Locality-Aware Scheduling of Software Repository Mining Workflows in Heterogeneous Environments

Ana Marković

Doctor of Philosophy

University of York
Computer Science
January 2025

Abstract

This thesis investigates advancements in job scheduling within heterogeneous computing environments, focusing particularly on the domains of software repository analysis. Two main contributions are presented: the Bidding Scheduler and the self-correcting worker mechanism, denoted as the Feedback-Based Estimator.

Firstly, the Bidding Scheduler is introduced as a novel approach that decentralises the job allocation process. In this model, worker nodes engage in a bidding process to acquire jobs, leveraging localised data and workload conditions to submit bids that reflect expected processing times. The master node collects these bids and allocates jobs in a manner that minimises overall execution time and optimises resource utilisation. This approach addresses the common inefficiencies in traditional master-worker architectures by integrating both data locality and worker heterogeneity into the decision-making process. Experimental work demonstrates the scheduler's ability to adapt to dynamic workloads and varying node capacities, resulting in improved performance metrics.

Secondly, the Feedback-Based Estimator mechanism is developed to enhance the accuracy of job completion time estimates. Acknowledging the limitations in developers' ability to predict precise processing times due to the variability in job attributes and system performance, this mechanism introduces a real-time adjustment process. Based on continuous feedback from executed jobs, the Feedback-Based Estimator dynamically updates the estimation formula to correct any persistent discrepancies between estimated and actual job completion times, thereby reducing the margin of error in time estimates.

Together, these contributions provide a framework for efficient and adaptive job scheduling in heterogeneous computational environments. Future work will focus on investigating applicability of these approaches beyond mining software repositories.

Contents

List of Figures

List of Tables

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Some of the material presented in this thesis has previously been published in the following papers:

- A. Markovic, D. Kolovos, and L. Soares Indrusiak, *Distributed data locality-aware job allocation*, Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23), Denver, CO, USA, pp. 2089–2096, ACM, 2023.
  DOI: 10.1145/3624062.3624287

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dr Dimitris Kolovos and Dr Leandro Soares Indrusiak, for their continuous guidance, support, and encouragement throughout this research journey.

A special thanks goes to Nemanja, for his unwavering support, for the many brainstorming sessions when I felt stuck, and for generously sharing his time and resources to help this research move forward when it mattered most.

1 Introduction

Mining software repositories is a fundamental endeavour for extracting valuable insights that inform software engineering practices, quality assurance, and project management. The principal objective of this research is to enhance the performance of repository mining workflows by leveraging distributed computation techniques. Specifically, we aim to distribute the workload across multiple machines, termed "workers", to optimise execution time and utilisation of computational resources.

1.1 Aim of Research

The primary aim of this research is to improve the performance of repository mining workflows by distributing the tasks across multiple machines, each with varying capabilities such as processing power and storage capacity. This distribution must be both efficient and effective to address the challenges of cloning and storing potentially large repositories, as well as meeting the computational requirements of analysing the source code. Central to this study is the concept of Mining Software Repository (MSR) workflows, which are defined formally below to ensure clarity and eliminate any ambiguity.

Definition of MSR workflows

In the context of this thesis, an MSR workflow refers to a structured set of predefined computational operations conducted on software repositories. These workflows are tailored specifically for mining repositories — that is, extracting meaningful information such as commit history, code structure, revision data, or other metrics — from repositories that could reside in remote environments. Unlike traditional workflow languages such as BPMN[5], which are designed around explicit representations of control flow (e.g., branches, loops, and synchronisation), MSR workflows in this thesis are implemented as task-oriented pipelines. These pipelines can consist of any number of computational steps, and while the examples used in this work are linear, the workflow abstraction itself does not preclude conditional or repeated steps. The defining aspects of an MSR workflow are outlined below:

- **Task decomposition.** An MSR workflow consists of discrete, atomic tasks that align largely with data-centric operations found in paradigms such as map-reduce [6]. These tasks may include operations such as parsing individual files, analysing commit messages, or examining repository structures. Tasks are typically designed to minimise interdependencies, ensuring that parallel execution is feasible across multiple worker nodes.

- **Step execution.** MSR workflows follow a step-based structure. Typically, a workflow consists of:

  - **Source step:** A single initial step responsible for initiating the workflow by retrieving the list of target repositories from remote locations.
  - **Operational steps:** A series of computational steps, often executed in parallel, which perform analyses of the cloned repositories. These may include parsing, metric extraction, or historical commit analysis.
  - **Sink steps:** One or more final steps designed to aggregate results from operational steps and save them in a structured, machine-readable format (e.g., JSON or CSV).

  The decomposition of MSR workflows into atomic tasks parallels the principles of the map-reduce computational paradigm. The 'map' phase comprises distributed tasks, such as parsing individual repository files or analysing commit-level data in parallel across worker nodes. The 'reduce' phase corresponds to the sink steps, where outputs from multiple tasks are aggregated, processed, and stored in a machine-readable format. The relationships between these steps are predominantly sequential or loosely coupled, with interdependencies being highly data-driven. For example, during the workflow execution, the output of the source step — a list of repository URLs — is used as the input for operational steps, wherein repositories are cloned to worker nodes for further analysis. Once an individual repository is processed, key metrics (e.g., commit history data or extracted features) are passed to the sink step for aggregation.

- **Transmission costs.** Although interim result transmission occurs between source, operational and sink steps, it is assumed to be negligible compared to the cloning of repositories or the computational overhead of code analysis. This assumption is based on the observation that the interim data, consisting of structured summaries (e.g., extracted metrics or historical data), is far smaller in volume than the original cloned repositories. This assumption simplifies the analytical model while focusing computational optimisations on tasks with substantial resource impacts.

For the scope of this study, the repository mining workflows of interest possess the following distinct characteristics:

- **Remote repositories:** The repositories to be mined are not locally available and must be cloned over the Internet. This introduces significant overhead due to network latency and bandwidth constraints.
- **Multiple analyses for each repository:** Each repository needs to be analysed multiple times during the execution of the workflow. This could entail examining different points in the repository's history or extracting diverse types of data from the same repository.
- **Heterogeneous workers:** The computational environment comprises several workers, each with distinct capabilities. These differences may include varying CPU power, memory size, storage capacity, and network

bandwidth. Effective task distribution must therefore account for this heterogeneity in order to maximise overall performance while minimising idle states and bottlenecks across the system.

1.2 Motivating Example

In the software ecosystem, obtaining comprehensive insights into software systems is highly important [7]. Analysing software repositories governed by version control systems (VCSs) such as Git emerges as a pivotal method in obtaining these crucial insights. These repositories serve as reservoirs of valuable historical information, encapsulating a project's status, development progress, and evolutionary trajectory [8]. As the field of Mining Software Repositories (MSR) continues to expand, the MSR researchers aim to extract and analyse the heterogeneous data housed within software repositories to reveal actionable and insightful intelligence about software systems [9]. For instance, these analyses can encompass a range of activities, from calculating code metrics to identifying specific code patterns or anti-patterns.

Consider, for instance, the objective of measuring how a set of GitHub repositories have grown in lines of code over time. Such an analysis entails several steps, all depicted in Figure 1:

- **Cloning the repositories of interest:** This involves using the *git clone* command to download the repository to a local machine.
- **Checkout files for analysis:** This is accomplished by checking out each commit sequentially and ensures that each version of the files is accessible for detailed inspection.
- **Commit-by-commit analysis:** For each commit in every repository, the total lines of text in relevant code files are counted.
- **Data aggregation:** The collected data is aggregated to visualise the growth of the repositories across different time periods.

This seemingly straightforward process, however, presents numerous computational and logistical challenges when scaled to a large number of repositories and commits. Distributed execution is crucial for the effective analysis of multiple repositories and their commits due to several factors associated with such large-scale data operations. Firstly, the volume of data and the number of operations required for a comprehensive analysis can be overwhelming for a single machine. Processing thousands of commits across numerous repositories necessitates significant computational power and storage capacity, which can easily surpass the capabilities of a single node. Distributed execution allows for workload partitioning across multiple machines, thereby leveraging their combined computational and storage resources to handle large datasets efficiently.

Furthermore, the time complexity associated with analysing each commit in a repository can be substantially high. Each commit may involve parsing code,

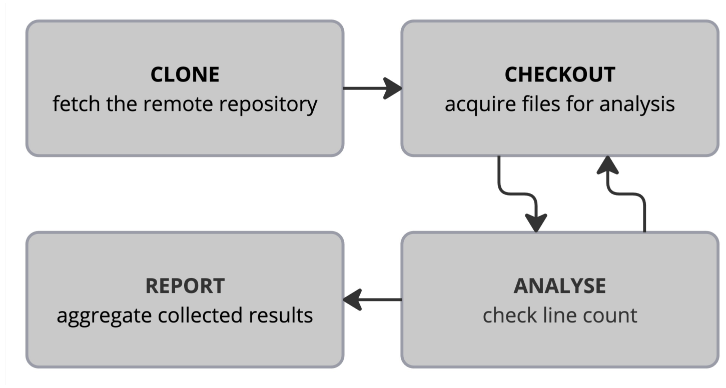

CLONE
fetch the remote repository
CHECKOUT
acquire files for analysis
REPORT
aggregate collected results
ANALYSE
check line count

Figure 1: MSR workflow for measuring line count growth in GitHub repositories

identifying changes, and computing various metrics, which are computationally intensive tasks. Consequently, executing these operations sequentially on a single machine could result in prohibitive processing times. Distributed execution mitigates this issue by enabling parallel processing of commits across different nodes. This parallelism significantly reduces the overall execution time, making it feasible to perform detailed analysis within reasonable timeframes. Moreover, by distributing the workload, the system can ensure higher throughput and better utilisation of resources, thereby optimising the overall performance of the analysis workflow.

In terms of work allocation, assigning one repository to each worker might initially seem efficient. However, this allocation strategy is suboptimal due to potential workload imbalances. Some workers might complete their tasks faster than others, depending on the size and complexity of the repositories they are processing. Consequently, this could result in idle workers while there are still pending repositories/commits to be analysed, thus leading to inefficient resource utilisation. Alternatively, naively allocating {*repository, commit*} jobs to workers in a round-robin manner could also be suboptimal. This strategy might result in multiple workers redundantly cloning the same repository multiple times, leading to unnecessary data transfer overheads and storage consumption. Such inefficiencies can significantly slow down the entire analysis process, especially when dealing with large and complex repositories.

Hence, in order to efficiently tackle execution of MSR workflows that demonstrate the properties discussed in Section 1.1, there is a need for a more sophisticated approach. This approach would involve striking the right balance between two key considerations:

- **Data locality optimisation:** Minimise the number of redundant repository clones by ensuring that each repository is cloned as few times as needed. This can be achieved by employing a job scheduling strategy that takes into account the current state of each worker and assigns them jobs in a way that reduces data redundancy.

- **Maximised worker utilisation:** Ensure an even distribution of workload across all workers so that no worker remains idle while others are still processing.

1.3 Thesis Structure

This thesis is organised into eight chapters, each addressing specific components of the research on MSR with a focus on distributed data processing frameworks and job allocation concerning data locality. The structure is designed to provide a clear trajectory from background and hypotheses formulation through experimental validation to practical implications and future directions.

Chapter 2 constitutes a comprehensive literature review pertaining to MSR tools and distributed data processing frameworks. This section examines existing methodologies, tools, and frameworks. Key works in the domain are scrutinised to lay a solid foundation for the subsequent chapters. Chapter 3 analyses these frameworks and tools, comparing and contrasting them to identify gaps and opportunities for advancement. It also presents the core research hypotheses and defines specific research goals. Emphasis is placed on the criteria and metrics that will be utilised to evaluate whether the research objectives are met. In Chapter 4, we describe a series of experiments conducted to evaluate several potential baseline frameworks. These experiments are critical to selecting the most optimal framework that serves as the foundation for the research. The criteria for performance include computational efficiency, job scheduling and data locality handling.

The core contributions of this thesis are encapsulated in Chapters 5, 6, and 7. These chapters detail our advancements in distributed mining and job allocation strategies, elaborating on the methodologies developed to optimise overall execution time through tackling data locality. The final chapter, Chapter 8, synthesises the findings from the entire thesis, offering a summation of key insights and achievements. Additionally, it presents a forward-looking discussion on potential future research directions that stem from the present work.

In conclusion, this thesis is structured to provide a logical and in-depth examination of the chosen research areas, ensuring clarity and coherence in the presentation of findings and contributions.

2 Literature Review

This chapter presents literature related to software repository mining and distributed data processing. The investigation began by formally defining the scheduling problem for MSR workflows described in Section 1.1. It continued by evaluating various approaches and tools for mining repositories to ascertain their capabilities in addressing the use cases described in Section 1.2, with a primary focus on the suitability of these tools for distributed execution, their scheduling mechanisms, and their efficacy in leveraging local repository copies. Subsequent to analysing tools specifically designed for MSR, our focus shifted to general-purpose data processing frameworks to assess their efficacy in executing workflows involving repository mining.

2.1 Characterising the Scheduling Problem in Mining Software Repositories

The central challenge in our MSR workflows lies in scheduling a stream of incoming jobs — each representing the analysis of a specific repository - across a fixed pool of distributed worker nodes. Each job must be assigned to a worker, where the corresponding repository is either cloned (if not already present) or reused from a prior analysis. Since cloning large repositories is time-consuming and computationally expensive, this setup phase introduces non-trivial overhead that must be managed effectively. This scenario naturally maps to a class of problems known as *Job Shop Scheduling with Sequence-Dependent Setup Times* (JSSP-STsd) [10], but with notable extensions.

In classical job shop scheduling formulations, each job consists of a sequence of operations that must be processed in a specific order on designated machines. Scheduling aims to optimise global objectives, such as minimising the total completion time (makespan), assuming that the set of jobs and their processing parameters are known in advance. When setup times are included, they typically depend on the transition from one job to another and are treated as fixed and deterministic (e.g., tool changes in manufacturing). These problems are commonly denoted in Graham's three-field notation as **J/STsd/Cmax** [11], where:

- **J** denotes a general job shop setting,
- **STsd** indicates sequence-dependent setup times,
- **Cmax** refers to the objective of minimising makespan.

In our case, however, the scheduling environment departs from this classical model in several critical ways:

- **Online scheduling:** Jobs arrive incrementally over time, one at a time, and must be scheduled immediately. This removes the possibility of computing globally optimal sequences and necessitates *online scheduling policies* that adapt to partial information.

- **Stateful setup cost:** Setup time is only incurred the *first* time a repository is scheduled on a given worker. If the worker has already cloned the repository, the setup time for subsequent jobs of the same repository is effectively zero. This represents a form of *state-aware setup*, which is not modelled in traditional JSSP-ST literature, where setup costs typically apply to every transition between job types.
- **Uncertain setup duration:** Repository clone times vary in practice, depending on repository size and network conditions. This introduces *variability and partial unpredictability* in setup times, diverging from the deterministic assumptions common in manufacturing-based scheduling models.

To formalise the setup behaviour, we define the setup time for job j_i on worker w as follows:

$$s_{i,w} = \begin{cases} 0, & \text{if repository } r_i \text{ is already cloned on worker } w \\ S_{r_i}, & \text{otherwise} \end{cases}$$

Here, S_{r_i} denotes the cloning time for repository r_i, which can vary based on repository size, its metadata and network conditions. From a scheduling theory perspective, this problem can be described using an extended Graham notation as:

J/STsd*/Cmax-online

Where:

- **J** indicates the job shop environment (multiple workers/machines).
- **STsd*** highlights a non-standard variant of sequence-dependent setup times where setup cost is *stateful* — i.e., it depends not only on the current job, but on persistent system state (repo clone presence).
- **Cmax-online** signifies that the objective is to minimise the makespan in an *online* scheduling context.

These distinctions position our problem within the class of *online job shop scheduling with stateful and non-recurring setup costs.* While there exists rich literature on JSSP with setup times (e.g., [12, 13]), most models assume deterministic, job-related setup costs and globally known job sequences — assumptions that are violated in our streaming analysis context.

To formally characterise this class of problems, we define a task as *locality-intensive* if the expected job setup time — specifically, the time spent retrieving remote data — is significantly larger than or in the same order of magnitude as the actual job processing time. That is, for a given job j_i, if $E[s_{i,w}]p_i$, where $s_{i,w}$ denotes the setup time on worker w and p_i the processing time, then we classify the task as locality-intensive. In our case, cloning large software repositories typically takes on the order of tens of seconds to minutes, whereas analysing a single commit for line counts may require only a fraction of that time.

Accordingly, we argue that existing job shop formulations are insufficient to capture the full complexity of our scenario. Our problem requires heuristics and dispatching strategies that (1) exploit local clone reuse to minimise redundant setup costs, (2) adapt to real-time system load and network variation, and (3) perform competitively in the absence of foresight. These requirements create an opportunity to contribute novel scheduling models and algorithms grounded in Job Shop theory, but tailored for dynamic, data-aware applications such as mining software repositories.

2.2 MSR Tools

In recent times, there has been an increase in the number of tools dedicated to the exploration and extraction of data from Git repositories and hosting services such as GitHub and GitLab. These tools have garnered considerable attention and interest as subjects of research. Despite the fact that multiple version control systems, such as Subversion and Mercurial, are well-suited for mining software data, Git stands out as the focal point of interest within this realm [14]. When confronted with the task of mining software repositories, researchers and practitioners face several choices. They can opt to leverage existing tools and frameworks that are specifically tailored for mining, incorporate their analyses into higher-level systems or infrastructures, or even develop custom-made tools that align with their precise requirements. The following section outlines existing MSR frameworks and libraries identified as commonly used according to a survey in [15] and work in [9], along with general-purpose frameworks classified as state-of-the-art as indicated by [16] that can be used for MSR workflows. Additionally, it examines their functionalities in relation to the MSR example provided in Section 1.2.

2.2.1 Single-Machine Tools for Mining Repositories

PyDriller is a Python framework within the software repository mining domain. It simplifies the data extraction process from Git repositories, enabling in-depth analysis of software projects by researchers and developers [9]. PyDriller facilitates the retrieval of information pertaining to commits, developers, files, and changes within a repository, thereby serving as a valuable tool for the examination of software engineering and project assessment.

The key functionalities provided by PyDriller include:

- **Commit analysis**: The feature comprehensively extracts commit details, encompassing commit messages, authors, dates, and modified files.

- **File analysis**: It analyses file modifications, detailing the count of added and deleted lines, alongside the types of alterations.

- **Developer analysis**: It tracks individual developer contributions and activity patterns.

- **Repository history**: This capability enables the traversal of a repository's history to study its evolution over time.

PyDriller is engineered with user-friendly design, integrating with prominent Python data analysis and visualisation libraries such as pandas [17] and matplotlib [18]. It is particularly beneficial for researchers conducting empirical studies on software development practices, for instance, in identifying correlations between code review length, comment quantity, and post-release bug occurrences. Moreover, developers can leverage PyDriller to comprehend and enhance their codebases, such as identifying files with extensive changes.

When executing Python code via PyDriller, the program operates exclusively in a single-threaded framework on a single machine. Consequently, all data mining and processing tasks are handled sequentially by one instance of the application on one machine. PyDriller inherently lacks the capability to distribute tasks across multiple machines or to process commits in parallel without external intervention. To achieve distributed execution across multiple cores or machines, a developer must manually configure parallelisation through additional programming efforts. This entails manually dividing the repository analysis into discrete segments, allowing for parallel analysis of different branches through separate scripts or processes. Despite this possibility, it is crucial to note that PyDriller itself does not inherently support distributed execution. Each individual process would operate its instance of PyDriller independently.

Conversely, PyRepositoryMiner, a more recent tool, offers capabilities akin to those of PyDriller but the main difference is that it enhances parallelisation by enabling multi-threaded environments for traversing and extracting data from Git repositories [19]. Nevertheless, neither of these tools address distributed processing of software repositories as they were inherently designed to run on a single machine.

RepoFS is a valuable tool that acts as a virtual filesystem linked to a Git repository [20]. The tool enables users to engage with the contents of a Git repository by performing common file system tasks like reading files, viewing directories, and navigating through the file structure. Importantly, users can do all this without requiring an in-depth understanding of Git's inner workings or commands.

A key feature of RepoFS is its mechanism of making the entire Git repository accessible through the filesystem, which uniquely provides directories for each revision. This feature eliminates the need for repetitive checkouts, streamlining the process of reviewing the repository's status at different time points. This functionality proves particularly beneficial for users examining changes and snapshots of the repository over various stages of its development.

Despite RepoFS offering a unique and valuable interface for interacting with Git repositories through a file system-like abstraction, it is not designed for distributed execution across multiple systems, hence it is also not addressing the challenges of pairing incoming queries with workers that possess the required data. Although RepoFS can indeed access remote repositories — repositories that are not stored locally but are accessible via the network — the execution and actual processing of repository data are confined to the local machine on which RepoFS is running.

Perceval [21] is an open-source utility and Python library designed to gather data from various platforms essential to software development processes, including version control systems, issue tracking systems, and code review systems. This tool can operate both as a stand-alone application and as a Python library. However, it is dedicated solely to data collection, necessitating the use of supplementary tools or additional effort for comprehensive analysis or visualisation. The data collected from a range of sources (such as Git repositories, issue trackers, and Continuous Integration (CI) tools) is stored locally in a JSON format, making it ready for further analysis. Perceval is primarily intended to run on a single machine, which, consequently, makes it unsuitable for workflows that require executing MSR tasks in a parallel fashion, such as the ones described in Section 1.2.

As previously established in Section 1.2, there is a necessity for a distributed execution, thus subsequent sections will explore frameworks that facilitate this capability, with a particular emphasis on the repository mining use case.

2.2.2 SmartSHARK

SmartSHARK [22] is a comprehensive software ecosystem designed for the extraction and analysis of data from software repositories. Its primary function is to collect data from varied sources and integrate this data into a unified database. Additionally, SmartSHARK offers tools and features that facilitate data enrichment, visualisation, and elementary analysis. However, its primary strength lies in data collection and preparation, with more advanced analytical tasks typically requiring the use of external tools or custom scripts.

Similar to Perceval, SmartSHARK is capable of collecting data from multiple sources, such as development history from version control systems (currently limited to Git) and issue tracking systems, including Jira, Bugzilla, and GitHub issues. The structured JSON format is employed for presenting the collected data, which ensures consistency once data from various sources is harmonised. For instance, SmartSHARK's data model for issue tracking is based on Jira; fields from other issue trackers are mapped to this model as closely as possible. The collected data is organised into JSON format and stored in MongoDB [23], which is suitable for managing large volumes of data using a document-based, schema-free approach with JSON-like representations.

Programs created with SmartSHARK libraries can be executed on a single machine or across a distributed cluster. While classified as a repository mining tool, SmartSHARK is more geared towards analysing data about the code rather than the code itself, making it less suitable for the workflows discussed in Section 1.2. This is due to its reliance on a centralised database, which stores extensive metadata about repositories but is of little benefit when analysing the source code worker nodes obtain through cloning.

2.2.3 World of Code

World of Code (WoC) [24] is an extensive, open-source platform intended for the mining, analysis, and examination of software project evolution. It systematically retrieves and indexes substantial data from various software repositories, including Git repositories across platforms such as GitHub, GitLab, and Bitbucket, aiming to facilitate empirical research in software engineering and related fields. The platform structures the data into a graph format, where nodes denote entities (e.g., files, commits, repositories, authors), and edges signify relationships (e.g., a file modified in a commit, a commit made by an author).

The dataset construction involves three principal stages:

- **Project Discovery:** Using search engines and search APIs to find repositories.
- **Project Retrieval:** Establishing a local copy of publicly available repositories on WoC servers.
- **Data Extraction:** Creating a dataset with information on files, commits, etc., ready for researchers to query.

WoC also provides several access libraries, such as Python, Shell, or Perl APIs, to facilitate querying the data. Typically, an application comprises a query or application layer and an API that communicates with the WoC servers housing the data. The dataset is maintained across six servers, which process the queries submitted via the WoC APIs.

Despite its distributed approach, WoC faces certain limitations. The dataset is predetermined and cannot be customised to meet the specific needs of individual researchers (e.g., querying private repositories). Furthermore, access to the execution environment is restricted to the servers hosting WoC, precluding the default execution of programs in cloud environments or private dedicated infrastructure.

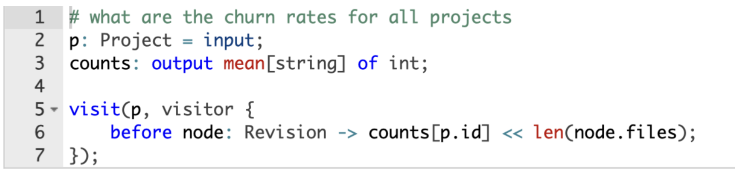
2.2.4 Boa

Boa is a specialised language and infrastructure designed for large-scale mining and analysis of software repositories [25]. It caters to users who may not have expertise in data mining or version control systems, simplifying the process of extracting and analysing data from large codebases and software repositories like those on GitHub. Boa offers a high-level abstraction that allows researchers and practitioners to specify data mining tasks without getting caught up in the intricacies of low-level data extraction and manipulation. This high-level language allows users to write queries that are specifically targeting common patterns and data types found in software development repositories, such as commits, files, ASTs (Abstract Syntax Trees), and metadata. The design of Boa's syntax is inspired by Sawzall, a procedural programming language designed by Google for processing large numbers of logs [26].

The primary goal of Boa is to ease the process of analysing ultra-large-scale software repositories, thus providing support for software engineering research activities, including software quality assessment and evolutionary studies. One example of an empirical study that can be conducted using Boa is determining the average number of changed files per revision, also known as the churn rate, across all projects [27]. Addressing this question typically requires a considerable amount of knowledge, including mining project metadata, understanding code repository locations, accessing the repositories, implementing additional filtering code, and managing controller logic.

However, using Boa makes this task much simpler. We start by declaring the input as a Project type and assign it an alias. Next, we define the output as accepting integer values and calculating the mean of all the integers it encounters, indexed by a string. Then, we iterate through the input data, including each code repository and its revisions. When we encounter a Revision, we send the number of files changed in that revision to the output variable, indexed by the project's ID. This computation yields the churn rate, represented by the mean value.

Figure 2 shows an example of Boa code illustrating this process.



```
# what are the churn rates for all projects
p: Project = input;
counts: output mean[string] of int;

visit(p, visitor {
    before node: Revision -> counts[p.id] << len(node.files);
});
```

Figure 2: Example of Boa code

Boa supports a couple of integrated development environments (IDEs), allowing users to write and manage Boa queries more efficiently [28]. Once a Boa program is written, it must be compiled into an intermediate Java program, similar to translating a high-level language into a lower-level format. The Boa source code is processed by the Boa compiler, which is implemented in Java. This compiler parses the Boa code and generates equivalent Java code, maintaining the same logic. Figure 3 below illustrates an example of Boa code alongside its generated Java counterpart.

Boa leverages a distributed computing infrastructure, based on the Apache Hadoop framework [29], to process large datasets across multiple computing nodes. Figure 4 describes the high-level architecture of the framework. Apart from the execution infrastructure, Boa also offers data infrastructure, through the use of a distributed file system, particularly HDFS (Hadoop Distributed File System) [30], to store both its input data (repositories) and the output from analyses.

The input data for Boa, which includes software repositories, is pre-processed and stored in a format suitable for efficient analysis, also known as cache. This data is typically organised and indexed to optimise common access patterns and queries. Hence, the generated code includes Hadoop's MapReduce functions,

```
counts: output int;
visit(input, visitor {
    before node: Commit -> {
        counts << 1;
    }
});
```

(a) Boa example

```
public class BoaGenerated {
    public static void main(String[] args) {
        int counts = 0;
        for (Commit commit : repository.getCommits()) {
            counts += 1;
        }
        System.out.println(counts);
    }
}
```

(b) Generated Java code from Boa

Figure 3: Boa code generation

such as mapping data to keys (e.g., project IDs), reducing data by keys (e.g., counting, summing data), and filtering data according to the developer's queries.

Boa offers an intuitive front-end, a web interface, through which users can compose and submit their programs. This interface significantly streamlines interactions with the underlying system, thus rendering the platform accessible to individuals who may not possess expertise in distributed systems or software repository mining. Upon submission of the Boa program via the web-based user interface, it is subjected to a compilation process and packaged into a JAR file, which is subsequently submitted to the Hadoop cluster for execution. During this phase, Boa's infrastructure is tasked with the scheduling and execution of the operations articulated by the program across the distributed infrastructure. It manages task distribution, resource allocation, and load balancing across the cluster. This stage is pivotal as it enables the distributed execution of the Boa program by harnessing the parallel computation capabilities inherent in the Hadoop ecosystem.

The same interface utilised for program submission is employed to monitor execution and retrieve results. Following execution, results are gathered from the HDFS and consolidated before being returned to the user. The garnered results can be accessed online or downloaded as a text file, which can then be imported into external visualisation tools such as Tableau [31], Microsoft Power BI [32], or various Python visualisation libraries (e.g., matplotlib).

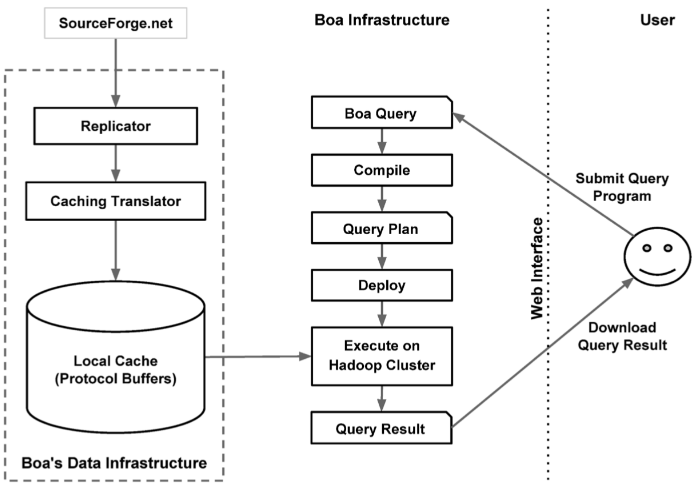


Figure 4: Boa architecture [1]

Regarding workload distribution and resource management, Boa relies on its underlying infrastructure — Hadoop. Consequently, the allocation of incoming tasks to worker nodes is determined by Hadoop's scheduler. Data locality within the Hadoop ecosystem refers to the strategy of relocating computation tasks closer to the physical location of the data rather than transferring substantial volumes of data to the computation nodes [33]. This approach reduces overall network congestion and enhances the system's throughput. Hadoop defines three levels of locality:

- **Node-local.** Data resides on the same node that is processing it.
- **Rack-local.** Data is located within the same rack but on a different node from the one processing it.
- **Non-local.** Data is positioned on a different rack compared to the worker node processing it.

The workflow is executed on a master/worker architecture, which is common among distributed Big Data processing engines. Hadoop utilises the MapReduce model designed for handling large-scale datasets across computer clusters [6], which is also independent of storage mediums. In Hadoop, the master node is called the *JobTracker*, while the worker node is referred to as the *TaskTracker*. When a MapReduce job is initiated, Hadoop first divides the input dataset into evenly sized data blocks (i.e. jobs). Each data block is then scheduled to a worker node for processing. The task allocation process adheres to a heartbeat protocol, operating on a First-In-First-Out (FIFO) basis. A worker node signals the master when it is idle, upon which the scheduler assigns new tasks to it. The scheduler considers data locality during data block allocation, striving to pair a node-local block with a worker. If this is not feasible, the scheduler reduces data locality by assigning a rack-local or random data block to the worker instead [34]. This methodology, combined with Hadoop's distributed computing capabilities, ensures that data analyses can be executed with both accuracy and efficiency, surpassing the performance of single-threaded tools examined earlier in this chapter.

However, it is arguable that the Hadoop's MapReduce is not a suitable candidate for implementing MSR workflows. This is mainly due to the fact that this platform was inherently designed to process the data already present in a storage system such as HDFS, and not handling external operations (like cloning repositories from their remote locations that would have to be done as a pre-processing step). Additionally, Boa programs are designed to query predetermined datasets and inherently lack support for the customised selection of repositories for analysis. This limitation is addressed by contributions such as [35], which endeavour to overcome this constraint by enabling custom datasets and using personal infrastructure for deploying Boa programs. Still, the custom data would need to be uploaded as a pre-processing step.

Even if implemented, the overhead involved in saving repository clones to the distributed file system impacts performance. Transferring to HDFS is considerably more time-consuming than storing files on a local disk, as each file must be uploaded, its replicas created on other nodes, and metadata updated in the HDFS namenode. This issue is more obvious when workflows involve many small files rather than a few large ones, as is typical with source code repositories (e.g., GitHub blocks files larger than 100MB [36], and filesystem's reports [37] indicate that 49.3% of files used in online applications are smaller than 1 KB, with only 0.8% exceeding 10 MB). In case of the small files the amount of memory for the metadata on the namenode dramatically increases [38] leading to significant performance overhead, known as the small-file problem [39]. For instance, storing 550,000 small files (ranging from 1KB to 10KB each) on HDFS takes approximately 7.7 hours, compared to just 660 seconds on the ext3 local file system (traditional filesystem used in Linux operating system) [40]. These challenges undermine the suitability of frameworks like Boidae that rely exclusively on HDFS for running the distributed mining repository workflows.

2.2.5 Crossflow

Crossflow is another specialised tool designed to address MSR requirements [41]. Crossflow necessitates the workflow specification in its own domain-specific language, which is not specifically tailored to MSR, but supports general design of Big Data workflows. It offers both graphical and textual notation, and its main purpose is to allow the engineer to define tasks and data types to support the workflow execution. In contrast to Boa which is limited to what its own domain-specific language has to offer, the flexibility of task logic in Crossflow is enabled through popular programming languages such as Java and Python, offering developers unrestricted capabilities.

Figure 5 showcases one MSR workflow specified in Crossflow, using the graphical notation. The purpose of this workflow is to check how often different software development technologies appear together in applications. The workflow first queries GitHub for Git repositories containing source code related to particular input technologies (this usually means they contain files of specific extensions) and looks for combinations in which these different technologies appear in applications together. The rounded boxes (e.g. Repository and RepositorySearchResult) represent different types of jobs in the workflow, where each job is defined as a piece of data required to process a task. A task example would be RepositorySearcher, which is shown in a rectangular shape. Each task represents a function to be used to process a piece of data wrapped in an incoming job. Whether a task is represented via the double rectangle or a single one indicates if it is going to be executed in parallel across multiple machines or on a single worker. The cylinders represent communication channels, allowing for different types of jobs to be input or output for connecting tasks.

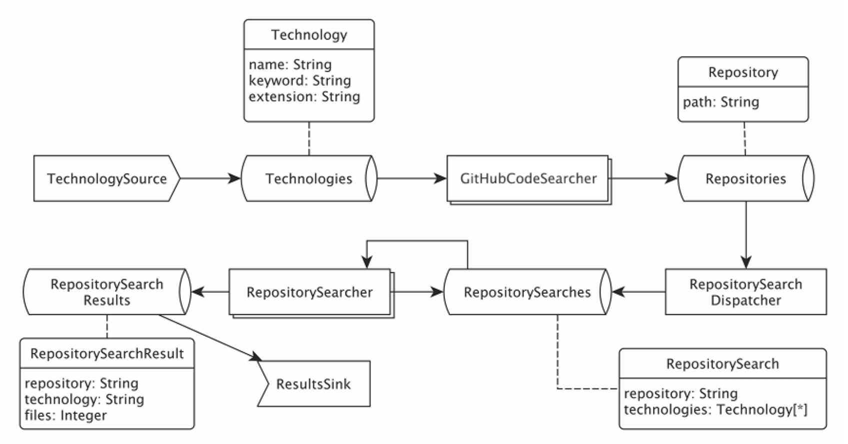


Figure 5: MSR example in Crossflow

While adhering to the master/worker architecture, Crossflow distinguishes itself from other technologies in terms of its component functions. Notably, Crossflow employs "opinionated" nodes, a unique feature that enables workers to decide on whether to accept or decline a job based on their preferences, without direction from the master node. In more detail, instead of the master pushing jobs to the workers it finds appropriate, Crossflow currently deals with scheduling by enabling worker nodes to pull jobs from the master. Before being executed, each pulled job is internally evaluated by the worker to check if it conforms to that worker's *acceptance criteria*. If it does, the job is processed, otherwise, it is returned to the master so another worker can consider it.

The intelligence of workers lies in this evaluation process, where they proceed to perform a task or decline it based on their internal state, i.e. their *opinions*.

In order to ensure the completion of all incoming jobs, workers are required to keep track of any jobs they have previously declined. This enables them to accept such jobs upon seeing them for the second time, at which point it is assumed that no other worker possesses the necessary capabilities to carry out the task.

Crossflow's scheduling approach bears resemblance to the MapReduce's Matchmaking technique [42], which empowers nodes to request jobs rather than passively receive them. In this model, a free node will attempt to pull tasks for which it possesses local data. Failing this, the node remains idle for one heartbeat interval. Upon a subsequent trial, the node is expected to accept a job, even in the absence of the necessary local data. However, in Crossflow, compared to the Matchmaking technique, the specific *acceptance criteria* are highly application-specific and are defined by the developer. For instance, criteria may be related to data locality, prompting worker nodes to scan the contents of their local cache. Alternatively, criteria could involve other conditions, such as the availability of CPU/RAM capacity or particular attribute-based preferences, such as job type or whether worker possesses a certain file.

From an architectural perspective, Crossflow is situated within the realm of model-driven technologies for distributed stream processing. Its architectural overview is illustrated in Figure 6. The Crossflow metamodel defines the syntax for Crossflow's modelling language, which is utilised to represent distributed data processing workflows. Each workflow model is required to conform to the specifications of the Crossflow metamodel and undergoes validation prior to code generation. The Crossflow domain-specific language is built upon the Eclipse Modelling Framework [43]. Once created and validated, the model is consumed by the code generator. This component is pivotal to the Crossflow framework, as it is responsible for generating the foundational scaffolding of the forthcoming Big Data application. Presently, the code generator outputs Java and Python code; however, Crossflow can be extended to support additional programming languages.

The base code produced by the code generator encompasses the essential functions needed to configure the distributed infrastructure, including the establishment of communication channels for worker nodes. It also provides specialised classes to encapsulate the intended behaviour of the application. Nevertheless, it is necessary for the developer to implement the specific algorithms required for data processing. After the compilation process, the generated Java code is packaged into JAR files — one designated for the master node and another for the worker nodes — readying them for deployment within a cluster environment.

Despite the Crossflow scheduler aiming to achieve a high degree of data locality, it presents several limitations when handling MSR workflows described in Section 1.2, which could be enhanced. First, upon the initial execution of the workflow, if the worker nodes do not have any data stored locally, these nodes are likely to reject repository-related jobs due to the lack of local data clones. This situation results in an initial overhead affecting performance.

Second, there is no mechanism ensuring that high-performance workers

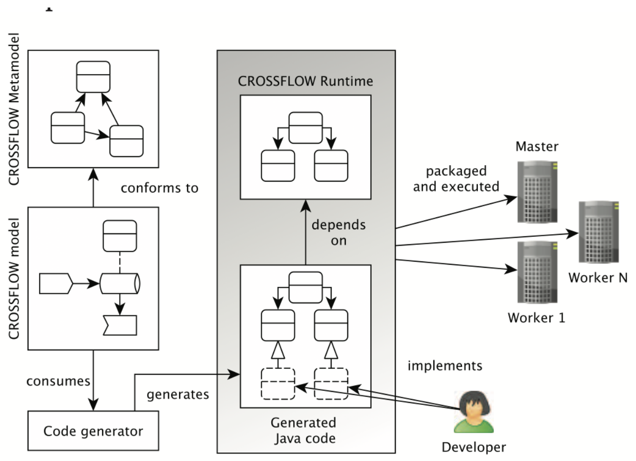


Figure 6: Crossflow architecture

are assigned to compute-intensive tasks, and less demanding tasks to lower-performance workers. Therefore, it is likely there will be redundant clones of the same repository if a node is offered a job it has previously seen, even though some other node has that resource locally but is currently occupied. This lack of alignment can lead to inefficient resource utilisation and increased overall execution times.

2.3 General-Purpose Big Data Frameworks

Although not specifically engineered for repository mining, arguably general-purpose Big Data engines such as Apache Spark [44], Apache Flink [45] and Apache Storm [46] could also be capable of executing MSR workflows effectively. These three frameworks are widely regarded as state-of-the-art solutions for batch processing and stream processing [16] respectively, hence we will delve into the details of their architectures and scheduling.

2.3.1 Apache Spark

Apache Spark is an open-source, distributed computing system optimised for large-scale data processing. It is a Java-based framework that utilises the master/worker architecture. A Spark workflow can be conceptualised as a series of Spark jobs, with each job representing a unit of work or a set of computations that accomplish a specific goal within a Spark application. Each Spark job consists of multiple stages, where each stage is made up of tasks that are identical in operation but process different physical partitions of data. For instance, analysing a repository can be seen as a task in Spark, as each analysis requires workers do some examinations but on different codebases. Consequently, a task represents the smallest schedulable unit of work, executed on a single machine, and tasks can run independently and in parallel across the cluster.

A pivotal component of Spark's architecture is the Driver manager, which operates on the master node and is responsible for coordinating the execution of the Spark application, including the scheduling of tasks across worker nodes. Executors are worker processes tasked with performing jobs in Spark applications. These executors are launched on worker nodes and communicate with the master. Each worker node accommodates one or more executors, which are essentially Java Virtual Machine (JVM) processes dedicated to executing individual tasks. Finally, the cluster manager is responsible for allocating resources to worker nodes, and is connected to the master node (driver program) through the SparkContext object. The worker node cache is the storage mechanism used by Spark's worker nodes to retain data in memory or on disk for processing during job execution. All of these components are depicted in Figure 7.

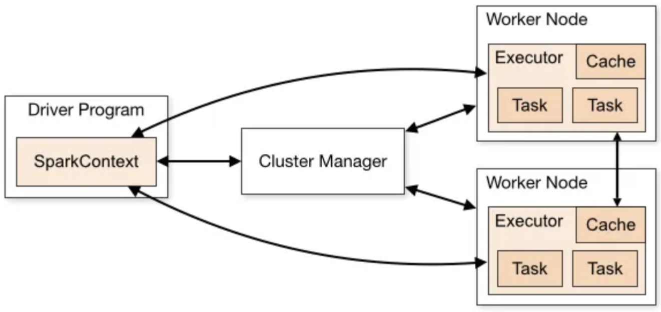


Figure 7: Apache Spark architecture [2]

Data locality is considered a fundamental concept that greatly affects the efficiency of distributed computing frameworks like Apache Spark [47]. Spark optimises job scheduling based on data locality, aiming to enhance computation speed by ensuring that data and the corresponding code are in close proximity. In Apache Spark, data locality is classified into several hierarchical levels, depending on how far data needs to be moved in order to be processed:

- **PROCESS_LOCAL**: This level represents the scenario where the data resides within the same Java Virtual Machine (JVM) as the executing code. Achieving the highest level of locality, accessing data from the same memory space as the code eliminates the need for network or inter-process communication, maximising efficiency.
- **NODE_LOCAL**: At this level, the data is located on the same physical node as the processing unit, but it might be in different processes. For example, the data can be stored within a Hadoop Distributed File System (HDFS) on the same node or across different executors on that node. The

communication between processes introduces slight delays compared to PROCESS_LOCAL.

- **RACK_LOCAL**: Data classified as RACK_LOCAL is located within the same rack of servers but on different nodes. In this situation, data transfer occurs across servers within a single rack, typically routed through a single network switch. This incurs moderate latency due to intra-rack communication.

- **NO_PREF**: This category is used when data has no specific locality preference, meaning it can be accessed equally from any location within the cluster. Hence, the job is to be scheduled without considering data locality.

- **ANY**: The broadest level of data locality, where data can be located anywhere within the network, potentially across different racks. Even though Spark can execute the task on any node, it but may prioritise nodes closer to the data if it can improve performance, unlike in the case with NO_PREF.

Spark aims to allocate computational tasks at the highest possible locality level to enhance execution speeds; however, perfect locality alignment is not always achievable. When a Spark job encounters a scenario where no idle executor has unprocessed data at a higher locality level, it defaults to lower levels. The system may either:

- Delay task execution until a processor becomes available within the same locality, or

- Relocate data to an available processor at a lower locality level.

Typically, Spark adopts a waiting strategy, similar to the delayed scheduling [48], pausing temporarily with the intent that a processor on the desired server becomes available. If this wait surpasses the pre-configured timeout period, the framework proceeds to transfer the data to an available processor, irrespective of the locality level. These timeout threshold is configurable through a parameter ($spark.locality.waitparameter$).

2.3.2 Apache Flink

Similar to Apache Spark, Apache Flink follows the master/worker architecture [49]. The architecture consists of a single master node, known as the JobManager, and multiple worker nodes called TaskManagers. Each worker is equipped with at least one TaskSlot, which represents a unit capable for work. Similarly to the concept of Executors in Apache Spark, when a task slot is available, it can be assigned an incoming job. An architectural overview of Flink can be seen in Figure 8.

In the workflow of Flink, when a workload is submitted to Flink, it parses the corresponding program and generates a JobGraph. The Client then submits

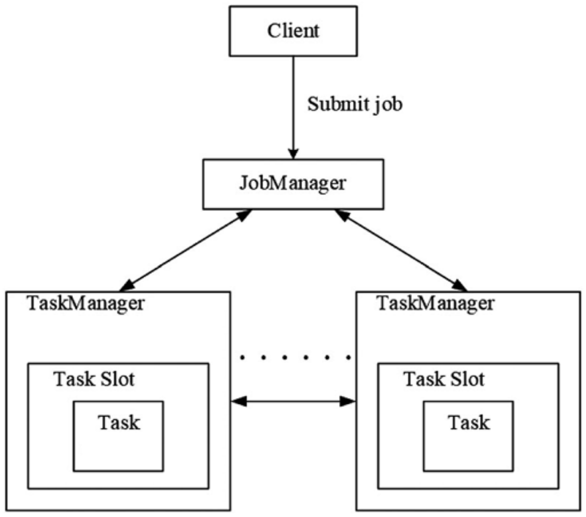


Figure 8: Apache Flink architecture

the JobGraph to the master node, which generates the execution graph. Next, the master node initiates the job scheduling process, which typically involves two phases: worker selection and job deployment.

The worker selection phase, also known as task assignment or job scheduling, determines which workers will be allocated for the incoming job. By default, Flink uses a sequential scheduler that follows the FIFO strategy. This means that during worker selection, the list of available workers is obtained first, followed by the list of workers that meet the resource requirement parameters. For Flink, these parameters include CPU cores, memory size and disk space. Once these requirements are evaluated, the master selects a worker from the list of available workers. It is important to note that Flink's scheduler does not consider the relationship between the job's resource demand and the resources available on the nodes, as it relies on random selection [50]. Thus, data locality is not a criterion for scheduling in Flink, and the data stored in local disk space is not taken into consideration when assigning jobs to workers, potentially making it suboptimal candidate for implementing the data-locality-heavy workflows that are of interest to this research.

2.3.3 Apache Storm

Apache Storm [46] is another popular Big Data stream processing framework. It focuses on real-time analytics, aiming to enable data engineers to collect data from streams, generated in real-time, and process it immediately [3]. Its architecture follows the master/worker paradigm, similar to Apache Spark and Apache Flink.

Figure 9 illustrates the physical architecture of Storm. In the figure, Nimbus acts as the master and Supervisor acts as the worker in Storm, with their metadata managed by Apache Zookeeper [51]. The master distributes and allocates jobs to workers, periodically monitors their status, and addresses faults. Workers directly carry out the jobs assigned by the master node and report their status back to the master. Communication between the master and workers is facilitated through reading and writing metadata in Zookeeper.

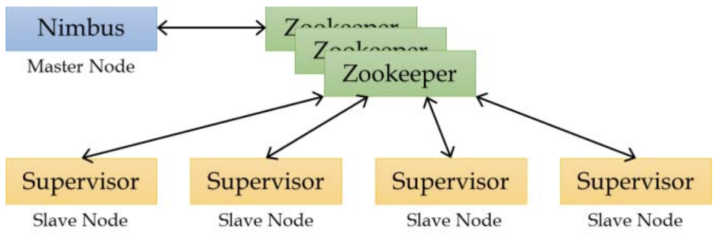


Figure 9: Apache Storm architecture [3]

Apache Storm has a default job scheduler, which distributes jobs within a workflow as evenly as possible among the distributed workers. An even job scheduler aims for high fairness by allocating jobs so that workers utilise system resources as equally as possible, irrespective of data location. This is akin to the Round Robin algorithm [52], which circulates tasks and ensures every task receives an equal amount of CPU time. The round-robin strategy results in a high average waiting time [53].

While Storm does not inherently manage data locality as explicitly as Spark does, some degree of locality can be achieved by employing Storm's stream grouping strategies, which aim to minimise network costs incurred by inter-node communication. To illustrate, consider a streaming application counting clicks per user. This workflow includes three tasks that can run in parallel: filter, aggregate, and store. The filter task receives a detected click (e.g. *"user_id": 123, "action": "click", "timestamp": 162758*) and formats its data before passing it on for aggregation. The aggregation task produces a new piece of data (e.g. *"user_id": 123, "click_count": 5*), which is then passed to the subsequent task that updates the corresponding database record. Storm's grouping strategies determine how these output pieces of data are passed through the system. One strategy, known as *local-or-shuffle grouping*, reduces communication delay between nodes by attempting to use the same worker to aggregate and store the data, as the one that filtered it, provided the necessary tasks are running on that worker node. Although this might adhere to locality principles to some extent, [3] argues that such an approach may lead to imbalances in workers' workloads as it disregards load balancing. Additionally, [54] contends that this approach contradicts the intentions of the even scheduler employed by Storm, potentially significantly degrading the performance of the distributed stream processing engine by causing issues such as worker misuse, worker overuse, or increased network communication costs.

Although Apache Storm can be configured to mine software repositories, given its even scheduler and its inherent design tailored for real-time event processing rather than complex, multi-step workflows, it is debatable whether it is a natural fit for this use case. Despite offering distributed processing, it may not effectively tackle data locality and could unnecessarily prolong workflow execution.

3 Analysis and Hypothesis

Upon examining the various methodologies detailed in the literature, we pinpointed several key insights that influenced our research. This chapter aims to highlight the limitations found in the referenced studies and define the research objectives that, if achieved, could address the identified shortcomings of these previous contributions.

3.1 Analysis

In examining the implementation approaches for MSR workflows, we observe several notable differences among tools that warrant discussion. Table 1 outlines key characteristics of tools and frameworks found in the literature with respect to executing MSR workflows such as the one described in Section 1.2. To provide a structured analysis of these characteristics, each column of Table 1 will be analysed separately.

Framework/ Feature	**Distributed**	**Suitable for MSR Workflows**	**Custom Dataset**	**Deployable to Cloud/Personal Infrastructure**	**Scheduler Considers Data Locality**	**Scheduler Considers Cluster Heterogeneity**
Boa	✓	✓	✗	✗	✓	✗
Crossflow	✓	✓	✓	✓	✓	Partially
Apache Spark	✓	✓	✓	✓	✓	✗
Apache Flink	✓	✓	✓	✓	✗	✗
Apache Storm	✓	✗	✓	✓	✗	✗
SmartSHARK	✓	✗	✗	✗	-	-
World of Code	✓	✓	✗	✗	-	-
PyDriller	✗	✓	✓	-	-	-
PyRepositoryMiner	✗	✓	✓	-	-	-
RepoFS	✗	✓	✓	-	-	-
Perceval	✗	✗	✓	-	-	-

Table 1: Comparison of frameworks and their features with respect to MSR workflows described in Section 1.2.

1. **Distributed**
   The "Distributed" column indicates whether the framework supports distributed computing. Distributed computing allows the framework to process tasks across multiple nodes, which is crucial for handling large-scale data. In this context, Boa, Crossflow, Apache Spark, Apache Flink, Apache Storm, SmartSHARK, and World of Code all support distributed computing, providing significant advantages in scalability and performance. Conversely, frameworks like PyDriller, PyRepositoryMiner, RepoFS, and Perceval do not support distributed computing, potentially limiting their applicability for large-scale analyses.

2. **Suitable for MSR workflows**
The "Suitable for MSR Workflows" column assesses the capability of a framework to effectively manage MSR workflows, which encompass activities such as repository cloning, codebase analysis, and aggregation of results. Frameworks like Boa, Crossflow, Apache Spark, Apache Flink, World of Code, PyDriller, PyRepositoryMiner, and RepoFS are appropriately equipped to handle these functionalities, thus making them suitable for MSR workflows. In contrast, frameworks such as Apache Storm, SmartSHARK, and Perceval exhibit limitations in this regard. Perceval primarily serves as a data collection tool, lacking features for querying and visualising the collected data, which diminishes its utility for comprehensive MSR workflows. SmartSHARK's focus is predominantly on analysing repository metadata rather than the source code within the repository, limiting its applicability in scenarios requiring detailed codebase analysis. Additionally, Apache Storm is inherently designed for real-time event processing and does not support the complex, multi-step workflows characteristic of repository mining, which is restricting its suitability for MSR applications.

3. **Custom dataset**
The "Custom Dataset" column denotes the framework's capacity to analyse any dataset imported by the user, as opposed to being constrained to predefined datasets. Frameworks such as Crossflow, Apache Spark, Apache Flink, Apache Storm, PyDriller, PyRepositoryMiner, RepoFS, and Perceval exhibit the capability to manage datasets created by the developer. Conversely, frameworks like Boa, SmartSHARK, and World of Code demonstrate limitations in this regard as they are confined to analysing datasets that are either preloaded or predefined within the framework. For instance, Boa confines operations to pre-cached and formatted repository snapshots made available within the framework. It relies heavily on Hadoop and is optimised solely for predefined datasets stored in the HDFS, requiring additional software to enable analysis of custom datasets, such as the one coming from GitHub.

4. **Deployable to cloud/personal infrastructure**
The "Deployable to cloud/personal infrastructure" column evaluates the adaptability of the framework for deployment across a range of infrastructural environments, from cloud-based services, such as AWS virtual machines, to personal computing hardware. The ability to deploy on diverse infrastructures is integral for meeting varied deployment requirements. Frameworks such as Crossflow, Apache Spark, Apache Flink, and Apache Storm exhibit support for such deployability, thereby offering enhanced operational flexibility with respect to deployment scenarios. In contrast, frameworks like Boa, SmartSHARK, and World of Code exhibit restrictions in terms of deployment capabilities. For example, the infrastructure underpinning Boa is deployed on clusters administered by Iowa State University, making it largely inaccessible to the wider user base [35]. Similar

to the constraints observed with World of Code and SmartSHARK, this specific arrangement prevents the standard execution of programs within cloud environments such as AWS or on private dedicated infrastructures. Frameworks aimed at single-machine execution, such as PyDriller or RepoFS, are disregarded in this case, as the main focus of this research is distributed execution of MSR workflows.

5. **Scheduler considers data locality**
   The "Scheduler considers data locality" column describes the extent to which a framework's scheduler is capable of optimising task execution based on the proximity of data. Frameworks such as Boa, Crossflow, and Apache Spark incorporate schedulers that actively consider data locality, however frameworks like Apache Flink and Apache Storm do not integrate data locality principles into their scheduling mechanisms.

   For instance, Spark takes data locality into account but follows a stage-by-stage operational model. In this paradigm, jobs of the same type — such as repository analysis — are scheduled collectively, and subsequent stages are only initiated upon the completion of the preceding ones [55]. This scheduling approach is critical because the output of one stage often serves as the input for the next. While Spark attempts to assign tasks to workers already possessing the requisite data, it may fail to fully leverage data locality if the data distribution among workers fluctuates during the execution of a stage, such is the case with new repositories being cloned to a worker's local storage at runtime.

   Crossflow attempts to address data locality by scheduling jobs individually. Nevertheless, this approach has its own shortcomings. Specifically, since workers operate independently, the master node does not maintain records of which worker previously handled a particular resource. Consequently, this can lead to scenarios where jobs are assigned to idle workers when allocating them to engaged workers, who already have part of the relevant data, could improve data locality and reduce processing times and data transfer costs. This can also happen when lower-performance workers are assigned data-intensive or compute-intensive jobs because they have already seen them in the past and the workers who would process these jobs more efficiently are occupied.

6. **Scheduler considers cluster heterogeneity**
   The "Scheduler considers cluster heterogeneity" column determines if the framework can efficiently allocate jobs considering the diverse capabilities of nodes within a cluster. This feature is essential in heterogeneous environments where nodes have varying computational power, memory, and storage capacities. While both Boa and Crossflow provide mechanisms for distributed mining, neither these tools nor general-purpose frameworks for distributed data processing, with their respective schedulers, adequately account for the dynamism inherent in execution environments at runtime,

such as network oscillations and differences in the physical characteristics of worker nodes, which are discussed in Section 6.1.

In the context of dynamic and heterogeneous environments, Crossflow exhibits a degree of adaptability, provided developers utilise the *acceptanceCriteria* to monitor real-time metrics such as CPU utilisation, RAM availability, and network connectivity speed. These criteria would enable worker nodes to pull incoming tasks aligned with their current state. Despite this flexibility, Crossflow does not inherently address the broader scope of cluster heterogeneity.

Moreover, frameworks such as Apache Spark, Flink, and Storm assume a homogeneous cluster setup, which can lead to inefficiencies in heterogeneous environments comprising nodes with varying specifications [56]. This presumption of uniformity results in suboptimal workload distribution and performance, particularly in clusters exhibiting significant disparities among worker nodes.

In summary, while distributed frameworks like Boa, Crossflow, Spark, Flink, and Storm offer various strategies for handling MSR workflows, they exhibit significant limitations in dynamic, heterogeneous environments. These downsides include inadequate runtime adaptability, selective approaches to data locality, and presumptions of cluster homogeneity — each of which can interferes with optimal performance and cost-effectiveness in complex, real-world scenarios. Moreover, the inherent limitations of Boa and Storm render them unsuitable as foundational frameworks for our research. Therefore, we will perform a series of tests to identify the most suitable framework among Spark, Flink, and Crossflow, as the frameworks identified to possess most of the key characteristics for executing MSR workflows. This evaluation will precede the development of methodologies aimed at proving the hypothesis and achieving the research goals outlined in the subsequent sections.

3.2 Hypothesis

The hypothesis of this study is that a specialised locality-aware job orchestration framework and algorithm can outperform current methods for repository mining workflows, with the following characteristics:

- **Distributable**. One critical aspect of our repository mining workflows is that they can be executed in parallel across multiple worker nodes, thereby leveraging the computational capabilities of a distributed system to accelerate the processing of large datasets.
- **Locality-intensive**. Another significant characteristic is that they operate on data that is costly to access over the network or move to the computation. Given that some workers may already possess local copies of the required data, it is highly advantageous to allocate relevant computation to those workers and minimise data transfer times and network congestion, thereby impacting overall workflow efficiency.

- **Heterogeneous workers**. The worker nodes involved in the processing have disparate capabilities, with variations in processing power, storage capacity, and network bandwidth. This heterogeneity can impact the efficiency of the workflow significantly if not managed appropriately. For example, a worker with high processing power but limited storage might be optimal for CPU-intensive tasks but suboptimal for tasks requiring significant data storage. Similarly, a worker with high network bandwidth would be better suited to tasks that involve extensive data transfer. An effective job orchestration framework must intelligently assign tasks to the appropriate workers based on these characteristics to maximise overall efficiency.

For the remainder of this research, such workflows will be referred to as Locality-Intensive Distributed Mining Software Repository (LID MSR) workflows. This terminology underscores the focus on locality-aware job orchestration in the context of distributed mining of software repositories.

3.3 Research Goals

To investigate the validity of this hypothesis, the following research goals have been established:

- **RG1**. **Use existing distributed data processing tools and frameworks to implement representative LID MSR workflows and measure their performance to establish a baseline for this research.** This research goal involves leveraging distributed data processing tools, such as Apache Spark, Apache Flink, and Crossflow, to develop and execute representative LID MSR workflows. These tools will be used to obtain critical performance metrics and establish a baseline to serve as a point of reference for subsequent evaluations of newly developed methodologies.

- **RG2**. **Design and implement novel scheduling algorithms tailored for LID MSR workflows.** This research goal is about creating new scheduling algorithms specifically designed for LID MSR workflows. These algorithms will be built to optimise workflow processing times by increasing data-locality. Moreover, these algorithms should account for the possible differences in physical characteristics of worker nodes.

- **RG3**. **Experimentally evaluate the efficiency of the developed algorithms against the baseline.** This research goal involves conducting experiments to compare the performance of the new scheduling algorithms with the established baseline by carefully measuring key performance metrics, like the time taken to complete the workflow and the amount of data transferred over the network, to provide clear evidence of how well the new algorithms perform.

3.4 Evaluation Metrics

This section outlines a set of metrics to be utilised for the comparative analysis of proposed methodologies with respect to data locality, resource utilisation, and data transfer in order to determine if research goals have been achieved. This set of metrics may include the following attributes, each to be quantitatively measured:

1. **End-to-end execution time.** The duration required to execute a workflow, encompassing the period from initiating the master and worker nodes to the termination of their processes.

2. **Data load.** The volume (in megabytes) of data that is not local and must be transferred to the worker nodes during execution. Given the nature of LID MSR workflows that deal with remote repositories, we assume all repositories must be downloaded over the Internet and that no data will be transferred between worker nodes.

3. **Cache miss.** The frequency with which workers lacked the requisite data locally and consequently had to either download the data or transfer it to their local environment to execute computations.

4. **Average worker usage.** The percentage of total execution time workers spent on processing jobs, rather that being idle or communicating with the master node.

The algorithms developed through this research aim to reduce the end-to-end execution time for workflows wherein workers necessitate local data to perform tasks. Consequently, this metric will be deemed the most important when evaluating the proposed contributions. While cache misses are inherently associated with data load, they are categorised as a separate metric due to the potential variability in repository sizes over time. Based on the scheduler's efficiency, the same number of cache misses does not necessarily equate to the same amount of data transferred, given that each repository has unique data sizes. Performance-wise, there is a significant distinction if workers fail to reuse a local copy of a smaller repository (e.g., 20MB in size) compared to a larger repository, which might be several hundred megabytes.

4 Baseline Selection

Before designing new solutions for LID MSR workflows, it is crucial to benchmark existing distributed data processing frameworks in order to identify the most efficient framework for executing them, so that we can then compare the novel scheduling algorithms developed in this research against it. Therefore, this chapter focuses on the first research goal as described in Section 3.3:

- RG1. **Use existing distributed data processing tools and frameworks to implement representative LID MSR workflows and measure their performance to establish a baseline for this research.**

The aim is to compare the execution processes and overall performance of a representative repository mining workflow. We have implemented this workflow using several data processing frameworks, each considered as a potential baseline. Based on insights from Chapter 2 and Section 3.1, our goal was to compare Apache Spark, Apache Flink, and Crossflow to determine the most suitable baseline framework. These frameworks were selected as they represented the top three options, fulfilling the most criteria when comparing their features and capabilities in relation to LID MSR workflows, as summarised in Table 1.

Apache Spark and Apache Flink, despite being designed as general-purpose frameworks, are both capable of executing MSR workflows. While Spark might initially be considered a more appropriate choice for described subset of repository mining workflows due to its data locality features, our evaluation extended to Flink as well. The rationale for this consideration lies in Flink's ability to process jobs as streams, as opposed to Spark's batch processing model, which could introduce significant variations in execution dynamics. Moreover, Crossflow was selected for its built-in data-locality features and opinionated nodes, which could prove advantageous in managing cluster heterogeneity.

4.1 MSR Implementation Example

We present one example of a simple LID MSR workflow, shown in Figure 10. In this context, the workflow will search GitHub for repositories containing source code related to specific technologies. The objective is to identify combinations of these technologies that coexist within applications. Such a workflow could comprise four distinct steps:

1. **GitHub search.** Query GitHub for technologies of interest and identify repositories to analyse.

2. **Clone.** Make local copies of the repositories identified in the GitHub search results.

3. **Analysis.** Calculate the frequency with which technologies appear together.

4. **Report.** Store the results in a CSV file.

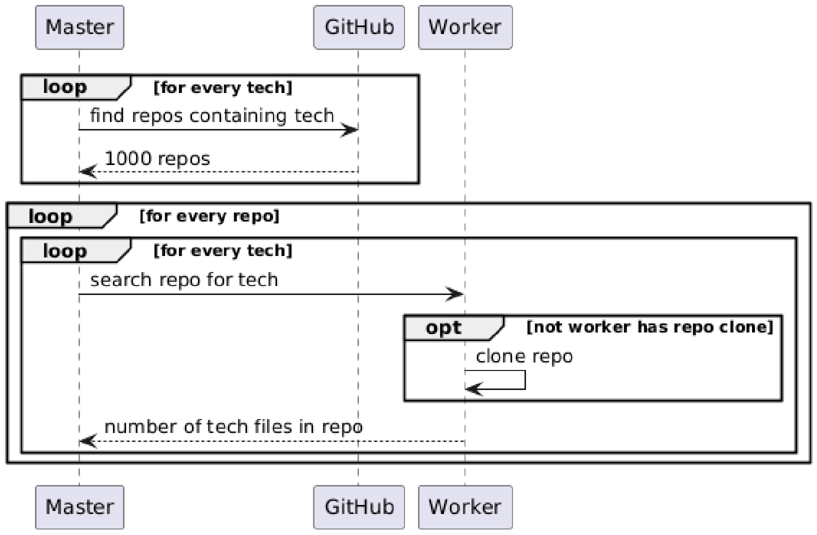


Figure 10: Sequence diagram showing one LID MSR workflow

For instance, we might analyse open-source GitHub repositories to evaluate the concurrent utilisation of various model-driven technologies. The procedure commences by documenting the technologies to be searched for in a structured format (such as JSON or CSV). An example of the technologies document is shown in Table 2). This serves as an input for the starting step of the workflow, which is searching GitHub for repositories containing files associated with a particular technology (identified by specific file extensions). For each of the documented technologies, we run a GitHub query (for example, in Java programming language this query would look like:

github.searchContent().q(technology).extension(extension)

In its default configuration, GitHub search returns up to 1000 results, each one corresponding to a file with the queried technology's extension. The associated repository for each file is also identified. This repository is then sent downstream for further processing.

All selected repositories undergo examination, during which they are inspected for the presence of other technologies listed in the input CSV file. When two technologies are found within the same repository, this instance is documented as a “co-occurrence hit.” After analysing all repositories, these “co-occurrence hits” are compiled into a results CSV file.

This workflow has been implemented in Apache Spark, Apache Flink, and Crossflow. The subsequent sections depict the methodology - in this case, the experiments carried out - and the manner in which each of these platforms addresses the execution of the workflow, as well as the degree to which they are capable of reutilising the local repository clones.

Technology	**Keyword**	**File extension**
eugenia	gmf.node	ecore
eol	var	eol
gmf	gmf.graph	figure

Table 2: Example of a CSV document with model-driven technologies to query GitHub

4.2 System Model

In this section, we describe the system model utilised in the development and evaluation of the potential baseline frameworks and our distributed job allocation methodologies. This model is structured to encapsulate the essential components and interactions present in a distributed computing environment. Each component's characteristics contribute to the overall system performance, and modifications to any of these elements can cause variations in execution outcomes.

The constituents of the system model are as follows:

- **Workflow**
  A workflow comprises a series of tasks that must be executed sequentially or in parallel on a collection of jobs. The workflow orchestrates the flow of jobs through various transformative and computational tasks, ensuring that each job undergoes the requisite operations. Sequential execution occurs when tasks have inter-dependencies and the outputs generated from the execution of one task serve as inputs for the subsequent task.

- **Task**
  A task represents a transformative function or a set of functions that are applied to each job. Tasks are the fundamental units of computation within the workflow. Each task takes a job as input, performs the designated operations, and produces an output.

- **Job**
  A job is a discrete piece of data that needs to be processed by a single task. Jobs are the payloads that traverse through the workflow's tasks. In case of LID MSR workflows, jobs are describing a repository that should be processed, hence they hold information about the repository, such as repository URL, size, commit hashes, number of files or number of lines of code.

- **Job configuration**
  Job configuration refers to the input collection of jobs in a single workflow that are to be processed by worker nodes. In a basic three-step workflow, which includes obtaining input jobs, processing them, and storing aggregated outputs, the job configuration represents all jobs to be processed during the entire workflow execution. For more complex workflows

involving multiple steps, the job configuration serves as the initial set of jobs. As the workflow progresses, additional jobs may be dynamically generated as a result of tasks executed by the worker nodes.

- **Master**
  The master node is the central coordinator in the distributed environment. It is responsible for running the job scheduler and managing the overall workflow execution. The master node collects and aggregates results from tasks executed by workers. The master node refers to the part of the application that can only orchestrate a single workflow at a time, and different workflows would have to be deployed as separate applications (having their own master nodes).

- **Worker**
  A worker is a node within the distributed environment tasked with executing individual tasks. Each worker node is capable of performing a single task multiple times by processing different jobs. Workers may vary in their computational capabilities and network bandwidth. The worker node refers to the piece of software that can only execute tasks from a single workflow at a time, and different workflows would have to be deployed as separate applications (having their own worker nodes).

- **Queue**
  A queue is a communication component that facilitates message exchange between the master and worker nodes. Queues can serve different purposes, such as job queues (attached to workers for job processing) or message exchanges (such as those used for sending bids or reporting results).

  The system utilises two types of queues: broadcast queues and direct queues. Broadcast queues involve the master node queuing messages, which are then received and read by all worker nodes. In contrast, in direct queues the master queues a message for a single worker to dequeue, or a worker queues a message for the master to dequeue.

  For direct communication, queues can be categorised as either dedicated or general. Dedicated queues facilitate direct interaction between the master node and a specific worker node. General queues involve the master node queuing a message that can be dequeued by any available worker node.

- **Job scheduler**
  The job scheduler is a component embedded within the master node. It encapsulates the logic for allocating jobs to workers that execute tasks associated with these jobs. For job allocation purposes, the scheduler uses either dedicated queues (for directly assigning each job to a specific worker), or general queues (for assigning jobs to the first idle worker).

- **Job allocation**
  Job allocation refers to the pairing of jobs with workers for processing. Each job is allocated to a single worker only.

In case of the system using dedicated queues, the scheduler may rely on the dedicated queue threshold for job allocation, which is the maximum number of jobs (n) that can be queued for each worker to keep the scheduler allocating unprocessed jobs. This threshold pauses job allocation temporarily once all workers reach (n) jobs in their queues. As soon as any worker completes some of its assigned jobs and its dedicated queue falls below the threshold, job allocation resumes. For general queues, such threshold is equivalent to 1, as the job allocation is triggered by any of the workers becoming idle and dequeueing the next unprocessed job.

It is important to emphasise that the use of a dedicated queue threshold is not a universal feature employed by all schedulers. Instead, it is an implementation choice that may be leveraged depending on the design and operational requirements of specific systems. For instance, the Crossflow framework utilises this approach to address scheduling constraints. In contrast, other general-purpose schedulers may adopt alternative methods that do not rely on queue thresholds, opting instead for other mechanisms to balance worker loads or trigger job allocation dynamically.

- **Speed**
  Speed is a critical attribute of a worker, denoting its efficiency in either transferring data to local storage (network speed) or processing a job (processing speed).

- **Data transfer time**
  The time, measured in seconds, required for a worker to clone a single repository. It is dependent on the repository size and the network speed of the worker.

- **Processing time**
  The time, measured in seconds, necessary for a worker to analyse a single repository. Is dependent on the worker's processing speed and specific job attributes, which can vary depending on the use case. For instance, cloning a repository correlates to its size and URL, whereas code linting might be influenced by the line count of code to be processed.

Problem statement

Using the model described above, we aim to optimise job allocation in distributed environments where worker nodes potentially operate at different network and processing speeds. The optimisation objective is to minimise the end-to-end execution time through increasing data locality (i.e. minimising data load) and average worker usage.

4.3 Experimental Setup for Baseline Framework Selection

To identify an optimal baseline framework for our research, an experimental setup needed to be established. To achieve this, we decided to control various components of the system model to ensure that the comparison of candidate frameworks was both fair and feasible, within the given time-frame and budget. This section outlines the experimental framework adopted for the baseline selection process and explains the rationale behind the controlled parameters within our simulation environment.

Controlled system model components

Similar to the methodologies described in [57, 58, 59, 60, 61, 62] that will be discussed in detail in Sections 6.1 and 7.1, to make our experimental comparisons fair and avoid misleading variability, we have chosen to regulate the some key parts of the system model:

1. **Job configuration**
   Job configuration is controlled to guarantee that exactly the same workload is used to run experiments across all candidate frameworks. Predetermining jobs within the workflow provided a controlled environment for evaluating the performance of these frameworks under various computational workloads - ranging from highly redundant workloads to completely unique workloads. Hence, we were able to systematically investigate the frameworks' efficiency in various scenarios. This approach enabled us to address a broader spectrum of use cases beyond the specific example's GitHub response, which might not generalise well across different LID MSR workflows.

2. **Worker speeds**
   The heterogeneity in network speeds and processing capabilities among workers is a common occurrence in distributed data processing environments. Some workers may be operating on high-speed fibre-optic connections with robust computational hardware, while others may be constrained by slower internet connections and less powerful processors. Testing under such varied conditions is crucial to understanding the performance dynamics of data processing frameworks like Apache Spark, Apache Flink, and Crossflow. Controlling these worker characteristics allowed us to perform extensive benchmarking without the logistical and temporal constraints associated with setting up a heterogeneous real-world infrastructure, and ensure the reliability and reproducibility of our results.

3. **Network and processing times**
   Network and processing times were controlled as functions of worker speeds. By containing these aspects in conjunction with worker speeds, we maintained consistency throughout the evaluations, thereby focusing our experiments on the actual performance of the frameworks rather than external network and performance conditions.

The adoption of simulation techniques which controlled both worker characteristics and job configurations was crucial in the analysis of data processing frameworks. Not only did it allow us to control and manipulate the experimental conditions finely, but it also provided a versatile platform for exploring a wide range of scenarios. From understanding the impact of network and processing heterogeneity to evaluating the performance under different job loads, this controlled setup enabled a degree of experimental rigour that would be unattainable with real-world infrastructure. Our findings, therefore, offer robust insights into the operational efficiency of frameworks like Apache Spark, Apache Flink, and Crossflow under varied conditions, providing a solid foundation for development of optimisation strategies.

4.3.1 Worker Speed Configuration

Across all experiments, we tested four different worker speeds configurations[1]:

- **All-equal.** A configuration where all workers possess the same, or nearly the same, network and processing speeds.
- **One-fast.** A configuration where one worker is significantly faster than the others in terms of network and processing speed.
- **One-slow.** A configuration where one worker is significantly slower than the others in terms of network and processing speed.
- **Fast-slow.** A configuration encompassing one slow worker and one fast worker, while the remaining workers have average download and processing speeds.

To assess the extent to which variations in workers' speeds impact overall execution, we configured the *fast* workers to be 5-10 times faster than the average worker (used in the *all-equal* configuration) and the slow workers to be 5-10 times slower than the average worker. Internet access speed can be represented as a function of the repository size and the worker's pre-configured speed. For example, an average worker node may have a pre-configured network speed of 40Mb/s for downloading repositories, meaning that an 800MB-sized repository will be cloned in approximately 20 seconds, depending on network fluctuations during the cloning process. In contrast, processing speed is more dependent on the specific workflow implementation. For instance, searching for popular *npm* libraries [4] in TypeScript projects is not computationally equivalent to performing static analysis of the code due to the fact that static analysis requires creating and traversing the abstract syntax tree and therefore uses significantly more RAM and CPU capacity than searching for *npm* dependencies in package .json files. These pre-configured speeds are used as a guideline, but they are subjected to noise to account for the situations when internet connectivity fluctuates, and the processor can have other pending tasks during execution. In our experiments we used a noise characterised by a Gaussian distribution with

[1]The details of worker configurations can be found in `https://github.com/ana-markovic/lid-msr/tree/master/baseline-modeldriven/tests-old/experiment/techrank/in`

a mean value that is 85% of the maximum speed and a standard deviation of 5% of the maximum speed.

4.3.2 Job Model

Similar to the worker speed configurations, the incoming jobs were set up to test variance in overall performance during workflow execution. We defined the total number of source code repositories to be processed during the workflow execution. For each repository in the input dataset we set the following parameters:

- **Repository name/URL.** The unique identifier of the repository, typically represented by the URL used to access the code.
- **Repository size.** The total number of bytes of all files stored in a repository.

These parameters, combined with the worker speed configurations, enable us to closely monitor the execution flow across various deployment and workload scenarios, thereby allowing us to determine when and if mechanisms of the observed framework offer advantages during execution.

4.3.3 Job Configuration

To evaluate the performance of the three frameworks in conjunction with the described workflow, we created five different job configurations with 120 jobs each. For all jobs across these configurations, we adhered to the structure described in Section 4.3.2, which entailed the pre-specification of URLs and sizes. The repositories varied in size, spanning small, medium, and large categories, with sizes ranging from 1MB to 1GB. Furthermore, the jobs were designed to be either unique or repetitive, depending on whether jobs requiring the same repository are repeated within the same workflow execution. The specific configurations are detailed as follows [2]:

- **All_diff_equal.** This configuration features an equal distribution of repository sizes, with each job in the scenario utilising a different repository.
- **All_diff_large.** Predominantly large repositories are utilised, with each job employing a different repository.
- **All_diff_small.** Predominantly small repositories are utilised, with each job employing a different repository.
- **Repetitive_large.** This configuration exhibits a repetitive pattern with predominantly large repositories, where 80% of the large-scale jobs require one of the three repeating repositories.

[2]The details of job configurations can be found in `https://github.com/ana-markovic/lid-msr/tree/master/baseline-modeldriven/tests-old/experiment/techrank/in`

- **Repetitive_small.** Similarly, this configuration adopts a repetitive pattern with predominantly small repositories, wherein 80% of the small-scale jobs necessitate one of the three repeating repositories.

For the purposes of this evaluation, we executed all combinations of worker and job configurations to scrutinise the variances in execution flow and the time required to complete the entire workload. Considering that the capabilities of worker nodes are predefined prior to the commencement of workflow execution, much like the jobs are, it is arguable that it is these differences in job schedules that impact the overall execution times. Therefore, the following subsection aims to elaborate the implications of job distribution for each of the processing frameworks.

4.3.4 Deployment Configuration

When conducting experiments, we created a local environment that initiates multiple processes on a single machine: one for each of the workers, one for the master, and one for the communication infrastructure (message queues). In this setup, we allocated dedicated filesystem space to each worker (e.g., a directory maintained by a single worker process), thus mimicking a situation in which files are not shared between VMs in a cloud environment. The local setup is advantageous for simulated experiments as they are independent of the physical characteristics of the machine on which they run. Instead, workers in simulated experiments utilise pre-determined speeds to support various configurations, meaning the time required to fetch or process the repository is dictated by the worker parameters and repository attributes.

4.4 MSR Execution Overview

4.4.1 Apache Spark

To begin, let us investigate the implementation of the depicted MSR workflow within Apache Spark[3]. By importing the job configuration, we can create a list of repositories to be searched for files related to specific model-driven technologies, subsequently providing their URLs as input to the Spark workflow. Upon receiving these URLs, Spark, as a distributed data processing engine, will allocate the jobs of cloning and code analysis among its workers.

For this purpose, the Spark scheduler first generates jobs for individual actions and subsequently organises them into stages, which are aggregations of jobs. In our particular scenario, we can distinguish two primary stages: one for cloning the repositories and another for analysing the codebase. As an example, one job in a stage might be the cloning of repository $r1$, while another could be examining $r1$ for technologies of interest.

As a batch processing platform, Spark schedules each stage in its entirety before the execution commences. Furthermore, the scheduler adheres to the

[3]The details of executing MSR example using Apache Spark can be found in `https://github.com/ana-markovic/lid-msr/tree/master/spark-modeldriven`

First-In-First-Out (FIFO) paradigm, where the first stage in the queue receives precedence over all available resources, provided its jobs are ready to be performed. Subsequently, following stages in the queue are prioritised accordingly. It is significant to note that if the initial stages are substantial in scale, there might be a notable delay for the subsequent stages and its jobs [63]. By default, Spark's operational paradigm mandates that workers must await the completion of all repository cloning activities before initiating the subsequent stage involving repository analysis. This bottleneck could be particularly pronounced in scenarios where the cloning process is time-consuming, leading to delays in downstream tasks and the underutilisation of available computational resources.

Considering data awareness and the scheduling of cloning operations, it is vital to recognise that the required data is stored on GitHub and is not initially local to any worker prior to execution (the cloning process relocates the necessary data closer to the computation). Each worker node in the Spark cluster is assigned a subset of repository URLs to clone. However, looking into the job configuration, if there are multiple URLs of the same Git repository, we cannot ensure that the scheduler will adhere to the principle of data locality during the scheduling as none of the repositories are local at the time of scheduling. Consequently, the scheduler will attempt to distribute the jobs evenly across all workers in the cluster. Data locality will become relevant only during the scheduling of the subsequent stage, wherein code analysis will be aligned with the workers that have local copies of the repositories stored on their hard drives. Once the repository analysis is complete, the results will be stored in the filesystem of the master node.

4.4.2 Apache Flink

In contrast, Flink[4], functioning as a stream processing engine, schedules jobs as they become available with the objective of achieving balanced resource utilisation. In the context of the described LID MSR workflow, the repository URLs are ingested as a stream of data records, and Flink distributes these repository URLs across its parallel processing workers. The primary advantage of this methodology is that Flink inherently attempts to prevent worker idleness, thereby avoiding scenarios where workers must wait for others to complete cloning before proceeding to the analysis phase. Due to the absence of mechanisms to address locality within Flink's scheduling, it is not possible to predetermine which worker will analyse which repository. Upon successful processing of a repository, the master will store the resultant data in a single CSV file.

[4]The details of executing MSR example using Apache Flink can be found in `https://github.com/ana-markovic/lid-msr/tree/master/flink-modeldriven`

4.4.3 Crossflow

In Crossflow[5], the list of repository URLs is similarly ingested as a data stream into the processing engine. The scheduler within Crossflow allocates these URLs for processing on a First-In, First-Out (FIFO) basis. Unlike Apache Spark and Flink, the master node in Crossflow does not directly manage job assignments; instead, it maintains a queue of unprocessed jobs. The *acceptance criteria* functions of the worker nodes are designed to enhance data locality by scanning the contents of local filesystems for existing repository clones. Crossflow's scheduler uses a general queue for job allocation by default, meaning that when workers become idle, they retrieve an available job from the queue and evaluate it against their *acceptance criteria*. If a worker node has previously encountered the incoming repository and already downloaded its contents, it will accept the job. Conversely, if it has not, the job will be rejected and returned to the queue for processing by another worker. The worker node that rejected the job will flag it in case the job is encountered again in the future. Should all worker nodes reject the job and return it to the queue, the repository remains uncloned; however, encountering the same repository subsequently will prompt the worker which flagged the incoming job to accept it.

An illustrative example of an execution environment comprising three worker nodes, a single master node, and six jobs is depicted in Figure 11 below. In this illustration, jobs associated with repositories within the worker nodes' storage, denoted by "C:", have already been accepted, whereas the fourth job in the general job queue has been observed by all workers but not yet accepted. All workers have noted this repository (indicated by "F:") and the next node will, on encountering *job4* again, check its flag and accept the job. This can happen when:

- executing the workflow for the first time, all worker nodes reject repository-related jobs due to the absence of local clones,
- a node processes a job it has seen before, even when another node holds the resource locally but is currently occupied, leading to unnecessary redundant clones.

While all three frameworks possess the capability to execute the workflow and collect the resultant data, each exhibits specific limitations associated with the execution of this type of MSR workflows. The subsequent section will provide a detailed examination of the collected metrics and will determine the baseline approach for our research.

4.5 Summary of Results

Table 3 presents a detailed comparison of the end-to-end execution times for the LID MSR workflow across Crossflow, Apache Spark, and Apache Flink. The

[5]The details of executing MSR example using Crossflow can be found in `https://github.com/ana-markovic/lid-msr/tree/master/baseline-modeldriven`

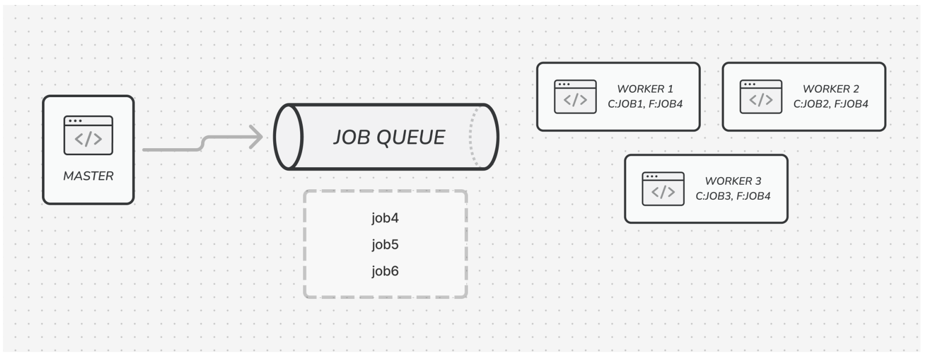


Figure 11: Snapshot of Crossflow's state for GitHub mining workflow

data clearly indicates that Crossflow surpassed Spark and Flink in performance in 80% of the test cases. Figure 12 summarises this data according to different worker speed configurations. On average, Crossflow exhibited execution speeds that were 1.14 and 5.31 times faster than those of Spark and Flink, respectively. It is particularly noteworthy that this speed advantage is most pronounced in scenarios where the worker nodes exhibit heterogeneous physical characteristics.

Worker Conf	**Job Conf**	**Time (s) Flink**	**Time (s) Spark**	**Time (s) Crossflow**
all_equal	all_diff_equal	325	306	285
all_equal	all_diff_large	802	1009	660
all_equal	all_diff_small	83	108	87
all_equal	repetitive_large	1365	196	412
all_equal	repetitive_small	169	138	95
one_fast	all_diff_equal	321	232	139
one_fast	all_diff	720	724	374
one_fast	all_diff_small	78	91	69
one_fast	repetitive_large	1372	141	221
one_fast	repetitive_small	123	128	85
one_slow	all_diff_equal	2559	673	803
one_slow	all_diff_large	9925	2068	1356
one_slow	all_diff_small	1200	851	123
one_slow	repetitive_large	1750	1794	636
one_slow	repetitive_small	2244	1422	414
slow_fast	all_diff_equal	2545	1558	1124
slow_fast	all_diff_large	10058	1477	1382
slow_fast	all_diff_small	1202	765	71
slow_fast	repetitive_large	1755	769	701
slow_fast	repetitive_small	2204	801	410

Table 3: Execution times (in seconds) for different worker and job configurations across Flink, Spark, and Crossflow.

Particularly, Crossflow's "opinionated" nodes introduce a nuanced layer of task allocation based on worker preferences, a feature absent in both Spark and Flink. In Spark and Flink, the master node uniformly distributes tasks without differentiating worker capabilities. This uniformity can result in delayed task

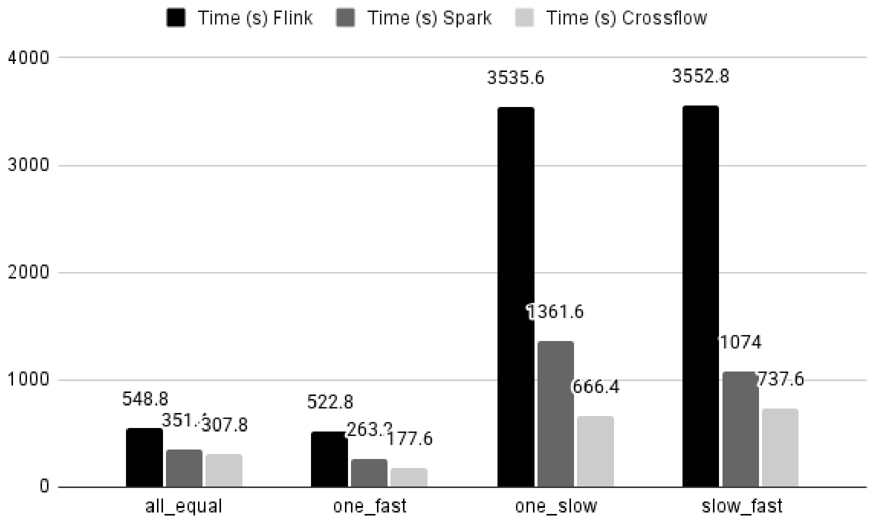


Figure 12: Average end-to-end execution time per worker speed configuration

completions, especially when slower workers are allocated larger repositories, as demonstrated in configurations such as *all_diff_large slow_fast*. Crossflow mitigates this inefficiency by allowing workers to selectively accept tasks based on their current availability and filesystem contents, thereby significantly reducing completion times for tasks that would otherwise throttle the entire workflow.

To further discuss the results, Figure 13 highlights a subset of numbers obtained from four experimental scenarios. The chart is structured into distinct column groups, each representing various worker and job configurations utilised to execute the workflow. For example, in the first column group, where the network speed and processing capabilities of one worker are substantially inferior to the others and the repositories for analysis are predominantly small (less than 100MB), Spark takes 6.91 times longer to complete the workflow compared to Crossflow. Under these identical conditions, Flink is found to be 9.75 times slower than Crossflow. This stark contrast underscores the inefficiencies in the scheduling mechanisms of Spark and Flink when confronted with heterogeneous worker capabilities.

Unlike Spark, which allocates all cloning jobs in advance without considering resources that become locally available during execution, Crossflow schedules repository cloning jobs sequentially. This approach accounts for repository clones that become accessible during runtime, thereby optimising the resource utilisation dynamically and enhancing overall efficiency. In the second column group, representing scenarios where all workers possess homogeneous characteristics and are tasked with processing larger repositories (exceeding 500MB), Crossflow outperforms Spark by a factor of 1.52 and Flink by a factor of 1.21.

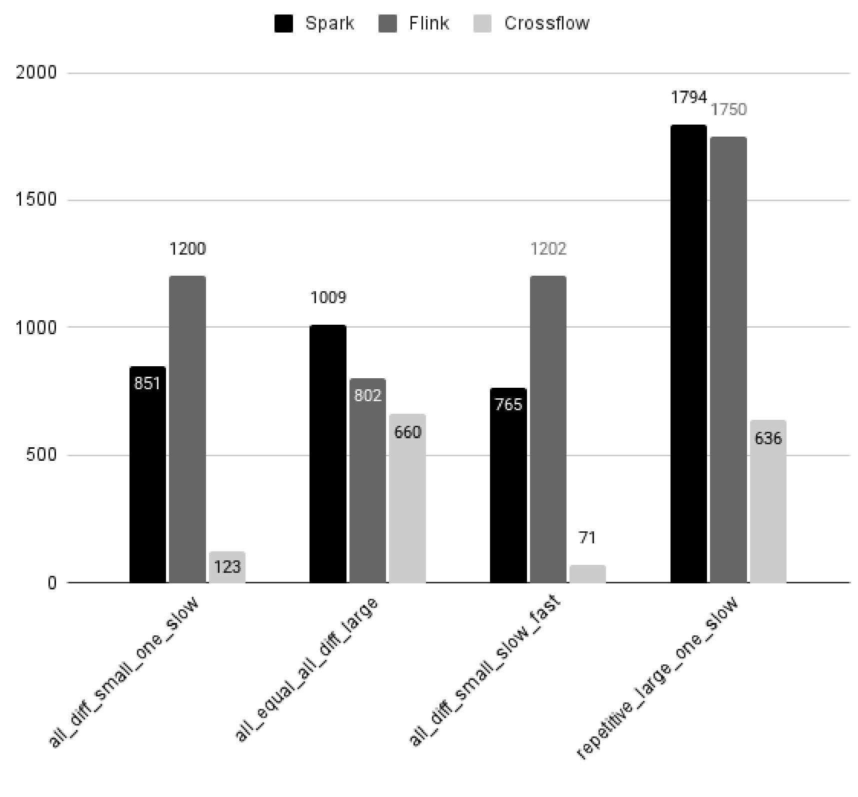


Figure 13: Execution times of specific example scenarios for Spark, Flink and Crossflow

Although the performance gap is narrower in this configuration, it is evident that Crossflow's sequential scheduling and resource-aware task allocation offers a distinct advantage, resulting in faster workflow completion times even in a uniformly distributed computing environment.

The third column group addresses the runtime for a non-repetitive dataset processed by workers with heterogeneous characteristics. In this scenario, Crossflow achieves 10.77-16.92 times faster execution, which further supports the efficacy of its sequential scheduling approach. Conversely, the fourth column group involves varying network and processing speeds while handling a repetitive dataset, wherein multiple jobs necessitated the same repository. In this complex scenario, Crossflow exhibits a 2.75-2.82 times speedup compared to both Spark and Flink, which can be attributed to its ability to dynamically adjust task allocation based on real-time resource availability and worker preferences.

When compared to Apache Spark and Apache Flink, Crossflow exhibits superior performance in executing the exemplary MSR workflow described in Section 1.2. Specifically, in one particular example and setup, Crossflow delivers up to 16.93 times faster processing than Flink and up to 10.77 times faster than Spark, primarily due to differences in scheduling methodologies. Considering Crossflow's substantial performance advantages over the two general-purpose frameworks, particularly in environments characterised by heterogeneous worker characteristics, we have selected Crossflow as the foundational framework for our proposed enhancements. The forthcoming sections will delve into methodologies aimed at augmenting Crossflow. Key areas of focus will include advanced scheduling algorithms that further exploit "opinionated" nodes to optimise job allocation in heterogeneous environments during runtime and techniques to increase adaptability of workers when it comes to dynamically changing environments. These enhancements, if successfully integrated, are expected to further distinguish Crossflow from both Spark and Flink in the context of executing LID MSR workflows.

4.6 Threats to Validity

The selection of an appropriate baseline is a fundamental aspect that underpins the integrity and relevance of any comparative study. In the context of this research, where the objective is to evaluate repository mining workflows using Apache Spark, Apache Flink, and Crossflow, it is imperative to acknowledge potential threats to validity that may arise from the baseline selection process.This section discusses potential threats to the validity of the experimental findings, following the established categories from Wohlin et al. [64]: conclusion validity, internal validity, construct validity, and external validity.

4.6.1 Internal Validity

Internal validity refers to whether observed effects can be attributed to the experimental variables (i.e., the choice of framework) rather than external or confounding factors.

- **Controlled environment.** All frameworks were evaluated using semantically equivalent MSR workflows within a local, simulated environment. Worker parameters and resource separation were explicitly defined to minimise unintended variance.
- **Framework-specific effects.** Despite efforts to equalise conditions, differences in framework internals — such as task scheduling logic, default caching strategies, or error handling — may influence observed performance independently of the framework's support for data locality.

4.6.2 Construct Validity

Construct validity concerns how well the metrics and experimental setup represent the concepts being studied.

- **Simulated timing variability.** While the experiments ran in a controlled local environment, simulated worker parameters (e.g., network speed, processing speed) and a controlled noise parameter were used to emulate heterogeneity. Although this enhances reproducibility, the simplified timing model might omit nuances present in real-world distributed systems.
- **Operationalisation of data locality.** Data locality was measured using a combination of cache hit/miss ratios and the volume of data transferred between workers (in MB). These are reasonable and commonly used proxies, but they may not fully capture qualitative aspects of data placement or locality awareness mechanisms internal to each framework.

4.6.3 External Validity

External validity refers to the extent to which results generalise beyond the experimental setup.

- **Limited data sample size.** The experimental setup explored a fixed number of repository sets and system configurations, as described earlier in Sections 4.3.1 and 4.3.3. While this provides meaningful variation in workload characteristics and execution environments, the dataset remains limited in scope, and broader statistical inferences should be made with caution.
- **Scope of workloads.** The repository sets were deliberately chosen to vary in structure, size, and data overlap. However, the findings primarily apply to MSR scenarios with repository or commit-level analysis. Workflows that do not involve data reuse, such as stateless transformations or purely sequential access patterns, may not benefit from locality-aware scheduling, and may instead favour alternative scheduling strategies focused on load balancing or resource fairness.

- **Controlled vs. real-world conditions.** The simulated environment approximates a distributed system by emulating download times and isolating worker storage, but lacks elements such as hardware diversity, network contention, and runtime faults. This abstraction supports controlled comparison but limits generalisability to production-scale systems.

5 Resource-Registry: a Worker Preference-Based Scheduling Algorithm

The objective of this chapter is to explore a novel scheduling approach for Crossflow, wherein worker preferences and capabilities are integrated into the job allocation procedure. In the current version of Crossflow, the scheduling framework operates on a straightforward producer-consumer model as described in Section 2.2.5. The master node functions as the job producer, responsible for the publishing jobs to job queue, while the worker nodes act as consumers, tasked with job execution. This mechanism, however, lacks any form of directed job allocation, wherein specific jobs are assigned to the most suitable workers based on predefined preferences or capabilities. This shortcoming can lead to extended execution times if, for example, a worker is given a job that necessitates a repository already cloned by a different worker that would thus be able to execute the job more efficiently.

5.1 Resource-Registry Approach

To address **RG2**, the proposed enhancement to Crossflow's scheduling mechanism is encapsulated in the Resource-Registry approach. The primary motivation behind this approach is to prevent the non-local assignment of jobs to workers when possible, thereby enhancing efficiency and performance. In the context of executing jobs that involve cloning and analysing software repositories, the absence of a direct job assignment protocol poses an efficiency challenge. Ideally, to optimise overall execution time and minimise data transfer costs, incoming jobs should be assigned to workers who have previously cloned the required repositories. This approach leverages existing data locality, thus avoiding redundant cloning processes which are time-consuming. The core issue arises when an optimal worker — one who has already cloned the needed repository — is preoccupied with another job. In the the Crossflow's default non-directed model relying on general job queue, the incoming can be allocated to an alternative worker. This new worker must perform the cloning operation afresh before proceeding to the analysis phase. This results in increased execution times and higher data transfer overhead compared to waiting for the more suitable, although currently occupied, worker to become available. For instance, if *Worker A*, which has repository R cloned, is busy, and the incoming job requiring repository R is assigned to Worker B, this worker must first clone repository R before analysis can begin. The cloning process incurs additional time, and this combined with the analysis could exceed the total time it would have taken had the system waited for *Worker A* to finish the ongoing job. Thus, the lack of directed job assignment can lead to suboptimal utilisation of resources, increased execution times, and elevated data transfer costs, ultimately impeding overall system efficiency.

To mitigate this issue, the Resource-Registry approach proposes a database (a key-value store) accessible to both the master and worker nodes, which facilitates the exchange of worker preferences and capabilities with the master node. This database serves as a critical intermediary, ensuring that jobs are assigned in accordance with the specific skills and preferences of available workers. To illustrate, consider the use case involving GitHub repository mining described in Section 1.2. In this case the proposed database could be used to enhance data locality as workers could indicate which repositories they have already cloned and are thereby positioned to inspect again without the redundancy of re-cloning. This information could then be used in guiding the master node's scheduling decisions.

The introduction of a key-value database is integral to implementing this approach. Such a database enables tracking worker capabilities and preferences. The master node associates each job in the workflow with a unique identifier, referred to as *jobID*, which serves as the key in the key-value store. Correspondingly, the value for each key would be a set of workers who are either skilled at or interested in handling the specified job. For instance, in the context of GitHub repository mining, the *jobID* may be represented as the URL of the repository in question. The corresponding value in the key-value store would then comprise the list of workers who have previously cloned or inspected the repository, thereby enabling tailored job scheduling in order to reduce redundancy and optimise resource utilisation.

Similar to the methodology presented in [65], this job scheduling approach is implemented through extending Crossflow's default use of general queues for scheduling with additional dedicated job queues. Each worker maintains its own dedicated job queue into which the master node can allocate jobs aligned with the worker's declared preferences. The pre-existing general job queue in Crossflow serves a queue where the master node publishes incoming jobs for which no worker has expressed specific interest or preference. Workers prioritise their directly assigned jobs (i.e., those for which they have shown interest). However, should the workers become idle and have no jobs in their dedicated queues, they will proceed to retrieve and execute the first available job from the general jobs queue.

For the Resource-Registry scheduler, the dedicated queue threshold is set to infinity. This configuration is due to the fact that scheduling decisions are predicated on the preferences recorded by workers in the database, rather than the idle status of worker nodes. Consequently, if a worker *w1* has declared a preference for jobs associated with repository *r1*, those jobs will be assigned to *w1* irrespective of the availability of other workers or the job counts in their respective queues. Imposing a limit on the number of jobs assigned to each worker within this approach can adversely impact the overall execution time. This is because the pause in job allocation does not affect the eventual assignment of a job to a specific worker; it merely defers the allocation, thereby potentially leading to a prolonged execution time.

The execution flow in this adapted Crossflow framework can be encapsulated in the following hypothetical scenario:

1. **Workflow initialisation:** The master node (**m1**) initiates the workflow

and reads the input CSV with technologies. Based on technologies, it queries GitHub for repositories of interest and instantiates jobs associated with these repositories.

2. **Registry check: m1** queries the key-value database. Assuming that the workers were initiated with clean slates, meaning their local file systems did not store any pre-existing data, the database returns an empty set, confirming the absence of workers with pre-cached resources.

3. **Job allocation:**

    - All workers are ready and waiting for incoming jobs.
    - An incoming job *j1* associated with the repository *r1* (**j1-r1**) is published to the jobs queue and first gets dequeued by worker **w1**. **w1** commits to performing the job (**j1-r1**) and is now busy.
    - Worker **w1** records its cloning of the repository **r1** in the registry.
    - Subsequently, another job associated with repository *r2* (**j2-r2**) is published to the jobs queue and gets dequeued by **w2**. **w2** commits to performing the job (**j2-r2**) and is now busy.
    - Worker **w2** records its cloning of the repository **r2** in the registry.

4. **Direct job assignment:** When **m1** encounters a subsequent job requiring repository **r1**, it queries the registry, identifies **w1** as already-possessing **r1**, and sends the job downstream to the dedicated job queue of **w1**.

5. **Registry maintenance:** The master node **m1** periodically receives heartbeat signals from the worker nodes and utilises this information to maintain the accuracy of the Resource-Registry. If an arbitrary worker node ceases to function or requires a restart, its records are removed from the database. Should the previously non-functional worker node become operational again with its local file system data intact, it will automatically update the Resource-Registry upon startup.

Thus, the Resource-Registry refines the scheduling process by incorporating a feedback loop wherein job assignment is guided by real-time worker capability data. This algorithm is executed across both master and worker nodes, as illustrated in Listings 1 and 2. The algorithm defined in Listing 1, "Resource-Registry Scheduler - master side", handles job allocation by explicitly considering worker preferences based on the resource requirements of each job. Commencing with the *sendJob* function on line 4, it begins by retrieving a list of workers with a preference for the job's resource from the *ResourceRegistry* (line 5). If the result is empty, indicating no worker preferences are established, it defaults to sending the job to a general jobs queue (line 7). In a LID MSR workflow this could happen when, for example, the workflow has been initialised but workers have no pre-cached repository clones in their local file systems and are yet to perform jobs and report their clones to the registry. Conversely, if workers with preferences are available (i.e. workers that already have a local copy of the required repository), it calls the *getPreferredWorker* function to identify an appropriate worker (line 9). This function, defined on line 14, iterates over the list of candidate workers (line 15) and selects the least burdened node from this list (lines 15-16). Upon selecting a preferred worker, the job is assigned to this worker via the *consumeJob* method (line 10). This approach effectively balances resource-specific job allocation and falls back to the general job queue when necessary.

Listing 2, titled "Resource-Registry Scheduler - worker side", describes the process executed by a worker when consuming a job, along with the startup sequence to initialise local resources. The primary *consumeJob* function begins on line 5, where worker updates the *ResourceRegistry* to associate the job's resource and the current worker ID (line 8). It proceeds to process the job, storing the output (line 6). The *sendJobResult* function is then invoked to return the job result to the master node (line 7). Complementarily, the *onStartup* function, beginning on line 11, scans local resources (line 11) and iterates over each resource (line 12), registering them with the *ResourceRegistry* using the current worker ID (line 13). Finally, it establishes a connection to the general job queue to participate in job processing for non-preferred jobs too (line 15).

Listing 1 Resource-Registry scheduler - master side

```
 1: global variables
 2:     ResourceRegistry, global map
 3: end global variables
 4: function SENDJOB(job)
 5:     workersWithPreference = ResourceRegistry[job.resource];
 6:     if workersWithPreference = empty then
 7:         sendToGeneralQueue(job);
 8:     else
 9:         preferredWorker = getPreferredWorker(job, workersWithPreference);
10:         preferredWorker.consumeJob(job);
11:     end if
12: end function
13:
14: function GETPREFERREDWORKER(job, workers)
15:     leastBusyWorker = findLeastBusyWorker(workers);
16:     return leastBusyWorker;
17: end function
```

While much of the code is generalised, it is the concrete implementation of the *getPreferredWorker(job, workers)* that defines the behaviour of the scheduler. This function incorporates the specific logic required to select the most suitable worker for a given job. It queries the key-value store to retrieve a list

Listing 2 Resource-Registry scheduler - worker side

```
global variables
    $ResourceRegistry$, global map
end global variables
function consumeJob(job)
    ResourceRegistry[job.resource].add(currentWorkerId);
    jobResult = process(job);
    sendJobResult(jobResult);
end function

function onStartup
    localResources = scanLocalResources();
    for resource in localResources do
        ResourceRegistry[resource].add(currentWorkerId);
    end for
    connectToGeneralQueue();
end function
```

of workers who have declared their preference for handling the job identified by *jobID*. If such a worker exists, the job is sent directly to them; otherwise, it is passed as per the default behaviour (sent to general job queue to be processed by the first worker that becomes idle). Hence, this is the method to be changed if one was to address a specific use case. For instance, the *getPreferredWorker* function can incorporate additional checks, such as minimum RAM requirement or specific job type worker is supposed to run. Any changes to the *getPreferred-Worker* are to be specified by the application developer.

For LID MSR workflows and associated challenges that motivated this research, the Resource-Registry serves as a mechanism for workers to declare the repository clones they already possess locally. Leveraging the Resource-Registry enables the master node to direct incoming jobs to workers who have previously cloned the repositories relevant to these jobs. By aligning job allocation with worker capabilities, the system optimises resource utilisation, mitigates redundancy, and enhances overall performance. Specifically, assigning jobs to workers already in possession of the required clones reduces the communication overhead by eliminating the initial round of job rejections and ensures that jobs are executed locally whenever possible, regardless of workers' availability at the time of job assignment.

However, several challenges accompany this approach. Initially, the key-value store must be populated with data regarding worker capabilities, a process that can introduce startup overhead. Additionally, maintaining the accuracy of the Resource-Registry through dynamic updates to reflect real-time changes in worker capabilities can also impose overhead.

The subsequent section evaluates the Resource-Registry scheduler by extending the Crossflow framework to incorporate a key-value database and a dedicated job queue for each worker. Experiments conducted based on an MSR application seek to determine whether this scheduler's potential benefits can surpass the performance metrics of the Crossflow baseline.

The current implementation of the Resource-Registry does not include mechanisms for reassigning jobs that fail during execution. This behaviour is consistent with Crossflow, the underlying framework on which the scheduler is built, which records and reports job failures but does not attempt to reallocate them. As a result, failed or unresponsive jobs remain unprocessed unless manually resubmitted. This is a known limitation of the current approach, which is to be addressed as part of future work.

5.2 Resource-Registry Evaluation

To evaluate the Resource-Registry approach against the Crossflow baseline, and thereby fulfil the objectives set by **RG3**, we employed the same MSR example described in Section 4.1, combined with the local experimental setup outlined in Section 4.3. We configured five worker nodes to work with a single master node. This evaluation encompassed various worker and job configurations depicted in Sections 4.3.1 and 4.3.2 to ensure a comprehensive analysis. The detailed instructions for how to setup and run the experiments using Crossflow with the Resource-Registry scheduler are shared as part of the research artefact [6].

Workers' preferences were expressed through repositories that had been previously cloned and analysed. The principal metric examined was total execution time. Additionally, we considered worker usage percentage, cache misses, and data load, all of which are vital for improved data locality, as detailed in Section 3.4. Figures 14 present the average end-to-end execution time, grouped by job and worker speed configuration, respectively. These aggregations were analysed to assess the performance of the Resource-Registry (RR) scheduler for LID MSR workflows, focusing on the influence of workload's locality intensity — whether involving repetitive repository jobs — and performance in heterogeneous environments.

In Figure 14a, workflows using the Resource Registry scheduler showed execution times 1.37 times shorter for workload types involving unique repositories with equal size distribution or predominantely large, whereas for the *repetitive_large* and *all_diff_small* job configurations execution times were on average 2.37 times longer. Figure 14b illustrates that the Resource-Registry scheduler performs worst in heterogeneous environments, with execution times 17% slower on average. Conversely, in homogeneous setups, the scheduler performed nearly the same as the baseline and it recorded a 28.20 seconds variance in execution time on average, which can be attributed to the noise in execution.

Figure 16 indicates that data load and cache misses are almost the same (± 3) for workflows running with either scheduler. Groupings by job configurations report that cache misses for workflows processing varied repositories are

[6] Instructions for running experiments using Crossflow with the Resource-Registry scheduler `https://github.com/ana-markovic/lid-msr/tree/master/rr-modeldriven`

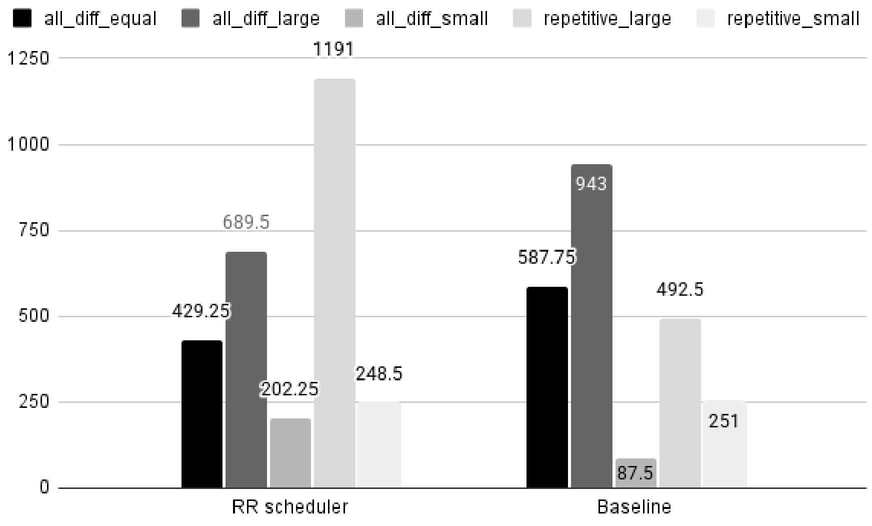
fairly consistent, regardless of worker node characteristics, as is the data load. This was expected, as the cache misses and data load are directly linked to incoming jobs and are not dependent on the worker environment. Thus, a part of the observed speedups for job configurations involving unique repositories can be attributed to eliminating rejection cycles for first-time repository jobs. With the Resource-Registry scheduler, in contrast to the Baseline, when a job is submitted to the general jobs queue and no worker node possesses the corresponding clone, an arbitrary worker node will process the job without rejection. Another aspect leading to more efficient executions are variations in average worker usage percentages as shown in Figure 15.

For instance, the Resource-Registry scheduler achieved a 1.36 times speedup for workflows processing unique repositories with equally distributed sizes (denoted as *all_diff_equal*), which correlated with an 11% higher worker usage compared to the Baseline. Likewise, for unique, predominantely large repository workflows, a 17.90% higher worker usage translated to a 1.38 times speedup. Conversely, for workflows involving large repetitive repositories, the Resource-Registry scheduler increased execution time by 2.31 times on average, attributable to a 24.8% decrease in worker usage percentage.

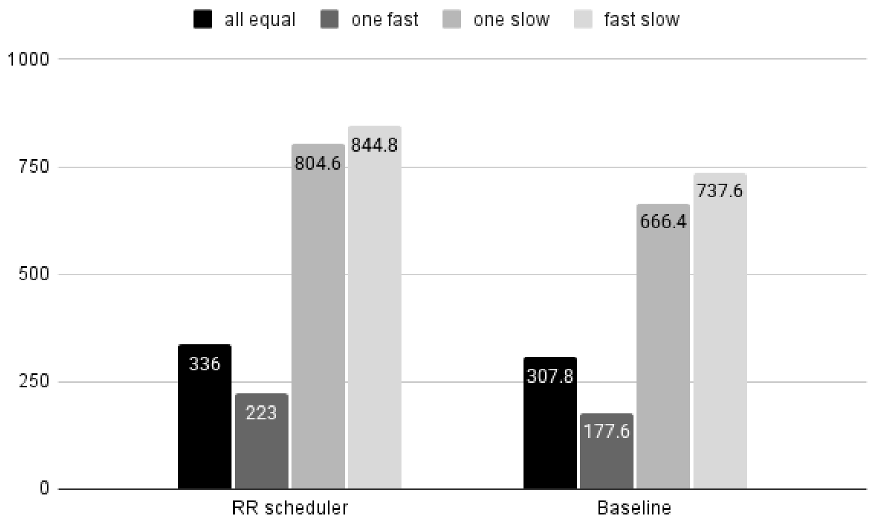
Let us examine the scenarios where the Resource-Registry scheduler demonstrated inferior performance relative to the Baseline. The differences were particularly evident in configurations involving large, repetitive repositories and workflows executed in heterogeneous environments where one worker was significantly slower than the rest. Tables 4a and 4b provide the total execution time in seconds for each scenario using the respective schedulers.

In Table 4b, the *repetitive_small* job configuration was executed within a similar timeframe, considering the variability in network and processing speeds of workers, whereas the most significant deviation occurs with the *repetitive_large* job configuration. This performance gap can be attributed to the method of job assignment employed by the Resource-Registry scheduler. Once a worker begins processing a job, it marks itself as interested in related jobs involving the same repository. Consequently, the master node directs all related jobs to this worker. While this approach complies with data-locality principles, it can negatively affect overall execution time by monopolising the worker's capacity for such jobs, leaving other workers idle. This monopolisation prevents other workers from expressing preference for the same repository, as they will not be assigned jobs that require it on the account of the worker that expressed the interest first, thereby nullifying potential load balancing benefits unless workers' storage contents are pre-populated ahead of execution.

To validate this assertion, let us consider the data locality details for this particular job configuration processed in a heteroeneous environment, as presented in Table 5. Execution of workflows with both schedulers resulted in 37 cache misses, however the workflow relying on the Resource-Registry recorded 637MB less of downloaded data. Nevertheless, the cache miss and work time distribution among worker nodes in the two experiments underscore the aforementioned issue. Specifically, worker *w5-slow* experienced only 1 cache miss with the Resource-Registry scheduler and recorded a total work time of 2212

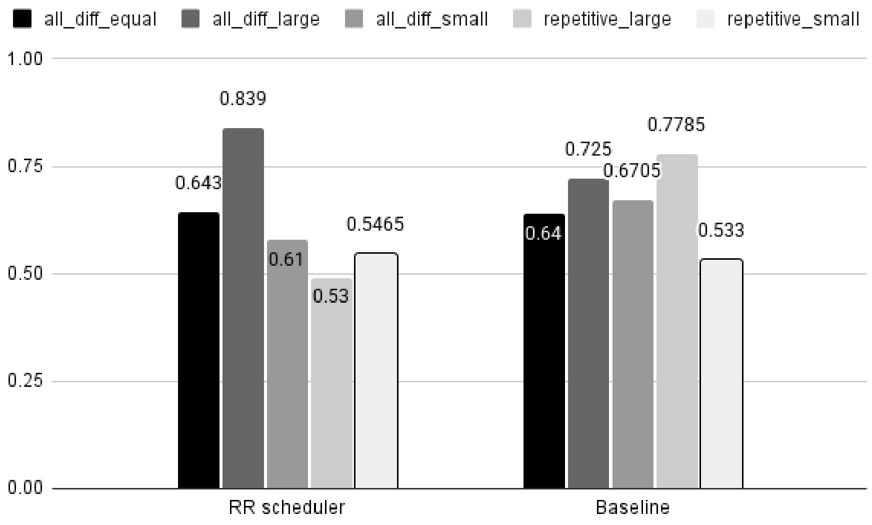


(a) Average total execution time (in seconds) per job configuration

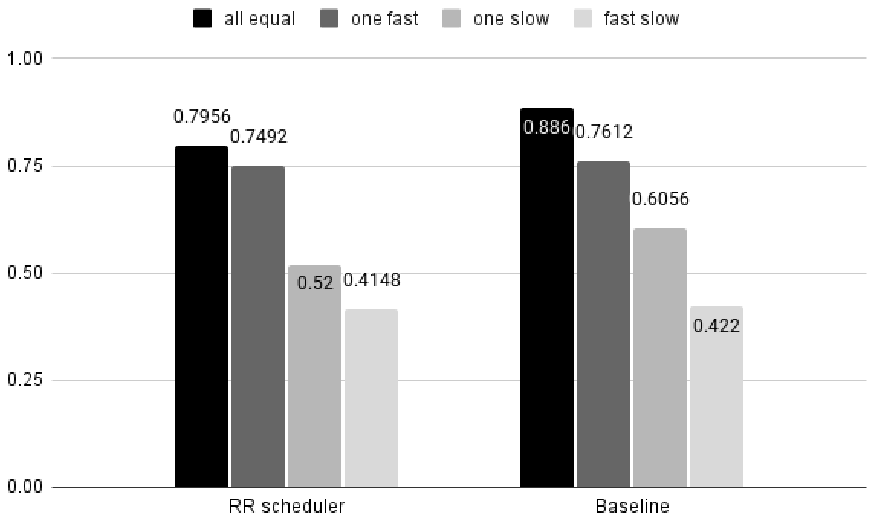


(b) Average total execution time (in seconds) per worker speed configuration

Figure 14: Accumulated results for end-to-end execution time (in seconds)

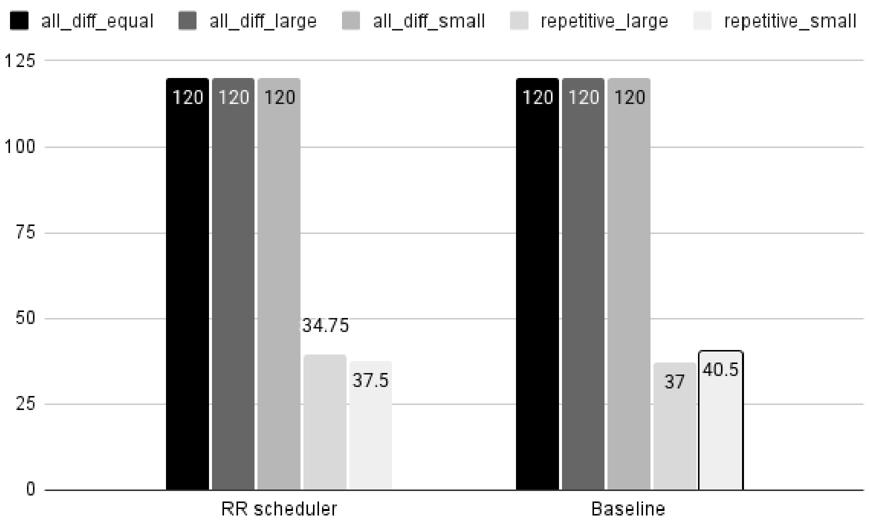


(a) Average worker usage per job configuration

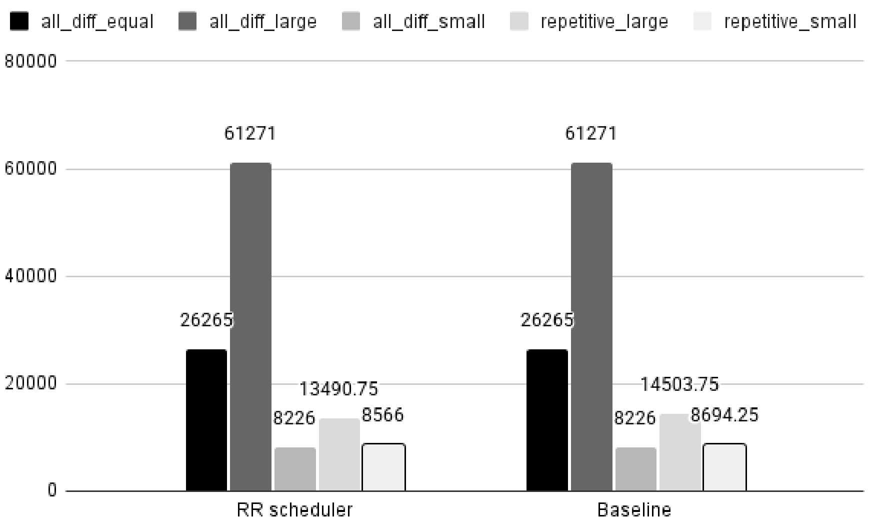


(b) Average worker usage per worker speed configuration

Figure 15: Accumulated results for worker usage (percentage, illustrated as a number between 0 and 1)



(a) Average cache miss count (number) job configuration



(b) Average data load per job configuration (MB)

Figure 16: Accumulated results for cache misses (number) and data load (MB)

seconds, compared to the Baseline's 1 cache miss and 695 seconds of execution time for the same worker. This indicates that *w5-slow* primarily handled local jobs, evidenced by the prolonged work time, however, the Baseline scheduler achieved better load balancing on the account of slighlty increased data load. To support this claim, we observe that the *w1-fast* remained idle for most of the execution time with the Resource-Registry scheduler, thereby necessitating 3.16 times more seconds to complete the workflow compared to the Baseline.

This trend is observed in the repetitive workflow scheduled by the Resource-Registry in all the worker environments, as depicted in Table 4a. Notably, results in the second, third and fourth rows, which are obtained from workflows executed on heterogeneous worker nodes, confirm that the Resource-Registry scheduler can prolong execution times when a slower worker picks up a repetitive job. Subsequently, this slower worker must handle all future jobs requiring the same repository, regardless of other workers' idleness, thus extending the execution time due to enforced data-locality principles.

	Resource Registry	**Baseline**
all equal	538	412
one fast	474	221
one slow	1534	636
fast slow	2218	701

(a) End-to-end execution times (in seconds) for workers processing *repetitive_large* job configuration.

	Resource Registry	**Baseline**
all_diff_equal	650	803
all_diff_large	831	1356
all_diff_small	604	123
repetitive_large	1534	636
repetitive_small	404	414

(b) End-to-end execution times (in seconds) for environments with a single slow worker processing different job configurations.

Table 4: Comparison of Resource-Registry scheduler and Baseline results across configurations in which the Resource-Registry scheduler underperformed.

	Resource-Registry			**Baseline**		
Worker	**Time (s)**	**Cache miss**	**Data load (MB)**	**Time (s)**	**Cache miss**	**Data load (MB)**
w1-fast	63	27	7575	201	26	8252
w2	451	3	1113	223	3	941
w3	435	2	934	236	5	1911
w4	92	4	1808	221	2	1508
w5-slow	2212	1	910	695	1	910

Table 5: Comparison of Resource-Registry scheduler and Baseline performance metrics for workers with heterogeneous physical characteristics processing the *repetitive_large* job configuration.

To summarise, the Resource-Registry effectively addresses the issue of all workers rejecting jobs that lack preferred attributes, thereby enhancing data locality and striving for entirely local execution. However, it has limitations, particularly in load balancing and managing the physical differences between worker nodes. Although the Resource-Registry method results in faster execution in some test scenarios, whether the most suitable node processes a job can depend on various external factors. These primarily include the order and timing of incoming jobs and the activity levels of other nodes, with the predominant factor being worker preference. Specifically, if a worker has indicated a preference for certain types of jobs, it will be assigned these jobs, even if other workers in the execution environment could perform these tasks more efficiently, for instance, by completing them faster. The following chapter introduces an approach that continues to consider data locality but also addresses the limitations identified in the Resource-Registry method.

6 Distributed Scheduling with the Bidding Scheduler

With regards to **RG2**, in order to make use of opinionated nodes and prevent suboptimal worker selection that can occur when scheduling work solely based on worker preferences, we developed a new scheduling mechanism that takes both data locality and physical characteristics of worker nodes into account. Again, it is proposed as an improvement to the Crossflow framework discussed in Section 2.2.5, however, it presents a general solution that could be integrated with other data processing engines too.

6.1 Background: Dynamic Job Scheduling in Big Data Applications

In Section 1.2, we highlighted the importance of considering data locality when scheduling jobs pertaining to the analysis of source code repositories. The issue of data locality has received substantial attention from the research community, and schedulers that achieve data locality have been proposed in the literature [66]. Nevertheless, the concern of data locality must be addressed in conjunction with the concern of load-balancing in task scheduling. This is because improving data locality — by assigning more tasks to nodes that possess the requisite input data — can lead to an imbalanced load distribution among nodes, thereby adversely impacting overall performance [65]. Consequently, one of the primary challenges of job allocation in master/worker environments is to distribute incoming jobs among workers in a manner that minimises job execution time while avoiding an imbalance of overloaded and underloaded workers [53]. This requires schedulers to achieve the right balance between data locality and load-balancing.

Furthermore, an additional challenge associated with MSR workflows is that data distribution, specifically the location of repository clones, is not predetermined, but instead emerges during execution. Prior to executing such workflows, it is possible that no local copies are available to worker nodes, necessitating the scheduler's data locality capabilities to become effective only once the analysis stage commences. In essence, the contents of workers' local storage will vary throughout the workflow execution, requiring the scheduling of incoming tasks to adjust to these fluctuations. The work in [65] deals with this problem by enabling job re-assigning during execution, particularly to worker nodes who may possess the necessary data locally. This approach also advocates a decentralised job scheduling, removing the explicit responsibility of task assignment from the master node. Instead, job allocation is entrusted to the worker nodes, which maintain separate queues for local and non-local jobs and implement logic to prioritise jobs for which they possess the necessary data locally. These workers can reassign non-local jobs to other workers more likely to have the required data, thereby enhancing both data locality and system throughput.

Similarly, the work in [67] delves into the application of the MapReduce framework with a distinct focus on optimising task scheduling to exploit the disparate capacities of various nodes within a cluster. The scheduler employed dynamically partitions the input data and subsequently allocates the resulting data blocks to nodes based on their respective processing capabilities. This allocation is contingent upon each node's estimated processing power, which is determined by metrics such as average job execution time and the availability of CPU power and RAM memory, all measured at regular intervals. This dynamic assessment ensures that the scheduler accommodates fluctuations in the performance of each node over time, allowing more powerful nodes to handle larger and more complex tasks, thereby optimising resource utilisation.

However, given that Big Data workflows can persist for multiple days or even weeks [68], not only might the contents of local storage change, but also the remaining storage capacity, CPU power, and network bandwidth available to worker nodes. Workers may exhibit variations in their physical characteristics prior to the start of execution, which is often referred as platform heterogeneity. In this context, heterogeneity is characterised by each node potentially having different physical parameters, such as data storage and processing units, coupled with the fact that Big Data analysis jobs may not have uniform data and computation requirements. The aforementioned scheduling strategies — Hadoop's default scheduler, Matchmaking, and Delay scheduling — demonstrate limited performance when applied to heterogeneous environments, which may result in diminished performance during workflow execution [67].

Collectively, this context establishes the groundwork for investigating job allocation techniques suitable for dynamic and heterogeneous environments in the realm of Big Data analytics.

6.1.1 Heterogeneous Environments and Volunteer Computing

Among the forerunners in the domain of heterogeneous environments is BOINC [69], a middleware system specifically designed for volunteer and grid computing. Initially developed at the University of California, Berkeley, the platform has proven to be crucial in leveraging otherwise underutilised computational power to facilitate a wide array of scientific projects. BOINC represents an open-source paradigm in distributed computing architecture, enabling researchers to access a global repository of computational resources.

The fundamental premise of BOINC is anchored in the concept of volunteer computing. In this model, individuals offer their idle computing resources to scientific initiatives, thus enhancing the collective computational capacity available for research. This approach builds upon the principles of distributed computing and parallel processing. Volunteer computing can be traced back to the concepts of grid computing and cloud computing, where computational tasks are distributed across multiple nodes to optimise resource utilisation and improve task completion efficiency. The system is composed of several key components:

- **Client software:** End-users install the BOINC client on their personal computers. The software manages the download, execution, and upload of computational tasks.

- **Server infrastructure:** Central servers distribute tasks to client machines and collect results. These servers handle application scheduling, data management, and result validation.
- **Project applications:** BOINC supports a variety of scientific applications, each encapsulated within its own project. Examples include SETI@home [70] for analysing radio signals from space and Rosetta@home [71] for protein folding simulations.

BOINC utilises a master/worker architecture, where the server functions as the master, responsible for distributing tasks to the worker clients. In terms of scheduling, the baseline policy implemented within BOINC is the Weighted Round-Robin (WRR) algorithm [72]. The WRR scheduling algorithm extends the classical Round-Robin approach by incorporating weight factors to manage varying priorities among tasks or projects.

In a traditional Round-Robin model, tasks are managed in a simple, sequential manner without regard to priority. In contrast, WRR allocates time slices to tasks based on their assigned weights, thereby enabling higher-priority tasks to secure a larger share of computational resources. The weight assigned to a task typically mirrors its importance, urgency, or the resource share specified by the project administrators.

Consider a project P with an assigned weight W_p. The number of time slices allocated to P within each scheduling cycle is proportional to W_p relative to the weights of other concurrent projects. The scheduling cycle is divided into discrete time slices, and projects receive a number of time slices correspond to their weights. For instance, if *Project A* has a weight of 3 and *Project B* has a weight of 1, then within a given cycle, *Project A* will receive three times more time slices than *Project B*.

This weighted allocation mechanism ensures that computational resources are systematically distributed in alignment with project priorities, thereby optimising resource utilisation and maximising overall system throughput. While BOINC could, in theory, distribute the tasks of cloning repositories and analysing code across multiple machines, this platform assumes that each job operates independently, requiring minimal inter-node communication and making it challenging to schedule work if cloning and analysing are configured as separate tasks. Moreover, it does not take into account the data locality, meaning that if workload requires scanning one repository more than once, there is no guarantee that the processing will be assigned to workers already possessing the local copy. The inefficiency becomes particularly pronounced when managing large volumes of data, such as large repositories, due to the logistical complexity and overhead associated with downloading this data to client machines, which may suffer from unstable network connections. Furthermore, BOINC projects utilise only a fraction of the RAM memory available on a given machine, thus the available resources may be inadequate to execute compute-intensive tasks, such as static code analysis, effectively.

6.1.2 Auction-Based Job Scheduling and Resource Allocation in Distributed Environments

The application of auction mechanisms for decentralised problem-solving is extensively documented in the literature on distributed load balancing and resource allocation within distributed computing systems [73]. In particular, the work by Malone et al. [74] utilises auctions to allocate tasks to computational resources, such as machines with varying processing speeds. Bids are formulated based on estimated completion times, which consider both the machine's speed and its current load. The task is then assigned to the machine with the earliest estimated completion time. However, it is noteworthy that in this study, the processing time estimates were part of the job's metadata — essentially an estimation provided by the user regarding the expected duration of the task. This estimate serves as a baseline for calculating the job's duration on each machine, indicating that discrepancies between estimated and actual processing times originate from the user's estimation errors, not the machine's functionality. The authors evaluated this auction-based approach against random task assignment through two variants: the "lazy" approach, which initiates auctions only when a processor is idle, and the "eager" approach, which permits continuous bidding regardless of processor activity. These comparisons were conducted within a simulated environment to evaluate scheduler performance under varying conditions, such as estimation errors (up to $\pm 100\%$ error in jobs), different machine speeds (machines were assigned one of eight predefined speeds) and job loads (by configuring job arrival rates and processing times).

Similarly, Ferguson et al. [57] present an auction model for load balancing in a distributed system. Jobs compete for resources (i.e., workers with CPU power) by placing bids. This approach introduces an additional parameter in bid construction by assigning a budget to each job at the beginning of the auction and it makes the bidding process iterative. Jobs can bid only within their budget constraints, with bids calculated based on service time and worker price. The worker determines the winner (the next job to process) by finding the bid that optimally balances service time and price — the objective being to achieve maximum value in the least amount of time. As with the work by Malone et al. [74], this auction model was evaluated in a simulated environment. However, this approach operates under a homogeneous environment assumption with no specific data being required for job processing, which poses a limitation in heterogeneous, data-intensive environments due to the lack of considerations for data movement and locality.

In [75], an adaptive scheduling approach is introduced through a dynamic reallocation mechanism. Here, workers can delegate tasks to other less burdened workers during execution. This task reassignment is facilitated via an auction mechanism, allowing workers to negotiate the completion price (i.e., the time required to finish a job) of incomplete tasks and delegate them to the worker node that wins the auction. This mechanism effectively reduces the workload of the most heavily loaded worker, thus decreasing the overall execution time of the Big Data workflow.

The concept of negotiation or auction-based job allocation is further explored in the context of the Internet of Things and High-Performance Computing systems. For instance, [76] proposes a bidding approach for incoming jobs, wherein the nodes bid for the allocation of incoming jobs based on their processing capacity (number of available cores). Each node aims to win the allocation of jobs to maximise the overall system profit. The bids reflect the computational power each node can offer for job execution. Additionally, this approach includes job pre-emption — a scenario where the execution of a low-value job can be paused to free up worker resources for a higher-value task. The paused job can be resumed when the necessary resources become available, collectively leading to reduced overall execution times and enhanced resource utilisation.

For resource allocation, Zhou and Bambos [77] propose an auction-based decentralised approach in client-server environments. In this model, clients (the bidders) submit job requests to a server (the resource provider), requiring resources such as CPU cycles, memory, or storage. Clients bid a specific amount they are willing to pay for the necessary resources, and the server allocates resources proportionally based on these bids. Each client's share of resources is proportional to their bid relative to the total bids. The approach is decentralised in way that the server does not require detailed knowledge of each client's job or resource needs, as the bids encapsulate the client's priority and willingness to pay.

Similarly, Zheng et al. propose an online auction mechanism for procuring Infrastructure-as-a-Service (IaaS) instances and scheduling parallel jobs in cloud computing environments. Users (the bidders) submit job requests to the cloud operator. Each user is job-oriented, meaning they focus on the completion of their jobs rather than specific resource details. The users bid for job execution with specific deadlines by submitting bids that include completion time (the deadline by which their job should be completed) and the willingness to pay (the monetary value they are willing to bid for completing their job within the specified deadline). This mechanism operates in real-time, making allocation decisions upon each user's arrival without knowledge of future requests.

6.2 Bidding Scheduler Approach

In this approach, which we will refer to as the Bidding Scheduler in the remainder of the thesis, we describe a job allocation method in which the distributed worker nodes influence scheduling decisions by reporting their internal state to the master node. The sequence diagram shown in Figure 17 illustrates a series of operations that enable continuous interaction between the master and worker nodes to distribute unprocessed jobs.

The following description explains each step of the process:

1. **Job publication initiation**
   The process begins with the master node attempting to assign each job in the job configuration. Each job can be published for bidding only provided

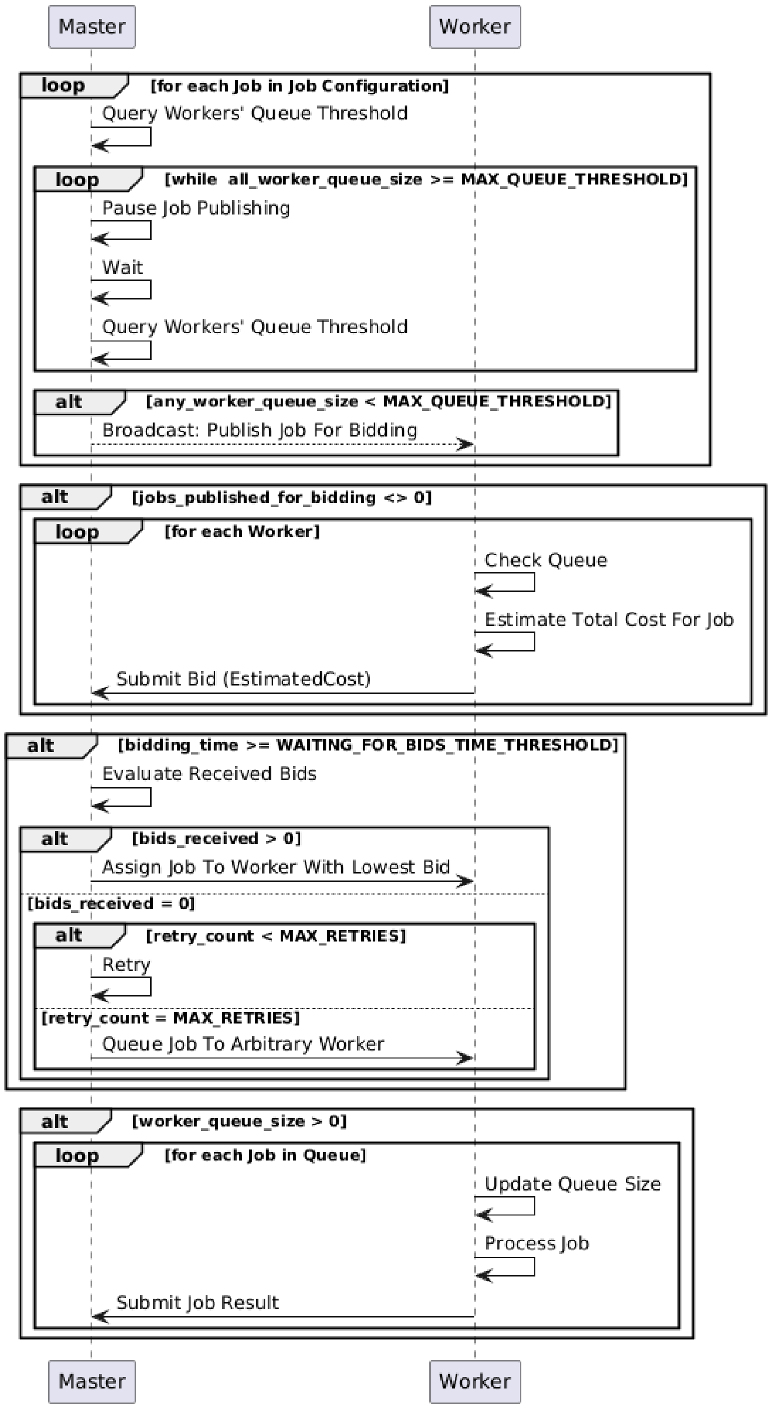

Master
Worker
loop [for each Job in Job Configuration]
Query Workers' Queue Threshold
loop [while all_worker_queue_size >= MAX_QUEUE_THRESHOLD]
Pause Job Publishing
Wait
Query Workers' Queue Threshold
alt [any_worker_queue_size < MAX_QUEUE_THRESHOLD]
Broadcast: Publish Job For Bidding
alt [jobs_published_for_bidding <> 0]
loop [for each Worker]
Check Queue
Estimate Total Cost For Job
Submit Bid (EstimatedCost)
alt [bidding_time >= WAITING_FOR_BIDS_TIME_THRESHOLD]
Evaluate Received Bids
alt [bids_received > 0]
Assign Job To Worker With Lowest Bid
[bids_received = 0]
alt [retry_count < MAX_RETRIES]
Retry
[retry_count = MAX_RETRIES]
Queue Job To Arbitrary Worker
alt [worker_queue_size > 0]
loop [for each Job in Queue]
Update Queue Size
Process Job
Submit Job Result
Master
Worker

Figure 17: Sequence diagram showing job scheduling with the Bidding Scheduler

that workers' queues are below the dedicated queue threshold (i.e. workers are not overloaded).

2. **Dedicated worker queue threshold monitoring**
   The Master node checks worker nodes present queue sizes by querying data about allocated and completed jobs. If all worker nodes queue sizes exceed the predefined dedicated queue threshold (MAX_QUEUE_THRESHOLD), a control flow is triggered to manage the overload. Specifically, the master node pauses job publication and waits a short period of time before re-querying the worker nodes' queue sizes.

3. **Job publishing**
   Whenever there is at least one worker node with a queue size below the maximum threshold (MAX_QUEUE_THRESHOLD), the master node proceeds to publish the job for bidding. This publication invites all workers to evaluate and bid on the job. In this process the master shares the job (with all the repository-related information as described in Section 4.2) with all the workers in the distributed environment by using a queue for broadcasting unprocessed jobs yet to be assigned.

4. **Bid submission by workers**
   Upon receiving the notification about the published job, worker nodes independently assess their capability to handle the job. This involves checking their current queue sizes and estimating the total cost for processing the job, which can include both data transfer and processing times. Each worker submits a bid to the master node, encapsulating the estimated cost.

5. **Master node bid evaluation**
   The master node evaluates all received bids to determine the most suitable worker for the job. This evaluation is performed within a specified timeout period (WAITING_FOR_BIDS_TIME_THRESHOLD). If bids are received within the timeout, the job is allocated to the worker presenting the lowest bid. In scenarios where no bids are received within the timeout, the master node may choose to either retry the job bidding process (until reaching a maximum retry threshold - MAX_RETRIES) or queue the job to an arbitrary worker if none of the workers submit bids for retried job publishings.

6. **Worker nodes job processing**
   Once a job is assigned, it enters the job queue of the designated worker node. The worker node then processes the job, which results in updating its queue size to reflect the new workload. Upon completion of the job, the worker node submits the job result back to the master node. This feedback loop enables the master to reassess the availability and load of worker nodes for subsequent jobs.

Process description

With the Bidding Scheduler, worker nodes are not responsible for accepting and rejecting jobs, but they enhance the traditional master/worker architecture by participating in the job allocation process and making scheduling a distributed decision-making activity. The main focus of the approach is that the bids the master receives provide it with insight regarding both data locality and previously committed workload and allow it to distribute work in a manner that will reduce the total execution time and data transfer costs. The master node broadcasts incoming jobs to all available worker nodes, and, in this algorithm, the workers create offers and bid for work. Their bids include their *estimate* of when they can get that job done, i.e. how much time they think they need to complete a particular job. Since worker nodes schedule tasks in FIFO order and do not take into account the jobs that are yet to be allocated after the job they are bidding for, this estimate must include the time to obtain any necessary resources and execute the new job, as well as the time to download resources and execute all unfinished jobs that have been previously allocated (i.e. bids won previously). Bidding for work is an asynchronous activity handled by a separate thread and the master waits for workers to make submissions within the threshold period, in our implementation set to one second. After that threshold period, the master looks into all the received bids before allocating the job to the worker who made a submission with the lowest estimate. The same applies to the situation when not all the workers sent their bids on time: the master chooses the bid with the lowest estimate of all received bids or waits for a short time window (parameterisable, and set as one second in our implementation) before re-publishing the job for bidding in case none of the workers submitted their estimates before the threshold period exceeded.

One of the main challenges in this approach is the nature of the workload, specifically its arrival pattern. The issue arises when all jobs arrive simultaneously. Although workers will bid for each job individually, when all jobs are available at once, the master is likely to distribute the entire workload before the workers complete the initial set of jobs. This creates a significant problem in dynamic environments, where the physical characteristics of the workers, such as network or processing speeds used for estimating job completion times, may fluctuate over time. In such cases, a worker might bid for a job at time (t), but only commence processing the job at time (t + x), when operating on different network speed, for example. This discrepancy can lead to substantial differences between the estimated and actual execution times, thus resulting in estimation errors. To mitigate these discrepancies, we use the dedicated queue threshold described in the Section 4.2. The threshold is used to set a maximum number of jobs (n) that can be queued for each worker to keep the master publishing more jobs for bidding, when there is still work awaiting allocation. This mechanism halts new bidding activities as soon as all workers reach (n) jobs in their queues and resumes it once any worker queue falls below the threshold. This strategy ensures that workers focus on completing current jobs and do not distribute additional work until at least one worker is on the verge of becoming idle, thereby minimising estimation errors resulting from performance variabilities over time. The value of (n) can be adjusted in code by the application developer, set to as low as 1 to delay bidding for jobs that require more processing time, or configured to a higher value for jobs that are quickly processed.

The Bidding Scheduler approach is designed to be flexible, with time estimates tailored to the specific nature of the jobs being processed. For software repository jobs, worker nodes will submit bids that include the time required to clone the repository onto the local filesystem and the time necessary to process it. Conversely, in scenarios where data transfer is not needed to perform computations, the estimate will exclude the time for data transfer. It is crucial to allow developers the autonomy to define their own time estimation formulas, as different workflows and tasks may require unique methods for accurate prediction.

Listings 3 and 4 contain pseudo-codes for the Bidding Scheduler from the master's and the worker's perspective when dealing with cloning and processing remote software repositories. In Listing 3, lines 2–7 define global variables critical to managing the bidding process. The *BidsStatus* map tracks the current bidding status (e.g., open or closed) for each job, while the *BidsValues* map stores bids submitted for each job. The *ActiveWorkers* array contains the list of currently available workers. The *JobStatus* map records the status of jobs, such as whether they are queued or being processed. The *RetryCount* map tracks the number of retry attempts for each job, ensuring retries are limited by the constant *MAX_RETRIES*, set to 3 in this implementation.

The *sendJob* function, defined in lines 10–15, initiates the bidding process when a new job arrives. It first publishes the job for bidding (line 11) and sets its status to *open* in line 12, indicating the bidding contest is active. The retry count for the job is initialised to 0 in line 13. To monitor the bidding process, an alarm is set in line 14, which will trigger after 1 second if the bidding is incomplete.

The *receiveBid* function, defined in lines 17–26, is responsible for processing bids as they are received. Line 18 identifies the job associated with the bid, and lines 19 and 20 store the bid in the *BidsValues* map. Line 21 invokes the *biddingFinished* function to determine whether the bidding contest should be concluded. If the contest is deemed complete, the status of the job is updated to *closed* in line 22, and the preferred worker is identified using the *getPreferred-Worker* function in line 23. Finally, the job is assigned to the selected worker in line 24 through the *sendToWorker* function.

The *biddingFinished* function, defined in lines 28–34, evaluates whether the bidding contest should end. It returns *true* if all active workers have submitted their bids or if at least one bid has been received and the contest has exceeded the 1-second threshold (lines 30–31). Otherwise, it returns *false*, allowing the bidding process to continue.

The *alarmTriggered* function, defined in lines 36–51, is invoked when the alarm set in *sendJob* expires. If no bids are received (line 38), the function checks whether the retry count for the job has reached the *MAX_RETRIES* limit. If retries remain (line 39), the job is republished for bidding in line 41, and the alarm is reinitialised for another 1-second period in line 42. If the retry limit is reached (line 44), the job is assigned to a random worker as a fallback in line 44, ensuring it does not remain unallocated indefinitely. If bids are received (line 46), the job status is updated to *closed*, and the job is allocated to the preferred worker in lines 48–349.

The *initialiseAlarm* function, defined in lines 53–55, sets a timer for the specified job and duration, linking it to the *alarmTriggered* function. This ensures that each job is monitored independently and handled appropriately based on its bidding progress.

The *getPreferredWorker* function, defined in lines 57–61, identifies the worker with the lowest estimated cost for the job. It sorts the received bids in ascending order based on cost in line 59 and returns the worker ID of the lowest bid in line 60.

Finally, the *sendToWorker* function, defined in lines 63–66, updates the job's status to *queued* (line 64) and instructs the selected worker to process the job (line 65).

The *alarmExpired* function, defined in lines 67–69, provides a minimal abstraction for checking whether the alarm timer associated with a job has elapsed. This helper function returns *true* if the timer has expired, indicating that the bidding period has reached its predefined 1-second limit. It is used within the *biddingFinished* function to determine whether the bidding process should be concluded based on elapsed time, without requiring explicit timestamp management.

Listing 3 Bidding Scheduler - master side

```
global variables
    $BidsStatus$, global map
    $BidsValues$, global map
    $ActiveWorkers$, global array
    $JobStatus$, global map
    $RetryCount$, global map
    $MAX\_RETRIES = 3$, constant
end global variables

function sendJob(job)
    publishForBidding(job);
    BidsStatus[job.id] = open;
    RetryCount[job.id] = 0;
    initialiseAlarm(job, 1s);
end function

function receiveBid(bid)
    job_id = bid.job_id;
    bids_for_job = BidsValues[job_id];
    bids_for_job.add(bid);
    if $biddingFinished(job\_id)$ then
        BidsStatus[job_id] = closed;
        w = getPreferredWorker(job_id);
        sendToWorker(job_id, w);
    end if
end function

function biddingFinished(job)
    bids_count = BidsValues[job.job_id].length;
    if $(bids\_count = ActiveWorkers.length) \vee (bids\_count > 0 \wedge alarmExpired(job))$ then
        return true;
    end if
    return false;
end function

function alarmTriggered(job_id)
    bids_count = BidsValues[job_id].length;
    if $bids\_count = 0$ then
        if $RetryCount[job\_id] < MAX\_RETRIES$ then
            RetryCount[job_id] += 1;
            publishForBidding(job_id);
            initialiseAlarm(job_id, 1s);
        else
            sendToWorker(job_id, randomWorker);
        end if
    else
        BidsStatus[job_id] = closed;
        w = getPreferredWorker(job_id);
        sendToWorker(job_id, w);
    end if
end function

function initialiseAlarm(job, time_duration)
    setTimer(job.id, time_duration, alarmTriggered);
end function

function getPreferredWorker(job_id)
    receivedBids = BidsValues[job_id];
    receivedBids.sort(bid as bid.cost_in_sec, ASC);
    return receivedBids[0].workerID;
end function

function sendToWorker(job_id, worker)
    JobStatus[job_id] = queued;
    worker.consumeJob(job_id);
end function
function alarmExpired(job)
    return timer for job has elapsed;
end function
```

The worker functionalities in this algorithm are grouped into two main functions. The first function, as depicted in lines 4 to 10, is responsible for estimating the workload and submitting a bid. In line 5, the worker estimates the time required to complete the current workload by aggregating the costs of previously won queued jobs. This estimation serves as the baseline for creating a new bid, as stated in line 6. The algorithm exhibits data awareness in line 7, wherein each worker is mandated to calculate the cost of acquiring the required data. The data transfer time could be determined by dividing the size of the repository by the current network speed, however the concrete formula is application specific and should be left to the developer to define. Minimum expenses are incurred when the worker possesses the data stored locally, which leads to lower time estimates and subsequently increases the chances of winning the bid. The bid amount is increased in line 8, according to the job processing time estimation. Similarly to the data transfer time, the processing time in the MSR example could be computed by dividing the repository size by the current processing speed, yet this is specific to different types of jobs and should be defined at the application level. Finally, the bid is transmitted to the master, as specified in line 9.

The algorithm's second function, described in lines 12 to 17, comes into play when the worker wins the bidding contest and is required to process the job. In line 13, the worker initiates the job processing by updating its status to *started*. The actual work is executed in line 14. The worker updates the current job status in line 15 and submits the result to the master node in line 16.

Listing 4 Bidding Scheduler - worker side

```
 1: global variables
 2:     JobStatus, global map
 3: end global variables
 4: function sendBid(job)
 5:     currentWorkloadCost = totalCostOfUnfinishedJobs();
 6:     bid = currentWorkloadCost;
 7:     bid += estimateDataTransferTime(job);
 8:     bid += estimateProcessingTime(job);
 9:     bidForJob(job, bid);
10: end function
11:
12: function consumeJob(job)
13:     JobStatus[job.jobID].status = started;
14:     jobResult = process(job);
15:     JobStatus[job.jobID].status = finished;
16:     sendJobResult(jobResult);
17: end function
```

Upon closer examination of the bidding algorithm, it becomes clear that no job needs to be rejected by all workers before being processed. This is because, unlike the original *acceptance criteria*, the "opinionated" nodes offer job completion estimates that factor in details, such as workers' network and processing speeds in this example, when allocating work. Moreover, this approach to scheduling where the job allocation is controlled by the dedicated queue threshold also allows for volatile environments, as workers' performance metrics can fluctuate over time and still be leveraged to reduce execution time and data load. Additionally, because the estimates are based on worker speed configurations, the Bidding Scheduler ensures that redundant resources (i.e. copies of the same repository cloned in different workers in this example) occur only to accelerate overall execution. This is the case if, for example, the worker that has resources locally has too many queued jobs, therefore the cost of waiting for it to become available might be greater than the cost of assigning the job to another worker that needs to download the resources for itself.

While classic auction-based scheduling algorithms often adopt a double-auction approach, where both jobs and resources submit bids or asks, this work employs a single-sided bidding mechanism. In the proposed scheduler, the master node publishes a job, and workers respond with bids based on their locally estimated cost of execution. Jobs are passive entities in this interaction and do not express preferences or priorities.

This design reflects the reactive and online nature of the scheduling context, where jobs arrive one at a time and must be assigned promptly. There is no global backlog of tasks to optimise over, and jobs cannot meaningfully act as bidders. In this setting, implementing a double-auction mechanism would introduce unnecessary coordination overhead without improving job assignment quality.

The simplified bidding approach supports decentralised, low-latency scheduling while preserving cost-awareness at the worker level. Similar trade-offs between decentralisation and complexity have been noted in prior distributed scheduling research, particularly when auction-based coordination is adapted to real-time or stream-based environments [78, 79].

6.3 Bidding Scheduler Evaluation

To address the **RG3** and assess the performance of our scheduling approach, we executed the MSR workflow from Section 4.1, employing five worker nodes and one master node. The setup involved seven instances total: one for each worker, one for the master, and one for the messaging system. We based the job configurations and worker speeds on the configurations detailed in Sections 4.3.1 and 4.3.2. The source code and instructions for running these experiments and reproducing the results are located in the GitHub repository dedicated to thesis artefacts [7].

We tested all combinations of worker and job settings. To manage network and processing speeds, like for baseline and Resource-Registy evaluation, workers were equipped with pre-set speeds that, in this approach, guided bid values.

[7]Instructions for running experiments using Crossflow with the Bidding Scheduler `https://github.com/ana-markovic/lid-msr/tree/master/bidding-modeldriven`

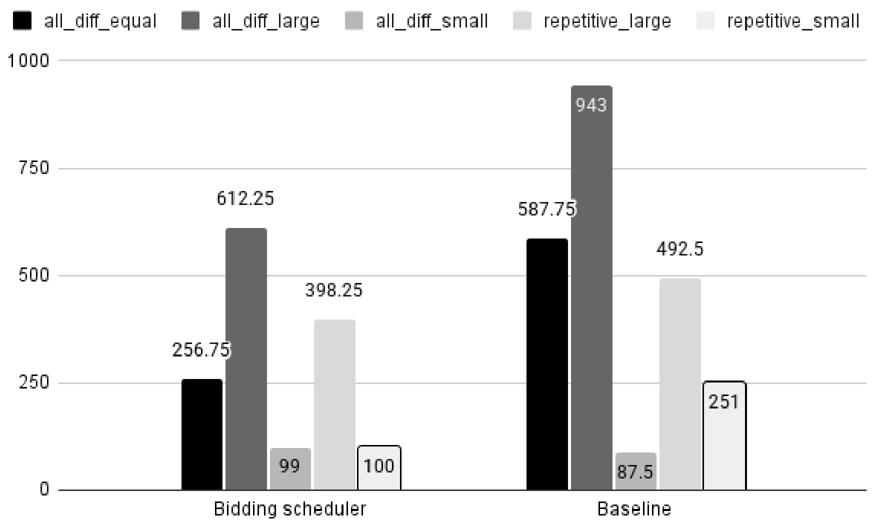
We have again applied a noise scheme to mimic real-world network fluctuations during job execution, ensuring bidding costs slightly differed from actual completion times.

Figures 18, 19 and 20 provide insights into the performance differences between the Bidding Scheduler and the Crossflow baseline. These charts illustrate four factors: average execution time per job and worker speed configuration in Figure 18, average worker usage per job and worker speed configuration in Figure 19 and average megabytes downloaded per job configuration together with the average number of cache misses per job configuration in Figure 20.

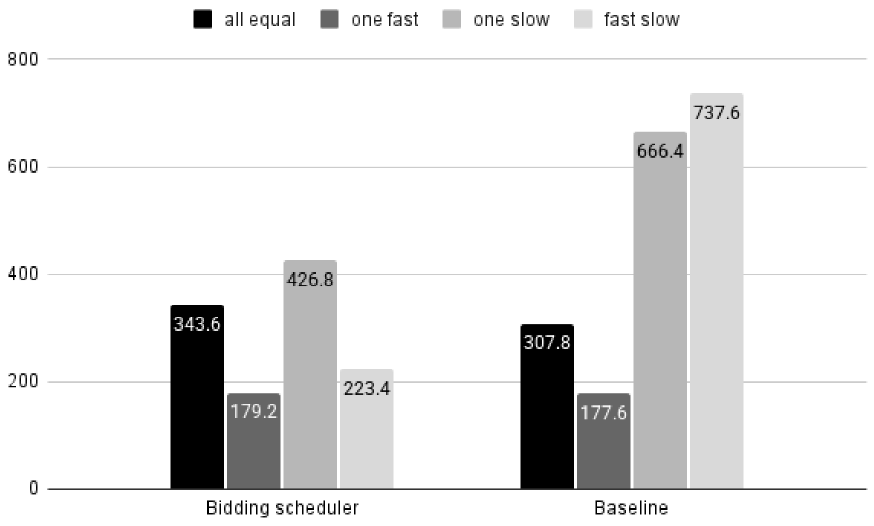
To evaluate the performance of the Bidding Scheduler, we aggregated the execution results per job configuration to observe how the bidding approach impacts different workloads and per worker speed configuration to observe performance variations in heterogeneous environments. The similar number of repository cloning operations (± 3) during execution for both schedulers, as illustrated in Figure 20a, indicates that the observed speedup results from eliminating the round of rejections that was outlined as an unnecessary overhead in the Crossflow baseline, and improved worker usage. The amount of data load, which is directly associated with cache misses, remains consistent across both schedulers for unique workloads, as depicted in Figure 20b. The 400-1000MB variances for repetitive ones correspond to the denoted difference in cache misses and present 1-3% of the entire workload's data load for respective configuration. However, Figure 18 indicates that workflows using the Bidding Scheduler generally complete faster, with speedups ranging from 1.23 to 2.51 times on average for different job configurations and from 1.56 to 3.30 times for different worker speed configuration, on average.

The Bidding Scheduler optimises worker utilisation by considering the workers' physical characteristics and not only their locally available resources. It creates a different work schedule based on various criteria, assigning jobs to workers that promise the fastest completion, considering current downloading and processing speeds and the time needed to complete previously queued jobs. This aim to optimise worker utilisation is shown in Figure 19. For instance, the Bidding Scheduler achieves 79.08% worker usage on average for environments with one worker slower than the rest, while the Baseline scheduler only reaches 60.56%, as illustrated in Figure 19b. This indicates that workers remain idle for 18.52% more time during execution with the Baseline scheduler, likely due to waiting for the slow worker to complete processing large repositories. This difference is even more prominent in environments with one slow and one fast worker, where the Baseline scheduler's average worker usage is 32.44% lower than the one obtained with the Bidding Scheduler.

Analysing the Bidding Scheduler against the Resource-Registry approach, particularly when the latter underperformed, reveals significant speedups for workflows executed with the Bidding Scheduler. Table 6 provides an overview of execution times, in seconds, for various worker speed configurations processing the *repetitive_large* job configuration. The most notable example is for the execution environment with both slow and fast workers where the Resource-Registry execution took 2218 seconds, significantly longer than the Baseline's

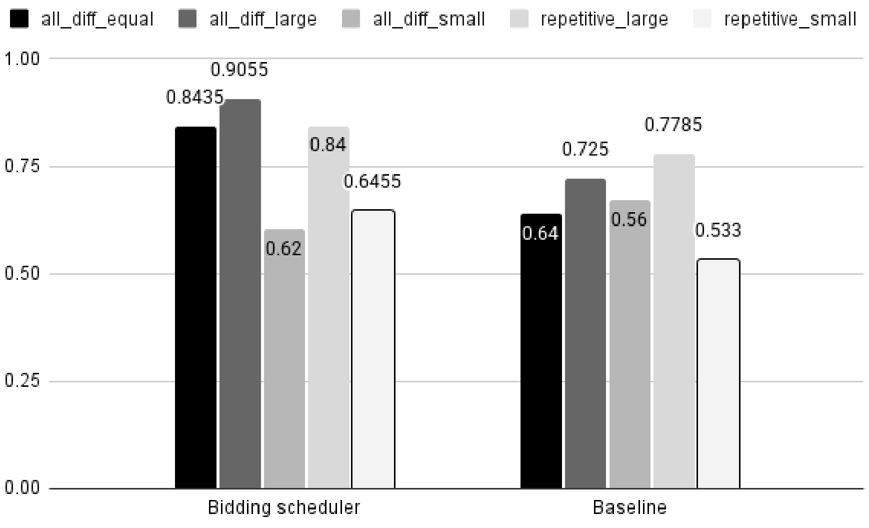


(a) Average total execution time (in seconds) per job configuration

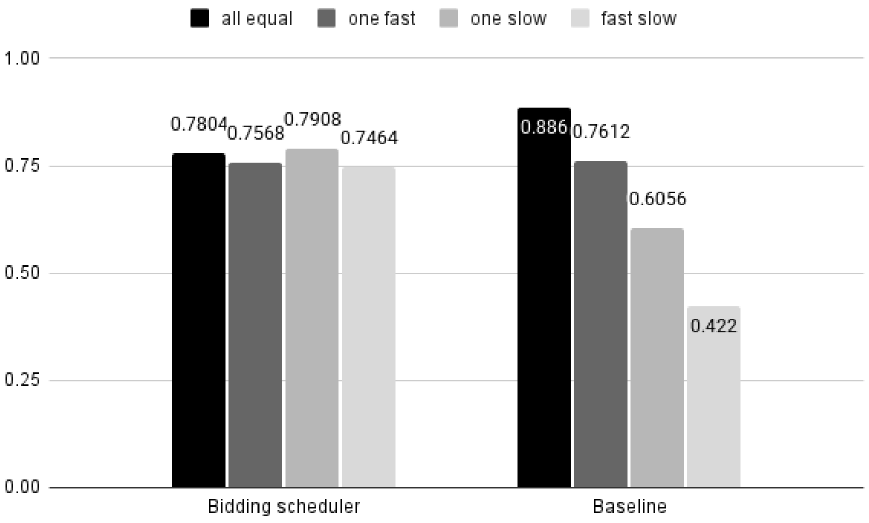


(b) Average total execution time (in seconds) per worker speed configuration

Figure 18: Accumulated results for end-to-end execution time (in seconds)

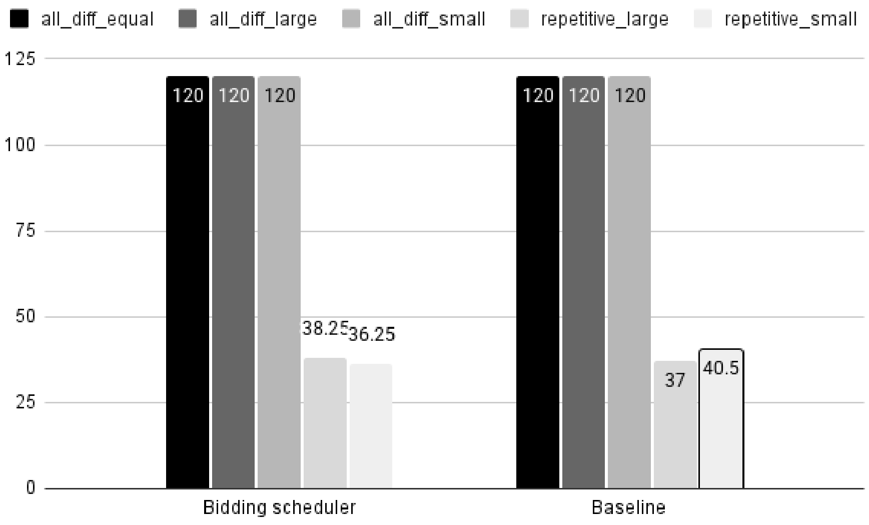


(a) Average worker usage per job configuration

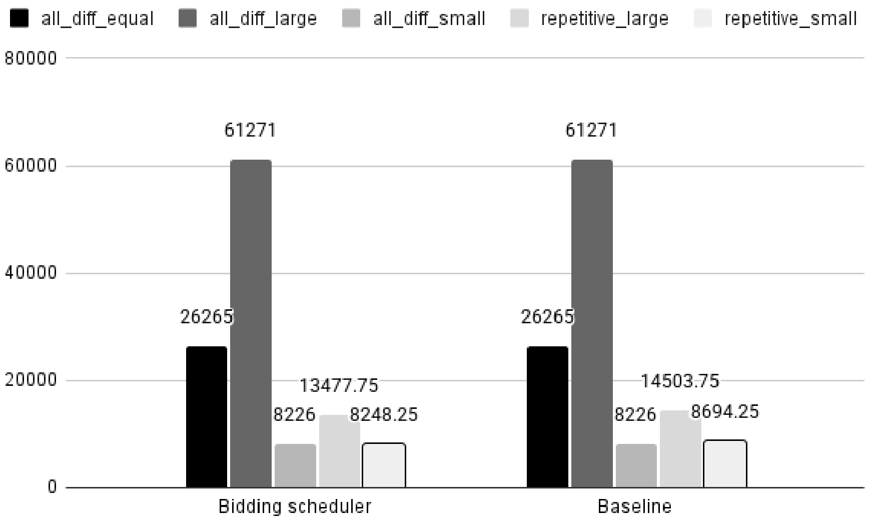


(b) Average worker usage per worker speed configuration

Figure 19: Accumulated results for worker usage (percentage, illustrated as a number between 0 and 1)



(a) Average cache miss count per job configuration



(b) Average data load per job configuration (MB)

Figure 20: Accumulated results for cache misses (number) and data load (MB) per job configuration

701 seconds, with an average worker usage of 0.45. This suggests the slow worker preferred the repetitive large job, resulting in the master routing all repetitive repository jobs to the slow worker. Conversely, the same workflow with the Bidding Scheduler completed in 342 seconds, with an average worker usage of 0.73, indicating a more balanced job distribution among worker nodes.

Jobs: repetitive_large	**Bidding**	**Resource-Registry**	**Baseline**
all equal	460	538	412
one fast	243	474	221
one slow	548	1534	636
slow fast	342	2218	701

Table 6: Execution times (in seconds) for experiments processing the *repetitive_large* job configuration with different worker speed configurations.

As stated in Section 3.2, our primary focus are LID MSR workflows running in heterogeneous execution environments, where workers do not have the same physical characteristics. Table 7 presents the execution times for workers with varying network and processing speeds. The greatest speedup of 6.48 times compared to the Resource-Registry approach occurs in repetitive workflows dealing with large repositories. Misassigning a single repository repeated in later jobs can greatly slow execution for Resource-Registry. Such misassignments reduce worker usage to 0.29, leading to higher idle times for faster workers. In repetitive workflows, though both Bidding and Baseline record similar cache misses (38-40) and data load for *repetitive_large* and *repetitive_small*, the Bidding Scheduler not only adheres to data locality principles but also distributes incoming work based on other worker characteristics, leading to 6.31 times faster execution for *repetitive_small* and 2.05 times for *repetitive_large*, with an average worker usage increase of approximately 9%.

In scenarios involving workloads composed exclusively of unique repositories, such as *all_diff_equal*, *all_diff_small* and *all_diff_large*, where the importance of data locality is diminished (under the assumption that workers do not maintain local copies of repositories acquired before execution), the Bidding Scheduler demonstrates clear advantages in situations when the job is considered intensive (i.e. repository is large and takes longer to clone and process). Specifically, in this case it achieves performance improvement of 1.99 times over the Resource-Registry scheduler and 2.98 times over the Baseline scheduler. The distinct nature of the repository jobs results in an identical number of cache misses and a consistent volume of data transmission over the network. Therefore, any observed improvements are attributed to more refined job allocation strategies and enhanced utilisation of worker resources, underscoring the Bidding Scheduler's suitability for environments with diverse worker capabilities. In situations where the repositories are of comparable size or the jobs to be processed are minor (resulting in repositories requiring minimal time for cloning and processing), the differences in execution times can be attributed to noise and minor variances in execution.

Workers: all_equal	**Bidding**	**Resource-Registry**	**Baseline**
all_diff_equal	187	604	1124
all_diff_large	464	925	1382
all_diff_small	59	65	71
repetitive_large	342	2218	701
repetitive_small	65	412	410

Table 7: Execution times (in seconds) for different job configurations running in environments with all worker nodes having equal physical characteristics

6.4 Empirical Assessment of Scheduling Efficacy

To assess the validity of our controlled tests, we performed various experiments in a real-world environment, engaging with factual repository analysis to observe the behaviour of both the Bidding and Baseline schedulers.

6.4.1 MSR Example

We examine a workflow which queries GitHub for Git repositories containing applications that rely on specific libraries, and we explore the frequency with which these libraries are utilised together. More precisely, we investigate the instances in which popular NPM libraries [4] for JavaScript co-occur in prominent, large-scale projects on GitHub (e.g., repositories exceeding 500MB in size with at least 5000 stars and forks) by examining *package.json* files in the repository and analysing their dependencies, if present. To do this, we initially documented all the target libraries found in [4]. Subsequently, we carried out the following steps to assess the co-occurrences of various libraries:

1. **Clone.** Search GitHub for favoured large-scale repositories (e.g. repositories larger than 500MB with at least 5000 stars and forks) and clone them.

2. **Checkout.** Query the repositories found in the previous step for *package.json* files and look into their dependencies.

3. **Analysis.** Calculate the number of times libraries appear together.

4. **Report.** Store the results in a CSV file.

6.4.2 Experimental Setup

For conducting a series of experiments executing the LID MSR example depicted in the previous section, we established an AWS cluster using *t3* instances, which are AWS instances designed to provide a cost-efficient and scalable compute option for applications with variable workloads that don't consistently require high CPU usage. These instances are burstable, meaning they offer baseline CPU performance but can "burst" above the baseline when needed, utilising accrued CPU credits [80]. This cluster was used to run experiments involving non-simulated workloads, where the time taken to complete incoming jobs depended solely on the characteristics of the employed *t3* instances. These experiments were reliant on the bandwidth and processing speeds available to the AWS instances, in conjunction with the responsiveness of the GitHub API. We executed the workflow with both schedulers three times each, to gather results and observe the nuances between executions. The source code for running these experiments and reproducing results can be found in the artefact repository[8] [9].

Considering all workers were configured as t3 instances, we expect them to possess approximately comparable physical characteristics, notably the network and processing speeds, although these speeds were not known to us in prior execution. Before executing the experiment, each worker's network and processing speeds were calibrated by analysing the time taken to clone and process an initial 100MB repository. These calibrated values were then used as input to the worker's initial job estimation model. These calibrated speeds were necessary as the workers needed to form their estimates for the first set of jobs. Configuring network speed based solely on AWS *t3* instance specifications could prove to be suboptimal for several reasons. Notably, AWS provides only the upper limit of network capabilities [81], without accounting for other critical factors influencing the cloning process. These factors include the size of the repository, the number of commits and branches, and the speed of the local disk [82, 83]. Consequently, conducting preliminary tests on workers using a single repository ensures that subsequent performance estimates are as accurate as possible. Upon completing each job, workers were required to compute their latest network and processing speeds. Network speed was calculated by dividing the repository size by the time taken to download, while processing speed was determined by dividing the repository size by the time taken to examine its contents. These newly obtained speeds were then utilised to set the network and processing speeds for the subsequent bid by calculating the historical average for all speeds determined from previous jobs. We also considered a windowed average (average of last N jobs), but this approach was not used in the end, due to the burstable nature of the AWS *t3* instances, which is characterised by periodic increases and decreases in network and processing speeds. A windowed average, being sensitive to recent data, would disproportionately reflect these temporary spikes or drops, leading to inaccurate and unstable estimates. For example, if an instance experiences an anomalously high burst in network speed during one recent job, a windowed average over the last N jobs would misleadingly inflate the estimated network speed for subsequent jobs.

[8]Instructions for running experiments dedicated to non-simulated MSR workflow using the Crossflow Baseline `https://github.com/ana-markovic/lid-msr/tree/master/baseline-npm-realdata`

[9]Instructions for running experiments dedicated to non-simulated MSR workflow using the Crossflow with the Bidding Scheduler `https://github.com/ana-markovic/lid-msr/tree/master/bidding-npm-realdata`

MSR	Bidding	Baseline
run 1	2670	3575
run 2	2432	3544
run 3	2493	4183

Table 8: MSR execution times (in seconds)

6.4.3 Discussion of Results

The results obtained from this set of experiments, organised according to the metrics outlined in Section 3.4, are presented in Tables 8, 9, and 10. In Table 8, it is observed that, when none of the workers have any locally downloaded repositories, the Bidding Scheduler completes execution with a 25.3%-40.4% reduction in time compared to the Baseline. For instance, during $run3$ this difference is greatest, with the Bidding Scheduler setup completing in 2493.2 seconds whereas the Baseline completes in 4183.5 seconds. Conversely, the lowest acceleration was noted for $run1$ where the Bidding Scheduler finished in 2670.5 seconds compared to the Baseline's 3575.55 seconds.

To further explore these findings, we should evaluate the data presented in Tables 9 and 10. In $run1$, the workflow using the Bidding Scheduler downloaded 332935.90 MB, whereas the same workflow with the Baseline scheduler transferred 891165.59 MB. This equates to 205 and 405 cache misses in Table 10, respectively, highlighting a significant reduction of 62.6% in data load due to a decrease of 49.4% in cache misses. In $run3$, as shown in Table 10, the Bidding Scheduler resulted in 200 fewer repository clones, contributing to a decrease in downloads by 559546.07 MB, as indicated in Table 9. Consequently, since all the workers can be considered equal due to running on the same AWS VMs, the Baseline scheduler demonstrated a 40.4% longer execution time owing to higher data transfer rates by approximately 62.9% compared to the Bidding Scheduler, indicating that the Bidding Scheduler effectively optimises the performance of LID MSR workflows by enhancing data locality. These findings align with insights obtained from the results obtained in a controlled environment, collectively demonstrating the efficacy of the Bidding Scheduler in real-world deployments and affirming the validity of our controlled experimental setup.

Figure 24, located in the appendix of this thesis, presents the aggregated results detailing the frequency with which popular npm libraries are used in conjunction. These results are derived from the comprehensive workflow depicted in Section 6.4.2. The table serves as a resource for understanding the patterns of library co-usage, providing granular data that can be utilised for nuanced analysis. However, it is not essential for further research in the domain of job scheduling for data processing platforms.

MSR	Bidding	Baseline
run 1	332935.90	891165.59
run 2	325461.08	847802.57
run 3	330048.70	889594.77

Table 9: Data load (MB)

MSR	Bidding	Baseline
run 1	205	405
run 2	191	394
run 3	186	386

Table 10: Cache miss count

6.5 Threats to Validity

The implementation of the Bidding Scheduler approach, though advantageous in several scenarios, presents specific risks and limitations that require careful consideration. This section discusses threats to the validity of the empirical assessment of the Bidding Scheduler, following the structure proposed by Wohlin et al. [64]: *conclusion validity*, *internal validity*, *construct validity*, and *external validity*.

6.5.1 Conclusion Validity

Conclusion validity refers to the degree to which conclusions drawn from the data are justified.

- **Limited number of configurations.** The experimental results were derived from a fixed set of repository workloads and worker configurations. Although these combinations were systematically constructed to vary job size and worker speed, the conclusions are limited by the absence of statistical tests or repeated trials.
- **Simplified timing model.** Job execution times were estimated from worker parameters and controlled download/processing speeds. This abstracted model facilitates controlled comparison but may lack the fidelity required to capture certain real-time scheduling dynamics.

6.5.2 Internal Validity

Internal validity concerns whether observed performance differences can be causally attributed to the use of the Bidding Scheduler, rather than to uncontrolled or confounding factors.

- **Queueing threshold interaction.** The use of a configurable queue threshold to limit how many jobs a worker can queue before triggering new bids introduces temporal coupling between worker speed and scheduling responsiveness. This may create bias in favour of faster workers who deplete their queues more quickly.

6.5.3 Construct Validity

Construct validity refers to how well the experimental design measures the concepts it intends to evaluate, in this case, performance, fairness, and locality-awareness.

- **Locality measurement.** The evaluation uses cache miss count and data transferred (MB) as proxies for data locality. While these are meaningful indicators, they may not fully capture finer-grained aspects of data affinity, such as checking out a specific branch or commit on a repository that is already available locally.
- **Workflow realism.** The jobs represent codebases from different repositories with varying levels of overlap. While this captures many aspects of locality-intensive workloads, the master's lack of global knowledge and the job-level granularity may not reflect more complex interdependent task structures found in other domains.

6.5.4 External Validity

External validity addresses the extent to which results generalise beyond the specific testbed.

- **Generalisation to other workflows.** The evaluation focuses on MSR workloads that involve repeated access to potentially remote repositories. In scenarios with very short-lived or homogeneous jobs, the overhead of bidding may outweigh its benefits, limiting the scheduler's applicability.
- **Real-world variability.** Although network variability was emulated via noise injection, the testbed remains a simulation of distributed behaviour. Factors such as hardware heterogeneity, concurrent user activity, or VM-level contention were not included and may influence scheduler behaviour in real deployments.

7 Enhancing the Bidding Scheduler with Self-correcting Workers

The Bidding Scheduler is argued to be advantageous for job scheduling of large and computationally intensive jobs in heterogeneous environments, particularly within the domain of LID MSR workflows. However, this approach is not without its inherent challenges. In the previous chapter we assumed that developers could provide an accurate formula for estimating the job completion time, as outlined in Section 6.2. However, this is not always feasible. In many workflows, the speed at which jobs are processed is influenced by various job attributes, and developers might not always have the precise formula needed. Additionally, if the workflow runs on the cloud, the performance of virtual machines might vary, as noted by Schroeder and Harchol-Balter [84]. The main problem here is the gap between the actual time a job takes to complete and the time estimated by a worker node, a problem stemming predominantly from estimation errors as opposed to normal system throttling. When the formulas for estimating job completion time are not accurate, it affects the scheduling process and can lead to inefficient use of resources and delays in completing jobs. These issues can significantly impact overall system performance, making it less efficient in using computational resources.

To illustrate, consider a scenario where a workflow needs to be processed within a specific time frame using heterogeneous workers. If the estimated time (t_e) for repository (r) given by a slow worker (w_s) is lower than all other estimations provided by other workers, due to the inaccurate estimation formula together with the slow worker having least jobs pending processing, the scheduling system might assign the (r) to (w_s), regardless of the fact that (t_e) is significantly less than the actual execution time (t_a), causing delays. Conversely, overestimations might lead to reduced worker usage percentage for performant workers, which is both costly and inefficient. Such discrepancies are particularly problematic in environments requiring precise job scheduling and resource optimisation to maintain high throughput and low latency.

A significant challenge lies in the reliance on the developer's domain knowledge to provide accurate formulas for bidding within the context of the implemented workflow. The presumption here is that the developer must have comprehensive knowledge of the domain to devise the correct formula. However, this formula is not always evident or formally documented. Take, for example, the process of verifying code style to ensure Java code complies with specified coding standards. It is reasonable to assume that the time required for such a task is closely linked to the volume of code, measured in lines. Nevertheless, this relationship is typically absent from the documentation accompanying the libraries performing the style checks. Consequently, the developer might resort to an empirical approach, running the checkstyle on a limited number of repositories, collecting data on line counts and execution times to establish a correlation.

However, this method is fraught with potential inaccuracies. The sample size of repositories may not be sufficiently comprehensive to derive an accurate formula. For instance, a limited dataset might not capture the variability in code complexity, coding patterns, or specific library behaviour. Furthermore, empirical methods may not account for outliers or edge cases, leading to skewed estimations. Additionally, workers with different physical characteristics, such as varying processing power and memory capabilities, may necessitate distinct formulas, as they operate at different speeds. This variability introduces further complexity into the estimation process. For example, a high-performance worker may process code style verification significantly faster than a less powerful one. Therefore, a one-size-fits-all formula may not suffice, as it does not account for the varying execution environments, even though it is often mandated in master/worker architectures. In such architectures, the deployment involves a single program executing on the master node and another on each of the worker nodes. Only these two programs are dispatched to the execution environment, with the worker program being replicated as many times as necessary to achieve parallelism in execution. This approach does not allow for the manual adjustment of processing speeds to account for individual worker characteristics unless the developer possesses detailed knowledge of how these physical attributes translate into processing performance. Therefore, the lack of specific control over worker speeds can result in suboptimal performance and inefficiencies in execution.

In environments where job scheduling is critical, such as real-time data processing or large-scale analysis tasks, the inaccurate estimation of job execution times can lead to cascading inefficiencies [59]. These inefficiencies can manifest as increased latency, resource misuse, and ultimately, reduced overall system performance with increased monetary costs for cloud deployments. Moreover, the scalability of the Bidding Scheduler is compromised, as it becomes challenging to maintain consistent performance across diverse workloads and systems.

7.1 Background: Scheduling Jobs with Estimation Errors

While auction-based schedulers are suitable for distributed environments, it must be noted that almost all of those reviewed in Section 6.1.2, with the exception of [74], fail to account for differences between the estimated execution times and the actual times observed. In other words, they do not consider the variation between the performance that workers promise in their bids and what they actually deliver, which may not always be the same.

In scenarios where job processing times are known beforehand, the Shortest Remaining Processing Time (SRPT) policy is proven to be optimal. This was demonstrated by Schrage [85], who showed that SRPT minimises the number of jobs in the system at any given time. The SRPT policy prioritises jobs with the smallest remaining processing time. It is a pre-emptive strategy, meaning that a new job with a shorter processing time can interrupt the currently running job.

However, assuming that job processing times are known exactly is often impractical [86]. For distributed data processing systems, while job sizes might be known, the exact processing times may not be. Schroeder and Harchol-Balter [84] point out that this is partly due to the latency inherent in network data transmission and the acknowledgment process. This necessitates the use of estimated processing times. Additionally, the performance of virtual machines, commonly used in web servers, can vary, further complicating accurate time estimations. Estimation errors can arise under these conditions, creating potential issues for scheduling policies that depend on known processing times, potentially leading to performance problems [86]. This becomes especially critical in distributed systems where users pay for processing time, where demand for computation resources exceeds availability, or where jobs must be processed as quickly as possible [87].

Research on scheduling approaches for distributed systems that consider errors in job processing time estimations is limited [88]. Lu et al. [58] designed a simulator to test various scheduling strategies, including SRPT, and determine the error threshold that significantly impacts performance. Dell'Amico et al. [59] demonstrated how underestimating job processing times can severely degrade performance, especially in distributions with a few large jobs among many small ones. If estimation errors scale with job size, underestimating large jobs can result in long delays for smaller ones.

Mailach et al. [86] introduced a scheduling method for distributed environments that classifies jobs based on estimated processing times. The estimated processing times are given the lower bound of 1 unit of time for short-running jobs, and the upper bound of 10^6 units of time for extremely large jobs. This method aims to overcome the limitations of SRPT, such as underestimated large jobs blocking smaller jobs. When a job arrives, the scheduler compares its estimated processing time with that of recently processed jobs and assigns it to a class. Shortest jobs are prioritised to be processed first. Extremely large jobs are given the lowest priority to avoid delaying smaller jobs.

In [87], a scheduler for distributed and heterogeneous environments was proposed, which estimates the time required by each worker to complete jobs based on historical performance data (using k-Nearest Neighbors). It also calculates the accuracy of these predictions by estimating the likely error for a job on a specific worker. Two strategies were tested: one prioritising execution of jobs with minimal estimation error and the other prioritising jobs with maximal estimation error. The former assumes the best case scenario with the minimum estimation error being zero, thereby ignoring the predicted estimation error in the estimated job execution time. To the contrary, the latter applies the maximum amount of predicted estimation error to the estimated job execution time. The results showed that considering estimation errors when allocating work led to more efficient scheduling compared to ignoring these errors.

The work in [58, 59] outlines the importance of having the estimated processing times of jobs in distributed systems with minimal estimation errors to prevent performance degradation. It is therefore crucial to predict the resource/performance requirements of jobs as accurately as possible before they are assigned, so that suitable resources can be allocated for their execution [89]. Predicting these requirements is a complex task due to the continually changing and diverse nature of modern distributed systems [90].

Regarding the prediction of job completion times, a user-guided resource prediction policy is discussed in various studies [74, 91, 60]. This technique is generally unreliable and can often lead to significant overestimation or underestimation of resources [61]. An alternative approach involves a feedback-guided job modeling scheme, as outlined in [89]. In this method, when a new job is accepted, its job model is created and compared with previously executed jobs, before being classified based on the detected similarity [92]. The new job's resource requirements are predicted based on this similarity, and the job is then scheduled for execution. After the job is executed, its runtime data is stored in an *Execution History* database, which is used to detect similarities among jobs through statistical methods. As the *Execution History* accumulates data, the scheduler becomes more accurate in predicting resource requirements for new jobs, leading to more optimal resource allocation without human intervention.

Similarly, Sen et al. propose a job scheduling approach for distributed systems based on clone detection [61]. Their objective is to use historical execution data to predict resource requirements and completion times for new jobs identified as "clones" of previously executed jobs. Like the feedback-guided job modelling schemes in [89] and [92], this system identifies "exact clones" from historical data. An "exact clone" refers to a new job that is identical to a previously executed job in terms of resource requirements and CPU cycles. The algorithm calculates the expected completion times for each job-resource pairing and prioritises those that minimise execution time while maintaining cost efficiency. When jobs are executed on "favoured resources" — those that closely match the historical execution environments of their clones — the scheduler is effective because predicted and actual execution times align closely. However, if jobs are assigned to non-favoured resources, the actual completion time may differ significantly due to varying resource capabilities.

Ali et al. [93] introduces a *Prediction Engine* to estimate the runtime and resource requirements (CPU, memory, and bandwidth) for jobs submitted to a grid computing system. The engine aims to improve resource allocation efficiency by enabling grid schedulers to select optimal execution sites. The prediction engine uses a history-based approach to estimate job runtimes. It operates on the principle that jobs with similar characteristics are likely to have similar resource requirements. Historical data from previously executed jobs is stored and analysed to identify patterns and similarities, through a notion of a similarity template, which includes attributes that strongly influence job runtimes. Jobs get classified based on similarity templates into an equivalence class, the predicted runtime is calculated using statistical measures (e.g. mean or linear regression) of runtimes for jobs in the same class. While maintaining historic records similar to [89, 61, 92], this approach offers both centralised and decentralised history based on whether there is a single database that maintains the entire history for all execution sites or each execution site maintains its own job history and estimates runtimes locally.

Reig et al. [62] propose a scheduler that focuses on a prediction system to improve the allocation of cloud resources, ensuring jobs meet their deadlines while adhering to Service Level Agreements (SLAs). Upon receiving a job, the scheduler queries the *Prediction System* and it decides, depending on the on the policy being used and resources' status, how to allocate resources to the incoming jobs. The *Prediction System* is in charge of predicting the minimum resource requirements needed to meet SLAs. The *Prediction System* leverages machine learning models for predicting job execution times using historical data. Completed job data — covering job specifications, resource allocations, and execution times — is stored in the *Prediction System* to train prediction models continually.

Bohlouli and Analoui [94] describe a centralised prediction model that utilises execution history to estimate resource requirements for grid computing jobs. This execution history database is employed to identify similar jobs and predict resource demands via statistical methods such as linear regression and similarity-based calculations. Historical job attributes are normalised for consistency, and a similarity algorithm determines the distance between new job attributes and those stored in the history. Based on the closest matches, resource usage estimates are derived using weighted averages or regression models. The execution history database is updated regularly to reflect new data, ensuring that the prediction model evolves and improves its accuracy over time. Duplicate or near-identical entries are purged to maintain database efficiency and relevance.

7.2 The Feedback-Based Estimator Approach

The Feedback-Based Estimator approach is an algorithm designed to optimise job scheduling and mitigate estimation errors within distributed systems dependent on time predictions. This approach utilises a self-correcting worker mechanism, which dynamically refines prediction formulas based on historical data. The sequence diagram depicted in Figure 21 illustrates the Feedback-Based Estimator approach by presenting an extension to the Bidding Scheduler. This extension introduces a series of operations executed by worker nodes to collect metrics related to actual executions and iteratively enhance their estimation formulas.

Building upon the Bidding Scheduler explained in Section 6.2, the Feedback-Based Estimator approach modifies the terminal phase of the bidding process:

1. **Worker nodes job processing**
   Upon the arrival of jobs in the worker node's dedicated queue, the worker node proceeds to sequentially process each job. The worker node dequeues an assigned job, processes it and submits the processing results back to the master node.

2. **Collecting execution metrics and storing historic data**
   In addition to processing the job and reporting the results back to the master node, the worker node is responsible for collecting metrics about the job's execution. Such metrics include, but are not limited to, actual processing time and relevant data from the job model, such as repository

size, file or line count. These data points are gathered and stored as historic data within the worker node's working memory or local storage. The historic data serves as a crucial resource for future estimations, enabling the worker node to maintain a comprehensive log of its performance over time.

3. **Analysing historic data and updating the estimation formula** The final aspect of the worker node's responsibilities involves the continuous improvement of its job execution time estimation formula. By analysing the historic execution data, the worker node can identify discrepancies between the estimated and actual processing times. Various analytical methods can be employed to update the estimation formula. For instance, the worker node might use a historic average of past processing times to derive a more accurate mean estimation. Alternatively, more sophisticated methods such as linear regression can be applied to identify linear relationships between job attributes and processing times, thereby developing a predictive model. In other instances, machine learning techniques such as random forest might be utilised to capture non-linear relationships within the dataset. The worker node iteratively refines its estimation formula based on the chosen method, thereby minimising prediction errors and enhancing the precision of future job time estimations.

Process description

As stated in the previous section, developers might face situations where they are uncertain about the exact bidding formula, such as during static code analysis, where the estimate might be linked to line count but lacks a clear-cut calculation method. Still, accurately estimating the time required to complete jobs is crucial for successful completion of the workflow with effective resource and cost management as described in Section 7.1, which also outlines some approaches to improve these estimations. To improve the accuracy of time estimations within the Bidding Scheduler, we propose a self-correcting worker approach that relies on historical data to refine prediction formulas dynamically and optimise job scheduling through reducing estimation errors.

When job attributes show a correlation with processing time that is not straightforward or linear, simple linear estimation methods fall short. For example, job attributes might affect processing times in complex ways due to various factors in the execution environment. To address this, we introduce a Feedback-Based Estimator (*fbe*) that can adjust estimation models based on past performance. Similar to the approach used in Section 6.4.2, where workers calculated the average download speed from past tasks to estimate the time needed for future repository cloning, workers in this approach can use historical data to adjust their future bids.

The Feedback-Based Estimator can be a simple average of speeds from previous runs or a more complex model such as linear regression or machine learning

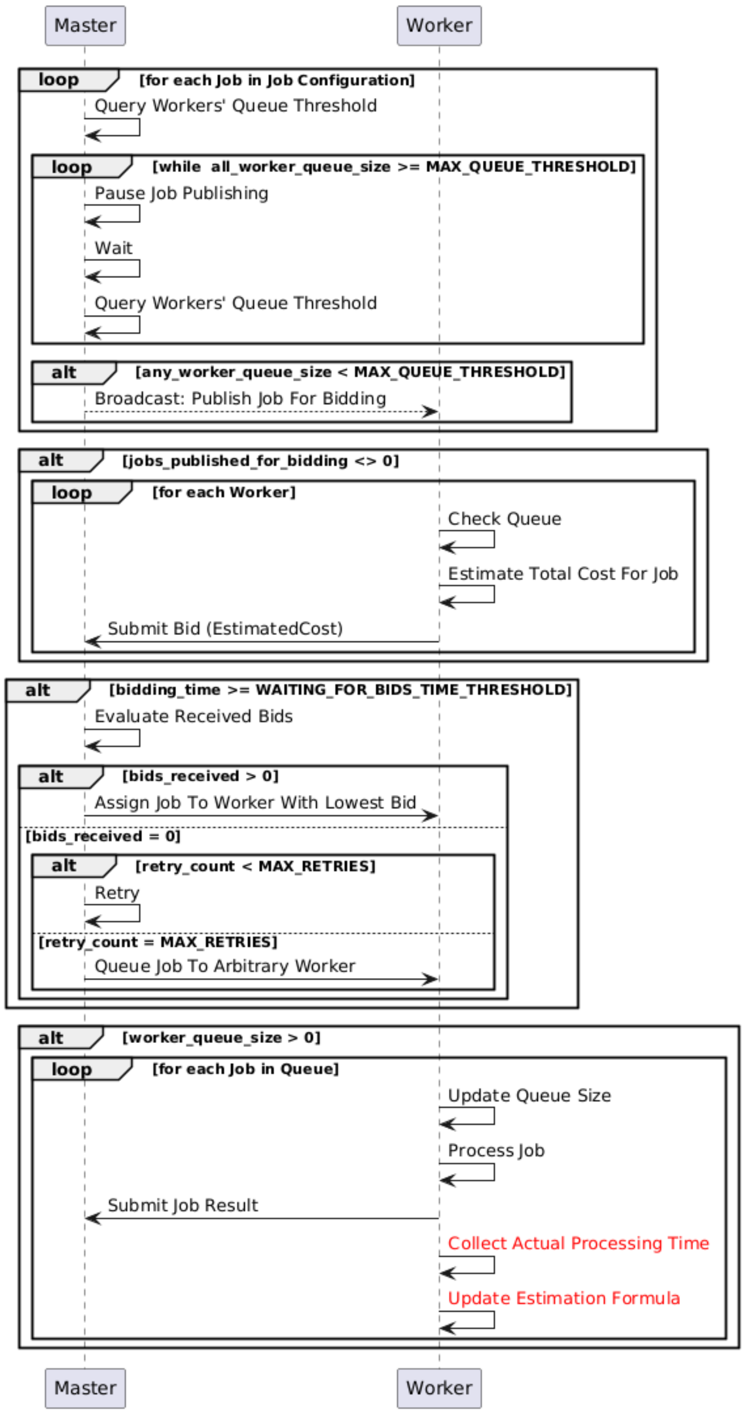

Master
Worker
loop [for each Job in Job Configuration]
Query Workers' Queue Threshold
loop [while all_worker_queue_size >= MAX_QUEUE_THRESHOLD]
Pause Job Publishing
Wait
Query Workers' Queue Threshold
alt [any_worker_queue_size < MAX_QUEUE_THRESHOLD]
Broadcast: Publish Job For Bidding
alt [jobs_published_for_bidding <> 0]
loop [for each Worker]
Check Queue
Estimate Total Cost For Job
Submit Bid (EstimatedCost)
alt [bidding_time >= WAITING_FOR_BIDS_TIME_THRESHOLD]
Evaluate Received Bids
alt [bids_received > 0]
Assign Job To Worker With Lowest Bid
[bids_received = 0]
alt [retry_count < MAX_RETRIES]
Retry
[retry_count = MAX_RETRIES]
Queue Job To Arbitrary Worker
alt [worker_queue_size > 0]
loop [for each Job in Queue]
Update Queue Size
Process Job
Submit Job Result
Collect Actual Processing Time
Update Estimation Formula
Master
Worker

Figure 21: Sequence diagram showing job scheduling with the Bidding Scheduler and self-correcting workers

(ML). ML models, for instance, can predict network speed spikes that are influenced by external environmental factors rather than specific job attributes.

The implementation of the Feedback-Based Estimator is designed to be flexible, allowing developers to choose whether and which type of Feedback-Based Estimator to apply based on the specific requirements of their workflows. This flexibility enables the integration of the model into various project scenarios without being restricted to a single predictive method. As a result, developers can benefit from a more accurate prediction mechanism that can be tailored to their unique needs.

A critical aspect of this approach is using the dedicated queue threshold (n) for each worker node within the Bidding Scheduler, described in Section 4.2. This threshold prevents workers from bidding on additional jobs as soon as all workers reach (n) queued jobs. This ensures that workers focus on completing current jobs and gathering valuable data to improve future time estimates. Without this mechanism, workers would continually bid for new jobs, leading to long queue lists and inaccurate estimates due to a lack of learning from current and queued jobs.

The description below breaks down the most crucial steps in the bidding process with self-correcting workers. A detailed example of this approach is as follows:

Initial bidding and job assignment

- **Step 1.1: Initial bidding**. Workers' queues are empty. The master publishes one job for bidding. Each worker in the system submits a bid for the published job. The bidding process encompasses an estimation formula provided by the developer. This formula can be based on job attributes such as the line count of code, the size of input data, etc., to predict the processing time for each job.
- **Example scenario:**
  - Worker A, B, and C all bid for Job 1, estimating it will take 3 hours to process 900000 lines of source code.
  - Worker A wins the job.

  This step loops until all of the workers have enough queued jobs (i.e., until all of their queues reach the dedicated queue threshold (n)).

Pausing bidding

- **Step 2a.1: Reach dedicated queue threshold.** The system imposes a pause on further bidding once all workers reach a predetermined maximum threshold of queued tasks.
- **Example scenario:**
  - Assuming the dedicated queue threshold is set to 3 jobs, Worker A, B, and C now each have 3 or more jobs queued.

Execution and monitoring

- **Step 2b.1: Job execution.** Workers execute the jobs assigned to them. During execution, each worker monitors the actual processing time and records additional job attributes.
- **Example scenario:**
  - Worker A completes Job 1 in 3.5 hours.
  - Worker B completes Job 2 in 3.8 hours.
  - Worker C completes Job 3 in 5.2 hours.
- **Step 2b.2: Data collection.** The actual processing times (3.5, 3.8, and 5.2 hours) are collected and compared against the initial estimates (e.g. 3, 4, and 5 hours) in each of the workers, respectively. Discrepancies between estimated and actual times are recorded.

Updating the estimation formula with the Feedback-Based Estimator

- **Step 3.1: Calculation.** Each of the workers calculates its own new estimation formula based on the Feedback-Based Estimator. This could be a simple average or involve complex statistical methods such as linear/polynomial regression or machine learning models.
- **Example scenario:** Suppose the Feedback-Based Estimator is incorporated through the application of linear regression, utilised to forecast subsequent bids based on the historic data. Each worker comes up with its own estimation formula. For instance:
  - Worker A: $a \times \text{lineCount} + a_1$
  - Worker B: $b \times \text{lineCount} + b_1$
  - Worker C: $c \times \text{lineCount} + c_1$

Resuming bidding with updated estimation The bidding is resumed once one of the workers' queues reaches is below the dedicated queue threshold.

- **Step 4.1: Updated bidding.** Workers now incorporate their new estimation formulas, dynamically acquired through their Feedback-Based Estimators, into their future bidding estimates to minimise prediction errors.
- **Example scenario:**
  - Worker A bids for Job 4 using its own estimation formula, estimating that for Job 4 it would need x number of hours.
  - Worker B bids for Job 4 using its own estimation formula, estimating that for Job 4 it would need y number of hours.

  - Worker C bids for Job 4 using its own estimation formula, estimating that for Job 4 it would need z number of hours.
  - Assuming that $x > y > z$, Worker C wins the bid. This bidding continues until all of the workers queues reach the dedicated queue threshold (n).

Iterative improvement

- **Step 5.1: Continuous learning.** The bidding-pausing-calculating cycle is repeated iteratively. Each cycle refines the accuracy of the time estimation process. With each iteration, discrepancies between estimated and actual processing times should decrease as the system learns and incorporates new data.
- **Example scenario:**
  - After several iterations, the Feedback-Based Estimator approaches a point where time estimates become sufficiently accurate, thereby improving the overall workflow efficiency.

By following these steps, the self-correcting worker approach leverages historical data and Feedback-Based Estimators to refine job time estimations continuously. This approach aims to ensure that time predictions become progressively accurate in order to optimise resource usage and overall system performance. The fact that some aspects of self-corrective behaviour can be tailored by the developer, makes this approach integrable into various workflow scenarios, thereby ensuring scalability and adaptability across diverse technological environments.

An important consideration when designing prediction models is the trade-off between model complexity and generalisability. More expressive models can potentially fit past observations more closely, but they also risk overfitting — capturing noise rather than true patterns — and may fail to generalise to future inputs. Simpler models, on the other hand, may underfit the data, missing important structure [95].

This phenomenon is well understood in the machine learning literature and is often described through the *bias–variance trade-off*. The total prediction error of a model, commonly measured using *mean squared error* (MSE), can be decomposed as:

$$\text{MSE} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Bias refers to error introduced by overly simplistic assumptions (e.g., assuming a linear relationship when a more complex pattern exists), while variance reflects sensitivity to fluctuations in training data. High-bias models tend to be stable but inaccurate, while high-variance models may fit past data well but perform poorly on unseen data [96].

In this chapter, each worker employs a Feedback-Based Estimator that updates a linear model of job execution time (based on repository line count) using only its own history of completed jobs. This estimation model is continually refined via linear regression as more data becomes available. The approach is fully decentralised: workers maintain separate historical views of job execution and evolve their prediction models independently, without any shared memory or coordination. This supports scalability and robustness in asynchronous distributed environments, where hardware variability, network bandwidth, and workload distribution can differ significantly across nodes.

The significance of this design lies in its combination of simplicity, adaptability, and autonomy. By enabling workers to learn from past job executions and customise their cost estimators based on local conditions, the system promotes more truthful and precise bidding over time. This, in turn, leads to more efficient scheduling decisions, improved load balancing, and faster job completions. The online nature of updating the Feedback-Based Estimator's model ensures that the model evolves with changing workloads, while remaining computationally inexpensive.

While linear regression provides a strong baseline, the approach could be extended with *adaptive model selection*. For instance, each worker could maintain a sliding window of recent jobs, fit several candidate models (e.g., linear, quadratic, and cubic) periodically, and select the best-fitting one based on MSE or adjusted R^2. To prevent overfitting, selection could incorporate complexity penalties or require a minimum improvement threshold before accepting a more complex model. Such enhancements would allow estimators to remain both accurate and generalisable as job patterns evolve.

7.3 The Feedback-Based Estimator Evaluation

This section aims to evaluate the impact of the Feedback-Based Estimator on estimation errors and job distribution within a single workflow when the precise estimation formula is either unknown or assumed. The setup for self-correcting workers makes certain assumptions to specifically examine the effectiveness of the Feedback-Based Estimator approach, deliberately excluding considerations of worker heterogeneity in terms of internet speeds that affect the data transfer times. Forecasting internet speeds required for repository cloning can experience unpredictable fluctuations, such as spikes and drops, necessitating complex machine learning models for accurate predictions and estimation error minimisation. Therefore, this setup focuses on estimating and correcting the predictions of processing time, which involves simpler predictive logic.

The primary assumption is that all workers maintain uniform internet speeds. This assumption ensures that data locality remains a critical factor, but it negates the adverse effects of any worker's ability or inability to accurately estimate data transfer times, as all workers would require approximately the same amount of time to clone an identical repository. With workers sharing a common initial formula for estimating data transfer times, any estimation error, denoted as t_e seconds, would manifest uniformly across workers. In this scenario, for the end-to-end execution time it is still crucial whether a worker has already cloned the repository, influencing whether the transfer time for that repository is zero or a positive value.

Consequently, assuming consistent computation performance across workers, we assessed the performance of our approach using the simpler-to-predict task of analysing source code to detect code quality issues, such as indentation consistency, adherence to naming conventions for variables, classes, and methods, and the identification of empty methods or catch blocks.

7.3.1 MSR Example

Let us describe a workflow designed to conduct historical code analysis of commits across large-scale public Java repositories, with the objective of identifying trends in code warnings, thereby providing insights into the evolution of code quality in the repositories under scrutiny. Initially, this workflow queries GitHub for repositories containing Java code and investigates their commit histories to generate jobs for processing. Each job is conceptualised as a pair $\{repository_n, commit_i\}$, which will be handled by a worker through the following steps:

1. **Clone.** Checkout the repository, unless there is a local copy of it already stored in the local filesystem.
2. **Checkout.** Check out the specific commit to be analysed.
3. **Analysis.** Execute the code analysis on the source code to identify trends in warnings.
4. **Report.** Record the identified trends into a CSV file.

7.3.2 Experimental Setup

The initial experiments were controlled, which is common approach in research focusing on estimating processing time and reducing estimation errors, where works in [86, 88, 58, 59, 61, 62] all rely on simulated environments to evaluate the effects of estimation errors. The experimental setup controlled several components of the system model depicted in Section 4.2. Controlling workers' processing speeds, the setup included total of five worker nodes with varying speeds: one fast, two slow, and two average workers, all managed by a master node. The source code for running this experiment is presented as a research artefact[10].

To test the self-correcting workers, we prepared a job configuration consisting of 120 repositories. These repositories were comparable to those described in Section 4.3.2 and were randomly selected from GitHub's API response for public Java repositories larger than 500MB. The assumption is that larger repositories have a more extensive codebase that take longer to analyse compared to smaller repositories, thereby benefiting more from reduced estimation errors.

[10]Instructions for running the MSR workflow with and without self-correcting workers: `https://github.com/ana-markovic/lid-msr/tree/master/bidding-cf`

The repositories from the job configuration were downloaded by worker nodes, and their commit histories were analysed to create realistic job models for predicting completion time. This lead to creation of additional jobs to be processed during workflow execution, in form of $\{repository_n,\ commit_i\}$. Since our main focus was on observing and correcting time estimations for processing the repositories, we controlled the processing speeds of worker nodes and their respective processing times.

Instead of actual source code processing, we simulated the execution of this task by introducing controlled delays (sleep intervals) that represented the actual processing time. This simulated processing time was calculated as a function of predefined worker processing speed and lines of code, with added variations to closely mimic real-world conditions. To introduce variances in the simulated processing time, we used a noise level parameter. For instance, by setting a noise level to 10%, the simulated processing time was constrained within a specified range: $t_p = (0.95 \pm 0.05) \times f_n(processingSpeed, lineCount)$. Worker speeds were set and controlled as described in Section 4.3.1.

Workers used an initial estimation formula to predict the time required to complete each job based on the number of lines in the source code. This formula was intentionally different from the predefined processing speeds of worker nodes to reflect a scenario where the developer formulated the estimation from a limited dataset. Predefined processing speeds were assigned to each worker type — "fast," "slow," and "average" — similar to the setup used for network and processing speeds in the Resource-Registry and Bidding Scheduler systems, as discussed in Sections 5.2 and 6.4.2.

Due to the inherent unpredictability in processing times and the expected deviations from estimated values, we incorporated a self-correcting mechanism to improve the precision of performance predictions. To achieve this, we employed a self-correction approach grounded in linear regression. Although considering the use of an average of past execution times or a windowed average, we recognised that predicting future task execution times using these averages is prone to errors when faced with a diverse set of jobs and workers [87]. In this context, not only were the workers varied, but the repositories' source code also differed significantly, ranging from tens of thousands to millions of lines of code. While size generally indicates the extent of the codebase, repositories may contain binary files and extensive code metadata, such as commit histories and branch details. Moreover, although neural networks and support vector machines can model job execution times, they typically require extensive prior data and training [97], rendering them unsuitable for LID MSR workflows that necessitate ad-hoc model re-creation and training during execution with limited datasets. The chosen linear regression method differs, as it allows for the creation of a new estimation formula ad-hoc, immediately after a worker completes processing a repository and execution details become available. Consequently, we utilised linear regression to iteratively refine the initial estimation formula by continuously adjusting it based on observed performance metrics. This approach seeks to align predicted processing times with the specific speeds predetermined for each worker type. The linear regression model functions by analysing the discrepancies between estimated and actual processing times and subsequently adjusting the estimation formula to mitigate these discrepancies over time.

7.3.3 Discussion of Results

The evaluation was conducted by executing multiple runs of the code analysis example described in Section 7.3.1 while controlling the system model as explained in the previous section. The experiments were executed both with and without the self-correction mechanism in place. This dual approach enabled a comparative analysis of worker performance under the two different configurations.

In configurations lacking the self-correcting mechanism, the duration required for each worker to analyse code — calculated based on the total lines of code within the repository and the worker's processing speed, often exhibited significant discrepancies from the estimates provided for that repository. This misalignment resulted in overall inefficiency in job completion. In contrast, the application of the self-correction mechanism demonstrated marked improvements in alignment between estimated and actual processing times, evidencing the efficacy of linear regression in bringing performance closer to the expected benchmarks. As anticipated, an increase in the percentage of noise correlates with more pronounced variances in processing times for identical repositories. Consequently, this diminishes the advantages derived from the self-correcting mechanisms.

The workers were instructed to analyse four commits of each repository present in the job configuration. The noise levels were ranging from 5 to 20%. To evaluate the results, we turn to the data illustrated in Table 11, which compares metrics obtained from executions with and without the self-correction mechanism, under conditions of a 10% noise applied to the actual processing time. Initially, our observations reveal an approximate 13% increase in speed when the self-correction mechanism is employed. The workflow incorporating self-correction transferred an additional 2455 MB of data (2% of the size of the dataset), a consequence of encountering seven more cache misses. Since both sets of executions operated under the same scheduler, we can conclude that the variations in execution times are attributable to differences in job schedules produced with and without the self-correcting workers and their Feedback-Based Estimators.

	Total execution time (s)	**Data load (MB)**	**Cache misses**
no correction	45 758	149 535	193
with self-correction	40 176	151 990	200

Table 11: Comparison of metrics for executions with and without self-correcting

To examine this hypothesis, assuming higher line counts correspond to more complex and time-consuming jobs, Table 12 details the number of lines processed by each worker, revealing that the fast worker analysed 28% more lines

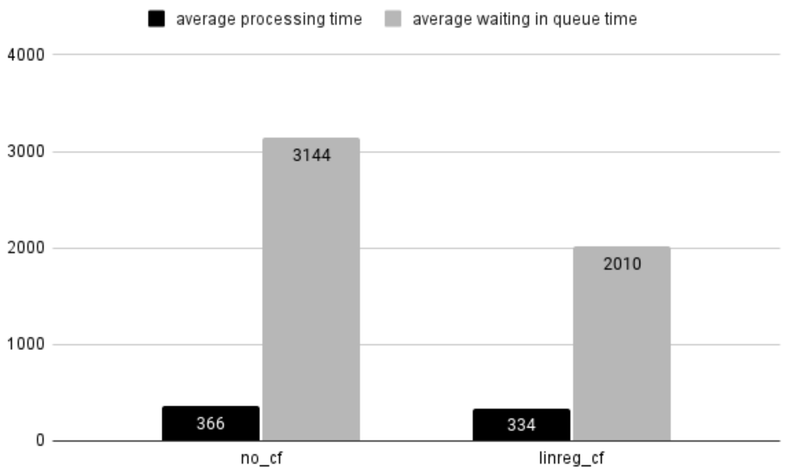
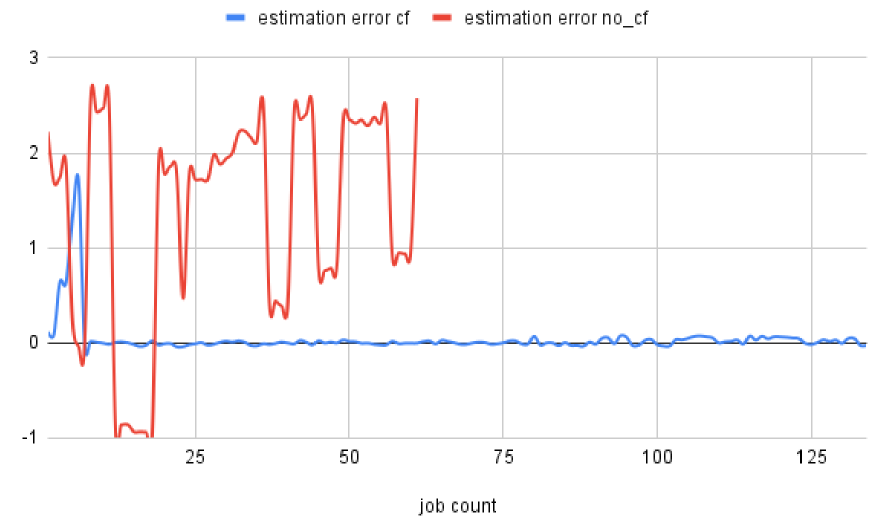


Figure 22: Average processing time and waiting in queue time

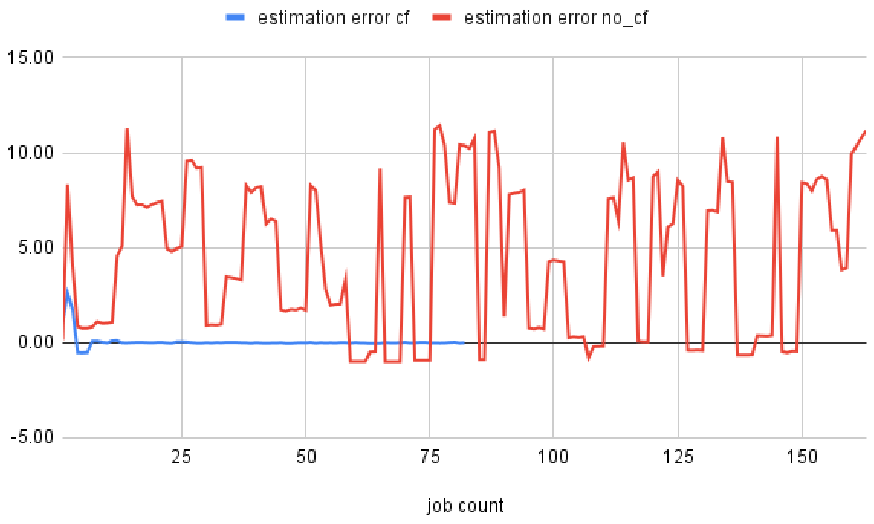
when the self-correction logic was applied, despite the increased data load. Conversely, slower workers processed 26,009,682 fewer lines overall, indicating that they were assigned less complex jobs. This suggests that their assignments were better matched to their capabilities under the self-correcting workflow. Additionally, Figure 22 highlights two crucial performance indicators: the average processing time for jobs remained consistent between the two configurations - implying comparable physical characteristics and workloads among workers - yet, with self-correcting workers, the average queue time (time after job allocation but before processing begins) decreased by 36%. These variations in job allocation can be attributed to the reduction of estimation errors resulting from the refined estimation formula, which leverages historical execution data over time, as illustrated in Figure 23. The red lines in this figure represent the discrepancies between estimated and actual processing times for two workers within the execution environment, whereas the blue lines depict these discrepancies when the Feedback-Based Estimator is applied. Clearly, the estimation errors diminish with an increase in the number of jobs processed by each worker node over time.

	No correction	**With self-correction**
slow worker 1	48 661 595	36 216 561
slow worker 2	47 798 969	34 234 321
average worker 1	81 455 695	83 988 390
average worker 2	86 675 345	77 706 130
fast worker	114 486 605	146 925 659

Table 12: Number of lines processed per worker with and without self-correcting



(a) Worker 1



(b) Worker 2

Figure 23: Estimation error (percentage) change over the number of processed jobs

Furthermore, it should be noted that the speed of the fastest worker significantly influences the start/pause bidding dynamics. The fastest worker, likely reaching the dedicated queue job threshold sooner than other workers, prompts a more frequent initiation of the bidding process. Consequently, average and slower workers require additional time to adjust their bids accurately. This adjustment process involves updating the estimation formula upon job completion, but misaligned bidding occurs whenever the threshold set by the fastest worker triggers the process. Thus, the system necessitates an overhead of (x) jobs to recalibrate, resulting in less pronounced speed improvements for smaller workloads compared to larger datasets. For instance, processing 120 repositories with four commits each yields an approximate 13% speedup; larger datasets are likely to exhibit even more significant performance gains.

7.4 Threats to Validity

This section reflects on the threats to validity associated with the evaluation of the dynamic estimation model for self-correcting workers, following the standard classification proposed by Wohlin et al. [64]: *conclusion*, *internal*, *construct*, and *external validity.*

7.4.1 Internal Validity

Internal validity refers to whether the observed improvements can be attributed to the dynamic estimation model rather than confounding factors.

- **Consistent baseline and comparison.** Both configurations with and without the self-corrective models were performed under the same scheduler and job definitions. However, since worker speeds and initial estimation parameters were predefined, residual effects from these values may confound the results — especially during the early stages before the model converges.

- **Fast-worker trigger effect.** Faster workers reach their dedicated queue thresholds earlier, which influences the timing of when bidding resumes. This dynamic may unintentionally favour faster workers in the early stages of learning, impacting the rate of convergence for slower workers' estimation models.

7.4.2 Construct Validity

Construct validity addresses whether the chosen metrics and setup accurately reflect the theoretical concepts being tested here: estimation accuracy, adaptability, and workload balance.

- **Controlled synthetic noise.** Execution times were simulated using controlled delays based on worker speed and job size, with noise factors added to approximate real-world variability. While this allowed for consistent comparisons, the artificial nature of this noise may not reflect all patterns of stochastic variability seen in practice.

- **Metric selection.** The evaluation uses job completion time, cache miss count, data transferred (MB), and average waiting time in queue as performance indicators. These metrics offer a reasonable approximation of system responsiveness and scheduling efficiency. However, queue time is an indirect measure of worker balance and may not capture all dimensions of fairness or adaptability.

- **Simplified estimation model.** The dynamic model was implemented as a linear relationship over a single job attribute (line count). This abstraction enables clarity and repeatability but may not capture multidimensional influences on job execution time, such as I/O intensity or branching structures in code.

7.4.3 External Validity

External validity refers to how well the findings generalise to other systems, workflows, or environments.

- **Task and domain specificity.** The experiments focused on a repository mining scenario involving large Java repositories and commit-level job definitions. Other domains with different data structures, runtime characteristics, or job types may not benefit equally from the dynamic estimation mechanism.

- **Overhead in lightweight workloads.** The benefits of dynamic estimation may diminish for workflows with very short-lived jobs, where the overhead of maintaining and applying estimation logic could outweigh performance gains. In such cases, simpler static estimators may suffice.

- **Synthetic environment.** The evaluation was conducted in a controlled local environment with simulated execution times. While this allowed fine-grained control over noise and performance factors, the absence of true distributed deployment (e.g., network variability, hardware contention) limits generalisability to real-world systems.

8 Conclusions and Future Work

This thesis has proposed methods aimed at enhancing the overall efficiency of distributed execution of MSR workflows. The proposed approaches focus on minimising end-to-end execution time while optimising data locality through job scheduling. The hypothesis of this research was articulated in Section 3.2:

> *A specialised locality-aware job orchestration framework and algorithm can outperform current methods for distributable and locality-intensive repository mining workflows executed in a heterogeneous environment.*

The structure of the remaining sections in this chapter is as follows. Section 8.1 provides a comprehensive overview of the primary discussions presented in the thesis. Section 8.2 elaborates on the contributions made to the field. Finally, Section 8.3 outlines potential directions for future research.

8.1 Summary

In this thesis, we have examined methodologies for addressing data locality within distributed repository mining and job allocation frameworks. The research is structured to enhance the understanding of distributed data processing in the context of mining software repositories. **Chapter 2** established the theoretical foundation by conducting a comprehensive literature review that scrutinised the capabilities and limitations of existing MSR tools and distributed data processing frameworks. This chapter effectively identified the critical gaps in current methodologies, setting the stage for subsequent investigative work.

Chapter 3 described the research problem, detailed the hypothesis, and defined research objectives, thereby providing a framework that underpinned the empirical investigations. This chapter also established the specific criteria and metrics required to evaluate the effectiveness and efficiency of the proposed methodologies.

Chapter 4 presented the experiments conducted to assess the performance of potential baseline frameworks. This involved comparative analysis of existing frameworks' performance with regards to various types of workloads executed by workers with varying physical characteristics to determine the most suitable environment for our research objectives, focusing on computational efficiency, job scheduling capabilities, and data locality management.

Chapters 5, **6**, and **7** embody the main contributions of this thesis, introducing novel strategies developed to optimise execution time through improved data locality handling. These chapters provide in-depth explanations of the distributed job allocation methodologies, supported by comprehensive experimental results that highlight the performance variations when compared to the baseline.

Finally, **Chapter 8** synthesises the key findings, offering a detailed summary of the insights obtained from the research, along with several concrete proposals for future work.

8.2 Thesis Contributions

The primary contributions of this research encompass the development of the Resource-Registry and the Bidding Scheduler (Chapters 5 and 6) for increasing data locality and reducing the overall execution time of LID MSR workflows, as well as the introduction of a Feedback-Based Estimator in Chapter 7 for minimising estimation errors when scheduling jobs based on completion time estimates. Specifically, this research includes: (a) the design and implementation of the Resource-Registry scheduler, which assigns jobs by aligning computational tasks with the location of necessary data resources, thereby minimising data transfer overhead; (b) the introduction of the Bidding Scheduler, which employs a competitive auction-based mechanism to dynamically allocate tasks to worker nodes based on bid values that reflect each node's current load and capabilities; (c) the integration of a Feedback-Based Estimator within the Bidding Scheduler to adjust for discrepancies in worker's performance estimations and actual performances, thus ensuring more accurate and efficient job distribution; and (d) a comprehensive evaluation and analysis of the effectiveness of these schedulers in both homogeneous and heterogeneous worker environments, providing insights into their performance, strengths, and limitations, and proposing directions for future research.

The **Resource-Registry scheduler** is a methodology designed to improve data locality and reduce data load, thereby decreasing the end-to-end execution time. The scheduler shows effectiveness in homogeneous worker environments, where all nodes have similar processing capabilities and configurations, as well as in allocating jobs with large, non-repetitive data resources, as it addresses a limitation identified in the Baseline where all worker nodes first reject a job they do not possess the resource for, leading to diminished performances when processing non-repetitive data resources. The scheduler's performance is less effective in cases where jobs require long processing times and most of them need access to the same resource, therefore prolonging the end-to-end execution time. This happens because Resource-Registry prioritises data locality, overlooking the idleness of other workers in favour of one that holds the preferred resource. In heterogeneous environments, where worker nodes have varying processing speeds, this limitation becomes more pronounced. For example, if a slow worker node is assigned the majority of tasks due to its resource preference, the overall execution time can increase. This is because other, faster worker nodes remain underutilised.

The **Bidding Scheduler** presents a novel approach to decentralised job allocation. This methodology leverages the concept of "opinionated" nodes, which submit bids for jobs based on locally available data and individual worker conditions. Consequently, the scheduler achieves notable efficiency by considering intrinsic attributes of worker nodes, such as network and processing speeds, thereby addressing heterogeneity within the cluster.

The adaptive workload allocation is a key advantage of the Bidding Scheduler, dynamically distributing workloads according to data locality and node-specific capabilities, ensuring an equitable task distribution. For example, if a node possesses high network latency but superior processing power, the scheduler might prioritise computationally intensive tasks for this node while assigning network-heavy tasks to nodes with faster connectivity. Such adaptability is particularly beneficial for long-running workflows, where nodes might exhibit fluctuations in performance characteristics over time.

Moreover, increased worker utilisation is achieved by tailoring job schedules based on workers' physical characteristics and current resource availability, leading to an average worker usage increase in heterogeneous work environments, effectively balancing the load across the system. The Bidding Scheduler manages to decrease the end-to-end execution time through reduced data load and increased worker utilisation, striking a balance between data locality and load-balanced scheduling, thus delivering faster overall workflow completion rates, with observed speedups ranging from 1.23 to 3.30 times on average across various job and worker configurations.

Nonetheless, the Bidding Scheduler is not without its challenges and limitations. It may underperform for small or inexpensive jobs because the overhead from negotiating job distribution can outweigh its benefits in such scenarios. Additionally, its advantages diminish in homogeneous worker environments that process non-repetitive jobs of nearly same sizes, where the uniformity of workers and job negates the necessity for the bidding process. Accurate job completion time estimation is crucial for the scheduler's efficacy, as inaccuracies can lead to suboptimal scheduling and inefficient resource utilisation.

In conclusion, while the Bidding Scheduler represents an advancement in decentralised job allocation by incorporating data locality and worker heterogeneity into its decision-making processes, its success relies on the specific characteristics of the jobs and the computational environment. A detailed understanding and careful consideration of these factors are essential to leverage the full potential of the Bidding Scheduler, ensuring optimal performance across varied computational scenarios.

The **Feedback-Based Estimator approach** represents a method for refining job completion estimates within the Bidding Scheduler, thereby reducing estimation errors over time. By iteratively improving the estimation formula, this methodology enhances the overall efficiency and reduces the end-to-end execution time. One of its primary advantages lies in its flexibility, capable of accommodating various statistical models, such as the historic average, linear or polynomial regression. This allows for adjustments tailored to different data distributions and workflow characteristics. This approach improves the performance of "opinionated" nodes by enabling them to use historical data to make more accurate time predictions. This is especially advantageous in heterogeneous environments, where reducing estimation errors leads to better job assignments. Consequently, more capable nodes are tasked with more complex jobs, resulting in a balanced and efficient system.

However, the complexity of the learning models can introduce significant overheads, potentially offsetting the benefits of improved accuracy if the models become excessively complex. Moreover, selecting an inappropriate model for the data, such as using a linear regression model when a polynomial model would better capture the relationship between job attributes and execution time, can lead to suboptimal scheduling decisions and reduced performance gains. Additionally, this approach is less suited for short-running workflows, as the iterative improvement process requires substantial historical data to develop accurate models. Such data may not be available within the limited timeframe of short-running tasks, restricting the applicability of the Feedback-Based Estimator approach in these scenarios.

In summary, the Feedback-Based Estimator approach offers means to enhance job completion time predictions and optimise scheduler performance, particularly in heterogeneous computational environments. However, its successful implementation necessitates careful consideration of the learning model selection and the availability of sufficient historical data to ensure effectiveness and avoid potential issues.

8.3 Future Research

The Bidding Scheduler and the Feedback-Based Estimator approach have demonstrated considerable potential in optimising job allocation and execution within heterogeneous computational environments. However, further research and development are essential to address existing limitations and extend their applicability.

1. **Investigating applicability beyond software repository mining workflows**
   While the current implementation of the Bidding Scheduler has proven effective within the context of LID MSR workflows, there is value in exploring its applicability across a wider array of computational workflows. Future research should address its integration into workflows involving different types of locality-intensive computing, such as image processing or large-scale machine learning training. By applying the Bidding Scheduler to these varied scenarios, we could evaluate its versatility and scalability across diverse domains.

2. **Job reassignment and fault recovery**
   While job failures are currently recorded by Crossflow, the framework does not support automatic job reassignment. Future work should address this by enabling the Resource-Registry and Bidding Scheduler to detect failed or stalled jobs and reassign them to other available workers. This could be achieved through heartbeat monitoring, execution timeouts, and soft-state tracking, and would contribute to improved fault tolerance and resilience under failure-prone conditions.

3. **Adaptive model selection for online estimators.** While this work used a linear Feedback-Based Estimator for its simplicity and efficiency, future versions could evaluate multiple candidate models (e.g., linear, quadratic, cubic) and select the best fit using goodness-of-fit metrics such

as mean squared error or adjusted R^2. By maintaining a sliding window of recent jobs, each worker could incrementally compare models and adapt its prediction strategy accordingly. This would enable each worker to remain both accurate and generalisable as job characteristics evolve.

4. **Monetary cost consideration**
   Another possible direction is to extend the Bidding Scheduler to factor in dual optimisation criteria — time and monetary cost. This involves incorporating mechanisms to optimise not only for task completion time but also for associated costs, particularly in cloud environments where resource usage translates directly into financial expenditure. While the Bidding Scheduler currently supports implementations that could minimise either time or cost, future work could develop models to find an optimal balance between both, depending on the priorities of the workflow. For example, in a cloud-based data processing scenario, the scheduler could choose between more expensive, high-performance instances for critical tasks and more cost-effective instances for less time-sensitive tasks.

5. **Mitigating estimation errors in early stages**
   Addressing estimation errors before accumulating sufficient historical execution data is crucial for the effectiveness of the Feedback-Based Estimator mechanism. Future research should explore methods for initial estimation error mitigation. Techniques such as leveraging similar jobs' performance data can be investigated. For instance, employing transfer learning where a model created based on one type of job can inform estimations for another similar new job, could be an interesting direction to explore to reduce early-stage estimation inaccuracies.

6. **Applicability in homogeneous worker environments**
   While the Feedback-Based Estimator approach has primarily proven beneficial for heterogeneous worker environments, future research should also investigate its applicability in homogeneous worker environments. In such settings, instead of relying on decentralised historical data tailored to individual worker characteristics, a shared repository of historical data could be utilised to collectively improve the estimation formula. For instance, in a homogeneous environment where worker nodes have similar performance characteristics, a centralised database containing historical job execution data could be maintained. This repository would allow the entire cluster to benefit from collective historical insights, thereby refining job completion time predictions to enable efficient scheduling decisions more rapidly than it is the case with storing the execution data separate for each worker node.

References

[1] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: an enabling language and infrastructure for ultra-large scale MSR studies. In Christian Bird, Tim Menzies, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 593–621. Morgan-Kaufmann, Waltham, MA, 2015.

[2] Spark - cluster mode overview, . URL `https://spark.apache.org/docs/3.5.3/cluster-overview.html#:~:text=Once%20connected%2C%20Spark%20acquires%20executors,to%20the%20executors%20to%20run`. Last visited: 06.12.2024.

[3] Siwoon Son, Sanghun Lee, Myeong-Seon Gil, Mi-Jung Choi, and Yang-Sae Moon. Locality aware traffic distribution in apache storm for energy analytics platform. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 721–724, 2018. doi: 10.1109/BigComp.2018.00135.

[4] 30 most popular NPM packages for Node JS developers. URL `https://www.turing.com/blog/top-npm-packages-for-node-js-developers/`. Last visited: 08.08.2023.

[5] BPMN. URL `https://www.bpmn.org`. Last visited: 16.06.2025.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `https://doi.org/10.1145/1327452.1327492`.

[7] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: An infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154, 2019. doi: 10.1109/MSR.2019.00031.

[8] Tamanna Siddiqui and Ausaf Ahmad. *Data Mining Tools and Techniques for Mining Software Repositories: A Systematic Review*, pages 717–726. 01 2018. ISBN 978-981-10-6619-1. doi: 10.1007/978-981-10-6620-7_70.

[9] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3264598.

[10] R.moghaddas and M.Houshmand. Job-shop scheduling problem with sequence dependent setup times. *Lecture Notes in Engineering and Computer Science*, 2169, 03 2008.

[11] Optimization and approximation in deterministic sequencing and scheduling: a survey. 5:287–326, 1979.

[12] Ali Allahverdi, Chee-Yee Ng, T. C. Edwin Cheng, and Mikhail Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, 2008. doi: 10.1016/j.ejor.2006.06.060.

[13] D. Sharma and A. Jain. A review on job-shop scheduling with setup times. *International Journal of Computer Applications*, 123(2):35–39, 2015. doi: 10.5120/ijca2015905483.

[14] Fabian Heseding, Willy Scheibel, and Jürgen Döllner. Tooling for time- and space-efficient Git repository mining. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 413–417, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528503. URL `https://doi.org/10.1145/3524842.3528503`.

[15] Adam Tutko, Austin Z. Henley, and Audris Mockus. How are software repositories mined? A systematic literature review of workflows, methodologies, reproducibility, and tools. *ArXiv*, abs/2204.08108, 2022. URL `https://api.semanticscholar.org/CorpusID:248228041`.

[16] Darshankumar Gorasiya. Comparison of open-source data stream processing engines: Spark Streaming, Flink and Storm. 09 2019. doi: 10.13140/RG.2.2.16747.49440.

[17] Pandas. URL `https://pandas.pydata.org`. Last visited: 26.06.2024.

[18] Matplotlib. URL `https://matplotlib.org`. Last visited: 26.06.2024.

[19] Fabian Heseding, Willy Scheibel, and Jürgen Döllner. Tooling for time- and space-efficient Git repository mining. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 413–417, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528503. URL `https://doi.org/10.1145/3524842.3528503`.

[20] Vitalis Salis and Diomidis Spinellis. RepoFS: File system view of Git repositories. *SoftwareX*, 9:288–292, January 2019.

[21] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Perceval: Software project data at your will. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 1–4, 2018.

[22] Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, and Jens Grabowski. The SmartSHARK ecosystem for software repository mining. In *Proceedings of the ACM/IEEE 42nd International*

Conference on Software Engineering: Companion Proceedings, ICSE '20, page 25–28, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371223. doi: 10.1145/3377812.3382139. URL `https://doi.org/10.1145/3377812.3382139`.

[23] MongoDB. URL `https://www.mongodb.com`. Last visited: 30.11.2024.

[24] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source vcs data. *Empirical Softw. Engg.*, 26(2), March 2021. ISSN 1382-3256. doi: 10.1007/s10664-020-09905-9. URL `https://doi.org/10.1007/s10664-020-09905-9`.

[25] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431, 2013. doi: 10.1109/ICSE.2013.6606588.

[26] Sawzall language. URL `https://code.google.com/archive/p/szl/wikis/Overview.wiki`. Last visited: 03.07.2024.

[27] About Boa language and infrastructure. URL `https://boa.cs.iastate.edu`. Last visited: 26.06.2024.

[28] IDE Support - Boa. URL `https://boa.cs.iastate.edu/ide/index.php`. Last visited: 02.07.2024.

[29] Hadoop. URL `https://hadoop.apache.org`. Last visited: 05.09.2022.

[30] HDFS. URL `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Last visited: 04.08.2022.

[31] Tableau. URL `https://www.tableau.com`. Last visited: 03.07.2024.

[32] Microsoft Power BI. URL `https://www.microsoft.com/en-us/power-platform/products/power-bi`. Last visited: 03.07.2024.

[33] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in MapReduce. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426, 2012. doi: 10.1109/CCGrid.2012.42.

[34] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: an in-depth study. 3(1–2):472–483, sep 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920903. URL `https://doi.org/10.14778/1920841.1920903`.

[35] Brian Sigurdson, Samuel W. Flint, and Robert Dyer. Boidae: Your personal mining platform. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 40–43, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705021. doi: 10.1145/3639478.3640026. URL `https://doi.org/10.1145/3639478.3640026`.

[36] About large files on GitHub, . URL `https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github`. Last visited: 01.12.2024.

[37] Peng Wang, Le Cao, Chunbo Lai, Leqi Zou, Guangyu Sun, and Jason Cong. InterFS: An interplanted distributed file system to improve storage utilization. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335546. doi: 10.1145/2797022.2797036. URL `https://doi.org/10.1145/2797022.2797036`.

[38] Tom White. *Hadoop: the definitive guide*. O'Reilly Media, Inc., 4th edition, 2015. ISBN 1491901632.

[39] Thanh Duong, Quoc Luu, and Hung Nguyen. Performance of distributed file systems on cloud computing environment: An evaluation for small-file problem. *ArXiv*, abs/2312.17524, 2023. URL `https://api.semanticscholar.org/CorpusID:266691006`.

[40] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *2009 IEEE International Conference on Cluster Computing and Workshops*.

[41] Dimitris S. Kolovos, Patrick Neubauer, Konstantinos Barmpis, Nicholas Matragkas, and Richard F. Paige. Crossflow: a framework for distributed mining of software repositories. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 155–159. IEEE / ACM, 2019. ISBN 978-1-7281-3412-3.

[42] Chen He, Ying Lu, and David Swanson. Matchmaking: A new MapReduce scheduling technique. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 40–47, 2011. doi: 10.1109/CloudCom.2011.16.

[43] Eclipse Modeling Framework. URL `https://eclipse.dev/modeling/emf/`. Last visited: 04.07.2024.

[44] Apache Spark, . URL `https://spark.apache.org`. Last visited: 07.07.2023.

[45] Apache Flink. URL `https://flink.apache.org`. Last visited: 07.07.2023.

[46] Apache Storm. URL `https://storm.apache.org`. Last visited: 07.07.2023.

[47] Spark data locality, . URL `https://spark.apache.org/docs/latest/tuning.html#data-locality`. Last visited: 23.05.2024.

[48] Matei Zaharia, Dhruba Borthakur, Joydeep Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. pages 265–278, 01 2010. doi: 10.1145/1755913.1755940.

[49] Theodoros Toliopoulos and Anastasios Gounaris. Adaptive distributed partitioning in Apache Flink. pages 127–132, 04 2020. doi: 10.1109/ICDEW49219.2020.00012.

[50] Hongjian Li, Jianglin Xia, Wei Luo, and Hai Fang. Cost-efficient scheduling of streaming applications in Apache Flink on cloud. *IEEE Transactions on Big Data*, 9(4):1086–1101, 2023. doi: 10.1109/TBDATA.2022.3233031.

[51] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[52] Subasish Mohapatra, Subhadarshini Mohanty, and K.Smruti Rekha. Analysis of different variants in Round-Robin algorithms for load balancing in cloud computing. *International Journal of Computer Applications*, 69:17–21, 05 2013. doi: 10.5120/12103-8221.

[53] Laila Bouhouch, Mostapha Zbakh, and Claude Tadonki. Online task scheduling of Big Data applications in the cloud environment. *Information*, 14(5), 2023. ISSN 2078-2489. doi: 10.3390/info14050292. URL `https://www.mdpi.com/2078-2489/14/5/292`.

[54] Siwoon Son and Yang-Sae Moon. Locality/fairness-aware job scheduling in distributed stream processing engines. *Electronics*, 9(11), 2020. ISSN 2079-9292. doi: 10.3390/electronics9111857. URL `https://www.mdpi.com/2079-9292/9/11/1857`.

[55] What is Spark Stage?, . URL
`https://sparkbyexamples.com/spark/what-is-spark-stage/#:~:text=Spark%20executes%20each%20stage%20in,different%20nodes%20in%20the%20cluster`. Last visited: 25.08.2024.

[56] Nishank Garg and Dharanipragada Janakiram. Sparker: Optimizing Spark for heterogeneous clusters. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 1–8, 2018. doi: 10.1109/CloudCom2018.2018.00017.

[57] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 491–499, 1988. doi: 10.1109/DCS.1988.12552.

[58] Dong Lu, Huanyuan Sheng, and P. Dinda. Size-based scheduling policies with inaccurate scheduling information. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings.*, pages 31–38, 2004. doi: 10.1109/MASCOT.2004.1348179.

[59] Matteo Dell'Amico, Damiano Carra, Mario Pastorelli, and Pietro Michiardi. Revisiting size-based scheduling with estimated job sizes. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 411–420, 2014. doi: 10.1109/MASCOTS.2014.57.

[60] Thamarai Selvi Somasundaram, Balachandar R. Amarnath, R. Kumar, P. Balakrishnan, K. Rajendar, R. Rajiv, G. Kannan, G. Rajesh Britto, E. Mahendran, and B. Madusudhanan. CARE resource broker: A framework for scheduling and supporting virtual resource management. *Future Generation Computer Systems*, 26(3):337–347, 2010. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2009.10.005. URL `https://www.sciencedirect.com/science/article/pii/S0167739X09001551`.

[61] Uddalok Sen, Madhulina Sarkar, and Nandini Mukherjee. Hybrid job scheduling in distributed systems based on clone detection. *2020 Sixth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pages 78–83, 2020. URL `https://api.semanticscholar.org/CorpusID:231645661`.

[62] Gemma Reig, Javier Alonso, and Jordi Guitart. Prediction of job resource requirements for deadline schedulers to manage high-level SLAs on the cloud. In *2010 Ninth IEEE International Symposium on Network Computing and Applications*, pages 162–167, 2010. doi: 10.1109/NCA.2010.28.

[63] Spark - Job scheduling, . URL `https://spark.apache.org/docs/latest/job-scheduling.html#job-scheduling`. Last visited: 28.05.2024.

[64] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated, 2012. ISBN 3642290434. doi: 10.1007/978-3-642-29044-2.

[65] Weina Wang, Matthew Barnard, and Lei Ying. Decentralized scheduling with data locality for data-parallel computation on peer-to-peer networks. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing, Allerton 2015*, pages 337–344. Institute of Electrical and Electronics Engineers Inc., April 2016. doi: 10.1109/ALLERTON.2015.7447024.

[66] Aysan Rasooli and Douglas G. Down. A hybrid scheduling approach for scalable heterogeneous Hadoop systems. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1284–1291, 2012. doi: 10.1109/SC.Companion.2012.155.

[67] Nenavath Srinivas Naik, Atul Negi, Tapas Bapu B.R., and R. Anitha. A data locality based scheduler to enhance MapReduce performance in heterogeneous environments. *Future Generation Computer Systems*, 90: 423–434, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.07.043. URL `https://www.sciencedirect.com/science/article/pii/S0167739X18308379`.

[68] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering*, 26(2): 22, February 2021. doi: 10.1007/s10664-020-09905-9.

[69] D.P. Anderson. BOINC: a system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004. doi: 10.1109/GRID.2004.14.

[70] SETI@home. URL `https://setiathome.berkeley.edu`. Last visited: 24.07.2024.

[71] Rosetta@home. URL `https://boinc.bakerlab.org`. Last visited: 24.07.2024.

[72] BOINC - Client scheduling. URL `https://boinc.berkeley.edu/trac/wiki/ClientSched`. Last visited: 24.07.2024.

[73] Pooja Dewan and Sanjay Joshi. Auction-based distributed scheduling in a dynamic job shop environment. *International Journal of Production Research*, 40(5):1173–1191, 2002. doi: 10.1080/00207540110098445.

[74] Michael T. Howard Thomas W. Malone, Richard E. Fikes. Enterprise: A market-like task scheduler for distributed computing environments. *CISR WP ; no. 111.Working paper (Sloan School of Management)*, pages 1537–84, 1983. URL `http://hdl.handle.net/1721.1/47750`.

[75] Quentin Baert, Anne Caron, Maxime Morge, and Jean-Christophe Routier. *Fair Multi-agent Task Allocation for Large Data Sets Analysis*, pages 24–35. 01 2016. ISBN 978-3-319-39323-0. doi: 10.1007/978-3-319-39324-7_3.

[76] Amit Kumar Singh, Piotr Dziurzanski, and Leandro Soares Indrusiak. Market-inspired dynamic resource allocation in many-core high performance computing systems. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 413–420, 2015. doi: 10.1109/HPCSim.2015.7237070.

[77] Zhengyuan Zhou and Nicholas Bambos. A general model for resource allocation in utility computing. In *2015 American Control Conference (ACC)*, pages 1746–1751, 2015. doi: 10.1109/ACC.2015.7170985.

[78] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *J. Artif. Int. Res.*, 1(1):1–23, August 1993. ISSN 1076-9757. doi: 10.5555/1618595.1618596.

[79] Zhichao Zhao, Fei Chen, T-H Chan, and Chuan Wu. Double auction for resource allocation in cloud computing. pages 301–308, 01 2017. doi: 10.5220/0006145403010308.

[80] Key concepts for burstable performance instances. URL `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html`. Last visited: 04.11.2024.

[81] Amazon EC2. URL `https://aws.amazon.com/ec2/pricing/`. Last visited: 13.08.2024.

[82] How to speed up Git cloning, . URL `https://hatchjs.com/cloning-git-repository-taking-too-long`. Last visited: 01.01.2025.

[83] How to optimize Git performance in large repositories, . URL `https://blog.pixelfreestudio.com/how-to-optimize-git-performance-in-large-repositories`. Last visited: 01.01.2025.

[84] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: how scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, February 2006. ISSN 1533-5399. doi: 10.1145/1125274.1125276. URL `https://doi.org/10.1145/1125274.1125276`.

[85] Linus Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968. ISSN 0030364X, 15265463. URL `http://www.jstor.org/stable/168596`.

[86] Rachel Mailach and Douglas G. Down. Scheduling jobs with estimation errors for multi-server systems. In *2017 29th International Teletraffic Congress (ITC 29)*, volume 1, pages 10–18, 2017. doi: 10.23919/ITC.2017.8064334.

[87] Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Scheduling in a dynamic heterogeneous distributed system using estimation error. *Journal of Parallel and Distributed Computing*, 68(11): 1452–1462, 2008. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2008.07.004. URL `https://www.sciencedirect.com/science/article/pii/S0743731508001287`.

[88] Douglas Down. Open problem — size-based scheduling with estimation errors. *Stochastic Systems*, 9, 09 2019. doi: 10.1287/stsy.2019.0041.

[89] Madhulina Sarkar, Triparna Mondal, Sarbani Roy, and Nandini Mukherjee. Resource requirement prediction using clone detection technique. *Future Gener. Comput. Syst.*, 29(4):936–952, June 2013. ISSN 0167-739X. doi: 10.1016/j.future.2012.09.010. URL `https://doi.org/10.1016/j.future.2012.09.010`.

[90] Pubali Datta, Madhulina Sarkar, Sarbani Roy, and Nandini Mukherjee. Resource requirement prediction techniques for near miss clone jobs. In *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, pages 184–190, 2013. doi: 10.1049/cp.2013.2590.

[91] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, volume 1, pages 283–289 vol.1, 2000. doi: 10.1109/HPC.2000.846563.

[92] Madhulina Sarkar, Sarbani Roy, and Nandini Mukherjee. Feedback-guided analysis for resource requirements in large distributed system. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 596–597, 2010. doi: 10.1109/CCGRID.2010.90.

[93] Arshad Ali, Ashiq Anjum, Julian Bunn, Richard Cavanaugh, Frank van Lingen, Richard Mcclatchey, Muhammad Mehmood, Harvey Newman, Conrad Steenberg, Michael Thomas, and Ian Willers. Predicting the resource requirements of a job submission. 10 2004.

[94] Mahdi Bohlouli and Morteza Analoui. Grid-HPA: Predicting resource requirements of a job in the grid computing environment. *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 2:2994–2998, 2008. URL `https://api.semanticscholar.org/CorpusID:16238483`.

[95] Erica Briscoe and Jacob Feldman. Conceptual complexity and the bias/variance tradeoff. *Cognition*, 118(1):2–16, 2011. ISSN 0010-0277. doi: https://doi.org/10.1016/j.cognition.2010.10.004. URL `https://www.sciencedirect.com/science/article/pii/S0010027710002295`.

[96] Hardev Ranglani. Empirical analysis of the bias-variance tradeoff across machine learning models. *Machine Learning and Applications: An International Journal*, 11:01–12, 12 2024. doi: 10.5121/mlaij.2024.11401.

[97] R.J. Carroll, D. Ruppert, and L.A. Stefanski. *Measurement Error in Nonlinear Models*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1995. ISBN 9780412047213. URL `https://books.google.rs/books?id=FS-x3tPdXeMC`.

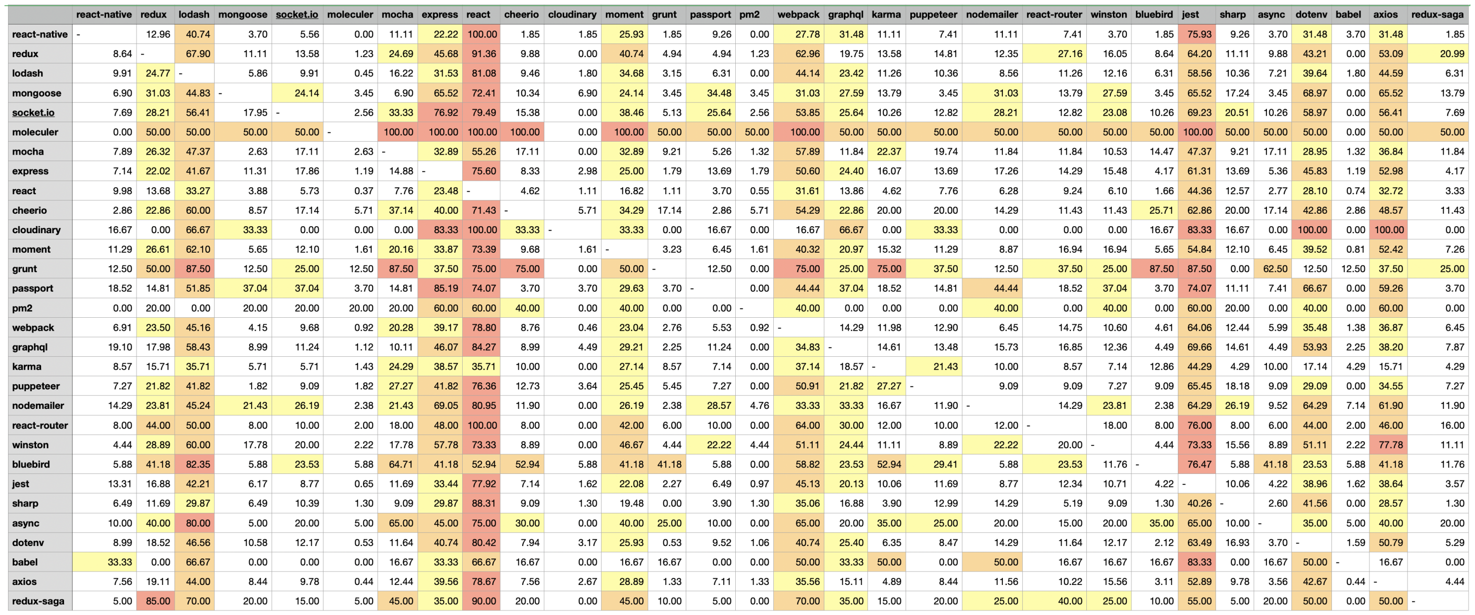
A Source Code and Artefacts

The complete source code for this research is available in the following GitHub repository:

`https://github.com/ana-markovic/lid-msr`

This repository includes project setups necessary to replicate the conducted experiments for baseline selection, as well as the evaluation of the Resource-Registry scheduler, Bidding Scheduler, and the Feedback-Based Estimator approach. It also contains a source code setup aimed at packaging the LID MSR workflow with the Bidding and Baseline scheduler for execution in cloud environments.

B Appendix: Additional Figures



	react-native	redux	lodash	mongoose	socket.io	moleculer	mocha	express	react	cheerio	cloudinary	moment	grunt	passport	pm2	webpack	graphql	karma	puppeteer	nodemailer	react-router	winston	bluebird	jest	sharp	async	dotenv	babel	axios	redux-saga
react-native	-	12.96	40.74	3.70	5.56	0.00	11.11	22.22	100.00	1.85	1.85	25.93	1.85	9.26	0.00	27.78	31.48	11.11	7.41	11.11	7.41	3.70	1.85	75.93	9.26	3.70	31.48	3.70	31.48	1.85
redux	8.64	-	67.90	11.11	13.58	1.23	24.69	45.68	91.36	9.88	0.00	40.74	4.94	4.94	1.23	62.96	19.75	13.58	14.81	12.35	27.16	16.05	8.64	64.20	11.11	9.88	43.21	0.00	53.09	20.99
lodash	9.91	24.77	-	5.86	9.91	0.45	16.22	31.53	81.08	9.46	1.80	34.68	3.15	6.31	0.00	44.14	23.42	11.26	10.36	8.56	11.26	12.16	6.31	58.56	10.36	7.21	39.64	1.80	44.59	6.31
mongoose	6.90	31.03	44.83	-	24.14	3.45	6.90	65.52	72.41	10.34	6.90	24.14	3.45	34.48	3.45	31.03	27.59	13.79	3.45	31.03	13.79	27.59	3.45	65.52	17.24	3.45	68.97	0.00	65.52	13.79
socket.io	7.69	28.21	56.41	17.95	-	2.56	33.33	76.92	79.49	15.38	0.00	38.46	5.13	25.64	2.56	53.85	25.64	10.26	12.82	28.21	12.82	23.08	10.26	69.23	20.51	10.26	58.97	0.00	56.41	7.69
moleculer	0.00	50.00	50.00	50.00	50.00	-	100.00	100.00	100.00	100.00	0.00	100.00	50.00	50.00	50.00	100.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	100.00	50.00	50.00	50.00	0.00	50.00	50.00
mocha	7.89	26.32	47.37	2.63	17.11	2.63	-	32.89	55.26	17.11	0.00	32.89	9.21	5.26	1.32	57.89	11.84	22.37	19.74	11.84	11.84	10.53	14.47	47.37	9.21	17.11	28.95	1.32	36.84	11.84
express	7.14	22.02	41.67	11.31	17.86	1.19	14.88	-	75.60	8.33	2.98	25.00	1.79	13.69	1.79	50.60	24.40	16.07	13.69	17.26	14.29	15.48	4.17	61.31	13.69	5.36	45.83	1.19	52.98	4.17
react	9.98	13.68	33.27	3.88	5.73	0.37	7.76	23.48	-	4.62	1.11	16.82	1.11	3.70	0.55	31.61	13.86	4.62	7.76	6.28	9.24	6.10	1.66	44.36	12.57	2.77	28.10	0.74	32.72	3.33
cheerio	2.86	22.86	60.00	8.57	17.14	5.71	37.14	40.00	71.43	-	5.71	34.29	17.14	2.86	5.71	54.29	22.86	20.00	20.00	14.29	11.43	11.43	25.71	62.86	20.00	17.14	42.86	2.86	48.57	11.43
cloudinary	16.67	0.00	66.67	33.33	0.00	0.00	0.00	83.33	100.00	33.33	-	33.33	0.00	16.67	0.00	16.67	66.67	0.00	33.33	0.00	0.00	0.00	16.67	83.33	16.67	0.00	100.00	0.00	100.00	0.00
moment	11.29	26.61	62.10	5.65	12.10	1.61	20.16	33.87	73.39	9.68	1.61	-	3.23	6.45	1.61	40.32	20.97	15.32	11.29	8.87	16.94	16.94	5.65	54.84	12.10	6.45	39.52	0.81	52.42	7.26
grunt	12.50	50.00	87.50	12.50	25.00	12.50	87.50	37.50	75.00	75.00	0.00	50.00	-	12.50	0.00	75.00	25.00	75.00	37.50	12.50	37.50	25.00	87.50	87.50	0.00	62.50	12.50	12.50	37.50	25.00
passport	18.52	14.81	51.85	37.04	37.04	3.70	14.81	85.19	74.07	3.70	3.70	29.63	3.70	-	0.00	44.44	37.04	18.52	14.81	44.44	18.52	37.04	3.70	74.07	11.11	7.41	66.67	0.00	59.26	3.70
pm2	0.00	20.00	0.00	20.00	20.00	20.00	20.00	60.00	60.00	40.00	0.00	40.00	0.00	0.00	-	40.00	0.00	0.00	0.00	40.00	0.00	40.00	0.00	60.00	20.00	0.00	40.00	0.00	60.00	0.00
webpack	6.91	23.50	45.16	4.15	9.68	0.92	20.28	39.17	78.80	8.76	0.46	23.04	2.76	5.53	0.92	-	14.29	11.98	12.90	6.45	14.75	10.60	4.61	64.06	12.44	5.99	35.48	1.38	36.87	6.45
graphql	19.10	17.98	58.43	8.99	11.24	1.12	10.11	46.07	84.27	8.99	4.49	29.21	2.25	11.24	0.00	34.83	-	14.61	13.48	15.73	16.85	12.36	4.49	69.66	14.61	4.49	53.93	2.25	38.20	7.87
karma	8.57	15.71	35.71	5.71	5.71	1.43	24.29	38.57	35.71	10.00	0.00	27.14	8.57	7.14	0.00	37.14	18.57	-	21.43	10.00	8.57	7.14	12.86	44.29	4.29	10.00	17.14	4.29	15.71	4.29
puppeteer	7.27	21.82	41.82	1.82	9.09	1.82	27.27	41.82	76.36	12.73	3.64	25.45	5.45	7.27	0.00	50.91	21.82	27.27	-	9.09	9.09	7.27	9.09	65.45	18.18	9.09	29.09	0.00	34.55	7.27
nodemailer	14.29	23.81	45.24	21.43	26.19	2.38	21.43	69.05	80.95	11.90	0.00	26.19	2.38	28.57	4.76	33.33	33.33	16.67	11.90	-	14.29	23.81	2.38	64.29	26.19	9.52	64.29	7.14	61.90	11.90
react-router	8.00	44.00	50.00	8.00	10.00	2.00	18.00	48.00	100.00	8.00	0.00	42.00	6.00	10.00	0.00	64.00	30.00	12.00	10.00	12.00	-	18.00	8.00	76.00	8.00	6.00	44.00	2.00	46.00	16.00
winston	4.44	28.89	60.00	17.78	20.00	2.22	17.78	57.78	73.33	8.89	0.00	46.67	4.44	22.22	4.44	51.11	24.44	11.11	8.89	22.22	20.00	-	4.44	73.33	15.56	8.89	51.11	2.22	77.78	11.11
bluebird	5.88	41.18	82.35	5.88	23.53	5.88	64.71	41.18	52.94	52.94	5.88	41.18	41.18	5.88	0.00	58.82	23.53	52.94	29.41	5.88	23.53	11.76	-	76.47	5.88	41.18	23.53	5.88	41.18	11.76
jest	13.31	16.88	42.21	6.17	8.77	0.65	11.69	33.44	77.92	7.14	1.62	22.08	2.27	6.49	0.97	45.13	20.13	10.06	11.69	8.77	12.34	10.71	4.22	-	10.06	4.22	38.96	1.62	38.64	3.57
sharp	6.49	11.69	29.87	6.49	10.39	1.30	9.09	29.87	88.31	9.09	1.30	19.48	0.00	3.90	1.30	35.06	16.88	3.90	12.99	14.29	5.19	9.09	1.30	40.26	-	2.60	41.56	0.00	28.57	1.30
async	10.00	40.00	80.00	5.00	20.00	5.00	65.00	45.00	75.00	30.00	0.00	40.00	25.00	10.00	0.00	65.00	20.00	35.00	25.00	20.00	15.00	20.00	35.00	65.00	10.00	-	35.00	5.00	40.00	20.00
dotenv	8.99	18.52	46.56	10.58	12.17	0.53	11.64	40.74	80.42	7.94	3.17	25.93	0.53	9.52	1.06	40.74	25.40	6.35	8.47	14.29	11.64	12.17	2.12	63.49	16.93	3.70	-	1.59	50.79	5.29
babel	33.33	0.00	66.67	0.00	0.00	0.00	16.67	33.33	66.67	16.67	0.00	16.67	16.67	0.00	0.00	50.00	33.33	50.00	0.00	50.00	16.67	16.67	16.67	83.33	0.00	16.67	50.00	-	16.67	0.00
axios	7.56	19.11	44.00	8.44	9.78	0.44	12.44	39.56	78.67	7.56	2.67	28.89	1.33	7.11	1.33	35.56	15.11	4.89	8.44	11.56	10.22	15.56	3.11	52.89	9.78	3.56	42.67	0.44	-	4.44
redux-saga	5.00	85.00	70.00	20.00	15.00	5.00	45.00	35.00	90.00	20.00	0.00	45.00	10.00	5.00	0.00	70.00	35.00	15.00	20.00	25.00	40.00	25.00	10.00	55.00	5.00	20.00	50.00	0.00	50.00	-

Figure 24: Co-occurence matrix of 30 popular npm libraries [4] in favoured GitHub projects