POLITECNICO DI TORINO

LM-32 (DM270) - Computer Engineering (Software)

Master of Science Thesis

An AI Agent System to Generate JMH Microbenchmarks



Supervisor
Prof. Luca Ardito

Co-Supervisor
Dr. Tommaso Fulcini

Candidate
Elena Favero

Chalmers Co-Supervisor
Prof. Philipp Leitner

Academic Year 2025-2026

Summary

Code performance is currently one of the key aspects of software development, since a performance inefficiency results in longer waiting times and higher costs, aspects that should be reduced as much as possible. Measuring code performance becomes essential and one way to evaluate it are microbenchmarks, performance tests that focus on individual instructions or methods. While several frameworks exist to support developers in writing them, including JMH for Java code, writing correct and meaningful microbenchmarks by hand remains a difficult and time-consuming task. Recent researches have therefore attempted to automate this process, although with certain limitations: AutoJMH [RCB16] requires at least one existing test case to generate the necessary initialization values, JUnit-to-JMH [Jan+23] depends on the availability of existing JUnit tests and Chatperftest [Zhi+25] uses Generative AI but without providing tools to the LLMs.

This project aims to provide a tool capable of automating the entire benchmark generation process. The system is designed as a multi-agent pipeline where AI agents, through tools and LLM calls, collaborate to identify target code and generate microbenchmarks. The system is structured into agentic steps: the target methods are first identified by the Planning Agent, followed by the Code Agent and the Configuration Agent, which handle the actual code generation, before the workflow concludes with compilation, running and validation checks, verifying compliance with a set of criteria such as the avoidance of JMH bad practices. The system is also equipped with feedback loops: whenever checks fail, the pipeline loops back with error context to regenerate the benchmark in a subsequent iteration.

The system was evaluated using metrics such as: the pass rate, defined as the proportion of benchmarks that are successfully generated by the system, the percentage of benchmarks considered meaningful and their effectiveness in detecting performance degradations, emulated through latency injection. In the case of less complex projects, the system attains an average pass rate ranging from 83.9% to 95.8%, whereas for the more complex projects, between 27.8% to 69.2%, decreasing significantly. However, even in this latter case, feedback loops maintain a fundamental role. Regarding the benchmarks generated for the less complex projects, the majority are meaningful (between 68% and 96%), and, with a single exception, all of them detect the injected performance degradation. In terms of time and cost, the generation of a single microbenchmark requires between 2.3 and 3.8 minutes with a worst-case cost of approximately $3 per hour (yielding 24 benchmarks), significantly lower than manual development.

The results obtained from highly complex projects revealed lower percentages, showing that the system's autonomous exploration of the source code is not always sufficient. To address this, future research could focus on tailoring the system to the requirements of each project, extending support to other programming languages and evaluating it across a broader range of codebases.

Acknowledgements

For the writing of this thesis, I would like to thank Professor Leitner, who was willing to supervise me from the very beginning and who introduced me to what doing research really means. His guidance was extremely enriching and formative: he encouraged me to think critically about problems while leaving me free to find my own solutions, to make mistakes, and to try new approaches. I would also like to thank Professor Ardito, who, despite my being on Erasmus abroad, remained available to support the project from the Politecnico's side.

I am also grateful to Lirong Yi, PhD, and Tommaso Fulcini, postdoctoral researcher, for their invaluable support. Both of them helped me work through difficulties by sharing their perspective as researchers and offering insights and solutions drawn from their experience in the field.

Your support made this special period in Sweden, and the writing of this thesis, a truly positive experience, one I will remember with joy in the years to come.

Contents

1 Introduction **1**
1.1 Problem Statement and Solution 1

2 Background **4**
2.1 JVM 5
2.2 JMH workflow 6
2.3 LLMs 7
2.4 AI Agents 7
2.5 Multi-Agent Systems 9

3 Related Work **11**
3.1 Efficiency in Performance Microbenchmarking 11
3.2 Test Generation 12
3.3 Benchmark Generation 12
3.4 LLMs for testing 13

4 Approach **14**
4.1 Methodology 14
4.2 System Design 15
4.2.1 Pipeline 15
4.2.2 Agents and tools 16
4.2.3 System architecture 16

5 Implementation **24**
5.1 Compilation and Runtime Environment 25
5.2 Frameworks 27
5.2.1 ReAct 27
5.2.2 LangChain - LangGraph 28
5.2.2.1 Tools 29
5.3 Method Verification - AST 30

6 Evaluation **31**
6.1 Results 31
6.1.1 RQ1 & RQ2: Pass Rate 32
6.1.2 RQ3: Meaningfulness and Latency Injection 38
6.1.2.1 Meaningfulness 38
6.1.2.2 Latency Injection 49
6.1.3 RQ4: Time and Cost 55
6.1.3.1 Time 55

6.1.3.2 Cost 56
6.2 Discussion 60
6.3 Limitations 61

7 Conclusions and Future Work **62**

List of Figures **62**

Bibliography **64**

Sitography **66**

1. Introduction

Performance, defined by metrics such as execution speed, throughput, CPU utilization and other quantitative metrics, has become a crucial aspect of software systems. As the physical process of shrinking transistor hits its limit, a slowdown associated with the end of the *Moore's law*, the necessity to optimize performance has become a strong priority. To understand how a system performs, testing is needed. While standard benchmarks evaluate the system as a whole, this work focuses on **microbenchmarks**. These are low-level tests designed to isolate and analyze the performance of specific methods or individual statements. This is because, to build a high-performance system, it is essential to first ensure that each small component performs efficiently.

To ensure stability, these performance characteristics must be thoroughly evaluated during the development of every software product, ideally making them part of the Continuous Integration (CI) pipeline [LGL21]. Software microbenchmarks are used for this purpose, as they function as small-scale performance tests designed to quantify specific metrics and detect potential degradation, such as slowdowns.

1.1 Problem Statement and Solution

The problem of microbenchmarks is that writing them in a way that yields correct insights on the performance of the benchmarked task turns out to be challenging. Developers often struggle to produce correct microbenchmarks, as they can easily become **ineffective or error-prone** due to various factors, including bad practices that, while known to be problematic, are common in open-source software [Laa+20] [Cos+19]. Because of the many pitfalls of performance microbenchmarking, the need for in-depth knowledge of methodology and a lack of appropriate tooling, multiple solutions have been introduced to help developers in writing reliable microbenchmarks, such as JMH for Java. Other approaches have focused on the automatic generation of JMH microbenchmarks, relieving developers from fully writing them manually. Some examples include: *AutoJMH* [RCB16], which generates them from code segments, *JUnit-to-JMH* [Jan+23], which derives them directly from JUnit tests, and *Chatperftest* [Zhi+25], which uses a single LLM model to produce them. However, these works have also revealed limitations, which include: requirement for manual configuration, a lack of scalability to large codebases, failure to prevent bad practices and a dependency on existing JUnit tests. Furthermore, for the *Chatperftest* tool, the generated benchmarks show a low percentage of executable and compilable benchmarks and only a single prompt is used for the initial generation phase.

This project stems from the need for a tool capable of automating the generation of microbenchmarks, to relieve developers of the complexities involved in writing syntactically and

architecturally correct performance tests, while also addressing the aforementioned issues. The architecture is structured as a multi-agent AI pipeline, wherein specialized agents work together to create a final benchmark suite that is both executable and compilable.

The main goal is to develop significant benchmarks that mitigate the risk of common errors while guaranteeing that there are no compilation or runtime issues. To make the process less prone to failures, the system includes multiple feedback loops: should a benchmark fail to compile, execute, or be considered insignificant by an evaluator agent, the workflow is reverted to an earlier stage of the pipeline. In this phase, the error context is fed back to the agent, enabling iterative refinement and correction. Furthermore, high complexity tasks within the pipeline have been decomposed into two distinct sub-phases, each assigned to a dedicated agent, to ensure a higher degree of accuracy and reliability in the final output.

This study addresses the following research questions:

- *RQ1: Can the agentic system generate compilable and executable microbenchmarks?*
  The pass rate is characterized as the proportion of benchmarks that the system successfully generates, which entails compiling, executing correctly and passing the validation check of the pipeline, in relation to the total number of benchmarks attempted. This metric is calculated automatically, considering the cases in which the system successfully terminates and those in which it stops after reaching a maximum threshold of generation attempts. Throughout the evaluation projects, the system attains an average pass rate ranging from 83.9% to 95.8%. However, this metric decreases for highly complex projects or those requiring demanding initial configurations, falling to percentages between 27.8% and 69.2%. Additionally, for simpler projects the system requires only a few attempts to generate valid benchmarks, whereas more intricate libraries converge more slowly, with later attempts still contributing a valuable share of successful benchmarks.

- *RQ2: Does allowing the pipeline to have feedback loops improve the quality of the generated microbenchmarks?*
  The pass rate was used in this case as well, as it not only reflects the overall generation success but also distinguishes between benchmarks successfully generated in the first iteration versus in subsequent ones. Across the least complex projects, feedback loops account for around one third (between 21.1% and 38.4%) of all attempted benchmarks, and in more complex projects they still play a key role, accounting for between 20% and 50% of all successfully generated benchmarks. These findings highlight that feedback loops are an essential step in the pipeline rather than an optional architectural choice.

- *RQ3: What is the quality of the generated benchmarks in terms of meaningfulness and ability to identify code slowdowns?*
  A static analysis evaluated 25 benchmarks per project based on specific criteria: target method invocation, realistic workload usage and the presence of polymorphism. Following this, a dynamic analysis was carried out on 20 generated microbenchmarks across all considered projects to determine their ability to detect an artificially introduced slowdown in the source code. The results reveal that most microbenchmarks are meaningful (between 68% and 96%) and that, among them, with a single exception, all successfully identify performance degradation in the invoked method. These analyses were restricted to three out of the five projects due to time and budget constraints.

- *RQ4: What is the computational and monetary cost of running the pipeline using the latest state-of-the-art large language models?*

This data was collected automatically by the system during execution, recording the elapsed time and the input/output tokens consumed by each LLM call. The generation of a single microbenchmark requires between 2.3 and 3.8 minutes and in the worst-case scenario, the cost of our system is approximately $3 per hour. Within this timeframe, the system successfully generates 24 benchmarks, a cost that is significantly lower than the compensation required for a developer to write them manually. Furthermore, the vast majority of tokens are invested in the code generation phase, which exceeds 80% of total usage across all three analyzed projects. These analyses were restricted to three out of the five projects due to time and budget constraints.

Overall, these results indicate that the proposed multi-agent system is capable of generating compilable, executable and meaningful microbenchmarks at a lower cost and time expense than manual writing. However, a key limitation lies in the complexity of the input projects which, due to non-trivial configuration setups and pitfalls, yield a lower pass rate and reduced meaningfulness.

2. Background

The Java Microbenchmark Harness (JMH) is the standard framework used to write and execute benchmarks written in Java. It was developed by OpenJDK from the necessity to have a tool that isolated and measured the performance of small portion of code. While unit and integration tests verify correctness, microbenchmarks offer insights into a system's low-level performance, measuring execution metrics at the method level to compare the behavior of various data structures and algorithmic variants under specific conditions.

The following figure shows the typical **JMH execution flow**:

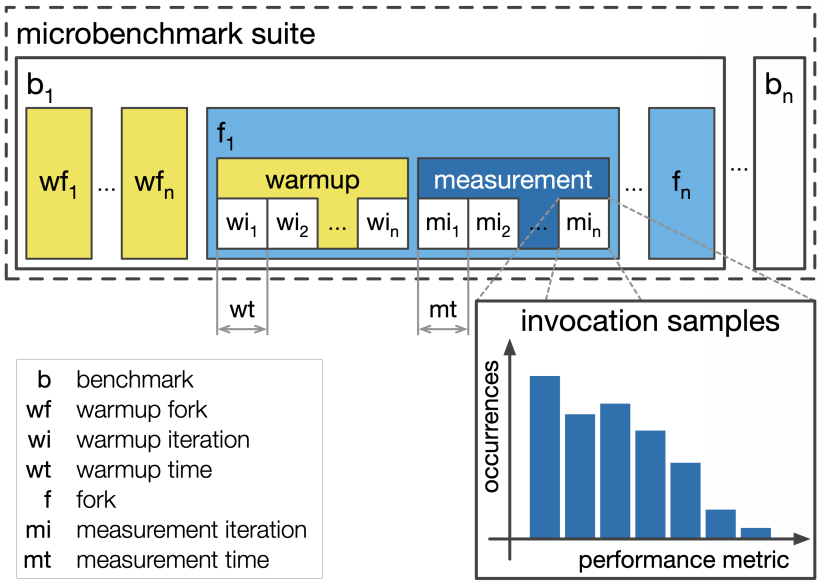


Figure 2.1: JMH execution flow [Source: [Laa+20]]

JMH identifies all methods annotated with the `@Benchmark` label and, rather than simply executing them, it prepares a controlled environment to isolate their execution.

This is an example of benchmark from the RxJava project where the throughput is observed:

```java
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class OperatorFlatMapPerf {

    @State(Scope.Thread)
    public static class Input extends InputWithIncrementingInteger {

        @Param({ "1", "1000", "1000000" })
        public int size;

        @Override
        public int getSize() {
            return size;
        }

    }

    @Benchmark
    public void flatMapIntPassthruSync(Input input) {
        input.flowable.flatMap(new Function<Integer, Publisher<Integer>>() {
            @Override
            public Publisher<Integer> apply(Integer v) {
                return Flowable.just(v);
            }
        }).subscribe(input.newSubscriber());
    }
}
```

2.1 JVM

To understand how JMH works, it is first necessary to define the **Java Virtual Machine (JVM)**. The JVM is the runtime environment in which every Java program executes [Ans25]. After the source code is written in a *.java* file, the *Java compiler* (javac) converts it into bytecode (stored in .class files). When a program starts, the *Class Loader* loads the *.class* files into memory, but the bytecode must still be translated into machine code before the CPU can execute it [Gee25]. To execute the bytecode, the JVM uses the *Execution Engine*, that is classified in different parts:

- **Interpreter**: it reads the bytecode line by line, translating and executing it on the fly. However, if the same instructions are repeated many times, the interpreter must analyze them individually, which significantly slows down the execution [Gee25].
- **JIT (Just-In-Time) Compiler**: while the interpreter is running, the JVM identifies frequently executed code fragments. The JIT compiler optimizes this code and converts it into machine code. Consequently, the next time that code is called, the JVM skips the interpreter and executes the ready to use machine code directly [Ora].

While these mechanisms provide major performance benefits, they introduce non determinism due to the dynamic nature of JVM optimizations. Indeed, at the beginning the JIT compiler performs many optimizations, until performance stabilizes.

To mitigate these issues, JMH organizes measurements into a structured **sequence of stages**, ensuring that performance data is collected only when the system has reached a steady state.

2.2 JMH workflow

The process begins with **forking**, which involves launching independent instances of the JVM as separate OS processes. This isolation ensures that different executions do not influence each other. JMH distinguishes between two types of forks [Opeb]:

- **Warmup forks**: preliminary JVM instances dedicated exclusively to ensure that the environment is stable before the actual testing begins. They are configured via the `warmups` parameter in the `@Fork` annotation and their results are entirely discarded.
- **Measurement forks**: JVM instances where the actual benchmarking occurs.

Within each fork, two types of iterations are executed [Laa+20]:

- **Warmup iterations**: these are used to bring the specific JVM instance to a steady state. During these iterations, the code is executed repeatedly to trigger optimizations, but the performance results are not included in the final statistics.
- **Measurement iterations**: these follow the warmup iterations exclusively within a measurement fork. Since the JVM has now reached a stable state of optimization, the data collected during these iterations represents the true performance of the benchmark.

At the conclusion of the process, JMH aggregates the results from all measurement iterations across all measurement forks. To provide a summary, the tool outputs three key statistical metrics [Obr24]:

- **Score**: the arithmetic mean of all measurement results, which represents the average time taken per operation (in `Average time` mode).
- **Error**: margin of error that shows the variability in the benchmark results.
- **Units**: the units of the score.

JMH allows the selection of different **Benchmark modes**, each focusing on a specific performance metric depending on the goal of the analysis. The most common modes are [Obr24] [Opeb]:

- **Throughput**: measures the number of times the `@Benchmark` method is invoked per unit of time, by continuously calling it during that time interval.
- **Average time**: measures the average execution time of a single invocation of the `@Benchmark` method.
- **Sample time**: it collects samples of the execution time of individual invocations of the `@Benchmark` method, allowing to infer the distributions, percentiles, etc...

- **Single shot time**: measures the execution time of a single invocation of the `@Benchmark` method. This is used to analyze cold start performance before the JIT compiler has applied any optimizations.

JMH provides many additional parameters for customize benchmarks, including `@State` for data scopes, `@Setup` for environment preparation and multi-threading configurations. These features are documented in detail within the official OpenJDK JMH repository. In the following chapters, components will be introduced and explained only as they become relevant to the implementation of this project.

2.3 LLMs

Large Language Models (LLMs) are advanced artificial intelligence systems designed to process, understand and generate natural language text [Str21]. In the context of this thesis, LLMs serve as the generative engine for creating Java benchmarks, using their ability to map natural language requirements into structured code.

LLMs operate as statistical prediction engines [Str21]. During training, the model performs a classification task by calculating the probability of the next unit of text, called token, within a given textual sequence. This process utilizes **Self-supervised learning**, where the subsequent word in a pre-existing text acts as the ground truth. By iteratively predicting these tokens and comparing them to the original source, the model identifies the underlying patterns and structures of both natural language and programming code [Stö23].

Once a token is predicted, it is appended to the existing sequence, and the model repeats the process to determine the following unit. This iterative cycle is what allows the LLM to generate continuous blocks of text or code, which is why it is categorized as **Generative AI** [Stö23].

To process and generate these sequences effectively, the architecture relies on two critical mechanisms:

- **Tokenization**: before any processing, the input text is decomposed into smaller units called tokens. These are not always whole words, but they can also be sequence of characters. This granularity allows the model to handle rare terms [Str21].

- **Attention mechanism**: LLMs are built on a type of neural network architecture called *Transformers* [Vas+17]. The primary innovation of the transformer architecture is the **Self Attention** mechanism. This allows the model to assign weights to different tokens within a sequence, regardless of their physical distance from one another [Str21]. This allows the model to focus on the relevant context just as a human would do.

2.4 AI Agents

An AI agent is an autonomous system designed to execute tasks by independently managing its own workflows through the use of specific *tools*. These agents own reasoning, planning and memory to navigate decision-making processes and adapt to new information. Beyond text generation, AI agents are characterized by their ability to interact with external environments, solve multi-step problems and perform actions to achieve specific objectives [Clo25].

Although AI agents are powered by LLMs, they significantly expand their capabilities by

using tool calling to surpass LLM's limitations, such as training cutoffs and restricted internal reasoning. This approach enables agents to access real-time data and interact directly with external sources [Gut24a].

The workflow of an AI agent can be divided into three stages [Gut24a]:

- **Goal initialization and Planning**: although autonomous, AI agents operate under goals and rules defined by developers and end-users. Starting from these objectives, the agent acts as a planner, breaking down a high level goal into a structured sequence of tasks. For simpler tasks, this step may be skipped.

- **Reasoning with tools**: standard LLMs are often restricted by knowledge boundaries due to the static nature of their training data. AI agents, however, overcome these limits by employing *Agentic Reasoning*. By autonomously accessing APIs, web searches and external datasets, agents can gather missing information in real time. This functionality facilitates the dynamic reassessment of plans, enabling the agent to refresh its internal knowledge and modify its strategies as needed. Ultimately, the capacity to integrate information from various tools empowers agents to tackle intricate problems that a solitary model would struggle to address.

- **Learning and Iterative Reflection**: to guarantee the precision of responses and their alignment with human requests, agents utilize feedback loops. By engaging in *Human-in-the-loop* interactions or receiving critiques from other agents, the system undergoes a process referred to as *Reflection*, which enables the agent to evaluate its errors and enhance its reasoning. Furthermore, by storing successful strategies and past obstacles in a long-term memory, the agent avoids repeating the same errors and adapts its behavior to evolving contexts.

There exists many architectures to build an AI Agent. The one used in this project is the **ReAct** (reasoning and action) framework [Yao+23]. ReAct, with the usage of prompt engineering, structures an agent's activity into a repetitive, formal pattern of three alternating steps [Ber25]:

1. **Thought**: the *Chain-of-Thought* (CoT) described in the prompt helps the agent to decompose complex goals into smaller and manageable subtasks.

2. **Action**: the agent interacts with external sources by invoking the provided tools and taking actions.

3. **Observation**: following the execution of an action, the model evaluates the obtained output to determine the next logical step: either deliver a final answer, if the goal has been met, or use the new data to inform a subsequent thought process. The observation might also consider prior information, from earlier in the model's context window or from an external memory component.

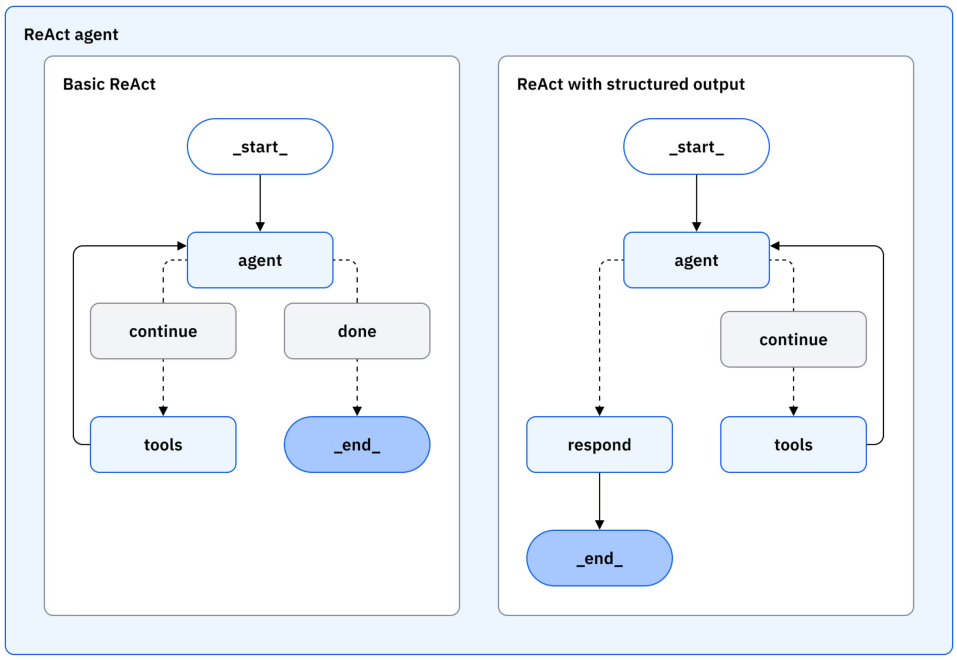


Figure 2.2: ReAct workflow diagram [Source: [Gut25]]

The ReAct framework establishes an iterative **feedback loop**, where the cycle of *thought, action, and observation* repeats until a termination criterion is met. At the conclusion of each iteration, once an action has been executed and its results evaluated, the agent must determine whether to initiate a new reasoning cycle or conclude the task. To prevent infinite execution, termination is typically governed by a maximum iteration threshold or by the fulfillment of a specific condition.

2.5 Multi-Agent Systems

A **Multi-Agent System (MAS)** consists of a network of agents interacting within a shared environment to solve complex tasks through distributed decision-making [Clo]. Although each agent pursue local goals, their collective interaction enables the system to achieve global objectives that would be beyond the capacity of a single model [Gut24b]. In this setup, static tool use is replaced by a collaborative approach where specialized agents share their reasoning and adapt their behavior based on collective feedback. Consequently, MAS architectures offer greater scalability than standalone models by distributing the workload to handle complex tasks [Clo].

An example of a MAS in software development is **ChatDev** [Qia+24], a framework in which specialized agents for the design, coding, and testing phases communicate through a specific pattern.

The system emulates a virtual community in which agents, each designated a particular role, engage with one another and where a chat-chain is employed to segment the workflow into tasks and sub-tasks. For every sub-task, two agents collaborate, exchanging information until they

achieve a consensus and formulate a solution for the corresponding sub-task. Additionally, to mitigate coding hallucinations, a communicative de-hallucination mechanism is implemented, enabling agents to seek additional information prior to providing a response.

Compared to a single-step approach, using a MAS allows for the inclusion of members with diverse expertise who handle more manageable sub-tasks, thereby reducing inconsistencies and inefficiencies, while achieving an outcome of higher quality.

A second example is **AutoSafeCoder** [Nun+24], a multi-agent framework for code generation that also integrates dynamic and static testing. It consists of three agents that collaborate acting as a code generator, a static vulnerability analyzer, and a dynamic tester for identifying runtime errors. In this case, the main advantage of a MAS is its ability to prioritize vulnerability detection, something that an approach with a single LLM would likely fail to do, as it would primarily focus on functional correctness.

3. Related Work

In this chapter, other related works in the field are presented, organized by topic, and their connections to this study are explained.

3.1 Efficiency in Performance Microbenchmarking

In the work done by *Leitner and Bezemer* [LB17] it is highlighted how in performance testing, with respect to functional testing, clear guide lines and best practices are missing. Performance tests are often managed by only a few individuals, are rarely updated and minimal time is spent on their creation and maintenance. To address these issues, the authors emphasize the need for a framework that integrates with standard CI tooling, simplifying both the creation and execution of performance tests. *Laaber and Leitner* explore the same problem in a subsequent paper [LL18], specifically evaluating the suitability of microbenchmark suites for Continuous Integration. They find that these suites, which can take hours or even days to run, often fail to provide the rapid performance feedback required by CI pipelines, making their integration impractical despite their importance for performance monitoring.

To solve this, *Laaber et al.* [Laa+20] proposed the first approach of dynamic reconfiguration, that determines the optimal stopping point for a benchmark execution in real-time. This point is reached as soon as the results are considered sufficiently stable. With this method, execution time shortens by 48.4% to 86.0% while maintaining the quality of the results unchanged for 78.8% to 87.6% of the benchmarks. In a subsequent study [LGL21], *Laaber et al.* faced the same issue with an alternative strategy: Test Case Prioritization (TCP). Although TCP is a widely used technique in functional testing, the authors adapted it for microbenchmarks. With TCP techniques, they were able to identify the three most significant performance changes after executing only 29% to 66% of the whole microbenchmark suit.

In a different approach, *Ding et al.* [DCS20] investigated whether existing functional tests can be reused as performance tests. Specifically, they examined if these tests are sensitive enough to detect code speedup following an optimization. Their results showed that while most performance fixes can indeed be validated using tests already available in the release pipeline, only a small fraction of them are actually effective for this purpose.

This work represents an attempt to create a tool that relieves programmers from the responsibility of creating microbenchmarks. Developers will no longer be required to know the techniques for writing technically correct microbenchmarks and the time needed for their creation will be reduced to zero, as the process will be fully automated. This approach addresses the problem stated in the first two papers presented in this section, however the duration of benchmark execution and the reuse of functional tests are not explored in this study.

3.2 Test Generation

Writing tests, whether for performance or functionality, has always been a time consuming manual task. Over time, various studies have attempted to automate this process. The following researches specifically focus on the automatic generation of unit tests.

The first study [FA13] introduces *Whole Test Suite Generation*. Instead of creating separate test cases for each specific goal, this approach focuses on test suites targeting an entire coverage criterion, making the process much more efficient. The second study [FA11] presents **EvoSuite**, the tool that applies this theory to Java programs. In addition to generating tests, EvoSuite automatically creates assertions by using mutation testing to identify which checks are truly effective at catching bugs, filtering out any unnecessary ones.

This study adopts the same approach but applies it in an orthogonal direction: rather than automating the generation of functional tests, which verify the functional correctness of a program, it aims to generate microbenchmarks, that instead evaluate the performance of small code fragments.

3.3 Benchmark Generation

Writing effective benchmarks is often a complex task. The reason is that developers may not be fully aware of the rigorous rules required to create correct and meaningful benchmarks, such as underestimating the impact of JIT compiler optimizations, thereby producing misleading results. The most common pitfalls are described by *Costa et al.* [Cos+19], who identified the 5 most common bad practices in JMH found in open source software. To address these challenges, several tools have been developed to automate the benchmarking process.

Among these, **AutoJMH** [RCB16] is a tool designed to automatically generate microbenchmark payloads for Java applications. It uses static slicing to extract the Segment Under Analysis (SUA) and all its dependencies into a compilable payload. However, AutoJMH presents some limitations, such as the requirement of at least one existing test case to generate the necessary initialization values as well as the fact that some compiler optimizations are still not correctly handled by the tool. Based on the idea of using existing tests, **ju2jmh** [Jan+23] is a framework that creates JMH microbenchmarks from existing JUnit tests in an automate way. The obtained results show that benchmarks generated with this framework outperform both JUnit tests and AutoJMH generated benchmarks, while remaining comparable to manually written benchmarks by developers. However this approach does not handle dead code elimination or constant folding and it was only evaluated on a limited number of open-source projects. Another approach was the one introduced by *Kuperberg et al.* [KOR10], who developed an automated tool for benchmarking large Java APIs. Unlike other approaches, this solution encourages the optimizations of the JIT compiler to ensure the benchmarking results are realistic.

The research conducted by *Toffola et al.* [TPG18] shifts the focus from benchmark generation to the identification of performance bottlenecks within a program. **PerfSyn** is a system that automatically generates test programs to identify performance bottlenecks in Java code. Its goal is to find the slowest execution path of a method by constantly modifying the tests until it discovers the specific data that makes the code run as slowly as possible. However, the tool has some limitations: it struggles with complex text inputs, it cannot handle objects that are too difficult to initialize and it can be inaccurate when testing very short methods.

Most of these works share the same goal as this research, namely the automatic generation of benchmarks. The approach proposed here avoids all five bad practices identified by *Costa et al.* in the generated benchmarks. Unlike AutoJMH and ju2jmh, it does not require to have an existing test case, as benchmarks are generated solely from the source code. A remaining limitation, also present in ju2jmh, is that the system has been evaluated on a limited number of open-source projects and therefore does not have universal validity. The system adopts the approach of the first papers cited regarding avoiding JIT compiler optimizations, departing from the one adopted by *Kuperberg et al.*. Bottleneck detection is also addressed, but in a different manner compared to PerfSyn: performance-critical code sections are not identified through test programs, they are instead detected by the Planner Agent, utilizing the tools and reasoning capabilities at its disposal.

3.4 LLMs for testing

The capabilities of LLMs can be used for test generation, as these models have become significantly more powerful and efficient in recent years. The following works employ LLMs in this manner, applying them to achieve efficient testing in different fields.

ChatUniTest [Che+24] is a framework based on LLM designed to automatically generate unit tests. To address the typical challenge of LLMs' limited context window, the system includes only the most important information of a method within the prompt. Furthermore, each generated test is passed to a validation component: if the validation fails, a repair strategy is initiated to correct the test. The results show that ChatUniTest obtained a line coverage of 59.6%. However, the authors specify that the selection of context included in the prompt can be improved and that a Chain-of-Thought mechanism is not yet implemented. The same strategy is adopted in **ChatPerfTest** [Zhi+25] for the creation of JMH microbenchmarks. Due to the specific nature of this field, it introduces an additional feature to manage common microbenchmarking bad practices, ensuring that they are avoided. An approach similar to that of ChatUniTest is adopted by **CoverUp** [PB25], a system that automatically generates regression tests by using the combination of code coverage analysis and large language models. Extending the application of LLM-based testing to a different field, a GUI testing tool was developed. **DROIDAGENT** [YFY24] is indeed a multi-agent system that tests Android GUIs by automatically creating and testing realistic tasks in plain language.

Most of these works inspired this research by showing how LLMs can automate test generation. While ChatUniTest focuses on unit tests, this study applies similar LLM techniques to microbenchmarks. To handle the limited context windows of LLMs, analogous summarization methods are used, along with a validation and repair step. The system is also similar to ChatPerftest as they share the same goals and both focus on avoiding bad practices. However, it distinguishes itself by providing specific agents with tools that enable autonomous operation, a characteristic also central in DROIDAGENT's multi-agent architecture.

4. Approach

This chapter presents the methodology adopted for the project's development, alongside the system design. It details the pipeline steps and the operation of the agents, both with and without tool usage. Furthermore, the system architecture is explained, including the specific roles of each agent, the prompts and tools employed, and examples of the outputs produced at each stage, starting from the selection of target methods to a fully generated system microbenchmark.

4.1 Methodology

This section presents the methodology employed in this study, which mainly consists of two phases: generating microbenchmarks through the deployed system and evaluating the generated microbenchmarks using a static and a dynamic analysis. To rigorously describe the creation and validation of the system, the research workflow was framed within the **Design Science Research Methodology** [Pef+07] , as detailed below.

The process initiated with problem identification based on current literature, which highlighted a need for an automated tool capable of generating JMH microbenchmarks. As discussed in Section 1.1, writing microbenchmarks manually is time consuming and error prone. Although some automated solutions have been proposed, they exhibit limitations and don't exploit multi-agent cooperation, an approach that has recently demonstrated significant potential across various domains. To address these limitations, the primary objective of this work was to construct a system capable of autonomously generating microbenchmarks for Java projects by using a pipeline of LLM-based agents.

Instead of generating everything in a single step, the system uses a multi-agent pipeline. Breaking the process into smaller and well defined steps helps the LLMs focus better, which improves their accuracy. Specifically, the pipeline follows these main stages: selecting the target methods, generating the benchmark code and configuration, verifying compilation and execution, checking the benchmark's quality, and diagnosing any errors. In addition, feedback loops allow the system to automatically recover from failures and retry generation. During development, the system was tested and refined on the RxJava project [Mic] to fix early bugs and validate the overall architecture.

The system was then evaluated on different Java projects. We measured the number of successfully generated benchmarks, the time and cost required, and the overall quality of the generated code. In this last phase, the goal was to check if the benchmarks were actually useful and functional in practice. To do this, we used two main approaches: static and dynamic analysis. First, we manually inspected a sample of benchmarks to check code quality, such

as the presence of the target method and proper use of polymorphism. Second, we tested if the benchmarks could detect real performance slowdown. Using AspectJ, we artificially added delay before each method call and verified whether the benchmarks detected a significant slowdown.

4.2 System Design

This section presents the pipeline structure and the roles of each step at a high level. It also explains the operation of the agents, both with and without tools.

4.2.1 Pipeline

The system is designed as the pipeline shown in Figure 4.1, composed of agents and LLM calls that automatically generate benchmarks for Java code, without any user intervention. The source code is analyzed to identify the most critical performance areas, and within these areas, the methods on which to create the benchmarks are selected. For each selected method, a generation loop is executed for the corresponding benchmark over multiple iterations. The initial one executes the primary generation attempt: the body of the benchmark is generated first and only afterwards the configuration instructions are added. The obtained benchmark is subjected to compilation, execution, and validation checks. If even a single one of these verifications fails, the benchmark is considered momentarily not ready to be added to the final result, and the generation process proceeds with the second attempt, in which the benchmark is modified so that, hopefully, the checks will be passed this time. The process continues until the benchmark is generated correctly, passing all controls, or until the maximum number of allowable attempts is reached, beyond which the generation of the benchmark is considered failed.

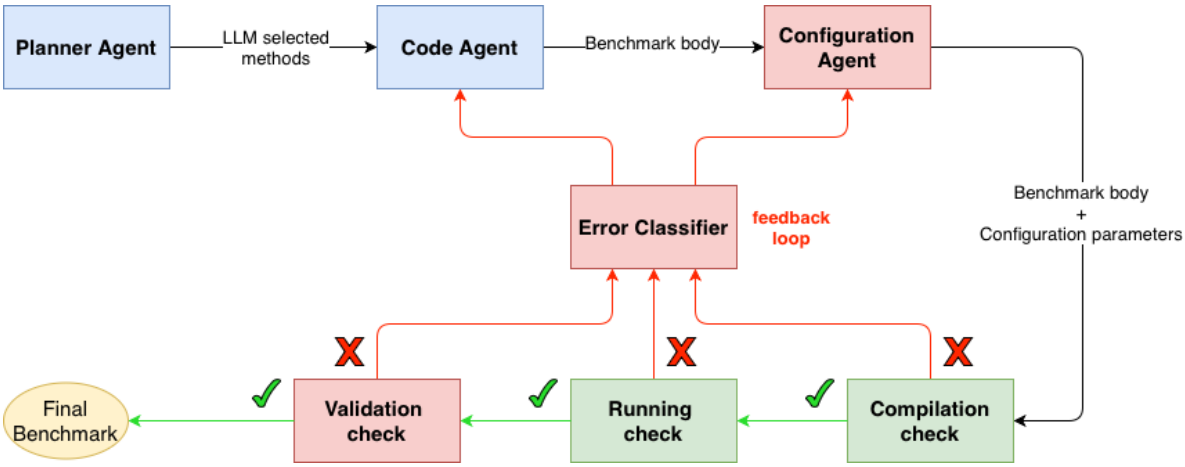


Figure 4.1: Pipeline of the system. The blue steps represent agents that use tools with reasoning capabilities (ReAct), the red steps represent agents that use simple LLMs calls and the green step are deterministic logic.

What makes the system effective and allows it to significantly improve performance is the use of **feedback loops**. When an error is detected during compilation, running, or validation checks, the feedback loop allows the workflow to loop back to a previous stage in the pipeline.

Specifically, the generated error message is passed to the *Error Classifier*, which is responsible for identifying the type of error and directing the workflow, or to the *Code Agent* if the error concerns the benchmark body, or to the *Configuration Agent* if it concerns configuration parameters. This allows the pipeline to be error-resistant and to recover from cases that failed the first time. To avoid infinite loops, a variable `MAX_ATTEMPS` = 5 was set. The value 5 was defined by observing that recovery of a benchmark will most likely occur within the first 5 attempts, since it is very unlikely that if it hasn't occurred within this limit, it will occur in subsequent iterations.

4.2.2 Agents and tools

This subsection provides a detailed explanation of the agents' roles within the pipeline.

Figure 4.2 shows the two types of agents: *standard agents*, which interact with the LLM by simply calling it, and *autonomous agents*, which are equipped with specific tools to perform independent actions. As for the former, it's sufficient to know that they make calls to the LLM with a specific prompt, receive the response and interpret it accordingly to determine the next step. The latter, however, are more complex, and their logic is explained below.

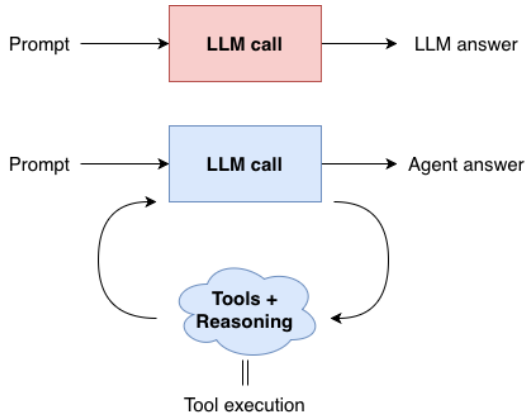


Figure 4.2: Agent types

The agents that are equipped with tools are the Planner Agent and the Code Agent. The reason these two steps were not implemented with simple LLM calls is that the result would have been much less accurate and more difficult to obtain. In the case of the *Planner Agent*, the goal of this project was also to delegate the selection of methods on which to create benchmarks to the LLMs. To make an informed choice, the LLM requires a sufficiently broad context, coinciding with the entire codebase. Since the entire source code doesn't fit into the relatively narrow context window of the LLMs, it was necessary to provide tools so that the agent could navigate the Java code autonomously. As for the *Code Agent*, it was necessary to use tools because, in this way, the agent had full awareness of the source code and was less inclined to invent classes or hallucinate imports.

4.2.3 System architecture

In the following, each step of the pipeline of Figure 4.1 is described in detail, including the specific roles of each agent, the prompts and tools employed, and examples of the outputs produced at each stage, starting from the selection of target methods to a fully generated system microbenchmark.

Planner Agent

The Planner Agent is responsible for conducting an Agentic Search within the project's source code directory to identify the most suitable methods for generating microbenchmarks. It is divided into two phases: the Exploring phase and the Selection phase.

In the *Exploring phase*, the agent is asked to find the top 10 performance hotspots areas within the codebase.

- **Prompt:** The initial prompt instructs the agent to focus on areas exhibiting: Core computational logic (e.g., algorithms and intensive data processing), High-traffic infrastructure (e.g., network, I/O, and buffer management), Complex state management (e.g., concurrency and custom collections). To prevent bias, the prompt guides the agent to ignore any existing benchmark directories within the project. Furthermore, it instructs the agent to avoid internal engine components that are too difficult to benchmark in isolation. The output is strictly constrained to a single JSON object.

- **Tool usage:** The agent navigates using a set of specialized tools in a ReAct loop:

  `list_module_layout` Lists the top-level module structure of the given directory, helping to understand how the repository is organized.

  `list_files` Navigates through the directory structure to obtain the list of files for the specified target packages.

  `list_java_symbols` Examines the specified Java files to identify declared classes, interfaces, enums and method signatures, assisting the agent in comprehending the fundamental responsibilities without the need to read the entire files.

  `read_file` Reads the full content of selected source files when deep code analysis is required.

Once the agent gathers sufficient context, it completes the phase by returning a structured JSON object containing the identified hotspots along with the underlying justification, as shown in Listing 1.

```
{
  "hotspots": [
    {
      "path": "t-digest-t-digest-3.2/src/main/.../stats/MergingDigest.java",
      "reason": "Primary high-level t-digest implementation; maintains ..."
    },
    {
      "path": "t-digest-t-digest-3.2/src/main/.../stats/Centroid.java",
      "reason": "Represents the fundamental statistical aggregation unit..."
    }
  ]
}
```

Listing 1: JSON output returned by the Planner Agent at the end of the Exploring phase.

After the most relevant areas are identified, the process moves to the *Selection phase*, where the agent is tasked with diving deep into the previously discovered hotspots and with selecting the most performance-critical methods for JMH benchmarking.

- **Prompt:** The previously identified hotspot areas are included in the prompt, which requests a deeper analysis of these areas to determine the specific methods deemed most essential for performance, indicating that 5 to 10 methods should be chosen for each hotspot. To guide this selection, the prompt configures the agent with explicit criteria, such as focusing exclusively on public methods, prioritizing routines that perform actual computation, data transformation, or scheduling logic, ensuring balanced coverage across algorithmic complexity, concurrency and data management, and rejecting internal utilities as well as low-level atomic management routines.

- **Tool usage:** The agent iterates through the candidate hotspot areas using tools in a ReAct feedback loop:

  `list_files` Finds and confirms the physical location of target `.java` files across all hotspot directories.

  `list_java_symbols` Scans the candidate source files to retrieve all declared class and method signatures, to filter out private or internal API candidates.

  `read_file` Reads the entire source files to access the full implementation of candidate methods and confirm their selection.

Once the agent gathers sufficient information, it completes the phase by returning a JSON array containing the selected methods along with their class names, file paths and the reasons why they were chosen, as shown in Listing 2. The result therefore contains the most performance-relevant methods, for which benchmarks will subsequently be generated.

```
[
  {
    "class_name": "TDigest",
    "method_name": "createMergingDigest(double)",
    "file_path": "t-digest-t-digest-3.2/src/main/.../math/stats/TDigest.java",
    "reason": "Primary public factory for the recommended digest implementation;
    ↪ benchmarking through this entry point captures realistic end-to-end
    ↪ construction workflows."
  },
  {
    "class_name": "TDigest",
    "method_name": "createAvlTreeDigest(double)",
    "file_path": "t-digest-t-digest-3.2/src/main/.../math/stats/TDigest.java",
    "reason": "Public factory for the tree-backed alternative implementation;
    ↪ useful for comparing the two main digest strategies under identical
    ↪ benchmark setup."
  }
]
```

Listing 2: JSON output returned by the Planner Agent at the end of the Selection phase.

Dividing this step into two phases overcomes the agent's tendency to focus exclusively on the first identified hotspot. This approach ensures a uniform consideration of all areas, which would have been compromised by a single-request structure.

Code Agent

This is the first step of the loop, which continues until a final and valid benchmark is obtained or the maximum number of attempts is reached. It represents the initial stage of the *Generation phase*, focused on constructing the complete benchmark code; the second part of this phase focuses on parameter configuration and is described later. At this point, only the body of the benchmark is generated, excluding parameter configurations (such as `@Param`, `@Warmup`, `@Fork`, etc.). If a previous iteration failed due to a compilation, runtime, or quality error labeled as a `CODE_ERROR`, the error log is provided back to the LLM to guide code refinement and regenerate the benchmark body. Due to the LLMs' limited context window, only the specific lines of the method are included in the prompt. Furthermore, tools are provided to the agent to localize and read the method within the source code, thereby minimizing errors.

- **Prompt:** The prompt includes the source code for the target method and tells the CodeAgent to create or fix only the microbenchmark's body without changing any configuration annotations. Depending on what is needed, the agent creates an initial version using JMH annotations such as `@Benchmark`, `@State`, and `@Setup` or fixes errors from past attempts while keeping the configuration annotations that are already present. As a result, the output consists strictly of the benchmark body during initial creation, as shown in Listing 3, but it keeps the configuration annotations previously added by the ConfigAgent when invoked for corrections.

  To make sure the agent can access protected and package-private methods, the prompt requires placing the benchmark class in the target's exact package. It also prohibits the use of dependencies on `.internal`, `.impl` or `.test` packages, which are often restricted. In addition, it follows JMH practices to prevent the JIT compiler from altering results: it makes sure to use `Blackhole.consume()` for any returned values or changed object states to avoid Dead Code Elimination, it requires input values to be stored in non-final state fields to prevent Constant Folding, it forces explicit terminal execution of lazy or asynchronous types and it requires parameterized inputs using `@Param`. The final output is strictly constrained to pure, compilable Java source code.

- **Tool usage:** The agent uses the following tools in a ReAct feedback loop:

`find_definition`	Finds the exact method position within its class.
`read_file`	Reads the entire source file to access the complete implementation of the method and inspect its exact signature, parameters, required imports, etc.

```java
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.Level;
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.Setup;
import org.openjdk.jmh.annotations.State;

public class MergingDigestCompress_NoArgs {
    @State(Scope.Thread)
    public static class BenchmarkState {
        @Param({"100", "1000"})
        public int itemCount;

        @Param({"50.0", "200.0"})
        public double compression;

        public double[] values;
        public com.tdunning.math.stats.MergingDigest digest;
        public int witness;

        @Setup(Level.Trial)
        public void setupValues() {
            values = new double[itemCount];
            for (int i = 0; i < itemCount; i++) {
                values[i] = ((i * 17) % 997) + ((i % 13) * 0.01d);
            }
        }

        @Setup(Level.Invocation)
        public void setupDigest() {
            digest = new com.tdunning.math.stats.MergingDigest(compression);
            for (int i = 0; i < itemCount; i++) {
                digest.add(values[i]);
            }
        }
    }

    @Benchmark
    public void compress(BenchmarkState state, org.openjdk.jmh.infra.Blackhole
    ↪ blackhole) {
        state.digest.compress();
        state.witness = state.digest.centroidCount();
        blackhole.consume(state.witness);
    }

```

Listing 3: The output returned by the CodeAgent during the initial creation of the microbenchmark for the `MergingDigest.compress()` method, which contains only the benchmark body. If the agent is used to fix an existing benchmark, the output also preserves the parameter configurations previously added by the ConfigAgent

Configuration Agent

This constitutes the second stage of the Generation phase, dedicated to integrating or correcting JMH configuration annotations into the body of the benchmark produced by the CodeAgent. Depending on the execution context, the prompt either instructs the agent to add these configuration parameters for the first time or, if a prior iteration fails due to a `CONFIG_ERROR`, to modify and correct the existing annotations based on the error log. Listing 4 highlights the configuration changes added to the output of the Code Agent shown in Listing 3.

- **Prompt:** In the prompt the agent is told to add performance parameters, such as a minimum of three forks to ensure JVM stability, at least five warmup and measurement iterations lasting two seconds each, and appropriate benchmark modes like throughput or average time. It strictly instructs the agent to avoid any modification to the core benchmark logic, class structure, or setup routines, as these tasks fall under the responsibility of the CodeAgent. Furthermore, it mandates the inclusion of annotation imports and returning a fully compilable Java file.

```
1
2  package com.tdunning.math.stats;
3
4  import java.util.concurrent.TimeUnit;
5  import org.openjdk.jmh.annotations.Benchmark;
6  import org.openjdk.jmh.annotations.BenchmarkMode;
7  import org.openjdk.jmh.annotations.Fork;
8  import org.openjdk.jmh.annotations.Level;
9  import org.openjdk.jmh.annotations.Measurement;
10 import org.openjdk.jmh.annotations.Mode;
11 import org.openjdk.jmh.annotations.OutputTimeUnit;
12 import org.openjdk.jmh.annotations.Param;
13 import org.openjdk.jmh.annotations.Scope;
14 import org.openjdk.jmh.annotations.Setup;
15 import org.openjdk.jmh.annotations.State;
16 import org.openjdk.jmh.annotations.Threads;
17 import org.openjdk.jmh.annotations.Warmup;
18
19 @BenchmarkMode(Mode.Throughput)
20 @OutputTimeUnit(TimeUnit.SECONDS)
21 @Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
22 @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
23 @Fork(3)
24 @Threads(1)
25 public class MergingDigestCompress_NoArgs {
26     ...
27 }
```

Listing 4: Configuration instructions added from the Configuration agent to the output of the Code agent.

Compilation check

This is the first of three checks that every generated benchmark must pass to be included in the final output. At this stage, the benchmark is compiled: if the process succeeds, the pipeline proceeds to the next step, otherwise, the compilation error is forwarded to the *Error Classifier*, which categorizes it accordingly. Details regarding how the compilation check is implemented can be found in Section 5.1

Running check

This phase handles benchmark packaging and execution. It begins by packaging the project into an executable JAR, followed by the actual benchmark run. As in other checks, in the event of a build failure or a runtime exception, the system redirects the output to the *Error Classifier* to identify the cause. Details regarding how the running check is implemented can be found in Section 5.1.

Validation check

The Validation check is the final stage of the process, performed only after the benchmark has successfully passed the compilation and execution phases. The objective of this step is to check whether the benchmark is free of typical JMH bad practices and is designed in such a way that the method under test is invoked in a meaningful fashion, considering concurrency, workload size and proper parameter setup. This is done using an LLM call with a proper prompt. In case of failed validation check, this stage produces critical flaws and warnings, and suggests how to improve the microbenchmark, as depicted in Listing 5. Such feedback is received by the *Error Classifier* and classified.

- **Prompt:** The microbenchmark code to be evaluated is included in the prompt so that it can be verified whether it meets certain quality criteria. These include preventing dead code elimination via Blackhole consumption, avoiding constant folding, preventing loop optimizations, managing `@Setup` correctly, and checking for an adequate number of forks, warmup and measurement iterations. Furthermore, the prompt assesses whether a representative level of parallelism and a realistic workload size are properly implemented.

```
{
    {
      "is_meaningful": false,
      "critical_flaws": [
        "Uses @Setup(Level.Invocation) to construct and populate the digest
        ↪  before every benchmark invocation; ..."
      ],
      "warnings": [],
      "suggestions": "Prefer moving object construction into the @Benchmark
      ↪  method if you want fresh-object compress cost, or use
      ↪  Level.Iteration/trial setup ..."
    }
}
```

Listing 5: Output of the Validation check in case of a negative evaluation.

Error Classifier

The Error Classifier manages the feedback loop within the pipeline. Its role is to label the generated error when one of the three checks fails, classifying it as either `CODE_ERROR`, `CONFIG_ERROR`, or `UNKNOWN_ERROR`. Based on that, it redirects the workflow to a previous step of the pipeline so that the generation of that specific benchmark can recover from the failure.

- **Prompt:** The error log is used in the prompt so that the LLM can classify it as `CODE_ERROR` if it is related to the benchmark logic, `CONFIG_ERROR` if it is caused by JMH configuration annotations, or `UNKNOWN_ERROR` in all other cases. It is also required that the output be exclusively the string identifying the error category, without any additional content.

Depending on the label assigned to the specific error, the workflow can proceed in the following ways:

- **`CODE_ERROR`:** The workflow is sent back only to the *Code Agent* so that the benchmark's body can be modified.
- **`CONFIG_ERROR`:** The workflow is sent back only to the *Configuration Agent* so that the benchmark's configuration instructions can be modified.
- **`UNKNOWN_ERROR`:** The workflow is sent back to both the *Code Agent* and *Configuration Agent*, so that everything can be modified and corrected.

5. Implementation

The following diagram shows the folder structure of the project:

```
MICROBENCHMARKS/
├── agents/
│   ├── brainstorming.py
│   ├── code.py
│   ├── config.py
│   ├── errorClassifier.py
│   ├── planning.py
│   └── verifier.py
├── Generated_Benchmarks/
│   ├── pom.xml
│   └── ...
├── logs/
├── source_project/
│   ├── pom.xml
│   └── ...
├── tools/
│   ├── agentic_search/
│   │   ├── LangGraph.py
│   │   ├── CodeExplorer.py
│   │   ├── planning_graph.py
│   │   └── ...
│   ├── compilerTool.py
│   ├── runnerTool.py
│   └── writerTool.py
├── llm.py
├── pipeline.py
├── pom.xml
└── README.md
```

`agents/` Contains the implementation of all agents and also includes `brainstorming.py`, which manages the workflow in case of errors.

`Generated_Benchmarks/` The directory where the generated benchmarks are stored.

`logs/` The directory used for storing runtime logs.

`source_project/` Contains the input project for benchmark generation.

`tools/` Contains the utility tools used by different pipeline stages, such as scripts for compilation and running checks.

`agentic_search/` Contains the ReAct framework implementation and the set of tools provided to the agents.

`llm.py` The file that handles the interaction with the LLM, managing API requests and response.

`pipeline.py` Serves as the main entry point of the project, orchestrating the interaction between agents and managing the benchmark generation loop.

`pom.xml` The Maven configuration file found in the root directory, as well as in `source_project/` and `Generated_Benchmarks/`. It is responsible for dependency management, project structure and build configurations.

The file `pipeline.py` contains the code underlying the system's operation and the interaction among the pipeline steps. The process starts with the *Planning Agent*, which selects the methods for which benchmarks are to be generated and returns their information, namely the class name, the method name, the associated file path, and the reason for its selection. Next, each method goes through the five microbenchmark generation steps: Code generation, Configuration generation, Compilation check, Execution check and Validation check. If all checks are passed, the microbenchmark is saved in the `Generated_Benchmarks/` folder, otherwise, a further generation attempt is undertaken, hoping to fix the benchmark and successfully pass the checks next time. The ability to carry out multiple generation attempts is made possible thanks to feedback loops, which route the workflow back to a previous pipeline step. To prevent infinite generation loops, the maximum number of attempts is limited by a threshold, beyond which the benchmark generation is considered a failure.

When an error is detected, it is categorized by the *Error Classifier*, and the `brainstorming.py` code handles routing it back to the corresponding previous step in the pipeline.

Depending on the error type, the system may choose to regenerate only the benchmark body, only the configuration, or both. In each scenario, the specific error message is integrated into the LLM prompt to enable effective error resolution.

5.1 Compilation and Runtime Environment

The compilation check within the pipeline uses Apache Maven, a build tool for Java projects. This step is managed through the Python `subprocess` library, which enables the direct execution of shell commands. Specifically, the following code executes the command inside the `Generated_Benchmarks/` directory and isolates the process to the specific benchmark under analysis. This ensures that the compilation remains unaffected by other files in the directory.

```
result = subprocess.run(
            [
                "mvn",
                f"-Dtest={full_package}.{benchmark_class_name}",
                "compile"
            ],
            cwd=os.path.join(self.project_src_path, "Generated_Benchmarks"),
            capture_output=True,
            text=True
        )
```

To perform the execution check, an executable .jar file is generated and subsequently executed. As with the previous stages, the `subprocess` library is used to manage both of these processes.

The following code snippet illustrates the process of generating the executable JAR file.

```
result = subprocess.run(
            ["mvn", "clean", "package"],
            cwd=benchmark_dir,
            capture_output=True,
            text=True,
            timeout=600
        )
```

The execution must be as fast as possible, otherwise, since this process is repeated one or more times for each generated benchmark, the overall system runtime would become impractically long. To achieve this, the execution parameters are set to their minimum possible values. This approach does not affect the outcome of the run (specifically, whether it executes successfully or triggers an error), which is the primary focus of this stage, but only impacts the precision of the benchmark's performance metrics, which are not relevant at this point in the pipeline.

```
command = ["java", "-jar", jar_path]
command.append(file_name)
command.extend([
    "-f", "1",      # 1 fork
    "-t", "1",      # 1 thread
    "-wi", "1",     # 1 warmup iteration
    "-i", "1",      # 1 measurement iteration
    "-r", "1s",     # each measurement iteration lasts 1 second
    "-w", "1s"      # each warmup iteration lasts 1 second
])

result = subprocess.run(
                command,
                cwd=benchmark_dir,
                capture_output=True,
                text=True,
                timeout=120
            )
```

5.2 Frameworks

In this section, the main frameworks used in the system will be presented: ReAct, LangChain, and LangGraph.

5.2.1 ReAct

The agent's architecture incorporates the **ReAct** paradigm [Yao+23] to implement an *Agentic Search* workflow. This integration allows the agent to autonomously explore the codebase by alternating between reasoning steps and tool execution.

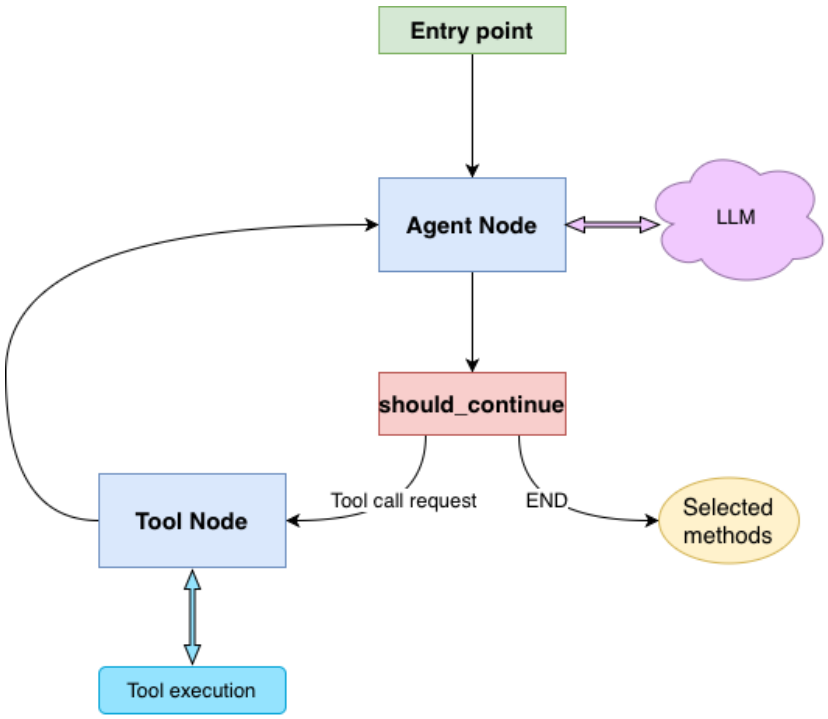


Figure 5.1: ReAct Paradigm

This process enables the agent to decompose a goal into smaller subtasks, invoke specific tools to complete them and iteratively decide whether to perform further actions or conclude the process. The frameworks used to implement this are LangChain and LangGraph.

The agent's reasoning capability is implemented through the iterative loop of Figure 5.1, where reasoning and execution alternate. The agent starts from the initial prompt and, judging it has not yet reached the final answer as the reasoning process has just begun, requests a tool call. Each tool corresponds to a specific action, which is executed upon request and whose result is returned to the agent. At this point, the agent analyzes the information gathered up to that moment and, if it believes it has obtained the final answer, breaks the loop and returns the result, otherwise, it requests another tool call. To maintain the history of the information gathered from the tool calls, a persistent **State** is used, initialized with the user prompt and to which all tool requests and their corresponding responses are appended.

Regarding the responsibilities of each framework, LangChain handles the definition of the tools and the LLM, while LangGraph manages the graph and the persistent State. More specifically, the logic adopted is as follows:

1. **Agent Node**: The Agent Node reads the current State at the beginning of each iteration and it evaluates the dialogue history to decide if the LLM can provide a final answer or if it requires external tools. If a tool is required, the state is updated with the tool request (e.g., State{initial_prompt, tool_call_1}).

2. **Conditional Routing (should_continue)**: This component directs the workflow based on the Agent Node's output. It evaluates two main scenarios:

   - *END*: If the LLM has formulated a final response, the loop terminates and the answer is returned to the user.

   - *Tool Call Request*: If a tool request is detected, the workflow transitions to the Tool Node.

3. **Tool Node**: This node executes the specific function requested by the LLM. Once the execution is complete, it updates the State with the result (e.g., State{initial_prompt, tool_call_1, tool_result_1}) and loops back to the Agent Node.

A 50-iteration limit is imposed to prevent potential infinite loops.

5.2.2 LangChain - LangGraph

LangChain and LangGraph are frameworks designed to facilitate the development of autonomous agents capable of tool invocation and complex reasoning.

Specifically, in the file `planning_graph.py`, **LangGraph** is used to manage the persistent state across the nodes and edges of the graph. The graph consists of an Agent Node and a Tool Node that are mutually interconnected, and a conditional edge acting as the fork where the agent must choose whether to continue the search or terminate with an answer.

Initially, the tools are bound to the LLM, which can then request them as needed. These tools are defined in `LangGraph.py` using **LangChain's** `@tool` decorator. This allows the system to transform standard Python functions into structured tools that the agent can understand and call during the execution of the graph.

```
code_explorer = CodeExplorer(project_root=".")

@tool
def read_file(file_path: str) -> str:
    """
    Reads the content of a specific file within the project.

    Args:
        file_path: The file path relative to project root (e.g.,
        ↪  "src/main/java/MyClass.java")

    Returns:
        The file content as a string, or an error message if the file is not
        ↪  found or outside project directory.
    """
    return code_explorer.read_file(file_path)
```

A key part of the architecture is that the tool's description is kept separate from its implementation. The LLM cannot see the actual source code, but it can only read the description provided in the comments under the `@tool` label. For this reason, the description must be very precise, clearly explaining the tool's purpose and its required parameters. Only after the LLM decides which tool to use and issues a request, the actual implementation, written in `CodeExplorer.py`, is triggered to execute the task.

```
def read_file(self, file_path):
        """Reads the content of a specific file."""
        self.logger.info(f"--- [CodeExplorer] read_file - Reading file
        ↪  '{file_path}' ---")

        if not self._is_safe_path(file_path):
            return "Error: File path is outside the project directory."

        full_path = os.path.join(self.project_root, file_path)
        try:
            with open(full_path, 'r', encoding='utf-8', errors='ignore') as
            ↪  f:
                return f.read()
        except FileNotFoundError:
            return f"Error: File not found at the resolved path: {full_path}"
        except Exception as e:
            return f"Error reading file {full_path}: {e}"
```

5.2.2.1 Tools

The tools utilized by the agents are listed below, along with a description of how they operate:

- `list_files`: Lists files and directories in a given sub-directory of the project.
- `list_module_layout`: Lists the top-level module structure of the given directory, helping to understand how the repository is organized.
- `read_file` Reads the content of a specific file within the project.
- `find_file_by_name` Finds the first file in the project that matches a specific filename, using case-insensitive matching.
- `list_java_symbols` Scans the candidate source file to retrieve all declared class, method and interface signatures.
- `find_usages` Finds where a symbol is used using fixed string search with `grep -F`. This is faster than regex and finds all occurrences of the symbol name.
- `find_definition` Finds definitions of a class, interface, enum or method using grep. Searches for patterns like "class SymbolName", "interface SymbolName", etc.

5.3 Method Verification - AST

Once the Planning Agent proposes a list of candidate methods for benchmarking, the system initiates a verification phase. The goal of this stage is to eliminate hallucinations, which happen when the LLM's answers contain non-existent methods, by searching for each method selected by the model among those actually existing in the source code.

Initially, both the signatures of the candidate methods returned by the LLM and those from the source code are normalized using the `normalize_java_type` function, which standardizes complex Java types into a format that is easier to compare.

The function `verify_and_format_methods` represents the step that executes the verification. By utilizing `javalang.parse.parse()`, the system generates an **Abstract Syntax Tree (AST)**, a structural representation that allows to navigate the code programmatically rather than treating it as plain text. The system then scans the AST to identify all `MethodDeclaration` nodes, which represent the definitions of methods within the class. By iterating through these identified methods, the system compare them with the agent's selected ones. During this stage, it compares both the number and the types of parameters to effectively resolve the *Method Overloading* problem, where multiple methods share the same name but differ in their parameters. Once a match is confirmed, the function extracts all the necessary details for the next stage: the precise method signature, the package and class name, and the start and end lines within the file.

6. Evaluation

This chapter presents the evaluation setup and interprets the results obtained by running the system on several projects, which will be introduced later. It also discusses key lessons learned throughout the development and analysis of the system, providing useful insights and perspectives for future work.

6.1 Results

To conduct the analysis, five open-source projects were selected as case studies: RxJava [Mic], t-digest [Dun], StreamAlg [Bib], Log4j2 [Fou] and Kafka [JR].

While RxJava, t-digest, and StreamAlg were evaluated across all research questions, the scope for Log4j2 and Kafka is limited to RQ1 and RQ2 (which focus on pass rate and feedback loops). This restriction is due to their significantly higher level of complexity, as well as their larger scale in terms of lines of code and method counts, as detailed in Table 6.1. Specifically, Log4j2 was run three times like the smaller projects, but due to poor initial performance for the pass rate, available resources were invested in more successful cases for the qualitative and cost analyses (RQ3 and RQ4). In contrast, Kafka was executed only once because of its high complexity, which resulted in very high execution times and monetary costs per run. Consequently, Kafka's results should be interpreted keeping this limitation in mind, as they may carry a lower degree of confidence compared to the other projects.

As can be seen from Table 6.1, these projects offer a substantial number of methods for benchmark generation and exhibit strong community support.

Project	Version	Methods	LOC	GitHub Stars
RxJava	3.1.8	9.7k	97k	48.2k
t-digest	3.2	177	2.9k	2.2k
StreamAlg	1.0	134	838	27
Log4j2	2.25.4	18.8k	188.1k	3.6k
Kafka	4.0.2	59.2k	444.9k	33.1k

Table 6.1: Overview of the analyzed projects and their characteristics.

The benchmarks for these projects were generated by the system using OpenAI's gpt-5.4 model, initialized without any custom parameters. This choice was driven by the fact that this model is currently one of the most powerful available, highly proficient in programming and agentic workflows, and supports a large context of up to 1M tokens [Opea].

To validate the microbenchmarks in an isolated environment, it was deemed appropriate to execute them within a virtual machine. This prevents the results from being altered by other processes running on the host machine and by background noise originating from the hardware and the operating system. The setup was particularly crucial for the latency injection phase, where the execution times of the benchmarks were analyzed in detail.

Specifically, the experiment was executed on a virtual machine hosted in an OpenStack cloud environment. The instance utilized KVM virtualization with 8 virtual CPUs, 8 GB of memory and ran Ubuntu 22.04.3 LTS with a GNU/Linux 5.15.0-91-generic x86_64 kernel.

The benchmarks generated on these repositories were evaluated using some metrics:

- **Pass rate** is defined as the ratio of benchmarks successfully generated by the system, meaning they compile, execute correctly, and pass the pipeline's validation check, relative to the total number of benchmarks attempted.
- **Feedback success** is defined similarly to the Pass Rate, but it only considers benchmarks successfully generated on the first attempt of the pipeline. It therefore expresses the ratio of benchmarks successfully generated during the *first pipeline iteration* relative to the total number of benchmarks attempted, thereby reflecting whether or not feedback loops were utilized.
- **Meaningfulness** is defined as a property that a benchmark exhibits only when it satisfies all these three conditions: it must invoke the target method, it must be executed under a realistic workload and its input data must utilize polymorphism. In certain cases, it was appropriate to include additional project-specific criteria, namely ensuring the correct level of parallelism and the proper management of state reuse.
- **Detected slowdown rate** represents the number of benchmarks in which the artificial slowdown injected into the source code was detected, compared to the number of cases where the detection lacked a sufficient margin.
- **Token usage** is the total amount of tokens, split into input and output tokens, used to generate the benchmarks for the project under review.
- **Estimated cost per benchmark** is the estimated cost of generating a single benchmark for each project under review, calculated based on the number of tokens consumed and the pricing of the LLM used.
- **Average time** is the average duration computed for two separate stages: the planning phase and the actual benchmark generation phase (which excludes the planning process).

6.1.1 RQ1 & RQ2: Pass Rate

RQ1 & RQ2: Code Validity and Feedback loops. These research questions aim to analyze the proportion of benchmarks that are successfully generated by the system, meaning they compile, execute correctly, and pass the pipeline's validation check, and to verify whether the presence of feedback loops contributes to this success.

The Table 6.2 shows the average pass rate and feedback success of all five projects.

The project with the highest pass rate is t-digest, followed closely by StreamAlg with only a

Project	Pass rate (Mean [Min-Max]) (%)	Feedback success (Mean [Min-Max]) (%)
RxJava	83.9 [82.1 - 85.6]	34.1 [28.5 - 38.4]
t-digest	95.8 [93.2 - 98]	28.8 [25 - 35.8]
StreamAlg	94.8 [94.7 - 95.1]	26.4 [21.1 - 29.3]
Log4j2	69.2 [60.6 - 74.1]	18.5 [14.8 - 20.7]
Kafka	27.8 [–]	13.9 [–]

Table 6.2: Average Pass rate and Feedback success of all five projects. Kafka was run only once, so it does not have min and max values.

one percent difference. These two projects have similar feedback success rates, with a difference of just 2.4%. On the other hand, RxJava shows the opposite behavior: its pass rate is lower by more than 10%, but its feedback success rate is at least 5.3% higher. The similar behavior in t-digest and StreamAlg, which is opposite to RxJava, is due to the different complexity of the projects. In fact, RxJava is the most complex of the three because it manages asynchronous events and coordinates threads. This higher complexity makes it harder for the system to generate working benchmarks on the first try. However, it increases the positive impact of the feedback loops, since creating a valid benchmark on the first attempt is much rarer. This behavior is even more evident in Kafka, the most complex project, where feedback loops account for 50% of the successfully generated benchmarks.

These figures are at first glance higher than those obtained by ChatPerfTest [Zhi+25], which reports 45.3% for compilable tests and 41.3% for executable tests, percentages which are surpassed by four out of five of our analyzed projects. However, the comparison must be analyzed with caution and without jumping to hasty conclusions, given that the two systems use different LLM models and are evaluated on different projects.

Specifically for StreamAlg, a project shared between the evaluation of the two systems, ChatPerfTest generates 12 benchmarks correctly out of the 18 considered, reaching a success rate of 66.7%, significantly lower than the 94.8% success rate achieved by our system. Regarding t-digest, ChatPerfTest correctly generated benchmarks for 4 out of the 5 target methods considered, equivalent to 80.0%, a figure surpassed by our system's success rate of 95.8%. However, in this final case, the sample size of evaluated methods is too small to draw definitive conclusions from the isolated comparison of this specific project.

The system was executed three times for each project and only once for Kafka, for the reasons explained at the beginning of this chapter. The results for each project are presented in detail below.

RxJava

From the RxJava [Mic] results, which can be viewed in Figure 6.1, **compilation and meaningfulness** errors were identified as the prevailing causes of failure. It is the project on which the system performed the worst among the three analyzed and where the causes of failure are the most diverse. This occurs because, given the complexity of the project, the system struggles to handle all the specific edge cases due to its asynchronous execution and concurrency, which also require elaborate `@Setup` methods. Furthermore, the number of operators offered by the library is very vast and this sometimes causes the system to struggle to find exactly what it needs, leading to hallucinations, invocations of non-existent methods, or in-

correct imports, as well as attempts to access internal classes (`.internal.`, `.impl.`) that are inaccessible in the compilation classpath. Isolating individual methods is difficult, since testing the target method requires numerous preliminary setup steps that pollute the final measurement. Additionally, for reactive (lazy) types, there is a risk that the benchmark limits itself to creating the pipeline without actually triggering its execution (via subscription or waiting for completion), producing a measurement that reflects only the pipeline construction cost rather than the real computational work.

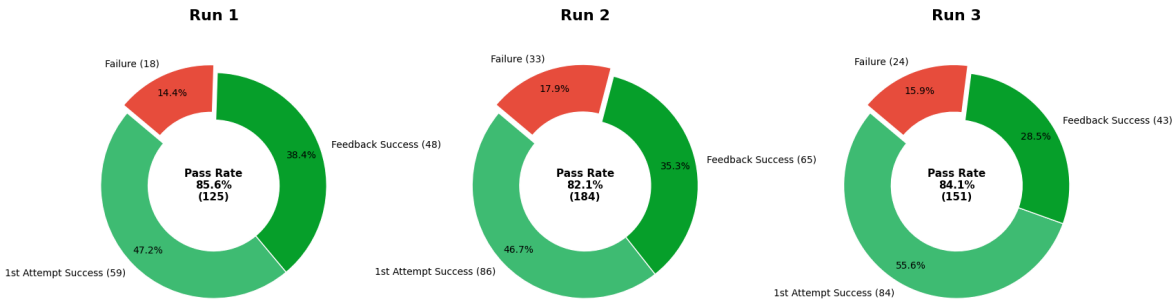


Figure 6.1: PassRate RxJava

t-digest

In t-digest [Dun], the causes of failure shown in Figure 6.2 are mainly due to **meaningfulness errors**. The number of classes and methods is lower compared to RxJava, making it easier for the system to identify them correctly, which results in a near absence of compilation errors. The library relies on digests, which must be correctly created and populated before the target method can be invoked. This does not always happen in isolation, causing the benchmark to measure significantly more than the target method. Furthermore, there are instances of: unrepresentative state, where the input used does not reflect a realistic usage of the library, missing target execution, meaning the benchmark fails to call the method under test, and constant folding & dead code elimination, where the use of hard-coded inputs and the lack of Blackhole consumption make the benchmark susceptible to these optimizations. Finally, there is a single case where the system attempts to call private methods that are inaccessible to the generated benchmarks, resulting in a *compilation error*.

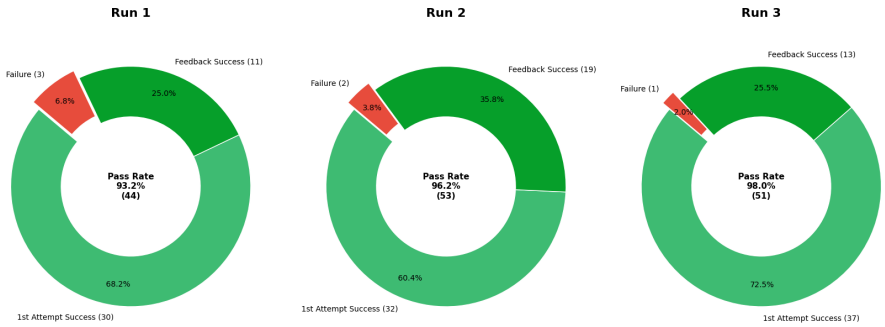


Figure 6.2: PassRate t-digest

Streamalg

StreamAlg [Bib] is the least complex case analyzed, featuring few operations, synchronous execution, and straightforward setup procedures. In all three runs shown in Figure 6.3, the methods causing the failures are the same, namely the `count()` method of the `ExecPullFactory`. The error generated is a timeout, meaning the execution takes too long to complete. This occurs because the implementation of the `count` method in the library contains an infinite loop and, therefore, erroneously never returns. Consequently, these errors are not due to the benchmark generation process but to the implementation of the tested project itself, showing that the system successfully generated correct benchmarks for all cases that did not present this intrinsic library issue.

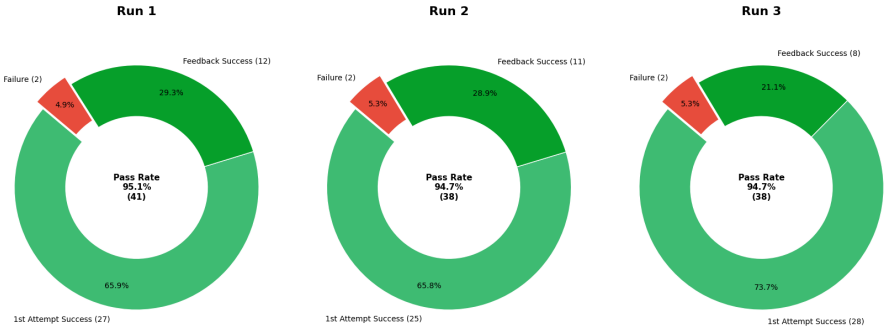


Figure 6.3: PassRate Streamalg

Log4j2 and Kafka

For the pass rate analysis, the Log4j2 [Fou] and Kafka [JR] projects were also evaluated, both of which present an even higher level of complexity.

In Log4j2, a framework that enables event logging for applications running on the JVM, several library components require external configuration files to be properly initialized, a condition that the system cannot replicate autonomously in the code. Consequently, the system attempts to reconstruct the context through alternative means, such as inventing non-existent methods, which in the worst case leads to a failure rate of nearly 40%, a figure significantly lower than the results obtained for the three previously analyzed projects. Even in this case, feedback loops play a key role, accounting for between 20% and 28% of all successfully generated benchmarks.

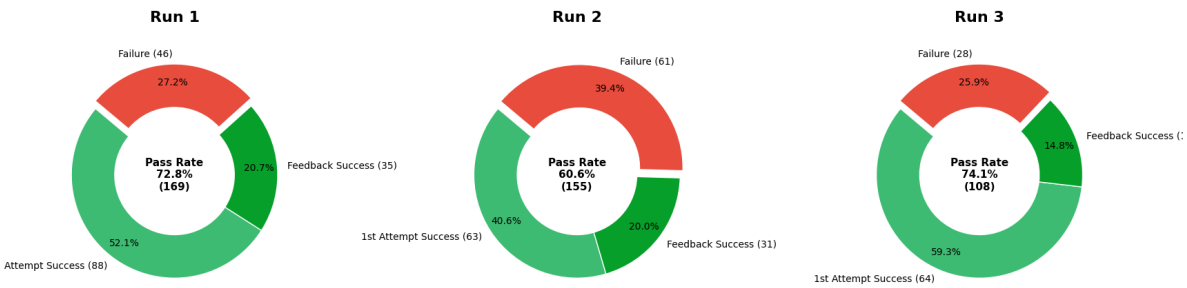


Figure 6.4: PassRate Log4j2

In the Kafka project, a distributed event streaming platform, these same difficulties combine with the need to manage network infrastructure and asynchronous operations across multiple threads. These conditions make benchmark configurations too difficult to reproduce, resulting in a pass rate of only 27.8%. Feedback loops prove to have a fundamental role, determining the correct generation of benchmarks in up to 50% of successful cases. This validates the hypothesis introduced with RxJava, where more complex projects benefit the most from feedback loops.

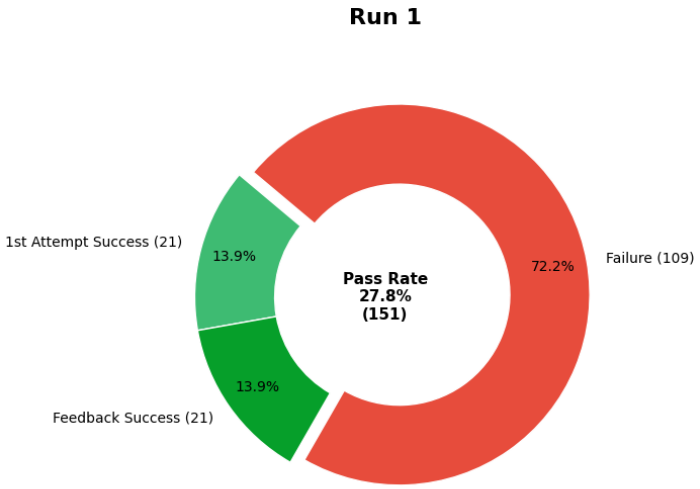


Figure 6.5: PassRate Kafka

As a final analysis, we evaluated the actual usefulness of allowing up to five generation attempts for each microbenchmark, to determine whether or not they are all necessary. If a microbenchmark fails any pipeline check and its generation is therefore unsuccessful, another attempt is triggered, capped at a maximum of five to avoid infinite loops.

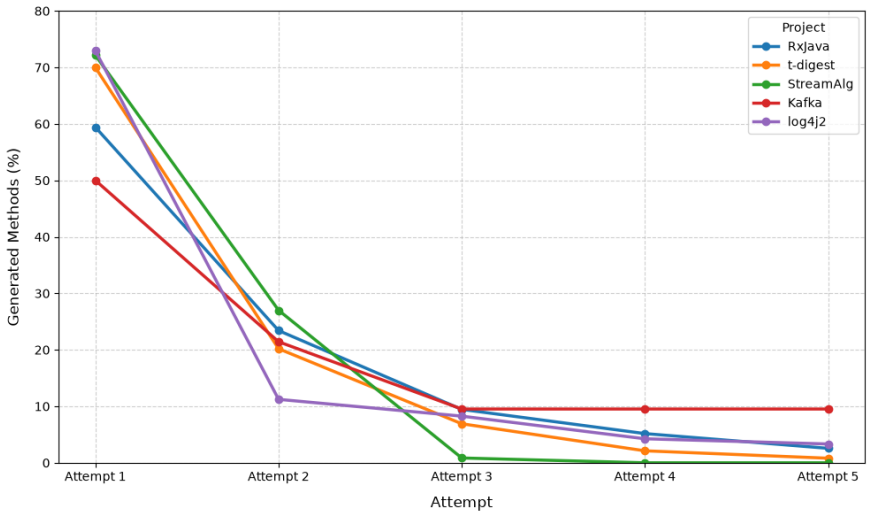


Figure 6.6: Percentages of methods correctly generated at every attempt over the total number of methods correctly generated, for each project (failed generations are therefore not included). For RxJava, StreamAlg, t-digest and Log4j2, the numbers represented are the average between the 3 runs, whereas for Kafka they are the ones of its single run.

Figure 6.6 shows a negative slope for the lines of all five projects, with first-attempt percentages representing between 50% and 73% of the total successfully generated benchmarks. With the second attempt, the cumulative success rate ranges between 71% and 99%, thus accounting for the vast majority of the total. Subsequent attempts provide progressively smaller contributions, with the third attempt contributing between 7% and 10% in nearly all cases and the fourth and fifth between 0% and 5%, with a single exception.

Analyzing the slopes in more detail, we observed behaviors related to the complexity of the projects. StreamAlg and t-digest are the least complex projects and exhibit the steepest slope, bringing the total number of correctly generated benchmarks to 99.2% and 90.2% by the second iteration, and 100% and 97.1% by the third iteration, respectively. Consequently, it is important to note that no benchmarks were correctly generated for StreamAlg in the subsequent iterations. Log4j2 and RxJava, libraries with a more intricate nature, converge less quickly, showing considerable percentages of 7.58% and 7.73% in the last two attempts. Kafka, the most complex project, shows the same behavior, converging even more slowly, with the first attempt contributing only 50% and the last two contributing 19%, which is a non-negligible percentage.

Allowing up to five iterations is therefore redundant for simpler source projects, as nearly all benchmarks are generated within the first three attempts. In contrast, for more complex projects, additional iterations provide a significant contribution, enabling the generation of an additional 7% to 19% of benchmarks that would have otherwise failed.

RQ1 & RQ2: results
Considering RxJava, t-digest and StreamAlg the pass rate ranges from 82.1% to 98.0%. Among these, tests generated with the assistance of feedback loops account for 21.1% to 38.4% of all attempted benchmarks. Overall, the trend indicates a sufficiently high pass rate, with feedback loops playing a pivotal role in this success. While a direct comparison should be taken with caution due to differences in the target projects and underlying LLMs, these figures appear higher than those obtained by ChatPerfTest [Zhi+25], which achieved 45.3% for compilable tests and 41.3% for executable tests in their original study. However, this performance remains highly sensitive to the complexity of the project under examination. For instance, RxJava exhibits a highly concurrent nature that is more difficult to manage than t-digest and StreamAlg, and this complexity is reflected in its lower pass rate, which falls between 82.1% and 85.6%. This is then confirmed by the results of the Log4j2 and Kafka runs, which exhibit even lower pass rates due to their structural complexity, but still benefit from the feedback loops, which account for between 20% and 50% of all successfully generated benchmarks.
The distribution of generations per attempt in Figure 6.6 shows a negative slope for all five projects, with first-attempt percentages representing between 50% and 73% of the total successfully generated benchmarks. By the second attempt, the cumulative percentages range between 71% and 99%, the third attempt contributes between 7% and 10% in nearly all cases, and the fourth and fifth between 0% and 5%, with a single exception. We observe that StreamAlg and t-digest, the simplest projects, converge rapidly to 99.2% and 90.2% by the second attempt, whereas for more intricate libraries like Log4j2, RxJava, and especially Kafka, later attempts are still required to generate a substantial percentage of tests. This occurs because, in simpler scenarios, the system manages to generate a valid benchmark almost immediately, whereas in more challenging cases, multiple attempts are necessary to achieve a valid result.

6.1.2 RQ3: Meaningfulness and Latency Injection

RQ3: Quality. In addition to verifying that the benchmarks meet the technical requirements for proper execution, it is essential to establish whether the generated tests are actually useful for measuring a method's performance and, even more importantly, whether they are capable of detecting the presence of a slowdown into the source code. To address this need, the analysis was divided into two phases: a first static phase focused on determining whether a benchmark is meaningful or not, and a second dynamic phase centered on latency injection.

6.1.2.1 Meaningfulness

The meaningfulness of microbenchmarks was assessed against a set of specific criteria, all of which must be satisfied for a test to be considered meaningful. This definition was established by the authors and the process of determining whether and which criteria were met was conducted by a single person. A more robust approach would have been to involve multiple evaluators in this phase to minimize subjectivity in the assessment.

In this phase, 25 correctly generated benchmarks were selected per project and evaluated according to the following criteria:

- **Target method**: Does the benchmark correctly invoke the target method for which it was created?

- **Real workload**: Is the benchmark executed with data that allows the method under test to perform actual computation? Empty inputs, zeros, or null values are considered invalid.
- **Polymorphism**: Is the benchmark executed with different data types and values?
- **Threading level** (only for RxJava): Is the concurrency level chosen in the benchmark between JMH Threads and RxJava Internal Threads representative of the method's intended use case?
- **State reuse** (only for t-digest): Is the state correctly reset between JMH iterations? It is crucial for cumulative data structures that show performance degradation as they grow.

For the RxJava project, given its concurrent nature, it was considered necessary to add a fourth criterion to evaluate the meaningfulness, with the aim of assessing whether the level of parallelism used is appropriate for the situation under analysis.

Parallelism, in fact, can be implemented in two ways:

- **JMH Threads**: The threads considered at this level are JMH Threads. These represent the number of concurrent threads created by the framework to execute the @Benchmark method. This configuration is relevant for evaluating the behavior of a method under shared access and measuring resource contention. To create them, the following annotations are used: `@Threads`, `@Group` and `@GroupThreads`.
- **RxJava Internal Threads**: Beyond the concurrency managed by JMH, these represent the threads created internally by the RxJava library during a single benchmark invocation. They are designed to highlight the asynchronous or parallel nature of the operator under test. Such threads are typically managed via schedulers, such as `Schedulers.computation()`, `Schedulers.single()`, or `Schedulers.io()`, or through parallel processing operators like `.parallel().runOn(...)`.

Consequently, the threading criteria does not only confirm the presence of threading where necessary, but also judges whether the correct concurrency model between JMH-level and RxJava-level was applied.

For t-digest, the fourth criterion considered was the proper management of state reuse, assessing whether the state of the data structures is reset to its original form between Java iterations. This is because t-digest data structures often grow significantly and rapidly, causing the performance of operations on them change substantially. To maintain consistency across measurements, it is advisable to reset the state each time, ensuring that every iteration starts from a common initial state.

Table 6.3 presents the meaningfulness percentages of the selected benchmarks across the three projects. For each project, 25 benchmarks were evenly selected from the successfully generated ones across the 3 runs. The highest value is reached by StreamAlg and the lowest by t-digest. Table 6.4 details the causes behind the benchmarks labeled as non-meaningful.

Project	Meaningful Benchmarks (%)
RxJava	80.0
t-digest	68.0
StreamAlg	96.0

Table 6.3: Meaningfulness comparison between the analyzed projects.

Cause	RxJava	t-digest	Streamalg
Misconfigured threading	4	0	0
State Reuse	0	3	0
Unrealistic workload	0	3	0
Insufficient polymorphism	1	3	1
Incorrect method invoked	0	1	0

Table 6.4: Distribution of non-meaningfulness causes per project.

All **RxJava**'s instances of non-meaningfulness can be traced back to the multithreaded management employed and, in one specific case, also to a lack of polymorphism, reflecting the system's difficulty in managing the concurrent nature of the project.

Most cases involve the incorrect management of work division among threads, specifically with the `.parallel()` method. This method only splits the stream into rails, which must then be distributed to the threads via `.runOn(scheduler)`, an operator that is frequently missing in the generated microbenchmarks, thus causing the execution to run completely sequentially on a single thread. As a result, the overhead of managing the parallel structure is measured without the structure actually being used. Another issue encountered is the use of the `@Threads` annotation to simulate concurrent access, but with the `@State(Scope.Thread)` scope, which allocates a data structure for each individual thread, completely negating the resource contention intended by the method under test.

Among the meaningful benchmarks, six execute the target method under synchronous conditions which, although isolating the method's logic, prevent representing its asynchronous nature. This highlights the system's limitation in accurately representing these scenarios.

In **t-digest** almost every examined category contributes to the generation of non-meaningful benchmarks and in some cases these issues overlap within a single benchmark. The errors that most reflect the nature of the project are those related to state reuse, where the digest is left unreset and is not pre-populated to a stable state between successive iterations. Consequently, JMH aggregates inconsistent performance data, mixing nearly instantaneous initial inserts with much slower and heavy operations as the structure fills up. For example, an add() operation on a sparsely populated digest (a typical t-digest data structure) is computationally far lighter than one performed on a digest already containing 10.000 elements, making the final average meaningless.

Additionally, 6 of the 20 benchmarks classified as 'meaningful' suffer from poor isolation. In these cases, the benchmark body includes extra tasks, such as creating objects, running validation loops, or making unnecessary Blackhole calls, that slow down the execution. Because these extra instructions are often as heavy as the method being tested, the final results measure

the entire block of code instead of isolating the target function.

StreamAlg achieves the highest rate of meaningful benchmarks, which is consistent with its lower complexity, as described earlier. The only instance of non-meaningfulness is due to a lack of polymorphism, where setting the test variable exclusively to `TRUE` causes the input data to unconditionally trigger the `then` branch, thereby never exercising the `else` branch.

It is important to highlight that in 6 out of 25 cases, despite satisfying the meaningfulness criteria, the benchmarks exhibit only a partial isolation of the method under test. In these scenarios, it was necessary to include auxiliary operations, such as stream consumption, to force the execution of the target method, which inevitably introduces measurement overhead.

Comparison with baseline benchmark - RxJava

In this example, a system-generated benchmark is evaluated against its counterpart from the official RxJava repository, focusing on the `Flowable.groupBy()` method. Since both target the same functionality, this comparison highlights their similarities and differences, revealing the distinct approaches behind the two versions.

RxJava's baseline benchmark:

```
1  package io.reactivex.rxjava3.parallel;
2
3  import java.util.Arrays;
4  import java.util.concurrent.TimeUnit;
5
6  import org.openjdk.jmh.annotations.*;
7  import org.openjdk.jmh.infra.Blackhole;
8  import org.reactivestreams.Publisher;
9  import io.reactivex.rxjava3.core.*;
10 import io.reactivex.rxjava3.flowables.GroupedFlowable;
11 import io.reactivex.rxjava3.functions.Function;
12 import io.reactivex.rxjava3.schedulers.Schedulers;
13
14 @BenchmarkMode(Mode.Throughput)
15 @Warmup(iterations = 5)
16 @Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
17 @Fork(value = 1, jvmArgsAppend = { "-XX:MaxInlineLevel=20" })
18 @OutputTimeUnit(TimeUnit.SECONDS)
19 @State(Scope.Thread)
20 public class ParallelPerf implements Function<Integer, Integer> {
21
22     @Param({"10000"})
23     public int count;
24
25     @Param({"1", "10", "100", "1000", "10000"})
26     public int compute;
27
28     @Param({"1", "2", "3", "4"})
29     public int parallelism;
30
```

```
    Flowable<Integer> groupBy;

    @Override
    public Integer apply(Integer t) {
        Blackhole.consumeCPU(compute);
        return t;
    }

    @Setup
    public void setup() {

        final int cpu = parallelism;

        Integer[] ints = new Integer[count];
        Arrays.fill(ints, 777);

        Flowable<Integer> source = Flowable.fromArray(ints);

        groupBy = source.groupBy(new Function<Integer, Integer>() {
            int i;
            @Override
            public Integer apply(Integer v) {
                return (i++) % cpu;
            }
        })
        .flatMap(new Function<GroupedFlowable<Integer, Integer>,
        ↪  Publisher<Integer>>() {
            @Override
            public Publisher<Integer> apply(GroupedFlowable<Integer, Integer>
            ↪  g) {
                return
                ↪  g.observeOn(Schedulers.computation()).map(ParallelPerf.this);
            }
        });
    }

    void subscribe(Flowable<Integer> f, Blackhole bh) {
        PerfAsyncConsumer consumer = new PerfAsyncConsumer(bh);
        f.subscribe(consumer);
        consumer.await(count);
    }

    @Benchmark
    public void groupBy(Blackhole bh) {
        subscribe(groupBy, bh);
    }
}

```

System's generated benchmark:

```

package io.reactivex.rxjava3.core;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.Setup;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Threads;
import org.openjdk.jmh.annotations.Warmup;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@Threads(1)
public class FlowableGroupby_WithFunctionFunction {

    @State(Scope.Thread)
    public static class BenchmarkState {
        @Param({"128", "1024"})
        public int itemCount;

        public int keyMod;
        public io.reactivex.rxjava3.core.Flowable<java.lang.Integer> source;
        public io.reactivex.rxjava3.functions.Function<java.lang.Integer,
        ↪  java.lang.Integer> keySelector;
        public io.reactivex.rxjava3.functions.Function<java.lang.Integer,
        ↪  java.lang.String> valueSelector;

        @Setup
        public void setup() {
            keyMod = java.lang.Math.max(2, itemCount / 8);
            source = io.reactivex.rxjava3.core.Flowable.range(0,
            ↪  itemCount).map(new
            ↪  io.reactivex.rxjava3.functions.Function<java.lang.Integer,
            ↪  java.lang.Integer>() {
                @Override
                public java.lang.Integer apply(java.lang.Integer v) {
                    return v + 1;
                }
            });
```

```
            keySelector = new
            ↪  io.reactivex.rxjava3.functions.Function<java.lang.Integer,
            ↪  java.lang.Integer>() {
                @Override
                public java.lang.Integer apply(java.lang.Integer v) {
                    return v % keyMod;
                }
            };
            valueSelector = new
            ↪  io.reactivex.rxjava3.functions.Function<java.lang.Integer,
            ↪  java.lang.String>() {
                @Override
                public java.lang.String apply(java.lang.Integer v) {
                    return "v" + v;
                }
            };
        }
    }

    @Benchmark
    public void groupBy(org.openjdk.jmh.infra.Blackhole blackhole,
    ↪  BenchmarkState state) {
        io.reactivex.rxjava3.core.Flowable<io.reactivex.rxjava3.flowables
            .GroupedFlowable<java.lang.Integer, java.lang.String>> grouped =
            state.source.groupBy(state.keySelector, state.valueSelector);
        blackhole.consume(grouped);

        java.util.List<io.reactivex.rxjava3.flowables
            .GroupedFlowable<java.lang.Integer, java.lang.String>> groups =
            grouped.toList().blockingGet();
        blackhole.consume(groups);

        for
        ↪  (io.reactivex.rxjava3.flowables.GroupedFlowable<java.lang.Integer,
        ↪  java.lang.String> group : groups) {
                blackhole.consume(group);
                java.lang.Integer key = group.getKey();
                blackhole.consume(key);
                java.util.List<java.lang.String> values =
                ↪  group.toList().blockingGet();
                blackhole.consume(values);
                for (java.lang.String value : values) {
                    blackhole.consume(value);
                }
        }
    }
}

```

Both benchmarks focus on testing the `groupBy` method on a `Flowable` of integers, measuring the throughput of the execution. However, there is a substantial difference in their behavior.

In the system-generated benchmark, only the intrinsic cost of `groupBy` is measured. Specifically, the following phases are analyzed: partitioning the data into sub-flows, transforming each element from an integer to a string, and collecting the sub-flows into a Java List. This microbenchmark isolates the operator and measures its performance in a purely sequential context. The entire execution is triggered by the instruction `group.toList().blockingGet()` inside a for loop. Due to the presence of `blockingGet()`, each sub-flow is processed synchronously and sequentially on the main thread: the selection of elements belonging to the current sub-flow and their transformation into strings happen one group at a time. This pattern does not reflect a typical real-world usage of `groupBy()`, since the operator is almost always combined with concurrency primitives such as `observeOn()` to achieve parallelism.

In the benchmark written by the RxJava developers, what is measured is the cost of distributing a dataset across multiple threads, each of which simulates heavy computational work on different CPU cores, and merging the results back into a single flow. In this case, the work on each thread is executed in parallel via `observeOn(Schedulers.computation())`, unlike the system-generated benchmark which, although isolating the operator, does not enable the standard use case of `groupBy()`.

Example of a Meaningful Benchmark - T-digest

This example illustrates a benchmark categorized as meaningful and it explains the reasons why it was classified as such. Specifically, the following case refers to the `compress()` method within the `MergingDigest` class and is considered meaningful for the reasons detailed below.

```
1  package com.tdunning.math.stats;
2
3  import java.util.concurrent.TimeUnit;
4  import org.openjdk.jmh.annotations.Benchmark;
5  import org.openjdk.jmh.annotations.BenchmarkMode;
6  import org.openjdk.jmh.annotations.Fork;
7  import org.openjdk.jmh.annotations.Level;
8  import org.openjdk.jmh.annotations.Measurement;
9  import org.openjdk.jmh.annotations.Mode;
10 import org.openjdk.jmh.annotations.OutputTimeUnit;
11 import org.openjdk.jmh.annotations.Param;
12 import org.openjdk.jmh.annotations.Scope;
13 import org.openjdk.jmh.annotations.Setup;
14 import org.openjdk.jmh.annotations.State;
15 import org.openjdk.jmh.annotations.Threads;
16 import org.openjdk.jmh.annotations.Warmup;
17
18 @BenchmarkMode(Mode.Throughput)
19 @OutputTimeUnit(TimeUnit.SECONDS)
20 @Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
21 @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
22 @Fork(3)
23 @Threads(1)
24 public class MergingDigestCompress_NoArgs {
```

```
25      @State(Scope.Thread)
26      public static class BenchmarkState {
27          @Param({"100", "1000", "10000"})
28          public int itemCount;
29
30          @Param({"50.0", "200.0"})
31          public double compression;
32
33          @Param({"ascending", "descending", "shuffled"})
34          public String order;
35
36          public double[] values;
37          public long shuffleSeed;
38          public com.tdunning.math.stats.MergingDigest digest;
39
40          @Setup(Level.Trial)
41          public void setupValues() {
42              values = new double[itemCount];
43              for (int i = 0; i < itemCount; i++) {
44                  values[i] = ((i * 31) % (itemCount + 17)) + ((i % 7) *
                    ↪ 0.125d);
45              }
46
47              if ("descending".equals(order)) {
48                  for (int i = 0, j = values.length - 1; i < j; i++, j--) {
49                      double tmp = values[i];
50                      values[i] = values[j];
51                      values[j] = tmp;
52                  }
53              } else if ("shuffled".equals(order)) {
54                  shuffleSeed = 0x9E3779B97F4A7C15L ^ itemCount ^
                    ↪ Double.doubleToLongBits(compression);
55                  java.util.Random random = new java.util.Random(shuffleSeed);
56                  for (int i = values.length - 1; i > 0; i--) {
57                      int j = random.nextInt(i + 1);
58                      double tmp = values[i];
59                      values[i] = values[j];
60                      values[j] = tmp;
61                  }
62              } else {
63                  shuffleSeed = 0x9E3779B97F4A7C15L ^ itemCount ^
                    ↪ Double.doubleToLongBits(compression);
64              }
65          }
66
67          @Setup(Level.Invocation)
68          public void setupDigest() {
69              digest = new com.tdunning.math.stats.MergingDigest(compression);
70              for (int i = 0; i < values.length; i++) {
71                  digest.add(values[i]);
72              }
```

```
        }
    }

    @Benchmark
    public void compress(BenchmarkState state,
    ↪   org.openjdk.jmh.infra.Blackhole blackhole) {
        state.digest.compress();
        blackhole.consume(state.digest.centroidCount());
    }
}
```

A brief overview of the library's mechanics is necessary before examining the benchmark, as it provides the context needed to verify if the benchmark was built according to the library's specific rules.

The t-digest is a library that estimates statistics of a dataset and allows it to be represented through centroids, which together form a digest. Each centroid has a mean and a weight, representing the mean of the points it encompasses and their count respectively, and the total number of centroids in the digest is determined by the compression parameter [Man18].

The method analyzed here performs the compression of a digest, which is the procedure that integrates new samples into the existing centroids while maintaining the structural constraints of the digest.

The setup process is structured into two levels:

- `Level.Trial`: These instructions are executed once at the beginning of each fork to generate an array of values, which is then sorted according to the specified order parameter (ascending, descending or shuffled). These operations do not need to be repeated for every benchmark run. The data vector is created only once and it is read by all benchmarks without being ever modified.

- `Level.Invocation`: This phase is executed before every call to the `@Benchmark` method. A new digest is created and is populated by adding all elements from the values array. The `add()` method only stores new data in a temporary buffer but it does not merge them into centroids. It is essential to create and populate a new digest for every single benchmark execution, otherwise the actual work would only be performed during the first call and all subsequent benchmarks would start with an already compressed digest.

The benchmark analyzes the performance of the `compress()` method by varying several parameters:

- `itemCount`: it represents the number of points contained in the initial data structure, where a larger value makes the compression operation more intensive.

- `compression`: represents the number of centroids allowed in the digest, where a larger value allows for greater precision but increase the maintenance cost.

- `order`: The performance depends on the input array's order. If the data is already sorted, the computation becomes faster.

With reference to the meaningfulness criteria explained above, the following observations can

be made for this specific case:

- Target method: The `compress()` method is correctly isolated and invoked within the benchmarked code.
- Real workload: The benchmark utilizes a vector initialized with pseudo-randomly distributed values. Furthermore, by testing across three different sorting orders the benchmark effectively covers the diverse scenarios found in real-world use cases.
- Polymorphism: all the parameters (item count, compression and order) have different values.
- State Reuse: The digest is re-initialized at `Level.Invocation`. This is mandatory because `compress()` irreversibly modifies the digest's state, making it impossible to 'decompress' the structure for subsequent iterations.

The benchmark satisfies all the criteria and is thus classified as meaningful.

Example of a Not Meaningful Benchmark - Streamalg

This example illustrates the only non-meaningful benchmark identified among those sampled across the three runs of the project. The specific reasons for this categorization are detailed below.

```
1
2  package gadt.evaluator;
3
4  import java.util.concurrent.TimeUnit;
5  import org.openjdk.jmh.annotations.Benchmark;
6  import org.openjdk.jmh.annotations.BenchmarkMode;
7  import org.openjdk.jmh.annotations.Fork;
8  import org.openjdk.jmh.annotations.Measurement;
9  import org.openjdk.jmh.annotations.Mode;
10 import org.openjdk.jmh.annotations.OutputTimeUnit;
11 import org.openjdk.jmh.annotations.Param;
12 import org.openjdk.jmh.annotations.Scope;
13 import org.openjdk.jmh.annotations.Setup;
14 import org.openjdk.jmh.annotations.State;
15 import org.openjdk.jmh.annotations.Threads;
16 import org.openjdk.jmh.annotations.Warmup;
17
18 @BenchmarkMode(Mode.Throughput)
19 @OutputTimeUnit(TimeUnit.SECONDS)
20 @Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
21 @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
22 @Fork(3)
23 @Threads(1)
24 public class IfAccept_WithVisitor {
25     @State(Scope.Thread)
26     public static class BenchmarkState {
27         @Param({"128", "1024"})
28         public int itemCount;
```

```
29
30          public java.lang.Boolean testValue;
31          public java.lang.Integer thenValue;
32          public java.lang.Integer elseValue;
33          public gadt.evaluator.If<java.lang.Integer> expr;
34          public gadt.evaluator.Visitor<gadt.evaluator.Id.t> visitor;
35
36          @Setup
37          public void setup() {
38              testValue = java.lang.Boolean.TRUE;
39              thenValue = java.lang.Integer.valueOf(42);
40              elseValue = java.lang.Integer.valueOf(7);
41              expr = new gadt.evaluator.If<>(
42                  new gadt.evaluator.BoolLit(testValue),
43                  new gadt.evaluator.IntLit(thenValue),
44                  new gadt.evaluator.IntLit(elseValue)
45              );
46              visitor = new gadt.evaluator.EvalVisitor();
47          }
48      }
49
50      @Benchmark
51      public void accept(BenchmarkState state, org.openjdk.jmh.infra.Blackhole
        ↪  blackhole) {
52          for (int i = 0; i < state.itemCount; i++) {
53              gadt.App<gadt.evaluator.Id.t, java.lang.Integer> result =
                ↪  state.expr.accept(state.visitor);
54              blackhole.consume(result);
55          }
56      }
57  }
58
```

The streamalg project includes, among other features, the ability to evaluate logical and arithmetic expressions. Each expression is represented by a tree where internal nodes represent `if` conditions, whose children correspond to the `then` and `else` branches, while the leaves contain concrete values. To traverse and evaluate the tree, an `EvalVisitor` object is used.

In the analyzed benchmark, the `if` node has a hardcoded `true` condition, causing the visitor to always execute the `then` branch and never reach the `else` branch. Since both branches are equally important for a complete evaluation of the If node, this represents a case of insufficient polymorphism, as the false case is never exercised, leaving half of the branching logic untested.

6.1.2.2 Latency Injection

The analysis of benchmark meaningfulness adopted a static approach which, while useful, does not detect whether the benchmark actually works in practice. The question now to be analyzed is: at execution time, do the benchmarks generated by the system truly detect a potential performance slowdown in the analyzed method? To answer this, a second analysis was conducted, focusing on latency injection. This approach consists of introducing an artificial slowdown/mutant into the target method, running the benchmark both with and without the

slowdown and verifying if the slowdown is effectively detected by the benchmark.

Table 6.5 illustrates how the 20 benchmarks analyzed in this phase were proportionally distributed across the three projects, focusing exclusively on those labeled as meaningful in the previous phase.

Project	Meaningful benchmark	Benchmark for Latency Injection
RxJava	20	6
t-digest	17	6
streamalg	24	8

Table 6.5: Number of selected methods for latency injection for each project.

Most benchmarks included multiple parameter configurations, meaning that the execution of a single benchmark consisted of several distinct runs, each with a different set of parameters. Since this phase focuses specifically on slowdown detection rather than evaluating polymorphism, a fixed parameter configuration was selected and analyzed for each benchmark.

Workflow The workflow follows an approach similar to that of Jangali et al. with ju2jmh [Jan+23], which involves the use of pa, a tool that enables running a statistical analysis on benchmark data with and without slowdown. In more detail, the following steps were performed for each benchmark considered in this phase:

- **Run the original benchmark**: The original benchmark is executed without any slowdown and the results are saved in JSON format.
- **Compute the slowdown**: The slowdown duration is calculated relative to the baseline benchmark execution (i.e., the run without performance degradation), where all benchmarks were executed using `Mode.Throughput`. The baseline execution yields the $average_throughput$, which represents the number of operations (invocations of the @Benchmark method) completed per second. From this, the $average_invocation_time$ is derived as $1/average_throughput$. This value represents the average time required for a single invocation of the @Benchmark method.
  To take into account also measurement noise, the $error_percentage$ is also introduced as $throughput_error/average_throughput$, which represents the error rate of the baseline execution.

  Two distinct types of slowdown thresholds were calculated and analyzed:

  - $slowdown_10\% = 10\% * average_invocation_time$
  - $slowdown_error = error_percentage * average_invocation_time * 3$

  The first threshold, $slowdown_10\%$, is an absolute metric that does not account for the specific variance of the benchmark. In contrast, $slowdown_error$ incorporates the benchmark's execution error. Here, the multiplication by 3 provides a safety margin to separate the slowdown from background noise, ensuring it can be detected without ambiguity.

  Two different slowdowns have therefore been considered for each benchmark and both

scenarios have been analyzed, resulting in a total of 40 cases, 2 per benchmark.

- **Inject the slowdown**: This process is repeated sequentially for both slowdowns: the slowdown is introduced, the benchmark is executed and the results are saved in JSON format. To inject the slowdown, AspectJ was utilized, a program paradigm that enables the execution of arbitrary code before and/or after a specific method. In this implementation, a slowdown is added immediately before each execution of the benchmark's target method, as follows:

```
@Aspect
public class SlowdownAspect {

    private static final long TOKENS = Long.getLong("slowdownTokens",
    ↪ 0L);

    @Around("execution(*
    ↪ gadt.evaluator.MethodClass.myMethod(methodArgs))")
    public Object injectSlowdown(ProceedingJoinPoint pjp) throws
    ↪ Throwable {
        if (TOKENS > 0) {
            Blackhole.consumeCPU(TOKENS);
        }
        return pjp.proceed();
    }
}

```

This is achieved by invoking Blackhole.consumeCPU(tokens) with a number of tokens proportional to the target delay. The required number of tokens is calculated as $target_slowdown / execution_time_per_token$.
The constant $execution_time_per_token$ was empirically determined with JMH by measuring the execution time of Blackhole.consumeCPU(tokens) using 100 and 1000 tokens and dividing it by its respective number of tokens.

- **Calculate the expected Throughput Degradation**: Starting from the temporal slowdowns, the corresponding throughput degradation is calculated, which represents the expected degradation that should occur when running the benchmark with the slowdown. To do this, the following steps are followed:

$$
\begin{aligned}
\Delta Throughput &= 1 - \frac{Throughput_{slowdown}}{Throughput_{original}} \\
&= 1 - \frac{Time_{original}}{Time_{slowdown}} \\
&= 1 - \frac{Time_{original}}{Time_{original} + slowdown} \\
&= 1 - \frac{1}{1 + \frac{slowdown}{Time_{original}}}
\end{aligned}
$$

where $Time_{original}$ represents the *average_invocation_time* without slowdown and $Time_{slowdown}$ represents the *average_invocation_time* with slowdown.

- **Calculate the Upper RCI and Bug Size**: Using *pa* in *Two version analysis* mode, the upper bound of the Relative Confidence Interval (RCI) of the ratio between the modified and the original execution is obtained. From this, the minimum performance degradation, also called Bug_Size, is derived as $1 - \text{RCI}_{upper}$, guaranteed with 99% confidence.

- **Evaluate the benchmark**: To evaluate the benchmark, it is necessary to verify whether it detected the injected slowdown or not. This is equivalent to achieving a Bug Size large enough to ensure it was not caused by random noise. To do this, a minimum threshold of 1% is defined and applied as follows:

  - If Bug_Size_% $> 1\%$: the slowdown is detected

  - else: the slowdown is not detected

Overall, the analyzed benchmarks detect the injected slowdowns in the vast majority of cases. Specifically, the slowdown is detected in all cases but one, where the Bug_Size is 0.42%, a value too small to draw any conclusion, as the observed performance variation could have been caused by random noise. In all other cases, the Bug_Size is well above the 1% threshold, and except for 3 cases, it also exceeds the 5% threshold.

It is worth noting that any degradation greater than 1% indicates that the benchmark successfully detected the injected slowdown. In some cases, the observed degradation closely matches the expected throughput degradation, confirming that the benchmark captures the slowdown accurately. In other cases, however, the two values differ: although the slowdown is still detected, the magnitude of the observed degradation differs from the expected one, due to internal factors in the code execution, such as JIT optimizations or the interaction between the injected delay and the benchmark workload.

The detected bug sizes and the corresponding theoretical throughput degradation values are reported below, where *thpt_10%* and *thpt_error%* represent the theoretical percentage throughput degradation corresponding to the two types of slowdown, while *bug_10%* and *bug_error%* denote the obtained results:

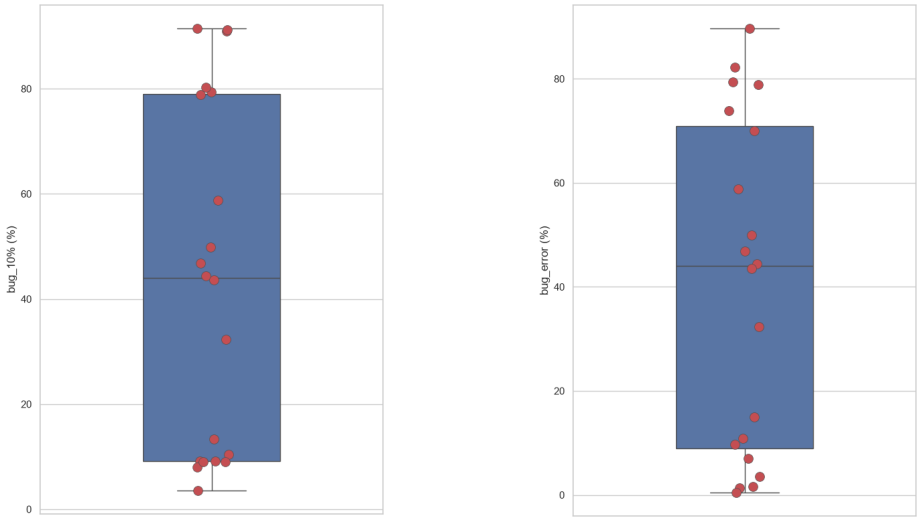
Project	Target method	thpt_10%	bug_10%	thpt_error%	bug_error%
RxJava	flatMap(Function)	9.09	9.14	1.29	1.55
RxJava	observeOn(Scheduler)	9.09	13.39	9.28	14.97
RxJava	switchMap(Function)	9.08	10.38	9.70	9.67
RxJava	fromCallable(Callable)	8.51	44.44	2.27	-
RxJava	groupBy(Function)	9.07	9.17	5.52	7.01
RxJava	buffer(int, int)	9.09	7.95	1.92	1.38
t-digest	add(double, int)	90.91	90.97	89.74	89.72
t-digest	cdf(double)	90.91	91.52	62.45	70.00
t-digest	sort(int[], double[])	9.09	9.02	3.89	3.52
t-digest	**compress()**	9.09	3.53	3.27	**0.42**
t-digest	quantile(double)	7.69	78.89	0.00	-
t-digest	add(double)	90.91	91.27	70.62	73.86
streamalg	map(Function, App)	8.95	32.30	3.35	-
streamalg	filter(Predicate)	9.62	49.91	4.08	-
streamalg	map(Function)	8.64	58.83	1.33	-
streamalg	caseIf(If)	9.38	79.40	1.69	-
streamalg	casePlus(Plus)	9.06	80.26	10.95	82.25
streamalg	flatmap(Function, App)	9.33	43.57	9.93	-
streamalg	reduce(Obj, BinOp, App)	9.72	46.84	1.52	-
streamalg	flatmap(Function)	9.04	9.04	5.00	10.93

Table 6.6: Bug size and expected throughput degradation of analyzed methods.

The data show that the slowdown is detected by nearly all benchmarks, however, it is necessary to specify two groups of tests that exhibit a different behavior than expected.

Some methods show an expected throughput degradation of around 70-90%, significantly diverging from the predicted 8–9% range. This is due to the fact that within these benchmarks, the target method is executed inside a loop. Consequently, the slowdowns injected before each method invocation accumulate, generating a much larger performance degradation. This effect can be identified in some of the outliers present in the boxplots in Figure 6.7.

The methods highlighted in green exhibit execution times ranging from tens to a few hundred nanoseconds, making them particularly fast. As a consequence, the corresponding target slowdowns fall between 0 and 40 ns, intervals too short to be accurately injected. In these cases, the overhead introduced by the injection mechanism itself exceeds the target delay, resulting in an observed Bug_Size much larger than expected. Furthermore, for some of these methods, the *slowdown_error* configuration was not computed since the benchmark's execution error was smaller than the delay already used in the *slowdown_10%* scenario and injecting an even shorter delay would have produced identical behavior, making the measurement redundant. For this reason, in these cases, in boxplot 6.7b the data for *bug_error%* have been replaced with the corresponding values from *bug_10%*.



(a) Distribution of *bug_10%*

(b) Distribution of *bug_error%*

Figure 6.7: Distribution of the throughput degradation actually detected by each benchmark in the two injected slowdown cases: *slowdown_10%* (*bug_10%*) and *slowdown_error* (*bug_error%*).

Considering the 1% threshold, it is exceeded in 38 out of 39 cases, while the 5% threshold is exceeded in 35 out of 39 cases, yielding a *Detected slowdown rate* of 97.4% and 89.7%, respectively. All methods with both slowdown thresholds injected were considered, excluding one of the two `quantile(double)` cases, specifically the one whose *thpt_error%* was 0%, since the injected slowdown was too small to be significant. This reduces the total number of cases considered from 40 to 39. While these results compare favorably with those obtained by ju2jmh, which stand at 80% and 48.4% respectively with a difference of 17.4 and 41.3 percentage points, such a comparison should be interpreted with caution due to differences in the experimental setup and evaluated workloads. It should be also noted that ju2jmh injects 155 mutants, compared to the 40 considered in our case. A larger sample generally yields a more statistically robust estimate, so this gives ju2jmh's result greater statistical relevance. Furthermore, in our case the methods selected for this phase were drawn from those labeled as meaningful, which may have led to selecting only the "best" benchmarks, resulting in higher performance in this phase compared to what would have been obtained by considering all the benchmarks initially generated by the system.

RQ3: results
The percentage of microbenchmarks categorized as meaningful ranges between 68%, a rate obtained for t-digest, and 96%, corresponding to StreamAlg. It can be observed that the system's difficulty in generating meaningful benchmarks is directly linked to the project's complexity. In RxJava, the causes can be traced back to the mismanagement of parallelism, as also highlighted in the comparison with the baseline benchmark. In t-digest, the causes are more heterogeneous, ranging from the incorrect handling of state reuse to the lack of a realistic workload and polymorphism. On the other hand, in StreamAlg, the least complex project, almost all microbenchmarks are meaningful. Furthermore, a poor isolation of the target method is observed in both t-digest and StreamAlg. This highlights that the system's difficulty in generating meaningful benchmarks increases with the complexity of operations and the requirements of the project under test.
Regarding slowdown detection, all considered microbenchmarks, with a single exception, are able to detect a performance degradation exceeding the 1% threshold, and in most cases, the 5% threshold as well. Specifically, the 1% threshold is exceeded in 38 out of 39 cases, while the 5% threshold is exceeded in 35 out of 39 cases, yielding a *Detected slowdown rate* of 97.4% and 89.7% respectively. While these results compare favorably with those obtained by ju2jmh, which stand at 80% and 48.4% respectively with a difference of 17.4 and 41.3 percentage points, such a comparison should be interpreted with caution due to differences in the experimental setup and evaluated workloads. Moreover, ju2jmh injects 155 mutants, compared to the 40 cases considered in this research, making their analysis statistically much more robust. In our analysis, the methods were selected among those labeled as meaningful, and this may have led to considering only the best benchmarks, which perform better even in this context.

6.1.3 RQ4: Time and Cost

RQ4: Time and Cost. In addition to evaluating the efficiency and relevance of the system's results, it was considered important to address the practical applicability of its use in terms of resources, both monetary and time-related. The primary goal of this project is, in fact, to relieve developers of the task of manually writing microbenchmarks, thereby saving the time involved and the costs associated with paying for it. It is therefore necessary to understand whether using the system is actually worthwhile in terms of cost and time, or whether writing microbenchmarks by hand still remains a better solution.

6.1.3.1 Time

This section illustrates the system's execution times.

The first graph in Figure 6.8 shows the time spent by the Planning Agent during the Planning Phase. For each project, it represents the average time across the three runs to find and select methods within the source code, on which benchmarks will then be generated.

The second graph in Figure 6.8 represents the average generation time per benchmark, excluding the Planning Phase. Specifically, for each project, this value is calculated as the average over the three runs of the mean time taken to generate a single benchmark.

It can be observed that the planning phase, while not instantaneous, accounts for only a small fraction of the total time required for the entire benchmark generation process. This behavior

stems from the fact that the planning phase, although complex due to its reliance on an agent that uses tools, is executed only once per run. Its execution time is therefore distributed across all methods and as a result, its impact on the total duration decreases as the number of generated benchmarks increases.

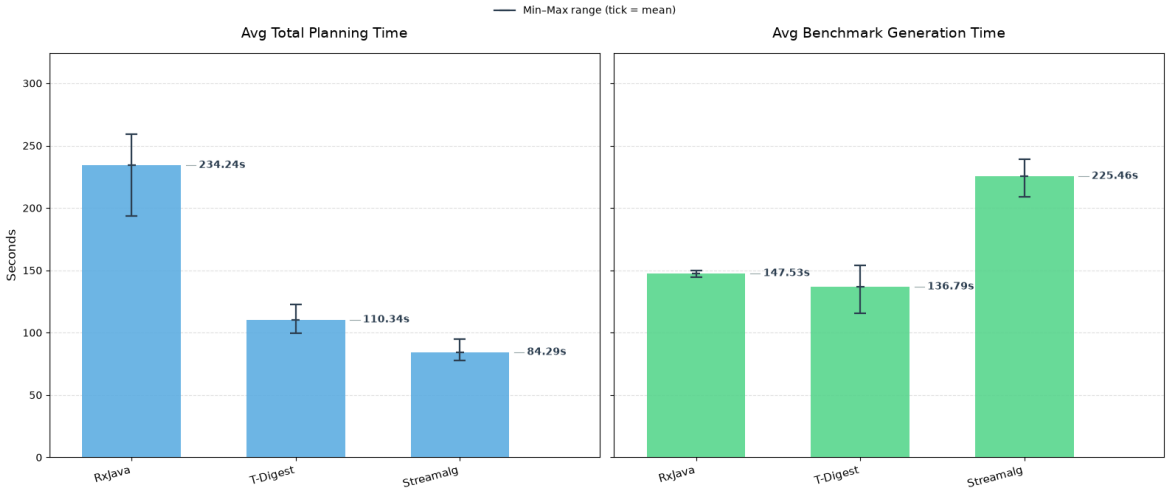


Figure 6.8: Average Planning phase duration and average benchmark's generation time

Considering the average total benchmark generation time, with the planning phase overhead uniformly distributed, the generation time per test ranges from 2.3 minutes for t-digest to 3.8 minutes for StreamAlg. This translates to an output of 15 to 25 microbenchmarks per hour, a rate that significantly exceeds the manual writing speed of a human developer. The complete data can be found in Table 6.7.

6.1.3.2 Cost

Figure 6.9 shows the execution cost for each project, also including the cost of the planning phase in the calculation. For each project, it displays the average number of input and output tokens, along with the average cost per benchmark across the three runs. The cost was calculated by multiplying the input and output tokens by their respective unit prices and dividing the total by the number of benchmarks. Pricing refers to the specific LLM model used (GPT-5.4).

ChatPerfTest [Zhi+25] utilizes 185.3k tokens to generate 223 microbenchmarks, averaging approximately 831 tokens per microbenchmark. In contrast, our system is significantly more token intensive: consumption ranges from 1.6 million tokens for 38 microbenchmarks in the case of StreamAlg, up to 5.6 million tokens for 153 benchmarks in RxJava. This translates to a demand of 7k to 25k tokens per microbenchmark. While this represents a major increase in token consumption, it is justified by the higher complexity of our system compared to ChatPerfTest. Our pipeline is structured into multiple specialized steps, which may be repeated several times in the event of errors. Furthermore, despite implementing context condensation techniques, a substantial amount of information must still be preserved across consecutive prompts.

Looking at Figure 6.9 it is worth noting that the number of input tokens significantly outweighs the output tokens. This is due to the extensive instructions required in each prompt to ensure

high-quality results. Furthermore, the system often needs to process a large context, including the source code of the target method, error logs from previous attempts or earlier versions of the benchmark that require further refinement.

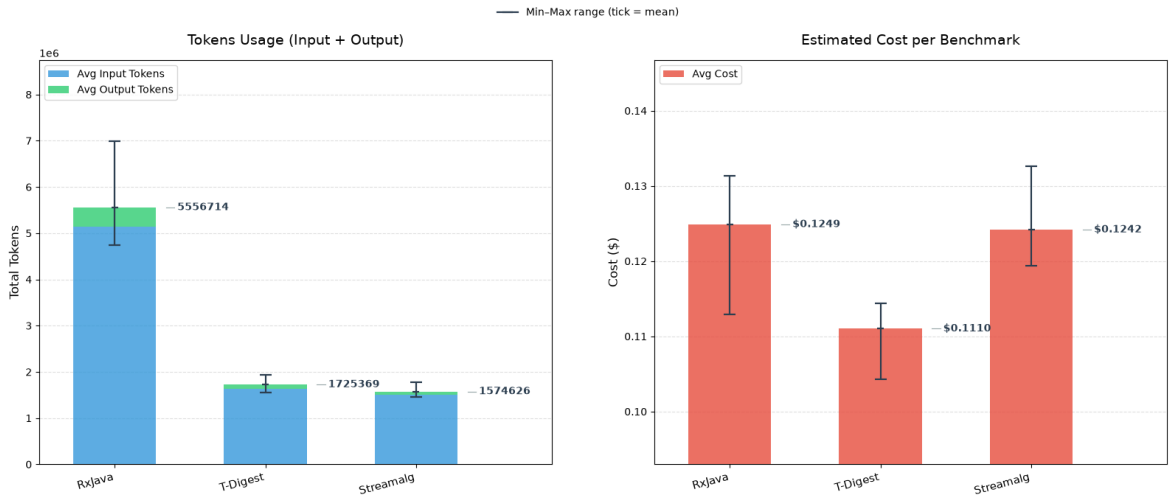


Figure 6.9: Average tokens spent and estimated cost per benchmark

To reduce the number of tokens per request, several techniques were implemented, drawing inspiration from previous work in the literature. Similar to COVERUP [PB25] and ChatPerfTest, entire source files are never included in the prompts. Instead, only the portions essential and relevant to the current task are selected. Specifically, rather than sending the entire source class, only the target method under test is extracted and provided. Furthermore, following the approach of COVERUP, the prompts explicitly instruct the model to respond solely with code, omitting any comments or explanations that would otherwise unnecessarily lengthen the output.

Based on the data in Figure 6.9, the operating cost of our system is approximately $3/hour in the worst-case scenario of RxJava. At this price point, the system successfully generates 24 benchmarks, a cost that is significantly lower than the compensation required for a developer to write them manually. The results and costs for all projects are detailed in Table 6.7, showing even more favorable outcomes.

Project	**Avg_total_time (min)**	**Benchmark/hour**	**cost_per_hour ($)**
RxJava	2.5	24	3.0
t-digest	2.3	25	2.8
StreamAlg	3.8	15	2.0

Table 6.7: Average time to generate a benchmark (including the planning phase), number of benchmarks generated per hour and relative system cost per hour for each project. Values represent the average of three runs.

To identify the most token-intensive step, we analyzed the token consumption of each phase in the pipeline that relies on LLMs. For each project, we calculated the percentage distribution of tokens across the pipeline phases (planning, code generation, code configuration, error classification and validation check) by aggregating data from all available runs.

For all phases except the planning phase, which is executed only once and for which the total number of token count is therefore straightforward, the total tokens were calculated as the sum of the tokens used across every generation attempt made for all methods in that run, and then aggregating these partial sums across all runs.

Finally, the percentage for each phase was computed as the ratio of total tokens consumed by that phase (across all runs of the project) to the total tokens consumed by the project overall (i.e. the sum of all five phase totals)

As illustrated in Figure 6.10, the Code Agent is clearly the most resource intensive step, accounting for over 80% of the total token consumption in all projects. The remaining portion, which hovers around 20%, is distributed among the other four phases. Specifically, the Planning, Config Agent and Validation Check phases exhibit similar results, ranging between 4% and 8.02%. Conversely, the Error Classifier stands out with the lowest consumption, ranging from 0.3% to 1.38%.

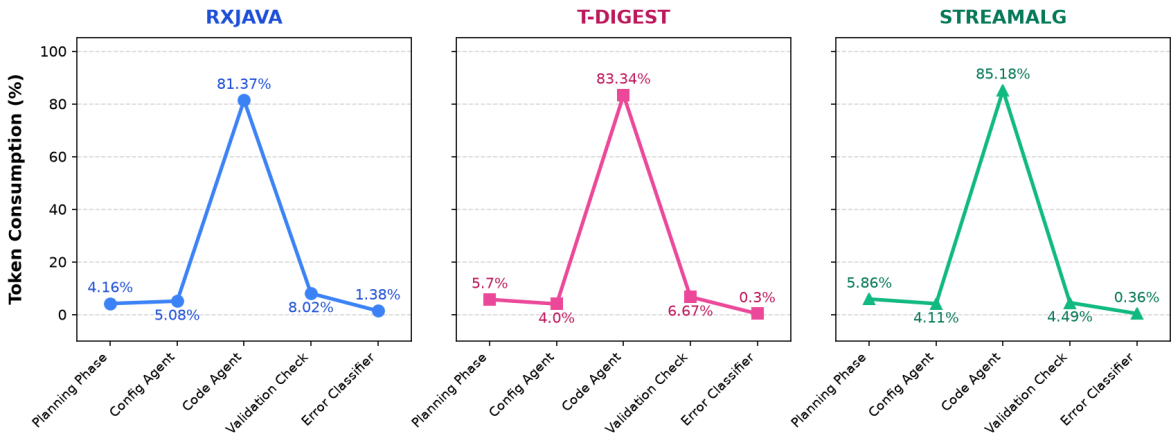


Figure 6.10: Token consumption percentage across pipeline phases, shown individually by project.

The reason for this behavior is that the Code Generation phase is invoked much more frequently during subsequent feedback loops than the Configuration Generation phase. This is because the vast majority of compilation or runtime failures stem from issues within the benchmark's logic itself, rather than its configuration parameters. Furthermore, the Code Generation phase is based on an agent implementing the ReAct framework. This agent relies on tools such as `find_definition` and `read_file` to explore the codebase, which often answer with entire source files, that can be very large. In LangGraph, these tool outputs are appended to the message history and sent back to the LLM in every subsequent call. In addition, when errors occur, the error messages and the previous benchmark attempt are also appended to the prompt. The combination of all these factors is what leads to a rapid increase in token usage with each execution of this pipeline step.

Although the Planning phase is also driven by an agent using the ReAct framework, it is executed only once per system run, meaning its overall impact on total token consumption is significantly mitigated. The other phases of the pipeline rely on agents that perform single LLM calls, which result in a much lower token usage.

RQ4: results
Benchmark generation times vary from 2.3 minutes for t-digest to 3.8 minutes for StreamAlg. This translates to an hourly output of 15 to 25 microbenchmarks, a rate that exceeds the manual writing speed of a human developer. Furthermore, even in the worst-case scenario represented by RxJava, the cost of our system is approximately $3 per hour. Within this timeframe, the system successfully generates 24 benchmarks, a cost that is significantly lower than the compensation required for a developer to write them manually.
Although the planning phase relies on a agent that uses tools, its overall impact on the execution time is minimal because it is executed only once per run. A similar trend is observed in token consumption, where the planning phase, along with the Config Agent and Validation Check phases, is among the steps that account for only 4% to 8.02% of the total tokens utilized during a system run. The least token-intensive phase is the Error Classifier, with percentages ranging from 0.3% to 1.38%. Instead, the vast majority of tokens are invested in the code generation phase, which exceeds 80 % of total usage across all three analyzed projects.
This high consumption is driven by the fact that in case of errors, the Code Generation phase is invoked much more frequently than the Configuration Generation phase. Second, this step is based on an agent implementing the ReAct framework. This setup relies on the execution of tools whose outputs, which can often be very long, are appended to the message history and sent back to the LLM in every subsequent call. Additionally, when errors occur, both the error messages and the previous benchmark attempts are appended to the prompt. The combination of these factors leads to a rapid increase in token usage.
In comparison, ChatPerfTest [Zhi+25] utilizes about 831 tokens to generate a single benchmark, whereas our system is significantly more token intensive, demanding between 7k and 25k tokens per microbenchmark. This is justified by the higher complexity of our system, which is composed of a multi-agent pipeline architecture. To mitigate these figures, inspired by the apporach followed by COVERUP [PB25] and ChatPerfTest, only the code portions essential and relevant to the current task are included in the prompts and the prompts explicitly instruct the model to respond solely with code, thereby omitting any explanations that would otherwise lengthen the output.

6.2 Discussion

This section presents the key lessons learnt during the development of this project, aiming to provide inspiration and a specific perspective on aspects that require closer attention for future research on microbenchmark generation systems.

It is relatively easy to produce benchmarks that execute successfully, but difficult to identify which parts of the project truly need to be benchmarked. Once the target method is identified, the system demonstrates a sufficient pass rate, ranging from an average of 83.9% to 95.8%, for projects of moderate complexity. This indicates that the LLM is generally capable of producing compilable and executable benchmarks. However, the success of the entire pipeline hinges on the initial selection phase, since yielding methods that lack representation of the library makes the entire subsequent generation process ineffective. Originally, the planning agent was designed as a single phase, which caused the LLM to fixate on the first performance hotspot it identified, exploring it in depth while ignoring the rest of the codebase. This limitation motivated splitting the task into two distinct phases to achieve a more balanced exploration of the source code. A possible future direction would be to further structure this phase, particularly by supplying the Planning Agent with external information regarding actual library usage, such as method invocation frequency if available, to ensure the selection of methods that are critical in real-world scenarios, rather than focusing exclusively on those that are computationally intensive.

It is hard to determine if a benchmark is actually useful. As previously noted, generating functional benchmarks is relatively straightforward, but verifying that the output is useful and relevant is one of the most challenging aspects. The only definitive way to validate this qualitative dimension would involve manual evaluation by a domain expert, however this approach wold defeat the purpose of automating the benchmark generation pipeline. In this study, the problem was addressed from the perspectives of meaningfulness and the detection of artificially injected slowdowns. A possible future step would be to verify whether the use case modeled by the generated benchmark accurately reflects the real-world use case of the method. This aspect is not covered in this work, as highlighted by the comparison of the `Flowable.groupBy()` method against the RxJava baseline in RQ3. In that specific case, although the microbenchmark compiled, executed successfully and was classified as meaningful, the represented scenario failed to reflect the typical usage of the method. Since these patterns cannot be deduced from the method signature alone, an improvement would be to provide the system with reference benchmarks written by domain experts, illustrating standard usage patterns for the library's most representative methods.

Feedback loops are a necessity. The results from RQ1 and RQ2 demonstrate that integrating feedback loops into the pipeline enabled the successful generation of 21.1% to 38.4% of all attempted benchmarks, for projects of moderate complexity. Even in the most complex projects, such as Log4j2 and Kafka, feedback loops remain essential, accounting for between 20% and 50% of all successfully generated benchmarks respectively. This leads to the conclusion that generation errors, whether caused by compilation or execution failures, or by a failure to meet quality requirements, are not isolated anomalies but expected scenarios, which must therefore be properly addressed through a recovery mechanism, such as the feedback loops in this system.

6.3 Limitations

The limitations of the system primarily concern its generalizability. In the evaluation phase, the analysis was restricted to three open-source projects, focusing exclusively on Java-based JMH microbenchmarks. Future studies will be essential to extend the system's applicability to other frameworks and programming languages, as well as to evaluate its performance across a broader selection of projects in terms of volume, domain and complexity. Furthermore, the evaluation of whether the criteria were met during the meaningfulness assessment was conducted by a single individual, which may introduce a degree of subjectivity. To enhance the robustness of this phase and ensure greater objectivity, future work should involve multiple evaluators.

A further limitation is the lack of a step evaluating whether the use case modeled by the generated benchmark accurately reflects the target method's real-world usage, as already discussed in the second implication of Chapter 6.2. Since these patterns cannot be deduced from the method signature alone, an improvement would be to provide the system with reference benchmarks written by domain experts, illustrating standard usage patterns for the library's most representative methods.

Moreover, the quality of the Planning phase has not been extensively explored. Yet, it represents a fundamental stage of the entire pipeline, as it should guarantee a minimum code coverage and ensure that the most critical and relevant methods within the source code are selected.

7. Conclusions and Future Work

The system presented in this work represents an initial approach to the automatic generation of JMH microbenchmarks for Java code using agentic steps and LLM calls. The proposed multi-agent pipeline begins with the selection of target methods, followed by two steps dedicated to actual code generation. Once an initial version of the benchmark is produced, it undergoes several validation checks and any failure triggers feedback loops that redirect the workflow to a previous step for further attempts. Evaluation of the results showed that the system correctly generates between 83.9% and 95.8% of all attempted benchmarks, and that feedback loops are responsible for 21.1% to 38.4% of them, making them an essential phase of the generation process. However, the success rate drops for more complex projects that require elaborated setups and configurations, falling to between 27.8% and 69.2%.

Even in these challenging scenarios, the feedback loops remain crucial, accounting for between 20% and 50% of all successfully generated benchmarks. Furthermore, the practical utility of the generated benchmarks was assessed through a dual analysis: a static evaluation of their meaningfulness, measured against predefined code criteria, and a dynamic latency injection experiment, where artificial slowdowns were introduced into the source code to verify if micobenchmarks could actually detect performance regressions. Excluding the most complex projects, the results reveal that most microbenchmarks are meaningful (between 68% and 96%) and that, with a single exception, all successfully identify performance degradation. In terms of time and cost, the generation of a single microbenchmark requires between 2.3 and 3.8 minutes with a worst-case cost of approximately $3 per hour (yielding 24 benchmarks), significantly lower than manual development.

Despite these positive outcomes, further studies are required to accurately determine whether the generated benchmarks align with what developers are genuinely interested in testing, such as widely used methods, critical hotspots and realistic use cases. To achieve this, it would be desirable to provide the LLM with a broader context, such as an example of typical method usage, method setup information not directly inferrable from its signature, or even typical pitfalls to avoid. Furthermore, to extend the system's validity, future research could consider other benchmarking frameworks and programming languages.

List of Figures

2.1 JMH execution flow [Source: [Laa+20]] 4
2.2 ReAct workflow diagram [Source: [Gut25]] 9

4.1 Pipeline of the system. The blue steps represent agents that use tools with reasoning capabilities (ReAct), the red steps represent agents that use simple LLMs calls and the green step are deterministic logic. 15
4.2 Agent types 16

5.1 ReAct Paradigm 27

6.1 PassRate RxJava 34
6.2 PassRate t-digest 35
6.3 PassRate Streamalg 35
6.4 PassRate Log4j2 36
6.5 PassRate Kafka 36
6.6 Percentages of methods correctly generated at every attempt over the total number of methods correctly generated, for each project (failed generations are therefore not included). For RxJava, StreamAlg, t-digest and Log4j2, the numbers represented are the average between the 3 runs, whereas for Kafka they are the ones of its single run. 37
6.7 Distribution of the throughput degradation actually detected by each benchmark in the two injected slowdown cases: *slowdown_10%* (*bug_10%*) and *slowdown_error* (*bug_error%*). 54
6.8 Average Planning phase duration and average benchmark's generation time . . 56
6.9 Average tokens spent and estimated cost per benchmark 57
6.10 Token consumption percentage across pipeline phases, shown individually by project. 58

Bibliography

[Pef+07] Ken Peffers et al. "A design science research methodology for information systems research". In: *Journal of Management Information Systems* (2007). DOI: 10.2753/MIS0742-1222240302.

[KOR10] Michael Kuperberg, Fouad Omri, and Ralf Reussner. "Automated Benchmarking of Java APIs". In: *Software Engineering 2010 conference*. 2010. URL: https://www.researchgate.net/publication/221232286_Automated_Benchmarking_of_Java_APIs.

[FA11] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". In: *ESEC/FSE '11:19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011. DOI: 10.1145/2025113.2025179.

[FA13] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". In: *IEEE Transactions on Software Engineering* (2013). DOI: 10.1109/TSE.2012.14.

[RCB16] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. "Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding". In: *IEEE/ACM International Conference on Automated Software Engineering*. 2016. DOI: 10.1145/2970276.2970346.

[LB17] Philipp Leitner and Cor-Paul Bezemer. "An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects". In: *ICPE '17: 8th ACM/SPEC oInternational Conference on Performance Engineering*. 2017. DOI: 10.1145/3030207.3030213.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017. DOI: 10.48550/arXiv.1706.03762.

[LL18] Christoph Laaber and Philipp Leitner. "An evaluation of open-source software microbenchmark suites for continuous performance assessment". In: *ICSE '18: 40th International Conference on Software Engineering*. 2018. DOI: 10.1145/3196398.3196407.

[TPG18] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. "Synthesizing programs that expose performance bottlenecks". In: *CGO '18: 2018 International Symposium on Code Generation and Optimization*. 2018. DOI: 10.1145/3168830.

[Cos+19] Diego Costa et al. "What's wrong with my benchmark results? Studying bad practices in JMH benchmarks". In: *IEEE Transactions on Software Engineering* (2019). DOI: https://doi.org/10.1109/TSE.2019.2925345.

[DCS20] Zishuo Ding, Jinfu Chen, and Weiyi Shang. "Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet?" In: *ICSE '20: ACM/IEEE 42nd International Conference on Software Engineering*. 2020. DOI: 10.1145/3377811.3380351.

[Laa+20] Christoph Laaber et al. “Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality”. In: *ESEC/FSE*. 2020. DOI: 10.1145/3368089.3409683.

[LGL21] Christoph Laaber, Harald C. Gall, and Philipp Leitner. “Applying test case prioritization to software microbenchmarks”. In: *Empirical Software Engineering* (2021). DOI: 10.1007/s10664-021-10037-x.

[Jan+23] Mostafa Jangali et al. “Automated generation and evaluation of jmh microbenchmark suites from unit tests”. In: *IEEE Transactions on Software Engineering* (2023). DOI: 10.1109/TSE.2022.3188005.

[Yao+23] Shunyu Yao et al. “ReAct: synergizing reasoning and acting in language models”. In: *ICLR 2023*. 2023. URL: https://arxiv.org/pdf/2210.03629.

[Che+24] Yinghao Chen et al. “ChatUniTest: A Framework for LLM-Based Test Generation”. In: *32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24)*, 2024. DOI: 10.1145/3663529.3663801.

[Nun+24] Ana Nunez et al. “AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing”. In: *38th Conference on Neural Information Processing Systems* (2024). DOI: 10.48550/arXiv.2409.10737.

[Qia+24] Chen Qian et al. “ChatDev: Communicative Agents for Software Development”. In: *Annual Meeting of the Association for Computational Linguistics* (2024). DOI: 10.18653/v1/2024.acl-long.810.

[YFY24] Juyeon Yoon, Robert Feldt, and Shin Yoo. “Intent-driven mobile GUI testing with autonomous Large Language Model agents”. In: *IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2024. DOI: 10.1109/ICST60714.2024.00020.

[PB25] Juan Altmayer Pizzorno and Emery D. Berger. “CoverUp: Coverage-Guided LLM-Based Test Generation”. In: *ACM International Conference on the Foundations of Software Engineering*. 2025. DOI: 10.48550/arXiv.2403.16218.

[Zhi+25] Chen Zhi et al. “Chatperftest: A framework for llm-based jmh microbenchmark generation”. In: *Social Science Research Network* (2025). DOI: 10.2139/ssrn.5081001.

Sitography

[Man18] Manikanta. *T-Digest: An interesting datastructure to estimate quantiles accurately.* 2018. URL: https://medium.com/@mani./t-digest-an-interesting-datastructure-to-estimate-quantiles-accurately-b99a50eaf4f7.

[Str21] Cole Stryker. *What are large language models (LLMs)?* 2021. URL: https://www.ibm.com/think/topics/large-language-models.

[Stö23] Andreas Stöffelbauer. *How Large Language Models work.* 2023. URL: https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f.

[Gut24a] Anna Gutowska. *What are AI agents?* 2024. URL: https://www.ibm.com/think/topics/ai-agents.

[Gut24b] Anna Gutowska. *What is a multi-agent system?* 2024. URL: https://www.ibm.com/think/topics/multiagent-system.

[Obr24] Alexander Obregon. *Introduction to Java Microbenchmarking with JMH (Java Microbenchmark Harness).* 2024. URL: https://medium.com/@AlexanderObregon/introduction-to-java-microbenchmarking-with-jmh-java-microbenchmark-harness-55af74b2fd38.

[Ans25] Salauddin Ansari. *Understanding the JVM: What It Is and How It Works.* 2025. URL: https://medium.com/@ByteOfJava/understanding-the-jvm-what-it-is-and-how-it-works-5c647eb5a659.

[Ber25] Dave Bergmann. *What is a ReAct agent?* 2025. URL: https://www.ibm.com/think/topics/react-agent#1793360183.

[Clo25] Google Cloud. *What are AI agents?* 2025. URL: https://cloud.google.com/discover/what-are-ai-agents.

[Gee25] GeeksForGeeks. *How JVM Works - JVM Architecture.* 2025. URL: https://www.geeksforgeeks.org/java/how-jvm-works-jvm-architecture/.

[Gut25] Anna Gutowska. *Deploy an IT support LangGraph ReAct agent with IBM Granite on watsonx.ai.* 2025. URL: https://www.ibm.com/think/tutorials/deploy-langgraph-react-agent-manage-it-support-tickets-watsonx-ai#1793360182.

[Bib] Aggelos Biboudis. *streamalg - github repository.* URL: https://github.com/biboudis/streamalg.

[Clo] Google Cloud. *Guide on multi-agent systems (MAS).* URL: https://cloud.google.com/discover/what-is-a-multi-agent-system.

[Dun] Ted Dunning. *t-digest - github repository.* URL: https://github.com/tdunning/t-digest.

[Fou] Apache Software Foundation. *log4j2 - github repository.* URL: https://github.com/apache/logging-log4j2.

[JR] Neha Narkhede Jay Kreps and Jun Rao. *kafka - github repository.* URL: https://github.com/apache/kafka.

[Mic] Microsoft. *RxJava - github repository*. URL: https://github.com/reactivex/rxjava.

[Opea] OpenAI. *OpenAI - gpt-5.4*. URL: https://openai.com/it-IT/index/introducing-gpt-5-4/.

[Opeb] OpenJDK. *Java Microbenchmark Harness (JMH)*. URL: https://github.com/openjdk/jmh.

[Ora] Oracle. *Oracle JVM Just-in-Time Compiler (JIT)*. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdev/Oracle-JVM-JIT.html#GUID-7EDD2308-CC98-454C-8440-DC97C9A7943C.