Grip on software: understanding development progress of SCRUM sprints and backlogs

Helwerda, L.S.

Chapter 2

Data pipeline

GROS gatherer: Agent-based acquisition of high-frequency data updates within existing dynamic software development ecosystems

Abstract of Chapter 2

Introduction: Research into software development processes relies on collecting frequent updates to data stored in systems that help with planning, development and quality control. For this purpose, we design and implement a novel data acquisition pipeline that fits in the existing software development ecosystem.

Research questions: How can we reliably collect data regarding SCRUM software development practices and consolidate the resulting artifacts inside a central database that constantly grows and allows adaptable queries?

Abstraction: We assess which existing concepts regarding pipeline construction, data collection and distributed data processing are relevant for our objective. We consider existing designs with asymmetric communication between agents and a controller, including where the "knowledge" is kept, which connections to other systems are established, how the status is tracked and what kind of processing steps are possible.

Implementation: We implement several components that collect data from issue trackers, version control systems, code review, quality control and other platforms. The distributed setup allows early linking of related entities, encryption of sensitive data and portability within networked setups. The components are configurable through an interface and the status is tracked in a dashboard. Further steps toward database import, data analysis and information visualization have their own components with novel highlights.

Discussion: Experiments show that the pipeline components are able to work within different setups at two organizations and in a central research location. The frequent collection runs do not impact the performance of the ecosystem's platform during normal usage. Steps were taken to allow further generalization and adaptation.

2.1 Introduction

As part of Grip on Software, our research into patterns and outcomes within SCRUM software development processes, we find that there is a need to collect data about activities of software developers related to SCRUM principles. Often, multiple systems are used by development teams in order to handle various aspects of the development process. This includes project management systems that keep backlogs of user stories, bugs and other tasks to work on—usually referred to as an issue tracker—but also version control of code repositories, automated quality control checks of the code and other artifacts, platforms to test a compiled build of the code and other organizational tools for access control.

Limited communication exists between these systems. Often, a change in one system cannot be firmly related to another, such as a quality measurement for a code change based on the resolution of a user story. This makes it difficult to get a complete picture of the situation in the development process, let alone understand past SCRUM sprints.

A data acquisition pipeline helps mitigate these shortcomings within the software development ecosystem by collecting data from multiple sources and generating artifacts explaining changes in a simple format. We have a need for such output in order to compile a centralized, consistent database. From this database, we produce features and relevant data points for further analysis and reporting, respectively through pattern recognition—backed by machine learning and estimation models—and information visualization. The choices made in this first step are important, as they make it possible to reproduce the research within other organizational ecosystems.

We consider the following research question and a selection of two sub-questions regarding the data acquisition pipeline, which we also refer to as the GROS gatherer:

RQ1 How can we reliably collect data regarding SCRUM software development practices and consolidate the resulting artifacts inside a central database that constantly grows and allows adaptable queries?

- **RQ1a** Based on relevant design principles from distributed data systems and agent-based networking, how do we design a data acquisition pipeline applied to multiple organizational ecosystems?
- **RQ1b** Which objectives related to inspection, curation and disclosure do we achieve with a data acquisition pipeline when taking our formulated requirements in mind?

In terms of scope, the SCRUM framework is part of the family of Agile software development methods. Among other values, an important factor of Agile development is the focus on individuals and their interactions within a team, more so than the processes and tools that they use within their work [3]. When teams decide on their own how to arrange their development process, different decisions regarding the use of these systems are made compared to other teams and organizations. This level of autonomy should not be seen as an inconvenience to outsiders. Instead, each application of the development method provides unique insights. For a data acquisition pipeline that supports such a development process on a meta-level, flexibility is an important factor in effectively applying it within multiple ecosystems.

2.1.1 Ecosystem

An important aspect of the implementation of a data acquisition pipeline for research purposes is to properly integrate it into the existing ecosystems that it meant to be used in. In our case, different software development organizations have their own networks, physical computing infrastructure, virtualization approaches, data storage solutions and security measures. A pipeline should be generic enough to fit with multiple of such resources. However, an organization should meet some prerequisites of the pipeline in order to be relevant for our study.

Within the SCRUM software development framework, teams are typically able to fill in how they work on the project in a flexible manner. As an Agile software development method, SCRUM also places the focus on the people within the team rather than the software that they use. Still, the systems in use for development and release are helpful. The ability to track progress centrally in an accessible manner makes the physical "wall of cards" quite outdated. It is reasonable that the maintenance overhead is much lower with a consistent system. The applications used within software development should aim at decreasing such workload.

The research project has a similar goal and it is sensible that a data collection pipeline avoids such burdens. The purpose and functions of individual elements of the pipeline should be open and understandable. At the same time, they should not place a heavy load on the existing ecosystem.

Not every organization uses the same development applications and network solutions. When different systems are used for the same purpose, such as issue tracking or version control, the pipeline should be versatile enough to collect data from one or more of them using similar interfaces and provide interchangeable artifacts. We are then able to use the source data in a comparable manner later on in our analysis. An organization chooses their own approaches in implementing the development ecosystem on different layers, although these layers typically contain elements of the items shown in Figure 2.1.

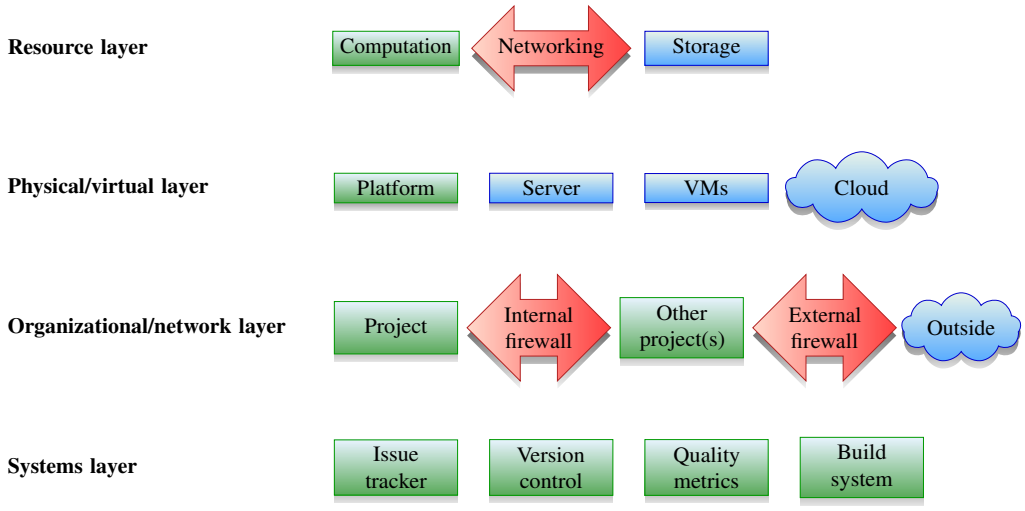

Figure 2.1: General layers of a software development ecosystem. Green blocks represent elements that are the main interests of our research approach, blue shapes are supporting elements and red arrows show interconnection elements.

2.1.2 Structure

This chapter's structure is balanced between theoretical concepts and practical implementation. In Section 2.2, we mention design principles and relevant concepts regarding the construction of a data collection system. We look into similar frameworks and abstractions of distributed systems, including agent-based networks. Some practical matters relevant for pipeline deployment within an organization's platform ecosystem are also brought up.

In the remainder of this chapter, especially Section 2.3, the focus is on the development, discussion and significance of the components related to the data acquisition, introduced in Section 2.3.1. This is the first step from the software development project management tools to intermediate data objects. Other steps within the process are discussed only briefly in this chapter. In Section 2.3.2, we discuss these components, with some technical details of the overall Grip on Software pipeline in Section 2.4. Section 2.5 describes some experiments and results obtained from practical usage of the data acquisition components. We conclude our findings in Section 2.6.

Other portions of the pipeline are further described—in context of the research—in their own chapters. Chapter 3 describes the database construction, Chapter 4 discusses feature selection for prediction models and Chapter 5 demonstrates the use of the data in information visualization.

2.2 Design

We consider the technical implementation of a data acquisition pipeline within the context of abstract design concepts. Often, data pipelines are constructed for a specific purpose within a predetermined environment, for example a business analytics context. There are however methods to make these approaches to data processing more portable and scalable. Therefore, we also look into existing approaches and design choices for certain use cases within organizational data acquisition. In this section, we bridge the gap between theory and established practice.

2.2.1 Distributed data systems

A data acquisition pipeline consists of multiple components that collectively allow to collect, combine, filter, mutate and reuse one or more data sets for one or more objectives, such as further analysis and reporting. When the original data sets—also referred to as *sources*—or the components that provide, i.e., the *sinks*, are separated into different systems, there is a rationale to also separate the corresponding components across systems. Sometimes this subdivision of concerns is a necessity, for example when these systems are required to live in different networks in practical situations.

The use of distributed systems is already established within different domains [25]. When it comes to analysis, there already exist frameworks for data processing and computation that parallelize tasks, allowing different systems to perform them simultaneously. Often, these frameworks focus on the tasks that take place downstream in the pipeline.

When it comes to handling multiple, large data sets, distributed data acquisition plays a major role within the pipeline. At each step, decisions are taken on how to collect, select, filter and alter data in order to make it usable in the next phase. The data sets provided by the sources come in different formats, which requires some overhead to make them usable in another component of the pipeline. Such transformations can be handled for each source. They could be split up according to the context, such as organizational units. This approach then suits existing network infrastructure.

Moreover, the data format plays a role in intermediate storage. Depending on how each component performs and interacts within the pipeline, data is often stored in a machine-readable format before further handling. The location of this storage also differs, ranging from disk-based files in directory structures to scalable database systems [26].

The means of data exchange between pipeline components is also based on the choices regarding the data format. A component could signal another that new data is available, potentially in the same format and message as the data itself.

Multiple dimensions of scalability exist in distributed systems for data collection [27]. First, the number of components—steps in the pipeline—is adjustable. Technical burden should be kept in mind here. A second dimension is the number of parallel workers of a single component. This number is flexible, assuming that the component was designed for reallocating such work. The third dimension is the number of storage locations for intermediate and final data sets. The effectiveness of distributing such tasks is limited by the availability of physical computing systems suitable for scaling the workload.

2.2.2 Agent-based communication

A distributed data-oriented system can be seen as a communication network. In this design, components are agents that perform semi-independent actions. Each agent interacts with some portion of an environment, including receiving input data. Once they have autonomously come up with an intermediate result, they make some parts of their neighborhood aware through signals. In some networks, the coordination of signals is handled by a control system. In another hierarchical network, the relations between agents depend on the stages they are in, similar to a pipeline [28].

Agent-based networks have concrete purposes within real-world applications, including motion tracking and formations of unmanned vehicles. The issues that come up in these practices are often well-modeled and lead to generalizable solutions for addressing effects of asynchronous updates and delays [29].

Another representation of data acquisition using distributed components is the Petri net [30]. In this type of network, nodes consume and produce tokens which are transferred between other nodes. This way, concurrency, control and other communication aspects is modeled for different components or agents within the network. Extensions to the basic Petri net enable the application of semantic attributes and activation rules to tokens, nodes and arcs within the network, making it more viable to follow the exact meaning at a less abstract level. This allows separation of workflows [31]. Detection and avoidance of deadlocks are also be possible through extensions [32]. Further, we are able to verify certain properties of the system that is described by the Petri net, enabling reasoning over a multi-agent system and making it safer to use [33].

2.2.3 Organizational approaches

Within corporate enterprises, data processing is often commonly associated with operational systems which produce a large amount of data, such as web servers. One approach to collecting the data from these systems is through an integrated system that passes messages around [34]. Often, the purpose of the data collection is to be further analyzed for business intelligence goals, leading to visualizations and reports.

In these circumstances, some terminology comes in place that lack a clear, generalized definition. *Big data* often does not refer solely to the size of the data set, but rather to the

complexity in establishing relations and discovering patterns. This complexity often leads to the introduction of data management systems that augment common databases, which are able to model or at least store the data types. Whereas a *data warehouse* stores data that is homogeneous in shape, leading to a common data model, often another option is taken where textual, visual, temporal and diverse data types are stored in potentially separate systems, leading to a *data lake* [35]. The disadvantage of this approach is that raw data is not easily processed and combined into a single model.

Other concerns for large-scale businesses in the area of data processing are auditing, access restrictions, on-premise versus cloud-based platforms and off-site archival systems [36]. The collected data might contain personally identifying information, which requires assessment and mitigation of the impact on privacy of users and staff members. Separation of components into virtual networks helps with reducing the possibility of unauthorized access. Still, both raw and processed data constitutes sensitive information for the organization. Accordingly, early encryption within the data pipeline helps addressing concerns regarding privacy and business information security.

2.3 Method

Prior to discovering what kind of patterns exist that provide indications of progress within a software development process, we need to determine which data holds these patterns and how we collect the appropriate data sets. As mentioned in Chapter 1, we focus on extracting these patterns from SCRUM software development processes, although the collection of data from multiple systems does not rely specifically on the presence of SCRUM terminology in the disclosed information. In fact, not all systems that we consider relevant to the process are specific to one development framework, as some are commonly used within software development for version control, source code quality checking and issue tracking.

The pipeline we design consists of multiple phases, where the data acquisition is the first step towards consolidation in a database, allowing the combined data set to be analyzed for various purposes within the domains of machine learning and information visualization. Figure 2.2 provides an overview of the pipeline at an abstract component level. In this figure, blocks represent states that data can be in, including potential data sources such as issue trackers, version control systems for source code repositories and quality control systems, which are used during the software development process. Later on in the pipeline, the blocks indicate intermediate artifacts and results. The arrows are actions that select, transform and move the data to its next state. These actions are performed by software components that we develop in several programming languages, including query languages. The code is fine-tuned for specific purposes within the pipeline.

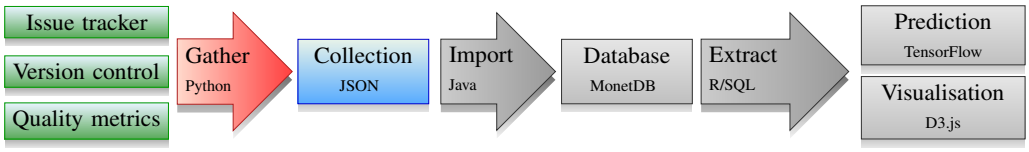

Figure 2.2: High-level overview of the Grip on Software pipeline, with the data sources (green), data acquisition components (red) and resulting artifacts (blue) highlighted. Gray elements are described in later chapters.

The main goal of our research is to improve software development practices by extracting, combining and analyzing data that would otherwise be left in separate systems, with no interconnection between the applications that the data originates from. As such, there are some important objectives when it comes to inspection, curation and disclosure of the data and the results:

- When data from different sources refer to the same entity, such as a team, project, component, developer or code change, we should establish this link as soon as possible. This way, we ensure that the equivalence or relationship is known for later components in the pipeline, when this was not available before the introduction of the pipeline. Additionally, this allows encryption of personal or project-sensitive data to take place early on.

- The pipeline has to work within different ecosystems. The existing infrastructure should remain unchanged, i.e., there are no additional requirements that stem from the pipeline. We split up components across networks without performance bottlenecks or difficult firewall configuration, by using typical connection protocols. Projects that are in separate networks have their own *agent* which collects relevant data. Intermediate artifacts are exchanged using an easily interpretable data interchange format to a central *controller*.

- Because the data comes from different sources, depending on which organization, team or project is involved, there should be an easy and versatile method of configuring the means of collection. This *configurator* indicates which web applications are approached by the data acquisition agents. It then stores the credentials to use and provides specific status information regarding the agent, which helps with control and monitoring of the pipeline.

- The pipeline's entire state should be easily visible within an overview. A *status dashboard* indicates problematic situations. This includes recentness of data collection by the agents as well as access to error logging. The dashboard also allows adjustments to scheduling in order to rerun specific tasks, for example if they failed.

There are further design principles and non-functional requirements that we consider when implementing the pipeline and its individual components. The pipeline should internally track when the most recent update of different data sources was successfully performed, so that we avoid loss of data. There should not be a performance impact on the usual development process of the teams included in the study, i.e., it should not lead to availability problems or other interference of the systems used as a source of data. Several more considerations help with reproducibility of the pipeline setup, such as testability and proper documentation of components and methods.

2.3.1 Data acquisition

We collect data regarding the progress of multiple SCRUM software development projects from various systems. In order to do so in accordance with our objectives, we introduce new components for our data acquisition pipeline.

The agent component connects to various application programming interfaces (APIs), using queries and filters to determine whether there are any changes since the most recent moment that data was collected for a project. The agent then produces structures that describe the fresh data using the JavaScript Object Notation (JSON) format. Table 2.1 lists some types of data collected from different categories of data sources. Additional components configure the data acquisition

Entity	Tracker	Version control	Review	Quality	Build
Issue (story, etc.)	✓	✗	✗	✗	✗
Sprint	✓	✗	✗	✗	✗
Comment	✓	✗	✓	✗	✗
Person (developer)	✓	✓	✓	✗	✗
Commit version	✗	✓	✗	✗	✗
Release tag	✗	✓	✗	✗	✗
Merge request	✗	✗	✓	✗	✗
Metric measurement	✗	✗	✗	✓	✗
Metric target	✗	✗	✗	✓	✗
Usage status	✗	✗	✗	✗	✓

Table 2.1: Types of data that are collected by the data acquisition components and the category of systems that provide this data (✓) or not (✗). Some types of data—metadata and dependent entities—are left out of this overview.

process, ensuring proper authentication at the defined systems. The components also track the frequency of the data collection runs. The agent passes errors through to a monitoring dashboard.

This setup allows us to deploy the components in separate ecosystems. For example, at one of the organizations involved in our research, ICTU, each software development project has its own virtualized network where resources such as a build platform, version control and quality monitoring systems are made available. The platform allows additional systems to be set up through a Docker-based setup [37]. The Docker instances were managed through a specialized interface known as BigBoat [VII] developed by the organization, although they later replaced it with another interface for practical purposes. Two data acquisition components—an agent which regularly collects data from known systems in the same network and a configurator that selects the systems and authentication. Together, the agent and configurator form the GROS gatherer, with one for each virtual network as a distributed system.

The intermediate artifacts are passed by each agent to a centralized controller within another network. A Secure Shell (SSH) connection protocol is specifically allowed through a firewall and adjusted to only accept secure data transfer from the agents. The controller provides additional environment information to the agents at the start of a data collection run, so that these do not need to be configured separately. The controller also acts as an additional checkpoint to adjust the collection interval, as a form of a *preflight* checklist. If this preflight check succeeds, then the controller provides encryption keys for early encryption of personal data.

Once the collection and transfer phases are complete, the resulting data is imported into a database hosted at the organization. Certain data that is not retrieved by the agent, e.g., from a centralized or cloud-based issue tracker, is retrieved by another instance of the gatherer running on a virtual machine (VM) or Jenkins [IV] instance. The controller sends a *trigger* notification to this instance to do so. Figure 2.3 displays the components and their connections in this setup.

The combined update of the data means that the organization's database is usable for local data analysis, prediction algorithms and visualization. The data is also regularly exported and uploaded to a central Grip on Software database, such that analysis also takes place using combined data from multiple organizations in one place.

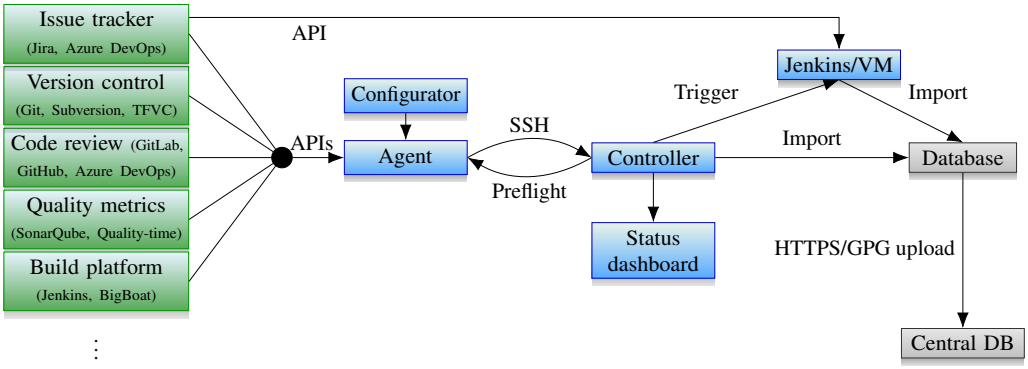

Figure 2.3: Overview of the data acquisition components and their interactions within the Grip on Software pipeline. The green blocks represent possible systems in the existing development ecosystem. Blue blocks are components of the pipeline. Gray blocks are components that are described in later chapters. Arrows indicate the direction that data travels in.

In other ecosystems, a distributed setup would be superfluous. When all teams work on a number of components of the same project, hosted on one platform—such as at Wigo4it—there is only a need for a single access point. The collected data is then either imported into a local database or uploaded directly to the central database.

For many of the specific systems that we use as a source of software development process data, we build modules that provide API connections to them. This makes it easier to update the pipeline components when changes to the APIs are not backward-compatible and unfortunately require changes within the pipeline code as well. For example, in Jenkins or older versions of Azure DevOps—previously known as Team Foundation Server (TFS) and Visual Studio Team System (VSTS) [VIII]—we use specific connections to remain compatible with existing deployments, such as the older version control component of TFS known as TFVC. We build around these modules to define our own models for the data types, which describe the data source systems on a higher level. This domain-based approach allows separating the authentication configuration from the connection itself, enhancing the modularization of each step.

The pipeline mostly operates automatically based on schedules. The agent frequently attempts to retrieve changes to the source data, as is common in a fast-paced SCRUM process. The configurator and the status dashboard are a means to adjust how the automated runs operate, by providing options to change how the agent connects to the other systems or to reschedule the agent's execution if earlier operations failed due to problems that have since been amended. The agent will then collect data as of the most recent finished run. The configurator provides a systematic access to this part of the pipeline. Figure 2.4 displays how the configurator looks like to the user—most likely researchers who monitor the pipeline's progress, but possibly including team members involved in the project that the agent collects data from.

In this interface, several systems are configurable, which includes filling in URLs, mappings of identifiers and authentication credentials. If code repositories or build toolchains are split up across multiple instances, additional systems can be configured as well. For security reasons, after the configuration is updated, the credentials in this interface are instead indicated with a placeholder, which keeps the old credentials if it is unchanged. The credentials are still modifiable

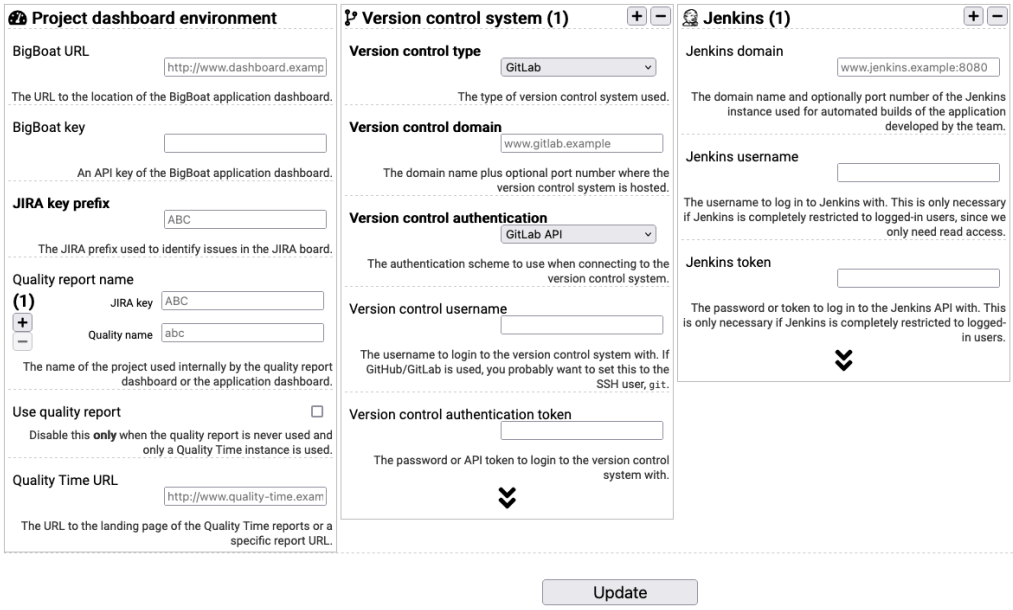

Figure 2.4: Screenshot of the configurator, with options to set up connections to various systems acting as a data source.

at a later moment. The agent always uses the most recently updated configuration to connect to the defined systems. The agent also selects more relevant systems from the quality control system and the tracked code repositories.

Personal data is collected by the agent and included in the artifacts in an encrypted manner. The encryption keys are used to perform a one-way encoding. This means that the data—e.g., a name or email address—can only be inspected if the attacker already has the original data as well as the keys. Still, this form of data protection is considered to be a pseudonymization rather than true anonymization, since it makes use of the original data for the encoding. In general, personal data is not used during the analysis since it is not relevant for the overall SCRUM process. We sometimes compare the resulting hashes against encrypted personal data from other systems or from an earlier run. We use the encrypted names of developers to avoid duplicate entries of the same person across multiple systems. Another instance of pseudonymous data use is when we extract the domain name of an email address to check if people are employed by the organization or by the client, for example. During exports to the centralized database, the encryption keys are not included in the upload and remain at the organization. The data is further secured through HTTPS and GPG encryption [IX], avoiding interception of the data during the transfer. The analysis only uses pseudonymous data.

2.3.2 Further pipeline steps

The data acquisition components form only the first phase of a pipeline that is aimed toward providing insight into patterns of software development processes. The proper functioning of the

pipeline depends on more phases that consolidate the data of software development projects in a (local or centralized) database and that extract relevant data points such as features from this database. These are then used within prediction and estimation models as well as in information visualization. These correspond to the remaining components that are colored gray in Figure 2.2.

Each phase has software in order to transfer data to the next phase and maintain the proper functioning of the pipeline. The following components are part of the pipeline of the Grip on Software research project. The code repositories of these components have all been made open source. The items in the list refer to the repositories whose references are found in Appendix A. We also indicate the programming languages and the kinds of tests for the component. The list is grouped by the phases of the pipeline, which are described more extensively in various chapters of this thesis.

Data acquisition (Section 2.3.1):

- `data-gathering` [a]: A collection agent for data from software development processes. The schedule-based gatherer has deployment options for different data sources and contexts. Module-based Python 3 code with type checking. Integration test runs are possible on Jenkins.
- `agent-config` [b]: A web-based configurator for the GROS data gathering agent. Intended to be deployed with the agent in a Docker Compose system. JavaScript-based, relatively well tested and formatted using routes, templating and some utility functions.
- `status-dashboard` [c]: A web-based dashboard providing an overview of data collection agents and their status, with scheduling options. Written in Python 3 with typing.

Data import and export (Chapter 3):

- `monetdb-import` [d]: A schema-based database importer. Works with MonetDB [38]. Java-based package format, schemas and update files plus some Python 3 and Bash code for database administration as well as validation. Contains some unit tests for utilities.
- `monetdb-dumper` [e]: A Java-based database exporter for MonetDB. The administrative interface for importing and exporting dumps tie in with this package.
- `export-exchange` [f]: A module that handles export of the database and secure upload to a centralized instance. Written in Python 3 with typing.
- `upload` [g]: A web server that receives secure uploads of database exports to load into a centralized database. Written in Python 3.

Analysis and pattern recognition (Chapter 4):

- `data-analysis` [h]: Analysis interface for aggregating data with source traceback from a filled MonetDB database. R-based integration [X]. Configurable, dynamic queries that adapt to organizational differences.
- `prediction` [i]: Models to predict, classify, analyze and estimate features and labels regarding SCRUM data. Python 3 with TensorFlow [XI] and Keras.

Visualizations (Chapter 5):

- `visualization-site` [j]: Landing dashboard site for different/related visualizations. Configuration backbone. Contains integration tests for deployment within a virtualized network based on Docker compose [XII] using several web servers that could act as proxies, testing the main site and the specific visualizations.
- `visualization-ui` [k]: JavaScript modules that are reused in various visualizations. Includes unit tests.
- `sprint-report` [l]: Dynamic generator of various comparative visualizations inter- and intra-project per-sprint.
- `prediction-site` [m]: Human-readable output of predictions.
- `timeline` [n]: Various temporal data from the software development process.
- `leaderboard` [o]: Project-level statistics.
- `collaboration-graph` [p]: Relations between development team members and projects that they worked on.
- `process-flow` [q]: Flowchart of the status progress of user stories with volume and time metrics.
- `heatmap` [r]: Software development code commit activity.
- `bigboat-status` [s]: System reliability graphs for the BigBoat development platform.
- `backlog-burndown` [t]: Effort burndown chart for product backlogs.
- `backlog-progression` [u]: Progression inspection chart for product backlogs.
- `backlog-relationship` [v]: Issue relationship chart for product backlogs.

Supplemental components:

- `deployer` [w]: A web application to deploy repositories to a managed machine, with configuration to use a quality gate, run commands, manage secret files and restart services. Written in Python 3 with typing.
- `server-framework` [x]: A Python 3 module to develop authenticated web servers.
- `jenkins-cleanup` [y]: Maintenance scripts to remove stale Docker, Jenkins and SonarQube data during pipeline development. Written in Python 3 with typing and Bash.
- `coverage-collector` [z]: A service that collects coverage information in JavaScript during browser tests. Used in combination with the visualization site integration/unit tests.

2.4 Technical considerations

We design our data pipeline with the purpose of gathering knowledge regarding SCRUM software development processes. We deploy the pipeline within ecosystems with their own requirements and existing behaviors. For these reasons, there are several considerations when deciding how to design, develop, document and deploy the pipeline components.

Aside from our objective of acquiring and understanding data, we consider the pipeline itself to be a subject for study. Based on the concepts that are realized in a distributed system, we discuss how we generalize the implementation to other development ecosystems. Additionally, we demonstrate how state-of-the-art algorithms augment the components.

In this section, we discuss some of these concepts and characteristics that we find relevant for further consideration.

2.4.1 Generalizability

In principle, the components that are used for collecting, disseminating and presenting data from different sources are able to be placed in different development ecosystems. The individual components work on build platforms such as Jenkins [IV], Docker-based platforms, virtual machines or servers, with a limited number of base dependencies installed. Due to this, the code has run at two different software development organizations as well as at a university, each with different requirements for collecting, publishing, sending and receiving data between internal and external networks.

The sources where we collect data are specific to our research scope. This includes various project management systems, version control systems, quality review systems, build platforms and organizational resources. We support TFS/VSTS/Azure DevOps Server [VIII], Jira [I], GitLab [III], GitHub [XIII], Subversion [XIV], SonarQube [V], Quality-time [VI], BigBoat [VII], Jenkins [IV], LDAP [39] and TOPdesk [XV].

As these project management products get updated or replaced in the future, the data gathering components continue to use APIs that become outdated and eventually become unable to collect new data. This is the main reason that it would require continuous and adaptive maintenance to keep the software relevant.

Certain pipeline components are harder to configure in order to work in a specific ecosystem. Setting up a MonetDB database with different parts written in Python 3, Java and R is cumbersome if there is no proper platform. At each point that a portion of pipeline is deployed to a specific environment, certain decisions need to be made in order to assure that the pipeline is able to work there. Some components provide a configurator, with intuitive means of selecting and entering the details of the connections that they make. The accompanying documentation indicates additional configuration options.

In the end, the pipeline was set up with a specific purpose in mind: collecting data regarding multiple development projects together in order to find relevant patterns. Some components will not be suited for different purposes, given that the data gathering is focused on specific systems. On some points, scalability and performance are kept in mind. The pipeline however does not work out of the box in a completely new ecosystem without prior knowledge. Thus the components are provided “as is”, meaning that some adjustments may be required. This is inherent to any open source system that seeks generalization; other interested parties can contribute to make it fit in a specific ecosystem.

2.4.2 Continuous integration

The components primarily use Jenkins [IV] as a continuous integration (CI) system, which is geared towards performing tests, building the software and collecting data automatically. Jenkins provides an interface to view and manage these steps. Jobs are started to execute tasks based on schedules and code changes.

Our code repositories define their own Declarative Pipeline, a syntax which indicates the requirements and stages to build, test, analyze and deploy the code. The usual purpose of a CI system is to allow rapid merging of development code into a stable version supported by automated review, which is still one of the features used for our pipeline components. Builds are regularly scheduled so that problems with changes and dependencies are found early. Several components have unit tests and integration tests. Code analysis is performed using a SonarQube scanner [V] in order to track the pipeline component's code quality, which also uses information from test results and coverage.

Jenkins usually works with a controller node—running on the server on which the CI system is installed—and agent-based executor nodes. Each node handles one or more concurrent tasks. This way, builds of multiple code repositories take place on separate machines, which are synchronized with regards to the platform dependencies. Additionally, specialistic nodes are possible, for example to perform machine learning tasks on a server equipped with a graphics processing unit (GPU). Portions of a build take place on different nodes, which are aggregated before publishing the results on another server.

2.4.3 Documentation

All of the pipeline components include documentation regarding deployment and configuration. The data acquisition component is documented on function, class and module level. The database component is documented through human-readable versions of the schema as well as individual import modules with documentation on parameters. The prediction and information visualization components also come with documentation for modules and methods. Data formats exchanged between all components are documented with JSON specifications[‡], including API definitions[§], which allows validation of instances of data artifacts against the specified schema as well as meta-validation.

Each code repository has some form of documentation. All components come with instructions that help with building, configuration and installation, usually with several example options using Docker, Jenkins or a standalone form. The documentation often describes how to start up the component or how to run individual programs, with details on command-line arguments through help systems. Web-based systems such as the configurator and the visualizations describe in detail what certain controls and options do, using tooltips, for example.

Aside from the standalone documentation, the implementations of scientific analysis systems are also documented through publications, including this thesis. Furthermore, we write comments for code lines, methods, modules and components, considering reproducibility and maintainability of the pipeline at each level.

[‡] https://gros.liacs.nl/schema/
[§] https://gros.liacs.nl/swagger/

2.4.4 Novelty

As mentioned before, the components of the pipeline beyond data acquisition are described in more detail in their own chapters. But some discussion does not fit the topic at hand when the main focus is database modeling, pattern recognition or information visualization, for example because this would make the narrative there too verbose and technical. Still, we find it relevant to describe some technical details, such as a new implementations of an algorithm.

For the MonetDB database, we developed some administrative programs [d] as well as an importer which is described in more detail in Section 3.4. The importer plays a crucial role when the data acquisition components provide new data that should be stored in the Grip on Software database. One potential issue that we resolved is compatibility with importing older versions of JSON artifacts. Sometimes, these files are kept as importable source objects after a software development project is completed and the original systems turned off. Aside from backward compatibility, forward compatibility could also be relevant if different versions of agents and importers were in use. Although the agents are able to check for updates—with the configurator reporting this—and a new version of the Docker image is easily deployed, the versions could still become desynchronized. The artifacts and importer were designed in such a way that later additions do not hinder the import process, while missing fields due to agents that were not updated are also handled properly. Often, these fields are filled in during a later import as well.

We frequently update the database itself to support a new schema through a program that validates the state of the model. Based on structured information of an update, the program checks if certain database columns or keys exist and have the proper types. Based on this detection system, schema updates will then take place or are skipped.

In order to extract the features that describe characteristics of SCRUM sprints, as mentioned further in Section 4.4.1, we developed an R toolkit program [h] that performs queries upon the database. The queries are based on templates which contain variables and other structures, enabling R code to fill in proper field names and conditions. This hybrid system within MonetDB properly selects data and filters out noisy data, such as the practice of giving a story a very high number of points to avoid working on it in a recent sprint. A structured list tracks the available definitions for these fields and conditions for reuse in other places. Based on the ecosystem, this enables some queries to select the same kind of features from different sources, such as Jira and TFS/VSTS/Azure DevOps. The queries themselves are also tracked, including metadata to describe the source through a specific, human-readable URL with the same data, as well as aggregation options, labels and unit formatting preferences. These come in handy for the display of such features in the information visualizations described in Chapter 5.

As for the pattern recognition methods, we make use of TensorFlow [XI] and related technologies to train, evaluate and predict outcomes of samples within data sets, which are made up from the features extracted for the sprints from the filled database. We implement several models to do so [i]. One novelty here is the use of TensorFlow to implement analogy-based effort estimation (ABE). This algorithm, described in Section 4.4.3, does not necessarily stem from machine learning, but still includes steps that are well-suited to be performed in batch, such as finding similar sprints based on a distance measure to neighboring samples. Such steps can then be performed upon a GPU using vector-based instructions generated through TensorFlow. This allows for further use of specialized hardware within our pipeline, leading to performance improvements. This lets us bring the estimation and prediction results to the SCRUM team members more quickly, demonstrating the advantages and usefulness of the pipeline's construction.

2.5 Results

In order to determine whether the pipeline components can be set up in a stable manner without influencing an existing software development ecosystem, we tested the performance of the components by enabling them for 22 different development projects. The development ecosystem is described in detail for the agent-based data acquisition system from Section 2.3.1: each project has its own virtualized network with a BigBoat [VII] deployment platform, including all services for development such as version control and code review, usually GitLab [III]. Code quality is monitored through SonarQube [V] and Quality-time [VI].

We deploy the agent and configurator on this platform using Docker Compose [XII]. The controller runs on a separate VM in an isolated network, along with the status dashboard. A Jenkins continuous integration system [IV] collects data for the projects from the Jira issue tracker [I] based upon a trigger from the controller.

We tracked the CPU processing load, RAM memory usage and disk storage space of the BigBoat platforms. Overall, the agent did not cause a noticeable impact to the normal usage trend of the platform. The performance did not differ between runs that took place hourly and every 15 minutes. More frequent schedules could be possible, but we do not consider this to be necessary, since most other components within the pipeline were not aimed at live data presentation. Additionally, if the complete collection run—both the data collection by the agent and the database import performed at the controller—would take longer than the frequency, a concurrent run is not allowed to be started for the same agent in order to prevent conflicts. This avoids an avalanche of jobs working on the same task of acquiring a large bulk of new data, for example when a project with a long lifespan is newly included in our research.

One problem that surfaced was the use of large source code repositories by some teams. The data collection would retrieve new versions of code commits, but for some version control systems this still caused some overhead, in particular Subversion. Figure 2.5 shows performance downgrades that we detected in this situation. We included some mitigations to reduce the time spent with file differences and to limit memory usage. This helped ensure that the platform remains available for its primary purpose.

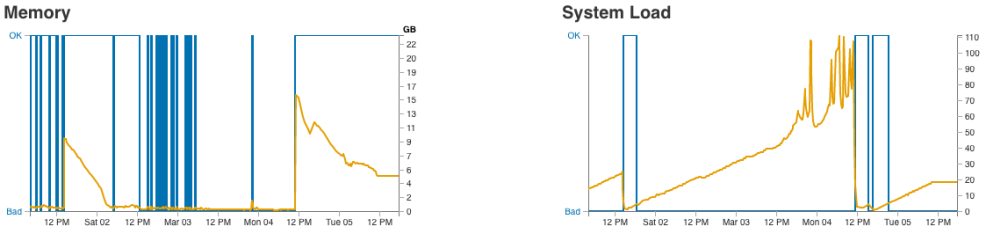

Figure 2.5: Graphs displaying the available memory (in gigabytes) and CPU processing load over time for a development project's platform when problems arise with collecting Subversion repositories. The blue line indicates when the platform signals a decrease in performance, while the yellow line shows the actual value of the metric. After this event, problems with load were mitigated.

2.6 Discussion

This chapter introduces the components of the pipeline for the Grip on Software research, aimed at understanding situations arising from the application of the SCRUM software development method. In particular, we detail the design and construction of the data acquisition components that constitute the first phase of our pipeline, aimed at consolidating various data sources for pattern recognition and information visualization. We demonstrate how we apply concepts from distributed computing and agent-based networking in order to overcome organizational differences in ecosystem structure. We document our objectives and mention the components of the entire pipeline. The other phases of the pipeline are discussed based on their technical aspects, such as source code and documentation. From these components, we highlight some non-functional requirements and new work that they provide.

We conclude that our data acquisition pipeline works well at the two organizations where it has been applied. The early encryption and audit logging provisions within the components help with ensuring secure communication and storage. We are able to uniquely identify entities across systems and keep track of where data came from, as a means of data provenance, making it easier to reproduce results of later stages of the pipeline.

Another non-functional requirement is the performance of the pipeline. We reuse the logging, scheduling and usage statistics to measure and improve the efficiency of the processes. Based on our results from experiments with the setup in the distributed networks, we determine that it is possible to follow updates close to real time, insofar that this is helpful for the machine learning models and reports, which are similarly updated frequently. We are able to resolve issues with high load through code simplification and usage limits without impacting our goals.

The components of the pipeline were developed for the purpose of collecting and analyzing data from software development processes. We demonstrate the application of the pipeline in multiple ecosystems that software development organizations have in practice. The code repositories of the components are open source, allowing reusability.

In the end, the pipeline is the instrument in order to substantiate our main research objective of finding relevant patterns within SCRUM software development processes. It is important to make this instrument work properly from the ground up. It would also be a waste if it would simply be shelved after it has been used just once.