CROSSMINER
Developer-Centric Knowledge Mining from Large Open-Source Software Repositories

Project Number 732223

D7.9 Web-based dashboards - Final Version

Version 1.0
27 June 2019
Final

Public Distribution

Bitergia

Project Partners: **Athens University of Economics & Business, Bitergia, Castalia Solutions, Centrum Wiskunde & Informatica, Eclipse Foundation Europe, Edge Hill University, FrontEndART, OW2, SOFTEAM, The Open Group, University of L'Aquila, University of York, Unparallel Innovation**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the CROSSMINER Project Partners accept no liability for any error or omission in the same.

© 2019 Copyright in this document remains vested in the CROSSMINER Project Partners.

PROJECT PARTNER CONTACT INFORMATION

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

DOCUMENT CONTROL

Version	Status	Date
0.1	Template for preparing Web based dashboard report	20 June 2019
0.5	First iteration	21 June 2019
0.8	Integrating partner review comments	25 June 2019
1.0	Final version integrating remaining comments	27 June 2019

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This deliverable provides the final version of the Web-based dashboards, developed in Task 7.4, that aim at presenting up to date, high level, and quantitative panoramic views of projects analysed with CROSSMINER technologies. The dashboards include specific visualizations for tracking key performance aspects and show details like who and how is contributing to a given project. Visualizations show summary, aggregated and evolutionary data, statistical analysis, faceted views, etc. for each project. The interface allows to drill down into the data combining different filters such as time ranges and projects.

This deliverable is structured as follows. Section 1 provides some context about the work done, Section 2 describes the approach underlying the creation of the Web-dashboards, Section 3 presents the Web-dashboards developed in the context of CROSSMINER, Section 4 concludes the report with a discussion on the requirements defined in D1.1.

1. INTRODUCTION

Web based dashboards are a critical piece needed to perform analytics for software projects. Bitergia is providing right now a complete stack for software analytics (GrimoireLab[1]) and with it, Bitergia helps its customers (the more exposed stakeholders that use or produce free open source software like non-profit foundations and large companies) to understand better how they are developing software, and specially, how they are developing Open Source.

Some companies and organisations are already using Bitergia Open Source Software Development monitoring tools. By means of them, Bitergia customers are able to get insights and metrics for the open source and inner source software projects in which they are involved through quantitative analysis of software development.

Bitergia has been continuously extending its services with the support of new data sources and metrics. Aligned with this goal, the integration of the analysis tools provided by CROSSMINER and its metrics are of great value for Bitergia. Such an integration will improve the current Bitergia services and provide new ones.

In order to achieve the above goals, a bridge has been built to connect CROSSMINER and GrimoireLab. Then, metrics, factoids and additional data provided by CROSSMINER have been shown through Web-based dashboards.

A Web-dashboard includes a set of visualizations, which span from common ones such as bar, line and pie charts to more sophisticated ones such as networks and heat maps. A dashboard offers plenty of functionalities to better understand the data visualized, for instance it allows to apply filters to focus on (i) a subset of projects, (ii) specific project attributes or (iii) given time ranges. Furthermore, all visualizations can be easily exported to CSV to perform analysis beyond the ones provided by the Web-dashboard. An example of Web-dashboard is shown below. It includes filters (on the top left corner), and several visualizations (summaries, heat map and bar charts) that allow to explore the contributors engagements around GrimoireLab.

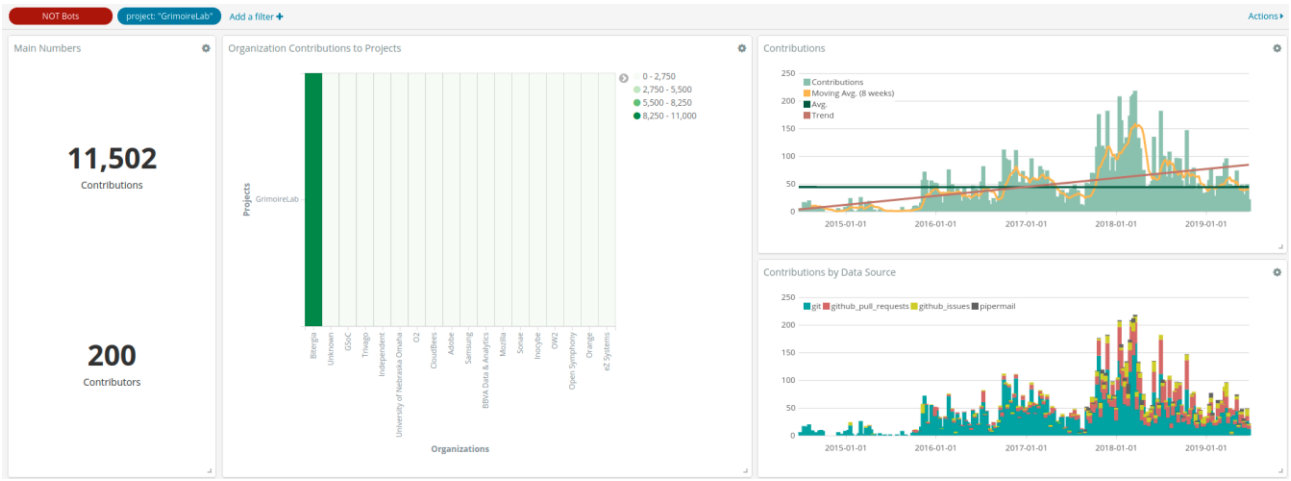


Figure 1: Example of Web-based dashboard taken from https://chaoss.biterg.io.

[1] https://grimoirelab.github.io/

2. APPROACH

This section shows how the Web-based dashboards conceived in CROSSMINER have been developed by relying on GrimoireLab, a mature and popular open source platform for software analytics. The rest of this section presents GrimoireLab and the integration with CROSSMINER.

2.1 GRIMOIRELAB

The overall structure of GrimoireLab is summarized in Figure 2. Its core is composed of four tasks: extracting software development data, storing it, managing contributor identities and analyzing (visualizing) the data obtained. Additionally, orchestration of all the tasks is also available. The details of each of the tasks, and the mapping to components, are described in the next sections.

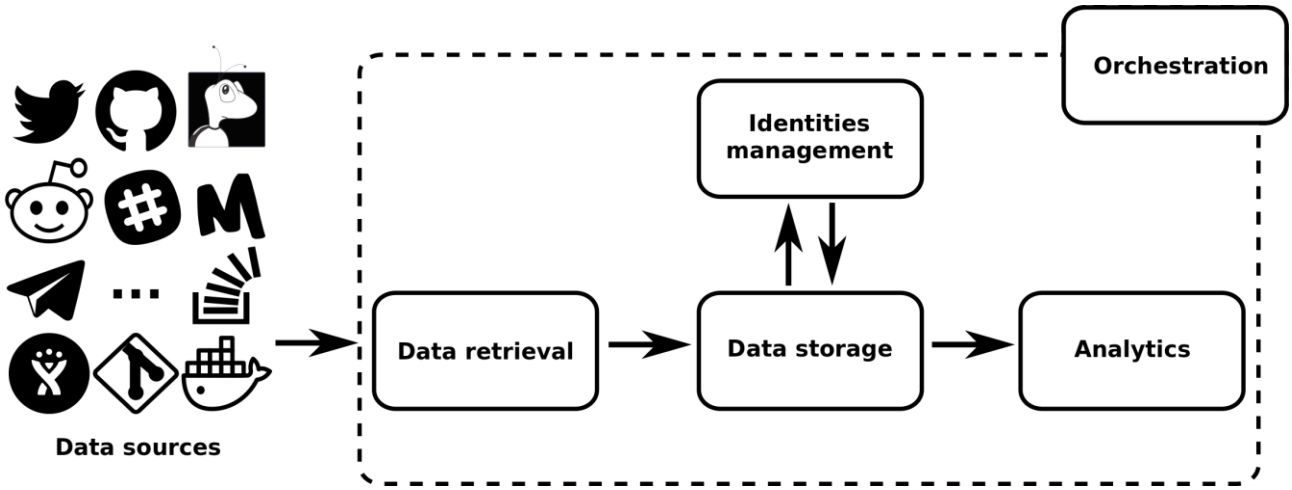


Figure 2: Overview of GrimoireLab. Software data is extracted, processed and visualized via four components, with one of them dedicated to managing identities. A further component allows to perform analysis over a set of target data sources by setting up and orchestrating together the other components.

2.1.1

2.1.2 Data retrieval

Data retrieval has been split in two components. The first one, Perceval[2], is designed to deal only with fetching the data, so that it can be optimized for that task. Usually, it works as a library, providing a uniform Python API to access software development repositories. The second one King Arthur[3], schedules and run Perceval jobs at scale through distributed queues.
Perceval provides access to the following data sources:

- Version control systems: Git.
- Source code review systems: Gerrit, GitHub.
- Bugs/Ticketing tools: Bugzilla, GitHub, JIRA, Launchpad, Phabricator, Redmine, GitLab.
- Mailing: Hyperkitty, MBox archives, NNTP, Pipermail, Groups.io
- News: RSS, NNTP
- Continuous integration: Jenkins
- Instant messaging: Slack, Supybot archives (IRC), Telegram
- Q/A: Askbot, Discourse, StackExchange

[2] https://github.com/chaoss/grimoirelab-perceval
[3] https://github.com/chaoss/grimoirelab-kingarthur

- Documentation: Confluence, Mediawiki
- Others: DockerHub, Meetup, Twitter

A common Perceval job consists of fetching a collection of homogeneous items (i.e., categories) from a given data source: tickets extracted from Bugzilla or GitHub issue trackers, commits from a git repository, or code reviews from a Gerrit instance. Each item is inflated with related information (e.g., comments and authors of a GitHub issue) obtained from the data source, and metadata useful for debugging and tracing (e.g., backend version and timestamp of the execution). The output of the execution is a list of Python dictionaries (or JSON documents), one per item.

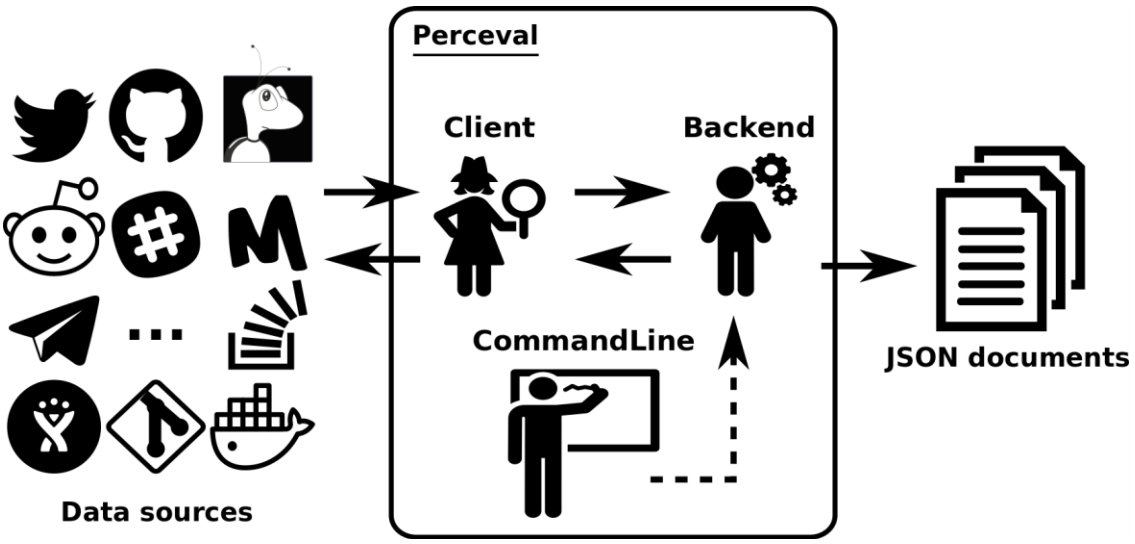


Figure 3: Overview of Perceval. The user interacts with the backend through the shell command line, which depending on the data source, retrieves the data with a specific client; the output is provided in form of JSON documents.

Perceval's design is shown in Figure 3. For each data source, it includes a Client, a Backend, and a CommandLine class. Backend organizes the gathering process for a specific data source. Backends share common features, such as incrementality and caching, and define also specific ones tailored to the data source they are targeting. For instance, the GitHub backend requires an API token and the names of the repository and owner, while the StackExchange backend needs an API token plus the tag to filter questions. The complexities for querying the data source are encapsulated in Client. Clients share some common features, such as handling connection problems, but most are specific to the data source. For example, long lists of results fetched from GitHub and StackExchange APIs are paginated, thus the corresponding clients have to take care of this. CommandLine is provided to make parameters for each data source available via the command line.

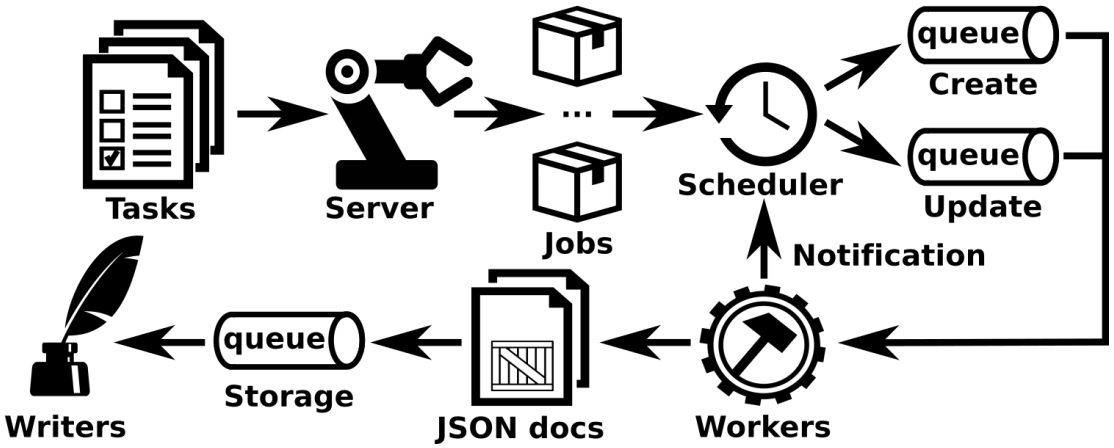


Figure 4: Overview of King Arthur. The server receives Perceval tasks, converts them to jobs and sends them to a scheduler, which holds two queues, the former for first-time jobs and the latter for jobs that have already been executed once. Workers execute the jobs and store the Perceval JSON documents to a queue which can be consumed by writers.

King Arthur, shown in Figure 4, provides an HTTP API (via its Server class), which allows for the management (submit, delete, list) of Perceval jobs, defined as JSON documents controlling the details of the job (e.g., Perceval parameters or maximum number of retries upon failures). Managed jobs are sent to the Scheduler class, which maintains queues for first-time and incremental retrievals, rescheduling in case of failures. These queues submit jobs to Workers (which can run on different machines), which are the key scalability element of King Arthur. When jobs are done, workers notify the scheduler, and in case of success, they send the JSON documents, resulting from Perceval data retrieval, to a storage queue, where they are consumed by writers, making it possible to live-stream data or serialize it to database management systems.

Perceval can be used on its own, usually from a Python script, if the data retrieval is not complex or too large. King Arthur can be managed from any script issuing HTTP commands.

2.1.3 Data storage

GrimoireLab pipelines usually involve storing the retrieved data, with two main goals: allowing for repeating analysis without needing new data retrieval, and having pre-processed data available, suitable for visualization and some analytics. For the first goal, a raw database is maintained, with a copy of all JSON documents produced by Perceval. For the second, an enriched database, which is a flattened summary of the raw documents, is produced and stored.

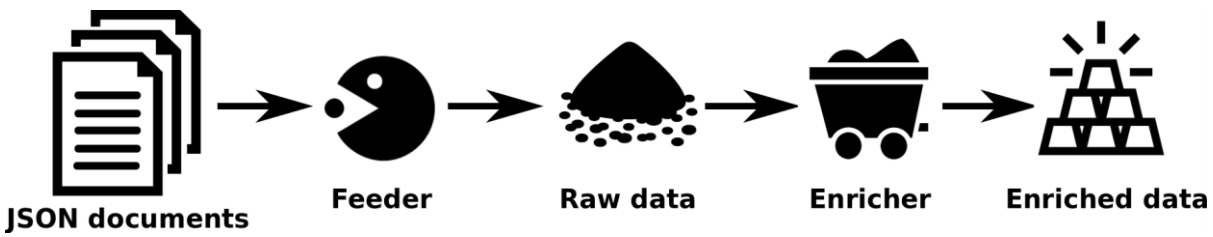


Figure 5: Overview of GrimoireELK. The Perceval JSON documents are stored as they are into a No-SQL database to allow multiple analysis without retrieving data over and over. The stored data is then

enriched with additional information and stored separately, its format can be consumed by the analytics component.

GrimoireELK (or GELK) is the main actor of this task, interacting with the database. The design underlying GELK is shown in Figure 5. A feeder collects the JSON documents produced by the data retrieval, storing them directly as the raw database. Dumps of this raw data can be easily created to make any analysis reproducible, or just to conveniently perform the analytics with other technologies beyond the ones provided by GrimoireLab.

The raw data is then enriched, summarizing the information usually needed for analysis and visualization, in some cases computing new fields. For example, pair programming information is added to Git data, when it can be extracted from commit messages, or time to solve (i.e., close or merge) an issue or a pull request is added to the GitHub data. The enriched data is stored as flat JSON documents, embedding references to the raw documents for traceability.

2.1.4 Identities management

Identity management is another task, which when performed, produces data which is included in the enriched database. It enables analysis where contributor identities and their related information such as organizations and affiliations are needed.

Depending on the kind of repository from which the data is retrieved, identities can come in different formats: commit signatures (e.g., full names and email addresses) in Git repositories, email addresses, GitHub or Slack usernames, etc. Furthermore, a given person may use several identities even in the same repository, and in different kinds of repositories. In some cases, an identity can be shared by several contributors (e.g., support email addresses in forums).

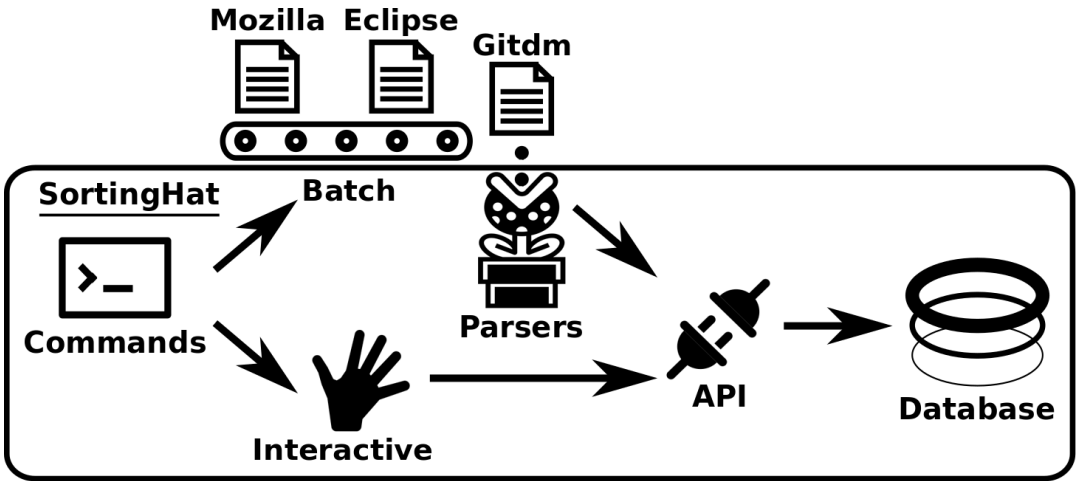


Figure 6: Overview of SortingHat. Shell commands allows to manipulate identities with manual and heuristic-based operations, which are translated to API calls and executed against the underlying relational database. SortingHat supports both interactive and batch modes to perform single operations or load bulk of identities via parsers.

SortingHat and Hatstall are the tools used in GrimoireLab for managing identities. In the usual pipeline, GELK feeds SortingHat with identities found in raw data, dealing with merging and affiliation according to its configuration, and sending them back to be added to the enriched data. For doing its job, SortingHat maintains a relational database with identities and related information, including the origin of each identity, for traceability.

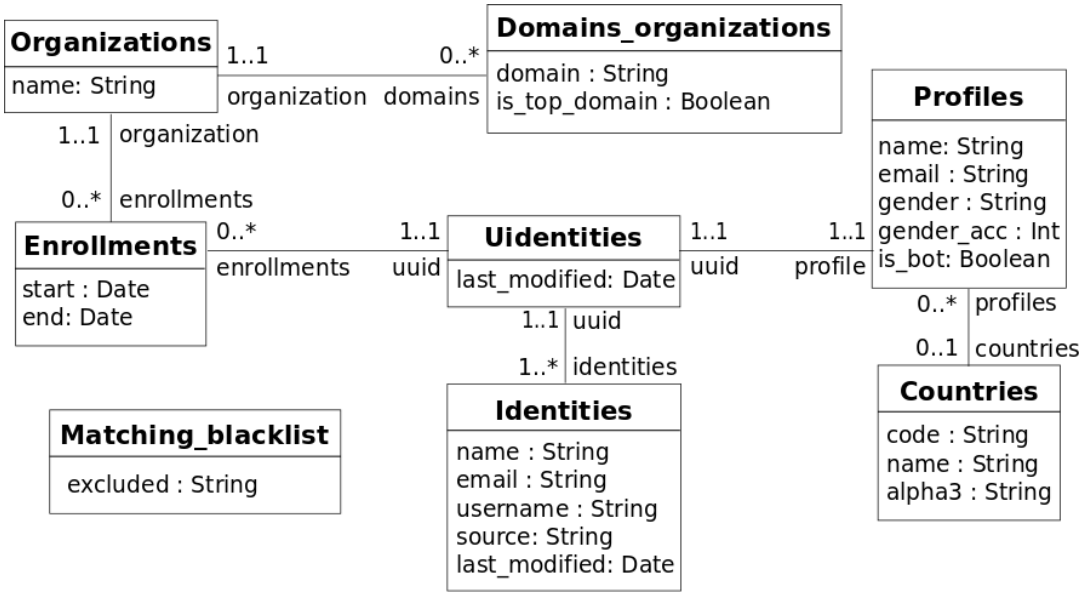


Figure 7: Overview of SortingHat conceptual schema, which models identities and their related information.

Hatstall allows to configure SortingHat through a web interface, for example manually merging identities for the same person. SortingHat may also automatically read identities-related data in some formats: Gitdm, MailMap, Stackalytics, and the formats used by Eclipse and Mozilla projects. The overall design of SortingHat is summarized in Figure 6. The conceptual schema of the SortingHat database is shown in Figure 7.

2.1.5 Analytics

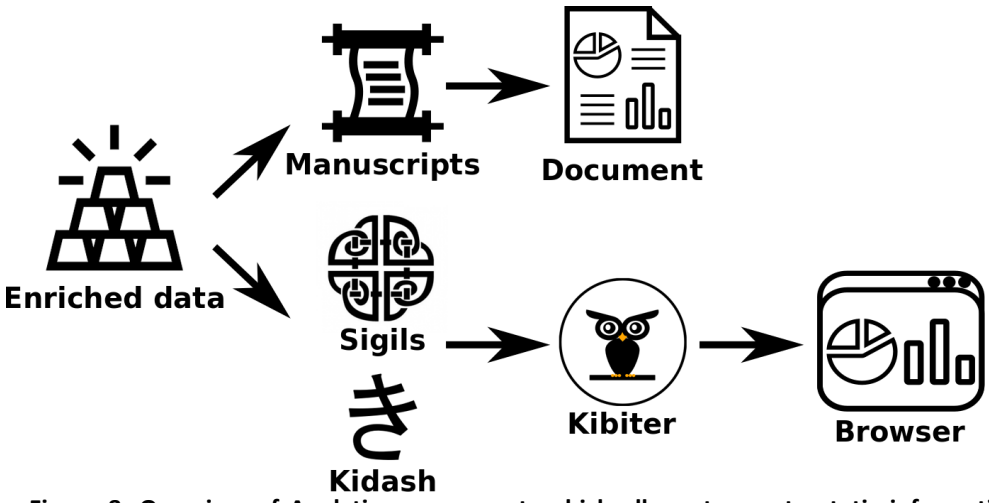


Figure 8: Overview of Analytics component, which allows to create static informative reports and interactive Web-based visualizations.

The analytics component is in charge of presenting the enriched data via static and interactive visualizations. The way of generating the visualizations is highlighted in Figure 8 and described below.

- Manuscripts is a tool that queries the enriched data and provides analytics results, as summary tables in standard reports, produced from templates. The tables are available in CSV format, thus they can be imported into spreadsheets or other programs. Documents can

be produced as PDF files, ready to be delivered to decision-makers, able in this way to easily identify relevant aspects of their projects.

- The creation and presentation of interactive visualizations involve three tools: Sigils, Kidash, and Kibiter. Sigils is a set of predefined widgets (e.g., visualizations and charts) available as JSON documents. Kidash loads Sigils widgets to Kibiter (a soft-fork of Kibana[4]), which performs the binding between the widgets and the enriched data, thus providing Web-based dashboards for actionable inspection, drill down, and filtering the software development data retrieved.

2.1.6 Orchestration

Mordred can be used to orchestrate all the other components to retrieve data from a set of repositories, produce raw and enriched data, load predefined widgets and generate documents and web-based dashboards. It uses some configuration files, designed to keep sensitive data separated from the one that can be publicly shared. These files include the details for accessing all repositories, including addresses and credentials, and all the servers (eg, the SortingHat database). Repositories can be arranged hierarchically in several levels (projects, sub-projects). Both files are described below.

- *Setup.cfg*. The setup file holds the configuration to arrange all processes underlying GrimoireLab. It is composed of sections which allow to define the general settings such as which components activate (e.g., a research could be interested in only retrieving data without using the other components) and where to store the logs, as well as the location and credentials for GELK, SortingHat and Kibiter which can be protected to prevent undesired accesses. Furthermore, it also includes other sections to set up the parameters used by the data retrieval component to access the software development tools (e.g., GitHub tokens, gerrit username) and fetch their data.
- *Projects.json*. The project file enables the users to list the instances of the software development tools to analyse, such as local and remote Git repositories, the URLs of GitHub and GitLab issue trackers and the name of Slack channels. Furthermore, it also allows the user to organize these instances into nested groups, which structure is reflected in the visualization artifacts (i.e., documents and dashboards). Groups can be useful to represent projects within a single company, sub-projects within a large project such as Linux and Eclipse, or the organizations within a collaborative project.

2.2 INTEGRATION WITH CROSSMINER

This section summarizes the components developed to allow the integration between GrimoireLab and CROSSMINER. Figure 9 highlights such an integration.The components can be grouped in 4 categories: *collection*, *enrichment*, *consumption* and *automation*. The collection relies on Perceval, the enrichment on GELK, while the consumption leverages on Kidash and Kibiter. All components are described below.

[4] https://www.elastic.co/guide/en/kibana/current/getting-started.html

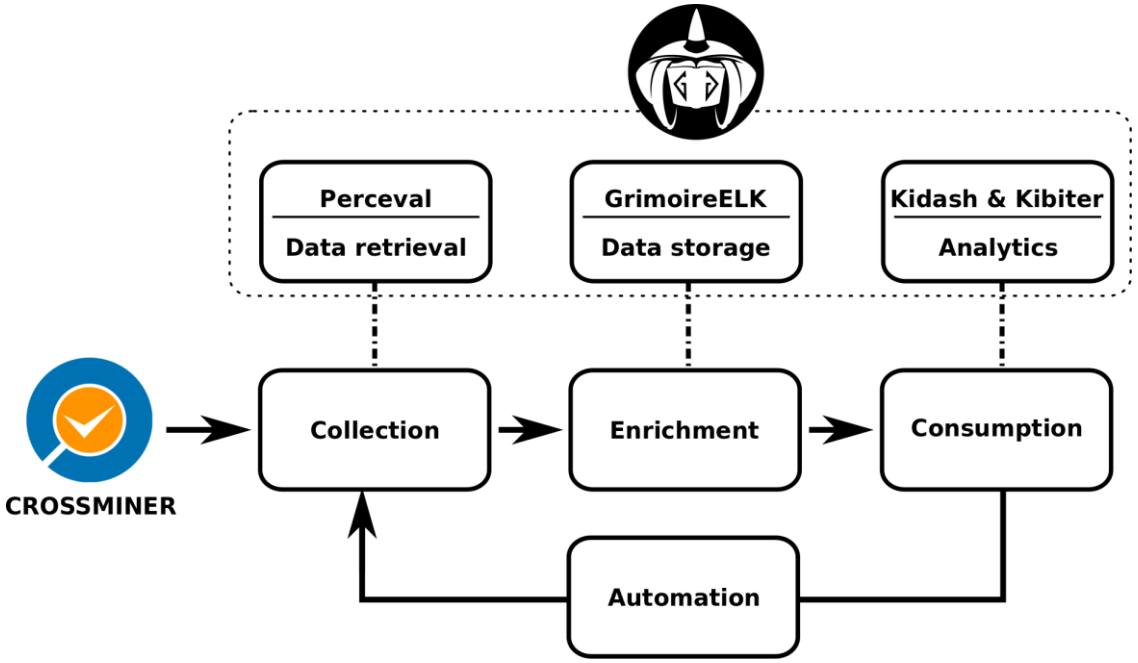


Figure 9: Overview of the integration between GrimoireLab and CROSSMINER. The collection, enrichment and consumption components rely on GrimoireLab technologies such as Perceval, GELK, Kidash and Kibiter.

2.2.1 Collection

CROSSMINER[5] is considered as an external data source for GrimoireLab, thus a Perceval backend able to fetch projects and recommendations data from the CROSSMINER API has been built. The project data consists of the list of projects available in CROSSMINER plus their corresponding metrics, while the recommendations are a list of projects similar to the one being analysed. The extraction of projects, metrics and recommendations is achieved by using different categories, which allow to drive the execution of Perceval. The categories are listed below:

- **Project.** It collects the list of projects available on CROSSMINER using the endpoint */projects*.
- **Metric.** It collects the metrics available for each CROSSMINER project via the endpoint */metrics/p/project-name*, and then it retrieves the corresponding metric values via the endpoint */projects/p/project-name/m/metric-name*.
- **Factoid.** It collects the factoids for the CROSSMINER projects, for each project the factoids are collected using the endpoint */projects/p/project-name/f* and later the factoid values are fetched via the endpoint */projects/p/project-name/f/factoid-name*.
- **User.** It collects the user activity for each CROSSMINER project using the following endpoint */projects/p/project-name/m/churnPerCommitterTimeLine*.
- **Development dependency.** It collects the OSGI and Maven dependencies for each project using the following endpoints:
  - *raw/projects/p/project-name/m/trans.rascal.dependency.maven.allMavenDependencies*

[5] Details on how to access CROSSMINER data are at: https://crossminer.github.io/scava-docs/developers-guide/

  - *raw/projects/p/project-name/m/trans.rascal.dependency.maven.allOptionalMavenDependencies*
  - *raw/projects/p/project-name/m/trans.rascal.dependency.osgi.allOSGiPackageDependencies*
  - *raw/projects/p/project-name/m/trans.rascal.dependency.osgi.allOSGiBundleDependencies*

- **Configuration dependency.** It collects the Puppet and Docker dependencies for each CROSSMINER project using the following endpoints:
  - *raw/projects/p/project-name/m/org.eclipse.scava.metricprovider.trans.configuration.docker.dependencies.DockerDependenciesTransMetricProvider*
  - *raw/projects/p/project-name/m/org.eclipse.scava.metricprovider.trans.configuration.docker.dependencies.PuppetDependenciesTransMetricProvider*

- **Version dependency.** It collects information about version dependencies (Puppet, Docker, OSGi, Maven) for each CROSSMINER project. Version data includes details about the current version used and the latest available for each project dependency:
  - *raw/projects/p/project-name/m/org.eclipse.scava.metricprovider.trans.newversion.docker.NewVersionDockerTransMetricProvider*
  - *raw/projects/p/project-name/m/org.eclipse.scava.metricprovider.trans.newversion.docker.NewVersionPuppetTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.newversion.osgi.NewVersionOsgiTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.newversion.puppet.NewVersionPuppetTransMetricProvider*

- **Configuration smell.** It collects information about design, implementation, antipattern smells obtained from Puppet and Docker for each CROSSMINER project. The target endpoints are:
  - *org.eclipse.scava.metricprovider.trans.configuration.puppet.designsmells.PuppetDesignTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.configuration.puppet.implementationsmells.PuppetImplementationTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.configuration.docker.smells.DockerTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.configuration.puppet.designantipatterns.PuppetDesignAntipatternTransMetricProvider*
  - *org.eclipse.scava.metricprovider.trans.configuration.puppet.implementationantipatterns.PuppetImplementationAntipatternTransMetricProvider*

- **Project relation.** For each CROSSMINER project, it collects the relations with other projects using the following endpoint: *raw/projects/p/project-name/org.eclipse.scava.metricprovider.trans.configuration.projects.relations.ProjectsRelationsTransMetricProvider*

- **Recommendation.** It collects recommendations data from the KB API. First given the project in CROSSMINER, the corresponding project ID in the KB is retrieved with the endpoint */recommendations/artifacts/artifact/mpp/project-name*, then for each similar method (e.g., Compound, Dependency, CrossRec), the first 10 recommended projects are retrieved with the KB endpoint */recommendation/similar/p/project-id/m/sim-method/n/10*.

The output of Perceval is a list of JSON documents, which is the format used by GrimoireLab to store data in its underlying ElasticSearch database. The Perceval backend is written in Python and available on GitHub[6].

2.2.2 Enrichment

The JSON documents obtained by the Perceval backend are processed by an enricher, available on GitHub[7]. First, the data contained in each Perceval document is extracted, then depending on its category, the data is processed in different ways. For the sake of brevity, we focus on the data coming from the category **Metric** which is translated to a common model and finally stored in ElasticSearch.

The metric model, shown below, is composed of 12 attributes and materialized as a JSON document. Most of the attributes are metric-related and come directly from the API, they allow to identify the class, name, description, type (i.e., *metric_es_compute*) and the id of the metric, plus when it was created/updated (i.e., *datetime*) and its value (i.e., *metric_es_value*, *metric_es_value_weighted*). The remaining attributes are used to identify the project and the macro project which the metric belongs to (i.e., *project*, *meta*) and the metric within the ElasticSearch database (i.e., *uuid*). The *uuid* is derived using the *metric_id*, the project and the datetime attributes, thus allowing to store to the database the same type of metric for different projects and time snapshots. It is worth noting that the *datetime* value is converted to a common format to avoid differences with respect to time zones.

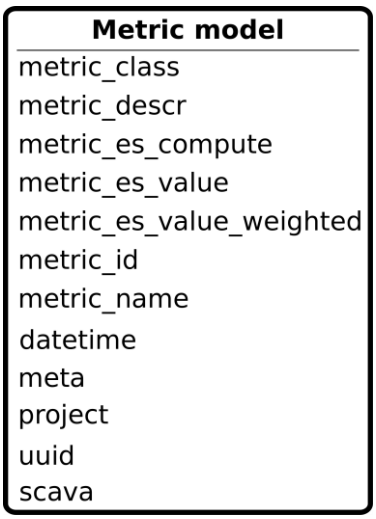


Figure 10: Overview of the metric model. It is composed of 12 attributes, most of them are metric-related while the rest identify the project, the time and the unique identifier of the item

[6] github.com/crossminer/scava/tree/dev/web-dashboards/perceval-scava
[7] github.com/crossminer/scava/blob/dev/web-dashboards/scava-metrics/scava2es.py

2.2.3 Consumption

The enriched data stored in ElasticSearch, can be consumed using Kibana dashboards (described in Section 3.3). Kibana is a tool, which provides visualization capabilities (e.g., scatter plots and pie charts) on top of the content indexed on an Elasticsearch database.

2.2.4 Automation

A Shell script, executed every 5 minutes, is in charge of triggering the collection and enrichment components plus uploading the Web dashboards to Kibana. It allows to automatically reflect the CROSSMINER data to the Web dashboards with a reasonable delay. The script is available on GitHub[8].

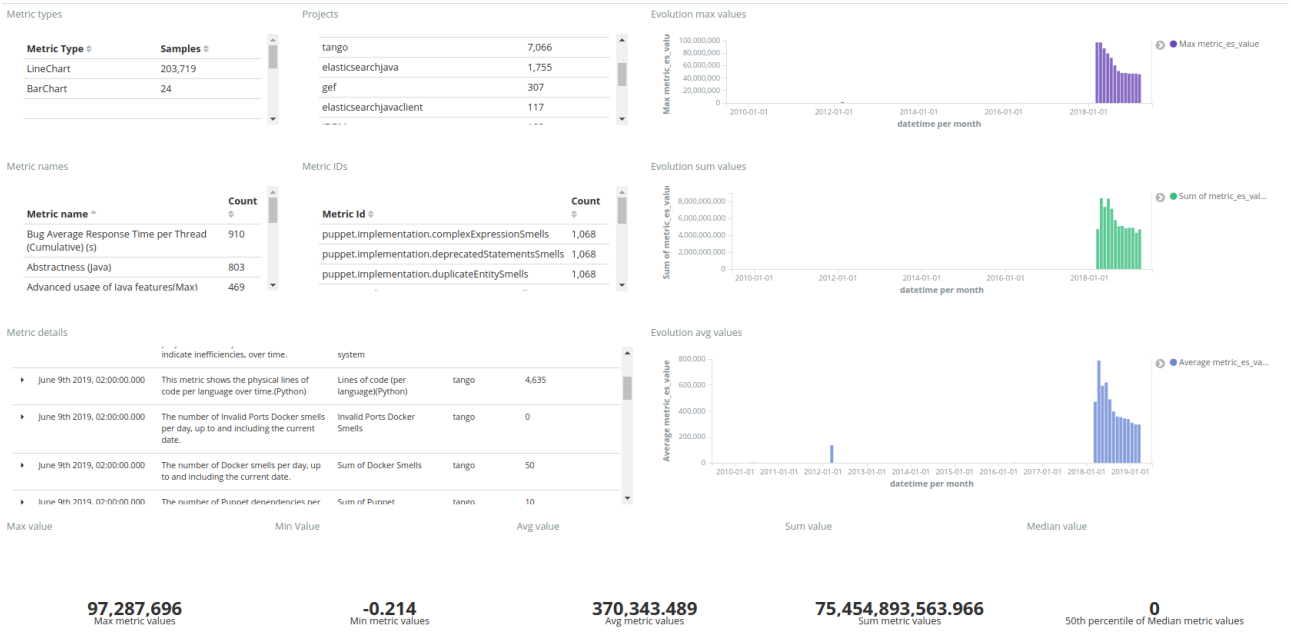
3. WEB-BASED DASHBOARDS

This section describes the various dashboards developed in the context of CROSSMINER.

3.1 PROJECT DASHBOARD

The Project Dashboard (dev/debug) allows to understand the CROSSMINER metrics in terms of global and time values. Thus, it shows all the metrics available in CROSSMINER, fetched from the API. It displays the global value and the evolution in time of the metrics. It includes also filters to ease the analysis of a specific group of metrics. The dashboard is composed of different tables and visualizations, the tables show some statistics on the metrics (e.g., number of metrics per type, name, and project), while the visualizations plot the average, sum and max values for every snapshot of a given metric.

A screenshot of the dashboard is shown below, as can be seen, the tables on the left contain global values, while the visualizations on the right allow to understand the evolution of the metrics. Finally, the last row of the dashboard shows a summary of statistics (average, median, max, min, and sum) about the metrics which help the users to understand how a given metric varies.



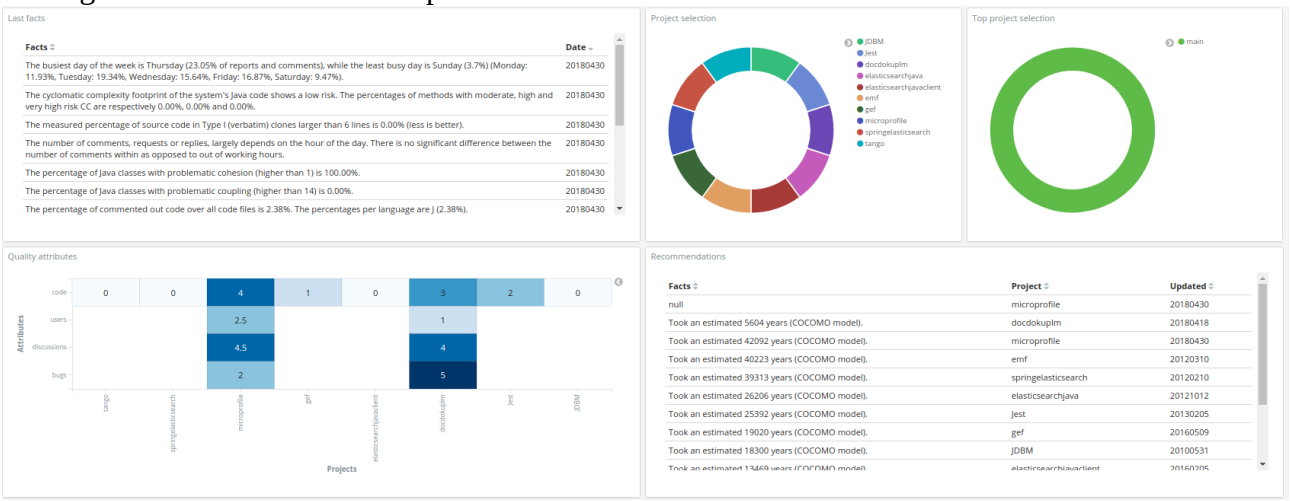
[8] github.com/crossminer/scava-deployment/blob/dashboard-plus-admin/dashboard/starter.sh

Figure 11: Project dashboard

3.2 OVERVIEW DASHBOARD

The Overview dashboard summarizes relevant metrics for the projects analyzed in CROSSMINER, including visualizations from the qm assessment, the facts, emotions, users, etc. In order to see the data detailed, it is necessary to go across different dashboards. Starting from top to bottom and from left to right, includes the following visualizations. First of all, there is a table that shows the list of last facts. Then there are two "selection" pies, a pie chart that shows the projects. It can be used to filter by project and a pie chart that shows the top projects. It can be used to filter by top project. To follow, there is a table with the last recommendations (they come from the trans.rascal.OO metrics) and a heat map that allows to compare the projects against the attributes of the quality model defined in Prosoul[9]. Then, there is a table that shows the similar projects in terms of recommendation, it describes if it's active, the type and the number of recommendations. The next row shows a bar chart that shows the evolution of bugs (fixed vs closed/non resolved) and another bar chart that shows the evolution of commits. The next two rows show two bar charts as well, starting with a bar chart that shows the evolution of percentage emotions (anger, joy, sadness, surprise, love) in bugs and a bar chart that shows the evolution of active and inactive users on bug tracker. Finally, the last row includes a pie chart and a bar chart showing the top 10 topics on comments and their evolution.

The figure below shows an example of the dashboard:



[9] https://github.com/Bitergia/prosoul

Figure 12: Overview dashboard

3.3 FACTOIDS DASHBOARD

The factoids dashboard is useful in order to understand the factoids that are in the projects, starting with a small brief of the last factoids and its distribution in order to continue with the analysis of Bugs, Commits and Code quality. Starting from top to bottom and from left to right it includes the following visualizations. The first table shows the number of projects that have from one to four factoids and the next two "selection" pies in order to filter the dashboard by project or top project if it is required. The following radar shows an overview of the project and how the number of factoids are distributed on them, the table shows the last factoids added. The following three heat maps

show different specific factoids with the projects, grouped them by bugs, commits and code quality. The last large heat map shows all the projects and all the factoids with the corresponding values.

The figure below shows an example of the dashboard:

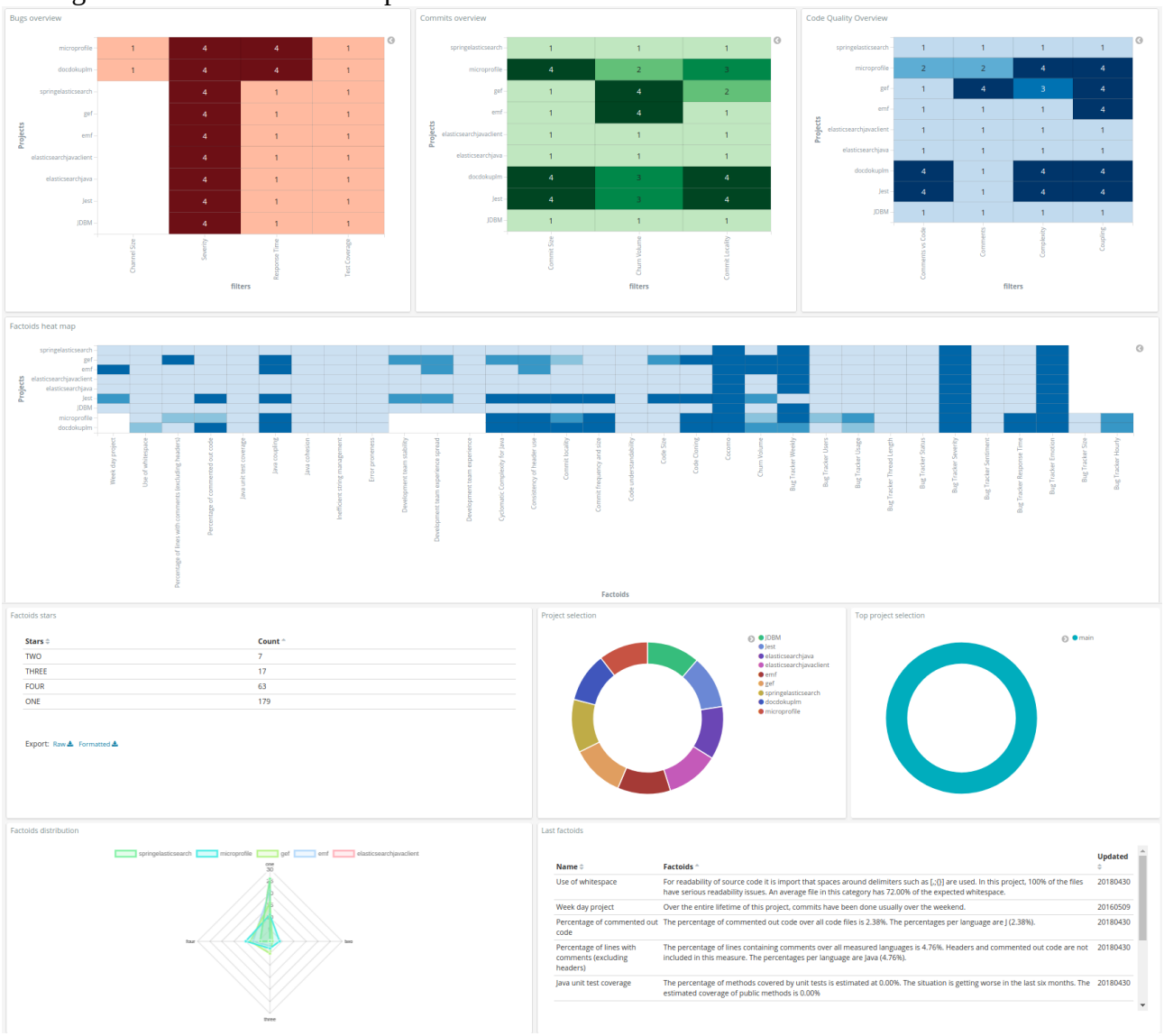


Figure 13: Factoids dashboard

3.4 DEVELOPMENT DEPENDENCIES DASHBOARD

The Dependency dashboard is useful to show the relations between projects based on the components they share. It contains the following visualizations/tables (from left to right, top to bottom). It starts with a visualization that shows the number of total dependencies and two pie charts, the first one is a pie chart that shows the dependencies grouped by project and the second one is a pie chart that shows them grouped by top project. Then, there are two tables, one that shows the dependencies grouped by type (e.g., OSGi and maven) and project, and another table that shows the dependencies grouped by type and top project. Below the tables, there is a graph that relates projects to dependency names and a bar chart that shows the evolution of project dependencies. To finish the dashboard has a table that provides dependencies details (name, versions, project, and datetime) and a table that shows the old and new version of each dependency.

The figure below shows an example of the dashboard:

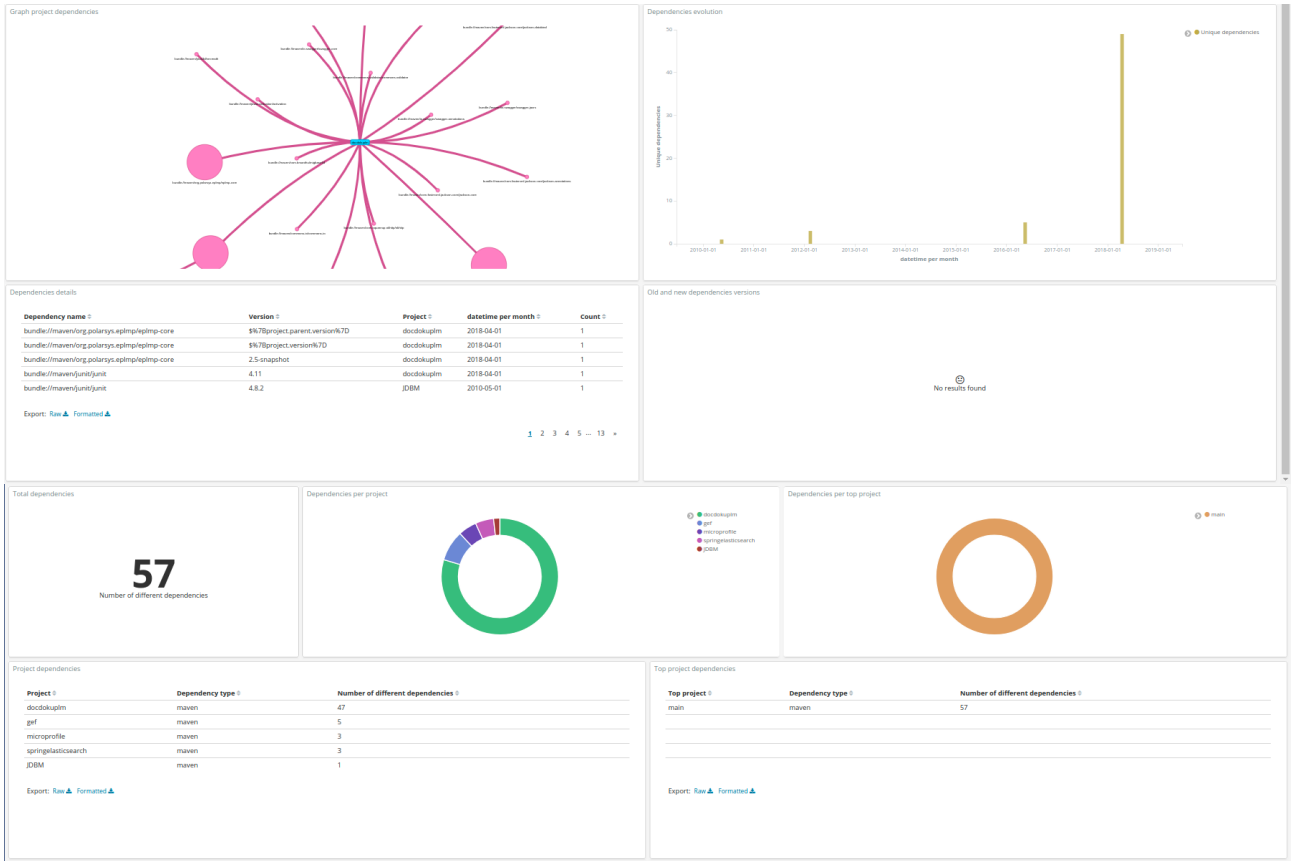


Figure 14: Dependencies dashboard

The dashboard provides a useful network visualization to understand how the projects are connected to each other based on the components they use. In the figure below, the projects and components are shown as boxes and circles, where the size of the latter is proportional to the number of times that component has been used by a given project. The edges highlights the connection between projects and components. By relying on this visualization, shared components can be easily identified, for instance this is the case for the maven.plugin.api, shared between the projects **docdokuplm** and **microprofile**.

Figure 15: Project dependencies network

3.5 DEVOPS DEPENDENCIES DASHBOARD

The Devops Dependencies dashboard is useful to understand the relations between projects in terms of their configuration files. Starting from top to bottom and from left to right it includes the following visualizations. First of all, it has a pie chart that shows the dependencies grouped by project, it has also a pie chart that shows the dependencies grouped by top project, these two pies can be used to filter in a specific project. then there are two tables, the first one shows the dependencies grouped by type (e.g., puppet and docker) and project, and the second one shows the dependencies grouped by type and top project. Below the tables, there is a graph that relates projects to dependency names and a bar chart that shows the evolution of project dependencies. Then, there are two tables, the first one provides details (name and versions) of the project dependencies and the second one shows the old and new version of each dependency. To finish, there is a graph that shows the relation between projects that are using configuration files, the relation can be derived from Puppet or Docker, the table shows the type of relation in addition.

The figure below shows an example of the dashboard:

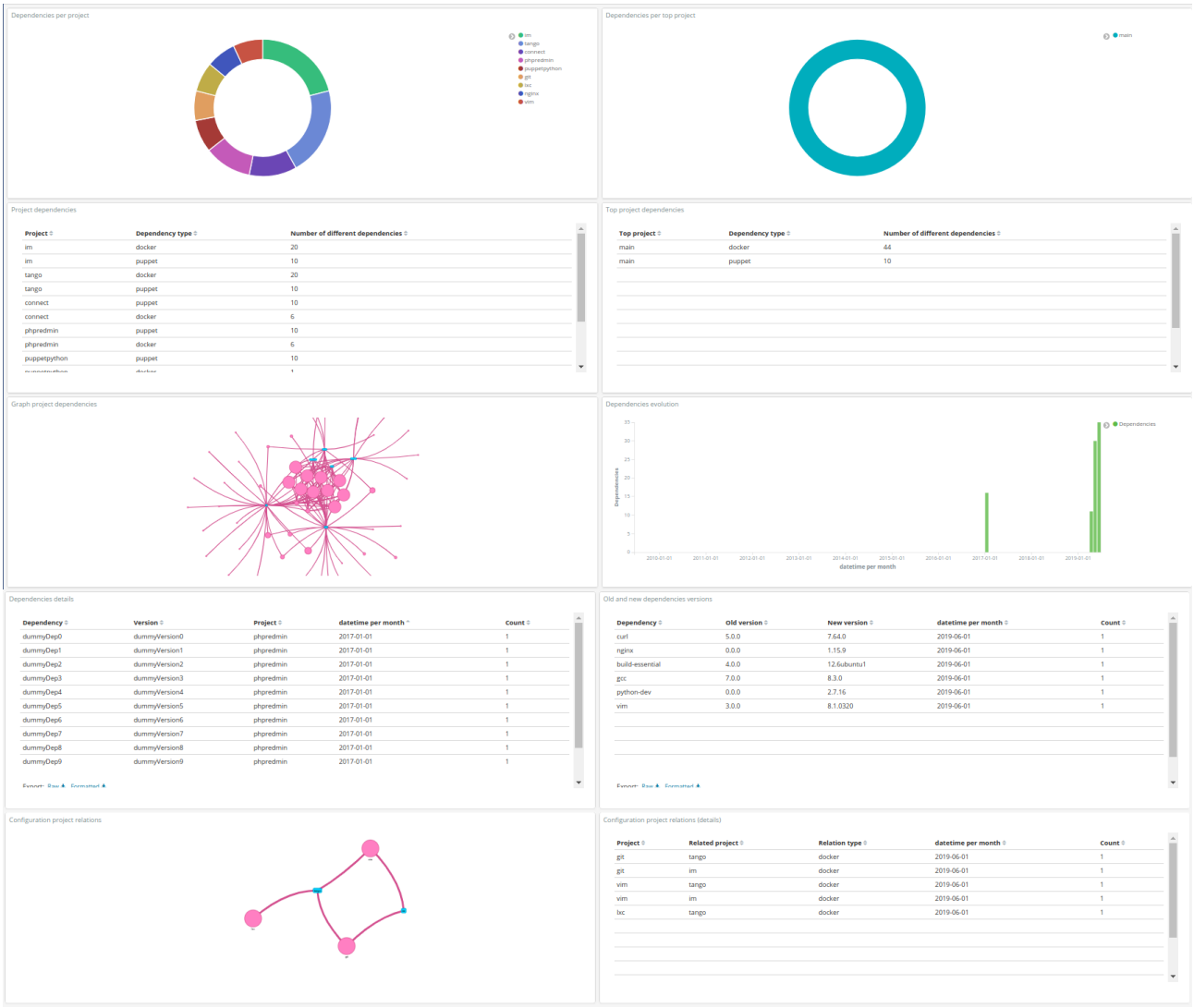


Figure 16: DevOps dependencies dashboard

3.6 DEVOPS SMELLS DASHBOARD

The DevOps Smells Dashboard showcases the information about smells detected in Docker and Puppet based projects. The insights that can be extracted from this dashboard can help a DevOps engineer to make a quality assessment about the status of the configuration code of the analysed project. This dashboard includes two visualizations to focus on a specific project and/or top project. Two bar charts show the evolution of the smells extracted from puppet and docker. Two tables allow to focus on the smells details. The former shows information about design and configuration smells, while the latter gives insights about antipatterns smells. Both of them have the information of the file and the line.

The figure below shows an example of the dashboard:

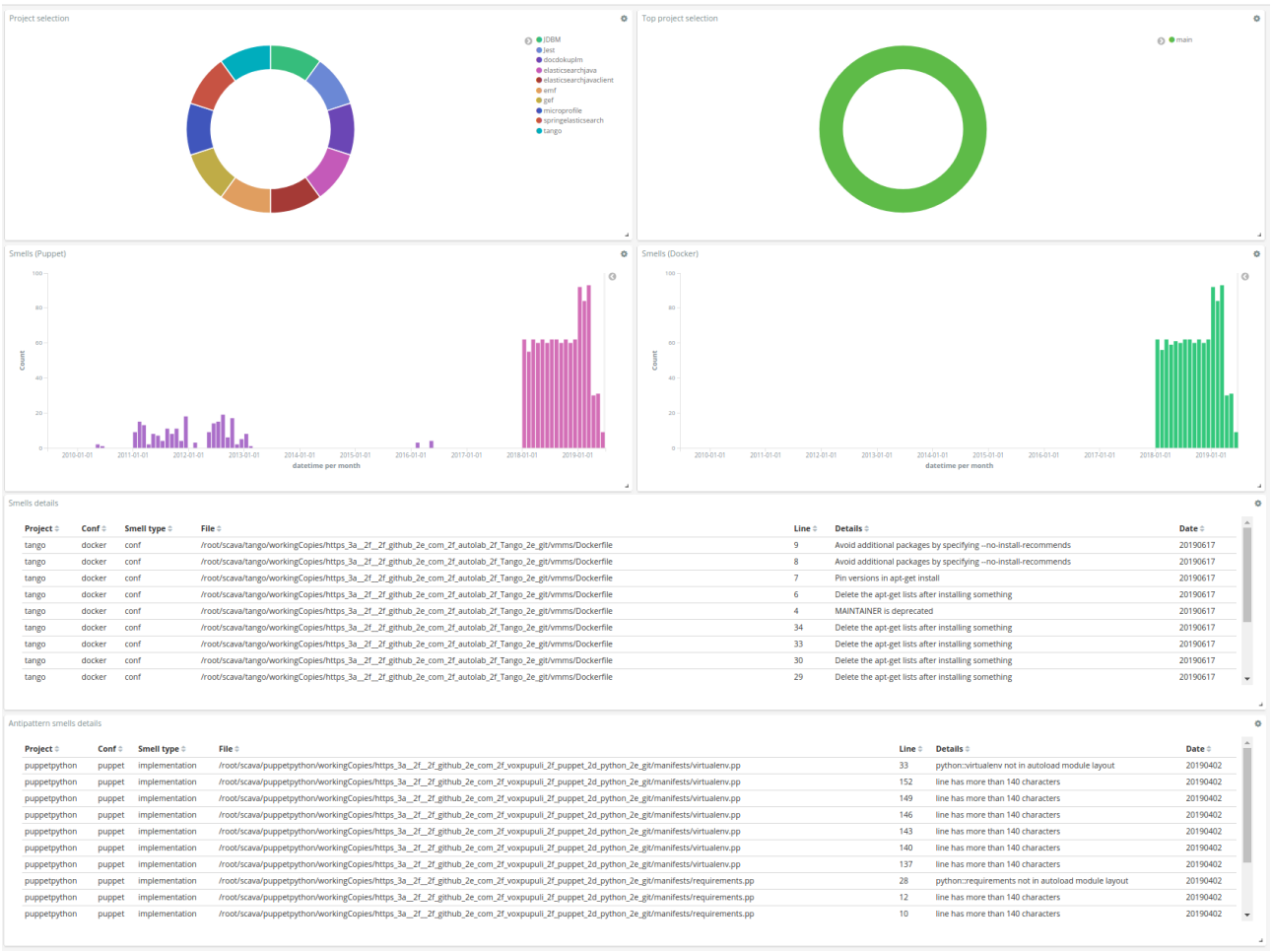


Figure 17: Devops smells dashboard

3.7 QUALITY MODEL DASHBOARD

The Quality model dashboard shows the quality model assessment based on the models defined with Prosoul. It starts with a table that ranks the projects based on the sum of their scores, which are the normalized values (0-5) of metrics based on the threshold defined in the quality model and a pie chart that shows the projects. It can be used to filter by project. Then there is a heat map that allows to compare projects based on the Quality model metric scores and a bar chart that shows the qm metric scores (useful to focus on one project). Following, it has a heat map that allows to compare projects based on the qm attributes (median of the corresponding metric scores) and a bar chart that shows the qm attributes (useful to focus on one project). It continues with another heat map that allows to compare projects based on the qm goals (median of the corresponding metric scores) and a bar chart that shows the qm goals (useful to focus on one project). To finish, there are three bar charts that show the evolution in time of the quality models by quarters.

The figure below shows an example of the dashboard:

Figure 18: Quality model dashboard

3.8 USER DASHBOARD

The User dashboard is useful to show user activity. It contains the following visualizations/tables (from left to right, top to bottom). First, it has two pie charts, a pie chart that shows the projects (it can be used to filter by project) and a pie chart that shows the top projects (it can be used to filter by top project). Then, there is a table that shows the top users based on the sum of their churns and a bar chart that shows the churn evolution (it can be filtered by user). To finish, there is a bar chart that shows the evolution of active and inactive users on bug tracker.

The figure below shows an example of the dashboard:

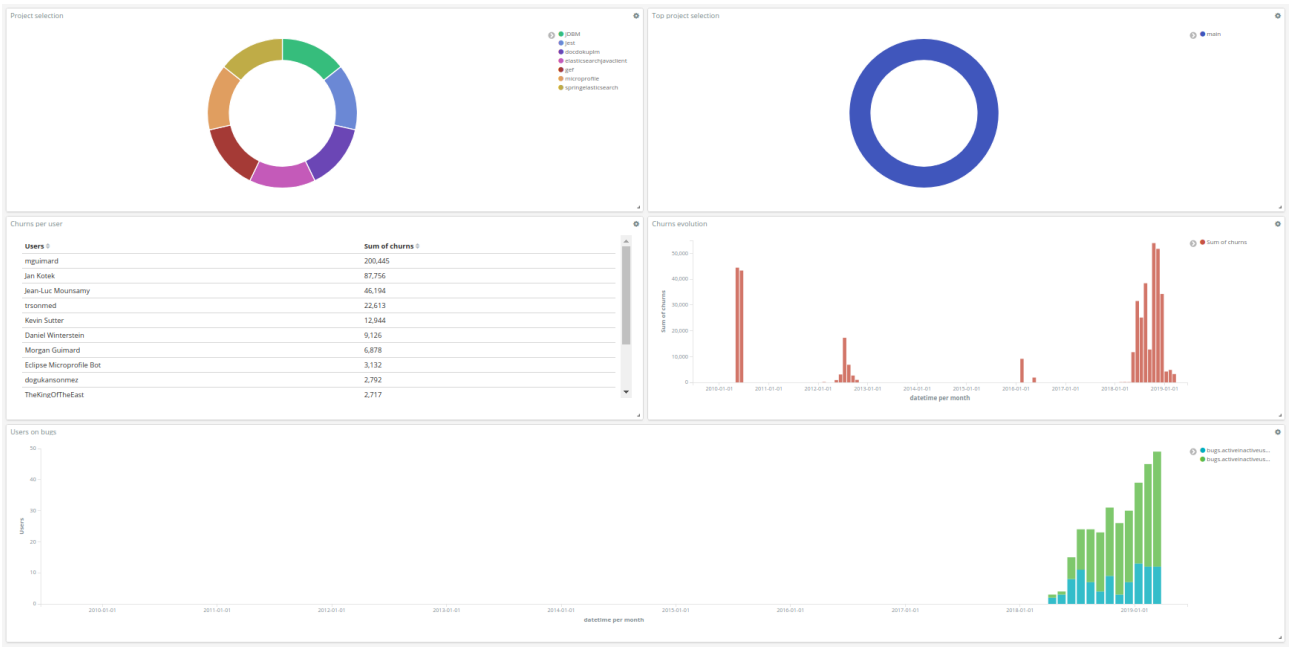


Figure 19: User dashboard

3.9 SENTIMENT AND EMOTION DASHBOARD

This dashboard is useful to show the level of emotion and sentiment that the projects have, it uses the respective metrics in order to show the following visualizations. The dashboard includes 2 visualizations to select a project and top project, a visualization that summarizes the number of emotions, 2 visualizations that show the emotion trend per project based on weighted values and the corresponding details, 2 visualizations that show the sentiment trend per project and details about the sentiment at the beginning and end of the threads.

The figure below shows an example of the dashboard:

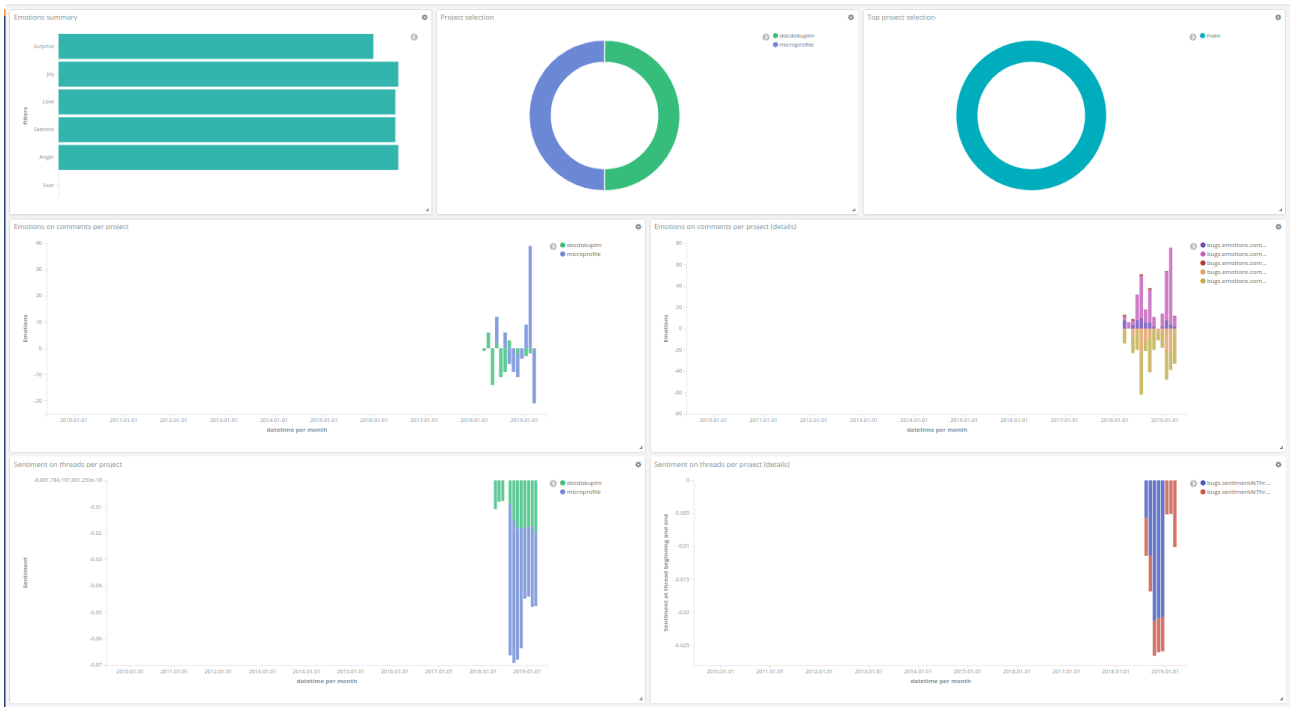


Figure 20: Sentiment and emotion dashboard

4. REQUIREMENTS

In this section a cross check is done for the requirements defined in the use cases and technical requirements in D1.1.

All the requirements coming from the use case partners and those defined in WP7 have been addressed.

4.1 USE CASES

No	Requirement	Priority
U199	Administrators can provide default templates for dashboard	SHALL
U200	Users can create and save specific profiles (sets of metrics and/or attributes) in the dashboard	SHALL
U201	Users can create and save their own profiles (sets of metrics and/or attributes) in the dashboard	SHALL
U202	Dashboard offers history on metrics	SHALL
U203	Dashboard offers the possibility to show several historical metrics on the same chart	SHALL
U204	Dashboard offers history on quality attributes	SHALL
U205	Dashboard offers time range selection on graphics	SHALL
U206	Dashboard offers ability to specify aggregation (month, week) of time range data	SHOULD
U207	Dashboard should be available via a CROSSMINER web application	SHALL
U208	CROSSMINER web application provides human accessibility features	SHOULD
U209	Plots can be exported and reused in external web sites	SHALL
U210	Dashboard offers a way to compare different projects according to user-defined criteria	SHOULD
U211	Dashboard offers a visual representation of the quality model	SHOULD
U212	Able to represent visually the project quality as a scorecard	SHALL
U213	Web dashboards are based on ElasticSearch and Kibana	SHALL
U214	Dashboard displays if project uses a third-party API and if that third-party API has new versions	SHOULD
U215	Dashboard able to display the number of projects using the different versions of a third-party API	SHOULD
U216	Dashboard indicates the amount of change required for migrating the project using an old version of a third-party API to use the latest version	SHOULD
U217	Dashboard allows the user to select the type of chart for representing selected metrics	SHALL

4.2 TECHNICAL

T109	Shall provide a dashboard with all metrics defined that could be visualised in a web browser.
T110	The dashboard shall support drill down navigation.
T111	The dashboard shall include a global search capability for dashboard contents.
T112	The dashboard shall support filtering capabilities using dates.
T113	The dashboard shall support filtering using predefined fields included in the data.
T114	The dashboard editor shall support the ability to create new dashboards and visualizations.
T115	The dashboard implementation shall offer a visualization library for which to build customised dashboards.
T116	The dashboard shall provide an interface to make it extensible with new visualizations.
T117	The dashboard shall offer an interface for showing the metrics in the dashboard.
T118	Administrators of the CROSSMINER system shall be able to provide default templates for dashboard (e.g. sustainability, activity, support).
T119	The dashboard web interface may support accessibility features for presenting results.
T120	The dashboard shall offer a way to compare different projects according to user-defined criteria.
T121	The dashboard shall be able to show if a project uses a third-party API and that third -party API has new versions.
T122	The dashboard shall be able to show the number of projects that use the different versions of a third-party API (as per version).
T123	The dashboard shall indicate the amount of changes required to migrate a project using an old version of a third-party API to using the latest API version.
T124	The dashboard shall allow the user to define personalised dashboards based on a selection of metrics.
T125	The CROSSMINER Platform shall expose a REST API that can be used to collect the data needed by the dashboard in order to publish it in Elasticsearch.