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

Project Number 732223

D5.6 Parallel and Distributed Workflow Execution - Final Version

Version 1.0
28 June 2019
Final

Public Distribution

University of York

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

PROJECT PARTNER CONTACT INFORMATION

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

DOCUMENT CONTROL

Version	Status	Date
0.5	Initial Version	17 June 2019
0.8	Internal Review Version	24 June 2019
0.9	Revised Version from Reviews	26 June 2019
1.0	Final Version	28 June 2019

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This document represents the final version of deliverable 5.6, i.e., the Parallel and Distributed Workflow Execution Engine component of Crossminer (from now on referred to as the *Crossflow* platform); its purpose is to provide an overview of this system and its capabilities, offering a complete picture of the functionality provided. It focuses on the execution of Crossflow workflows (D5.2), i.e., user-defined knowledge extraction workflow specifications created using the workflow development tools (D5.5), in a parallel and distributed manner. Within this deliverable, the focus is on detailing the various technologies used to achieve this result as well as describing how they interoperate. The document is structured as follows. First, an overview of the system is provided, alongside a running example used to provide a narrative use-case of the system throughout this report. Secondly, an updated version of the workflow metamodel (language) is presented, focusing on why these changes were necessary for achieving the additional functionality required by the workflow engine, when compared to the previous version (D5.2). Finally, the various technologies used by the system are presented, detailing their roles and how they interoperate for producing a cohesive platform for the creation, manipulation and execution of workflows.

1. INTRODUCTION

The delivery of an integrated platform for the development of complex software systems that offers monitoring, in-depth analysis and evidence-based selection of OSS components, can require knowledge extraction and analysis from large OSS repositories. For example, before deciding which projects need to be analysed by the Crossminer platform, users can identify a list of potential projects of interest by means of a distributed workflow. Hence, the development of a system that is both capable of describing generic task descriptions as well as efficiently processing such tasks is necessary.

The goal of this deliverable is to report on the final version of the execution engine that supports the parallel and distributed execution of workflows. This execution engine enables engineers to run *Crossflow* workflows both locally, i.e., parallelising execution over multiple cores, and in high-performance computing infrastructures, i.e., parallelising execution over multiple network-connected nodes.

Figure 1 depicts the integration of Crossflow in the Crossminer architecture. On one hand, workflow specifications, i.e. Crossflow models created by the employing workflow development tools described in Sections 3-4 of D5.5, can be executed by metric providers by instantiating the Crossflow Java API, detailed in Appendix A. On the other hand, Crossflow workflow tasks can retrieve data from the Crossminer knowledge base REST API and use it accordingly.

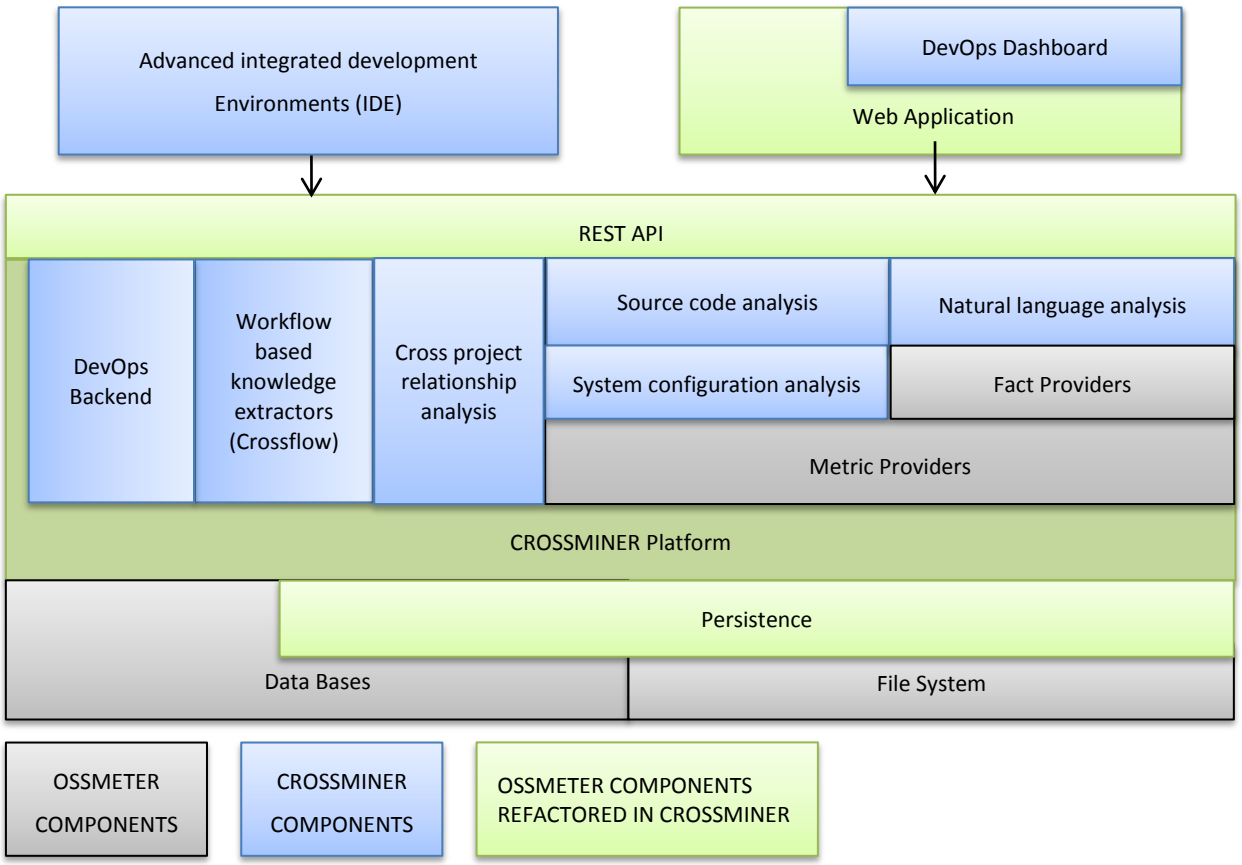

Figure 1: Integration of Crossflow in Crossminer architecture.

This work has been published in the Mining Software Repositories (MSR 2019) conference (Appendix B.2).

2. MOTIVATION

In order to motivate the need for such a platform, we will use the following scenario, a simplified overview of which is seen below:

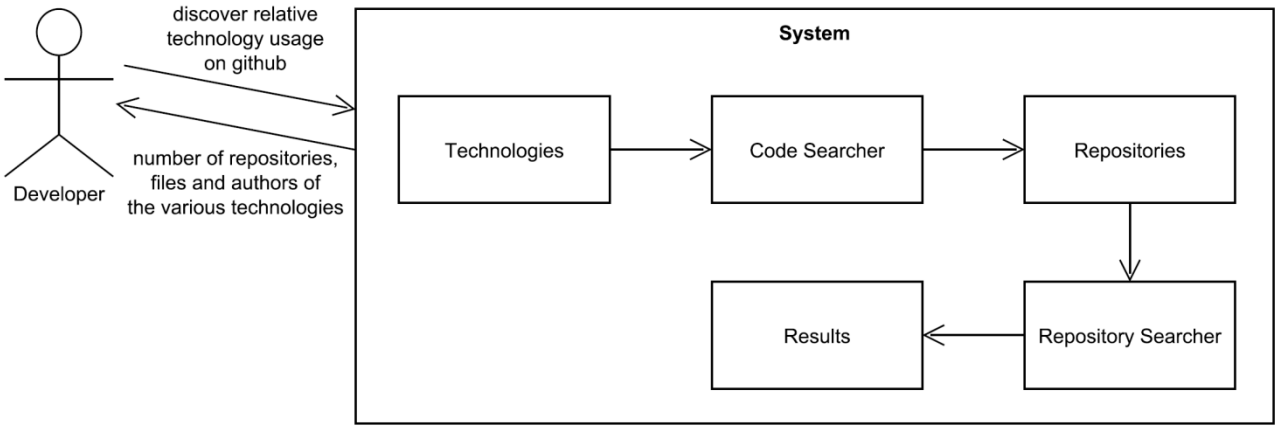

Figure 2: Simplified motivating scenario

In this scenario, a developer wishes to find out how popular various technologies are on GitHub, in order to decide which of them to start monitoring using the Crossminer platform. More specifically, they are interested in how many files of that technology are found in repositories containing at least one such file. A file is said to be of a specific technology if it has a specific extension (such as *.java* for Java files) as well as containing a specific keyword in its contents (such as the keyword *class* for Java files).

In order to perform this analysis, the developer would have to perform the following steps:

1. Search GitHub (using its public-repo API) for all files fitting the abovementioned conditions
2. For each file retrieve the repository it is contained in
3. For each repository perform the necessary analysis for finding out the number of files of each technology of interest, as well as other data such as the number of authors contributing to relevant files in that repository, number of commits etc.

This process can be done using a custom program but will quickly reveal a variety of challenges, such as those faced by [1]. Not only is such a program difficult to create, it is also specific to the needs of this single scenario: it is bound to the GitHub API as well as to performing analysis on GitHub repositories. Furthermore, executing such a process will likely require a substantial amount of resources in order to process the vast amount of data that will be retrieved from GitHub, something that is well suited for parallel and distributed execution, further increasing the complexity of the process.

As such, a general-purpose system for the creation of knowledge extraction workflows would be of benefit to the developer, who can then focus on their core mining/analysis logic and delegate the infrastructure and distribution concerns to the platform.

2.1 TECHNOLOGY CHANGES

The final version of this platform uses a different set of libraries for its execution when compared to D5.2. The reasoning for this change is motivated below.

Early research into distributed execution engines like Apache Spark [2], Storm [3], Samza [4] and Flink [5], as well as distributed logging services like Apache Kafka [6] revealed that they are not well-suited for use by such a general-purpose workflow engine. More specifically, the following shortcomings were discovered:

- Such tools do not have worker-specific capabilities as a primary concern (each worker is a homogeneous process in the workflow)
    - Workers may want to reject some of the tasks entirely (only perform computations for a specific subset of tasks in the workflow)
    - Workers may want to reject some of the input (of a specific task) given to them (only perform computations for input they are optimised for), and return the rest to be processed by some other worker
- They do not consider caching of intermediate results
- They do not offer an easy way to create tasks in multiple programing languages, for a single workflow

As such, the platform uses a simple queueing framework: Apache ActiveMQ for communicating between its various components. This allows for fine-grained control over all aspects of the system, alleviating the constraints imposed by the other higher-level technologies. An in-depth look at how the platform achieves this can be found in Sections 6 and 7.

3. CROSFLOW ARCHITECTURE

This section contains the architecture of the final version of Crossflow: a general-purpose, language-agnostic distributed workflow execution platform. The section focuses on presenting an overview of the system, leaving any in-depth details and analysis to the remainder of the document.

Figure 3 shows the high-level architecture of Crossflow:

- The Crossflow *metamodel* (originally presented in D5.2, an updated version of which can be found in Section 0) is used to create Crossflow *models* through a graphical editor (details can be found in D5.5). Such models define the structure of the workflow as well as providing the necessary information for the creation of the generated code.
- The Crossflow *code generator* uses this model, as well as the Java code provided by the Crossflow *runtime* to create:
    - Orchestration code for joining together the various tasks of the workflow so that they can be executed in a parallel and distributed manner.
    - Base classes for each workflow task, which need to be extended by the developer

- Implementation classes (stubs) that the developer fills-in with their execution logic
- Packaging of the code to be executed on one or more machines (further details on this can be found in D5.5)

- The packaged code can then be executed either programmatically or using the Crossflow *web UI* (detailed in D5.5)

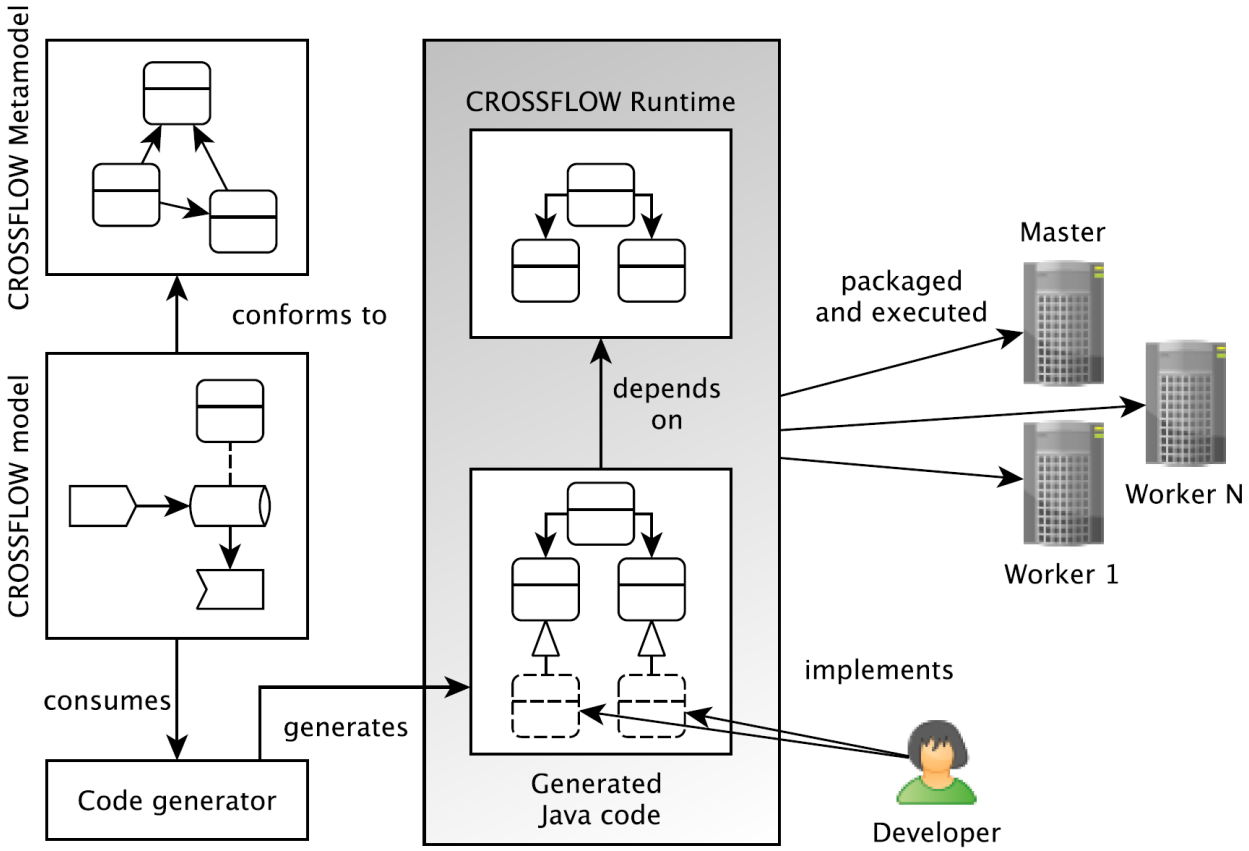

Figure 3: Crossflow architecture

4. RUNNING EXAMPLE

This section summarizes the running example originally described in D5.2, which will be used throughout the document in order to provide a cohesive narrative for the various capabilities of the Crossflow platform.

Figure 4 shows a graphical representation of this example, created using the Crossflow diagram editor (detailed in D5.5). This example looks for instances of files from specific technologies on GitHub (using their file extensions and a keyword contained in the file as the matching metric). The repositories containing such files are then cloned locally and various analysis tasks are performed to calculate the number of repositories, files and authors for each such technology. Finally, this analysis data is aggregated and output.

More specifically, 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 repository search tasks. The next time the workflow is executed (e.g. after a bug fix or after adding more technologies to the input CSV file), repository search jobs, i.e. executing queries to the

GitHub API, are routed to nodes that already have clones of relevant repositories from the previous execution (if available). Thus, unnecessary cloning of the same repositories in different nodes as required by subsequent repository analysis jobs, i.e. requiring access to repository clones, is prevented.

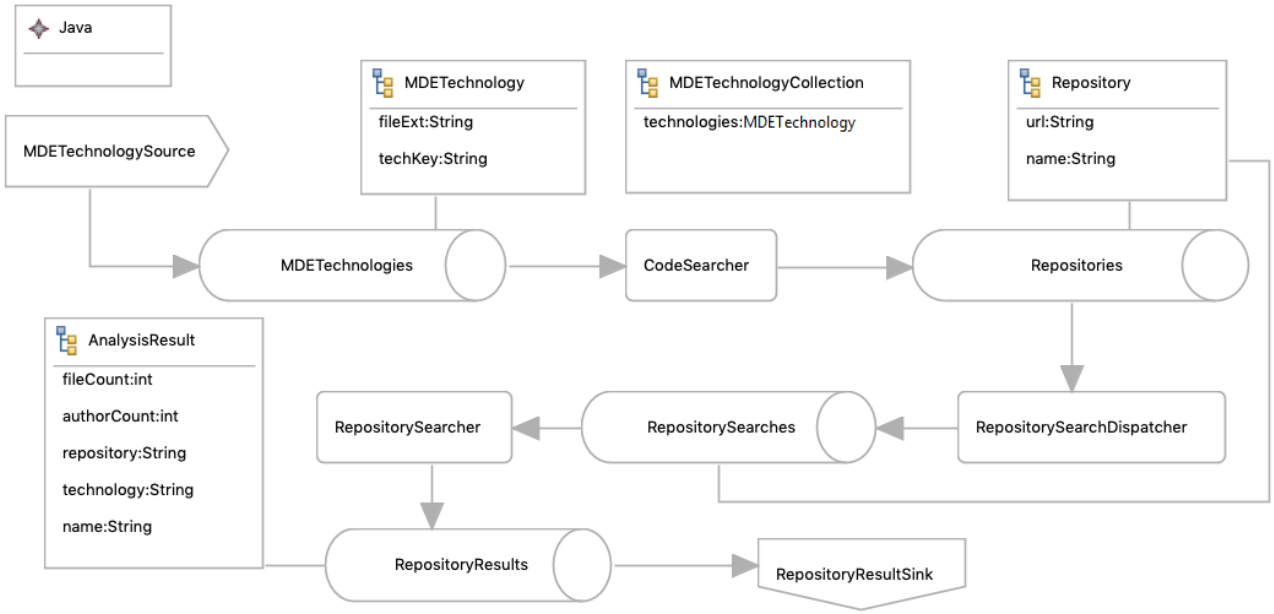

Figure 4: Running example (shown in Crossflow graphical editor)

5. UPDATED CROSSFLOW LANGUAGE (METAMODEL)

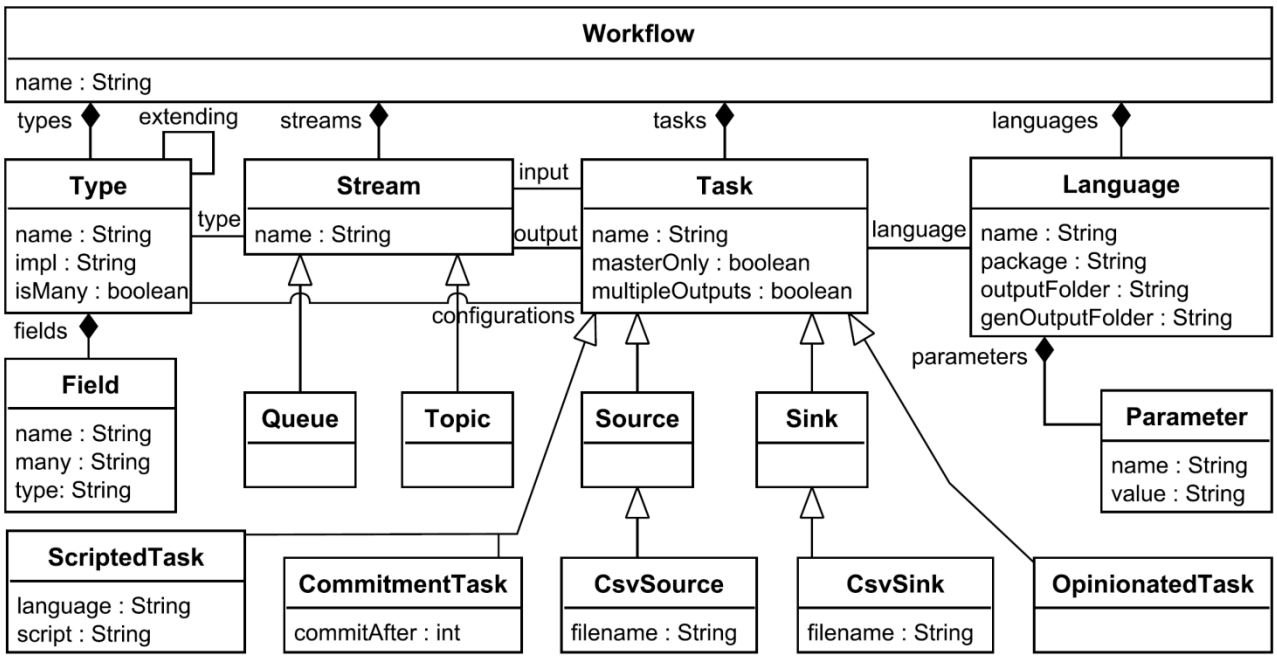

Figure 5: Crossflow language – Final Version

Figure 5 shows the final version of the Crossflow language, updated from D5.2: Figure 2. Key elements are detailed below:

Task: Instances of *Task* are defined by *name*, execution restriction to only allow the master node to execute them (*masterOnly*), and output multiplicity (*multipleOutputs*). Moreover, tasks may manifest as instances of *Source* or *Sink*. Sources and sinks produce input and output for their containing workflow, respectively.

Stream: Instances of *Stream* either represent queues or topics in a workflow model. Streams manifest as instances of *Queue* or *Topic* and receive input from tasks as well as produce output for tasks. Queues and topics are following the point-to-point and publish-subscribe subscription model, respectively.

Type: Instances of *Type* represent data types for streams, and may specify multiplicity (*isMany*), existing (or intended) implementations through the value of the *impl* attribute as well as instances of their *Fields*, detailed by a *name*, multiplicity (*many*), and *type*.

The key updates are detailed below:

- The *Configuration* element is removed; its contents are now set in the Crossflow API instead (detailed in section 7).
- *TaskTypes* is removed and handled using inheritance of the Task element:
    - *Source* denotes a task with no input streams, commonly used as the starting point of a workflow

- *Sink* denotes a task with no output streams, commonly used as the termination point of a workflow
  - *CsvSource*/*CsvSink* are specializations of such tasks that specifically read and write from/to Csv files.
- *CommitmentTask* is a new type of task that does not automatically accept its input, but instead waits to receive it again (n times) before it commits to processing it. When it does not accept an input it returns it to its originating channel, to be re-distributed to available workers. Such tasks are also referred to as tasks with preferences, as they will prioritise inputs they have already received in the past (for example input that had required data to be saved to the hard disk of the worker, which can now be re-used). Details into how this works can be found in Section 7.1.5.
- *OpinionatedTask* is a new type of task that only accepts input fitting a specific condition (defined in the task itself). Similarly to *CommitmentTask*, any input rejected by this task will be re-sent to its originating channel to be re-distributed. Such tasks are also referred to as tasks with capabilities, as they will only accept input that they are able to process (for example a task receiving urls will only accept them if it is currently able to access the internet). Details into how this works can be found in Section 7.1.7.
- *ScriptedTask* denotes a task that is implemented using a scripting *language*, by writing the script in the *script* attribute of the model itself. The code generator will then use an appropriate interpreter to run this script when the workflow is executed.

- *Language* is a new element denoting the programing language the task is going to be implemented in. This allows for heterogeneous language workflows whereby some tasks are written in Java, some in Python etc. and will hence only be processed by workers written in the same language. NB: the master will always be running in a Java runtime environment.
- *configurations* is a new reference a task can have to zero or more *Type* elements, denoting that this task needs to be configured before its execution. Details into how this works can be found in Section 7.1.4.

6. CROSSFLOW EXECUTION ENGINE

This section presents the runtime engine of Crossflow and how it interoperates with the code generators (presented below in Section 7). The engine is written in Java and relies on Apache ActiveMQ for inter-node communication.

In Figure 6, the primary artefact is the *Workflow*, in charge of coordinating the execution of the various tasks and receiving and sending data between itself and the ActiveMQ broker. One such *Workflow* instance per execution will be the *master*, in charge of the overall execution and capable of running master-only tasks as well as sources and sinks.

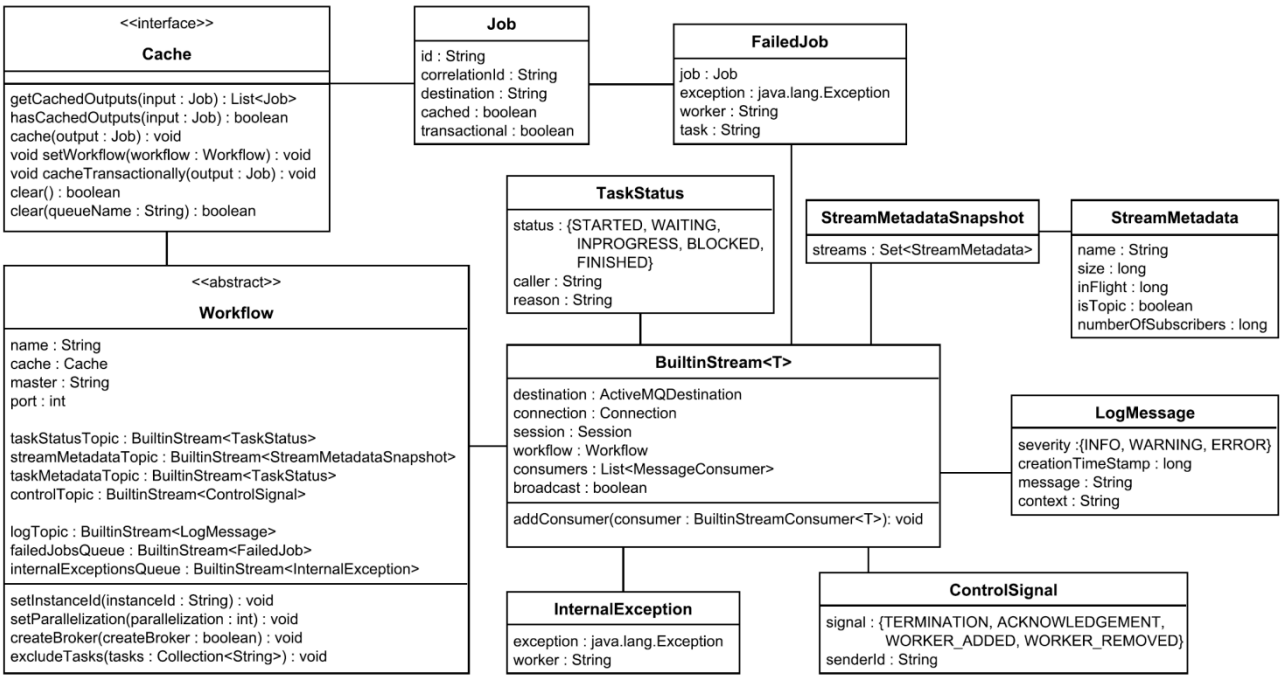

Figure 6: Crossflow core runtime classes

*Workflow*s have various built-in streams which expose execution-specific metadata to other workflow nodes as well as anyone subscribing to them (such as the web UI for example):

- *taskStatusTopic* broadcasts information about the execution of tasks in this *Workflow* instance
- *streamMetadataTopic* broadcasts information about the queues in the workflow (only the master will send messages to this channel)
- *taskMetadataTopic* broadcasts information about the tasks in this workflow (only the master will send messages to this channel). This is a small subset of the data sent to the *taskStatusTopic* in order to avoid overloading external consumers with too many messages
- *controlTopic* is used for coordinating multiple *Workflow* instances (for example a master and two workers) for keeping the workflow in-sync
- *logTopic* broadcasts log messages from any *Workflow* instance
- *failedJobsQueue* keeps any failed job to be handled
- *internalExceptionsQueue* keeps any engine-specific failures to be handled

A *Workflow* has a name, a unique ID (workers need to use the ID of the master in order to connect to it), a parallelization factor (each task in that *Workflow* instance will be parallelized n times) as

well as optionally a collection of Task identifiers for Tasks that this *Workflow* instance does not want to execute at all (commonly used when a worker wants to avoid processing specific tasks).

Finally, a *Workflow* can have a *Cache* so that any computations it performs (in one of its tasks) will be stored in it so that the next time the same computation is required, it is obtained from the cache instead. The cache is always stored by the master, defaulting to using the file system of the node (machine) it is running on. An in-depth look into how this is done can be found in Section 7**,** as the cache is directly used by the streams connecting the different tasks together (which do not exist at this stage as they are generated using the Crossflow model created by the user).

It is worth noting that at this stage, *Workflow* instances do not know about tasks or channels between tasks, it is the job of the code generator to extend *Workflow* in order to orchestrate the execution of the actual tasks. An in-depth look into how this is done can be found in Section 7.

7. CODE GENERATORS

Other than the core orchestration code described above in Crossflow Execution Engine 6, the remained of the code necessary for creating a workflow is generated from a Crossflow model instance. To demonstrate this we will be using the running example from Section 4.

Figure 7 shows a high-level view of the generated code for the running example. In order to keep a more detailed figure suitably sized, Figure 8 only shows one instance of a task (*CodeSearcher*), alongside all other relevant artefacts related to it. In both figures, yellow elements are core runtime classes, white elements are generated base classes and blue elements are implementation classes. In the latter figure we have the following generated base and implementation classes:

- *TechnologyAnalysis*: this class, extending *Workflow*, is the orchestrator for running an instance of the running example workflow; it contains references to the various tasks and streams the workflow has. During deployment, one instance of this class will run for each worker (and one for the master). Note that for each task that is a not source or a sink, the reference is to a list of tasks, used to parallelise their execution for this node (machine) instance.

- *Technologies*/*Repositories*: these classes, extending *JobStream*, represent the various logical queues connecting the workflow tasks together. Each such stream contains:

    - A list of its consumers: in the case of *Technologies* its only consumer is the task *CodeSearcher*

    - Three maps containing various ActiveMQ destinations (ActiveMQ queues); the keys to these maps are the identifiers for the matching destination. In the case of *Technologies*, these maps contain the following physical queues:

        - *post*: A single queue sending messages to the task *CodeSearcher*.

        - *pre:* A single queue receiving messages from the source task *TechnologySource*.

        - *destination*: A single queue sending messages to the matching post queue for the *CodeSearcher* task.

The reason for having collections of physical queues instead of a single one is that if there is more than one task consuming from the logical queue, each such consumer will have its own physical queue generated.

Furthermore, the reason for having three such collections of physical queues (pre, destination and post) is to facilitate the caching of intermediary results:

- Messages sent to the logical queue are actually only sent to its *pre* queue(s).
- *pre* queues are consumed by the appropriate *destination* queues: If the cache is disabled, these messages are simply forwarded to the appropriate *post* queue(s) and consumed by task(s) (in the case of the *Technologies* queue they are consumed by *CodeSearcher*). If the cache is enabled, the engine checks it for any stored outputs from this queue and this specific message. If the cache does, then this output is retrieved and forwarded directly to the relevant output queue(s) of the consumer task(s), instead of being sent to them at all (in the case of the *Technologies* queue they are sent to the *Repositories* queue).
- *post* queues are the only physical queues actually sending messages to tasks, one such queue for each consumer task, resulting in evenly distributing the messages amongst all workers and all parallel instances of such tasks (if the logical queue is not marked as broadcast, in which case it does not share the messages but instead replicates them to each consumer instead).

Caches are only stored on the master workflow node since all queues communicate through the master.

- *CodeSearcherBase*: this base class, extending *Task*, functions as a wrapper around the actual task implementation class, in order to provide the following functionalities:
  - Status updates whenever a task begins processing an element and when it finishes processing it
  - If the task has multiple outputs, then it will create a transaction waiting for the task to finish before sending any of them to their relevant streams. Hence if a task fails in the middle of processing an element none of its outputs will be sent, retaining a consistent state.
  - If the parallelization of the workflow is greater than 1, a shared *Threadpool* is used amongst the parallel instances of the task, and this wrapper is responsible for using it to execute the actual logic when there is a slot available

  Section 7.1 shows listing for the implementations of the various tasks of the running example.

- *Technology*: this class, extending *Job* represents a single instance of a message to be sent along the workflow. In this case, this message is created by the source task *TechnologySource* to the stream *Technologies* and consequently consumed by the task

CodeSearcherBase. It contains fields for the data it stores, in this case two Strings: *filExt* and *techKey*.

Finally, other than for the master workflow that is always running in Java, if any tasks are defined for any other programing language (through their *language* reference in the Crossflow model), then appropriate code for that language will be generated (calling the relevant code generators for that language), allowing for clients written in that language to be deployed, which can then process these tasks.

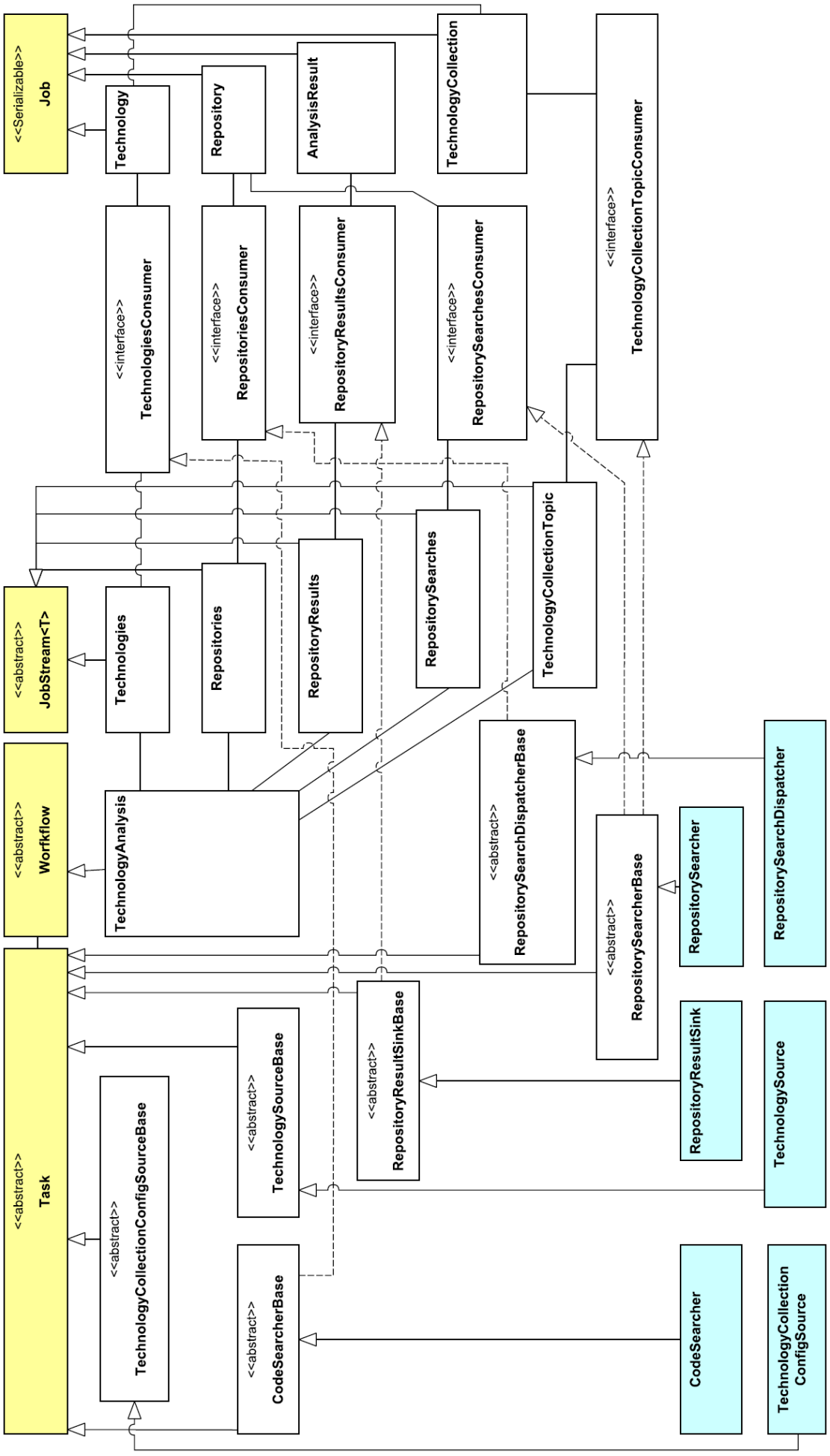

Figure 7: Running Example Class Hierarchy

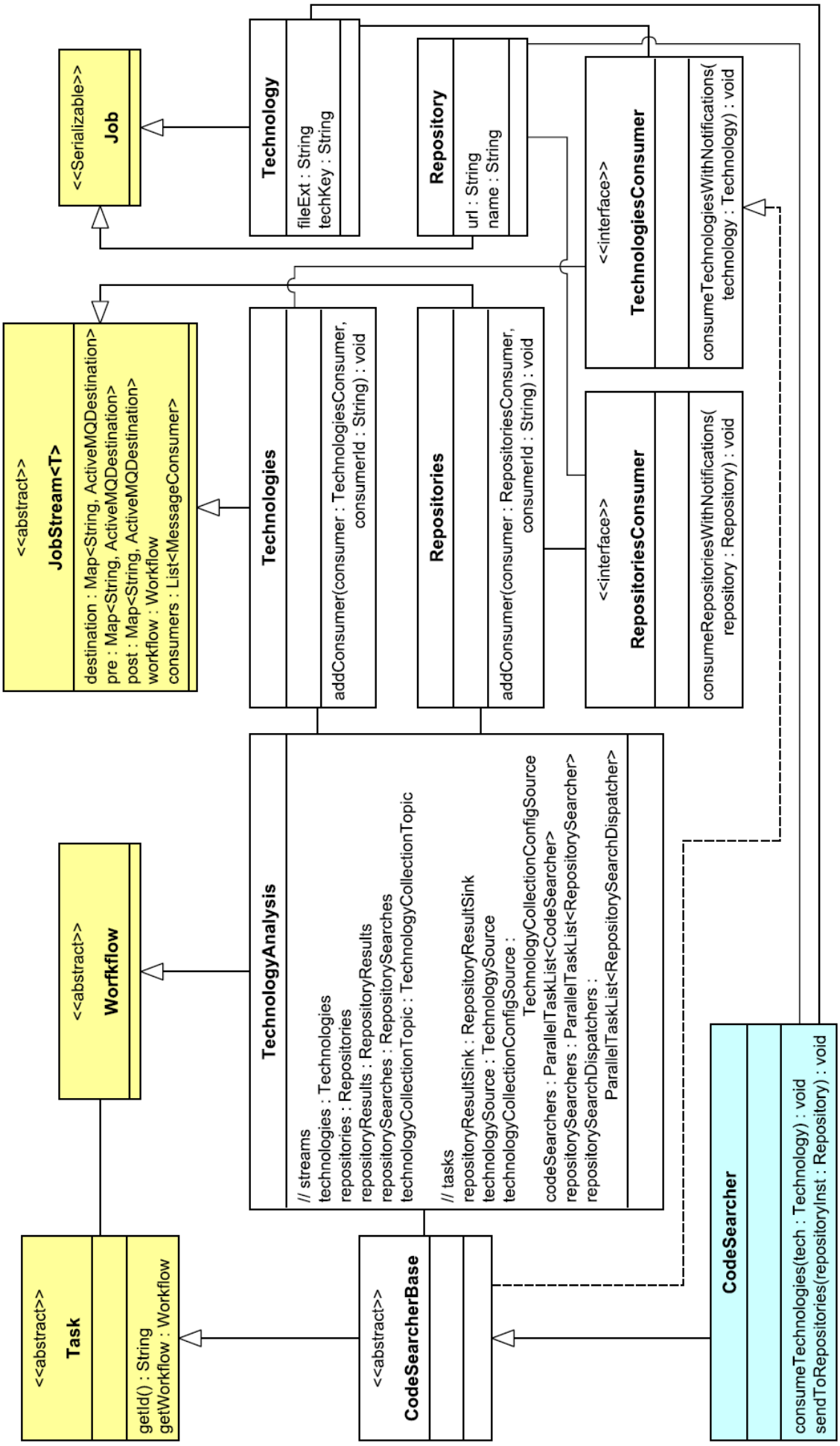

Figure 8: Running Example Sample Classes

7.1 JAVA IMPLEMENTATION CLASSES

After the code generation is complete, the developer can then complete the relevant implementation classes (stubs of which are created by the generator) with their analysis logic. An example of each type of such class is shown below:

7.1.1 Source task

```
10 public class TechnologySource extends TechnologySourceBase {
11
12     @Override
13     public void produce() {
14         try {
15             final CsvParser parser = new CsvParser(
16                     new File(workflow.getInputDirectory(), "input.csv").getAbsolutePath());
17             Iterable<CSVRecord> records = parser.getRecordsIterable();
18
19             for (CSVRecord record : records) {
20                 Technology technologyTuple = new Technology();
21                 technologyTuple.setFileExt(record.get(0));
22                 technologyTuple.setTechKey(record.get(1));
23                 sendToTechnologies(technologyTuple);
24
25             }
26         } catch (Exception e) {
27             workflow.log(SEVERITY.ERROR, e.getMessage());
28         }
29     }
30
31 }
```

Such tasks always have a *produce* method (line 13) that will get called once when the workflow starts. In this method, the developer can send any number of messages to the queues the source is connected to, by calling the method *sendTo*()* where * is the name of the queue in question. For example in line 23, an instance of *Technology* is sent to the queue *Technologies*.

Line 16 uses *workflow.getInputDirectory()*, whereby *workflow* is a variable available to all tasks that references the workflow instance they are a part of. One of the methods of workflow is *getInputDirectory()* that returns the path of the input directory of this workflow, a folder containing files to be used by this workflow, commonly for the initial inputs of its source(s).

Line 20 creates a new instance of *Technology* (a *Job*). This is the generated class for the data type and hence contains fields for the type's contents. In this case the variables *fileExt* and *techKey* can be set using their setter methods (lines 21 and 22).

All tasks have access to the workflow's logger, such as on line 27. This logger sends messages to its own stream (as detailed in Section 6) and can be freely used by the developers for sending any messages they wish to see during task execution.

7.1.2 Generic Task

```
10 public class CodeSearcher extends CodeSearcherBase {
11
12     @Override
13     public void consumeTechnologies(Technology tech) throws Exception {
14         search(tech);
15     }
16
17     private void search(Technology tech) {
18
19         String query = new CodeSearchQuery().create(tech.getTechKey())
20                 .extension(tech.getFileExt()).inFile().build().getQuery();
21
22         IDataSet<SearchCode> ret = GitHubUtils.getOAuthClient()
23                 .getSearchCode("asc", query, null);
24
25         List<SearchCode> repoFiles = ret.observe().toList().blockingGet();
26
27         // files in current repo
28         for (SearchCode resultItem : repoFiles) {
29             org.eclipse.scava.crossflow.restmule.client.github.model.SearchCode.Repository
30                 resultRepo = resultItem.getRepository();
31             Repository repositoryInst = new Repository();
32             repositoryInst.setUrl(resultRepo.getHtmlUrl());
33             repositoryInst.setName(resultRepo.getFullName());
34
35             sendToRepositories(repositoryInst);
36         }
37
38     }// search
39
40 }
```

Such tasks will commonly have one or more streams they are receiving messages from and one or more streams they are sending messages to. In this case the task receives messages from the *Technologies* stream, through the method *consume*()* in line 13. This task sends messages to the *Repositories* stream by calling the method *sendTo*()* in line 35.

7.1.3 Master-only Task

```
6  public class RepositorySearchDispatcher extends RepositorySearchDispatcherBase {
7
8      private Set<String> repos = new HashSet<>();
9
10     @Override
11     public void consumeRepositories(Repository repository) throws Exception {
12
13         // send each repository only once to be cloned/analysed
14         if (repos.add(repository.name))
15             sendToRepositorySearches(repository);
16
17     }
18
19 }
```

Such tasks are identical in their Java API to the generic tasks above, but since they are defined (through their Crossflow model instance) as *MasterOnly*, they will only be executed on the master workflow. As such, any local variables they use, like the *repos* field in line 8, will never be replicated amongst multiple workers and can be safely used for aggregating data and performing global decisions.

7.1.4 Configurable Task

```
25 public class RepositorySearcher extends CommitmentRepositorySearcherBase {
26
27     private class RepositoryClone {...
45
46     private Map<String, String> technologies = new HashMap<>();
47
48     // results for each technology
49     Map<String, AnalysisResult> results = new HashMap<>();
50
51     @Override
52     public void consumeRepositorySearches(Repository repository) throws Exception {
53         results.clear();
54
55         // clone repo if not already present
56         RepositoryClone localClone = cloneRepo(repository);
57
58         // use clone to get files (and count)
59         countFiles(localClone);
60
61         // use clone to get authors (and count)
62         countAuthors(localClone);
63
64         for (AnalysisResult r : results.values())
65             sendToRepositoryResults(r);
66     }
67
68     @Override
69     public void consumeTechnologyCollectionTopic(TechnologyCollection technologyCollection)
70             throws Exception {
71
72         for (Technology tech : technologyCollection.technologies)
73             technologies.put(tech.getFileExt(), tech.getTechKey());
74
75     }
76
77     private RepositoryClone cloneRepo(Repository repository) {...
145
146     public AnalysisResult getOrCreateResult(RepositoryClone repositoryClone, String tech) {...
159
160     // utility methods for cloning
161     private String createUniqueFolderForRepo(String name, String url) {...
184
185     // utility methods for file counting
186     private void countFiles(RepositoryClone repositoryClone) {...
202
203     // utility methods for author count
204     private void countAuthors(RepositoryClone repositoryClone) {...
244
245 }
```

Such tasks are similar to generic tasks but will also have a method for consuming one or more configurations. As such, line 69 contains the *consume** method for the configuration with type *TechonologyCollection*. This method will have to return before any other consume method is called for this task (aka the *consumeRepositorySearches* method in line 52 will be blocked until the configuration method was received and processed its message). Hence the field technologies in line 46 will always be fully populated before it is accessed by the remainder of the task's methods.

In order to provide such configuration files to their appropriate tasks, for each configuration type element a source is created upon code generation:

```
 9 public class TechnologyCollectionConfigSource extends TechnologyCollectionConfigSourceBase {
10
11     @Override
12     public void produce() throws Exception {
13         try {
14             final CsvParser parser = new CsvParser(
15                     new File(workflow.getInputDirectory(), "input.csv").getAbsolutePath());
16
17             TechnologyCollection collection = new TechnologyCollection();
18
19             for (CSVRecord record : parser.getRecordsIterable()) {
20                 Technology technologyTuple = new Technology();
21                 technologyTuple.setFileExt(record.get(0));
22                 technologyTuple.setTechKey(record.get(1));
23                 collection.technologies.add(technologyTuple);
24             }
25             sendToTechnologyCollectionTopic(collection);
26
27         } catch (Exception e) {
28             workflow.log(SEVERITY.ERROR, e.getMessage());
29         }
30
31     }
32
33 }
```

Similarly to any other source, it has a *produce()* method (line 12), but it will always send messages to a pre-defined configuration topic named **Topic* where * is the name of the configuration type, in this case *TechnologyCollection* (line 25).

7.1.5 Commitment Task

The task listing in Section 7.1.4 (task *RepositorySearcher*) is also a commitment task, as well as a configurable task. Commitment tasks are identical in their Java API to generic tasks but will implicitly perform selective acceptance of incoming messages based on whether they have already seen the same message beforehand. For example the method in line 52 of the *RepositorySearcher* listing in Section 7.1.4 will not get called the first time a repository with name crossminer/scava is received, but only after the same repository have already been received (and hence rejected) *N* times where *N* is the value of the *commitAfter* attribute of the task in the Crossflow model. As mentioned in Section 0, every time such a task rejects a message, it returns it to its originating channel to be re-distributed amongst any available workers.

7.1.6 Sink Task

```
22 public class RepositoryResultSink extends RepositoryResultSinkBase {
23
24     protected HashMap<String, AnalysisResult> results = new HashMap<String, AnalysisResult>();
25
26     private boolean started = false;
27     private Timer t = new Timer();
28
29     @Override
30     public void consumeRepositoryResults(AnalysisResult analysisResult) throws Exception {
31         System.out.println("sink consuming: " + analysisResult);
32
33         if (!started)
34             t.schedule(new TimerTask() {
35                 public void run() {...
36             }, 2000, 2000);
37         started = true;
38
     ...
56     }
57
58     @Override
59     public void close() {
60         t.cancel();
61         flushToDisk();
62     }
63
64     private synchronized void flushToDisk() {...
90     // ----
91
92 }
```

Such tasks are similar to generic tasks, but do not have any outgoing streams they can send messages to. As such, they are commonly used as termination points of workflows by storing output on disk or printing to console or the workflow logger.

7.1.7 Opinionated Task

The running example does not contain an instance of an opinionated task so the listing below uses a simple example instead.

```
3  public class OccurencesMonitor extends OpinionatedOccurencesMonitorBase {
4
5      protected int occurences = 0;
6
7      @Override
8      public void consumeWords(Word word) throws Exception {
9          occurences++;
10     }
11
12     @Override
13     public boolean acceptInput(Word input) {
14         //logic for when to accept tasks for this instance of OccurencesMonitor goes here.
15         return input.getW().equals(favouriteWord);
16     }
17
18     public int getOccurences() {
19         return occurences;
20     }

32 }
```

In this example, this opinionated task receives instances of the Job *Word* (in the *consume** method in line 8) and will count how many of them it has received, but only for the single word it has denoted as a *favoriteWord*.

Opinionated tasks always have a Boolean method *acceptInput()* (line 13), which gets called before any of the *consume*()* methods of the task. If this method is true for this element, it is accepted by this task.

As mentioned in Section 0, every time such a task rejects a message, it returns it to its originating channel to be re-distributed amongst any available workers.

7.2 PYTHON GENERATORS

Code generators for workflow workers in Python have been developed, offering similar functionality to those presented above, but generating base classes and implementation stubs in Python instead of Java. Since Crossflow aims at providing a language-agnostic framework and the use-case providers only need Java as their task implementation language, Python was chosen as an example of another popular language to be supported out-of-the-box.

For example, the task *CodeSearcher* can be seen below:

```
from org.eclipse.scava.crossflow.examples.techanalysis.CodeSearcherBase import CodeSearcherBase
from org.eclipse.scava.crossflow.examples.techanalysis.Repository import Repository

class CodeSearcher(CodeSearcherBase):

    def __init__(self):
        super().__init__()

    def consumeTechnologies(self, technology):
        query = CodeSearchQuery().create(tech.getTechKey()).extension(tech.getFileExt())\
            .inFile().build().getQuery()

        ret = GitHubUtils.getOAuthClient().getSearchCode("asc", query, null)

        repoFiles = ret.observe().toList().blockingGet()

        for resultItem in repoFiles:
            resultRepo = resultItem.getRepository()
            repositoryInst = Repository()
            repositoryInst.setUrl(resultRepo.getHtmlUrl())
            repositoryInst.setName(resultRepo.getFullName())

            self.sendToRepositories(repositoryInst)
```

Only tasks able to run on workers can be written in languages other than Java, as the master will always be running in a Java environment. As such, *Sources*, *Sinks* and *MasterOnly* tasks are not able to be written in and executed by other programing languages.

8. INTEGRATION WITH THE CROSSMINER PLATFORM

Crossflow is integrated with the Crossminer platform in two ways: Workflows can be executed by metric providers and Workflows can retrieve data from the knowledge-base to be used in their execution.

8.1 EXECUTING WORKFLOWS FROM METRIC PROVIDERS

If a metric provider wishes to execute a Crossflow workflow, it is able to do so by using the Java API of Crossflow:

- The metric provider developer creates a Crossflow model using one of the tools demonstrated in D5.5, such as the graphical editor.
- The developer generates the source code for this workflow using the context menus provided by the Eclipse user interface of Crossflow, as demonstrated in D5.5.
- The developer bundles this code as an executable either by using the Eclipse plugin paradigm or by exporting it as a standalone jar archive, as demonstrated in D5.5.
- The developer uses the Java API of Crossflow to create a new workflow and execute it within their metric provider. Crossflow's Java API is detailed in Appendix A.

8.2 RETRIEVING DATA FROM THE CROSSMINER KNOWLEDGE-BASE TO BE USED WITHIN WORKFLOWS

The Crossminer knowledge base exposes its contents through a REST API, as detailed in D6.5, Section 9.4. As such, a workflow wanting to retrieve data found in the knowledge base can simply use this REST API by creating the appropriate HTTP GET/POST request (available requests are seen in D6.5, Figure 36) within any workflow Task. For example a source task may retrieve project clustering data from the knowledge base and output the results obtained to a queue, so that they can then be analysed by further tasks in the workflow.

9. RESTMULE

RestMule [7] (Appendix B.1) is a framework, detailed in D5.4, comprising a set of reusable facilities that handle request rate quota, network failures, caching and asynchronous paging of http requests to online data sources like GitHub and StackOverflow, in a transparent manner. It does so by providing a code generator that produces API-specific Java libraries from open source service provider interfaces descriptions, i.e., a combination of OpenAPI-confirming JSON descriptions and service policy models that employ reusable facilities of the RestMule CORE plugin. RestMule allows developers to produce resilient remote-API clients from enabling them to focus on their core mining and analysis logic. Our initial evaluation demonstrated that such resilient clients can be used to express queries in a much more concise manner, whilst providing the resilience that would be necessary to execute them, which would otherwise have to be manually implemented.

RestMule has been designed to return Data Access Objects (DAOs), which implement the RxJava3 *Observer* interface, offering asynchronous access to the returned results, as they are being obtained. Moreover, both types of aforementioned DAOs return *ReplaySubjects* that maintain a copy of the data pushed to its subscribers such that all observers can get the same data upon request. This allows for RestMule to be easily integrated into Crossflow either as a *Source* task with pre-defined configurations into what it will fetch, or as an intermediary workflow task receiving as input the data required to form its queries to the remote data sources. Finally, by using Crossflow's caching, retrieving data more than once is not necessary regardless of how many times a workflow accessing it is executed, allowing for an optimal use of possibly rate-limited remote API calls.

10. CONCLUSIONS

This deliverable presented Crossflow, the Crossminer component responsible for parallel and distributed execution of language-agnostic analysis workflows. Crossflow comprises a high-level domain-specific language used to define workflows, code generators used to convert this model into executable classes, and an execution engine used to execute them. Crossflow's architecture is presented, focusing on how its various components are linked together in the system. The Crossflow language is detailed, focusing on any changes made for this final version, when compared to the earlier versions mentioned in the previous deliverables. A running example is used for demonstrating Crossflow is used in practice, focusing on the code-related aspects of the tool, leaving many of the user-facing components such as the model editors and web interfaces for D5.5. Finally it presented a summary of how Crossflow can be used by metric providers in Crossminer as well as how workflows can use data from the Crossminer knowledge base for analysis.

Tables on the final status of the user and technology requirements related to WP5 of the Crossminer project can be found below, containing the requirement name and description, its overall priority as well as its final status.

REFERENCES

[1] D. S. Kolovos, N. Matragkas, I. Korkontzelos, S. Ananiadou and R. Paige, "Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects," in *Proceedings of the Open Source Software for Model Driven Engineering Workshop (OSS4MDE'15)*, Ottawa, 2015.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker and I. Stoica, "Apache Spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, pp. 56-65, 2016.

[3] Apache, "Apache Storm," 2018. [Online]. Available: http://storm.apache.org.

[4] Apache, "Apache Samza," 2018. [Online]. Available: http://samza.apache.org.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28-38, 2015.

[6] K. M. M. Thein, "Apache Kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, pp. 9478-9483, 2014.

[7] B. A. Sanchez, K. Barmpis, P. Neubauer, R. F. Paige and D. S. Kolovos, "RestMule: Enabling Resilient Clients for Remote APIs," in *Proceedings of the ACM/IEEE 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29*, 2018.

TABLE ON FINAL STATUS OF USE-CASE PARTNER REQUIREMENTS FOR WP5

ID	Description	Overall Priority	Status
U122	Provides a mechanism to define a retrieval, cleaning and analysis process based on reusable components	SHOULD	Supported
U123	Provides a mechanism to play individual parts of the retrieval and analysis process	SHOULD	Supported
U124	Provides a mechanism to easily analyse new data sources and define new measures	**SHALL**	Supported
U125	Able to use data from all existing data collectors	SHOULD	Supported
U126	Provides a mechanism to analyse and visualise data in an external tool (e.g. R, Tableau, .. Excel)	SHOULD	Supported
U127	Provides a library of reusable components	SHOULD	Supported
U128	Provides a list of available data sets when building a new workflow	SHOULD	Unsupported
U129	Provides R as a computing engine for analyses	SHOULD	Supported
U130	Able to identify if the developer is not using the most recent version of a library and provide notification	**SHALL**	Supported
U131	Provides a means to execute a workflow across data sets	SHOULD	Supported
U132	Provides a means to execute a workflow across forges	SHOULD	Supported
U133	Able to support different execution priorities	*MAY*	Unsupported
U134	Allows compositions of Crossminer and external results to support decisions	**SHALL**	Supported
U135	Able to define specific formatting for the results	**SHALL**	Supported
U136	Able to process data sets from GitHub and StackOverflow	**SHALL**	Supported

TABLE ON FINAL STATUS OF TECHNOLOGY REQUIREMENTS FOR WP5

ID	Description	Overall Priority	Status
D47	The framework shall provide built-in support for network/API error recovery	**SHALL**	Supported
D48	The framework shall provide built-in support for data caching	**SHALL**	Supported
D49	The framework shall provide support for graphical editors for specifying knowledge extraction workflows	**SHALL**	Supported
D50	The framework shall provide a Java API for specifying knowledge extraction workflows	**SHALL**	Supported
D51	The graphical workflow editors should provide support for auto-completion, navigation and refactoring	SHOULD	Supported
D52	The framework shall provide parallel workflow execution capabilities	**SHALL**	Supported
D53	The framework shall provide distributed workflow execution capabilities	**SHALL**	Supported
D54	The framework shall provide debugging facilities	**SHALL**	Supported
D55	The framework shall provide workflow execution monitoring facilities	**SHALL**	Supported
D56	Workflow execution facilities shall be architecturally consistent with the platform so that workflows can be executed as metric providers	**SHALL**	Supported
D57	Connectors shall be implemented for the APIs of GitHub, StackOverflow and Bugzilla	**SHALL**	Supported
D58	Connectors should be implemented for GHTorrent, GitHub Archive and JIRA	SHOULD	Unsupported
D59	The platform shall expose a REST API that workflow components can consume	**SHALL**	Supported
D60	The API of the platform should be formally specified	SHOULD	Supported
D61	Mining tools developed in WPs 2-4 and 6 should be embeddable as components in knowledge extraction workflows	SHOULD	Supported
D62	The platform shall provide facilities for running custom workflows and displaying the results in appropriate dashboards	**SHALL**	Supported

APPENDIX A: EXECUTION USING JAVA API

A workflow can be executed through a pure Java API after the relevant generation and implementation steps have been performed (as described in D5.5 as well as Section 7 (of this deliverable)). Using the generated class extending *Workflow*, all necessary configuration and lifecycle methods can be accessed. Using the running example and its generated *TechnologyAnalysis* class, below are the core methods available:

method	description
setName(String)	Sets the name of the workflow
setInstanceId(String)	Sets the unique identifier of the workflow – all workers need to use this identifier of the master workflow to connect to it
setMaster(String)	Sets the URL of the master workflow – all workers need to use this URL of the master workflow to connect to it
setPort(int)	Sets the port of the master workflow – all workers need to use this port of the master workflow to connect to it
createBroker(boolean)	Applicable to master workflows only – sets whether this master starts a new embedded ActiveMQ broker (otherwise it needs to connect to an existing one)
excludeTasks(Collection<String>)	Any task names added here will not be executed by this instance of workflow, allowing deployment of workers performing only select subsets of tasks
hasTerminated() : boolean	Returns whether this workflow has terminated, a blocking version of this method is *awaitTermination()*
setCache(Cache)	Applicable to master workflows only – enables caching by setting the cache – the in-built implementation is *DirectoryCache*
setInputDirectory(File)	Sets the directory where input files should be placed to be used during workflow execution
setOutputDirectory(File)	Sets the directory where output files will be created/updated during execution
setParallelization(int)	Can only be set before the workflow has started – sets the parallelization factor for this instance of workflow
TechnologyAnalysis(Mode)	Constructor – sets the Mode of this workflow to MASTER, MASTER_BARE (a master which is not also a worker) or WORKER
run() / run(long)	Starts the workflow with an optional initial delay of *x* ms
createWorker()	Convenience method to create a new worker for this master – should be called after the current instance is fully configured
get*() -- where * is the name of any Task or Stream (such as CodeSearcher or Technologies)	Since all tasks have access to their relevant workflow object, these methods allow for any task to access any other task or stream, allowing for them to send messages to any stream or access local data/variables from other tasks. NB: any data obtained from other tasks will be local data from the node/machine the current workflow is running on

The generators will create a simple example app using this API that can be executed to run the workflow on the current machine. The app will be named *App.java and will be found in the output folder of the generated code (alongside the implementation classes), where * is the name of the workflow, i.e. for the running example, the app will be named TechnologyAnalysisApp.java.

APPENDIX B: PUBLICATIONS

B.1: RESTMULE

RestMule: Enabling Resilient Clients for Remote APIs

Beatriz A. Sanchez
Department of Computer Science
University of York
York, UK
basp500@york.ac.uk

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

Richard F. Paige
Department of Computer Science
University of York
York, UK
richard.paige@york.ac.uk

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

ABSTRACT

Mining data from remote repositories, such as GitHub and StackExchange, involves the execution of requests that can easily reach the limitations imposed by the respective APIs to shield their services from overload and abuse. Therefore, data mining clients are left alone to deal with such protective service policies which usually involves an extensive amount of manual implementation effort. In this work we present RESTMULE, a framework for handling various service policies, such as limited number of requests within a period of time and multi-page responses, by generating resilient clients that are able to handle request rate limits, network failures, response caching, and paging in a graceful and transparent manner. As a result, RESTMULE clients generated from OpenAPI specifications (i.e. standardized REST API descriptors), are suitable for intensive data-fetching scenarios. We evaluate our framework by reproducing an existing repository mining use case and comparing the results produced by employing a popular hand-written client and a RESTMULE client.

CCS CONCEPTS

• **Information systems** → **RESTful web services**; • **Software and its engineering** → *Reusability*; *Error handling and recovery*;

KEYWORDS

Resilience, OpenAPI Specification, HTTP API Clients

ACM Reference Format:
Beatriz A. Sanchez, Konstantinos Barmpis, Patrick Neubauer, Richard F. Paige, and Dimitrios S. Kolovos. 2018. RestMule: Enabling Resilient Clients for Remote APIs. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3196398.3196405

MSR'18, May 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196405

1 INTRODUCTION

Research in the area of Mining Software Repositories (MSR) [6] investigates challenges associated with handling information that originate from public software development data sets, source code repositories, Q&A knowledge bases, requirements and issue tracking systems, and are commonly offered by service providers in form of remote APIs.

Service providers commonly constrain the access to public data sets in term of service policies to protect their services from saturation and abuse. For example, such protective measures may be implemented by imposing a limit to the number and rate of client requests, returning multi-page responses, and black-listing abusive clients. However, there are a number of research applications, such as evaluating the popularity of software technologies [7, 9], that require the execution of a number of requests to remote APIs that easily exceed rate limits imposed by service provides. Consequently, as a result of requiring clients to handle service provider policies during the execution of their collection and analysis tasks, their implementation is considered to be a cumbersome and time-consuming activity [5].

Although some service providers offer client libraries for their services in a variety of programming languages and thus greatly reduce implementation effort by providing an abstraction from low-level remote API behavior, such as informal client-server interaction contracts over the native HTTP protocol, such a library usually (*i*) does not address server-side protection mechanisms, such as request throttling, (*ii*) requires extensive manual implementation effort during its development and maintenance, and (*iii*) is limited by the functionality and behavior exposed through its operations.

Research in the area of MSR has brought to light a set of reusable frameworks, such as BOA [4], as well as replications of public repositories, such as GHTorrent [5], the Stack Exchange's Data Dump [2] and the Maven Repository Dataset [12], that enable bypassing service provider policies. However, employing data sets mirroring large public repositories exposes several challenges, like the requirement of (*i*) maintaining a large infrastructure to import, store, maintain, and provide access to mirrored repository information, (*ii*) handling inconsistencies in replicated data sets [5], and (*iii*) dealing with outdated information.

The work presented in this paper offers an approach, which has been implemented in terms of the RESTMULE framework, that

addresses several repository mining challenges by providing a solution to semi-automatically generate resilient clients from standardized REST API specifications. In general, our approach generates executable clients for remote APIs that are formally defined in terms of OpenAPI [10] specifications, which has been proposed as a machine-readable format for describing the architecture of services offered as RESTful remote APIs [11]. Moreover, resilience is achieved by specifying service policies, such as rate limits and pagination, and handling them, as well as network failures and response caching, accordingly and in a graceful and transparent manner.

Roadmap. The rest of the paper is structured as follows. Section 2 presents our approach alongside its architecture and design within the RestMule framework. Section 3 demonstrates the evaluation of our framework by reproducing an existing repository mining use case based on GitHub and comparing the respective results produced by employing a prominent hand-coded client and a RestMule-generated client. Section 4 compares with existing literature and frameworks in the area of intense data collection and Mining Software Repositories, in general. Section 5 concludes our work by summarizing findings and highlighting future work.

2 APPROACH

This section presents our approach on semi-automated generation of resilient clients by focusing on the design and architectural components of its implementation within the RestMule framework[1].

2.1 Architecture and Design

The RestMule framework is designed on a layered architecture to reduce the coupling between the two main components, i.e., RestMule core and RestMule codegen, as well as to reduce the amount of generated code.

In general, the RestMule core component, c.f. circle 1 in Fig. 1, and the RestMule codegen component, c.f. circle 2 in Fig. 1, act as general purpose layer and API-specific layer, respectively. Moreover, the general purpose layer encapsulates the functionality that can be shared across different API-specific components. The RestMule API client component, c.f. circle 3 in Fig. 1, is responsible to bridge the gap between the functionalities offered by the RestMule core component and API-specific RestMule-generated clients. RestMule has been designed to return Data Access Objects (DAOs) that handle the different types of successful HTTP response payloads as well as provide insights to the data acquisition status. The entities responsible for submitting requests to the service provider are wrappers of plain API requests. Furthermore, these entities (*inner clients*) offer a list of services that encapsulate and handle various aspects of the system, such as API-specific request-limits and pagination. The RestMule framework defines a number of inner clients that is equal to the number of rate request limit policies implemented by the service provider. For example, GitHub implemented two different request limit policies for two different groups of HTTP endpoints, i.e., one for those starting with `/search/*` and one for all other endpoints, and thus requires defining two inner clients to handle both groups of endpoints appropriately.

[1]https://github.com/beatrizsanchez/RestMule

Inner clients are associated to user *sessions*, i.e., employed to authenticate HTTP requests, and may affect the request rate limit value of individual endpoint groups. For example, in GitHub the aforementioned `/search/*` endpoints (only) allow 10 request per minute to a public session and 30 requests per minute to an authenticated session. RestMule abstracts multiple inner clients by providing users with a single *entrypoint* that offers services and delegates their execution to any appropriate inner client. Moreover, in case a response is available as a valid cache entry, no request is issued to the service provider.

Resilient Client Generator. In general, the resilient client generator, c.f. circle 2 in Fig. 1, takes an OpenAPI Specifications (OAS) in JSON as well as a service policy description as input and produces a *RestMule model* that conforms to the *RestMule metamodel*. In more detail, the service policy description is represented by an Epsilon Object Language (EOL) [8] script and contains information, such as pagination and rate-limits. Finally, the *RestMule model*, which conforms to the *RestMule metamodel*, c.f. Fig. 2 for a simplified version, is consumed by the *M2T* transformation for the generation of the resilient Java client, c.f. circle 3 in Fig. 1. The generated code is structured as an Eclipse Plugin[2] and can be employed as Java library by third party-applications. In the following we describe functionalities and behaviors that are shared among RestMule-generated clients.

Data Access Objects (DAOs). In addition to simple types, RestMule API client can handle three types of HTTP response payloads: single objects, arrays of objects, and objects containing pointers to data needing to be fetched (wrappers). The latter two may contain information regarding limitations on the number of items to be provided or stored, specially if they are in disagreement with the total count returned by the service provider. For example, GitHub returns details of a maximum of 1000 items as well as 100 items per page but its response may indicate that a total of 2000 items have been found, i.e., addressed by the RestMule API client in terms of supporting capped results.

The data returned by the DAOs implements the Observer interface (from RxJava[3]). Both types of aforementioned DAOs return ReplaySubjects, which keep a copy of the data pushed to its subscribers such that all observers can get the same data upon request.

Pagination and Wrappers. To manage paged responses from the remote sources, two components are used: page traversal and response body wrapping. For the page traversal strategy, RestMule core is used by RestMule API client, with the API-specific pagination parameters provided and relevant methods defined. Response body wrappers wrap appropriate responses to provide common accessors to be used within the page traversal methods. As various sources may define different collection schemas, they are API-specific. For example the one for GitHub provides the mapping to the specific JSON fields that GitHub provides.

As data is retrieved through callbacks, the handling of these responses (successful or unsuccessful – of asynchronously-sent requests) for different pages that are associated to the same result

[2]https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
[3]https://github.com/ReactiveX/RxJava

JDK 1.8.0_92. Since running all 18 technologies would take in the order of weeks to execute, we decided to replicate the experiment for a subset of these technologies, more specifically for those used to create graphical model editors – Eugenia, GMF and Sirius.

```
IDataSet<SearchCode> searchCode =
  gitHubAPI.getSearchCode("asc", query, "indexed");
```

Listing 1: Query snippet for RestMule

```
try {
  key = Cache.key(gitHub.queryCode(query));
  if (Cache.contains(key)) { return cache entry; }
  else {
    response = gitHub.queryCode(query);
    Cache.put(key, response);
  }
  for (page : response.pages){
    // request additional pages and consider caching
  }
} catch (NetworkException e1){ handle(e1);
} catch (AuthenticationException e2){ handle(e2);
} catch (ServerException e3){
  handle(e3); // for similar responses
  waitAndRetry(); // for rate limits
}
```

Listing 2: Algorithm for hand-written resilient Java code

The code required to run the experiment for these technologies was in the order of 100 lines of code for both RestMule and GAJ (88 and 139 LOC repsectively), much lower than the orignal code of 550 LOC. As seen in Listings 1 and 2, a query expressed in a single line in RestMule will require a much more verbose algorithm when written in a generic non-resilient manner. It is worth noting that extending the experiment to run on all 18 technologies in both cases would not take more than a couple of extra LOC.

Both tools produced the same results for Eugenia and GMF, whilst GAJ did not terminate for Sirius in our tests[8]). As both tools offer similar capabilities and we obtained the same results after running this case-study, we have gained confidence that the code generated by RestMule (for GitHub) is as good as the hand-written code used by GAJ, and they are both superior to the original experiment's code that was tailored to a specific use-case and much more verbose. Since RestMule is a generic resilient client generation tool designed for any repository with an available OpenAPI Specification, these results are promising.

Threats to Validity. Although the generated client has been built to enable resilience against request restrictions and network failure, a more extensive set of unfavorable scenarios, e.g. contemplating remote API exceptions, has to be considered.

Several remote APIs, such as GitHub, StackExchange, and Bugzilla, have been explored to identify their commonality, yet the results of our evaluation are limited to generating and using the GitHub remote API. Further investigation is required to gain confidence that RestMule can successfully generate clients for other technologies and offer the same resilience as the GitHub client.

4 RELATED WORK

This section presents related work within the MSR research community. Although research using data from collaborative development environments such as (*i*) source control management systems (e.g. GitHub, SourceForge), (*ii*) bug tracking systems (e.g. JIRA, Bugzilla), and (*iii*) archived project communications (e.g. Eclipse Forums, StackOverflow) is relevant to MSR research, we focus on existing work that employs them during the construction of data sets and analysis tools and, in particular, highlight differences to RestMule.

Massive Datasets. Intense data collection tools usually involve mirroring the server contents into massive public datasets. These datasets are built either in a non-evolving fashion, such as the Maven Repository Dataset, or an evolving fashion, such as Stack Exchange's Data Dump and GHTorrent. Non evolving datasets hold information based on a single snapshot of a software repository while evolving datasets usually follow a schedule to make their updated versions publicly available. Consequently, at the time their information is queried, both evolving and non-evolving datasets may be inconsistent when compared to their origin.

GHTorrent and the Maven Repository Dataset use complex infrastructures to retrieve, store and share their data. Their public availability relies on the benevolence of its maintainers [2], which is associated with risk from data stagnation and corruption. For example, both datasets reported data corruption issues that may be a result of data processing tasks, such as cleansing, abstraction, transformation, or a collection thereof. As opposed to dealing with the entire contents of remote software repositories, RestMule processes near real-time data required to satisfy a given user query.

The restriction imposed on the number of requests that can be issued within a particular timespan to a remote API, such as offered by GitHub, has been presented as a major challenge for data collection in the MSR research community and often motivates the use of mirroring datasets. To our knowledge, this restriction has only been addressed by GHTorrent and our framework. GHTorrent manages this restriction by using their collection of user access-tokens[9] and dispatching requests with different user accounts, which are persisted in a shared database, and consequently achieve an increased request rate limit. Currently, RestMule handles request limitations based on a single account. However, our intention is to provide support for collaborative and distributed queries in the future.

Analysis Frameworks. In general, existing frameworks for mining information from software repositories differ based on their orientation, either metric-oriented or workflow-oriented. Metric-oriented frameworks, such as OSSMETER [1] and RepoGrams [13], focus on collecting data for producing metrics, such as software quality, static source code, and changes. Workflow-oriented frameworks like SmartSHARK [14], CODEMINE [3], and BOA [4], aim to provide a shared environment for data analysis purposes. RestMule can act as a complementary component, which can help with the activity of data collection from remote APIs in such systems.

5 CONCLUSIONS AND FUTURE WORK

Massive data collection has proven to be a challenge in the MSR community, although not exclusively. In order to mine data from remote software repositories, applications may perform significant numbers of requests, commonly to public HTTP APIs. Client applications can be blocked by strategies implemented by API providers to protect their servers from saturation. Mining applications tend

[8]The raw data obtained from these experiments can be found at: https://doi.org/10.6084/m9.figshare.5840991

[9]Built with voluntarily offered access-tokens from GitHub user accounts

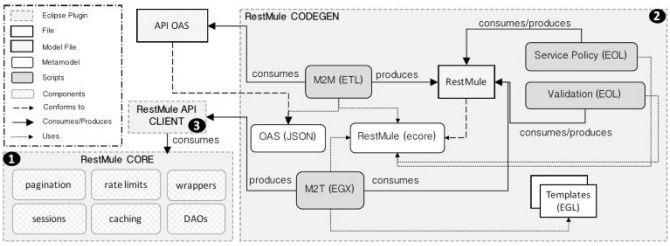

Figure 1: RestMule Architecture

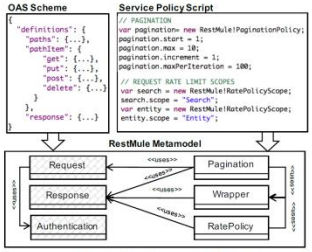

Figure 2: Simplified *RestMule Metamodel*

data set is needed. RestMule uses the abstractions provided by the OkHttp library[4], which deals with asynchronous response types.

Sessions. They are currently used within the HTTP interceptors context to update the request rates available to the user every time a new response is received from the Internet; this verifies that a given session has a non empty request allowance before dispatching them to the network. The relevant generated API-specific classes provide methods to create sessions based on the supported authentication schemes offered by the service provider. These methods return a session interface which can be used to access its request limits.

Inner Clients and Main Entrypoint. The RestMule API client layer exposes API calls that hide pagination and request limit handling to the end-user and return Data Access Objects. This is achieved by delegating the API calls to inner clients that use different request dispatcher engines to deal with specific request-limit policies regardless of the authentication scheme used to identify the user. Based on the pagination policy employed by the service-provider (e.g. page numbered, limit-offset), the inner clients know how to traverse the pages of a multi-part response from the server. The request dispatcher engine will monitor the request allowance based on the user session and await for allowances to be refilled before sending further requests to the network.

The main entrypoint for the user is a facade to the exposed methods of the multiple inner clients; it is responsible for instantiating the inner clients based on the configuration that the user passes to its builder (e.g. allow caching, user session).

Caching. This is defined in the core layer; API-specific extensions are generated which define how to load and put index entries that represent HTTP responses. Since the OkHttp library used by RestMule offers reliable caching capabilities, these are the ones currently used by the system. If more fine-grained control of the cache is desired though, RestMule allows for the incorporation of a custom manager instead.

3 EVALUATION

In order to gain confidence that RestMule is able to offer its intended capabilities, a two-way evaluation has been performed between RestMule and a popular Java-based GitHub API client[5], i.e., referred to as GAJ in the sequel, using the methodology of a published case-study. This evaluation aims at reproducing the experiment methodology published in previous work that assessed the use of MDE technologies [9], using RestMule. In that work, hand-written imperative code in a Javascript-like language was used to obtain data from GitHub regarding the use of 18 different model-driven engineering technologies. The resulting data comprised (*i*) the number of repositories using these technologies, (*ii*) the number of files in those repositories that contained code written in the relevant MDE language, and (*iii*) the commits and authors of those files.

Method. To evaluate the functionality of RestMule, we generated a client for GitHub's HTTP API v3[6] from an unofficial OpenAPI Specification[7] in JSON. The resilient Java client is generated with the aid of the *RestMule* Metamodel as well as a service policy script that captures restrictions imposed on requests and pagination.

To reproduce the analysis workflow used to assess MDE techologies ([9]), an initial search on GitHub is performed looking for all files that, for a given MDE technology, contain a specific keyword and have a specific file extension that identifies a technology. This query is repeated for each of the 18 technologies in question. Since GitHub imposes a limit of 1000 results regardless of the type of query, and since such queries may return more than this number, the following workaround was used in that study: For each of the files returned, a new search is performed to access the repository of the file, then a new query is used to retrieve all files that contain the keyword and file extension on that repository. This is in hope that more relevant files can be retrieved than in the initial limited search query. From this new set of files, other information is extracted like (1) the number of commits a file has been involved in and (*ii*) the number of authors that have written these commits. With this information it is possible to generate statistics like total number of repositories, total number of files, estimated number of developers (based on commit authors), etc.

This evaluation considers (*i*) asynchronous page traversal, i.e., single multi-paged requests, (*ii*) variable request policies, i.e., for groups of API endpoints, (*iii*) response awaiting when blocked, i.e., consecutive queries that exceed request limits, (*iv*) response caching, i.e., for repeated queries, and (*v*) network failure, i.e., await for reconnection.

Results. For both tools, the experiment ran on a quad core i5-4670k CPU @ 3.40 GHz, with 32GB of RAM and an SSD hard-disk. The JVM was provided with up to 5GB of memory and ran Java 8 on

[4] https://github.com/square/okhttp
[5] https://github.com/kohsuke/github-api
[6] https://developer.github.com/v3/
[7] https://api.apis.guru/v2/specs/github.com/v3/swagger.json

to re-implement from scratch data collection infrastructures due to the lack of reusable frameworks able to deal with these restrictions.

We have presented RestMule, a framework that comprises a set of reusable facilities that handle request rate quota, network failures, caching and asynchronous paging in a transparent manner; providing a code generator that produces API-specific Java libraries from OpenAPI specifications, that employ those facilities. RestMule allows developers to produce resilient remote-API clients enabling them to focus on their core mining and analysis logic.

Initial evaluation demonstrates that such resilient clients can be used to express queries in a much more concise manner, whilst providing the resilience that would be necessary to execute them, which would otherwise have to be manually implemented. As such, we encourage HTTP service-providers to publish OpenAPI specifications to ease service consumption and enable client code generation.

Future Work. In terms of future work, we plan to develop more sophisticated approaches to handle request rate policies. These approaches include adding support for collaborative clients (multi-user workflows) —in a similar fashion to GHTorrent [5]— and enabling (shared) distributed analysis workflows.

Furthermore we plan to enable the extension of the resilience strategies with user-defined policies.

ACKNOWLEDGMENTS

The work in this paper was supported by the Mexican National Council for Science and Technology (CONACyT) under Grant No.: 602430/440678 and the European Commission via the CROSSMINER Project (732223).

REFERENCES

[1] Bruno Almeida, Sophia Ananiadou, Alessandra Bagnato, Alberto Berreteaga Barbero, Juri Di Rocco, Davide Di Ruscio, Dimitrios S Kolovos, Ioannis Korkontzelos, Scott Hansen, Pedro Maló, et al. 2015. OSSMETER: Automated Measurement and Analysis of Open Source Software.. In *STAF Projects Showcase*. 36–43.

[2] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. 2012. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. IEEE Press, 26–30.

[3] Jacek Czerwonka, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. 2013. Codemine: Building a software development data analytics platform at microsoft. *IEEE software* 30, 4 (2013), 64–71.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2015. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 7.

[5] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 12–21.

[6] Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance* 19, 2 (2007), 77–131. https://doi.org/10.1002/smr.344

[7] Nafiseh Kahani, Mojtaba Bagherzadeh, Juergen Dingel, and James R Cordy. 2016. The problems with Eclipse modeling tools: a topic analysis of Eclipse forums. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 227–237.

[8] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, and Richard Paige. 2016. The epsilon book. (2016).

[9] 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.. In *OSS4MDE@ MoDELS*. 20–29.

[10] openapi:online 2017. StackExchange API version update [Online]. (2017). Available at: https://github.com/APIs-guru/openapi-directory/issues/217 [Accessed: May 18, 2017].

[11] Cesare Pautasso. 2014. RESTful web services: principles, patterns, emerging technologies. In *Web Services Foundations*. Springer, 31–51.

[12] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 221–224.

[13] Daniel Rozenberg, Ivan Beschastnikh, Fabian Kosmale, Valerie Poser, Heiko Becker, Marc Palyart, and Gail C Murphy. 2016. Comparing repositories visually with repograms. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 109–120.

[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. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 97–108.

B.2: CROSSFLOW

CROSSFLOW: A Framework for Distributed Mining of Software Repositories

Dimitris Kolovos*, Patrick Neubauer*, Konstantinos Barmpis*, Nicholas Matragkas*, Richard Paige*†
†Department of Computing and Software, McMaster University, Canada
*Department of Computer Science, University of York, United Kingdom
{firstname.lastname}@york.ac.uk

***Abstract*—Large-scale software repository mining typically requires substantial storage and computational resources, and often involves a large number of calls to (rate-limited) APIs such as those of GitHub and StackOverflow. This creates a growing need for distributed execution of repository mining programs to which remote collaborators can contribute computational and storage resources, as well as API quotas (ideally without sharing API access tokens or credentials). In this paper we introduce CROSSFLOW, a novel framework for building distributed repository mining programs. We demonstrate how CROSSFLOW can delegate mining jobs to remote workers and cache their results, and how workers can implement advanced behaviour such as load balancing and rejecting jobs they cannot perform (e.g. due to lack of space, credentials for a specific API).**

***Index Terms*—Open source software, Public domain software, Data collection, Data integration, Data analysis, Pipeline processing, Data flow computing, Software engineering, Client-server systems, Computer aided software engineering, Modeling, Scalability, Distributed processing.**

I. INTRODUCTION

In [1], we set out to assess the "popularity" of 22 different model-driven engineering technologies by measuring their use in open-source GitHub repositories. To achieve this we started by querying GitHub for files with relevant extensions and keywords. To expand our search and retrieve as many files as possible, we collected all repositories of the files returned by the first round of searches (1,900+ repositories overall), and we then queried every repository again (through the GitHub API) for files of all technologies of interest. This step alone required $22 \times 1,900$ GitHub API calls, which vastly exceeded the 5,000 calls rate limit per hour imposed by GitHub. Given that our team comprised several collaborators, one way around this limit would have been for each collaborator to generate a GitHub OAuth personal access token and to then collect these tokens in one machine and rotate among them. As not everyone felt comfortable with this option, we decided to use one GitHub account and make our data collection program wait for the API rate limit to be replenished, which led to an overall execution time of more than 8 hours.

This experience motivated us to investigate approaches for distributed execution of software repository mining programs that would allow remote collaborators to contribute their API call allowances and computational resources, without having to share credentials or pre-authorised OAuth keys. This work resulted in the development of CROSSFLOW, a Java-based framework for development and distributed execution of multi-step software repository mining programs (workflows). Before resorting to implementing a new distributed execution framework, we attempted to build on top of existing frameworks such as Apache Spark and Flink, but they were not able to accommodate our locality scheduling requirements, discussed in Section II-F, and do not provide sufficiently fine-grained job-result caching facilities.

The rest of the paper is organised as follows. Section II, presents the architecture of CROSSFLOW, the domain-specific language it uses for specifying software repository mining workflows, as well as key features such as caching and locality scheduling. Section III presents related work and section IV concludes the paper and discusses directions for future work.

II. CROSSFLOW

CROSSFLOW is a novel distributed data processing framework tailored to the needs of collaborative repository mining. CROSSFLOW supports distributing tasks of multi-step repository mining programs, which we call *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 (Apache ActiveMQ [2] in our current implementation). We explain the building blocks and facilities of CROSSFLOW through a running example.

In this example we wish to discover the degree to which different technologies (e.g. programming languages, tools) are used together in the same GitHub repositories. For instance, we wish to discover if projects using the Eclipse Graphical Modelling Framework (GMF)[1] are more likely to also use the ATL [3] or the QVTo[2] 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 (technologies.csv) in Table I, captures a known file extension and keyword for each technology of interest;
- Query GitHub to find repositories containing files of interest for each technology (the GitHub search API returns the details of up to 1000 files for each search);
- Clone the repositories of all collected files and search the local clones for files of all technologies of interest;
- Compute a co-occurrence matrix like the one in Table II.

[1]https://www.eclipse.org/gmf-tooling/
[2]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

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

TABLE I: Technologies with file extensions and keywords

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

TABLE II: Technology co-occurrence matrix

A. Architecture

As illustrated in Figure 1, CROSSFLOW provides a purpose-built domain-specific language for modelling repository mining workflows, and a code generator that produces implementation scaffolding in Java, which depends on a reusable runtime library. 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 behaviour of the modelled sources, tasks and sinks. This is expressed using hand-written Java code embedded in the generated scaffolding. Once the desirable behaviour has been implemented, the compiled workflow-specific code and the reusable core runtime library are bundled in a self-contained runable JAR file, which is executed on the nodes participating in the execution of the workflow. The following sections discuss the components of the CROSSFLOW architecture in detail.

B. CROSSFLOW *DSL*

The CROSSFLOW DSL has been implemented on top of the Eclipse Modelling Framework and its abstract syntax (metamodel) is illustrated in Figure 2. We explain its building blocks using a 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 like Table I, producing *Technology* jobs, consisting

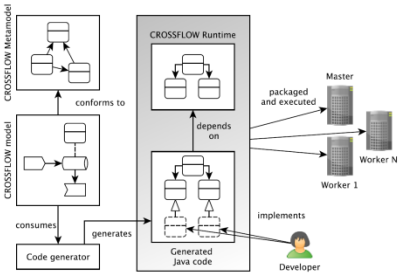

Fig. 1: CROSSFLOW Architecture

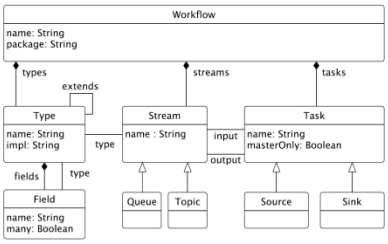

Fig. 2: CROSSFLOW Language Metamodel

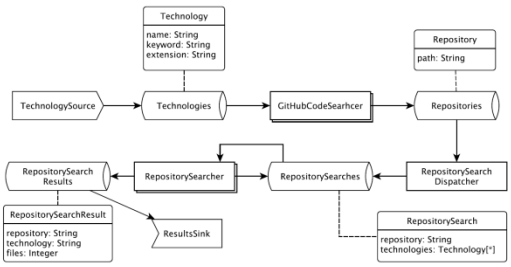

Fig. 3: CROSSFLOW Model of the Running Example

of a name, an extension and a keyword, into the *Technologies* stream. Sources are only executed on the master node.
Streams are message channels to which sources and tasks of the workflow can send jobs that other tasks can perform, 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 send 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 (and more importantly, its own rate limit);
- 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 streams and receive results to aggregate/persist. For example, the *ResultsSink* sink in the example, collects *RepositorySearchResult*s, builds the co-occurrence matrix illustrated in Table II, and periodically persists it into another CSV file (results.csv). As with sources, sinks are only executed on the master node.

C. Code Generator

The CROSSFLOW code generator can consume a workflow model and produce strongly-typed scaffolding Java code. In particular:

- For every *Task* and *Sink* in the model, it produces an abstract base class, as well as a skeleton subclass which 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 the generator produces an abstract *GitHubCodeSearcherBase* class as well as a concrete *GitHubCodeSearcher* class that extends the base class and contains an empty implementation of a *void consumeTechnologies(Technology technology)* method, which is called by the workflow when a new *Technology* job is received. The hand-written body of the method is illustrated in Listing 1.
- Similarly, for every *Source* in the model, the generator produces an abstract base class with infrastructure-communicating code and an abstract *void produce()* method, as well as a concrete sub-class for developers to implement the latter and specify the behaviour of the source (e.g. in the case of *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 a *void sendToRepositories(Repository repository)* that is 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 Java class which contains code that subscribes instances of the concrete task classes to the underlying ActiveMQ topics/queues [2]. Unlike with tasks, sources and sinks, developers do not need to write additional code to specify the behaviour of streams.
- For every *Type* in the model that doesn't specify an *implClass* (i.e. is not a proxy for an existing Java class), the generator produces a Java 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 Java class is made to extend the built-in *Job* class, which provides ID/correlation ID fields as well as serialisation capabilities which are required for caching as discussed below.

```
1  @Override
2  public void consumeTechnologies(Technology t)
       throws Exception {
3    List<String> paths = ...; // runs GitHub search
4    for (String path : paths) {
5      Repository r = new Repository();
6      r.path = path;
7      r.corellationId = t.id;
8      sendToRepositories(r); } }
```

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

The generator also produces a main Java 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 IP address.
- For nodes in worker mode, relevant parameters can define the 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.

D. Worker Error Handling

Exceptions produced during the execution of hand-written code in *consumeXYZ(...)/produce(...)* methods[3], such as *consumeTechnologies(...)*, in workers are caught by the generated base classes and sent to a dedicated stream (*InternalExceptions*) together with the job that caused them. In the current version of CROSSFLOW, jobs that cause exceptions during their execution are not rescheduled, they remain in the *InternalExceptions* stream for inspection by developers. 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.

E. 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) ID and an optional correlation ID which records the ID of the job of which it

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

is an output. For example, Listing 1 shows a redacted version of the implementation of the *consumeTechnologies(...)* method of the *GitHubCodeSearcher* class, where outgoing *Repository* jobs are associated to the incoming *Technology* by setting the correlation ID of the former to the ID of the latter (line 7).

The master node intercepts jobs submitted to all streams and caches outputs against their respective inputs based on IDs and correlation IDs so that previously-seen jobs are not re-executed in subsequent runs of the workflow, but instead their cached outputs are re-used.

F. Locality Scheduling

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 in the technologies.csv input), *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 powerful approach which eliminates the need for centralised book-keeping by enabling CROSSFLOW worker tasks to reject jobs allocated to them. Using this feature, we can achieve the desired locality scheduling in *RepositorySearcher* as follows:

- *RepositorySearcher* receives a *Repository* to analyse
- If the worker has a clone of the repository in question, it accepts and performs the job
- If it does not have a clone of the repository:
  - If it is the first time the worker encounters this job it adds the ID of the job to a list of encountered jobs and rejects the job
  - If the ID of the job is already in the worker's encountered job list upon reception, it assumes that all other nodes have previously 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 once (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. The described

III. Related Work

Boa [4] is a domain-specific language and infrastructure for mining software repositories. Boa's infrastructure leverages distributed computing techniques to execute queries against a multitude of software projects. The Boa language enables the specification of analysis of Git repositories, but it does not allow the specification of more complex workflows, such as retrieving and combining data from two different sources.

Another tool for distributed mining of software repositories is King Arthur[4], which is part of the GrimoireLab[5] tool chain. King Arthur is a distributed job queue platform that schedules and executes data retrieval jobs from software repositories using Perceval [5], a dedicated Python library. This platform enables the orchestration and distribution of data retrieval jobs only, while CROSSFLOW enables the distribution of 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.

Finally, Boinc [6] is a software system that facilitates the creation and execution of public-resource computing projects and therefore it can 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 any high-level, declarative way to specify workflows.

IV. Conclusions and Future Work

This paper introduced CROSSFLOW, a novel framework for development and distributed execution of multi-step repository mining programs. CROSSFLOW provides a domain-specific language for designing distributed workflows as well as a code-generator that produces implementation scaffolding for developers to complement with hand-written Java code. CROSSFLOW uses asynchronous message-based communication and provides built-in support for job-level caching and locality scheduling.

We are currently working on the implementation of concrete workflows for the needs of our industry partners in the collaborative project supporting the development of CROSSFLOW, and extending the framework with new facilities in the process.

[4]https://github.com/chaoss/grimoirelab-kingarthur
[5]https://chaoss.github.io/grimoirelab/

REFERENCES

[1] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige, "Assessing the use of eclipse mde technologies in open-source software projects." in *OSS4MDE@ MoDELS*, 2015, pp. 20–29.

[2] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning Greenwich Conn., 2011, vol. 47.

[3] Frédéric Jouault and Ivan Kurtev, "Transforming Models with the ATL," in *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, ser. LNCS, Jean-Michel Bruel, Ed., vol. 3844, Montego Bay, Jamaica, October 2005, pp. 128–138.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 422–431. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606588

[5] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 1–4.

[6] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004, pp. 4–10.

[7] A. Bagnato, K. Barmpis, N. Bessis, L. A. Cabrera-Diego, J. D. Rocco, D. D. Ruscio, T. Gergely, S. Hansen, D. S. Kolovos, P. Krief, I. Korkontzelos, S. Laurière, J. M. L. de la Fuente, P. Maló, R. F. Paige, D. Spinellis, C. Thomas, and J. J. Vinju, "Developer-Centric Knowledge Mining from Large Open-Source Software Repositories (CROSSMINER)," in *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, 2017, pp. 375–384. [Online]. Available: https://doi.org/10.1007/978-3-319-74730-9_33

[8] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Eclipse development tools for epsilon," in *Eclipse Summit Europe, Eclipse Modeling Symposium*, vol. 20062, 2006, p. 200.

[9] S. Madani and D. S. Kolovos, "Re-Implementing Apache Thrift using Model-Driven Engineering Technologies: An Experience Report," in *Proceedings of the 16th International Workshop on OCL and Textual Modelling co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016.*, 2016, pp. 149–156. [Online]. Available: http://ceur-ws.org/Vol-1756/paper11.pdf

[10] D. S. Kolovos and R. F. Paige, "Towards a modular and flexible human-usable textual syntax for EMF models," in *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018.*, 2018, pp. 223–232.

APPENDIX

The source code of the tool is currently available on a public repository on GitHub[6]. The tool is developed in the context of a large collaborative research project, Crossminer, involving several industrial and academic partners [7]. All of them have a vested interest in continuing the development and maintenance of the tool in the future, and thus the tool will remain available in the future on GitHub. Currently, the tool is build on Eclipse Epsilon [8], Apache Tomcat, Apache ActiveMQ [2], and Apache Thrift [9].

There are two use cases for the tool: a new mining workflow is specified and then executed; an existing workflow is executed. Detailed instructions on how to execute the tool are provided in the repository. A brief version is provided below.

Specify new workflow: To specify a new workflow, a user needs an Eclipse Epsilon distribution[7], Apache Tomcat[8], ActiveMQ[9] and Thrift[10]. Instructions on how to install these are provided on our repository. Once these dependencies are installed, the user will need to clone our repository and import the projects in Eclipse. Our tool uses Apache Ivy[11] for dependency management, so once the projects are imported to Eclipse all dependencies should be resolved automatically. To specify the new workflow, the user has to create a new Eclipse project and specify the workflow in an XML-based, human-friendly notation called Flexmi [10]. Once the workflow is specified, the code generator can be invoked to generate the code skeleton. Developers then need to extend the generated source, task and sink classes with the desirable behaviour. Once the code is implemented and compiled, it needs to be bundled into a runnable JAR which can then be executed as a standard Java application on the master and worker nodes.

Execute workflow: To execute a workflow, the user simply has to execute the runnable JAR produced in the previous step on the command line. The JAR has to be executed on every machine, which wants to contribute computational resources to the analysis of the workflow. The user can specify the name of the workflow, the type of the node (master or worker), the IP address of the master node, and the listening port of the master node by means of the parameters *name*, *mode*, *master*, and *port*, respectively.

The current repository of the tool comes with detailed instructions on how to perform the above process[12]. Moreover, the repository contains predefined workflows that users can execute in order to experiment with the platform. The repository contains the model and the code for the examples, as well as how to generate JARs in order to execute them.

[6] https://github.com/crossminer/scava/tree/crossflow/crossflow

[7] https://www.eclipse.org/epsilon/download/

[8] http://tomcat.apache.org

[9] http://activemq.apache.org

[10] https://thrift.apache.org

[11] http://ant.apache.org/ivy/

[12] https://github.com/crossminer/scava/tree/crossflow/crossflow/README.md