.

Repository information

Previously, the repository information was stored in a JSON-file, similarly to the repository store. However, the repository information never changes once it has been generated, except for when the checked-out commit is updated. Since this happens even less frequently than an access to the repository store, we implemented the same solution using a mutex. Although the repository information is read relatively often, and a mutex does not allow multiple concurrent reads, each read should be very quick and therefore have minimal impact on performance.

For a "proper" implementation, the locks would have to distinguish between read- and write-locks. However, every access must first check whether the repository information exists (read operation) and then generate the file if it does not exist yet (write operation). Such a sequence of operations would require an upgradeable read-lock mechanism, which is not trivial to implement correctly. Considering the added complexity of an upgradeable read-lock, we were satisfied with the simpler mutex solution.

Panorama and overlay images

In contrast to the previous resources, we have no interest in sharing generated images across clients, since two clients using the exact same configuration is highly unlikely. Therefore, we opted for a different approach here and store the generated images separately in memory for every request. This means there are no more concurrent accesses to a shared resource, automatically solving the issue. The drawback, however, comes in a higher memory footprint and the requirement to somehow clean up no longer used images.

A popular solution for storing data with limited life-time is Redis[27]. For this Bachelor thesis, the integration of a Redis service seemed out-of-scope. Hence, we opted for a simpler solution: the Haskell cache[28] library, offering very similar functionality to Redis, but without the requirement of attaching a separate process.

[26] `https://www.sqlite.org`
[27] `https://redis.io`
[28] `https://hackage.haskell.org/package/cache`

Using the afore-mentioned cache library, we can now store the images as Base64-encoded[29] strings, indexed by the job-ID generated on the client's request. Whenever any client requests to generate new images, a new job-ID is generated and a new entry in the image-registry is created. Clients can then retrieve the images using the generated job-ID until the result expires. This expiration is currently not configurable outside from modifying the source-code and is hard-coded to a value of one hour after the image has been generated.

[29] `https://tools.ietf.org/html/rfc4648`

3.7 Git integration

The main backbone of CodePanorama is its integration with git as a version control system. Not only does CodePanorama use git to clone the repository to be analyzed, it also extracts important statistics from the git history. To accomplish this, some sort of binding between the Haskell server application and the git structure on the file system must be built.

Our first attempt was to use the Haskell package aptly named `git`. Although promising at first, the longer we used it, the more features we needed were either incomplete or missing entirely. Since the `git` package worked reasonably well for what we had used it so far, we decided to keep it for functionality where we could use it and build a new integration for the missing features. We wanted this new binding to be easy to implement and be guaranteed to provide all the features we will need. Therefore, we decided on creating a binding with the CLI version of git.

Using the "process" Haskell package, running CLI commands as an external binding from Haskell code is not too hard:

```
gitCli :: FilePath -> [String] -> IO (ExitCode, String, String)
gitCli repoDir args =
    readCreateProcessWithExitCode (proc "git" args)
        { cwd = Just repoDir
        , std_out = CreatePipe
        }
        "" -- stdin
```

The `gitCli` function above can simply be called as follows:

```
gitCli repoDir [ "fetch", "origin" ]
```

Our internal git CLI binding API also provides abstractions for e.g. only retrieving the process' standard output or forwarding the process' standard and error outputs to the CodePanorama loggers.

Using the git CLI as an external binding has the benefit of having access to the rich command palette available through the executable. On the other hand, this comes at the cost of requiring the git executable to be installed on the runtime environment where the CodePanorama server is deployed. Using docker images, however, we can easily pre-install git alongside the CodePanorama executable, so users never have to actually worry about the runtime dependencies.

3.8 Single-page application

Since we wanted to have a client that is both easy to deploy and based on modern web development practices, we decided to implement it as a Single Page Application (SPA).

This approach is supported by Elm and a readily available example exists[30]. We used this example as a baseline and adapted it to our specific needs.

We did, for example, implement the `Shared` module, which is used to store data that is needed by more than one page (e.g. the URL for the API). We also wanted a stronger separation of the different parts that make up a single page. In the original example, the update- and view-logic as well as the corresponding model are always in the same module. While this approach works great for small pages without much functionality, it quickly leads to code that is hard to read and maintain. Thus, we decided to split up the code for every page into smaller modules, each serving a particular purpose. This makes it easy to extend the existing pages with new features and overall results in better structured code.

[30] `https://github.com/rtfeldman/elm-spa-example`

3.9 Testing

An integral part of making sure software is correct, is automated testing. Since CodePanorama consists of two main components (the server and the client), we have separate tests for each.

All of our tests are executed automatically on every commit by the CI/CD pipeline in Gitlab.

Server testing

In the server we have three categories of tests: property tests, unit tests, and integration tests. We use property tests to validate functions with clear properties that can be verified by generating random test data. Whenever we want to test a more complicated function, we use unit tests, where the function is only tested with pre-selected values — preferably edge cases. Lastly, for testing the server's REST-API, we have integration tests, where actual REST requests are sent to a temporarily started web server and the returned response is verified.

Property tests are implemented using the `QuickCheck` package, unit tests make use of the `hspec` package, and integration tests rely on the `hspec-wai` extension to `hspec`.

```
it "deserializes to the String we started with" $
    property (\(v :: String) ->
        serialize v |> deserialize |> (== Right v))
```

Listing 3.3: A property test, verifying that serializing then deserializing a value returns the original value

```
it "fails on unrenderable files" $
    assemblePanorama config [ unrenderableFile ] `shouldBe` ([], [
        ( "Unrenderable.txt"
        , "Unprintable non-space Character: '\\NUL'"
        )])
```

Listing 3.4: A unit test, checking that an unrenderable file cannot be included in a panorama

```
bracketTest
    (encodeFile testRepoInfoFile testRepoInfo) -- Set up test
    (removeFile testRepoInfoFile) -- Clean up after test
    $ it "Returns the repository info" $
        get (path <> "/test-repo")
            `shouldRespondWith` json 200 testRepoInfo
```

Listing 3.5: An integration test, ensuring that the `RepositoryInfoServer` returns information from the correct file

Client testing

The client does not contain much business or application logic. Therefore, our test suites are not as extensive here as for the server. We only have unit tests in this area, making use of the official `elm-explorations/test` package.

```
test "mixed characters" <| \_ ->
    Expect.equal
        "\\{\\$a\\.b, c\\(12\\)\\[9\\]\\^\\\\\\}"
        (RegexUtil.escape "{$a.b, c(12)[9]^\\}")
```

Listing 3.6: A unit test, checking proper RegEx-escaping

4 Results

4.1 Architecture

Figure 4.1 shows an overview of the architecture of CodePanorama. The different layers and modules are briefly described in the following sections.

Layers

The application is divided into two layers. One being the presentation-layer and the other a combination of an application- and business-layer. So far, only trivial persistence was necessary, which we implemented with plain JSON files. Therefore, there is currently no "real" data-layer, rather the persistence is embedded into the business-layer. Communication between the presentation and application layers is based on HTTP.

Modules

Client

The `Client` module contains all the code for the web-based user interface. It communicates with the server through a REST API. Parts of the client-code are generated by the OpenAPI-CodeGen module, based on the API specifications. The client is written in Elm, HTML, JavaScript, and LESS.

OpenAPI-CodeGen

Reads an API-description file as defined by the OpenAPI specification and generates the necessary data-types and request-functions as Elm-code.

Swagger

Generates an API-description file as defined by the OpenAPI specification based on endpoint- and data-type-descriptions from the `Server` module. Can also be used to run an instance of Swagger-UI based on the API-description file.

Server

Defines and implements the API, and thus handles all the requests made by the client (or through Swagger-UI) and passes them on to the `Core` module for further processing. It has built-in asynchronous functionality and can serve multiple clients at the same time.

Standalone

Originally used to test various libraries and their viability for the project. Could still be used as a CLI-tool for very simple tests but has none of the functionality provided by the client.

Overlays

Contains the generic components and helpers to allow implementing overlays as outlined in section 3.3. Overlays.

Generator

Contains the functionality and helpers to generate code panoramas.

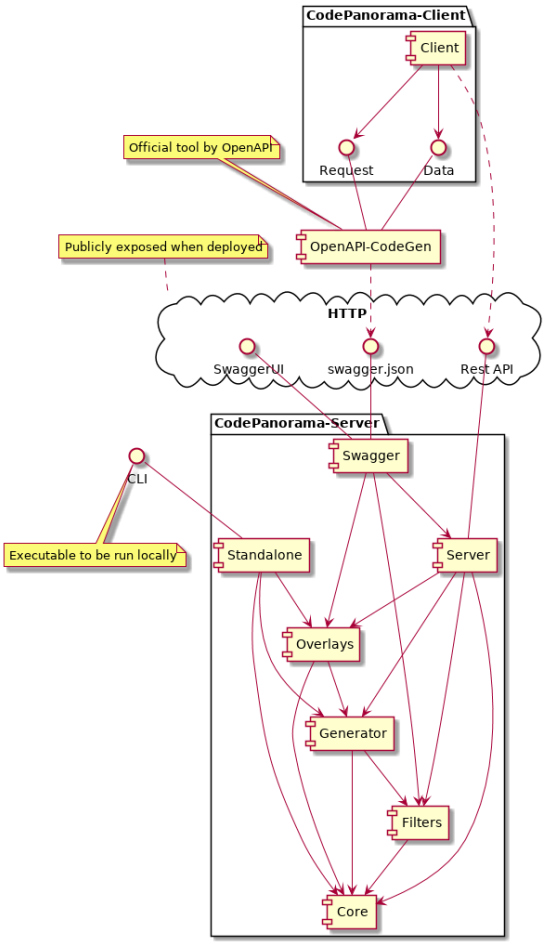
Filters

Contains the generic components to allow implementing filters as outlined in section 3.2. Filters.

Core

The actual backend which does all the “heavy-lifting”. Contains sub-modules to collect repository-data and generate panoramas.

Figure 4.1: Architecture of CodePanorama

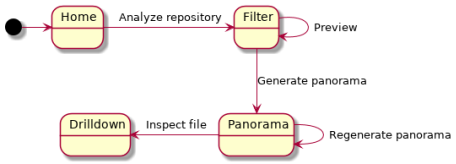


4.2 Design diagrams

UI flow diagram

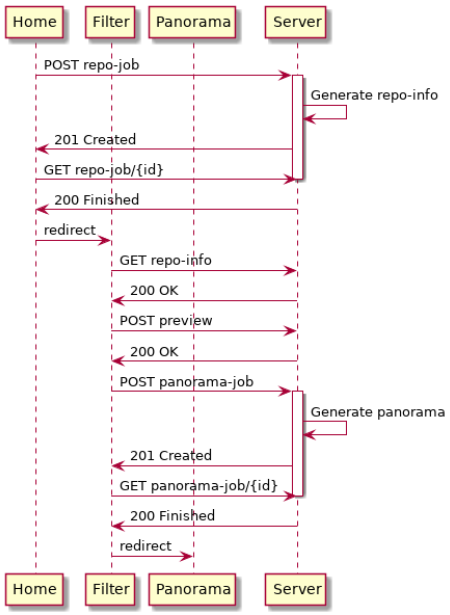
Figure 4.2 shows the (simplified) UI flow of CodePanorama. The *Home* page is always the start page. Here, a user will enter a URL to a git repository, and credentials if necessary, or select a local repository. The user will then press the *Analyze repository* button and be redirected to the *Filter* page. On this page, the filters can be adjusted and every adjustment will trigger a call to the *Preview* endpoint. Once the user is satisfied with their selection, they will press the *Generate panorama* button. This will generate all the necessary images and redirect the user to the *Panorama* page. Here, different overlays can be selected, parameterized and regenerated. It is also possible to click on a specific file and switch to the *Drilldown* page.

Figure 4.2: UI flow diagram



Sequence diagram (client-server)

Figure 4.3: Sequence diagram — Client-Server communication



The diagram in figure 4.3 gives a quick overview of how the different pages in the client communicate with the server, and in which situations redirects happen.

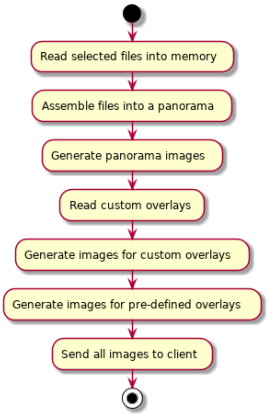
The `GET repo-job-` and `GET panorama-job`-requests are sent periodically. The server responds to each request with the current status and progress of the job running in the background, if available.

The `POST preview`-request can also be sent multiple times, but will always be sent at least once when entering the filter-page.

PanoramaJob

The activity diagram in figure 4.4 shows a simplified version of the `PanoramaJob`. This job contains most of the logic responsible for generating code panoramas. A major part of this is the activity *Assemble files into panorama*. There, a bunch of files are transformed into an internal representation, which is then used by the subsequent steps to generate images. In the last step, all that is left to do is send the images to the client.

Figure 4.4: PanoramaJob



4.3 Artifacts

The main resulting artifact of this bachelor thesis is the CodePanorama application. Although the source code is published as open-source[1], we also produced binary artifacts for users.

Docker images

The primary way of running CodePanorama uses docker images. As described in subsection *Docker* of section 3.1, using docker greatly simplifies deployments, as the deployment artifact is almost entirely independent of the host environment. For CodePanorama, we published two separate docker images: `hsrcodepanorama/code-panorama-client`[2] and `hsrcodepanorama/code-panorama-server`[3].

Detailed instructions on how to use these docker images can be found in Appendix C.3 Deploying CodePanorama with docker.

CodePanorama.io

The Institute for Software (IFS) at HSR has offered to host a production deployment of CodePanorama. This instance provides its API at `https://api.codepanorama.io`. Using Gitlab's Pages feature[4], we deploy the client directly from the CI/CD pipeline, accessible at `https://codepanorama.io`. This client is configured to access the backend hosted by IFS. Lastly, the API specification is available at `https://swagger.codepanorama.io` (also hosted by IFS).

[1] `https://gitlab.com/CorrectnessLab/codePanorama/issues`
[2] `https://hub.docker.com/r/hsrcodepanorama/code-panorama-client`
[3] `https://hub.docker.com/r/hsrcodepanorama/code-panorama-server`
[4] `https://docs.gitlab.com/ee/user/project/pages/`

4.4 Development

There are many additional desirable features and improvements left for CodePanorama. We — and the Haskell community in general — strongly believe in open-sourcing software and allowing the public to improve and develop code. This is one of the main reasons why we designed CodePanorama (especially the `filters` and `overlays` modules) with high extensibility in mind.

Open source repository

All of CodePanorama's source code is publicly available for inspection and merge requests at `https://gitlab.com/CorrectnessLab/codePanorama`. This repository also includes a public issue tracker intended for reporting bugs and requesting new features by users.

Developer guide

Instructions on how to get started developing CodePanorama can be found in the developer guide markdown file in the repository. This developer guide can also be found in Appendix C.1 Developer guide. In general, it is advisable to start with the afore-mentioned developer guide and read the `README.md` files of the modules to be worked on.

4.5 Known issues

This section lists known issues of the CodePanorama software that could not be fixed as part of this bachelor thesis. The list is not complete, but includes the most important and impactful issues.

Critical

There are currently no known critical issues. A critical issue would either prevent some functionality from being used, have the potential to crash the application, or be related to a security vulnerability.

Major

An issue is categorized as major, if it either has a well noticeable impact on usability, or the possibility to break a small part of the application.

Files at end of page are cut off The last file of every code panorama page usually does not fit entirely onto that page. Currently, we do not handle such overflowing files very well. Instead, these files are not rendered at all. Possible strategies would include simply rendering the lines that fit into the page and discard the rest, or rendering the rest of the file onto the next page. Neither solution seems very satisfactory, though.

Last page has empty space Related to the issue above, the last page (especially when there are many pages) has a large empty space at the end. The source is not completely clear, but it is likely related to the discarding of overflowing files on each page.

Panorama cuts off partial columns Currently, the filter page allows for arbitrarily precise customization of the panorama dimensions in pixels. If the width of the panorama image is not a multiple of the configured column width, the right-most column is partially cut-off. Not only does this not look aesthetic, for some reason these partial columns cannot be hovered in the client and therefore not be drilled-down into.

Fix links in documentation The documentation page in the web UI is generated automatically from the source markdown files. These markdown files contain relative links to other markdown files. Those links are not translated when generating the HTML and therefore do not make sense in the web-context. Possible solutions include either using absolute URLs in the source markdowns, or finding a reasonable way to translate relative URLs during conversion.

Minor

Minor issues are either hardly noticeable, or the impact of when the issue arises is very small.

SVG hover map fails on one-line files If a file containing only a single line is included in the panorama, it is currently impossible to hover it and therefore drill-down into it. Likely, there is an off-by-one error in the generation of the SVG model. For longer files this error is not really noticeable, therefore we categorized this issue as minor.

Request body is not logged By introducing the `hslogger` framework as our logging tool of choice, we had to manually wire the `wai` middleware to intercept requests and log them. This did not work easily out-of-the-box. Since this was not a high priority for us, we did not invest too much time to investigate this issue. Since this is only noticeable when development mode is active on the server, this issue is only minor.

4.6 Software metrics

While there are many established software metrics available for other languages, there are only a few for Haskell [RT05]. Unfortunately, there are hardly any tools for computing metrics for Haskell to be found. The only tools we could find are *argon*[5], which is outdated, and *homplexity*[6], which does not produce easy-to-parse output.

While *homplexity* could be leveraged to produce more detailed statistics, this would require some effort. A quick run of the tool, however, gives the information that CodePanorama has only two functions that exceed a cyclomatic complexity of 10. Both of these functions are tests, so the measure is not very relevant. The highest cyclomatic complexity of any non-test function is 6, of which there are four functions.

Furthermore, 32 functions are deemed too long (based on a threshold of 20 lines per function). Lastly, 7 functions violate the default threshold of a maximum of 5 arguments, where the highest violation is 9 arguments, but only 6 in all other cases.

Table 4.1 shows simple line counts generated by Git and PowerShell and do not include empty lines or generated code.

Table 4.1: Lines by language of CodePanorama

	Avg per file	Max per file	Total
Haskell	60	393	6 990
Elm	90	609	5 417

[5] `https://hackage.haskell.org/package/argon`
[6] `https://hackage.haskell.org/package/homplexity`

5 User Studies

This chapter is a brief description on how we conducted our user studies and what insights we gained from them. We designed this study to analyze the expectations and usage patterns of experienced developers interacting with CodePanorama for the first time.

We conducted these at the beginning of this bachelor thesis. Many of the features that CodePanorama offers now were not available at that time. Hence, some of those features, for example overlays, appear as suggestions or ideas in the sections below.

5.1 Method

The interview is partitioned into three parts: a pre-interview, the practical study and a post-interview.

The handout in Appendix E. User Study Handout is intended to be printed before the user study. It contains all the questions that should be asked during the study. It is also available from the *docs* directory at the root of the CodePanorama repository.

Pre-Interview

The pre-interview questions are intended to be asked prior to the subject ever having seen or interacted with CodePanorama.

Practical Study

During the user study, the subject will be asked to perform a series of tasks without additional help. Some of the task might not be possible to complete and we want to observe how the user reacts to an impossible task, or how the user identifies that a task is not possible.

Additionally, a timer should be used to keep track of how long it takes the subject to complete each task.

The “Practical Study” page of the handout should be given to the study subjects for them to follow. The study conductor should not need to assist in any way during the practical part, but should observe and take notes on the subject’s actions.

The subject is encouraged to think aloud, or otherwise the conductor should ask questions to gain insight into the subject’s thought process.

Post-Interview

To finish off the interview, the subject will be asked questions about their perceived experience with CodePanorama, suggestions, ideas, etc.

5.2 Result

While we would have liked to interview more subjects, we only had time to conduct eight interviews. Therefore, the following results are not necessarily representative of a larger user-base.

Pre-Interview

The first thing we asked everyone during a study was, what they think when they hear the expression "Code Panorama". While some people stated that they do not really have an idea and would like to look it up on the internet, most guessed that it must be some kind of "zoomed-out" view, where a lot of code can be seen at the same time.

When asked how they would approach reviewing a large code base, the answers were all in the same vein: Look at the documentation and test cases, use an IDE or generally look at the structure of the source code.

The methods to find code smells (e.g. code duplication) are also similar and only really differ in the exact tool that is used (e.g. SonarQube, IDE-plugins). Surprisingly, not everyone was that sure whether their method really is a reliable way of finding code-smells.

Practical Study

The tasks of the practical part of the study were met with different levels of success. Some people quickly understood how CodePanorama works and how they had to use the filters and options to get to the answers. Nevertheless, there was one person who was quite a bit lost when looking at the panorama images, since they were not very familiar with the main programming language used in the test repository and thus had difficulties identifying things such as incorrect formatting of code or unusual looks of a file.

Regarding the user experience, we heard similar things from everyone: the back button of the browser is not working correctly, error messages are easy to miss, there should be some tooltips explaining how certain things work and how they are supposed to be used, and file-borders should be visible without hovering.

Post-Interview

This is likely the most interesting part of the study. Here we asked people what they think of CodePanorama in its current state, whether they would consider using it and what features they think are missing.

Quite a few of the responses for the missing features coincided with what was already outlined in the goals of this bachelor thesis or with things we discussed in meetings with our supervisor. An often named thing was, for example, that the change frequency, authors or file types should be displayed in some way, with some people already suggesting to use a colored overlay for that.

Other ideas include (in no particular order):

- Incorporation of syntax highlighting in the panorama images
- Clearer separation of files
- Single-file view on the panorama page
- Calculation of suggested panorama size
- Language-specific overlays displaying things such as number of assignments or control structures (e.g. conditions, loops).
- Option to hide comments or imports

Lastly, we asked in which use-cases/scenarios people would consider using Code Panorama. The answers mostly match with what we had already written in the abstract before we conducted the user studies: performing code reviews, get familiar with new code-bases and finding the interesting parts of a code-base.

5.3 Conclusion

Based on the findings presented above, we prioritized existing tasks, and filed new tasks and bug reports.

Most of the bugs that were reported by our test subjects have been fixed, tooltips have been added where useful and sensible, and error messages are now harder to miss and more clearly formulated.

Feature-wise, we implemented overlays for change frequency, authors and file types. We decided against implementing the suggested language-specific features. While those would be an interesting addition to CodePanorama — similar features were also implemented in *SeeSoft* [ESS92] —, they are not easy to implement and require additional knowledge about the respective language. This is, in our opinion, out-of-scope for this bachelor thesis. However, it is not out of question that such additions will be implemented in the future.

6 Conclusions

Multiple studies have been conducted on the topic of code visualization in the form of the code-map metaphor. Popularized in text editors and IDEs, the code mini-map is a well established spin-off. With CodePanorama we created a web application with a modern UI to analyze any git repository, based on the insights of previous studies and products.

In our opinion, CodePanorama does indeed provide a new angle for code reviews, with helpful color overlays. Which overlays and configurations turn out to be most practical and lead to useful results, remains to be seen.

6.1 Lessons learned

Working on a software product providing innovative functionality, we learned the importance of flexible and agile planning. In a context, where specifications and user preferences are very unclear (to both the developers and the users), it is imperative to quickly adapt to new requests and ideas. One must not be scared to try out wild ideas, even if most of them are scrapped later.

As for working on a modern web application, the impact of UI design cannot be understated. Even only a couple days of effort into making the application look and feel nicer feels equally impactful for the user like months' worth of features.

6.2 Encountered problems

Relying on established software design principles at the beginning greatly simplifies future refactorings. The impact is so large that in some cases we decided to live with the current (subpar) implementation, only because the technical debt of re-writing that part of the application was not feasible.

An example of this is the implementation of how the lines in a repository are arranged in the panorama image: Being deeply intertwined with our initial inceptions of how a code panorama should look, it became way too costly to add well-recognizable file borders to the image itself.

Finally, we knew for quite a while that our server-side implementation could not handle concurrent users on the same repository well. However, we decided to ignore this issue, in favor of hoping such a use-case would never arise.

Unfortunately, such a use-case could occur pretty often in the academic setting: The speaker demonstrates CodePanorama and multiple students follow along. In light of this, we had to spend a lot of time and effort into retrospectively making the server thread-safe. Again, this would have been much easier, if we had acknowledged this issue from the start.

7 Perspectives

There is an almost unlimited number of extensions, new features, UI improvements and other ideas available for CodePanorama. To allow anyone to extend CodePanorama with either features already conceived or with ideas of their own, we open-sourced the code.

Some of the highest impact features that have not yet made it into CodePanorama are listed below (in no particular order):

- Implement some sort of caching to improve panorama generation performance
- Allow exporting and importing of filter (and overlay) configurations
- Automatically suggest an optimal panorama size to the user
- Contributor heatmap, showing how many different contributors have worked on each file
- Rework / improve the directory filter tree
- Add visible file borders to the panorama image
- User-management where a user can see all their analyzed repositories and configurations
- Administration view to clear caches / clean up job results, etc.

We are looking forward to using CodePanorama in real-world use cases in the near future. We plan to further maintain CodePanorama as a side-project. Furthermore, we are looking for potential business partners and opportunities in the consulting area, where CodePanorama could prove a valuable asset.

References

[Abr17] Alexis Abril. *A comparison between WebSockets, server-sent events, and polling.* Sept. 2017. URL: https://aquil.io/articles/a-comparison-between-websockets-server-sent-events-and-polling (visited on 01/04/2020).

[BE19] Patrick Bächli and Marc Etter. "Code Panorama". Term Project. University of Applied Sciences Rapperswil, Mar. 2019. URL: http://eprints.hsr.ch/id/eprint/742.

[BMK18] Ivan Bacher, Brian Mac Namee, and John Kelleher. "The Code Mini-Map Visualisation: Encoding Conceptual Structures Within Source Code". In: Sept. 2018, pp. 127–131. DOI: 10.1109/VISSOFT.2018.00023.

[BNK17] Ivan Bacher., Brian Mac Namee., and John Kelleher. "The Code-Map Metaphor - A Review of Its Use Within Software Visualisations". In: *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP, (VISIGRAPP 2017).* INSTICC. SciTePress, 2017, pp. 17–28. ISBN: 978-989-758-228-8. DOI: 10.5220/0006072300170028.

[CK94] S. R. Chidamber and C. F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493. ISSN: 2326-3881. DOI: 10.1109/32.295895.

[Del+06] Robert Deline et al. "Code Thumbnails: Using Spatial Memory to Navigate Source Code". In: Jan. 2006, pp. 11–18. DOI: 10.1109/VLHCC.2006.14.

[ESS92] S. C. Eick, J. L. Steffen, and E. E. Sumner. "Seesoft-a tool for visualizing line oriented software statistics". In: *IEEE Transactions on Software Engineering* 18.11 (Nov. 1992), pp. 957–968. DOI: 10.1109/32.177365.

[FD04] J. Froehlich and Paul Dourish. "Unifying artifacts and activities in a visual tool for distributed software development teams". In: vol. 26. June 2004, pp. 387–396. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317461.

[ISO11664-5] CIE International Commission on Illumination. *Colorimetry — Part 5: CIE 1976 L*u*v* colour space and u', v' uniform chromaticity scale diagram.* Standard. Geneva, CH: International Organization for Standardization, Sept. 2016.

[Kov17] Laura Kovács. "Symbol Elimination for Program Analysis". In: Oct. 2017. URL: https://files.sri.inf.ethz.ch/website/events/workshop2017/kovacs.pdf (visited on 01/03/2020).

[Mac86] Jock Mackinlay. "Automating the Design of Graphical Presentations of Relational Information". In: *ACM Trans. Graph.* 5 (Apr. 1986), pp. 110–141. DOI: 10.1145/22949.22950.

[McC76] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 2326-3881. DOI: 10.1109/TSE.1976.233837.

[OH92] P. Oman and J. Hagemeister. “Metrics for assessing a software system’s maintainability”. In: *Proceedings Conference on Software Maintenance 1992*. Nov. 1992, pp. 337–344. DOI: 10.1109/ICSM.1992.242525.

[PF07] Charles Parkhurst and Robert Feller. “Who invented the color wheel?” In: *Color Research & Application* 7 (Mar. 2007), pp. 217–230. DOI: 10.1002/col.5080070302.

[Rob+98] George Robertson et al. “Data Mountain: Using Spatial Memory for Document Management”. In: *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*. UIST ’98. San Francisco, California, USA: Association for Computing Machinery, 1998, pp. 153–162. ISBN: 1581130341. DOI: 10.1145/288392.288596. URL: https://doi.org/10.1145/288392.288596.

[RT05] Chris Ryder and Simon Thompson. “Software Metrics: Measuring Haskell”. In: *Trends in Functional Programming*. Ed. by Marko van Eekelen. Trends in Functional Programming. Bristol, UK: Intellect Books, Sept. 2005. URL: https://kar.kent.ac.uk/14265/.

[Sea18] Owen Searls. *The Power of the Source Code Minimap*. Apr. 2018. URL: https://sites.tufts.edu/owenhumanfactors/2018/04/05/the-power-of-the-source-code-minimap/ (visited on 01/01/2020).

[Ski09] Jon Skinner. *Sublime Text 1.2 Now Available*. June 2009. URL: https://www.sublimetext.com/blog/articles/2009/06 (visited on 01/01/2020).

List of Figures

List of Tables

Glossary

API	Application Programming Interface.
CD	Continuous Delivery.
CI	Continuous Integration.
CIELUV	CIE 1976 L*u*v*.
CLI	Command-line Interface.
CSS	Cascading Style Sheets.
GHC	Glasgow Haskell Compiler.
HSR	Hochschule für Technik Rapperswil.
HSV	Hue-Saturation-Value.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
IDE	Integrated Development Environment.
IFS	Institute for Software.
JSON	JavaScript Object Notation.
lerp	Linear Interpolation.
LESS	Leaner Style Sheets.
NPM	Node Package Manager.
PNG	Portable Network Graphics.
RegEx	Regular Expression.
REST	Representational State Transfer.
SPA	Single Page Application.
SSE	Server-Sent Events.
SVG	Scalabe Vector Graphics.
UI	User Interface.
URL	Uniform Resource Locator.
UUID	Universally Unique Identifier.
XML	Extended Markup Language.

A Project Plan

A.1 Phases / Iterations

Table A.1: Phases / iterations of project plan

#	Phase	Timespan
1	Inception Brainstorming and definition of problem statement.	19.08.2019–15.09.2019
2	Elaboration Creation of project plan, priorisation of features, research of scientific material.	16.09.2019–15.10.2019
3	Construction Implementation of features and improvements. Conduct user studies and surveys.	16.10.2019–10.12.2019
4	Transition Completion of started features. Clean-up of source-code, UI and documentation. Bugfixing. Creation of final report.	11.12.2019–10.01.2020

A.2 Milestones

The milestones continue the numbering from the precursory study project, where the final milestone (Hand-In) was M5.

M6 — Project Plan

Project plan is created, milestones are defined, and the most important tasks are described. The problem statement has been created, refined, and accepted by all parties.

Planned: Completed by September 24.
Actual: Completed by September 23.

M7 — Research

Existing tools have been analyzed. Previous research on this topic has been studied. Initial documentation is created. User studies have started.

Planned: Completed by October 15.
Actual: Completed by October 15.

M8 — Improved Filtering

Filter page is overhauled both feature- and design-wise.

Planned: Completed by October 29.
Actual: Completed by October 29.

M9 — Overlays

The Code Panorama offers options to colorize the image based on user selection and pre-defined metrics.

Planned: Completed by November 19.
Actual: Completed by November 12.

M10 — Deployment Optimizations

Code Panorama can effortlessly be deployed in configurable variations, both on a hosted server and locally on a developer machine.

Planned: Completed by November 26.
Actual: Completed by November 26.

M11 — UX Improvements

General overhaul of Code Panorama's UI, based on user surveys, studies, and overall backlog.

Planned: Completed by December 24.
Actual: Completed by December 30.

M12 — Hand-In

Application is deployed to `https://codepanorama.io`. Code-base and this report have been handed-in according to all requirements.

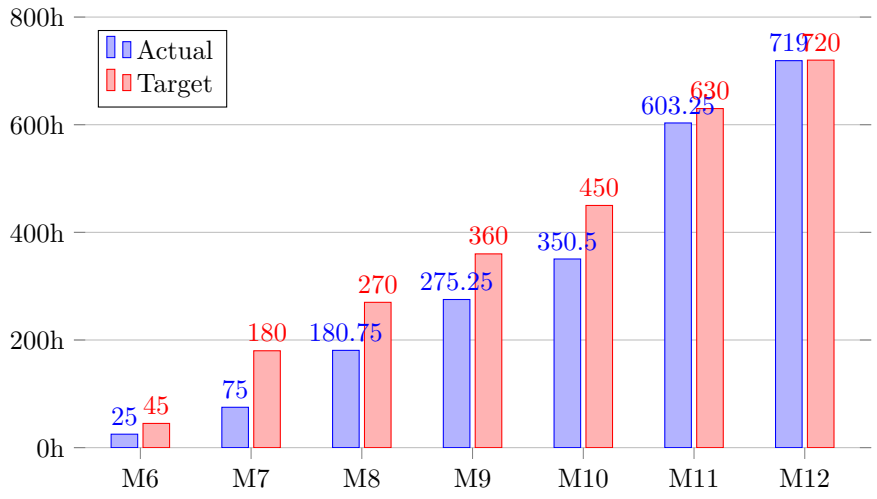
Planned: Completed by January 10.

B Time Reports

The following time reports show work logged in Jira. The target values are calculated according to the official ECTS formula of 1 ECTS = 30h [1]. Dividing the total amount of work required by the number of weeks available results in $\frac{12 \text{ ECTS} \times 30\text{h}}{16 \text{ weeks}} = 22.5$ hours per week per student.

The time reports shown below do not contain the efforts of handing in the report, preparing and conducting the final presentation, and the oral examination. We estimate the discrepancy to amount to around 4–5 hours per student.

Figure B.1: Accumulated work logged by milestone



[1] https://swisseducation.educa.ch/en/european-credit-transfer-and-accumulation-system-ects

Figure B.2: Work logged by milestone

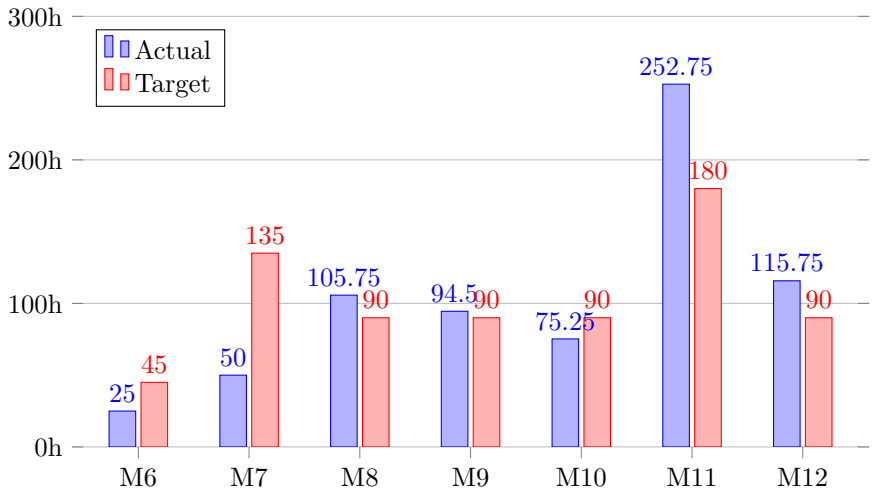


Figure B.3: Work logged by task priority

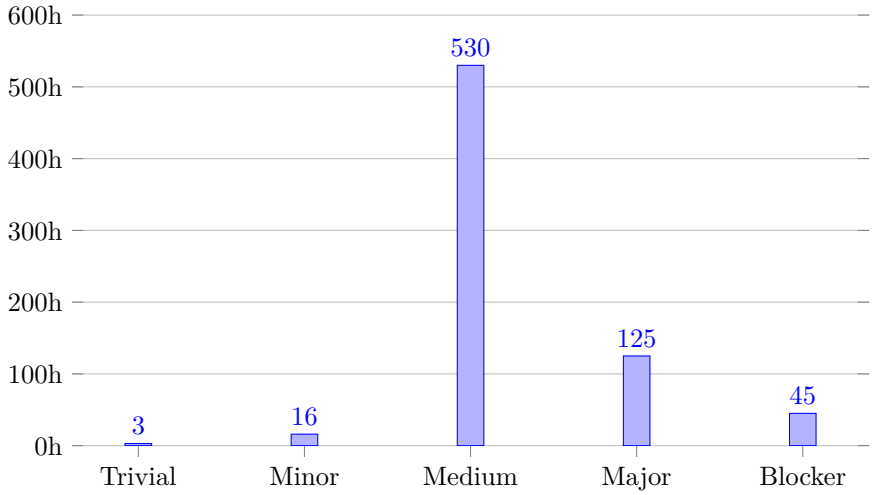


Figure B.4: Work logged by task type

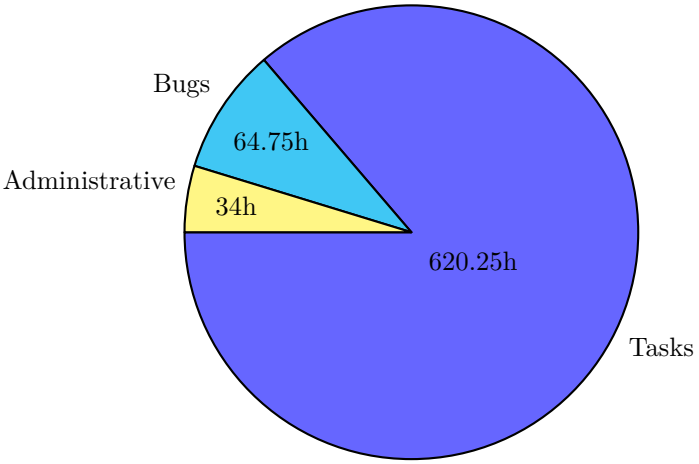
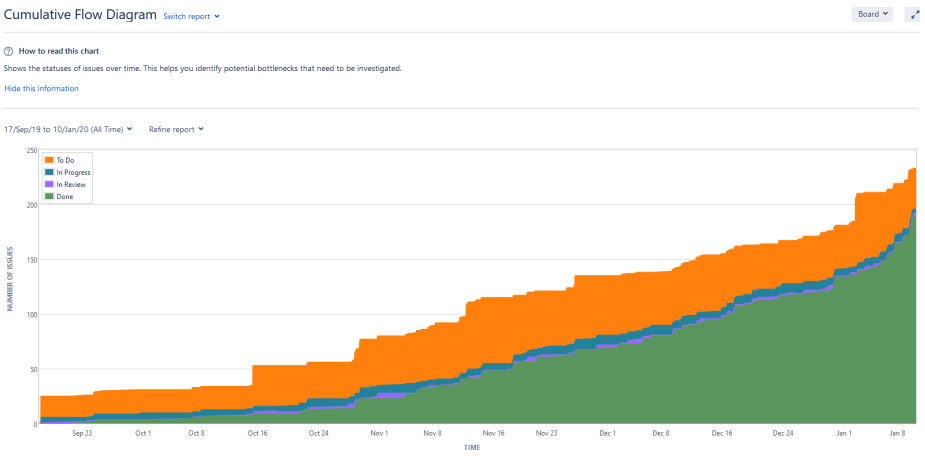


Figure B.5: Cumulative flow diagram of Jira issues



C Manuals

The contents of the sections below are generated automatically from the contents of the *docs* directory at the root of the CodePanorama repository. They are also available online at `https://codepanorama.io/#/documentation`.

C.1 Developer guide

This guide contains instructions, recommendations, and best-practices we gathered during the course of this project for working on CodePanorama and with Haskell/Elm in general.

Although it is possible to work with basically any IDE, we found Visual Studio Code to work well. This section describes our recommended setup, which we used ourselves.

Prerequisites

The following software is required to be pre-installed on your machine:
All versions are the ones used at the time of writing - no guarantees are made for newer or older versions

- Windows 10 1909 (Building should be possible on other operating systems, but hasn't been tested outside of the docker build)
- Java JRE 11+28
- NPM 6.13.4
- stack 2.1.3 (GHC is not necessary, as stack will install it's own GHC anyway)

Installation of stack might take an hour or longer, depending on your machine and internet connection!

Visual Studio Code Extensions

The following Visual Studio Code extensions worked well for us:

- Elm 0.7.4
- Haskell Syntax Highlighting 2.7.0
- haskell-linter 0.0.6
- Haskero 1.3.1
- hoogle-vscode 0.0.7
- indent-rainbow 7.4.0
- Rainbow Brackets 0.0.6
- stylish-haskell 0.0.10

Additionally, some of these extensions require external software:

```
stack install hlint
stack install intero
stack install stylish-haskell
```

Workspace Setup

For the code extensions to work properly, we found it best to use two Visual Studio Code windows, where the `CodePanorama-Server` and `CodePanorama-Client` are

used as root workspaces, respectively.

Working with the project

For convenience, we have setup `npm` shortcuts for all commonly used tasks during development. Take a look at `package.json` to see a list of all commands, but the most commonly used ones are:

`npm run build` - builds server, generates client code and builds client
`npm run test` - tests server and client
`npm run swagger` - generates swagger.json specification and client code
`npm run build-server` - only builds the server
`npm run build-client` - only builds the client
`npm run server` - starts CodePanorama-Server
`npm run client` - starts CodePanorama-Client (hot-reloading live-server) and opens a browser window with the home page
`npm run swagger-ui` - starts Swagger-UI
`npm run all` - builds and starts everything

C.2 Building docker images locally

This guide describes all steps necessary from cloning the source code to having a (local) instance up and running, by compiling and building all images locally using Docker.

Prerequisites

- You have the CodePanorama repository cloned locally.
- You have the node package manager (NPM) installed locally.
  - *tested with npm 6.11.3 (nodejs 12.11.0)*
- Docker is installed on your system
  - *tested with Docker for Windows 18.06.1-ce*
- Your docker daemon is configured with at least 4GB of memory
  - Linux: *no configuration necessary, as long as your system has enough memory*
  - Mac: https://docs.docker.com/docker-for-mac/#resources
  - Windows: https://docs.docker.com/docker-for-windows/#advanced

Setup

A couple of local NPM dependencies are required for the build steps to succeed. These are simply installed as follows:

```
npm install
```

Now you are setup to create all the docker images!

Create build images

Note: This step might take 10-20 minutes, as many dependencies will have to be downloaded.

In this step, the docker images required to build the application will be built.

```
npm run docker-build
```

Or if you only want to build the server/client:

```
npm run docker-server-build
npm run docker-client-build
```

Create deploy images

Now that the images required for building the application are ready, the deployment images can be built:

```
npm run docker-deploy
```

Or, again, building the server/client separately:

```
npm run docker-server-deploy
npm run docker-client-deploy
```

Run images locally

Finally, the deployment images can be run locally:

```
npm run docker-local
```

This deploys the application as follows:

Component	URL
Client (UI)	http://localhost:8080
Server (API)	http://localhost:6868
Swagger-UI	http://localhost:6869

See the compose file and the server README for details and configuration options, e.g. if you need to modify the port bindings.

You can shutdown the local images by issuing the following command:

```
npm run docker-local-stop
```

Persisting the workspace

If you want to keep the server's workspace (i.e. repo clones, panorama PNGs, and repository infos) persistent, you can add the following to the docker-compose file at the end of the `code-panorama-server` section (replacing `C:/dev/CodePanorama/ws` with any path on your machine where you would like to save the server's workspace):

```
volumes:
  - "C:/dev/CodePanorama/ws:/opt/ws/workspace"
```

Run image on a server

If you wish to deploy CodePanorama to your own server using the above built docker images, follow these steps (after building the deploy images as above):

```
npm run docker-prd
```

This command will tag the previously built deploy-images as `latest`. Make any desired configuration changes (e.g. your server's domain name) to the `docker-compose-prd.yml`. Now, you should be able to deploy the application to your server using this compose file:

```
docker-compose --file docker-compose-prd.yml up -d
```

The command above needs to be run on the server you wish to deploy to. Additionally, the locally built images need to be pushed to the server.

C.3 Deploying CodePanorama with docker

This guide describes all steps necessary to run CodePanorama locally (or on your own server) by using the pre-built Docker images.

Prerequisites

- Docker is installed on your system
  - *tested with Docker for Windows 18.06.1-ce*

Using Docker-Compose

If you have docker-compose installed, the most straight-forward way to deploy CodePanorama locally, is as follows:

Create a `docker-compose.yml` file on your system with the following contents (or download it here):

```yaml
version: "3.7"

services:
  code-panorama-server:
    image: hsrcodepanorama/code-panorama-server:${TAG:-latest}
    command: "/opt/ws/CodePanorama-Server-exe"
    environment:
      CP_LOCAL_REPOS_ENABLED: "true"
      CP_PRIVATE_REPOS_ENABLED: "true"
    ports:
      - "6868:6868"
    # Use the volume mapping below to persist the workspace
    #volumes:
    #  - "C:/dev/CodePanorama/ws:/opt/ws/workspace"

  code-panorama-swagger-ui:
    image: hsrcodepanorama/code-panorama-server:${TAG:-latest}
    command: "/opt/ws/CodePanorama-SwaggerUI-exe"
    environment:
      CP_API_HOST: localhost:6868
    ports:
      - "6869:6869"

  code-panorama-client:
    image: hsrcodepanorama/code-panorama-client:${TAG:-latest}
    environment:
      API_BASE_PATH: http://localhost:6868
    ports:
      - "8080:80"
```

Then start the services as follows:

```
docker-compose -f path/to/your/docker-compose.yml up -d
```

The application should now be deployed according to the following table:

Component	URL
Client (UI)	http://localhost:8080
Server (API)	http://localhost:6868
Swagger-UI	http://localhost:6869

You can shutdown the services with the following command:

```
docker-compose -f path/to/your/docker-compose.yml down
```

Using plain Docker

The following commands replicate the behavior of the compose file above:

Running the Server

```
docker run -d --name code-panorama-server -e
↪  CP_PRIVATE_REPOS_ENABLED="true" -p "6868:6868"
↪  hsrcodepanorama/code-panorama-server
↪  /opt/ws/CodePanorama-Server-exe
```

Running the Client

```
docker run -d --name code-panorama-client -e
↪  API_BASE_PATH="http://localhost:6868" -p "8080:80"
↪  hsrcodepanorama/code-panorama-client
```

Running the Swagger UI

```
docker run -d --name code-panorama-swagger-ui -e
↪  CP_API_HOST="localhost:6868" -p "6869:6869"
↪  hsrcodepanorama/code-panorama-server
↪  /opt/ws/CodePanorama-SwaggerUI-exe
```

C.4 Running your own instance

This manual describes how to run a local instance of CodePanorama, e.g. if you want to keep your private repositories truly private (which we can totally understand).

If you intend to make changes to the code, we recommend using the *Local* approach described below. If you simply want to run your own instance on your local machine or on your own server, we recommend using the *Docker* approach.

Prerequisites

The following software is required to be pre-installed on your machine:
All versions are the ones used at the time of writing - no guarantees are made for newer or older versions

- Windows 10 1909 (Building should be possible on other operating systems, but hasn't been tested outside of the docker build)
- Java JRE 11+28
- NPM 6.13.4
- stack 2.1.3 (GHC is not necessary, as stack will install it's own GHC anyway)

Installation of stack might take an hour or longer, depending on your machine and internet connection!

Build

The simplest way to build CodePanorama is to use the following commands from the root of the cloned project:

Warning: The first build might take multiple minutes, as all dependencies are downloaded!

```
npm install
npm run build
```

Deploy

The easiest way to start the locally built instance is:

```
npm run all
```

This will start multiple processes to run the CodePanorama-Server, Swagger-UI, elm-live server for the CodePanorama-Client, as well as less-file-watch. This enables hot-reloading any changes to files in `CodePanorama-Client` and manual testing of the API through Swagger-UI.

A browser window/tab should open automatically, with CodePanorama running. If not, open http://localhost:8000. Swagger-UI will be running at http://localhost:6869 and CodePanorama-Server at http://localhost:6868.

If you do not wish to use any development features (such as extended logging, hot-reloading, etc.), use the following to start the server:

```
cd CodePanorama-Server
stack exec CodePanorama-Server-exe
```

Then, open `CodePanorama-Client/public/index.html` in your browser of choice.

Deploying to your own server

If you wish to deploy CodePanorama to your own server (i.e. not on localhost), the following change needs to be made, so that the client will find your backend:

1. Edit the `apiBasePath` in `CodePanorama-Client/env/prd.js` to point to your own server/domain, where CodePanorama-Server will be running.
2. Build the client code (`npm run build-client-ci`).
3. Apply the `prd` configuration (`npm run env-prd`).

Alternatively, you can edit the `apiBasePath` in `CodePanorama-Client/public/out/env.js` instead, but this will be overwritten, if you build the client code anew - but may be quicker, if you simply want to change the `apiBasePath` in a pre-built distribution.

No changes are necessary to deploy CodePanorama-Server to your own server.

C.5 Using custom overlays

This guide describes how custom overlays can be created and used with CodePanorama.

Usage

A custom overlay can be created by creating a json file according to the specification below. Once such a file has been created it has to be saved as `custom.overlay.json`, where `custom` can be replaced with a name of your choosing. Move this file into the root directory of your repository and it will then be parsed automatically and sent to the client as a possible option for the *Custom overlay*-selection.

Custom overlay generators

It is also possible to implement custom overlay generators. For one such generator, please see the JaCoCo-generator and its description here.

Example

```
{
  "description": "This is a description of this overlay.",
  "lineSpecifications": [
    {
      "lineStart": 1,
      "lineEnd": null,
      "file": "src/main/java/org/example/Main.java",
      "schemeValue": 0,
      "colorOverride": null
    },
    {
      "lineStart": 2,
      "lineEnd": null,
      "file": "src/main/java/org/example/Main.java",
      "schemeValue": 1,
      "colorOverride": null
    }
    {
      "lineStart": 3,
      "lineEnd": null,
      "file": "src/main/java/org/example/Main.java",
      "schemeValue": null,
      "colorOverride": {
        "red": 0,
        "green": 0,
```

```
        "blue": 255,
        "alpha": 1
      }
    }
  ],
  "colorScheme": {
    "solid": {
      "colors": [
        {
        "red": 255,
        "green": 0,
        "blue": 0,
        "alpha": 1
        }
      ]
    },
    "gradient": {
      "start": {
        "red": 67,
          "alpha": 1,
          "green": 198,
          "blue": 172
        },
      "end": {
        "red": 25,
        "alpha": 1,
        "green": 22,
        "blue": 84
      },
      "midPoints": [
        {
          "percentage": 0.5,
          "color": {
            "red": 198,
            "alpha": 1,
            "green": 67,
            "blue": 129
          }
        }
      ]
    }
  }
}
```

Specification

Below you find descriptions for the properties used in the example above.

It is to be noted that it is not possible to use a gradient- and a solid-scheme at the same time. Also, some of the features are already defined and represented in the data-types but haven't been implemented yet (marked below).

Top-level

Property	Data type	Description
description	string	Text describing this overlay (e.g. meaning of colors used).
lineSpecifications	array	Array of line specifications.
colorScheme	object	Color scheme.

Line specification

Property	Data type	Description
lineStart	integer	First line this specification will be applied to. Lines are counted starting at 1.
lineEnd	integer	Last line this specification will be applied to. *(currently not implemented)*
file	string	File this specification will be applied to.
schemeValue	float	Gradient-scheme: Point on the gradient in the range [0,1]. Will be between the start- and end-color and possible mid-points. Solid-scheme: 0-based index of the color to be used.
colorOverride	object	Color to override the scheme color. *(currently not implemented)*

Color scheme

Property	Data type	Description
solid	object	Definition of a solid scheme.
gradient	object	Definition of a gradient scheme. *(currently not implemented)*

Solid scheme

Property	Data type	Description
colors	array	Array of colors.

Gradient scheme

Property	Data type	Description
start	color	First color of the gradient.
end	color	Last color of the gradient.
midPoints	array	Array of mid-points for the gradient.

Mid-point

Property	Data type	Description
percentage	float	Point on the gradient at which this color is located in the range [0,1].
color	color	Color of this mid-point.

Color

Property	Data type	Description
red	integer	Red component in the range [0,255].
green	integer	Green component in the range [0,255].
blue	integer	Blue component in the range [0,255].
alpha	float	Alpha value in the range [0,1].

C.6 Using local repositories

This guide describe all steps necessary to run CodePanorama with local repositories.

This features provides users with the possibility to locally analyze repositories that cannot be cloned over a network (e.g. due to security restrictions). It also offers running analyses on directories that are not git repositories (i.e. contain no `.git`-folder) with the limitation that only the non-git parts of CodePanorama can be used.

Prerequisites

This feature is the easiest to use if you run CodePanorama itself locally. Please refer to *Run with Docker* or *Run with Stack* to read up on how to do that.

Using Docker

If you are running CodePanorama using Docker, you have to mount a host-directory into the `local-repos`-directory in the container. You can do this by adding the following to the docker-compose file at the end of the `code-panorama-server` section (replacing `C:/dev/CodePanorama/local-repos` with the path on your machine where your local repositories are stored):

```
volumes:
  - "C:/dev/CodePanorama/local-repos:/opt/ws/local-repos"
```

If you are working with *Docker for Windows*, you also need to enable drive-sharing for the host-drive you would like to use.

Using Stack

If you are running CodePanorama using Stack, you can simply copy or link a directory into `CodePanorama-Server/local-repos`.

Configuration

To actually be able to use local repositories, you have to turn that feature on by setting the environment variable `CP_LOCAL_REPOS_ENABLED` to `true`.

You can also configure where exactly CodePanorama will look for the local repositories by changing the value of the environment variable `CP_LOCAL_REPOS_DIR` (defaults to `local-repos`).

D Specifications

This section specifies various features to be integrated into Code Panorama. Not all of these features are planned to be integrated and this section does not list the specifications in any particular order (specifically not in priority).

D.1 Filters

A filter is a configuration available to the user. This configuration specifies which files will be included in or excluded from the code panorama. Multiple filters can be combined, where only files passing all selected filters will be included.

Filetype

This filter allows the user to include or exclude all files of the same file type. Only the file extension (e.g. ".txt") is considered for this filter, not the content.

Directory

The user can limit the code panorama to only include files from certain directories. Only directories, but not individual files can be filtered using the directory tree.

Line number

The user can specify the minimum and maximum number of lines a file must contain. Otherwise, the file is excluded.

Commit number

The user can specify the minimum and maximum number of git commits a file must have been part of. Otherwise, the file is excluded.

Git branch / tag

The user is presented with a dropdown list of branches and tags automatically populated from the repository data. Upon selection of a different branch or tag and confirming the selection, all displayed data is updated to the state on the selected branch or tag.

Regular expression

The user can specify regular expressions to either include or exclude files based on the file name.

D.2 Overlays

An overlay adds color information on top of a code panorama. An overlay might either color individual lines separately, or entire files uniformly. Overlays are

sorted by category:

General

This category contains overlays where the color itself does not necessarily contain information. Instead, the colors are simply used to group or differentiate files based on some criterion.

File type

The file type overlay groups all files by file extension.

Heatmap

Heatmaps are overlays using colors in a specific range, where the hue or saturation represents a high or low value according to the selected metric.

Change frequency

This heatmap shows how often files have been changed according to the Git history. The user can enter a start date. The heatmap then shows the number of commits a file has been part of since that date. If the user selects the "normalize by age" option, the heatmap value is calculated as follows:

$$\text{value} = \frac{\text{number of commits} \times \text{seconds since start date}}{\min(\text{seconds since file creation}, \text{seconds since start date})}$$

Due to technical limitations of Git, the change frequency can only reasonably be retrieved on the file-level and not on the line-level. E.g. how should a line be tracked if the line number changes due to insertions/deletions of other lines?

Number of contributors

This heatmap shows how many different authors have contributed to a given file according to the Git history.

Latest change

This heatmap shows how recently a file has been changed.

Age

This heatmap shows how long a file has existed.

Highlight

Highlights are overlays where only a subset of files or lines are emphasized.

Custom

Users can deposit custom overlay specifications either directly in the repository, or upload them through the UI.This user-defined overlay is then displayed as specified.

For example, a code coverage report could be converted to a custom overlay by the user and then displayed over the code panorama.

Author

The user is presented with a dropdown list of all contributors of this repository. This overlay can then be displayed in one of two different ways: By frequency, or by latest change.

By frequency. This variant highlights all files where the selected author(s) has/have contributed at least one commit. The highlight is colored similar to a heatmap showing how many commits the selected author(s) has/have contributed to each file. The number of commits per file are summed over all selected authors to determine the color.

By latest change. This variant assigns a unique color to every selected author. Then, every line is highlighted with the author's color where the author has contributed the latest change. Due to usability limitations with color contrasts, this variant limits the maximum number of selected authors.

Regular expression

The user can enter a regular expression in a text input. All files with a file name matching the regular expression are then highlighted.

E User Study Handout

Code Panorama — Pre-Interview

These questions should be asked and documented by the study conductor.

What do you think of when you hear the expression "Code Panorama"?

How would you approach reviewing a large code base? Which tools would you use to help you?

What are, in your opinion, the most significant code smells? (i.e. strong indicators of very bad code)

How would you identify code duplication?

Which, if any, are — in your opinion — language-agnostic code smells?

Do you think your reviewing approach reliably finds the above mentioned code smells?

Code Panorama — Practical User Study

Name *(optional)*: _____ Date: _

Please complete the following tasks using only the website `https://codepanorama.io`.
Note: If you think a task is impossible, please mark the task as such and move on to the next task.

How many lines does the longest Java file in the repository `https://github.com/teiler/api.teiler.io` have? _____

On the same repository — without looking at the source code — find a code section with...
(please note the files / sections with an explanation)
... duplicated or boilerplate code

... a very high complexity

... incorrect / inconsistent formatting

... a (in your opinion) too long function

... an unusual / unexpected look (for its file type)

Does looking at the source code validate your suspicions?

Create the largest possible code panorama of `https://gitlab.com/Elewyth/session-summary-generator` containing only (but all) JavaScript files. Document the settings used:

Create a code panorama of the same repo on the branch 'redux'. Document the settings used:

How many Haskell source files (.hs) does the repository `http://gitlab.com/Elewyth/gitlab-discord-middleware` contain? _____
Use the following credentials:
Username: CodePanorama
Password: codepanoramaHS19

Code Panorama — Notes on Practical Study

Longest Java file:

Duplicated / boilerplate code:

High complexity:

Incorrect formatting:

Too long function:

Unusual look:

Suspicions validated?

Large JavaScript panorama:

Impossible branch panorama:

Private repository:

Code Panorama — Post-Interview

These questions should be asked and documented by the study conductor.

Which code smells do you think are more easily identified using Code Panorama (as opposed to other review approaches)?

Which tasks did you find intuitive and useful, which less so?

Which visual information / indicators would you find useful in the Code Panorama?

What features do you think Code Panorama would need additionally?

In which use-cases / scenarios would you consider using Code Panorama?

Other Notes:

F Self Reflection

F.1 Report by Marc Etter

Ever since the completion of the term project and the CodePanorama prototype, I had been looking forward to this bachelor thesis. It felt satisfying to devote a large amount of time to this software product and conduct real user studies. Diving into the research topic revealed highly interesting work done previously in the same field.

Although I have worked on multiple enterprise software projects, this bachelor thesis was the largest project where I was (partly) responsible for the entire project planning and organization. Repeatedly having to plan, assess, and re-prioritize the product backlog proved a challenge, but an important experience.

As for the technologies used, after this bachelor thesis I feel competent enough in both Haskell and Elm that I can confidently add these languages to my résumé. I would not be afraid to apply for a job using either of these languages.

I am looking forward to receive feedback on CodePanorama from our advisor. Furthermore, I will look out for business opportunities to employ CodePanorama in my consultant jobs.

F.2 Report by Patrick Bächli

From the moment we generated our first code panorama as it looks like today, I thought to myself, “This is something I want to continue working on”. Understandably, I was excited when our supervisor asked us, if we would like to pick up where we left off and work on CodePanorama as our bachelor thesis, and even more so when we finally started.

I have worked on some software projects in the past. However, none of those really required any kind of literary research or well thought out user studies. Searching on the internet for similar software, and digging through various papers was highly interesting to me and I was surprised at the amount of research that had already gone into topics like the code-map metaphor. Conducting the user studies and evaluating the results reassured me that we are working on something that people would like to use for their daily work.

Working with Haskell and Elm has also given me some exercise and I feel more confident in using both of these languages. Nevertheless, I think there is still quite some room for improvement and I will try to further hone my skills in these areas.

Now that we are on the home straight of this bachelor thesis, I am looking forward to hear what our supervisor and external examiner think of CodePanorama. Moreover, I would like to continue working on CodePanorama in the future and also try to incorporate it into the workflows used at my current job.

G Meeting Minutes

The meeting minutes are stored in a private wiki and have thus been exported to make them available in the following sections.

G.1 September 17, 2019

☒ **Finalisation of the formal task description (should be done by then)**

- What are possible additions that could be implemented in the future? E.g. colourized filters.
- Focus on finalising UI and workflow, "polishing" of the application.

☒ **Questions regarding role MeF and project information page https://wiki.hsr.ch/FarhadMehta/wiki.cgi?ProjectInformation**

☒ **General availability for meetings**

- Generally and preferably on tuesdays from 0815 to 0900

☒ **Date and time for next meeting**

- 01.10.2019, 1315-1400

- Issue-management and time-tracking
  - JIRA for issue-management and time-tracking
  - JIRA guest-login for MeF
- Repository
  - IFS-Gitlab for WIP
  - gitlab.com for "releases" (currently every push on master is synchronised)
- Docker-images don't work "out-of-the-box" on Mac

G.2 October 1, 2019

☒ **Wann ist der Abgabetermin?** -> 06.01. Erfassung des Abstracts, 10.01. 17:00 Abgabe Bericht + Dokumente (siehe Skript-Server: Bachelor-Arbeit_Informatik/BAI14/Termine)

☒ **CI-Integration** -> Erst wenn alles andere sauber läuft (auf allen Plattformen)

- Christian Spielmann => MacOS-Testsystem
- Benutzbare Cmd-Applikation wäre gut (aber ohne das vollständige Featureset der Website)
- Einfärbung von Sonderzeichen eher nicht, wenn dann eher eine sprachspezifische Einfärbung von Sonderzeichen und Keywords
- Evt. User-Management nicht einbauen, dafür mehr Fokus auf Bedienbarkeit der Web-App sowie Lauffähigkeit der Docker-Images auf allen Plattformen

G.3 October 15, 2019

☒ **Aufgabenstellung unterschreiben? (Steht so in der Anleitung für Dokumentation)**

- Gedruckte Version zum Unterschreiben nächstes Mal mitnehmen

☒ **Welches Lizenzierungsmodell soll für abschliessendes Publishing gewählt werden? (Open-Source MIT? Report CC-BY?)**

- Formular von MeF-Wiki für Vereinbarung
- Das Wie zu OpenSource gegen Ende des Projekts nochmals anschauen
- Grundsätzlich sollen alle die Applikation weiterentwickeln können

☒ **Abbildungs- und Tabellenverzeichnis vor dem Inhalt? Oder nach dem Inhalt / vor dem Quellenverzeichnis?**

- Am Schluss nochmals schauen, grundsätzlich beides in Ordnung

☒ **Priorisierung von Filter festlegen**

- Directory, Git branch / tag, Regular expression, Filetype
- Branch / Tag auswählen als neuen Schritt zwischen Repository-Auswahl und Filter-Seite

☒ **User Study Questionnaire besprechen**

☒ **Evtl. Priorisierung von Overlays festlegen**

- Overlays direkt auf Zeilenebene statt Dateiebene
- Möglichkeit, die Overlays per Konfigurationsdatei auszuwählen / einzustellen
- Berechnung für Heatmapwerte einfach erweiterbar / überschreibbar machen

☒ **Paging bei grossen Panoramas?**

- Grundsätzlich mit Auflösung des Clients arbeiten, damit die “ideale” Menge von Zeilen angezeigt wird
- Dem Benutzer vorausrechnen, wie viele Seiten bei der gewählten Panoramagrösse entstehen würden
- Evt. auf dem Server die ideale Panoramagrösse berechnen, damit alles auf eine Seite passt
- In Projektplan aufnehmen

- Erweiterungen von Code Panorama sollen im Code vorgenommen werden; eine Erweiterung zur Laufzeit (z.B. mit zusätzlichen Executables) ist nicht vorgesehen
- Keine CLI-Applikaiton, Fokus auf einfachem lokalem Deployment

- Deployments
  - Hosted docker (keine privaten Repositories möglich)
  - Local docker (mit Einbindung von Host-Filesystem)
  - Local executable
- Research
  - Für Seesoft evaluieren, wie weit verbreitet dass es ist und ob es gute Bewertung / Reviews hat
  - Evt. Entwicklungsumgebungen wie VSCode referenzieren, da dort häufig auch eine Art Code Panorama angezeigt wird
  - SE-Vorlesungen nochmals anschauen

G.4 October 28, 2019

☒ **Aufgabenstellung unterschreiben**

- Farhad's Unterschrift genügt

☒ **Filter-Demo**

- Commit Hash zum Branch/Tag hinzufügen, damit man weiss, ob der Branch aktuell ist
- Auswahl von Directory sollte alle Sub-Directories auswählen und dann muss man ungewünschte Sub-Directories wieder abwählen
- Export Filter (als RegEx für "Include" field, maybe as JSON to restore checkbox state) button
- Idee: Panorama-Preview als fixed Sidebar
- "Progress" bar / gauge für "Vollheit" des Panoramas

☒ **Overlay Priorisierung**

- Priorisierung gut wie im JIRA -> wichtig: 1 heatmap und 1 highlight

---

-> Custom Import: Code Coverage (e.g. lcov) anschauen, Format ähnlich aufbauen und evtl. Konvertierungstool für 1 Format -> Möglichst 1 einheitliches, abstraktes Format für intermediate overlay Format (keine technische Unterscheidung zwischen Highlight und Heatmap)

G.5 November 12, 2019

- ☒ **Look at time tracking so far**
- ☒ **Paging Demo**
- ☒ **Overlay Demo**

Feedback Usability: * Show file-borders * Re-think file-ordering (e.g. (root) files before directories, BFS instead of DFS) * Allow multiple custom overlays (with names) * Allow adding color hint to overlay.json that can be displayed as info in overlay-selection * Customize Panorama Size (in pixels) in Client (e.g. input explicit height/width) * Add button to download all generated PNGs as a ZIP * Make maximum number of pages configurable (via UI, but maybe bounded by deployment configuration)

- Farhad provides repo access to Mr. Kerckhove on gitlab.com
- Deployment test subjects:
  - Mario Meili (Mac)
  - Fabian Hauser (Windows/Linux)

G.6 November 26, 2019

Meeting

- Push Docker-images into DockerHub

Mid-term presentation

Overlays:

- Syntax highlighting
- Project-wide grep to search for specific pieces of text within source-code
- How many people have worked on a file (one contributer: red, more contributors: green)
- Colour-blind-friendly colouring / Maybe use patterns (triangles, stripes etc.)
- Hover on colour-legend highlights corresponding files in the Panorama
- Change frequency on a per-line basis (git blame, look at how GitHub does it)

Other things:

- Use local-directory instead of repository-url
- Nix-build
- Store repository-data in-memory instead of JSON-files
- Timeout on really large repository
- Shallow clone on large repository
- Increase maintainability so other people could add things

For final presentation:

- How did we do testing?

G.7 December 10, 2019

- ☒ **Demo of current state**
- ☒ **Discuss priorities for last development weeks**
  - Git blame overlay
  - Overlay documentation
- ☒ **Re-design panorama with "grid" borders? (look at design mockup)**
  - Priority on maintainability and documentation
- ☒ **Any last important formalities for hand-in?**

G.8 December 23, 2019

☒ **Demo current state**

- Add hint to download button on how the pictures are structured
- Move API Link from footer to development documentation
- Custom Overlay: At least add documentation for color scheme to JSON - actually displaying it in UI is secondary
- Concurrency must be fixed

☒ **Any last-minute requests?**

- Add (very brief) Quick-Start guide to UI

☒ **Further proceedings**

- Only digital report necessary, no print or CD
- Review abstract directly with Farhad before uploading to ABT

G.9 January 7, 2020

☒ **Look at report outline** => looks good
☒ **Look at time sheets: how detailed in report?** => re-use gitlab.com as if it were public, but we actually make it public at a later date
☒ **OpenSource: New repo, or make existing gitlab.com repo public?** => Just want to see what we did concerning project management

---

- If possible, add simple color selection (at least for inversion of base image)
- Hide local repos tab, if not enables (e.g. on HSR)