

Comparison of Java and .NET Reflection Mechanisms for Dynamic Module Loading: A Performance Benchmark Study

Michał Mazur*, Sebastian Maruszak, Marek Miłośz

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This paper presents an experimental performance analysis of dynamic module loading via reflection on the Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR). Three cold-loading benchmark scenarios are defined and executed with $N=1000$ post-warm-up iterations each: (A) Java URLClassLoader (non-modular JAR), (B) Java JPMS ModuleLayer (modular JAR), and (C) .NET AssemblyLoadContext. Each iteration measures four consecutive phases: module loading, type resolution, instance creation, and method invocation. After JIT warm-up ($W=50$ iterations) and IQR-based outlier removal, .NET AssemblyLoadContext achieves the lowest total mean time of 0.419 ± 0.104 ms, whereas Java JPMS incurs the highest mean of 1.299 ± 0.532 ms driven primarily by ModuleLayer construction (5.2x overhead vs. URLClassLoader). .NET type resolution is 20.9x faster than Java URLClassLoader (0.030 ms vs. 0.626 ms). Non-parametric statistical analysis (Mann–Whitney U, Cohen’s d : 0.66–2.26) confirms that all pairwise differences are highly significant ($p < 0.001$). Cached-reflection baselines show that both platforms reduce to sub-10 us once class/assembly metadata is reused.

Keywords: dynamic module loading; Java; .NET; performance benchmarking; URLClassLoader; JPMS

*Corresponding author

Email address: s97677@pollub.edu.pl (Michał Mazur)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

The ability to load and invoke code at runtime—commonly realised through reflection—is a cornerstone of modern software architecture. Plugin systems, dependency injection (DI) containers, object-relational mappers, scripting engines, and test harnesses all rely on the runtime discovery and invocation of code that was not available at compile time [1–3].

Both the Java platform and Microsoft .NET provide rich reflection APIs [4, 5]. However, the two platforms have evolved substantially different module systems in recent years. Java 9 introduced the Java Platform Module System (JPMS, JSR-376 [6]) alongside the legacy URLClassLoader approach, while .NET Core introduced the collectible AssemblyLoadContext (ALC) as the standard mechanism for plugin isolation [7, 8].

Despite the practical importance of reflection performance, comparative studies that decompose the cost into orthogonal phases—module loading, type resolution, object instantiation, and method invocation—across both platforms are scarce. Prior work either focuses solely on JVM [9, 10] or presents aggregate benchmarks without phase-level decomposition or rigorous statistical testing [11].

The contributions of this paper are:

1. A reproducible, open-source four-phase benchmark spanning three cold-loading scenarios across JVM (Java 21, OpenJDK Temurin) and CLR (.NET 10);
2. Empirical results with IQR-based outlier removal assessed over $N=1000$ post-warm-up iterations per scenario, complemented by cached-reflection and direct-invocation baselines;

3. Non-parametric statistical significance testing (Mann–Whitney U) and Cohen’s d effect-size analysis for all pairwise comparisons.

1.1. Research Hypotheses

Three hypotheses guide the experiment:

H1. The total cost of dynamic module loading and reflective invocation is measurably different between Java URLClassLoader, Java JPMS, and .NET ALC under equivalent workloads.

H2. JPMS (ModuleLayer) incurs a significantly higher module-loading overhead than URLClassLoader, because ModuleLayer resolution involves graph-based dependency analysis absent in the URLClassLoader path.

H3. .NET ALC loads assemblies faster than Java URLClassLoader, whereas .NET reflective instance creation and method invocation are slower than their Java counterparts.

2. Related Work

The performance of Java reflection has been studied in the context of object-oriented frameworks [9]. I. Forman and N. Forman showed that reflective invocation is approximately 10–50x slower than direct method calls in early JVM implementations [9]; subsequent JIT advances have substantially reduced this gap. Paleczny et al. described the HotSpot JIT compilation strategy that turns reflective call-sites into intrinsics after sufficient invocations [12].

Benchmarking methodology is discussed in depth by Georges et al. [13], who stress the need for warm-up phases and statistical reporting (confidence intervals) rather than single-run timings. The Computer Language Benchmarks Game [11] provides aggregate throughput

comparisons between Java and .NET but does not isolate dynamic loading or decompose phases.

Pradel et al. [10] analysed the cost of dynamic dispatch in prototype-based languages; their findings highlight that per-call reflection overhead drops significantly with JIT call-site specialisation. The Java Microbenchmark Harness (JMH) [14] provides infrastructure for eliminating JIT and dead-code artefacts; our custom harness implements equivalent warm-up and isolation strategies without requiring the JMH dependency.

To our knowledge, no prior study applies a controlled, phase-level decomposition to the comparison of modern Java module systems—both URLClassLoader and JPMS ModuleLayer—against .NET AssemblyLoadContext, with rigorous non-parametric statistical testing.

3. Materials and Methods

3.1. Test Environment

All measurements were performed on a single machine to eliminate hardware variability. The configuration is listed in Table 1.

Table 1: Test environment configuration

Component	Value
CPU	Intel Core i7-12700H
RAM	32 GB DDR5-4800
OS	Windows 11 Pro (23H2)
JVM	OpenJDK Temurin 21.0.6
.NET	.NET 10.0 (SDK 10.0.201)
Storage	NVMe SSD (sequential read >3 GB/s)
Python	CPython 3.12.6 (analysis scripts only)

3.2. Benchmark Architecture

The benchmark measures the cost of dynamically loading a pre-compiled module and invoking a single service method through reflection. The module under test (TestService) exposes: (a) a zero-argument constructor, and (b) a compute (int n) method performing an arithmetic sum $\sum_{i=0}^{n-1} i$ for constant $n = 100$.

The computation is intentionally lightweight so that invocation overhead dominates over actual work.

Three cold-loading scenarios are defined:

- Scenario A – Java URLClassLoader: loads a non-modular JAR via URLClassLoader, then resolves the type, creates an instance, and invokes the method using `java.lang.reflect`. A fresh URLClassLoader is created every iteration.
- Scenario B – Java JPMS ModuleLayer: loads a modular JAR by calling `ModuleFinder.of()`, building a Configuration, and invoking `ModuleLayer.defineModulesWithOneLoader()`. A new ModuleLayer is constructed every iteration.
- Scenario C – .NET AssemblyLoadContext: constructs a collectible AssemblyLoadContext, loads the DLL via `LoadFromAssemblyPath()`, resolves the type, and performs reflective instantiation and invocation through .NET `System.Reflection`. A new AssemblyLoadContext is created every iteration.

Two additional baseline scenarios are also measured to contextualise the cold-loading overhead: Cached Reflection (reusing pre-fetched Constructor/ConstructorInfo and Method/MethodInfo references without reloading) and Direct Invocation (calling `TestService.compute()` directly without any reflection mechanism).

3.3. Measurement Phases

Each iteration is divided into four timed phases, measured with `System.nanoTime()` (JVM) and `System.Diagnostics.Stopwatch` (.NET), both providing nanosecond-resolution hardware counters. Results are recorded in milliseconds.

- Phase 1 – Module Load. Construct the loader and load the binary into the runtime (URLClassLoader, ModuleLayer, or AssemblyLoadContext.LoadFromAssemblyPath).
- Phase 2 – Find Type. Resolve the target class or type by name: `ClassLoader.loadClass()` (Java) or `Assembly.GetType()` (.NET).
- Phase 3 – Create Instance. Obtain the constructor reflectively and create an object: `Constructor.newInstance()` (Java) or `Activator.CreateInstance()` (.NET).
- Phase 4 – Invoke Method. Obtain the method reflectively and call it: `Method.invoke()` (Java) or `MethodInfo.Invoke()` (.NET).

Memory consumption (heap delta per iteration) is measured using `MemoryMXBean.getHeapMemoryUsage()` (JVM) and `GC.GetTotalMemory(false)` (.NET).

3.4. Experimental Protocol

Warm-up. Each scenario is preceded by $W=50$ unrecorded warm-up iterations. This allows the JVM HotSpot C2 compiler and .NET tiered JIT to compile the critical path and eliminates first-run OS disk-cache miss costs for JAR/DLL loading.

Sample size. $N = 1000$ iterations are recorded per scenario. The final effective sample sizes after outlier removal range from 839 to 958 across all seven scenarios. Outlier removal. Outliers are removed per scenario using Tukey's IQR fence [15] applied to the total iteration time:

$$[Q_1 - 1.5 \cdot \text{IQR}, Q_3 + 1.5 \cdot \text{IQR}] \quad (1)$$

Across all seven scenarios, on average $\approx 8.8\%$ of measurements are removed.

Source availability. The complete benchmark source code, build scripts, and raw CSV results are publicly available in the project repository.

3.5. Statistical Methods

We report the mean ($\bar{x}$), standard deviation (s), median ($\tilde{x}$), 95th and 99th percentiles (P_{95} , P_{99}), and the half-width of the 95% confidence interval:

$$\text{CI}_{95} = t_{0.975, n-1} \cdot \frac{s}{\sqrt{n}} \quad (2)$$

Pairwise comparisons use the two-sided Mann–Whitney U test (non-parametric; does not assume normality of timing data). Effect sizes are quantified with Cohen's d

using pooled standard deviation; thresholds are $|d| < 0.2$ (negligible), **0.2–0.5** (small), **0.5–0.8** (medium), **> 0.8** (large).

4. Results

4.1. Phase-Level Timing (Cold-Loading Scenarios)

Table 2 presents descriptive statistics for the three cold-loading scenarios. The best value in each column is shown in bold.

Table 2: Phase-level mean $\pm$ standard deviation after IQR outlier removal

Scenario (n)	Load (ms)	Type (ms)	Create (ms)	Invoke (ms)	Total (ms)	CI ₉₅ (ms)
Java UCL (n=940)	0.138 $\pm$ 0.090	0.626 $\pm$ 0.287	0.103 $\pm$ 0.080	0.111 $\pm$ 0.066	0.977 $\pm$ 0.430	$\pm$ 0.028
Java JPMS (n=958)	0.719 $\pm$ 0.341	0.393 $\pm$ 0.178	0.099 $\pm$ 0.061	0.088 $\pm$ 0.055	1.299 $\pm$ 0.532	$\pm$ 0.034
.NET ALC (n=885)	0.040 $\pm$ 0.021	0.030 $\pm$ 0.010	0.216 $\pm$ 0.062	0.132 $\pm$ 0.033	0.419 $\pm$ 0.104	$\pm$ 0.007

Several findings are immediately apparent:

- Java UCL – The "Find Type" phase constitutes the majority of the total execution time (64% of total). `loadClass()` scans the JAR zip structure on every iteration since each call constructs a fresh `URLClassLoader`, bypassing any per-loader class cache.
- Java JPMS – Module loading represents the primary bottleneck, accounting for 55% of the total overhead. `ModuleLayer` construction involves dependency-graph resolution (`Configuration.resolve()`) and module-descriptor parsing, a fixed cost absent in the `URLClassLoader` path.
- .NET ALC – The "Create" and "Invoke" phases jointly dominate the total time, accounting for 83% of the overhead. Assembly loading (0.040 ms) and type lookup (0.030 ms) are the fastest across all scenarios; reflective instantiation (0.216 ms) and invocation (0.132 ms) are the bottleneck.

4.2. Distribution and Percentile Statistic

Table 3 shows selected distribution statistics. `URLClassLoader` and JPMS exhibit heavier right tails (P_{99}/x of 2.24 and 2.09, respectively) compared to .NET ALC (1.81), suggesting that intermittent OS-level delays affect JVM JAR access more than .NET PE-file mapping.

Table 3: Distribution statistics for total iteration time

Scenario	Med. (ms)	Min (ms)	Max (ms)	P_{95} (ms)	P_{99} (ms)
Java UCL	0.883	0.371	2.306	1.834	2.191
Java JPMS	1.246	0.441	2.865	2.387	2.714
.NET ALC	0.385	0.296	0.792	0.649	0.759

4.3. Statistical Significance Testing

Table 4 reports pairwise Mann–Whitney U test results and Cohen’s d for the three cold-loading scenario comparison.

Table 4: Pairwise Mann–Whitney U test results (total time)

Pair	p-value	Cohen’s d
UCL vs. JPMS	<0.001	-0.66 (medium)
UCL vs. ALC	<0.001	1.77 (large)
JPMS vs. ALC	<0.001	2.26 (large)

All three comparisons are statistically significant at $\alpha = 0.001$ and all effect sizes meet the *large* criterion ($|d| > 0.8$), except the UCL–JPMS pair which is *medium*.

4.4. Cached-Reflection and Direct-Invocation Baselines

Table 5 contextualises the cold-loading results against scenarios where module metadata is reused or bypassed entirely.

Table 5: Mean total times for the baseline scenario and overhead relative to the corresponding direct-invocation baseline

Scenario (n)	Total (ms)	Ratio vs. Direct
Java UCL (n=940)	0.977	≈ 2257
Java JPMS (n=958)	1.299	≈ 3000
.NET ALC (n=885)	0.419	≈ 794
Java Cached (n=909)	0.003	≈ 6.4
.NET Cached (n=933)	0.001	≈ 1.1
Java Direct (n=839)	0.0004	1 (baseline)
.NET Direct (n=918)	0.0005	1 (baseline)

Once class/assembly metadata is cached, both platforms drop to sub-10 us create/invoke cycles (Java Cached: 0.003 ms; .NET Cached: 0.001 ms), converging towards the direct-invocation baseline. The cold-loading overhead is therefore dominated by the module-loading and type-resolution phases rather than the reflection API itself.

4.5. Memory Consumption

Mean per-iteration heap allocations: Java UCL 0.034 +/- 0.259 MB; Java JPMS 0.081 +/- 0.395 MB; .NET ALC 0.0019 +/- 0.0033 MB. The .NET runtime reclaims assembly memory deterministically after `AssemblyLoadContext.Unload()`, whereas the JVM retains class metadata until the next GC cycle, resulting in measurably higher cumulative heap pressure for repeated cold-loading scenarios.

5. Discussion

5.1. Hypothesis Evaluation

H1 - Measurable differences across scenarios. Confirmed. The total mean times span from 0.419 ms (.NET ALC) to 1.299 ms (Java JPMS), a factor of 3.10x, with all pairwise differences statistically significant at $p < 0.001$ (Cohen's d : 0.66 – 2.26). H1 is confirmed.

H2 - JPMS ModuleLayer overhead vs. URLClassLoader. Confirmed. ModuleLayer construction is 5.2x slower than URLClassLoader module loading (0.719 ms vs. 0.138 ms). At the scenario level, JPMS is also slower overall than Java URLClassLoader (1.299 ms vs. 0.977 ms, $p < 0.001$, $d = -0.66$, medium effect). This overhead originates from the mandatory dependency-graph resolution and module-descriptor parsing in `Configuration.resolve()`, which constitutes a one-time cost per layer and makes JPMS unsuitable for high-frequency cold-loading. JPMS partially compensates in type resolution: once the layer is built, `loadClass()` (0.393 ms) is 1.59x faster than URLClassLoader (0.626 ms), because module metadata is pre-indexed at layer construction time.

H3 - .NET ALC faster to load; slower to create/invoke. Partially confirmed. Assembly loading in .NET (0.040 ms) is 3.5x faster than Java UCL (0.138 ms) and 18x faster than JPMS (0.719 ms). Type resolution via `GetType()` (0.030 ms) is 20.9x faster than URLClassLoader.`loadClass()` (0.626 ms), because the PE binary format is fully memory-mapped and directly indexed by the CLR. These advantages are partially offset by reflective instantiation via `Activator.CreateInstance()` (0.216 ms), which is 2.1x slower than Java (0.103 ms), and method invocation (0.132 ms vs. 0.111 ms, only 19% slower). Overall, .NET ALC (0.419 ms) is 2.33x faster in total than Java UCL (0.977 ms), primarily due to the overwhelming PE-metadata advantage in loading and type resolution (Cohen's $d = 1.77$, large effect).

5.2. Practical Implications

Based on the experimental results, the following usage guidelines emerge:

- Cold-path plugin loading (loading a module once per session): URLClassLoader and ALC remain sub-millisecond on average, whereas JPMS remains within a low single-millisecond range. JPMS should be avoided in scenarios where a new module layer is instantiated per request, due to its 5.2x loading overhead.
- Framework / DI use-cases (many constructor or method invocations on a cached type): caching Constructor and Method references on JVM, and ConstructorInfo/MethodInfo on .NET, eliminates the loading overhead and brings both platforms to sub-10 us create/invoke cycles.
- Memory-constrained environments: .NET ALC offers roughly 18–44x lower per-iteration heap consumption than the Java cold-loading scenarios and supports deterministic unloading via `Unload()`, which is advantageous when assembly instances must be collected promptly.

5.3. Threats to Validity

Internal validity. Timer granularity in the sub-microsecond range may introduce noise for individual phases; however, aggregation over $N=1000$ post-warm-up iterations renders this noise negligible ($CI_{95} < 0.034$ ms for all cold-loading scenarios). Benchmarks run single-threaded to avoid scheduling noise.

External validity. Results may not generalise to: (a) OS platforms other than Windows; (b) JVM/CLR implementations other than HotSpot 21 and .NET 10; (c) modules with complex class hierarchies or many transitive dependencies. Network-mounted file systems and anti-virus I/O filters may inflate JAR/DLL loading times substantially.

Construct validity. The `compute()` workload is intentionally minimal to expose reflection overhead. Real-world method bodies are typically larger, which would dilute the relative importance of invocation overhead but would not affect the loading and type-resolution phases.

6. Conclusions

This paper conducted a controlled four-phase benchmark comparing three dynamic module-loading mechanisms on Java 21 (JVM) and .NET 10 (CLR): URLClassLoader, JPMS ModuleLayer, and AssemblyLoadContext. After JIT warm-up and IQR outlier removal, the key findings are:

1. .NET ALC achieves the lowest total mean time (0.419 ms, $CI_{95} + -0.007$ ms), primarily due to fast PE memory-mapping and direct ECMA-335 metadata indexing.
2. Java JPMS incurs the highest total mean time (1.299 ms), with ModuleLayer construction alone consuming 55% of the budget-5.2x more than URLClassLoader module loading.
3. .NET type resolution is 20.9x faster than Java URLClassLoader (0.030 ms vs. 0.626 ms); reflective

instantiation is 2.1x slower in .NET (0.216 ms vs. 0.103 ms).

4. All pairwise differences are statistically significant ($p < 0.001$, *Mann – Whitney U*) with Cohen's *d* ranging from 0.66 (medium, UCL vs. JPMS) to 2.26 (large, JPMS vs. ALC).

For high-frequency cold-loading use cases .NET ALC is the fastest option, while Java URLClassLoader represents a balanced middle ground. JPMS should be preferred for its encapsulation and access-control guarantees rather than loading performance. Both platforms converge to sub-10 us with cached reflection, confirming that the dominant overhead lies in loader construction and type resolution rather than in cached reflective invocation itself

References

- [1] R. Johnson, Expert One-on-One J2EE Design and Development, Wrox Press, Birmingham, 2002.
- [2] M. D. McIlroy, Mass Produced Software Components, in P. Naur, B. Randell (eds.), Software Engineering, Scientific Affairs Division, NATO, Brussels, 1969, pp. 138–155.
- [3] OSGi Alliance, OSGi Core Release 8 Specification, <https://docs.osgi.org/specification/>, [03.04.2026].
- [4] Oracle Corporation, The Reflection API - Java SE 21, <https://docs.oracle.com/javase/tutorial/reflect/>, [03.04.2026].
- [5] Microsoft Corporation, Reflection in .NET, <https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/overview>, [03.04.2026].
- [6] M. Reinhold, J. Steffan, JSR 376: Java Platform Module System, <https://jcp.org/en/jsr/detail?id=376>, [03.04.2026].
- [7] Microsoft Corporation, AssemblyLoadContext Class - .NET 10, <https://learn.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=net-10.0>, [03.04.2026].
- [8] Microsoft Corporation, .NET Modularity and Plugins using AssemblyLoadContext, <https://learn.microsoft.com/en-us/dotnet/core/tutorials/creating-app-with-plugin-support>, [03.04.2026].
- [9] I. Forman, N. Forman, Java Reflection in Action, Manning Publications, Greenwich, 2004.
- [10] M. Pradel, P. Ratanaworabhan, T. R. Gross, Lean and Mean: Efficient Dynamic Dispatch in Prototype-Based Programs, Proceedings of ECOOP 7 (2011) 195-219.
- [11] B. Fulgham, S. Nanz, The Computer Language Benchmarks Game, <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>, [03.04.2026].
- [12] M. Paleczny, C. A. Vick, C. Click, The Java HotSpot Server Compiler, Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM '01), Monterey, CA, USA, April 23–24, 2001, USENIX Association, pp. 1–12.
- [13] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, ACM SIGPLAN Notices 42(10) (2007) 57-76.
- [14] A. Shipilev, Java Microbenchmark Harness (JMH), GitHub repository, OpenJDK, <https://github.com/openjdk/jmh>, [03.04.2026].
- [15] J. W. Tukey, Exploratory Data Analysis, Addison-Wesley, Reading, MA, 1977.
- [16] H. B. Mann, D. R. Whitney, On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other, The Annals of Mathematical Statistics 18(1) (1947) 50-60.
- [17] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, 2nd ed., Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.