Profiling-Guided Bayesian Optimization of JVM Configurations

ABDELRAHMAN BAZ, The University of Texas at Austin, USA
WING LAM, George Mason University, USA
AUGUST SHI, The University of Texas at Austin, USA

Regression testing is essential for maintaining software quality but often incurs substantial time costs. While regression testing time can often be reduced by selectively running fewer tests, prior work has demonstrated that tuning Java Virtual Machine (JVM) configuration flags can also reduce testing time in Java projects, even while running all tests and preserving original testing outcomes. However, finding effective flag combinations remains challenging due to the vast configuration space and complex interactions between flags. Random search and direct modeling approaches that map flag configurations to testing time have shown limited effectiveness in navigating this complex optimization landscape.

We present PROBO (**PRO**filing-**G**uided **B**ayesian **O**ptimization), an iterative approach that leverages JVM runtime metrics (e.g., garbage collection frequency, just-in-time (JIT) compilation rates, and memory allocation) to guide Bayesian optimization for testing time reduction. Unlike prior work using Bayesian inference that directly models the relationship from flag configurations to testing time, PROBO decomposes the prediction problem through observable runtime behaviors: a metrics model predicts how flag configurations affect runtime metrics, and a performance model predicts how those metrics affect testing time. PROBO collects 44 runtime metrics using a profiler during test execution, and propagates feature importance scores through both models to identify which flags most strongly influence testing time-predictive metrics. Finally, PROBO generates candidate flag configurations through three complementary strategies guided by expected testing time improvement.

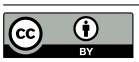
We evaluate PROBO on 16 open-source Java projects, comparing against random search and BOCA (a Bayesian optimization baseline). PROBO achieves an average testing time reduction of 10.7% across all projects when evaluating 20 configurations per project, outperforming random search (5.8%) by 1.85× and BOCA (4.3%) by 2.49×. PROBO successfully generates configurations that substantially reduce testing time for all 16 projects, with reductions ranging up to 27.4%. With a one-hour time budget for search, PROBO maintains its advantage with 8.0% average reduction, demonstrating practical applicability. PROBO-generated configurations remain effective across software evolution, maintaining 8.8% average reduction over an average of 88 future commits per project. Our analysis reveals that metrics related to JIT compilation, particularly Total Compilation Rate (methods compiled per second) and C1 Compilation Rate (first-tier JIT compilation rate), are the strongest predictors of testing time, accounting for 67.2% of consistently important metrics across projects.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: JVM configurations, Bayesian optimization, testing speedup, regression testing

ACM Reference Format:
Abdelrahman Baz, Wing Lam, and August Shi. 2026. Profiling-Guided Bayesian Optimization of JVM Configurations. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA162 (October 2026), 23 pages. https://doi.org/10.1145/3832253

Authors' Contact Information: Abdelrahman Baz, The University of Texas at Austin, Austin, TX, USA, ambaz@utexas.edu; Wing Lam, George Mason University, Fairfax, VA, USA, winglam@gmu.edu; August Shi, The University of Texas at Austin, Austin, TX, USA, august@utexas.edu.



ACM 2994-970X/2026/10-ARTISSTA162
https://doi.org/10.1145/3832253

1 Introduction

Regression testing is a fundamental practice in software development and continuous integration, ensuring that code changes do not introduce new faults [57]. However, regression testing often imposes substantial time costs on development workflows, creating bottlenecks that slow down release cycles and reduce developer productivity [15, 24, 30, 33, 39, 40, 51]. To help reduce this time cost, various techniques, such as regression test selection [25, 26, 28, 34, 39, 43, 44, 50, 53, 58] and test-suite reduction [22, 27, 31, 37, 45, 48, 49, 52, 59], have been proposed to run only a subset of tests. However, running fewer tests incurs the risk of missing detection of faults [49, 60].

For Java projects, code is executed through the Java Virtual Machine (JVM), including tests. The JVM provides hundreds of configuration flags that control runtime behavior, including garbage collection policies, just-in-time (JIT) compilation strategies, memory management, and threading [3]. In our prior work [13], we showed how constructing specific JVM configurations via flag combinations can reduce test suite execution time, by up to 43.9%, without requiring any changes to application or test code, or the set of tests that are run. This non-intrusive optimization is particularly attractive, because it can reduce the cost of test runs without the risk of missing faults from running fewer tests.

However, finding effective JVM configurations remains challenging. The vast search space, which consists of hundreds of binary and multi-valued configuration flags, makes exhaustive exploration impractical. In our work, even after we restrict the flag search space to only the ones that we manually determine could affect execution time by inspecting the official Java documentation [3] (Section 4.3), the search space still consists of 72 flags, representing up to $2^{72} \approx 4.7 \times 10^{21}$ configurations. Moreover, the relationship between flags and testing time is complex and non-linear, with intricate interactions between different flags [13]. Generating configurations by randomly combining different flags can occasionally reveal configurations that reduce testing time. However, prior work found that beneficial configurations are rare within the vast space of possible flag combinations, meaning that random search may need to evaluate many configurations before finding one that yields meaningful improvements [13]. This inefficiency motivates the need for more intelligent search strategies. Previous work also explored using machine learning approaches to model the direct relationship between flag settings and program execution time using various techniques, such as Bayesian optimization [41]. For instance, BOCA [21], a Bayesian optimization approach originally designed for compiler autotuning, uses Random Forest models to predict the execution time of programs compiled using a given configuration and guides the search toward promising regions of the configuration space. We previously adapted this approach [13] for the problem of tuning JVM configuration flags to reduce testing time, but found that such direct modeling achieves only limited success, often performing worse than randomly generating configurations. These results suggest that the opaque, end-to-end mapping from flags to testing time when running in a JVM may be too complex for standard machine learning models to capture effectively.

Our key insight is that incorporating intermediate JVM runtime metrics—quantitative measurements of JVM behavior during execution, such as garbage collection frequency, JIT compilation rates, and memory allocation patterns—can substantially improve machine learning-guided optimizations. Rather than treating the relationship between flags and test execution time as a black box, we hypothesize that explicitly modeling how flags influence observable runtime behavior can help the optimization process understand *how* flags affect performance and, consequently, testing time. The JVM exposes rich runtime profiling data through tools like Java Flight Recorder (JFR) [4], including detailed metrics about garbage collection, compilation activity, memory usage, CPU utilization, and threading. By capturing these metrics during test execution and using them as intermediate features in a prediction model, we can decompose the complex optimization problem

into more manageable sub-problems: predicting how flag combinations influence runtime metrics, and then predicting how those metrics influence testing time.

We present PROBO (**PRO**filing-**G**uided **B**ayesian **O**ptimization), an iterative approach that leverages JVM runtime metrics to guide Bayesian optimization for reducing testing time. PROBO decomposes the optimization problem through observable runtime behaviors: a metrics model predicts how flag configurations affect runtime metrics, and a performance model predicts how those metrics affect testing time. This profiling-guided design captures the relationship between flags and testing time through measurable JVM behavior, identifying flags with genuine performance impact. PROBO propagates feature importance scores through both models to identify which flags most strongly influence the runtime metrics that are themselves the most predictive of testing time. Using these importance scores, PROBO generates candidate configurations through three complementary strategies and selects the most promising candidates based on the expected improvement acquisition function [16], which balances exploring uncertain regions of the configuration space with exploiting known promising regions. PROBO uses the metrics model and performance model to predict the testing times of each candidate configuration without running them, only evaluating the top candidates (those with highest expected improvement) to collect the actual testing time and profiled runtime metrics associated with running tests using those configurations. This additional information is fed back into the model training set, which is used to retrain and refine the models for another iteration. Through iterative evaluation and model refinement, PROBO efficiently navigates the vast configuration space to find better configurations.

We evaluate PROBO on 16 real-world open-source Java projects, comparing its performance against Random Generation (randomly generating configurations) and BOCA [21], a representative Bayesian optimization baseline that directly predicts testing time from flag combinations. Our results demonstrate that PROBO consistently outperforms both approaches. PROBO achieves an average testing time reduction of 10.7% across all projects, compared to 5.8% for Random Generation and 4.3% for BOCA. BOCA's weaker performance suggests that directly modeling the relationship from flag configurations to testing time without intermediate metrics can lead the search toward spurious correlations rather than truly beneficial configurations. These results represent a 1.85× improvement over Random Generation and a 2.49× improvement over BOCA. Notably, when allowed to evaluate 20 configurations per project, PROBO successfully generates configurations that reduce testing time for all 16 projects, with reductions ranging up to 27.4%. Even under a constrained one-hour time budget, which typically allows evaluating fewer configurations, PROBO maintains strong performance with an average testing time reduction of 8.0%, demonstrating practical applicability in real development workflows. Our analysis of feature importance across projects reveals that JIT compilation metrics, particularly Total Compilation Rate (methods compiled per second) and C1 Compilation Rate (first-tier JIT compilation rate), are the strongest predictors of testing time, accounting for 67.2% of consistently important metrics across projects. Furthermore, PROBO-generated configurations remain effective across software evolution, maintaining an average reduction of 8.8% over an average of 88 future commits per project.

This paper makes the following contributions:

- **Approach**: We present PROBO, a profiling-guided Bayesian optimization approach that leverages JVM runtime metrics to guide the search for configurations that reduce testing time. PROBO decomposes prediction through observable runtime behaviors, identifying flags with genuine performance impact.
- **Evaluation**: We evaluate PROBO on 16 open-source Java projects, demonstrating 1.85× and 2.49× improvements over Random Generation and BOCA, respectively, with configurations remaining effective across software evolution.

- **Analysis**: We analyze the relationship between JVM runtime metrics and testing time across multiple projects, identifying JIT compilation metrics as the strongest predictors and providing insights into which runtime behaviors most substantially impact testing time.

2 Background

2.1 JVM Flags and Effects on Testing Time

The Java Virtual Machine (JVM) provides hundreds of configuration flags that control various aspects of runtime behavior, including garbage collection policies, just-in-time (JIT) compilation strategies, memory management, and threading [3]. We define a *JVM configuration* to be a combination of JVM flags and values set for those flags, used to configure a JVM for running. For example, the following configuration combines four flags that modify JIT compilation, garbage collection, and memory management:

```
java -XX:TieredStopAtLevel=1 -XX:+UseParallelGC -XX:+UseCompressedOops -Xms512m
```

where the four flags, in order, limit JIT compilation to the first tier, so that methods are compiled by the fast but lightly optimizing C1 compiler rather than the slower, more aggressive C2 compiler; use the parallel garbage collector, which reclaims memory using multiple threads; enable compressed object pointers, which store object references as 32-bit offsets to reduce heap footprint; and set the initial heap size to 512MB. We refer to the *default configuration* as whatever configuration a project already uses to run its tests, e.g., some projects limit tests to run using a maximum amount of heap memory. For most projects, the default configuration is simply running the JVM without any additional flags.

Formally, a configuration can be represented as a binary vector, where each element indicates whether a specific flag is enabled (1) or disabled (0). We can encode flags that accept non-binary values (e.g., `-XX:TieredStopAtLevel`) as multiple binary options, where each distinct value setting is treated as a separate flag in the flag space.

JVM flags can be categorized into several groups based on their functionality [3]. *Non-standard flags* control general-purpose JVM options, such as maximum heap size (`-Xmx`) and class data sharing (`-Xshare`). *Advanced runtime flags* modify JVM execution behavior, including options like compressed pointers (`-XX:+UseCompressedOops`) and performance data collection (`-XX:-UsePerfData`). *Advanced JIT compiler flags* govern the just-in-time compilation and optimization process, with options, such as tiered compilation levels (`-XX:TieredStopAtLevel`) and escape analysis (`-XX:-DoEscape Analysis`). *Advanced garbage collection flags* control memory reclamation policies, including collector selection (`-XX:+UseSerialGC`, `-XX:+UseParallelGC`) and various tuning parameters, such as collection thread configuration (`-XX:ParallelGCThreads`, `-XX:ConcGCThreads`).

The impact of these flags on testing time is complex and often non-linear due to interactions between flags. For instance, disabling profiling for JIT compilation (`-XX:-C1ProfileCalls`) can reduce overhead in scenarios where tests execute quickly enough that profiling costs outweigh optimization benefits. Similarly, limiting JIT compilation to lower optimization levels (`-XX:TieredStopAtLevel=1`) can improve performance for short-running tests by avoiding the profiling overhead required for higher-level optimizations. Garbage collection policies can have substantial effects on testing time, particularly for test suites that stress memory allocation or exhibit specific memory usage patterns [13]. Finding the right combinations of JVM flags to form an effective JVM configuration to reduce testing time is challenging. Prior work shows that configurations yielding testing time improvements are rare within the vast space of possible flag combinations, making random search surprisingly competitive against more sophisticated machine learning-based approaches [13].

2.2 Bayesian Optimization and BOCA

Bayesian Optimization. Bayesian Optimization (BO) is a sequential model-based optimization technique designed for expensive black-box functions where each evaluation is costly [16, 41]. BO is particularly effective when function evaluations are time-consuming or resource-intensive, making it impractical to exhaustively search the input space.

BO operates by maintaining a surrogate model that approximates the relationship between input parameters and the objective function being optimized. Common surrogate models include Gaussian Processes [16, 42] and Random Forests [18], which can model complex, non-linear relationships and provide uncertainty estimates. The BO process follows an iterative cycle: (1) start with an initial set of observed configurations and their performance values, (2) train a surrogate model to approximate the objective function, (3) analyze the surrogate to identify influential parameters and regions of high uncertainty, (4) generate candidate configurations and use an acquisition function to select the most promising one, (5) evaluate this candidate by running the actual system, (6) update the training data with the new observation, and (7) repeat until a termination criterion is met (e.g., time budget or iteration limit). Common acquisition functions include *expected improvement* (EI) [16], which balances *exploration* (trying inputs in uncertain regions of the search space) with *exploitation* (trying inputs likely to be near-optimal based on current knowledge). EI scores each candidate by how much improvement it is expected to yield over the best objective value observed so far. The expectation is taken over by the surrogate model's prediction for that candidate, which supplies both a predicted value and an estimate of how uncertain that prediction is. A candidate therefore scores highly either because its predicted value improves on the best result so far, or because its prediction is uncertain enough that it may still turn out substantially better.

BO has been successfully applied to various optimization problems, including hyperparameter tuning for machine learning models [17], database system configuration [55], and compiler flag optimization [21]. BO is particularly well-suited for configuration tuning because it handles expensive evaluations efficiently—each configuration may require minutes to benchmark—and works with black-box objectives, where the relationship between configuration parameters and performance is unknown. Unlike grid search or random search, BO learns from previous evaluations to make informed decisions about which configurations to try next.

BOCA. BOCA is a BO-based compiler autotuning approach [21] that exemplifies how BO can be adapted for configuration optimization. BOCA was designed to find a combination of flags for C/C++ compilers, such that the performance of the generated binary is optimized.

Figure 1 shows an overview of BOCA. BOCA first randomly generates an initial set of configurations and then measures the running time of the program compiled with each configuration (1). It then trains a Random Forest (RF) [18] model using the collected data, where the input features are the configurations and the output is the running time of the binary on a benchmark (2). BOCA identifies the most important flags using the RF model's feature importance scores (3).

BOCA then generates candidate configurations through systematic exploration (4). It first creates 2^k base configurations by enumerating all possible combinations of the top-k flags with the highest importance scores. For each base configuration, BOCA generates r neighboring configurations by randomly exploring different settings of the remaining less-impactful flags. The neighborhood size r is dynamically adjusted to balance exploration and exploitation as the search progresses:

$$r = r_0 \cdot \exp\left(-\frac{\max(0, n - n_0)^2}{2\sigma^2}\right) \tag{1}$$

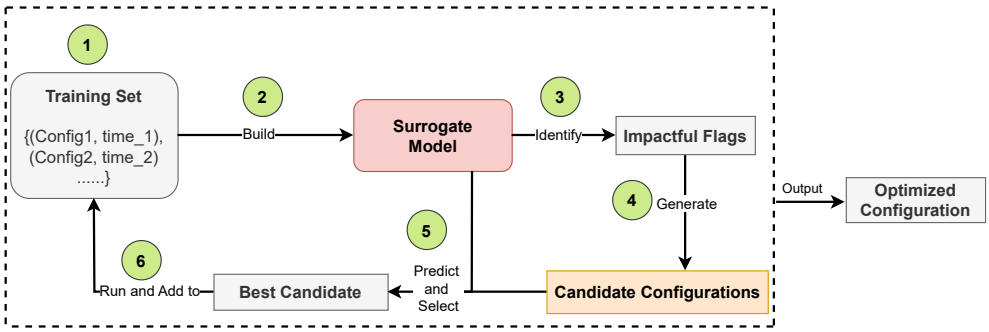


Fig. 1. BOCA overview.

where n is the number of configurations evaluated so far, r_0 is the initial neighborhood size, n_0 is the offset before decay begins, and σ controls the decay rate, computed as $\sigma^2 = -scale^2/(2\log(decay))$; *scale* and *decay* are further configurable variables. This decay strategy ensures larger neighborhoods (more exploration) in early iterations and smaller neighborhoods (more exploitation) as the search converges.

Finally, BOCA selects the most promising candidate based on the EI acquisition function [16] (5) and evaluates it by compiling and running the program. The new configuration and its running time are added to the training data (6), and the RF model is retrained in the next iteration.

BOCA's approach can be adapted to JVM configuration tuning by treating JVM flags as input features and testing time as the output to be minimized. However, prior work has found that this direct adaptation achieves limited improvements in reducing testing time [13]. The key limitation is that BOCA learns a direct, opaque mapping from flags to performance outcomes without understanding the underlying mechanisms through which flags influence testing time. This black-box approach makes it difficult to distinguish between truly important flags and those that correlate spuriously with performance in the limited training data.

3 PROBO

We present PROBO (**PRO**filing-**G**uided **B**ayesian **O**ptimization), an iterative approach for generating JVM configurations that minimize testing time. Unlike BOCA [21], which directly models the relationship between flags and testing time, PROBO incorporates JVM runtime metrics collected during test execution to guide the search. Figure 2 shows an overview of PROBO's workflow, which consists of six main steps executed iteratively.

Step 1: Initial Exploration and Metrics Collection. Similar to BOCA, in the first iteration PROBO generates an initial set of random flag combinations, along with the default configuration (1). For each configuration, PROBO runs the project's test suite 5 times to measure testing time. PROBO then performs an additional profiled run to collect runtime metrics using Java Flight Recorder (JFR), a low-overhead profiling tool built into the JDK [4]. JFR imposes minimal overhead (typically <2%), while capturing detailed runtime behavior [4]. PROBO configures JFR to record events related to garbage collection, JIT compilation, memory allocation, CPU usage, threading, and I/O operations, producing 44 metrics across seven categories (detailed in Section 3.1).

Step 2: Profiling-Guided Model Training. PROBO builds a metrics model and a performance model to capture the relationship: flags → metrics → testing time (Figure 3) (2, Section 3.1). This profiling-guided design enables PROBO to understand *how* flags affect testing time through

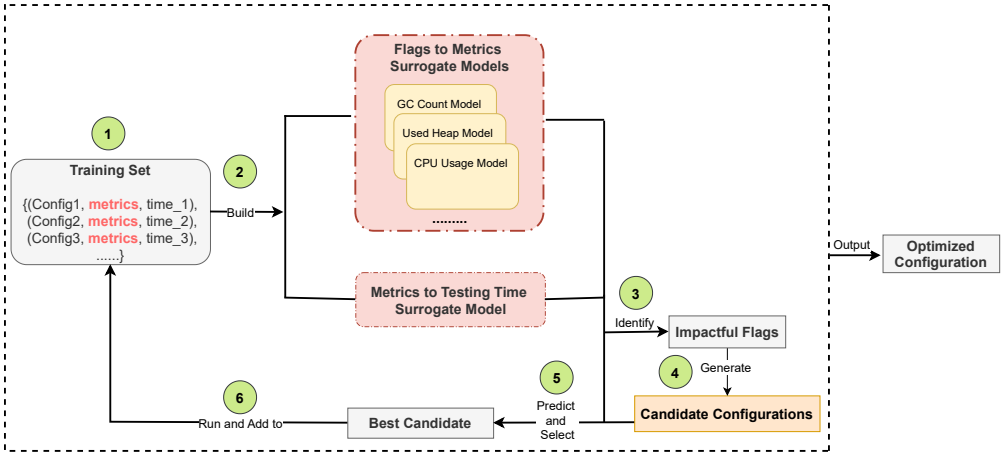


Fig. 2. PROBO overview.

their influence on observable runtime metrics, rather than learning a direct opaque mapping. The explicit modeling of runtime metrics as intermediate predictors allows the model to capture the mechanisms by which flags impact performance.

Step 3: Flag Importance Calculation. PROBO computes the importance of each flag by propagating feature importance scores through both models (3, Section 3.2). This step identifies which flags most strongly influence testing time-predictive metrics, enabling efficient candidate generation.

Step 4: Candidate Generation. Using the flag importance scores, PROBO generates a large pool of candidate configurations through three complementary strategies (4, Section 3.3). These strategies ensure diverse exploration of the flag space while prioritizing flags identified as influential.

Step 5: Candidate Selection. PROBO ranks all candidates using the EI acquisition function [16] and selects the top configurations based on EI (5, Section 3.3).

Step 6: Evaluation and Iteration. PROBO evaluates the selected configurations by running the test suite 5 times per each configuration to measure actual testing time. An additional profiled run collects runtime metrics. PROBO then adds the evaluated configuration, its testing time, and collected metrics to the training data (6).

PROBO repeats Steps 2–6 for subsequent iterations, continuously retraining the model with accumulating data. PROBO runs up to a set number of iterations or until a preset time budget is exhausted (e.g., 1 hour), then reports the configuration that achieved the greatest reduction in testing time.

3.1 Metrics Collection and Profiling-Guided Prediction

PROBO extracts 44 runtime metrics from JFR output (18 are metrics we indirectly derive from explicitly reported metrics, which we detail in our artifact [5]). We classify these metrics into seven categories:

- **Memory** (9 metrics): Max/min/mean committed and used heap sizes, heap usage ratios. These metrics describe how much heap the JVM commits and how much of it holds data while tests run.
- **CPU** (2 metrics): Mean user and system CPU percentages. These metrics measure the share of processor time spent in user mode and in kernel mode.

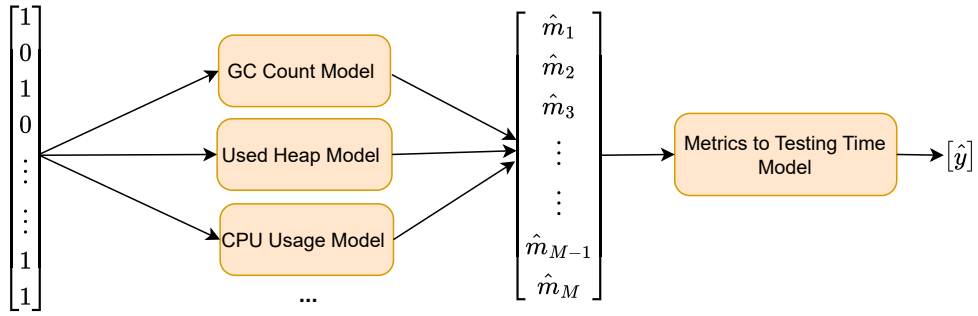


Fig. 3. Profiling-guided prediction showing how PROBO predicts testing time for candidate configurations. The metrics models predict metric values from flags; the performance model predicts testing time from metrics.

- **Threading** (9 metrics): Max/min/mean active threads, daemon threads, thread sleep duration, and thread contention rate. These metrics describe how many threads run, how long they sleep, and how often they block on contended locks.
- **Compilation** (12 metrics): Classes loaded, classloading duration, methods compiled, compilation duration, C1/C2/total compilation rates, compilation success rate, hot method recompilation rate, compilation burst intensity, code cache pressure, and C1/C2 ratio. These metrics capture how much work the JIT compiler and the class loader perform.
- **Garbage Collection (GC)** (6 metrics): GC count, pause time, frequency, pause variance, timing regularity, and allocation rate. These metrics capture how often collection runs, how long it pauses the application, and how quickly tests allocate memory.
- **I/O** (4 metrics): File read/write amounts and socket read/write amounts. These metrics measure how much data the tests read and write through files and sockets.
- **Other** (2 metrics): TLAB efficiency and safepoint frequency. TLAB efficiency reflects allocation served from per-thread buffers rather than the shared heap, and safepoint frequency counts how often the JVM brings all threads to a stop to perform internal operations.

PROBO performs correlation-based feature selection to retain only metrics with meaningful relationships to testing time. Using Kendall-Tau correlation [32], PROBO removes metrics with weak correlations ($|\tau| < 0.3$ [23]) to testing time, as well as metrics that remain constant across all configurations. This filtering reduces the 44 initial metrics to M selected metrics, where M varies per test suite depending on which metrics prove informative.

PROBO then constructs two models (Figure 3):

Metrics model: Flags → Metrics. For each of the M selected metrics, PROBO trains a separate Random Forest regressor that models how flag configurations influence that metric. The input is a binary matrix $X_{\text{flags}} \in \{0,1\}^{n \times f}$, where n is the number of evaluated configurations and f is the total number of flags in the search space. Each row represents one configuration; each column indicates whether a specific flag is enabled (1) or disabled (0). The output for each model is the predicted value of a single metric m_j under a given configuration. Thus, the metrics model comprises M independent Random Forest models, one per metric.

Performance model: Metrics → Testing Time. PROBO trains a single Random Forest regressor that predicts testing time from the M selected metrics. The input is the metrics matrix $X_{\text{metrics}} \in$

$\mathbb{R}^{n \times M}$, where each row contains the M metric values observed for a configuration c. Before training, metrics are standardized to zero mean and unit variance to ensure all metrics contribute equally regardless of their original measurement scales. The output is the predicted testing time $\hat{y}(c)$.

During candidate selection (Step 5), PROBO uses the metrics models and performance model to predict testing time for each candidate: the metrics models first predict the M metric values from the candidate's flag vector, then the performance model predicts testing time from those predicted metrics (Figure 3).

3.2 Flag Importance Calculation

PROBO computes flag importance by propagating feature importance scores through both models:
Step 1: Extract Metric Importance. Random Forest models provide built-in feature importance scores based on how much each feature reduces impurity when used for splitting [18]. Similar to how BOCA ranks flags [21], PROBO extracts these scores from the performance model ($\text{RF}_{\text{metrics}\to\text{testtime}}$), obtaining a score $I_M(m_j)$ for each metric m_j that indicates how predictive that metric is for testing time.
Step 2: Aggregate Flag Importance Across Metrics. For each flag f_i, PROBO aggregates its importance across all metrics using a weighted sum:

$$I_F(f_i) = \frac{1}{Z} \sum_{j=1}^{M} \text{FI}_{f_i \to m_j} \times I_M(m_j) \tag{2}$$

where:

- $\text{FI}_{f_i \to m_j}$ is the feature importance of flag f_i in the metrics model $\text{RF}_{\text{flags}\to m_j}$
- $I_M(m_j)$ is the importance of metric m_j for testing time prediction (from Step 1)
- M is the number of selected metrics
- Z is the sum of all flag importances (normalization constant) ensuring $\sum_{i=1}^{f} I_F(f_i) = 1$

This equation captures the intuition that a flag is important if it strongly influences metrics that themselves strongly predict testing time. For example, if a flag significantly affects GC pause time (high $\text{FI}_{f_i \to \text{GC pause}}$) and GC pause time is highly predictive of testing time (high $I_M(\text{GC pause})$), then that flag receives a high importance score. Conversely, a flag that does not influence any retained metric receives a low importance score.

3.3 Candidate Generation and Selection

Using the computed flag importance scores, PROBO generates a diverse pool of candidate configurations through three complementary strategies:
Strategy 1: Systematic Exploration. PROBO adapts BOCA's systematic exploration strategy [21], generating 2^k base configurations by enumerating all combinations of the top-k most important flags. For each base configuration, PROBO creates r neighborhood variants by randomly toggling remaining flags; r follows an exponential decay (Equation 1) that reduces the neighborhood size as more configurations are evaluated. Similar to BOCA, we set $k = 8$ and compute σ using $scale = 10$ and $decay = 0.5$. Given PROBO's smaller evaluation budget, we use a smaller initial neighborhood size ($r_0 = 30$) and offset ($n_0 = 10$).
Strategy 2: Weighted Random Sampling. PROBO generates an equal number of candidate configurations using weighted random sampling, drawing the number of flags to enable uniformly at random and then sampling that many flags with probability proportional to their importance scores $I_F(f_i)$. This strategy provides stochastic exploration that favors important flags, while allowing discovery of unexpected interactions.

Strategy 3: Pure Systematic Combinations. PROBO generates all non-empty subsets of the top-k important flags without neighborhood perturbation. This strategy ensures exhaustive exploration of high-impact flag combinations.

All strategies filter out previously evaluated configurations and validate each candidate against the flag space's dependency and conflict constraints (Section 4.3) to ensure only valid configurations are generated.

Candidate Selection. After generating candidates, PROBO uses the EI acquisition function to rank them. For each candidate configuration c (represented as a binary vector), PROBO predicts testing time $\hat{y}(c)$ using the metrics models and performance model (Figure 3):

(1) The metrics models predict metric values $\hat{m}_1(c), \ldots, \hat{m}_M(c)$ from the flag vector
(2) The performance model predicts testing time from the predicted metric values

PROBO selects the top $t = 3$ configurations with the highest EI for evaluation per iteration. Together with the size of the initial random set and the number of iterations, t determines PROBO's total evaluation budget (Section 4.4). Similar to BOCA, all of PROBO's search hyperparameters are configurable. During evaluation, PROBO verifies that each configuration preserves test correctness: all tests must produce the same outcomes (passing, failing, error, or skipped) as when running with the default configuration, because we cannot allow tests to behave differently even as we make them run faster. PROBO also enforces a timeout of twice the default testing time to prevent configurations that cause tests to hang or run indefinitely. If a configuration produces different test outcomes or exceeds the timeout, PROBO discards it and selects the next-best candidate from the ranked pool. This process continues until t valid configurations are successfully evaluated per iteration.

4 Experiment Setup

We address the following research questions:

- **RQ1**: How much reduction in testing time can PROBO achieve compared to baselines?
- **RQ2**: Which JVM runtime metrics are most predictive of testing time?
- **RQ3**: How stable are PROBO-generated configurations across software evolution?

We address RQ1 to establish whether PROBO can effectively identify JVM configurations that significantly reduce testing time compared to the baseline default configuration, evaluating both under a fixed number of configurations and under a practical time budget. We address RQ2 to understand which JVM runtime metrics are most predictive of testing time. This analysis provides insights into the mechanisms through which JVM configurations affect testing time, helping developers understand what to monitor and optimize, as well as runtime characteristics of their own tests. Finally, we address RQ3 to evaluate whether the benefits of optimized configurations persist across software evolution, which is essential for practical deployment in CI pipelines where code changes frequently.

4.1 Baselines

We compare PROBO against two baselines:

Random Generation. We randomly generate JVM configurations by sampling flags from the flag space, validating against dependencies and conflicts. In other words, we continuously run Step 1 of PROBO as described in Section 3, without any of the later steps to provide guidance for subsequent candidate configurations.

BOCA. We adapt BOCA [21], a Bayesian optimization approach originally designed for compiler flag tuning (Section 2.2), to the JVM configuration for optimizing testing time. BOCA uses a single

RF model that attempts to directly predict testing time from a given configuration (flags → testing time) and generates candidates based on flag importance scores from this model.

4.2 Subjects

Table 1. Dataset summary.

ID	Project	SHA	# Tests	Testing Time (s)
P1	commons-email	00cc321	191	5.1
P2	compile-testing	b6e19e9	231	5.5
P3	commons-csv	547c5a2	829	5.8
P4	logstash-logback-encoder	7044d87	462	5.9
P5	jsoup	afc38d8	1181	6.2
P6	commons-codec	e2cecc7	1339	10.1
P7	commons-jexl	f8725b2	964	14.5
P8	commons-collections	dc6d9f8	6379	17.3
P9	JSqlParser	0ec2600	1608	22.2
P10	commons-configuration	ed32b41	2873	25.8
P11	commons-imaging	c021adf	991	29.0
P12	asterisk-java	5c16184	358	30.8
P13	jackson-core	84a73dd	1476	36.0
P14	tabula-java	8bfa3ad	210	38.3
P15	commons-compress	74256e9	2400	46.1
P16	commons-net	26fbd9e	415	68.4

Our dataset consists of single-module Maven open-source projects used in our prior work [13] to improve testing time using different combinations of JVM flags. We select the projects that build using Java 11 and do not exhibit flaky test behavior [36] when running under the default configuration, leading to 16 projects. We target Java 11 because JFR became part of OpenJDK in this version [1], rather than a commercial feature. For each project, we take the same commit as used in prior work [13], except for project jackson-core where we use a more recent commit that successfully builds using Java 11.

We then generate different configurations for this commit. Table 1 lists for each project an ID, the commit we use, the number of tests, and the testing time under the default configuration.

4.3 Flag Space

We use the same flag space as in our prior work [13], adapting the flags values to Java 11 [3] as different Java versions have different default values of flags. This flag space includes four categories: (1) four *Non-Standard Flags*, which are general purpose flags, e.g., `Xmx` configures the max heap memory; (2) four *Advanced Runtime Flags*, which control the runtime behavior of the JVM, e.g., `UseCompressedOops` enables the use of compressed pointers; (3) 13 *Advanced JITCompiler Flags*, which control the just-in-time (JIT) compilation and optimizations, e.g., `TieredStopAtLevel=2` stops the JIT compilation at optimization level 2; and (4) 19 *Advanced Garbage Collection Flags*, which control JVM garbage collection (GC), e.g., `UseParallelGC` forces the JVM to use the parallel garbage collector.

We use the same process as in our prior work [13] to select the values of flags and to extract dependencies (i.e., enabling a flag requires another flag enabled as well) and conflicts (i.e., two

flags cannot both be enabled). Specifically, for binary flags, we include only the non-default value; for flags with enumerable values, we include all possible values; for numerical flags, we include half and double the default value. The flag space excludes flags unrelated to runtime performance (e.g., debug output). In the end, our flag space contains 72 JVM flags, 6 dependencies, and 59 conflicts in total. These dependencies and conflicts are general-purpose constraints derived from JVM documentation [3] and apply to any project.

4.4 Experimental Steps

To answer RQ1, we run PROBO and each baseline 5 times per project to generate optimal configurations under two settings. First, we evaluate a fixed budget of 20 configurations: for PROBO, each run evaluates 5 initial random configurations followed by 5 optimization iterations selecting 3 candidates each, plus the default configuration. We use the same budget of 20 configurations for baselines. Second, we evaluate under a one-hour time budget, where each approach can generate and evaluate as many configurations as possible within the time limit. We report the average reduction in testing time across five runs for each setting. For each configuration evaluated, we run the test suite 5 times to collect a distribution of testing times. We measure reduction as the difference between the average testing time in the default JVM configuration and the average testing time using the best configuration generated per approach, divided by the testing time in the default configuration (the higher the reduction, the better). To measure statistical significance, we use the Mann-Whitney U-test [38] to compare the distribution of testing times from each configuration against the distribution from the default configuration. We use a non-parametric test because it does not assume that testing times follow a normal distribution, which need not hold for measurements subject to runtime variance. We only report a reduction if it is positive and statistically significant ($p < 0.05$), and we show 0 otherwise, indicating that a developer should use the default JVM configuration for that project. To quantify the magnitude of each reduction, we also compute the Vargha-Delaney A_{12} effect size [56], classifying effects as negligible, small, medium, or large.

To answer RQ2, we analyze the feature importance scores from the model trained after the final iteration of each PROBO experiment run. We extract the top 5 most important metrics (as ranked by the RF model's feature importance scores from metrics to testing time) from the final iteration of all 80 models (16 projects × 5 runs per project). We identify a metric as *consistently important* for a project if it appears in the top 5 for at least 3 out of 5 runs. We report:

(1) the number of projects for which each metric is consistently important
(2) the average importance score for each metric across the projects where it is consistently important, and
(3) the distribution of consistently important metrics by category (Section 3.1).

We focus on the final iteration because, after evaluating 20 configurations, the model has ideally learned the most reliable relationships between metrics and testing time. This analysis reveals which JVM runtime metrics are most consistently predictive of testing time across different projects.

To answer RQ3, we apply the best configuration from RQ1 up to 100 subsequent commits with source code changes (after the original commit in Table 1) that successfully build with Java 11, measuring testing time reduction compared to running with the default configuration on each commit.

4.5 Running Environment

We run all our experiments in a Docker container built from an Ubuntu 22.04 Docker image, using Java 11, and Maven 3.8.3. We limit the container to use 2 CPUs and 8GB of RAM, simulating the

Table 2. Test time reduction percentage ("Red.") achieved by each approach under fixed iterations (20 configurations) and fixed time budget (1 hour). "Time" shows the average experiment duration in minutes. "# Conf." shows the average number of successfully evaluated configurations.

	20 Configurations						1-Hour Budget					
	Random		BOCA		PROBO		Random		BOCA		PROBO	
ID	Red.	Time	Red.	Time	Red.	Time	Red.	# Conf.	Red.	# Conf.	Red.	# Conf.
P1	15.5	24.4	9.8	32.6	14.3	40.4	17.5	48.8	11.0	35.4	16.0	34.8
P2	14.4	21.4	3.0	27.8	10.3	27.7	10.6	54.6	1.9	38.8	14.4	35.0
P3	12.7	25.6	1.7	32.9	19.7	29.6	10.2	48.6	9.4	34.2	5.9	32.4
P4	14.3	26.1	10.5	35.2	15.2	34.5	14.7	44.0	11.2	33.0	15.6	34.2
P5	8.8	93.0	6.3	42.3	18.8	44.3	2.1	4.0	16.2	14.4	9.5	4.0
P6	3.5	74.1	0.0	229.1	6.5	65.7	2.4	17.2	0.0	3.4	7.6	11.8
P7	0.0	158.5	13.2	68.5	27.4	90.8	0.0	18.0	11.1	11.0	21.1	14.4
P8	4.8	61.1	5.0	67.8	6.9	76.2	4.2	19.2	3.4	17.0	2.2	13.4
P9	0.0	254.1	1.2	155.5	7.2	387.2	2.8	9.0	5.6	7.6	6.4	6.8
P10	9.1	86.2	11.5	93.4	11.0	94.2	5.3	15.6	11.9	13.6	8.8	11.2
P11	0.7	95.9	0.0	109.5	8.0	118.5	0.0	11.2	0.0	10.6	2.2	9.4
P12	0.0	71.6	0.0	88.0	3.0	92.7	0.0	14.8	0.0	13.4	0.0	12.4
P13	4.4	100.9	0.0	112.4	4.3	136.0	4.1	11.6	1.6	11.2	4.4	9.8
P14	0.0	87.3	1.5	106.8	9.1	119.6	0.0	13.0	0.0	12.0	8.8	11.0
P15	4.0	131.1	3.8	136.5	8.0	152.0	2.7	9.2	3.8	9.0	4.5	7.4
P16	0.2	128.8	1.1	142.0	1.5	167.4	0.0	9.0	0.0	8.2	0.7	7.0
Avg.	**5.8**	**90.0**	**4.3**	**92.5**	**10.7**	**104.8**	**4.8**	**21.7**	**5.4**	**17.1**	**8.0**	**15.9**

environment commonly used in continuous integration services [2, 6], which is where we expect most developers to run their tests.

5 Evaluation

5.1 RQ1: Reduction in Testing Time

Table 2 shows the results of applying Random Generation, BOCA, and PROBO on our dataset of 16 projects under two experimental settings: a fixed budget of 20 configurations and a fixed time budget of one hour.

20 Configurations. When each approach evaluates exactly 20 configurations, Random Generation achieves an average testing time reduction of 5.8% from default across all projects, successfully generating configurations that significantly reduce testing time for 12 out of 16 projects. This result is consistent with prior work [13] in that randomly sampling flags can still yield testing time improvements. BOCA achieves an average reduction of 4.3%, performing worse than Random Generation on average, which suggests that directly modeling the relationship from flags to testing time struggles to capture the complex interactions between JVM flags. However, BOCA does outperform Random Generation on 6 projects, most notably on P7 where BOCA achieves 13.2% reduction while Random Generation finds no improvement. PROBO achieves the highest average reduction of 10.7%, which is 1.85× higher than Random Generation and 2.49× higher than BOCA. Notably, PROBO successfully generates configurations that significantly reduce testing time for *all* 16 projects, with reductions ranging from 1.5% (P16) to 27.4% (P7). All statistically significant reductions have large A_{12} effect sizes, averaging 0.986 across the 61 of 80 project-run pairs in which PROBO finds a significant reduction. The "Time" column shows variation across projects because some generated configurations fail for project-specific reasons (e.g., exceeding memory limits). Our

techniques discard such configurations and evaluate the next best one (Section 3.3), until finally successfully evaluating 20 configurations.

1-Hour Time Budget. Under a one-hour time budget, the number of configurations each approach can evaluate varies based on test suite duration and approach overhead. Random Generation evaluates the most configurations (average of 21.7) due to its lack of profiling or model training overhead, yet achieves only 4.8% average reduction. BOCA evaluates fewer configurations (17.1 on average) due to model training overhead, achieving 5.4% average reduction. PROBO evaluates a similar number of configurations as BOCA (15.9 on average) due to the overhead of metrics collection and model training, yet achieves the highest average reduction of 8.0%, outperforming Random Generation on 12 projects and BOCA on 11 projects.

Comparing the two experimental settings, PROBO maintains its advantage in testing time reduction under both, despite its additional overhead from profiling and model training. For projects with shorter-running test suites (e.g., P1, P2), the one-hour budget allows evaluating enough configurations to match or exceed the 20-configuration results. For projects with longer-running test suites (e.g., P16, P15), the 20-configuration setting yields better results because it guarantees sufficient exploration regardless of test duration. For the four projects with the shortest test suites (P1–P4), Random Generation evaluates more than twice as many configurations under the one-hour budget, yet its average reduction for these projects is 1.0 percentage points lower than under the 20-configuration budget, indicating that evaluating more configurations does not by itself yield larger reductions.

Across all experiment runs in both settings, PROBO discards 293 configurations that produce different test outcomes than the default configuration. For 276 of these configurations, the difference appears on the first of the 5 test suite runs. The remaining 17 produce the same outcomes as the default on the first run but differ on a later run, meaning these configurations do not produce consistent outcomes across the 5 test suite runs. As our subjects contain no tests that are flaky under the default configuration (Section 4.2), we attribute this non-determinism to the configuration. Overall, the number of configurations discarded is relatively small compared to the total number of configurations run during the search across all projects. Furthermore, these configurations with different outcomes are heavily concentrated in a single project, P6, which accounts for 217 of the 293, suggesting that some projects and their tests are more susceptible to configuration-induced outcome changes than others. PROBO discards such configurations during search (Section 3.3), so our reported times only consider configurations that preserve the same original testing outcomes, which still provide reductions in testing time.

> ***RQ1:*** *PROBO consistently outperforms both baselines across experimental settings. With 20 configurations, PROBO achieves 10.7% average reduction (1.85× over Random Generation, 2.49× over BOCA), successfully reducing testing time for all 16 projects. Under a one-hour time budget, PROBO maintains its advantage with 8.0% average reduction despite evaluating fewer configurations than Random Generation (15.9 vs. 21.7), demonstrating that PROBO's metrics-guided search enables more effective optimization.*

5.2 RQ2: Runtime Metrics Importance

Table 3 shows the top metrics ranked by how consistently they appear as important across all 16 projects. The "Frequency" column indicates the number of projects for which each metric appears consistently (in the top-5 for at least 3 out of 5 runs). The "Avg. Importance" column shows the mean importance score across all projects where the metric appears consistently; these scores are

Table 3. Most important JVM metrics for predicting testing time.

ID	Metric	Frequency	Avg. Importance	Category
M1	Total Compilation Rate (events/s)	9/16	0.151	Compilation
M2	C1 Compilation Rate (events/s)	7/16	0.154	Compilation
M3	Hot Method Recompilation Rate	6/16	0.204	Compilation
M4	Compilation Duration (ms)	6/16	0.172	Compilation
M5	Mean System CPU %	5/16	0.151	CPU
M6	Mean User CPU %	4/16	0.308	CPU
M7	Compilation Burst Intensity	4/16	0.246	Compilation
M8	Allocation Rate (KB/s)	2/16	0.261	GC
M9	File Read Amount (KB)	2/16	0.236	I/O
M10	Methods compiled	2/16	0.211	Compilation
M11	C2 Compilation Rate (events/s)	2/16	0.183	Compilation
M12	Compilation Success Rate	2/16	0.155	Compilation
M13	Classloading Duration (ms)	1/16	0.304	Compilation
M14	GC Pause Time (ms)	1/16	0.151	GC
M15	Mean Active Threads	1/16	0.141	Threading
M16	File Write Amount (KB)	1/16	0.138	I/O
M17	Safepoint Frequency (events/s)	1/16	0.126	Other
M18	Mean Daemon Threads	1/16	0.117	Threading
M19	Thread Contention Rate (events/s)	1/16	0.109	Threading

normalized by the RF model to sum to 1.0 across all metrics, with higher values indicating stronger predictive power for testing time. The "Category" column shows the category we assigned to each metric (Section 3.1).

In total, we identify 19 metrics that appear consistently across at least one project. The most consistently important metric is Total Compilation Rate, appearing in 9 out of 16 projects with an average importance score of 0.151. This metric measures how frequently the JVM optimizes and compiles methods as part of JIT compilation during test execution, and its dominance confirms that compilation activity is the primary driver of performance variation across configurations. C1 Compilation Rate follows closely, appearing consistently in 7 projects with an average importance of 0.154, indicating that the C1 compiler's activity is particularly impactful on testing time.

Hot Method Recompilation Rate (6 projects, average importance 0.204) and Compilation Duration (6 projects, average importance 0.172) appear with similar frequency, showing that both the JVM's adaptive optimization behavior and the time spent compiling consistently matter across projects. The presence of both CPU metrics, Mean System CPU % (5 projects) and Mean User CPU % (4 projects), reinforces that CPU utilization is consistently predictive.

The importance scores across all 19 metrics range from 0.109 (Thread Contention Rate) to 0.304 (Classloading Duration), indicating substantial variation in how strongly different metrics influence testing time when they consistently appear in the top-5.

Figure 4 shows per-project metric importance patterns. The heatmap reveals substantial heterogeneity in which metrics are important for different projects. For instance, M6 (Mean User CPU %) is critically important for P10 (importance score 0.50) but absent for 12 other projects. Similarly, M8 (Allocation Rate (KB/s)) is important for P8 (importance score 0.22) and P11 (importance score 0.30) but absent for 14 other projects. Even for more universally appearing metrics like M1 (Total

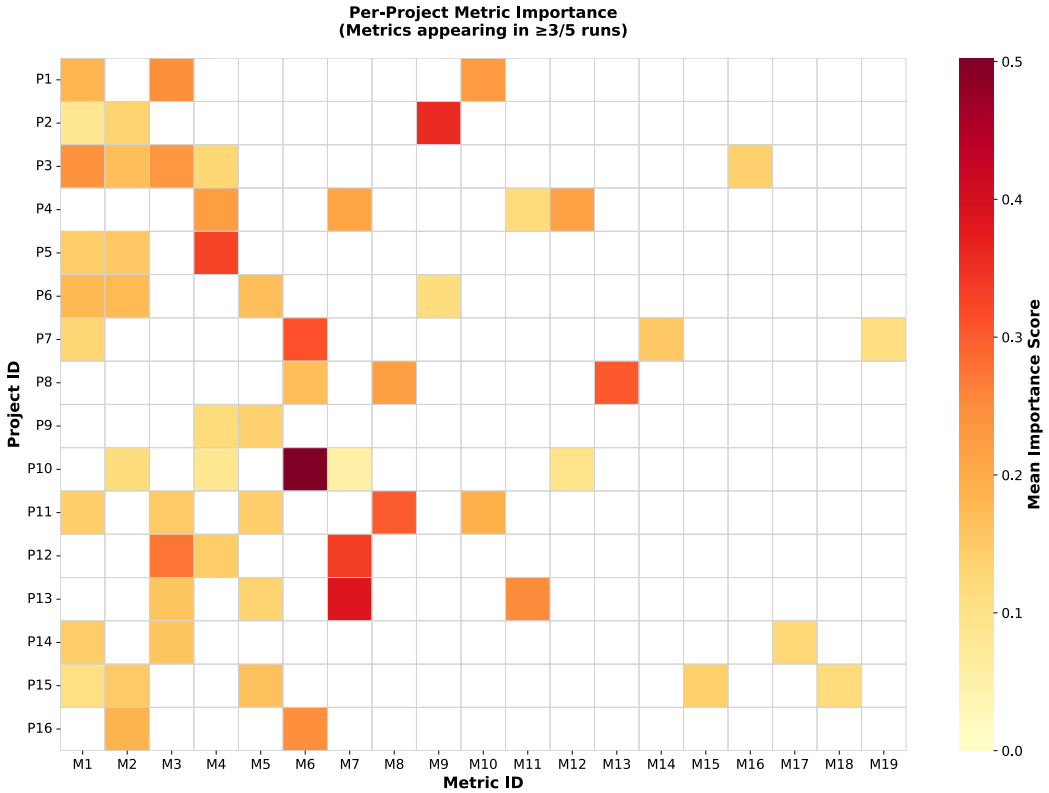


Fig. 4. Per-project metric importance heatmap. Color intensity indicates mean importance score across 5 runs; white cells indicate metrics not consistently appearing in top-5 (≥3/5 runs).

Table 4. Distribution of metric categories among top-5 important metrics across all models.

Category	Count	Percentage (%)
Compilation	39	67.2
CPU	9	15.5
GC	3	5.2
I/O	3	5.2
Threading	3	5.2
Other	1	1.7

Compilation Rate (events/s)), importance scores vary considerably across projects, ranging from 0.08 to 0.24.

Table 4 shows the distribution of metric categories among consistently important metrics across all projects. Compilation metrics dominate with 39 appearances (67.2%), confirming that JIT compilation behavior, including compilation rates, duration, and burst patterns, is consistently the strongest predictor of testing time across different projects. CPU metrics appear 9 times (15.5%), making them the second most important category. This finding is expected, as CPU utilization

Table 5. Configuration stability across future commits. For each project, we apply the best configuration from RQ1 to subsequent commits and measure testing time reduction.

ID	# Commits	Avg. LOC	Avg. Red. (%)	Std (%)
P1	100	155.8	11.1	4.1
P2	100	2.1	16.0	2.7
P3	100	66.6	12.3	4.6
P4	100	65.7	13.3	1.8
P5	98	71.4	10.9	3.8
P6	100	173.3	5.7	2.6
P7	99	274.4	25.2	3.6
P8	100	870.2	1.5	1.4
P9	98	209.5	9.2	5.3
P10	100	7.9	8.6	1.8
P11	100	49.5	3.1	2.4
P13	5	198.8	4.4	3.4
P14	24	27.5	6.3	4.8
P15	100	24.9	4.0	2.3
P16	100	12.6	0.1	0.7
Avg.	**88**	**147.3**	**8.8**	**3.0**

directly reflects the computational intensity of test execution. Together, Compilation and CPU metrics account for over 80% of all consistent metric appearances, suggesting that testing time is primarily driven by how efficiently the JVM compiles and executes code.

GC, I/O, and Threading metrics each appear three times, suggesting they are important for certain projects but not universally predictive. Other metrics appear only one time (1.7%), showing minimal consistent impact across projects.

We observe that no Memory metric is consistently important for any project, so Memory does not appear in Table 4. Moreover, 12 of the 44 metrics do not appear in the top-5 for any project, such as minimum and maximum thread counts and committed heap sizes.

RQ2: *Compilation metrics are the strongest predictors of testing time, accounting for 67.2% of consistently important metrics across all projects. Total Compilation Rate and C1 Compilation Rate are the most consistently important metrics, appearing in 9 and 7 out of 16 projects, respectively. CPU metrics are the second most important category (15.5%), while GC, I/O, and Threading metrics are less frequently predictive. These findings reveal that compilation and CPU utilization are the dominant factors influencing testing time across projects.*

5.3 RQ3: Configuration Stability Across Software Evolution

Table 5 shows the stability of PROBO-generated configurations across future commits of each project. The "# Commits" column shows the number of future commits evaluated, "Avg. LOC" shows the average lines of code changed per commit, "Avg. Red. (%)" shows the average testing time reduction across future commits, and "Std (%)" shows the standard deviation of those reductions.

Across 15 projects, PROBO-generated configurations maintain an average reduction of 8.8% (std 3.0%) over an average of 88 future commits per project. Reductions vary by project: P7 (commons-jexl) maintains the highest at 25.2% across 99 commits, while P16 (commons-net) shows minimal reduction (0.1%), consistent with its low initial reduction in RQ1. Notably, configuration benefits persist despite code changes averaging 147.3 lines per commit, suggesting that configurations often remain useful even as developers make code changes.

> ***RQ3:*** *PROBO-generated configurations remain effective across software evolution, maintaining an average testing time reduction of 8.8% over an average of 88 future commits per project (std 3.0%), suggesting that configurations remain useful even as developers make code changes.*

6 Threats to Validity

Our implementation of PROBO may contain bugs that affect the accuracy of our results. We have carefully reviewed the code to ensure correct implementation of the metrics model and performance model, flag importance calculation, and candidate generation strategies. We also validate each generated configuration against the flag space's dependencies and conflicts to ensure only valid configurations are evaluated.

The testing time reductions we observe may not generalize to all Java projects. We selected 16 diverse projects from prior work on test execution [13], spanning different domains, sizes, and testing characteristics.

Runtime measurements are subject to variance from system load, background processes, and other environmental factors. To mitigate this, we run each configuration 5 times and report the average testing time. We also run each approach five times per project and report the average reduction over those runs. We run all experiments in controlled Docker containers (2 CPUs, 8GB RAM) with fixed resource allocations to ensure consistency across runs. Running in a container adds overhead relative to bare metal [47], and the cost of testing within continuous integration varies across project domains [9], so the absolute testing times we report are specific to our environment. We run experiments with both PROBO and the baselines in identical containers and environments, so our comparisons are not biased one way or the other. We also note that the average testing time reductions we report for Random Generation and BOCA baselines are different than those reported in prior work [13], due to differences in experimental setup and the specific projects evaluated. For Random Generation, if we consider the *best* average testing time reduction from the five runs per project, we find the overall average reduction to be comparable to the prior work.

Our approach relies on 44 JVM runtime metrics collected via JFR as intermediate predictors of testing time. While JFR provides comprehensive runtime observability with minimal overhead, these metrics may not fully capture all factors influencing testing time. Some performance characteristics may be difficult to observe through profiling, and the relationship between metrics and testing time may vary across different projects. Our correlation-based feature selection (Kendall-Tau ≥ 0.3) helps retain only informative metrics, but this threshold is a design choice that could impact model quality. The profiling-guided model architecture and the specific hyperparameters we use are also design choices that may affect the generalizability of our results.

Our evaluation focuses on a specific subset of 72 JVM flags from Java 11, chosen based on their documented impact on runtime behavior and prior work [13]. The flag space may not include all potentially beneficial flags, and newer JVM versions may offer additional optimization opportunities. We compare PROBO against Random Generation and BOCA, which represent reasonable baselines from prior work, but other optimization techniques (e.g., genetic algorithms, other machine learning

approaches) could potentially achieve different results. Our goal is to demonstrate the value of incorporating runtime metrics as intermediate predictors rather than claim superiority over all possible optimization methods.

7 Related Work

Regression Testing. The regression testing literature has established techniques to manage test execution costs, including regression test selection [26, 29, 34, 44] and test-suite reduction [27, 45, 48]. While these approaches reduce testing time by selecting or removing tests, they all carry some risk of reduced fault detection due to not running key tests [49, 50, 60]. Alternatively, test-case prioritization [46, 57] still runs all tests but reorders them as to put the ones most likely to fail and detect faults to run earlier, targeting faster fault detection rather than a reduction in the overall testing time. PROBO aims to reduce testing time by optimizing JVM configurations while executing the complete test suite, preserving full test coverage and fault-detection capability.

Speeding Up Testing. Other research directions have investigated test acceleration techniques that maintain complete test suite execution. Stratis and Rajan introduced a cache-aware test ordering technique that reduces test runtime by minimizing cache misses across test executions [54]. We previously proposed a test prioritization approach to reduce testing time by searching for optimal test orderings based on relative positioning of test classes [14]. Li et al. developed a test class transformation technique that reduces test runtime by minimizing the number of fixture setup and teardown invocations [35]. PROBO builds most directly on our prior work, which investigated JVM configuration tuning for test acceleration, demonstrating that alternative JVM flag settings can substantially reduce test runtime while running the whole suite [13]. PROBO extends this line of work by applying profiling-guided Bayesian optimization to JVM configuration tuning, using runtime metrics to more effectively navigate the configuration space compared to prior random or direct modeling approaches.

Configuration Tuning. Compiler autotuning research focuses on identifying optimal compiler optimization flag configurations to maximize the performance of executing the generated and optimized binaries [7, 10, 11, 20, 21]. These machine learning-based search techniques navigate the compiler flag configuration space to discover flag combinations that yield high-performance binaries across diverse inputs. OpenTuner [8] provides a general-purpose framework for autotuning arbitrary configuration spaces through user-defined objectives. Canales et al. tune the JVM rather than the compiler, applying feature modeling to represent JVM flag configuration spaces and employing genetic algorithms to search for configurations that improve program execution time [19]. Unlike prior autotuning approaches that directly model flags to performance, PROBO decomposes prediction through observable runtime behaviors, using JVM runtime metrics as intermediate predictors to enable more interpretable and effective optimization. Awad et al. proposed DEHB, which combines differential evolution with Hyperband to evaluate many configurations under a small budget and promote only the promising ones to larger budgets [12]. Such multi-fidelity scheduling determines how much budget each configuration receives rather than which configurations to evaluate, so it is orthogonal to PROBO and could be combined with it.

8 Conclusions and Future Work

We present PROBO, a profiling-guided Bayesian optimization approach that leverages JVM runtime metrics to guide the search for JVM configurations that reduce Java testing time. Unlike prior approaches that learn an opaque mapping from JVM configuration flags to testing time, PROBO decomposes the prediction problem through observable runtime behaviors: a metrics model predicts how flag configurations affect runtime metrics, and a performance model predicts how those metrics

affect testing time. PROBO propagates feature importance scores through both models to identify which flags most strongly influence the metrics that are themselves most predictive of testing time. Using these importance scores, PROBO generates candidate configurations through three complementary strategies and selects the most promising candidates based on the expected improvement acquisition function. Through iterative evaluation and model refinement via Bayesian optimization, PROBO efficiently navigates the configuration space to find high-quality configurations.

We evaluated PROBO on 16 open-source Java projects, comparing its performance against Random Generation that randomly combines flags to generate configurations and BOCA, a Bayesian optimization baseline that directly predicts testing time from flag settings. PROBO achieves an average testing time reduction of 10.7%, representing a 1.85× improvement over Random Generation and a 2.49× improvement over BOCA. PROBO successfully generates configurations that reduce testing time for all 16 projects, with reductions ranging up to 27.4%. Even under a constrained one-hour time budget, PROBO maintains strong performance with an average reduction of 8.0%, demonstrating practical applicability in real development workflows. Moreover, PROBO-generated configurations remain effective across software evolution, maintaining an average reduction of 8.8% over an average of 88 future commits per project. Our analysis reveals that compilation metrics, particularly Total Compilation Rate and C1 Compilation Rate, are the strongest predictors of testing time, accounting for 67.2% of consistently important metrics across projects.

These results demonstrate that incorporating JVM runtime metrics as intermediate features enables more effective configuration optimization, allowing PROBO to distinguish between truly important flags and those that correlate spuriously with performance. Future work could extend PROBO's approach in several directions. First, while our evaluation focuses on single-module Maven projects, investigating PROBO's effectiveness on multi-module projects would broaden its applicability, with challenges in how to generate a configuration that is effective across multiple modules as they each ones' tests are run in separate JVM processes. Second, investigating whether PROBO's profiling-guided modeling approach can be adapted to other configuration-tuning domains beyond JVM flags, such as compiler flags or database configurations, could demonstrate the broader value of incorporating intermediate system metrics in Bayesian optimization. Finally, developing techniques to automatically recommend configurations based on static analysis of test suite characteristics could further reduce the time required to find effective configurations.

Data Availability

We share our scripts for running experiments, the code for PROBO, and the experiment results [5].

Acknowledgments

We thank Tami Takada for her help in developing the initial code and scripts for profiling using JFR. We thank the anonymous reviewers for their comments and feedback. This work is partially supported by the US National Science Foundation grant nos CCF-2145774, CCF-2217696 and CCF-2338287, as well as the Jarmon Innovation Fund.

References

[1] 2018. JEP 328: Flight Recorder. https://openjdk.org/jeps/328.
[2] 2026. GitHub Actions. https://github.com/features/actions.
[3] 2026. Java 11 Documentation. https://docs.oracle.com/en/java/javase/11/tools/java.html.
[4] 2026. JDK Flight Recorder. https://docs.redhat.com/en/documentation/red_hat_build_of_openjdk/11/html/using_jdk_flight_recorder_with_red_hat_build_of_openjdk/openjdk-flight-recorded-overview.
[5] 2026. Profiling-Guided Bayesian Optimization of JVM Configurations. https://sites.google.com/view/probo-jvm-configs.
[6] 2026. Travis-CI. https://travis-ci.org.

[7] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. 2006. Using Machine Learning to Focus Iterative Optimization. In *International Symposium on Code Generation and Optimization*. 295–305. doi:10.1109/CGO.2006.37

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. 303–316. doi:10.1145/2628071.2628092

[9] Robert Arntzenius, Xutong Liu, and Andy Zaidman. 2026. On the Energy Consumption of Continuous Integration in Open-Source Java Projects. In *International Workshop on Green Software Evolution*. 181–188. doi:10.1109/SANER-C67878.2026.00030

[10] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2019. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys (CSUR)* 51, 5 (2019), 1–42. doi:10.1145/3197978

[11] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. 2014. A Bayesian Network Approach for Compiler Auto-tuning for Embedded Processors. In *Symposium on Embedded Systems for Real-time Multimedia*. 90–97. doi:10.1109/ESTIMEDIA.2014.6962349

[12] Noor Awad, Neeratyoy Mallik, and Frank Hutter. 2021. DEHB: Evolutionary Hyperband for Scalable, Robust and Efficient Hyperparameter Optimization. In *International Joint Conference on Artificial Intelligence*. 2147–2153. doi:10.24963/ijcai.2021/296

[13] Abdelrahman Baz, Milos Gligoric, and August Shi. 2024. Impact of JVM Configurations on Test Runtime. In *International Conference on Software Maintenance and Evolution*. 249–261. doi:10.1109/ICSME58944.2024.00032

[14] Abdelrahman Baz, Minchao Huang, and August Shi. 2024. Prioritizing Tests for Improved Runtime. In *International Conference on Automated Software Engineering (New Ideas and Emerging Results Track)*. 2273–2278. doi:10.1145/3691620.3695298

[15] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*. 433–444. doi:10.1145/3180155.3180164

[16] Romain Benassi, Julien Bect, and Emmanuel Vazquez. 2011. Robust Gaussian Process-Based Global Optimization using a Fully Bayesian Expected Improvement Criterion. In *International Conference on Learning and Intelligent Optimization*. 176–190. doi:10.1007/978-3-642-25566-3_13

[17] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning*. 115–123.

[18] Leo Breiman. 2001. Random Forests. *Machine Learning* 45 (2001), 5–32. doi:10.1023/A:1010933404324

[19] Felipe Canales, Geoffrey Hecht, and Alexandre Bergel. 2021. Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm. In *International Conference on Performance Engineering*. 183–186. doi:10.1145/3447545.3451177

[20] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization*. 185–197. doi:10.1109/CGO.2007.32

[21] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *International Conference on Software Engineering*. 1198–1209. doi:10.1109/ICSE43902.2021.00110

[22] T. Y. Chen and M. F. Lau. 1998. A New Heuristic for Test Suite Reduction. *Journal of Information and Software Technology* 40, 5-6 (1998), 347–354. doi:10.1016/S0950-5849(98)00050-0

[23] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Lawrence Erlbaum Associates.

[24] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245. doi:10.1145/2635868.2635910

[25] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *International Symposium on Empirical Software Engineering and Measurement*. 22–31. doi:10.1145/1414004.1414011

[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222. doi:10.1145/2771783.2771784

[27] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-Demand Test Suite Reduction. In *International Conference on Software Engineering*. 738–748. doi:10.1109/ICSE.2012.6227144

[28] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 312–326. doi:10.1145/504282.504305

[29] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Transactions on Software Engineering* 27, 3 (2001), 248–263. doi:10.1109/32.

910860
[30] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less Without Sacrificing Quality. In *International Conference on Software Engineering*. 483–493. doi:10.1109/ICSE.2015.66
[31] Dennis Jeffrey and Neelam Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *IEEE Transactions on Software Engineering* 33, 2 (2007), 108–123. doi:10.1109/TSE.2007.18
[32] Maurice G. Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 30, 1-2 (1938), 81–93. doi:10.1093/biomet/30.1-2.81
[33] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A Large-Scale Longitudinal Study of Flaky Tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 202:1–202:29. doi:10.1145/3428270
[34] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594. doi:10.1145/2950290.2950361
[35] Chengpeng Li, Abdelrahman Baz, and August Shi. 2024. Reducing Test Runtime by Transforming Test Fixtures. In *International Conference on Automated Software Engineering*. 1757–1769. doi:10.1145/3691620.3695541
[36] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653. doi:10.1145/2635868.2635920
[37] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. 2005. Test-Suite Reduction using Genetic Algorithm. In *International Conference on Advanced Parallel Processing Technologies*. 253–262. doi:10.1007/11573937_28
[38] Thomas W MacFarland and Jan M Yates. 2016. Mann–Whitney U Test. *Introduction to Nonparametric Statistics for the Biological Sciences Using R* (2016), 103–132. doi:10.1007/978-3-319-30634-6_4
[39] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. 91–100. doi:10.1109/ICSE-SEIP.2019.00018
[40] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. 233–242. doi:10.1109/ICSE-SEIP.2017.16
[41] Jonas Mockus. 1989. *Bayesian Approach to Global Optimization: Theory and Applications*. Vol. 37. Springer Netherlands. doi:10.1007/978-94-009-0909-0
[42] Carl Edward Rasmussen and Christopher K.I. Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press. doi:10.7551/mitpress/3206.001.0001
[43] Gregg Rothermel and Mary Jean Harrold. 1994. A Framework for Evaluating Regression Test Selection Techniques. In *International Conference on Software Engineering*. 201–210. doi:10.1109/ICSE.1994.296779
[44] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology* 6, 2 (1997), 173–210. doi:10.1145/248233.248262
[45] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical Studies of Test-Suite Reduction. *Journal of Software Testing, Verification and Reliability* 12, 4 (2002), 219–249. doi:10.1002/stvr.256
[46] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 1999. Test Case Prioritization: An Empirical Study. In *International Conference on Software Maintenance*. 179–188. doi:10.1109/ICSM.1999.792604
[47] Eddie Antonio Santos, Carson McLean, Christopher Solinas, and Abram Hindle. 2018. How does Docker Affect Energy Consumption? Evaluating Workloads in and out of Docker Containers. *Journal of Systems and Software* 146 (2018), 14–25. doi:10.1016/j.jss.2018.07.077
[48] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-Offs in Test-Suite Reduction. In *International Symposium on Foundations of Software Engineering*. 246–256. doi:10.1145/2635868.2635921
[49] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *International Symposium on Software Testing and Analysis*. 84–94. doi:10.1145/3213846.3213875
[50] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 187:1–187:29. doi:10.1145/3360613
[51] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjørner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *International Conference on Software Engineering*. 689–699. doi:10.1109/ICSE.2017.69
[52] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-Suite Reduction and Regression Test Selection. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 237–247. doi:10.1145/2786805.2786878
[53] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *International Symposium on Software Reliability Engineering*. 228–238. doi:10.1109/ISSRE.

2019.00031

[54] Panagiotis Stratis and Ajitha Rajan. 2018. Speeding up Test Execution with Increased Cache Locality. *Journal of Software Testing, Verification and Reliability* 28, 5 (2018), 1–17. doi:10.1002/stvr.1671

[55] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning through Large-Scale Machine Learning. In *International Conference on Management of Data*. 1009–1024. doi:10.1145/3035918.3064029

[56] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. doi:10.3102/10769986025002101

[57] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. doi:10.1002/stvr.430

[58] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *International Conference on Software Engineering*. 199–209. doi:10.1145/3180155.3180198

[59] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *International Symposium on Software Reliability Engineering*. 170–179. doi:10.1109/ISSRE.2011.26

[60] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *International Conference on Software Engineering*. 430–441. doi:10.1109/ICSE.2019.00056

Received 2026-01-30; accepted 2026-04-16