Polyglot and Distributed Software Repository Mining with Crossflow

Konstantinos Barmpis
Department of Computer Science
University of York
York, UK
konstantinos.barmpis@york.ac.uk

Patrick Neubauer
Department of Computer Science
University of York
York, UK
patrick.neubauer@york.ac.uk

Jonathan Co
Department of Computer Science
University of York
York, UK
jonathan.co@york.ac.uk

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Nicholas Matragkas
Department of Computer Science
University of York
York, UK
nicholas.matragkas@york.ac.uk

Richard F. Paige
Dept. of Computing and Software
McMaster University
Ontario, Canada
paigeri@mcmaster.ca

ABSTRACT

Mining software repositories at a large scale typically requires substantial computational and storage resources. This creates an increasing need for repository mining programs to be executed in a distributed manner, such that remote collaborators can contribute local computational and storage resources. In this paper we present Crossflow, a novel framework for building polyglot distributed repository mining programs. We demonstrate how Crossflow offers delegation of mining jobs to remote workers and can cache their results, how such workers are able to implement advanced behavior like load balancing and rejecting jobs they either cannot perform or would execute sub-optimally, and how workers of the same analysis program can be written in different programing languages like Java and Python, executing only relevant parts of the program described in that language.

CCS CONCEPTS

• **Information systems → Data mining**; • **Software and its engineering → Concurrent programming structures**.

KEYWORDS

Mining software repositories, domain-specific modeling language, scalable, ease of use, lower barrier to entry

ACM Reference Format:
Konstantinos Barmpis, Patrick Neubauer, Jonathan Co, Dimitris Kolovos, Nicholas Matragkas, and Richard F. Paige. 2020. Polyglot and Distributed Software Repository Mining with Crossflow. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387481

In [9], we presented a short (4-page) overview of an early version of Crossflow, a Java-based framework for development and distributed execution of multi-step software repository mining programs (workflows). This preliminary work focused on motivating the need for a tool facilitating distributed execution of software repository mining programs that would allow remote collaborators to contribute their local computational and storage resources. Crossflow was developed to address a number of challenges we identified in the context of a repository mining use-case [10], where we set out to assess the "popularity" of 22 different model-driven engineering technologies by measuring their use in open-source GitHub repositories. The analysis involved more than 40,000 GitHub API calls, well over the 5,000 calls per hour offered by GitHub, and as such we had to introduce artificial delays to our mining program resulting in an execution time of over 8 hours. We did not wish to use multiple credentials or pre-authorized tokens from the same machine as this is against GitHub's API policy[1].

In addition, given the fragility of the network and the intermittent errors that GitHub's API would produce (e.g., when it was overloaded), our mining program needed to include a substantial amount of defensive (wait-and-retry) code against network errors, as well as several bespoke persistent caches to minimize the number of API calls that would have to be repeated in case of an unexpected program crash. Weaving all these concerns together lead to a tightly-coupled program that was hard to reuse and to run in a distributed manner (e.g., to share the load between the different collaborators involved in this work).

This experience motivated us to investigate approaches for distributed execution of software repository mining programs that would allow remote collaborators to contribute their computational and storage resources. Moreover, the fluctuating availability of workers as well as the elimination of centralized book-keeping motivated us to implement a locality scheduling mechanism that allows worker nodes to dynamically reject jobs during runtime. Additionally, storage and bandwidth limitations associated with

[1]https://help.github.com/en/github/site-policy/github-terms-of-service#h-api-terms

centrally-stored data motivated us to develop a worker-based capability approach that enables jobs, which require particular repository data, to be picked up by workers that have the required repository available in their local storage.

This paper extends the preliminary work in [9] and provides a more comprehensive description of Crossflow[2], focusing on new capabilities such as polyglot support. It also reports on empirical evaluation of Crossflow against an existing repository mining tool, using a case-study from the literature. The rest of the paper is organized as follows. Section 1, presents the architecture of Crossflow, the domain-specific language it uses for specifying multi-step software repository mining programs (workflows), as well as key features such as caching, locality scheduling and polyglot code generation. Section 2 presents related work, section 3 presents a qualitative and quantitative analysis of Crossflow in comparison to an existing tool and case-study, and section 4 concludes the paper and discusses directions for future work.

1 CROSSFLOW

Crossflow is a novel polyglot distributed data processing framework, tailored to the needs of collaborative repository mining. Crossflow supports distributing tasks of multi-step repository mining programs, which we will refer to as *workflows* in the remainder of the paper, over multiple computing nodes (workers), which communicate through, and are orchestrated by, a master node using messaging middleware. Each such worker is written in one of the programing languages supported by Crossflow and will execute relevant steps (tasks) of the workflow written in that language. The current implementation of Crossflow employs Apache ActiveMQ [12] as messaging middleware and supports workflow tasks that are implemented in Java and Python.

Technology	Extension	Keyword
GMF	.gmfgraph	figure
ATL	.atl	rule
QVTo	.qvto	transformation

Table 1: Technologies with file extensions and keywords

We explain the building blocks and facilities of Crossflow through a running example. In this example the goal is to identify the degree to which different technologies (e.g. programming languages, tools) are used together in the same GitHub repository. For instance, a goal is to identify if projects which employ the Eclipse Graphical Modelling Framework (GMF)[3] are more likely to also use the ATL [6] or the QVTo[4] model transformation languages. One way to achieve this is to:

- Record information about the technologies of interest in a structured format. For example, the CSV file seen in Table 1, captures a known file extension and keyword for each technology of interest;
- Query GitHub (using its public repository search API) to find repositories containing files of interest for each technology (this API returns the details of up to 1000 files for each distinct search);
- Clone the repositories hosting the collected files and search these local clones for files of all technologies of interest;
- Compute a co-occurrence matrix like the one in Table 2.

	GMF	ATL	QVTo
GMF	-	16	25
ATL	16	-	7
QVTo	25	7	-

Table 2: Technology co-occurrence matrix

1.1 Architecture

As illustrated in Figure 1, Crossflow provides a purpose-built domain-specific language for modeling repository mining workflows, and code generators that produce implementation scaffolding, which depends on reusable runtime libraries (currently supporting Java and Python). While a Crossflow model specifies the sources, tasks, streams and sinks of a workflow, how they are wired, and where tasks are executed (in all workers vs. only in the master node) it does not capture the behavior of the modeled sources, tasks and sinks. This is expressed using hand-written code embedded in the generated scaffolding; such skeletons will be in the programing language defined for that task in the model (or in Java by default). Once the desirable behavior has been implemented, this workflow-specific code and the reusable core runtime library are bundled into self-contained runnables, to be executed on the nodes participating in the execution of the workflow. As such, a node is able to run one or more Crossflow workers in one or more of the languages defined in the Crossflow model, contributing to the analysis of tasks implemented in those languages. The following sections discuss the components of the Crossflow architecture in detail.

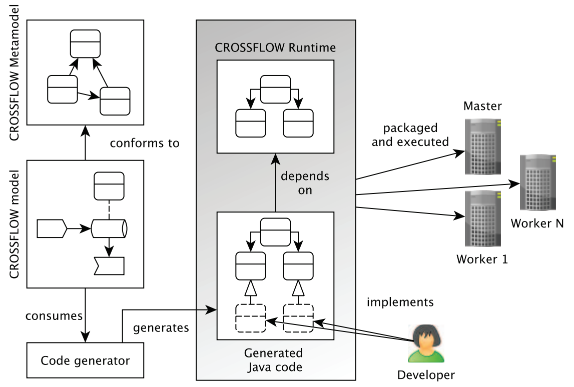


Figure 1: Crossflow Architecture, based on [9]

1.2 Crossflow DSL

The Crossflow DSL has been implemented on top of the Eclipse Modeling Framework and its abstract syntax (metamodel) is illustrated in Figure 2. We explain its building blocks using a graphical model (cf. Figure 3) of our running example.

Sources feed the workflow with jobs based on user input. In our example, *TechnologySource* reads a comma-separated file structured

[2]https://github.com/crossflowlabs/crossflow
[3]https://www.eclipse.org/gmf-tooling/
[4]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

like Table 1, producing *Technology* jobs, consisting of a name, an extension and a keyword, into the *Technologies* stream. Sources are only executed on the master node and will only run once, upon workflow initialization. Since sources (like all Tasks) are able to provide their data asynchronously, one element at a time, a source can be providing new jobs for an arbitrary amount of time during the execution of a workflow.

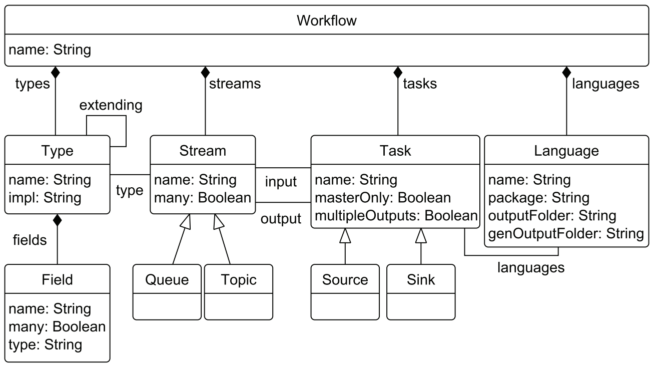


Figure 2: Crossflow Language Metamodel

Streams are message channels to which sources and tasks of the workflow can send jobs that other tasks can carry out, or results that sinks can aggregate and persist. In Crossflow, job steams are typed: for example the *Technologies* stream only accepts jobs of type *Technology* (dashed line in the diagram). Crossflow supports two types of streams: queues which send each job to only one of the subscribed workers, and topics which broadcast each job to all subscribed workers.

Tasks subscribe to one or more (incoming) streams and receive jobs posted there by other tasks or sources. They can also post new jobs to one or more outgoing streams. For example:

- the *GitHubCodeSearcher* task subscribes to the *Technologies* stream, processes incoming jobs of type *Technology* by searching for files with the specified keyword and extension through the GitHub API, and for each result, it pushes its repository path (wrapped into a *Repository* job) to the *Repositories* stream. In a distributed execution of this workflow, each worker contributes an instance of *GitHubCodeSearcher* (hence the double rectangle node shape in the diagram) which can perform such GitHub searches under its own credentials;
- the *RepositorySearchDispatcher* task receives these repositories, and for every repository it has not encountered before, it produces one *RepositorySearch* job into the *RepositorySearchesStream*. Unlike *GitHubCodeSearcher*, *RepositorySearchDispatcher* is represented with a single rectangle, signifying that only one instance of the task is executed and this instance lives on the master node. This does not have a significant impact on the performance of the workflow as the cost of filtering out duplicate repository IDs is negligible to that of querying GitHub and cloning Git repositories;
- for each *RepositorySearch* job, the *RepositorySearcher* makes a shallow clone of the Git repository and counts the number of files with the extension and containing the keyword relevant to each technology.

Sink components can subscribe to one or more streams and receive results to aggregate/persist. For instance, the *ResultsSink* sink in the example collects *RepositorySearchResult*s, builds the co-occurrence matrix seen in Table 2, and periodically persists it into another CSV file. As with sources, sinks are only executed on the master node.

Languages define the programing language(s) that a task is to be implemented in. The running example defines two languages: Java and Python[5]; the task *GitHubCodeSearcher* is both a Java and a Python task hence it has two redundant implementations and can be executed by a Crossflow worker running in either Java or Python. Sources and Sinks will always only be run by the master, which runs in Java.

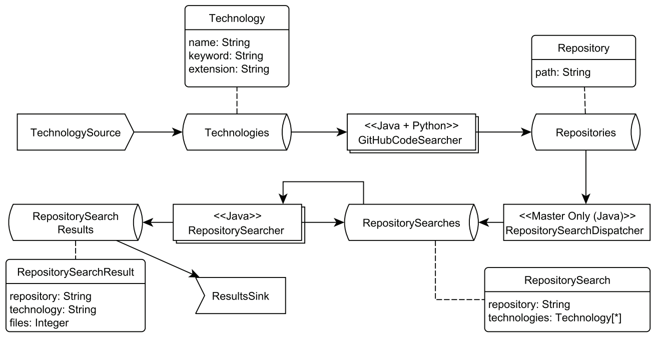


Figure 3: Crossflow Model of the Running Example

1.3 Code Generator

Each language supported by Crossflow has an associated code generator that can consume a workflow model and produce the appropriate scaffolding code. Java generation is always performed as master node mode is only supported by Java and it's use as the default language. For all other languages, the associated generator is only executed if the language is defined within the workflow model. Figure 4 shows a simplified type hierarchy for the running example, in particular:

- For every *Task* in the model that is applicable for the generator's language, an abstract base class containing infrastructure-communication code and a skeleton subclass is produced. The skeleton contains one *consumeXYZ(...)* method for every incoming stream, where hand-written code needs to be added to handle incoming jobs. For example, from the *GitHubCodeSearcher* task of Figure 3 both the Java and Python generators produce an abstract *GitHubCodeSearcherBase* base class and concrete *GitHubCodeSearcher* class that extends the base class. Within the concrete class, an empty stub implementation of a *consumeTechnologies(...)* method is produced, which is called by the workflow when a new *Technology* job is received. Partial hand-written bodies of these methods are illustrated in Listing 1 and Listing 2.
- Similarly, *Sinks* are generated in the same manner as *Tasks* though only to Java as both *Sources* and *Sinks* must be run on a master node.

[5]even though direct links between language and task are hidden (to avoid cluttering the model), all tasks can be associated to one or more languages (with Java being used as a default when no languages are specified)

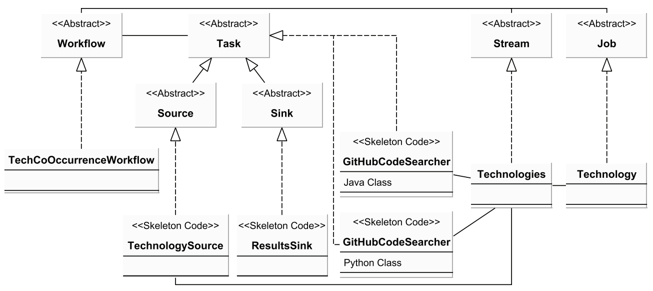


Figure 4: Generated Classes for the Running Example

- For every *Source*, an abstract base class and skeleton subclass is produced by the Java generator. Within the *Source* skeleton subclass an empty *produce()* method is generated for developers to implement and specify the behavior of the source (e.g. for *TechnologySource*, the implementation of *produce* reads file extensions and keywords from an input CSV file and pushes *Technology* jobs to the *Technologies* stream).
- The abstract base classes generated from *Task*s and *Source*s, also contain one *sendXYZ(...)* method for each outgoing stream that developers can use to send outgoing jobs to the respective stream. For example, *GitHubCodeSearcherBase* contains *sendToRepositories(...)* used by hand-written code in its concrete subclass to send *Repository* jobs to the *Repositories* stream, for *RepositorySearchDispatcher* to consume.
- For every *Stream* in the model, the generator produces a concrete class which contains code that subscribes instances of the concrete task classes to the underlying ActiveMQ topics/queues [12]. Unlike with tasks, sources and sinks, developers do not need to write additional code to specify the behavior of streams.
- For every *Type* in the model the generator produces a concrete class which contains properties, setters and getters for the type's fields. If there is at least one stream typed after the type in question, the generated class extends the built-in *Job* class, which provides JobID/correlation ID fields as well as serialization capabilities which are required for caching as discussed below.

The generator also produces an entry (main) Java/Python class named after the workflow instance in the model, which starts and coordinates the execution of the workflow on a node, supporting command-line parameters through which users can specify:

- Whether the node runs in *master* or *worker* mode. Each workflow execution can be coordinated by one master node. In the master mode, additional command line parameters specify whether the workflow needs to start an embedded (ActiveMQ) messaging broker that will manage the message channels of the workflow, or can use an existing one at a specified DNS/IP address.
- For nodes in worker mode, relevant parameters can define the DNS/IP address where the ActiveMQ broker instance is running. An additional parameter (*-exclude*) can be used to exclude particular tasks from their execution by the worker. For example, a worker may exclude *GitHubCodeSearcher* from its execution if it doesn't have GitHub credentials, and contribute to the workflow through cloning and searching repositories by means of the *RepositorySearcher* task.

```
@Override
public void consumeTechnologies(Technology t) throws Exception {
  List<String> paths = ...;  // runs GitHub search
  for (String path : paths) {
    Repository r = new Repository();
    r.path = path;
    r.correlationId = t.jobId;
    sendToRepositories(r); } }
```

Listing 1: The consumeTechnologies(...) hand-written method of GitHubCodeSearcher in Java

```
def consumeTechnologies(self, t:Technology):
  paths = ... # runs GitHub search
  for p in paths:
    r = Repository()
    r.path = p
    r.correlation_id = t.job_id
    sendToRepositories(r)
```

Listing 2: The consumeTechnologies(...) hand-written method of GitHubCodeSearcher in Python

Listing 3 shows bash commands for starting the master node and three worker nodes of the bundled implementation of the example above in four computers with the monikers pc1 . . . pc4. Executing these commands leads to the runtime distribution illustrated in Figure 5. Note how there is only one instance of *RuntimeSearchDispatcher* at runtime (running on pc1 alongside the source and the sink of the workflow) due to the *masterOnly* flag in the model discussed above. Also, while instances of *RepositorySearcher* run on all three workers, only two instances of *GitHubCodeSearcher* in Java run, due to the excludes flag in line 17 of the bash command that starts the node on pc4 as well as the one in Python.

```
1   pc1: java -jar techrank.jar -mode master_bare
2      -instance technologyanalysis
3      -in technologies.csv -out results.csv
4   pc2: java -jar techrank.jar -mode worker
5      -instance technologyanalysis
6      -brokerHost pc1.acme.com
7   pc3: java -jar techrank.jar -mode worker
8      -instance technologyanalysis
9      -brokerHost pc1.acme.com
10  pc4: java -jar techrank.jar -mode worker
11     -instance technologyanalysis
12     -brokerHost pc1.acme.com
13     -exclude GitHubCodeSearcher
14  pc4: python3.8 main.py -mode worker
15     -instance technologyanalysis
16     -brokerHost pc1.acme.com
```

Listing 3: Running a master and three worker nodes through the command line

To facilitate clean separation between generated and hand-written code, the Crossflow generator uses different directories for these, which can be specified by the user. By default, Java code is placed in a *src* and a *src-gen* directory. Code under *src-gen* is meant to be exclusively generated and can be overwritten by the generator in subsequent invocations, while files under *src*, containing the implementation of the behavior of sources, tasks and sinks, are never overwritten and once generated they need to be maintained manually. Similarly by default Python code is located within the *py* and *py-gen* directories.

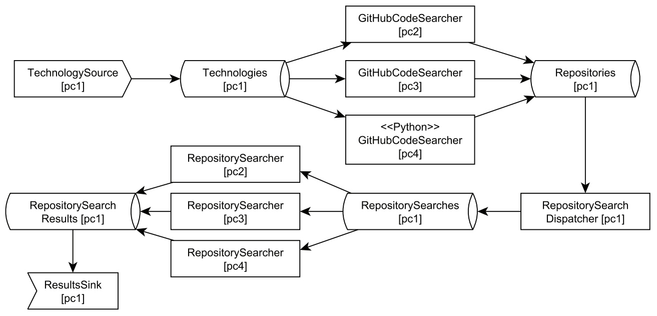


Figure 5: Runtime Component Distribution

It is worth noting that Crossflow can be used as a Java/Python library, without leveraging any of its code generation capabilities, by building code directly on top of the core runtime classes.

1.4 Error Handling, Logging and Monitoring

Exceptions produced during the execution of hand-written code in *consumeXYZ(...)*/*produce(...)* methods[6], such as *consumeTechnologies(...)*, in workers are caught by the generated base classes and sent to a dedicated stream (*FailedJobsTopic*) together with the job that caused them. Temporary loss of network connectivity between the master and the worker nodes is handled through the message persistence and wait-and-retry capabilities of the supporting (ActiveMQ) messaging middleware. For logging and monitoring, there are various dedicated Streams provided by Crossflow:

- The LogTopic contains any log messages added by users to a Task, using the Crossflow log API, such as: log(LogLevel.INFO, "message");
- The TaskStatusTopic contains a continuous stream of Task statuses, such as a task being in progress (evaluating a job) or waiting for a job to arrive. Users can add further statuses to a Task in their implementation, by using the Crossflow task status API, such as: getWorkflow().setTaskBlocked(this, "this task is blocked");
- The StreamMetadataTopic and TaskMetadataTopic provide continuous monitoring information about streams (like the current number of messages in each stream or the number of consumers it has) and tasks (like when a task is idle)

1.5 Caching

Jobs performed in the context of a repository mining workflow can require fetching large volumes of remote data (e.g. cloning Git repositories) or making calls to rate-limited APIs. To avoid repeated execution of such jobs, Crossflow provides built-in support for job-level caching. In Crossflow, each job has a unique (auto-generated or manually set) JobID and an optional correlation ID, recording the JobID of the job of which it is an output. For example, Listing 1 shows a redacted version of the implementation of the *consumeTechnologies(...)* method of *GitHubCodeSearcher*, where outgoing *Repository* jobs are associated to the incoming *Technology* by setting the correlation ID of the former to the JobID of the latter (line 7).

[6]The generated signatures of such methods allow exceptions to be thrown during their execution.

The master node intercepts jobs submitted to all streams and caches outputs against their respective inputs (based on JobIDs and correlation IDs). Hence, previously-seen jobs are not re-executed in subsequent runs of the workflow, but instead their cached outputs are reused. This will only happen for jobs that define such a correlation ID and only if the cache is enabled for that workflow.

The default cache implementation in Crossflow is file based and is organized as follows.

- The cache for a workflow execution is stored in a file-system directory;
- For each stream in the workflow, there is a directory named after the stream under the root cache directory;
- For every job submitted to the stream, a nested directory is created with a unique name (UUID) derived by hashing the values of the fields of the job;
- Every output correlated to the job through its correlation ID field is stored in an XML file under this directory, named with the content-derived UUID of the output job.

An example directory structure for the workflow of Figure 3 appears in Figure 6.

```
cache
|__Technologies
|   |__<UUID of input job A> (folder)
|   |   |__<UUID of output job B> (file)
|   |   |__<UUID of output job C> (file)
|   |   |__...
|   |__<UUID of input job D>
|       |__...
|__Repositories
|   |__...
|__RepositorySearches
|   |__...
|__RepositorySearchResults
    |__...
```

Figure 6: Example Cache Directory Structure

The main rationale behind a filesystem-based cache implementation (as opposed to e.g. a database-backed implementation) is that it simplifies backing up or discarding parts of the cache. For example, if after the first execution of the workflow a defect is detected in the part of the *RepositorySearcher* task that searches within cloned repositories, the developer (or an API call to Cache.clear(String streamName)) can delete the *RepositorySearches* folder of the cache and run the workflow again. In the new execution all other cached results will be reused (as well as the previously cloned repositories - see section on locality scheduling below) and only the code that searches for files within the cloned repositories will need to be re-executed.

1.6 Locality Scheduling and Worker Capabilities

The first time the example workflow is executed in a distributed setup, different worker nodes will end up with different cloned Git repositories as a result of the execution of their *RepositorySearcher* tasks. The next time the workflow is executed (e.g. after a bug fix or after adding more technologies to the input csv), *RepositorySearch* jobs should ideally be routed to nodes that already have clones of relevant repositories from the previous execution to avoid unnecessary cloning of the same repositories in different nodes.

To achieve this, we originally considered delegating the required book-keeping to the master node. In this approach, the master node would be responsible for "remembering" how Git repositories (and other expensive to re-fetch/compute resources) were distributed between workers. However, given that workers can appear/disappear at any point during the execution of the workflow, we opted for a simpler and more flexible approach, which eliminates the need for centralized book-keeping by enabling CROSSFLOW worker tasks to reject jobs allocated to them. This feature (an OpinionatedTask in the CROSSFLOW model) is one which delegates the Job acceptance logic to the developer. In this case, the Task contains a method *acceptJob(Job job)* whereby each incoming job can be rejected before it is carried out by the task (and consequently returned to its originating stream to be re-dispatched), by evaluating the boolean condition defined in that method. Using this structure, the developer has total control over which Jobs are accepted by that Task; for example implementing the acceptance logic for *RepositorySearch* is as follows:

- *RepositorySearcher* receives a *Repository* to analyze
- If the worker has a clone of the repository in question, it accepts (*acceptInput(Repository repo)* returns true) and performs the job
- If it does not have a clone of the repository:
  - The first time the worker encounters this job it adds the JobID of the job to a list of encountered jobs and rejects the job (keeping track of how many times this has already happened)
  - If the JobID of the job is already in the worker's encountered job list upon reception, it assumes that all other nodes have also rejected the job, and accepts it

The main advantages of this approach is that it eliminates the need for book-keeping at the master node and that it allows worker nodes to dynamically reject jobs, which is useful in several scenarios (e.g. when a worker runs out of GitHub API calls or out of space in its local filesystem). On the flip side, it incurs a runtime overhead as in the first execution of the workflow above, all *RepositorySearch* jobs will be rejected N times (i.e. sent back to the master node) by each worker before they start getting accepted. This can become an issue in cases where job messages carry a lot of data so developers of CROSSFLOW programs are encouraged to keep such messages small and provide pointers to larger data as opposed to embedding it when possible (e.g. the path of a file on GitHub as opposed to its contents). In terms of fair allocation of work across the workers, this is delegated to the respective facilities of the ActiveMQ messaging middleware (round-robin message distribution). Preliminary experiments have provided no evidence of unfair allocation but this is an area for additional investigation.

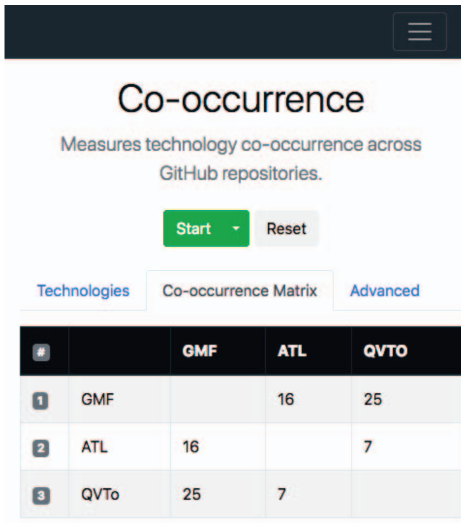
Finally, the part of the logic above pertaining to rejecting a job unless it has already been encountered by the worker already has been abstracted into a pre-defined type of Task: a CommitmentTask, in the CROSSFLOW model. In this abstraction, a job is rejected by the Task unless it has already been encountered N times already. As this is a probabilistic approach, increasing N allows for a higher probability that the correct worker is chosen, but will incur a higher network overhead.

1.7 Web-based Admin Interface

To simplify the deployment and execution of CROSSFLOW workflows, we have developed a basic supporting web application. The application provides a root directory ("experiments"), under which CROSSFLOW workflows can be deployed. Each CROSSFLOW experiment[7] lives under its own directory, which contains an *experiment.xml* descriptor that provides the following configuration parameters:

- A human-readable title and short summary for the workflow
- The fully qualified name of the main class of the workflow
- The name of the runnable JAR of the workflow
- The paths of input/output CSV files that the web app should display as HTML tables
- A description of how remote workers can contribute to the execution of the workflow (i.e. by downloading and running the workflow's JAR file locally with appropriate parameters)

The web application allows users to browse available experiments, to start a master node for a selected experiment, to view the contents of its declared input and output CSV files, to stop the workflow execution, and to reset its cache. A screenshot of the application with the running example can be seen in Figure 7.



#		GMF	ATL	QVTO
1	GMF		16	25
2	ATL	16		7
3	QVTo	25	7	

Figure 7: Co-occurrence Matrix (Output) Tab

2 RELATED WORK

PyDriller [13] is a Python framework for extracting information from Git repositories such as source code and repository metadata. It provides a high-level API that can be used to directly interact with a repository, and tools for manipulating the data retrieved (e.g. code diffs). Other libraries for interacting with individual Git repositories include JGit (eclipse.org/jgit) for Java and PyGit (pygit2.org) for Python. CROSSFLOW is complementary to these tools and developers

[7] An "experiment" is an instance of the workflow with specific input data.

can make use of them to implement Git-related functionality in Java/Python tasks of Crossflow mining workflows.

There is also a substantial body of work on tools for large scale analysis on software repositories. The first tool of this kind was Alitheia Core [8], which provided a service-based architecture for distributed analysis of software repositories. In comparison to Crossflow, Alitheia Core has a predefined processing workflow and, while extensible, it was not built with the intention to support arbitrary repository mining applications. Moreover, it does not provide some of the more advanced features of Crossflow, such as opinionated workers or job caching.

Boa [4] is a domain-specific language and infrastructure for mining software repositories. Boa's infrastructure leverages distributed computing techniques to execute domain-specific queries against collections of Git repositories. However, Boa does not allow the specification of more complex workflows that require information from non-Git sources (e.g. a bug tracker or the StackOverflow API).

SmartSHARK [15] presents the implementation of an approach that is focused on tackling issues related to the reproduction of software repository mining studies. The implementation is composed of a platform with a series of plugins such as *vcsSHARK* and *mecoSHARK* for obtaining historical information from repository clones and computing revision-level metrics, respectively. Similar to Boa, this approach is designed to pre-compute and store any data that may become relevant during the stage of analysis job submission. Figure 8 summarizes the use of smartSHARK. The *SmartSHARK platform* is represented by a web server running a user interface (UI) which developers may access to install plugins and provide *software project* repositories. Next, plugins such as vcsSHARK and mecoSHARK are responsible for preprocessing provided repositories (i.e. by employing external *code analysis frameworks* such as Source Meter [5]) and inserting precomputed analytics into a *MongoDB* database that is shared among developers, the SmartSHARK platform, and a *Hadoop cluster*. Finally, developers implement *analytics programs* and submit them either directly or through a web form to a Hadoop cluster for distributed job execution. Results produced by a Hadoop job may be stored in the shared database from which developers may access them.

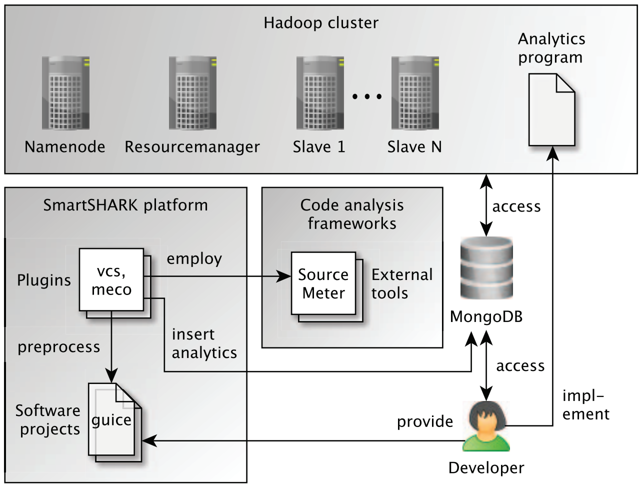


Figure 8: Overview of SmartSHARK

Another tool for distributed mining of software repositories is King Arthur[8], which is part of the GrimoireLab[9] tool chain. King Arthur is a distributed job queue platform that schedules and executes data retrieval jobs from software repositories using Perceval [3], a dedicated Python library. This platform enables the orchestration and distribution of data retrieval jobs only, while Crossflow enables the distribution of entire (data retrieval + analysis) mining workflows. Moreover, Crossflow workers can selectively choose jobs to undertake depending on their capabilities. This is not the case with King Arthur workers, which simply pick the next job from a queue whenever they are idle.

General-purpose distributed stream processing frameworks such as Apache Spark [16] and Apache Flink [2] can be used for mining software repositories in a distributed manner, however, unlike Crossflow, they do not provide support for polyglot/opinionated workers and for job-level caching.

Finally, Boinc [1] is a software system that facilitates the creation and execution of public-resource computing projects and hence it could be used to support the execution of mining workflows. Boinc shares many similarities with Crossflow. Namely, it supports distributed computation, workers are assigned jobs based on their computational capabilities, and locality scheduling is used. At the same time though, Crossflow offers particular features that make it more suitable for repository mining workflows. First, although Boinc supports selective execution of jobs from workers, it is the server, which decides on the distribution of jobs based on their estimate of computational requirements. On the other hand, in Crossflow workers choose their jobs as the master node is completely unaware of the exact composition of the system. This results to increased robustness to specific faults, such as as network and time-out errors. Moreover, Boinc does not provide a high-level, declarative way to specify workflows.

3 EVALUATION

The motivation of the evaluation presented in this section is twofold. First, a qualitative evaluation is presented, analyzing Crossflow and in particular focusing on the capability to express a state-of-the-art open source software project effort estimation scenario that has been employed by the software repository mining approach SmartSHARK [15]. Secondly, a quantitative performance and resource analysis of Crossflow during the execution of said effort estimation scenario is presented. The employed scenario has originally been introduced by Gousios et al. [7] and Robles et al. [11] and estimates developer contributions in months based on factors such as lines of code (LOC) which have been committed to publicly available software project repositories. The amount of person months invested in an open source project are computed based on an automated approach that takes into account the number of commits and the number of days a developer has actively contributed to a project repository. More specifically, the former and latter represent the number of changes to source code per developer (i.e. acting as proxy of the amount of activity per person) and the number of days with commits per person (i.e. acting as proxy of the time periods when a person is actively developing software), respectively.

[8]https://github.com/chaoss/grimoirelab-kingarthur
[9]https://chaoss.github.io/grimoirelab/

3.1 Evaluation Scenario

This section describes the procedure associated with establishing effort estimates for a set of open source software projects by employing Crossflow. The execution and monitoring of the tool is performed independently (i.e. on separately provisioned cloud computing nodes) and produces quantitative results and in particular performance and resource usage statistics.

The result of the effort estimation analysis is a list of metrics for each open source software project including LOC$_{\text{added}}$, LOC$_{\text{deleted}}$, number of commits, number of developers, project duration, and LOC. These metrics can used to establish a prediction of effort required for the development of new projects and in particular their time to release in months. In other words, the estimated time to release $t = \frac{x}{devs \cdot C \cdot L}$ where x is the estimated size of a project release, $devs$ the number of developers, C the mean number of commits per developer per month, and L the mean number of LOC.

Figure 9 depicts a Crossflow model capturing the effort estimation workflow. The execution of master node and individual worker nodes is initiated similarly as described in Listing 3. The initial Java task *ProjectSource* is executed by the master node and creates instances of *Project* by parsing lines of the workflow input CSV file that are composed of Github repository owner, repository name, and commit hash value (i.e. acting as head revision). Next, fitting workers pick up the execution of opinionated Java task *RepositoryCloner* which entails facilitating the information of *Project* instances to establish local repository clones as well as creating instances of *Repository* to keep track of local and remote repository location and commit hash value. Then, workers execute the opinionated Java task *JavaRepositoryAnalyzer* similarly and in particular by creating instances of *JavaRepositoryAnalysisResult* (i.e. extending instances of *Repository* produced by task *RepositoryCloner*) holding the computed total size and number of files of the repository clone based on the commit hash value. Afterward, instances of *JavaRepositoryAnalysisResult* (i.e. created by the previously described task) are picked by workers running the opinionated Python task *PythonRepositoryAnalyzer*. More specifically, the Python library PyDriller [13] is employed for iterating through each repository commit and in particular to count the total number of lines, commits, and developers; sum up the the number of lines added and deleted; as well as compute estimates on the duration of a project in months. Finally, the Java task *RepositoryAnalysisResultSink* is executed by the master node and creates the workflow output CSV file by serializing instances of *PythonRepositoryAnalysisResult* (i.e. created by the previously described task).

3.2 Experimental Setup

This section outlines the experimental setup of Crossflow for running the above-mentioned open source software project effort estimation scenario. The subjects of study are represented by a set of 21 publicly available open source software projects (i.e. the set employed by Trautsch et al. [14]).

Table 3 depicts a number of node configurations offered by a cloud provider and employed for the execution of the effort estimation workflow presented in Figure 9. More specifically, node configurations are defined by *name*, number of CPU *cores* (i.e. of

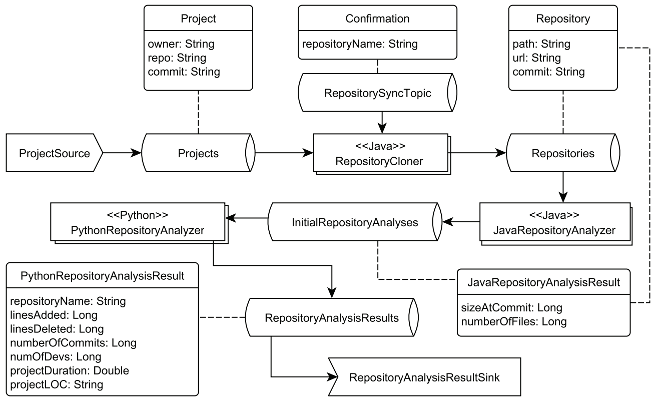


Figure 9: Effort estimation workflow in Crossflow

type Intel Xeon Gold 6140 2.30GHz), amount of *memory* in GB, number of *instances* (i.e. individual computing nodes) running Ubuntu 18.04, and *parallelization* count. Further, deployment configurations that are distributed over several instances employ a single instance that runs a bare master node (i.e. a master node that only performs tasks marked as "master only") alongside an internal ActiveMQ broker and non-distributed deployment configurations employ a normal master node (i.e. a master node that also acts as a worker) and an external ActiveMQ broker.

Name	Core(s)	Memory	Instance(s)	Parallel
Deployment$_{\text{c1m1}}$	1	1	1	1
Deployment$_{\text{c1m1}}$d	1	1	5	1
Deployment$_{\text{c1m2}}$d	1	2	5	1
Deployment$_{\text{c1m2}}$	1	2	1	1
Deployment$_{\text{c2m2}}$	2	2	1	1
Deployment$_{\text{c2m2}}$p	2	2	1	2
Deployment$_{\text{c4m8}}$	4	8	1	1
Deployment$_{\text{c4m8}}$p	4	8	1	4

Table 3: Cloud provider node configurations

3.3 SmartSHARK

In [15], the authors present how this evaluation scenario is executed in SmartSHARK:

(1) The user adds the repositories they wish to analyze to SmartSHARK, either through the ServerSHARK web UI, or directly in the MongoDB SmartSHARK uses to store its results.
(2) The user runs vcsSHARK, which will retrieve these projects and perform various pre-defined analyses including obtaining LOC$_{\text{added}}$ and LOC$_{\text{removed}}$ for each commit.
(3) The user runs mecoSHARK, which will perform further analysis on these repositories, such as obtaining the total LOC of the project.
(4) The user runs a custom analysis program in Spark, on the MongoDB (that now contains all relevant data required), in order to synthesize the variables needed for the effort estimation calculation and to perform it for each project.

Note that in this process, only the last step can be performed in a distributed manner, as there is no support for executing the remaining steps (or the external tools they run) in a similar fashion.

3.4 Qualitative results

The output produced by the modeled effort estimation workflow is depicted in Table 5. In total, 18,164 commits have been processed as produced by 538 individual developers within the estimated project duration period of 694 months, and 14,530,653,071 and 592,861,925 LOC have been added and deleted during the observed period, respectively. Finally, a total of 16,877,171 LOC have been counted in the final version of the set of 21 analyzed projects. Thus, in comparison to smartSHARK, identical information about the examined projects (cf. Table 3 in [15]) has been generated by Crossflow.

Moreover, in comparison to Crossflow, the following qualitative observations have been made: Firstly, smartSHARK requires users to perform a series of manual steps, such as the addition of projects to be analyzed as well as cloning their respective repositories; in Crossflow this is done through reading a CSV source and defining a cloning task, respectively. Secondly, smartSHARK relies on a shared database and the concept of precomputing a pre-defined set of information, such as repository metrics, which may be used during the execution of an analysis job. This choice of architecture is similar to that of Boa [4] and focuses on scenarios that envision the analysis of project repositories for which preprocessing has already occurred at time of analysis job submission. Although similar behavior may be implemented by Crossflow, the computation of a pre-defined set of information is not enforced thus enabling users only to perform the computation of metrics that are relevant to the analysis at hand. As a result, a lower use of resources, such as computing power, storage space, and execution time, can be achieved. Thirdly, users of smartSHARK define analysis jobs by the use of native APIs of frameworks such as Apache Spark, hence being provided very limited abstraction from distributed execution concepts. Finally, we note that the smartSHARK plugin mecoSHARK employs Source Meter [5] as an external source code analysis tool capable of analyzing code written in a set of supported programming languages.

3.5 Quantitative results

The performance of running the described workflow on various node configurations is depicted in Table 4. Note that the amount of memory used by ActiveMQ is excluded by distributed and included by non-distributed deployments. The execution of deployment$_{c1m1}$ failed to complete successfully due to the combination of a low amount of memory and the use of an internal ActiveMQ broker running on the same node. The execution of deployment$_{c1m2}$ succeeded and shows that increasing the memory from one to two GB is sufficient to execute the workflow on a single machine also acting as an ActiveMQ broker, in approximately three quarters of an hour. In comparison, deployment$_{c2m2}$p shows that parallelizing the workflow execution over two cores reduces execution time by approximately 14 percent. Deployment$_{c2m2}$ and deployment$_{c4m8}$ illustrate the overhead of executing the workflow *without* the use of the Crossflow workflow parallelization capability on two and four cores, respectively. The use of parallelization on a deployment with a number of four cores (i.e. in case of deployment$_{c4m8}$p) reduces the execution time to a total of approximately 28 minutes and shows that the effective overhead of parallelization is reduced (i.e. resulting in an execution time decrease of approximately 37% when compared with deployment$_{c1m2}$). Deployment$_{c1m1}$d illustrates that the execution of the workflow succeeds with the same low-amount of resources as employed in deployment$_{c1m1}$ however with one node acting as bare master and ActiveMQ broker and four nodes as workers as opposed to a single node acting as master and running an ActiveMQ broker. The distributed configurations deployment$_{c1m1}$d and deployment$_{c1m2}$d (i.e. increasing the amount of memory by one GB) complete the described workflow in approximately 30 and 27 minutes, respectively.

Figure 10 illustrates that the mean CPU use for the execution of the workflow on all the deployment configurations is dominated by Python, in particular by the task *PythonRepositoryAnalyzer*, which employs the Python library PyDriller [13] to iterate through repository commits and create instances of *PythonRepositoryAnalysisResult* (i.e. requiring to compute values for LOC$_{added}$, LOC$_{deleted}$, numOfDevs, projectDuration, and projectLOC).

Configuration	Duration	Total (MEAN) Memory use	Total (MAX) Memory use
Deployment$_{c1m1}$	failed	failed	failed
Deployment$_{c1m1}$d	00:29:49	1021	1211
Deployment$_{c1m2}$	00:44:46	300	412
Deployment$_{c1m2}$d	0:27:15	1098	1400
Deployment$_{c2m2}$	1:13:04	396	614
Deployment$_{c2m2}$p	0:38:23	589	732
Deployment$_{c4m8}$	1:23:57	858	1368
Deployment$_{c4m8}$p	0:28:03	1300	1424

Table 4: Crossflow execution in hours and Megabytes

Figure 11 shows the percentage of maximum memory use by Java and Python as well as the percentage of remaining memory which may be consumed by Crossflow or other processes running on that computing node. It shows that this use-case does not require a large amount of memory to run, even when provided with a surplus.

3.6 Threats to Validity

The results reported in Table 4 have been extracted through a single execution of the different configurations. While the execution time and memory footprint measurements are consistent with our observations over multiple executions of similar mining workflows and deployment configurations, this needs to be highlighted as a threat to the validity of the results of this experiment.

4 CONCLUSIONS

This paper presented Crossflow, a novel framework for development and distributed execution of multi-step repository mining programs. Crossflow provides a domain-specific language for designing polyglot distributed workflows as well as a code-generator that produces implementation scaffolding for developers to complement with hand-written code. Crossflow uses asynchronous message-based communication and provides built-in support for job-level caching and locality scheduling. Preliminary evaluation shows promising results with regards to the scaling of Crossflow in both parallel and distributed execution. Beyond more systematic and larger-scale evaluation, future work includes offering explicit

Project	$\text{LOC}_{\text{added}}$	$\text{LOC}_{\text{deleted}}$	Commits	Developers	Duration	LOC
cursynth	74 725 950	443 694	219	8	22	121464
cxxnet	720 994 114	1 949 969	852	36	17	303276
elasticsearch-hadoop	12 192 988	4 389 727	1243	8	29	528719
fatal	40 596 648	8 917 237	401	5	11	324743
guice	1 970 605 371	365 021 777	1441	36	106	7688021
HackerNews	136 843	14 847	12	2	1	5202
k3b	1 970 605 371	365 021 777	1441	36	106	7688021
ksudoku	231 941 927	94 591 343	668	69	81	512211
libxcam	9 743 146	1 444 773	250	12	7	190432
libyami	40 122 874	15 399 910	487	26	24	506490
log4j	92 438 210	38 423 498	3266	21	138	2492060
mxnet	6 023 262	1 711 834	223	13	3	99410
oclint	3 616 606	1 848 673	733	25	32	166164
ohmu	60 807 833	8 576 687	226	8	16	354109
openage	27 051 105	10 191 251	1761	62	23	684688
osquery	10 865 948 515	15 984 362	2208	70	12	854425
passivedns	3 338 015	1 288 034	220	15	45	132595
SMSSync	104 497 422	9 638 622	1395	28	61	823391
swift	15 601 967	7 424 690	496	22	38	412470
wds	3 150 807	778 034	238	11	10	107643
xgboost	247 119 468	4 822 963	1825	61	18	569658
$\sum$	14 530 653 071	592 861 925	18 164	538	694	16877171

Table 5: Output produced by effort estimation workflow

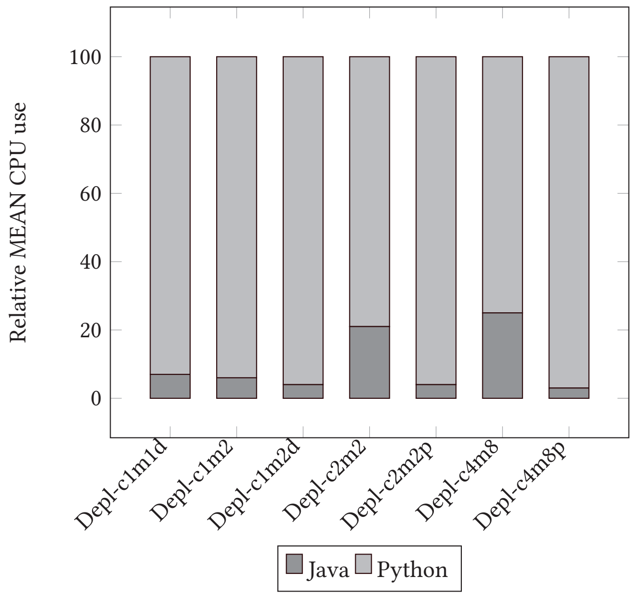


Figure 10: Relative MEAN CPU for each experimental setup

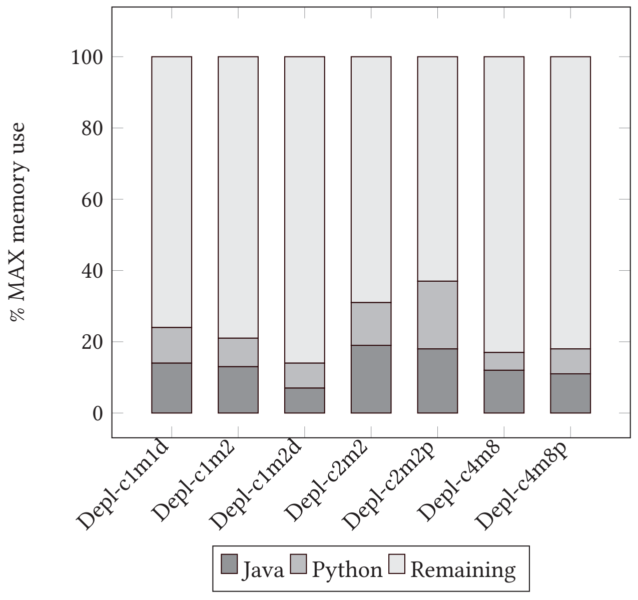


Figure 11: % MAX memory for each experimental setup

traceability for jobs throughout their path in a workflow, improving upon task scheduling by monitoring resource use of each worker node and re-allocating tasks based on current surpluses present in each, as well as further improving usability by adding further functionality through the Web UI such as spawning worker nodes.

ACKNOWLEDGMENTS

The work in this paper was supported by the European Commission via the CROSSMINER (732223) and TYPHON (78025) projects, and by the EPSRC via the MANATEE Project (EP/S000143/1).

REFERENCES

[1] David P Anderson. 2004. Boinc: A system for public-resource computing and storage. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (2004), 4–10.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[3] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2018. Perceval: software project data at your will. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), 1–4.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. *Proceedings of the 35th International Conference on Software Engineering* (2013), 422–431.

[5] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota. 2014. Source Meter Sonar Qube Plug-in. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (Sep. 2014), 77–82.

[6] Frédéric Jouault and Ivan Kurtev. 2005. Transforming Models with the ATL. *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005* 3844 (October 2005), 128–138.

[7] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. 2008. Measuring developer contribution from software repository data. *Proceedings of the 5th International Conference on Mining Software Repositories* (2008), 129–132.

[8] Georgios Gousios and Diomidis Spinellis. 2009. Alitheia core: An extensible software quality monitoring platform. *Proceedings of the IEEE 31st International Conference on Software Engineering* (2009), 579–582.

[9] Dimitris Kolovos, Patrick Neubauer, Konstantinos Barmpis, Nicholas Matragkas, and Richard Paige. 2019. Crossflow: A Framework for Distributed Mining of Software Repositories. *Proceedings of the 16th International Conference on Mining Software Repositories* (2019), 155–159. https://doi.org/10.1109/MSR.2019.00032

[10] Dimitrios S Kolovos, Nicholas Drivalos Matragkas, Ioannis Korkontzelos, Sophia Ananiadou, and Richard F Paige. 2015. Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects. *OSS4MDE@ MoDELS* (2015), 20–29.

[11] Gregorio Robles, Jesús M. González-Barahona, Carlos Cervigón, Andrea Capiluppi, and Daniel Izquierdo-Cortazar. 2014. Estimating development effort in Free/Open source software projects by mining software repositories: a case study of OpenStack. *Proceedings of the 11th International Conference on Mining Software Repositories* (2014), 222–231.

[12] Bruce Snyder, Dejan Bosnanac, and Rob Davies. 2011. *ActiveMQ in action*. Vol. 47. Manning Greenwich Conn.

[13] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), 908–911.

[14] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. 2016. Adressing problems with external validity of repository mining studies through a smart data platform. *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), 97–108.

[15] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. 2018. Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering* 23, 2 (2018), 1036–1083.

[16] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.