Unveiling Method-Level Performance Trends

An Empirical Analysis of Java Code Performance Evolution in Open-Source Projects

Kaveh Shahedi, Heng Li
Polytechnique Montréal
Summer/Fall 2024

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

What is “Performance Evolution”?

```python
# Version 1 (Commit 1)
def calculate_total_price(items: list[Item]) -> float:
    total : float = 0
    for item in items:
        total += get_price_from_database(item)
    return total
```

```python
# Version 2 (Commit 2)
def calculate_total_price(items: list[Item]) -> float:
    price_cache : dict[Item, float] = {}
    total : float = 0
    for item in items:
        if item not in price_cache:
            price_cache[item] = get_price_from_database(item)
        total += price_cache[item]
    return total
```

And of course, in **Java**

```java
# Version 1 (Commit 1)
public double calculateTotalPrice(List<Item> items) {
   double total = 0;
   for (Item item : items) {
      total += getPriceFromDatabase(item);
   }
   return total;
}
```

```java
# Version 1 (Commit 2)
public double calculateTotalPrice(List<Item> items) {
   Map<String, Double> priceCache = new HashMap<>();
   double total = 0;
   for (Item item : items) {
      total += priceCache.computeIfAbsent(item,
            this::getPriceFromDatabase);
   }
   return total;
}
```

Current Works/Research We’re Doing?

“Insights on Method-Level Performance Changes”

“Performance-Oriented Software Refactoring”

Upcoming

“Automated Generation of Performance Regression Unit Tests Using Adaptive Instrumentation and Code Analysis”

Primary components in this work

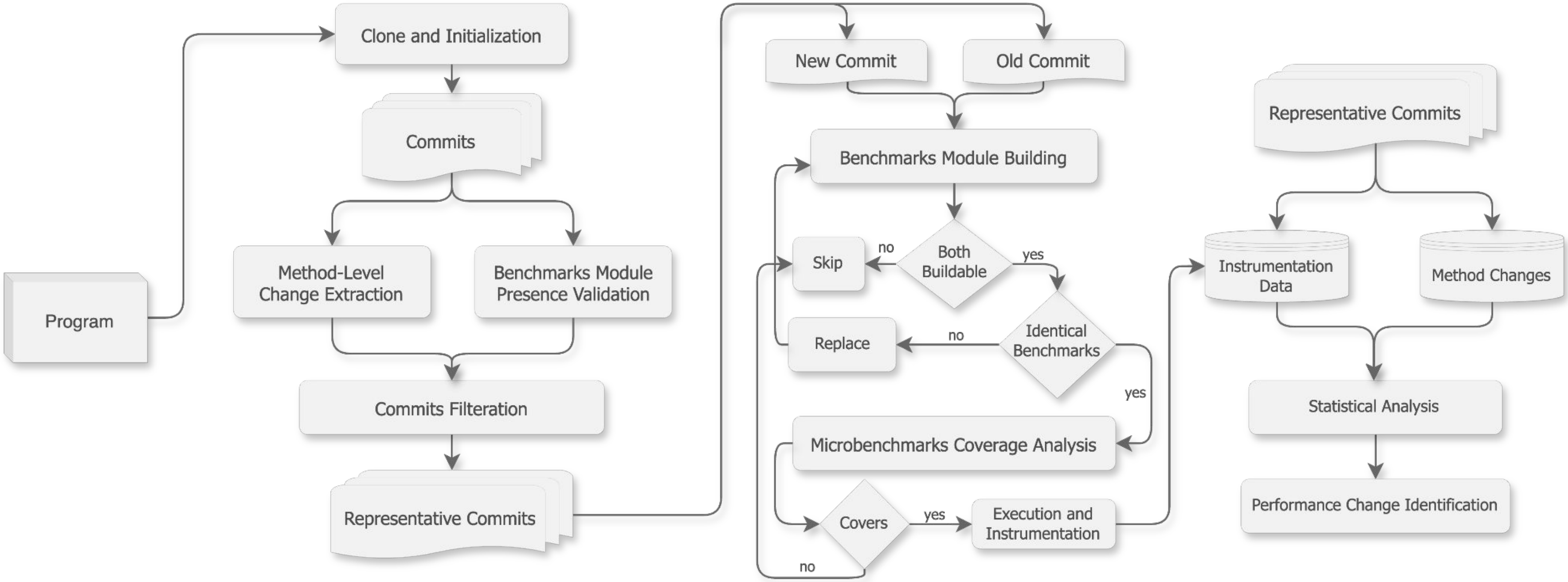
JPerfEvo

Java Performance Evolution Buddy
The primary **pipeline** for mining, analysis, and benchmarking

JIB

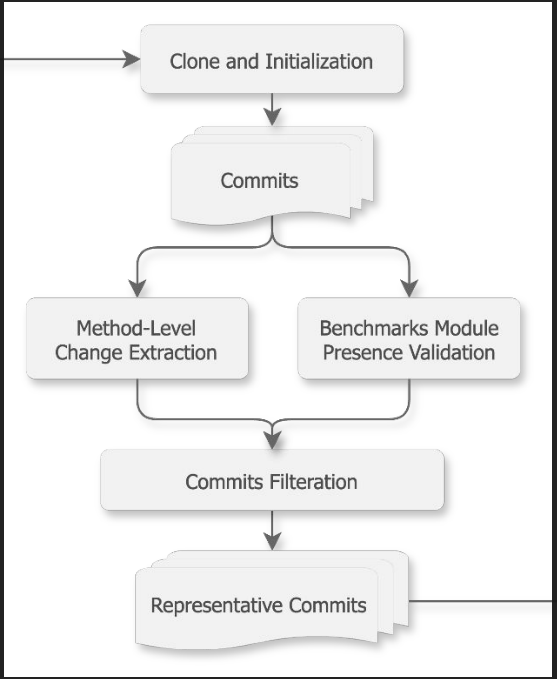
Java Instrumentation Buddy
A lightweight instrumenting agent for Java

Overview



Program
Clone and Initialization
Commits
Method-Level Change Extraction
Benchmarks Module Presence Validation
Commits Filteration
Representative Commits
New Commit
Old Commit
Benchmarks Module Building
Skip
no
Both Buildable
yes
Replace
no
Identical Benchmarks
yes
Microbenchmarks Coverage Analysis
Covers
yes
Execution and Instrumentation
no
Representative Commits
Instrumentation Data
Method Changes
Statistical Analysis
Performance Change Identification

Step 1: Project Initialization and Data Collection



1. **Clone** the project
2. **Iterate** through its **commits**
   i. Should have **at least one valid method-level code change**
   ii. Should have **JMH module**
   iii. Should **not** be a **merge** commit
   iv. Should have **valid pom.xml**
   v. **CI build** should be **successful**
3. **Extract method-level changes** -> for **before**/**after** the commit
4. Save as **Representative Commits**

Which **projects** are we analyzing?

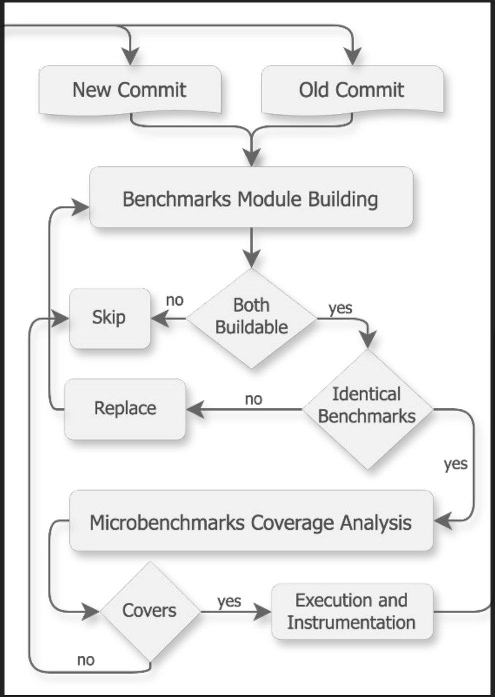
Project	Commits	KLoC	Commits with Method Changes	Commits with Benchmark	Representative Commits	Executed Commits	Collected Changed Methods
jetty.project	30,160	339.06	2,472	12,720	2,470	56	**124**
netty	11,604	216.98	4,241	7,669	4,240	57	**97**
jdbi	5,709	28.49	1,266	1,919	313	90	**136**
fastjson2	4,372	178.5	1,726	3,752	1,726	220	**615**
Chronicle-Core	3,911	13.25	780	3,170	585	2	**3**
SimpleFlatMapper	3,433	51.79	911	1,969	485	45	**68**
apm-agent-java	3,066	80.22	891	2,984	889	86	**176**
zipkin	2,955	23.51	656	2,726	615	46	**93**
feign	2,063	17.42	351	1,384	229	54	**114**
protostuff	1,603	42.29	448	1,354	448	4	**4**
JCTools	1,043	31.48	339	1,042	339	26	**52**
objenesis	1,049	2.69	107	784	72	12	**14**
client_java	866	27.38	155	667	154	9	**11**

I know it sounds weird, but project building procedure is not an easy task folks…

SPECIALLY FOR JAVA

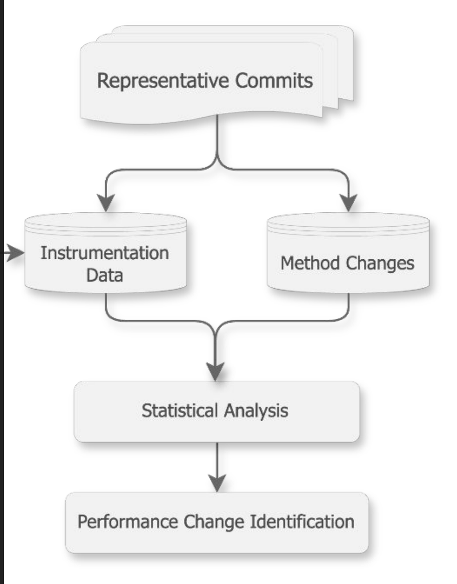
Step 2: Benchmarking and Instrumentation

1. **Build** JMH **benchmarks** (bef/aft)
   -> **if** any **fails**, **skip**!
2. Check for **identical benchmarks**
   -> **replace with old** if not same
   -> **if not compatible**, use **newer**
   -> **if fails** again, **skip**!
3. **Get** microbenchmarks **coverage**
   -> **if no coverage**, **skip**!
4. **Execute** and **instrument** microbenchmarks



Step 3: Performance Change Analysis

1. With before/after **trace data**
   i. **Mann–Whitney U Test**
      to check **significance**
   ii. **Cliff's Delta Effect Size**
      to get **significance size**
2. **Indicate** performance **change**
   -> **improvement**? **regression**? **neutral**?
3. (Exclusive) **Label** code change **type**
   -> **algorithmic**? **data structure**? other?
4. (Exclusive) **Analyze** the performance **trend**



This pipeline (JPerfEvo) is submitted to **"International Conference on Mining Software Repositories (MSR) 2025 - Data and Tool Showcase Track"**

RQ1

What Are the Patterns of Performance Changes in Java Projects Over Time?

The distribution of performance changes (i.e., effect size) over time across all projects

The distribution of performance changes (i.e., effect size) over time across all projects.

Performance regressions show slightly larger effect sizes than improvements, with both demonstrating increased stability over time

The distribution of performance change effect size categories based on the performance change type

The distribution of performance change effect size categories based on the performance change type

Most significant performance changes tend to have a small effect size

Distribution of code change impacts on performance across project lifecycle stages

Project Stage	Change Type		
	Improvement	Regression	Unchanged
Early	**19.20%**	**21.43%**	59.38%
Middle	14.84%	18.13%	67.03%
Late	12.76%	16.87%	**70.37%**

The evolution across life stages shows a trend toward increased stability as projects mature

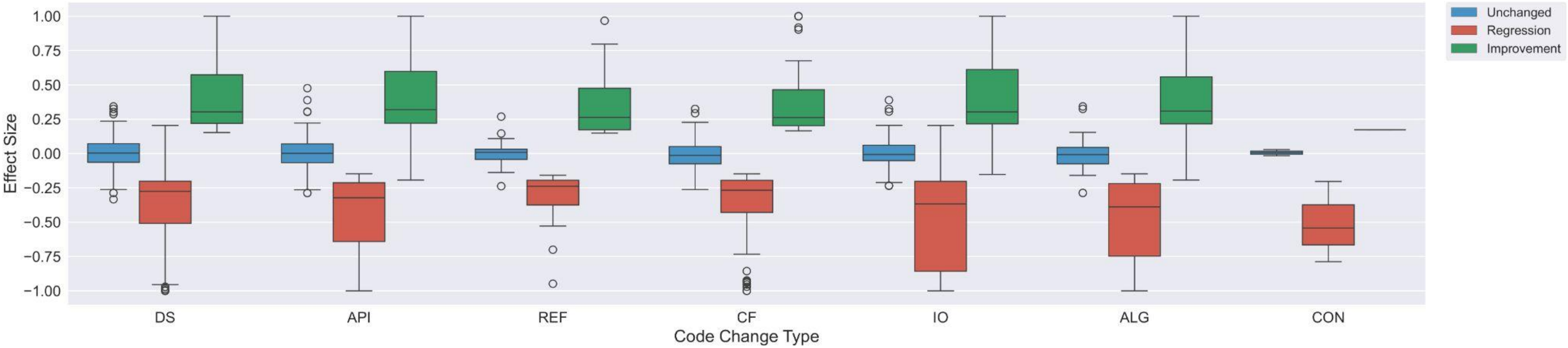
Distribution of code change impacts on performance across project lifecycle stages

Project Stage	Change Type		
	Improvement	Regression	Unchanged
Early	**19.20%**	**21.43%**	59.38%
Middle	14.84%	18.13%	67.03%
Late	12.76%	16.87%	**70.37%**

RQ2

What is the Correlation Between Code Changes and Performance Impacts, and What Defines Commits with Significant Performance Shifts?

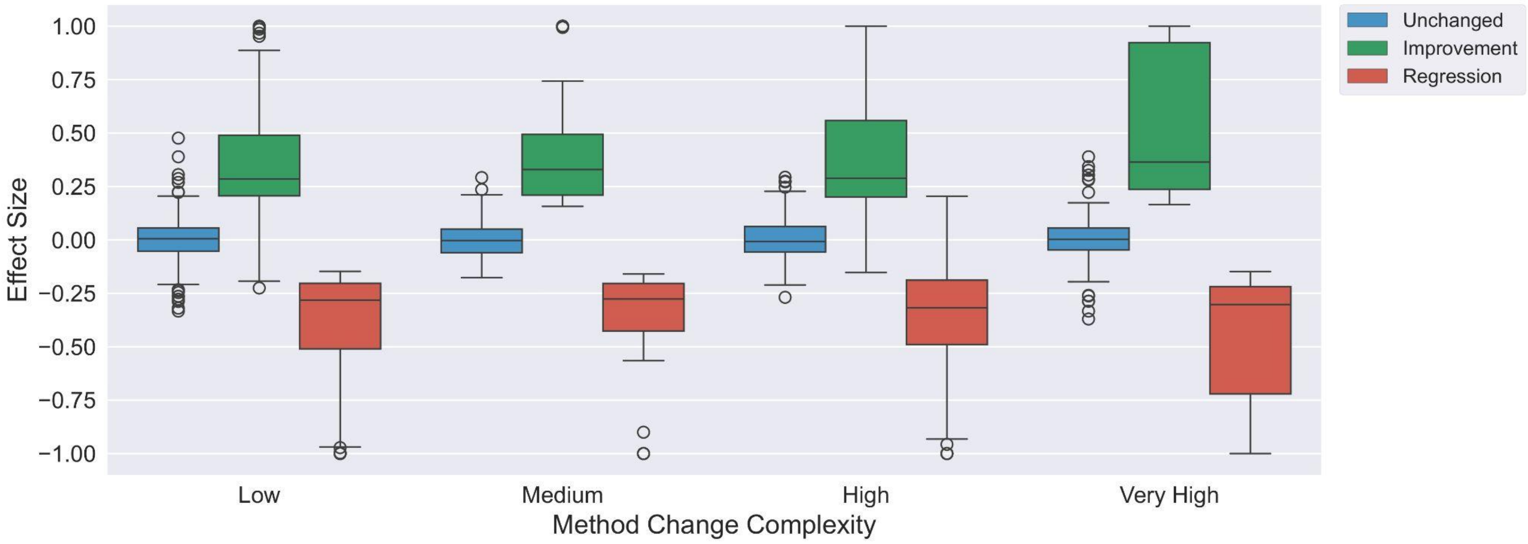
How each change type contributes to performance improvements, regressions, and neutral changes



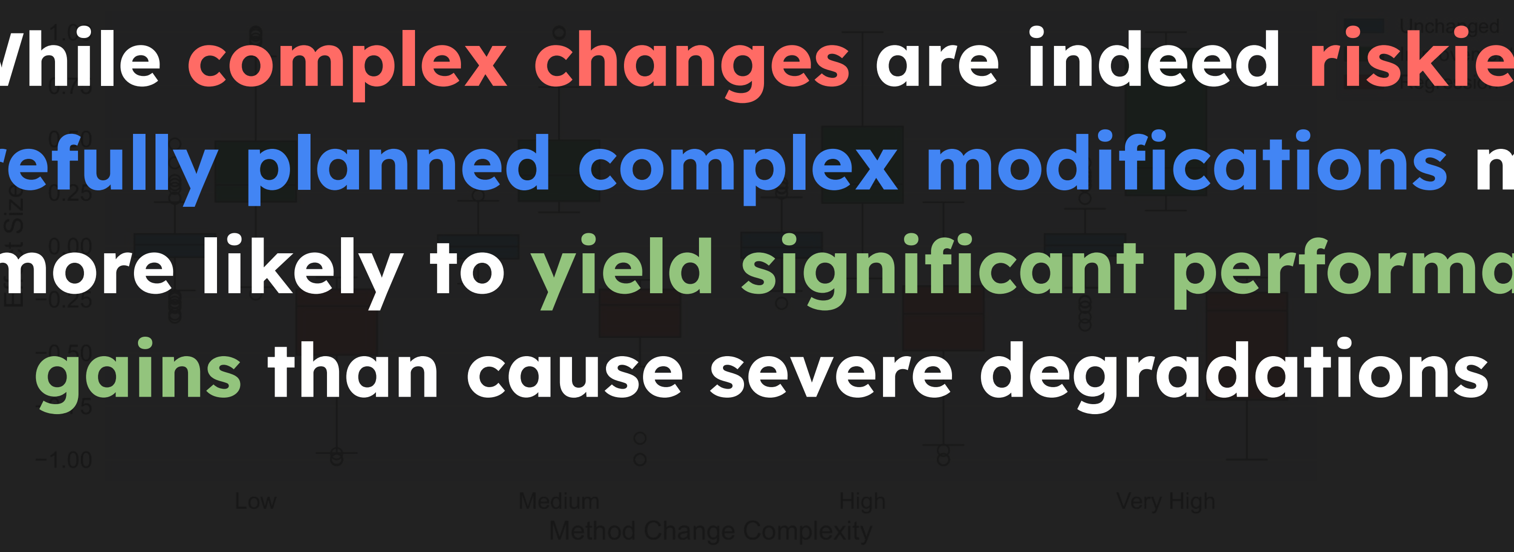
The distribution of performance changes (i.e., effect size) over time across all projects

API/Library Call and Algorithm Change modifications tend to have the greatest positive impact on performance, while Exception and I/O Handling changes contribute the most to performance regressions

Effect size distribution by method change complexity



The distribution of performance changes (i.e., effect size) over time across all projects



While complex changes are indeed riskier, carefully planned complex modifications may be more likely to yield significant performance gains than cause severe degradations

Performance change distribution (in percentage) based on commiter’s experience

Author Experience	Change Type		
	Improvement	Regression	Unchanged
Junior	13.73%	**19.41%**	66.86%
Mid	13.97%	16.44%	**69.59%**
Senior	**17.22%**	17.78%	65.00%

Performance change distribution (in percentage) based on commiter's experience

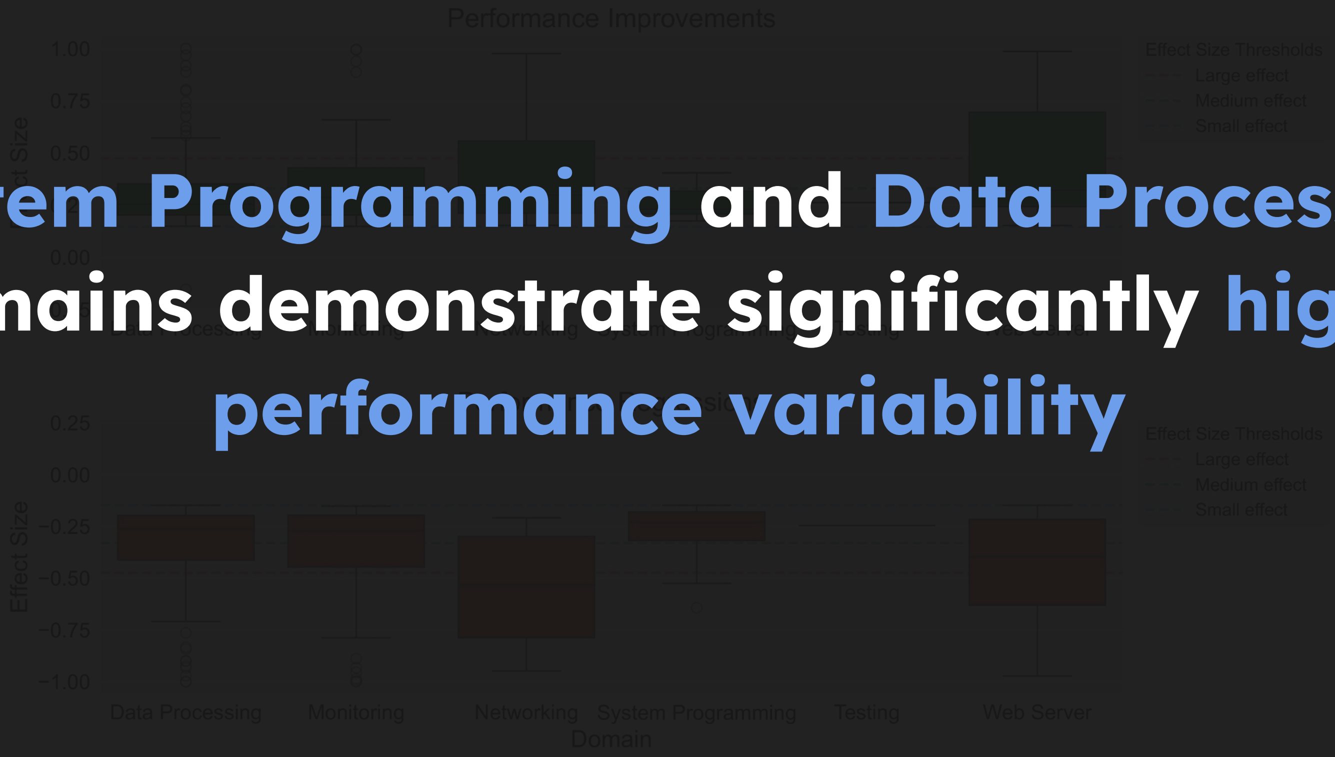
Balanced performance **maintenance** may be better achieved through the **collaborative work of mid-level developers'** careful approach and **seniors' optimization expertise**

RQ3

Are There Significant Differences in Performance Evolution Patterns Across Different Domains or Project Sizes?

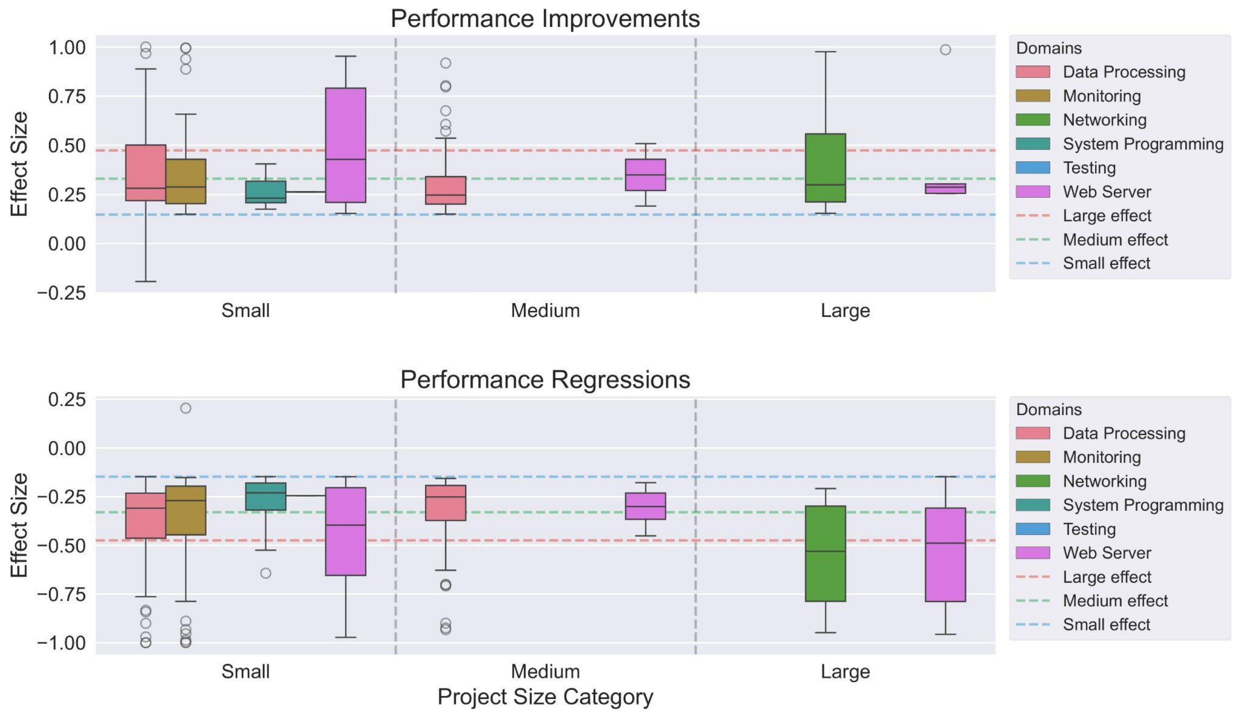
Performance change effect size in each project's domain

Performance change effect size in each project's domain



System Programming and Data Processing domains demonstrate significantly higher performance variability

Comparison of performance change affected by project's size, indicated for each domain



So, small projects need careful handling due to high variability, medium-sized projects benefit from stability, and large projects require proactive measures to prevent regressions.

The complete study is also submitted to
“Can’t say it, it’s double-blind”

Java's all grown up—focus on tiny tweaks and watch out for sneaky regressions!

Big algorithm and I/O changes are like juggling chainsaws—great rewards but great risks!

One size doesn't fit all—tailor your performance strategy to your project's quirks, and remember: small projects can cause big surprises!

Thanks!