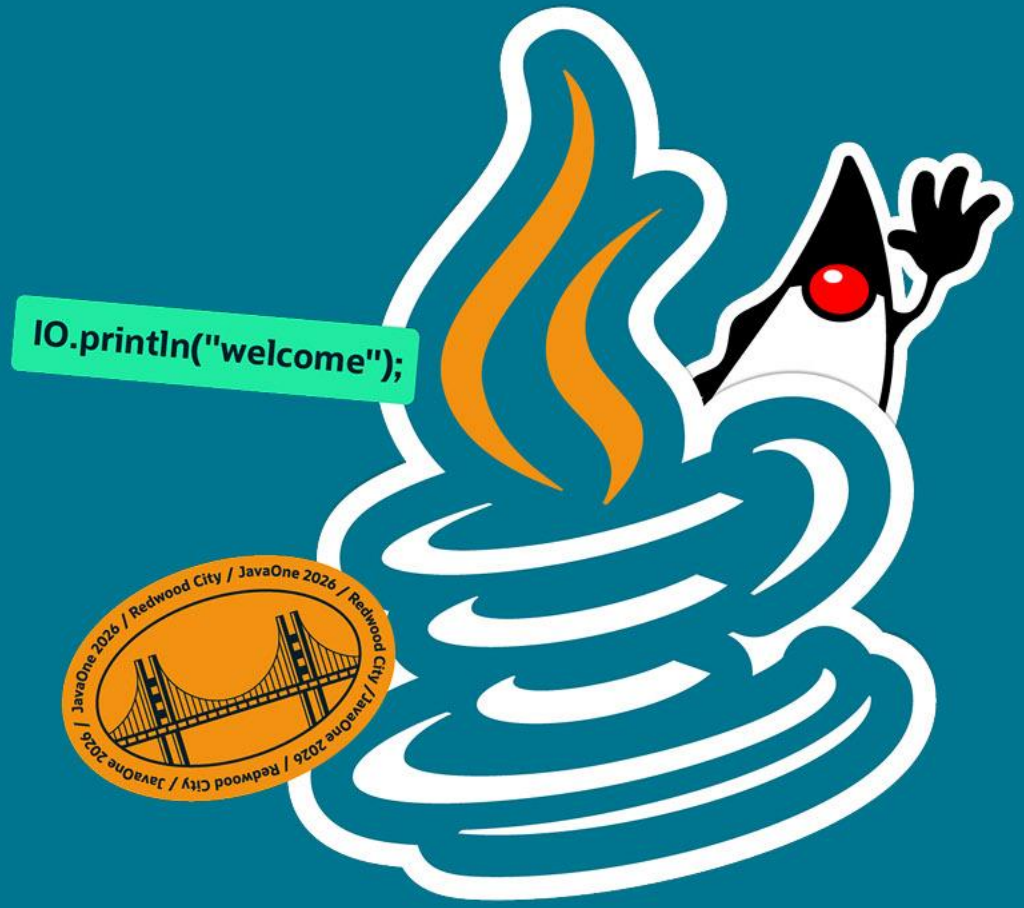
JavaOne™

Training Java:

Ahead of Time Updates from Project Leyden



Dan Heidinga

JVM Runtime Architect

Java Platform Group, Oracle

March 18, 2026

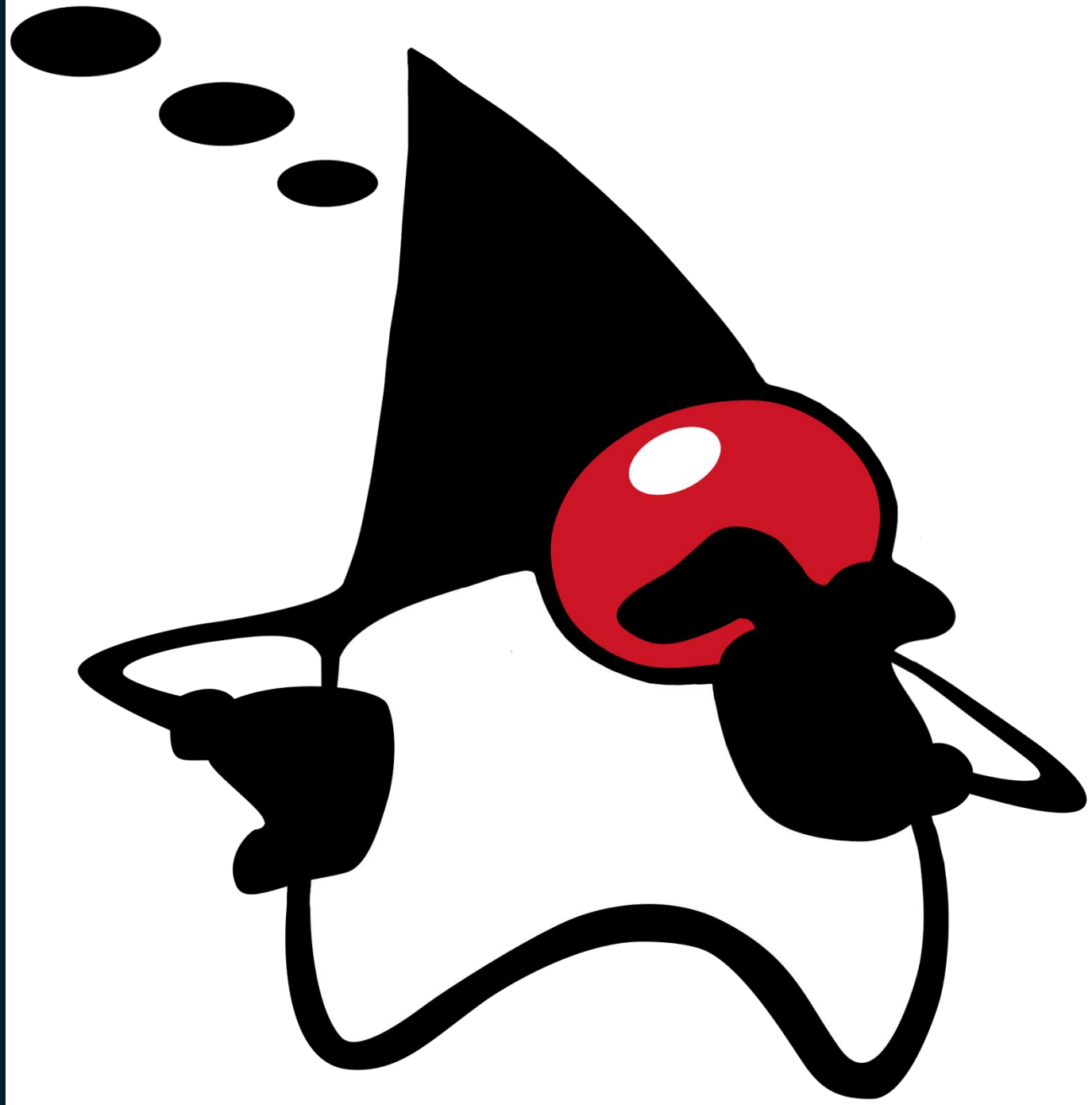
Project Leyden

Improve the startup time
and warmup time
of Java applications

by shifting computation
earlier or later in time.

Static Analysis

Dynamic Observation



Training Runs

Purpose

Exercise the startup and warmup code paths, under observation.

Discover ahead-of-time what you'd otherwise find early at runtime.

What Constitutes a Training?

The best training run is observing the application in production.

Usually needs writing a small driver program (like an integration test).

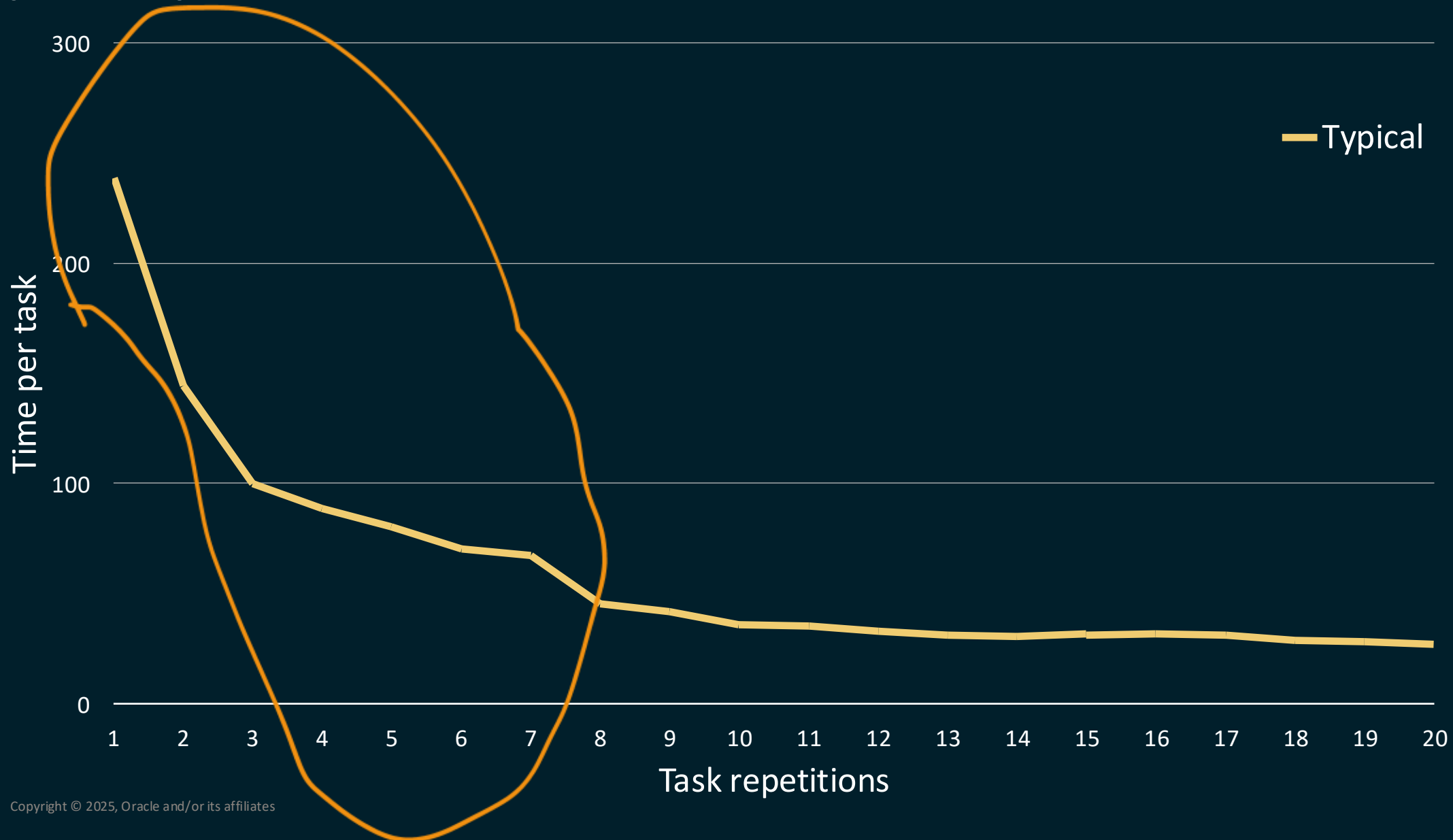
Runs at build time (similar to an integration test).

Why They Work?

Effective for the same reason dynamic compilation is.

Analysis is driven by what the program actually does.

Application performance



JDK 18
JDK 19
JDK 20
JDK 21
JDK 22
JDK 23
JDK 24
JDK 25
JDK 26
JDK 27
JDK 28
JDK 29
JDK …

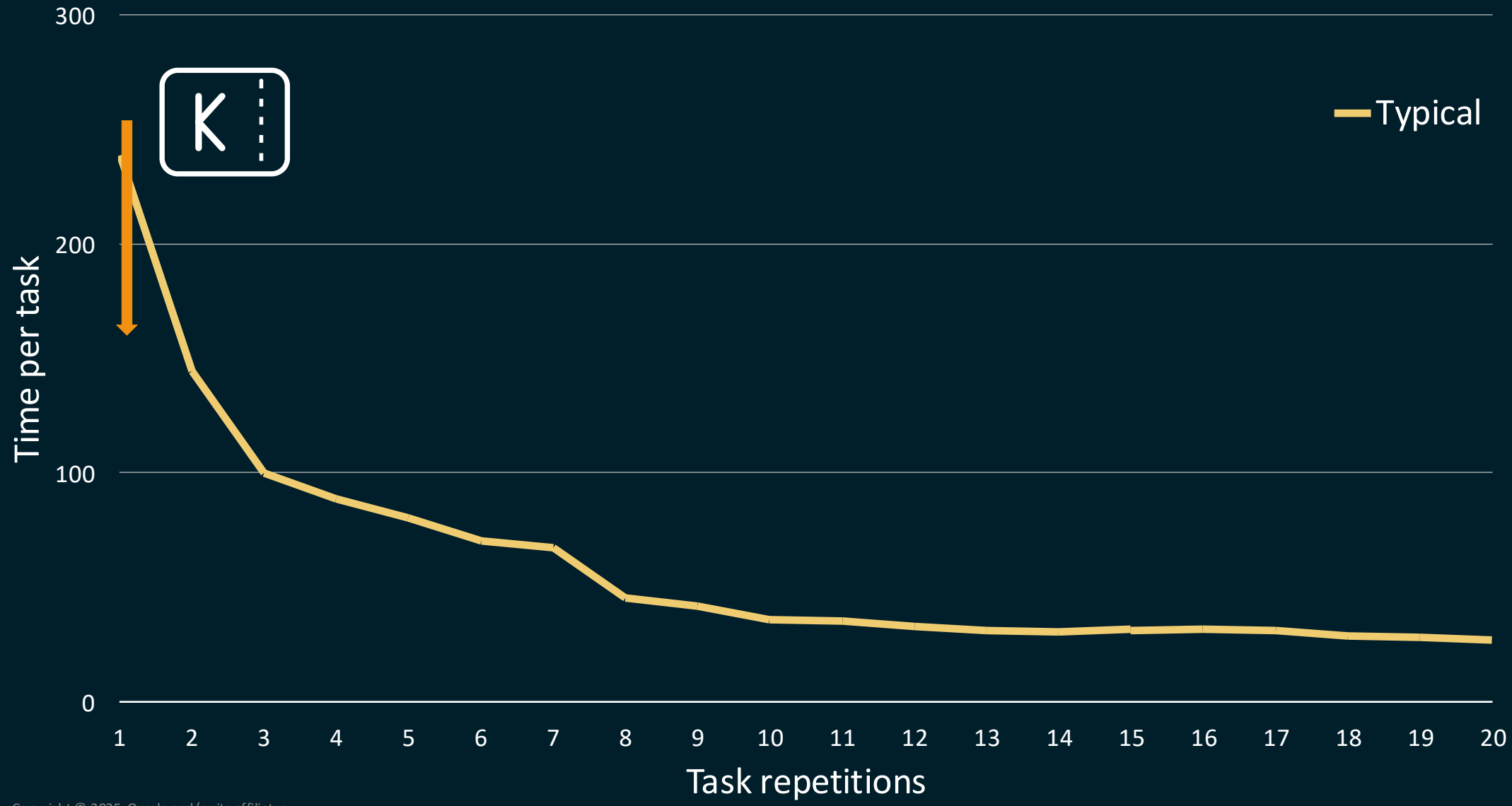
JEP 483: Ahead-of-Time Class Loading & Linking

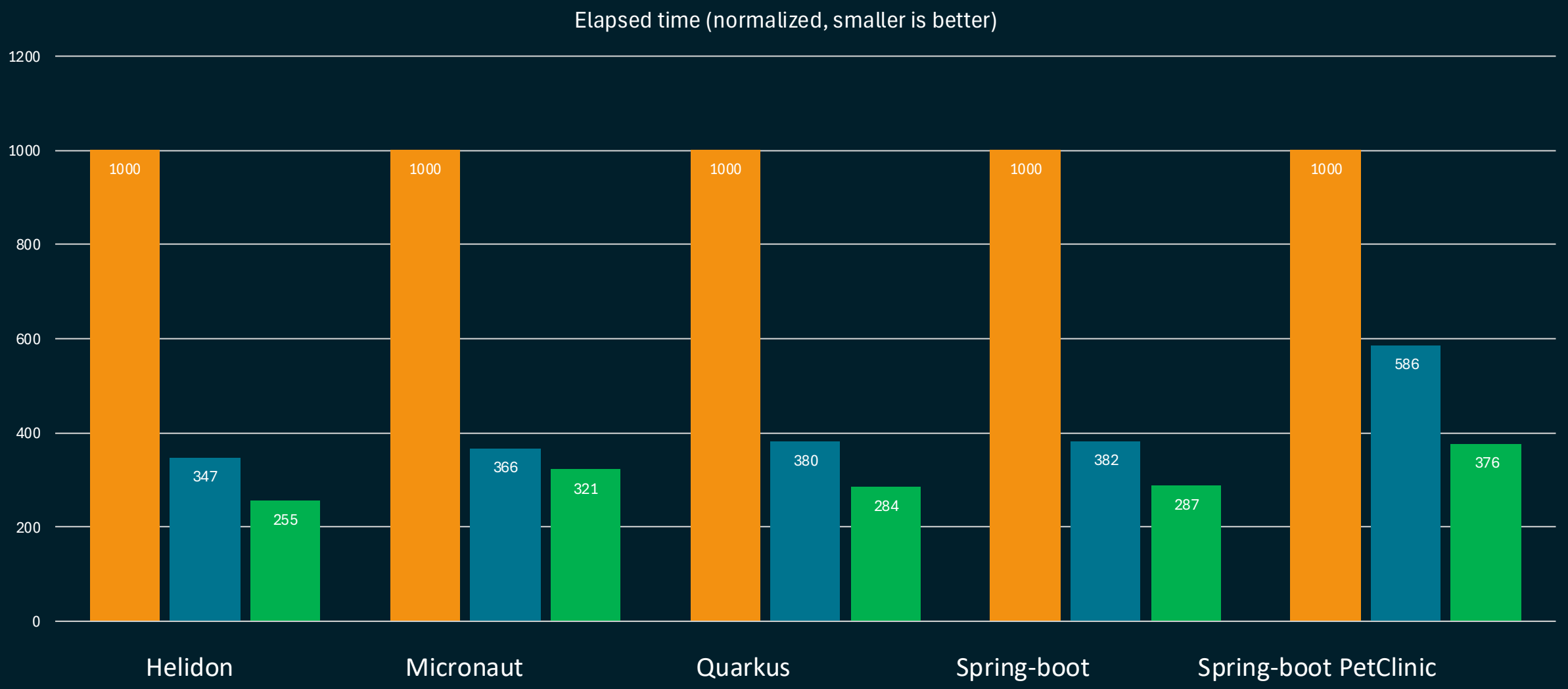
Authors	Ioi Lam, Dan Heidinga, & John Rose
Owner	Ioi Lam
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	24
Component	hotspot / runtime
Discussion	leyden dash dev at openjdk dot org
Reviewed by	Alex Buckley, Brian Goetz, Mark Reinhold, Vladimir Kozlov
Endorsed by	Vladimir Kozlov

JDK 24’s Three-Phase Workflow

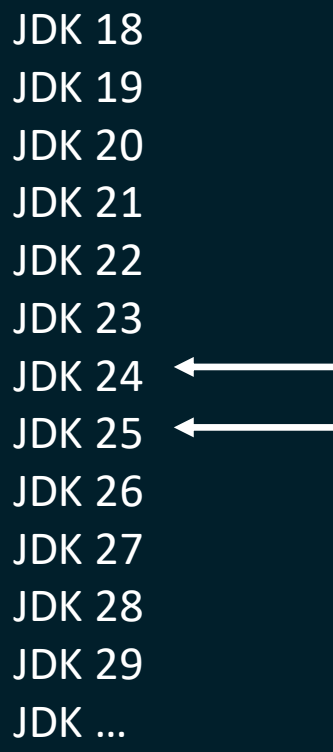
```
$ # Training Run
$ java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconf \
       -cp app.jar com.example.App ...

$ # Assembly Phase
$ java -XX:AOTMode=create -XX:AOTConfiguration=app.aotconf \
       -XX:AOTCache=app.aot -cp app.jar

$ # Deployment Run
$ java -XX:AOTCache=app.aot -cp app.jar com.example.App ...
```

Application performance: Preload and Prelink classes





Source: https://github.com/openjdk/leyden/blob/634547513c2a2b707ae43a735dc24fd1977da2ae/README.md



JEP 483: Ahead-of-Time Class Loading & Linking

Authors	Ioi Lam, Dan Heidinga, & John Rose
Owner	Ioi Lam
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	24
Component	hotspot / runtime
Discussion	leyden dash dev at openjdk dot org
Reviewed by	Alex Buckley, Brian Goetz, Mark Reinhold, Vladimir Kozlov
Endorsed by	Vladimir Kozlov

JEP 515: Ahead-of-Time Method Profiling

Author	Igor Veresov & John Rose
Owner	John Rose
Type	Feature
Scope	Implementation
Status	Proposed to Target
Release	25
Component	hotspot / compiler
Discussion	leyden dash dev at openjdk dot org
Effort	M

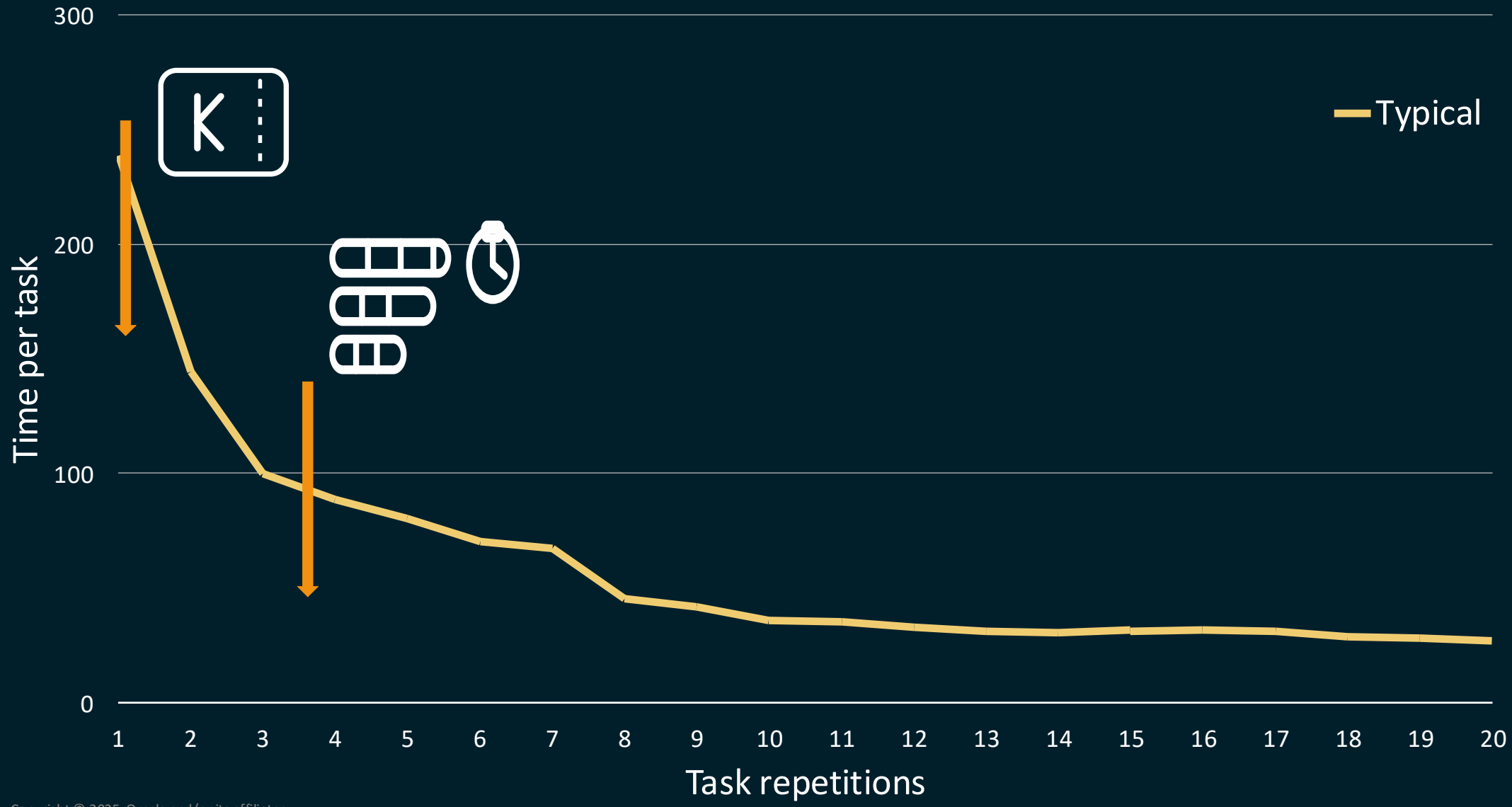
JEP 514: Ahead-of-Time Command-Line Ergonomics

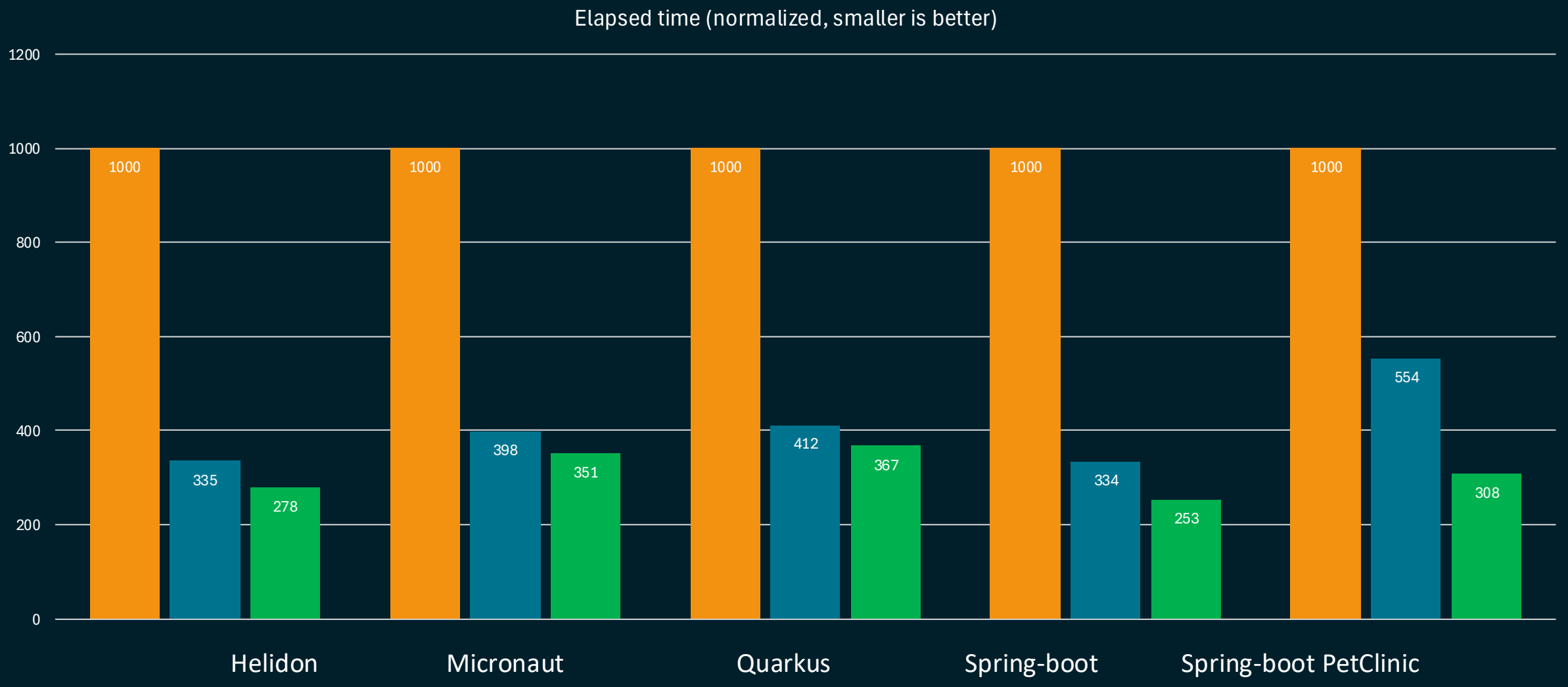
Owner	John Rose
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	25
Component	hotspot / runtime
Discussion	leyden dash dev at openjdk dot org
Effort	M
Duration	S
Relates to	JEP 483: Ahead-of-Time Class Loading & Linking

JDK 25’s Two-Phase Workflow

```
$ # Training Run + Assembly Phase
$ java -XX:AOTCacheOutput=app.aot \
       -cp app.jar com.example.App ...

$ # Deployment Run
$ java -XX:AOTCache=app.aot -cp app.jar com.example.App ...
```

Application performance: AOT Method Profiles

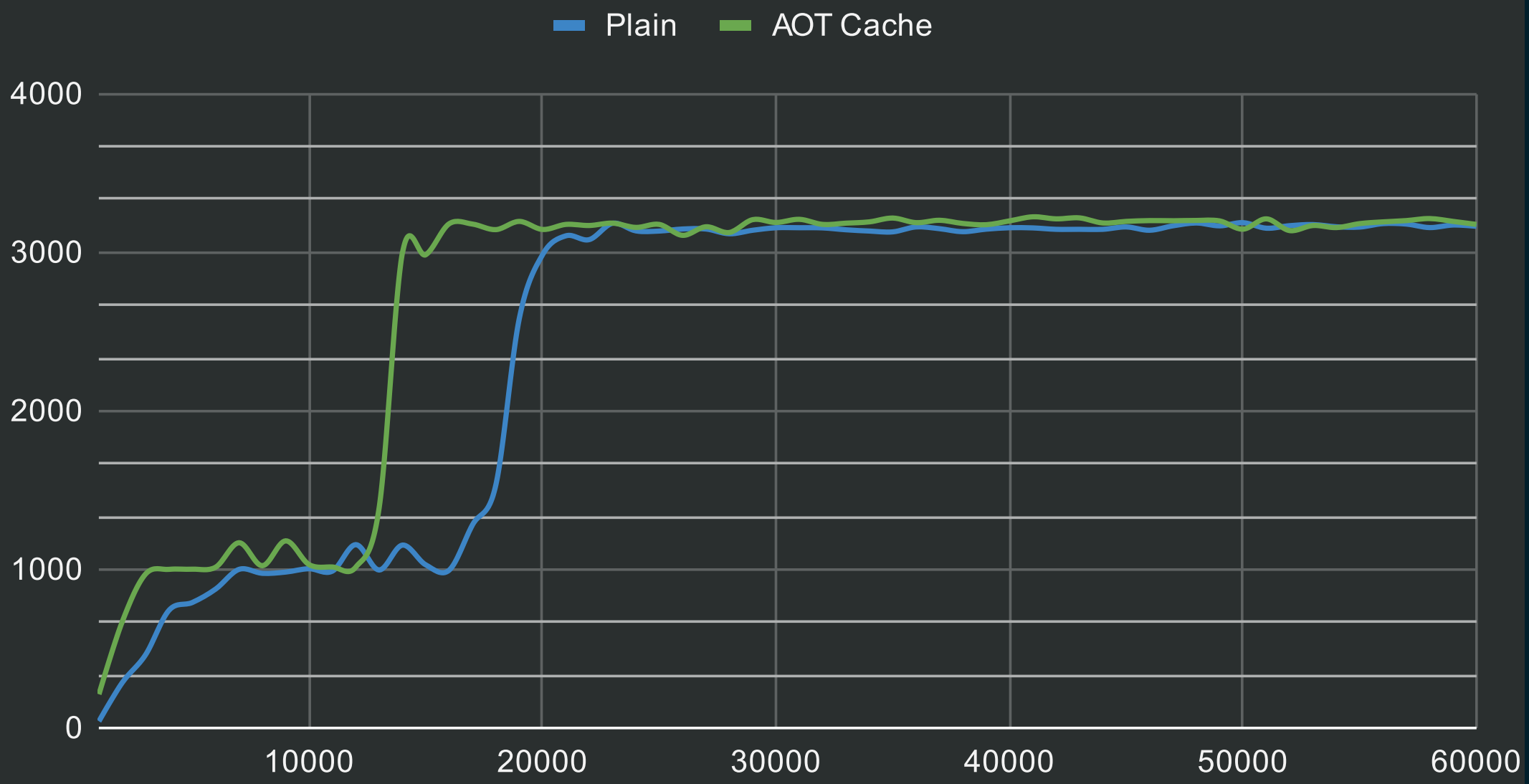




Source: https://github.com/openjdk/leyden/blob/premain/README.md

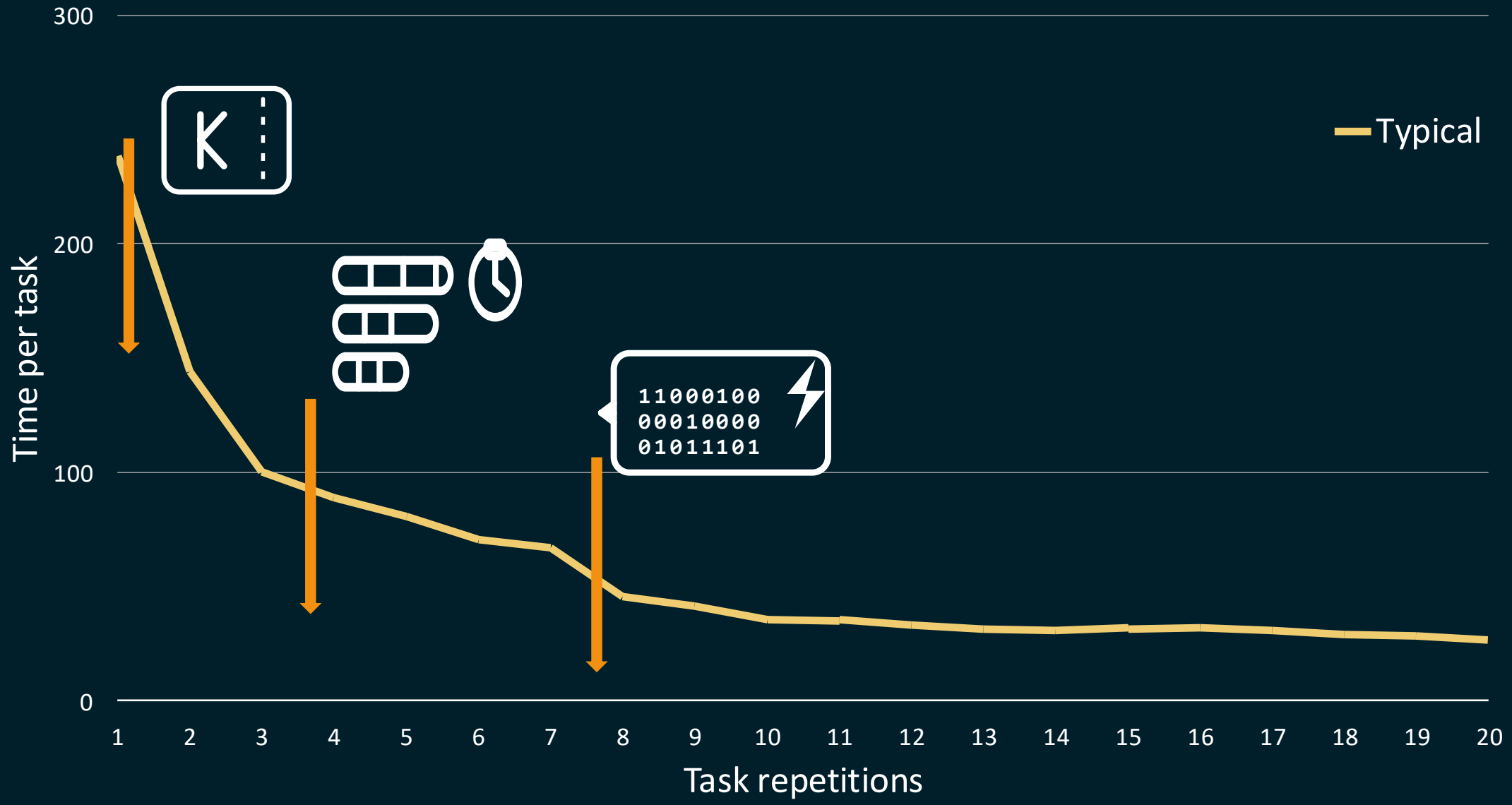
Requests / second (1 CPU)

Spring Boot 4.0.0-M3, Java 25



From: Supercharge your JVM performance with Project Leyden and Spring Boot,
Ana-Maria Mihalceanu & Moritz Halbritter, 2026

Application performance: AOT compiled code



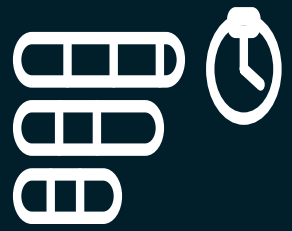
That seems simple, why aren’t you done yet?

Preloaded and Prelinked Classes

- Requires deterministic classloaders
- Requires consistent classpaths (i.e. same jars)
- Must be loaded as a group
- Enables pre-resolving constantpool indexes
- Enables pre-resolving invokedynamics

Cached Profile information

- Records the behaviour of the application
- Method counts, branch frequencies, classes encountered
- Not sufficient to collect profiles at JVM exit
- Different compilations have different views of the profiling data



Precompiled code

- Precompile the method to reuse the generated code in later runs
- Wait! Which compiled code? C1? C2?
- If profiles are more complicated than a single set of data, then maybe compiled methods are more complicated too!

Ahead-of-Time Method Profiling Deep Dive

JEP 515: uses profile data from training runs to shift JIT compilation earlier in production runs

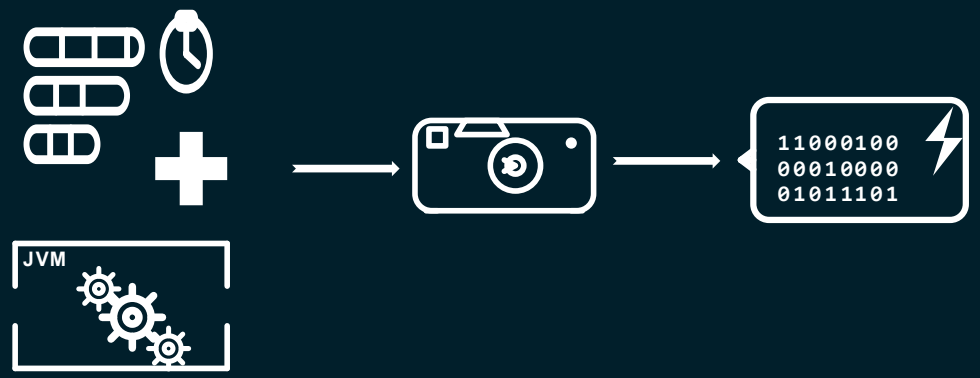
Profile data is collected during execution in MethodData Objects (MDO) and can be persisted for AOT use:

- Method invocation counts
- Loop counts
- Callsite type profiles (receiver type distribution)
- Branch taken/not taken counts
- Megamorphic dispatch sites

Other information is implicit in the environment and queried when a method is compiled

- Which classes are loaded
- Which classes are initialized
- Which methods are already compiled for inline vs call considerations
- Which tier a method has been compiled at (C1/C2)

Collecting the right profiles at the right time



A JIT compiled method is the product of the current profiles and the state of the JVM.

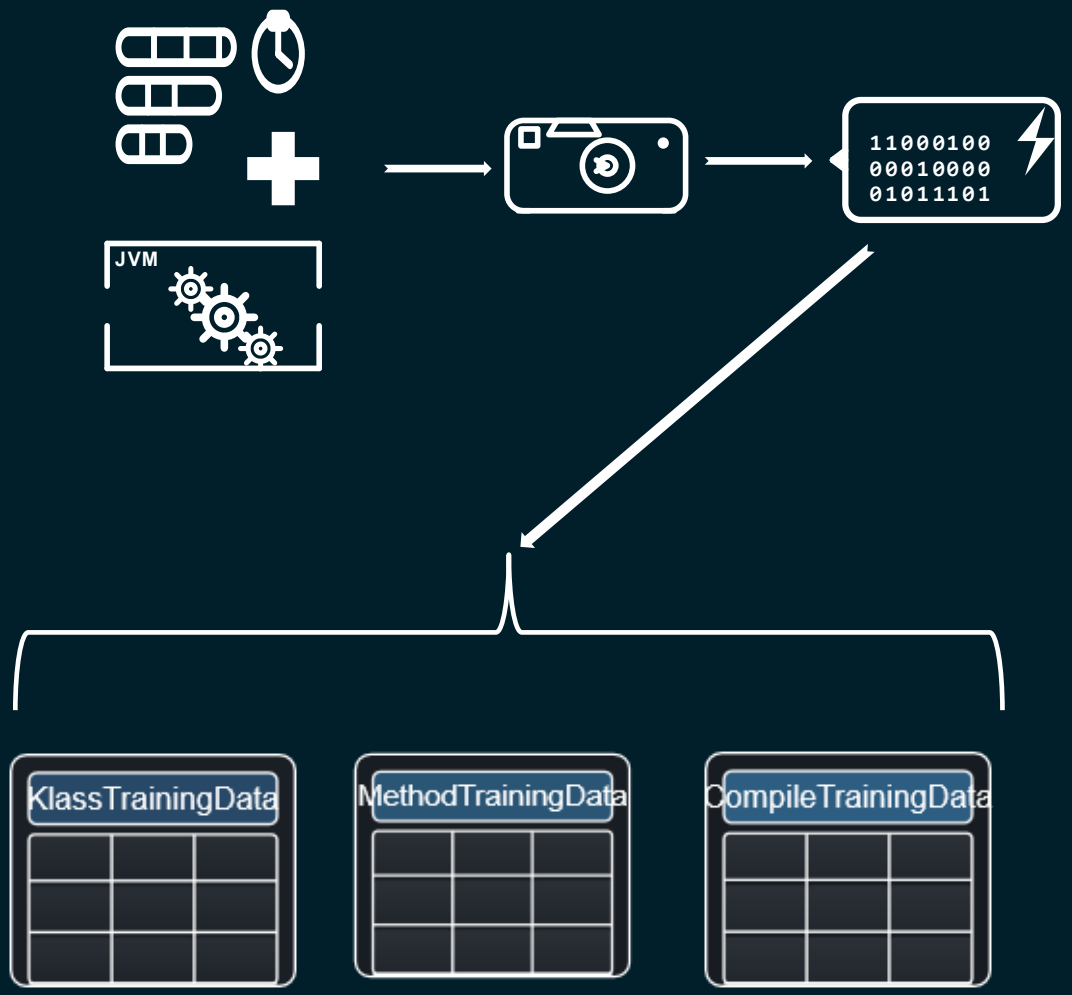
If the profile data isn't collected until JVM exit, then it represents the state at exit; not during startup.

This is only part of the picture!

Exit profiles are less useful for optimizing startup then profiles that represent the state of the JVM during startup

- the set of loaded and initialized classes are different
- the set of resolved constantpool entries are different
- the set of already compiled methods is changing

Collecting the right profiles at the right time



The JVM remembers both the following as TrainingData:

- the profile data at the time of the compile, and
- the questions the JIT compiler asks of the JVM during that compile

Each compilation – C1 & C2 – generates unique TrainingData.

Some TrainingData must wait for the relevant classes to be loaded and initialized before it using it will be a benefit

- Using it too early just leads to bad compiles or deopts

Shifting early is good, but too early not so much

Using the wrong profiles too early can lead to worse performance:

- Required classes aren’t yet initialized resulting in less efficient code
- More uncommon traps occur resulting in deoptimization and recompilation

Using the invocation counts from the training run for all methods can cause some methods to be compiled too early

- Methods that aren’t compiled during training still increment their invocation counts
- These “lukewarm” methods never compile in training, but may jump ahead of newer methods during production due to training run method counts
- Compilation queues can be overloaded with these methods in production, delaying the right methods from JITTing

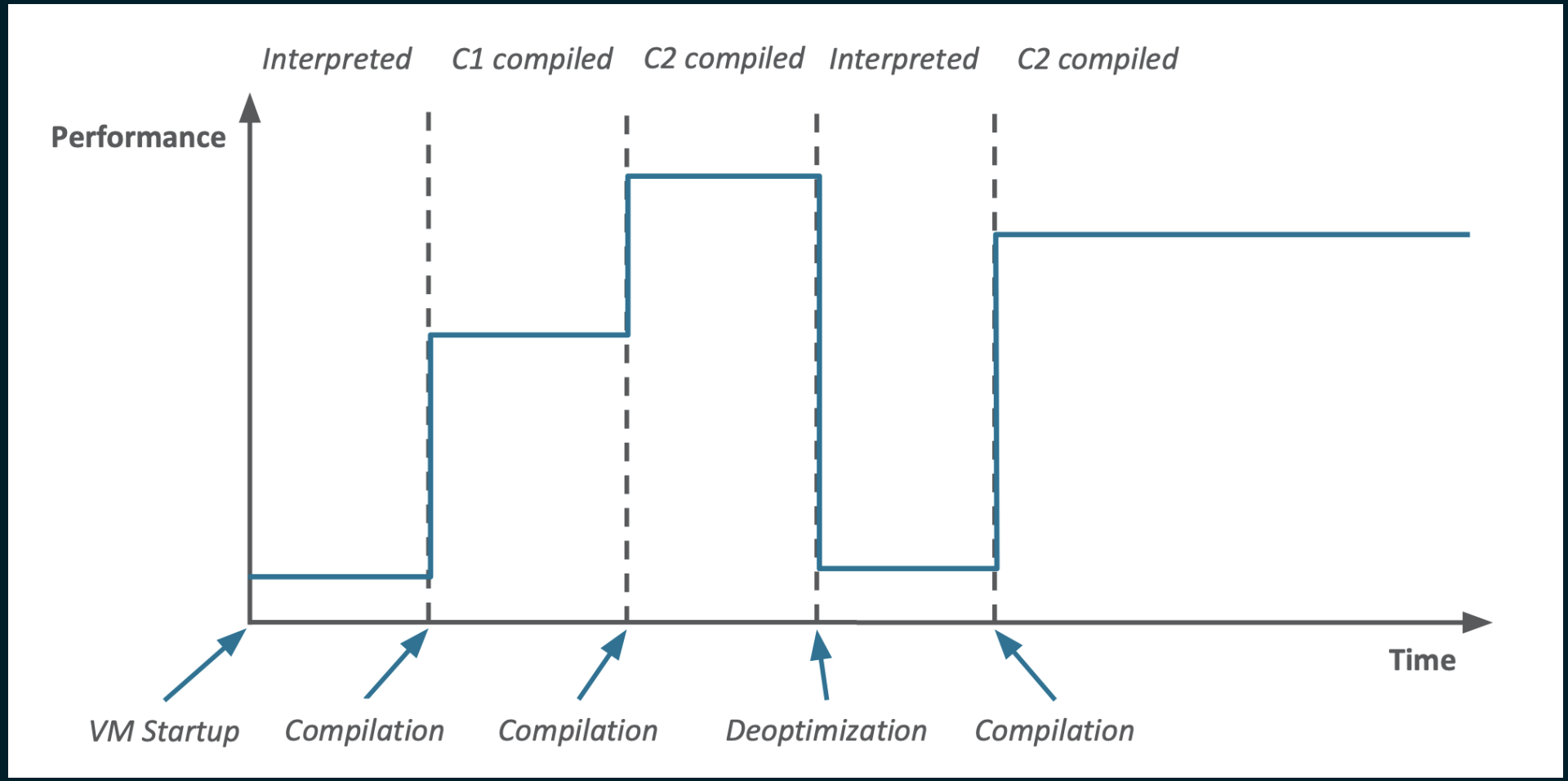
Principle: Start optimizing with our regular policy in production runs where we left off in training.

- Wait for “training run number of invocations” before lukewarm methods become eligible for compilation in production

If Ahead-of-time cached method profiles are complicated, maybe Ahead-of-time cached methods are too…..

How could performance over time look?

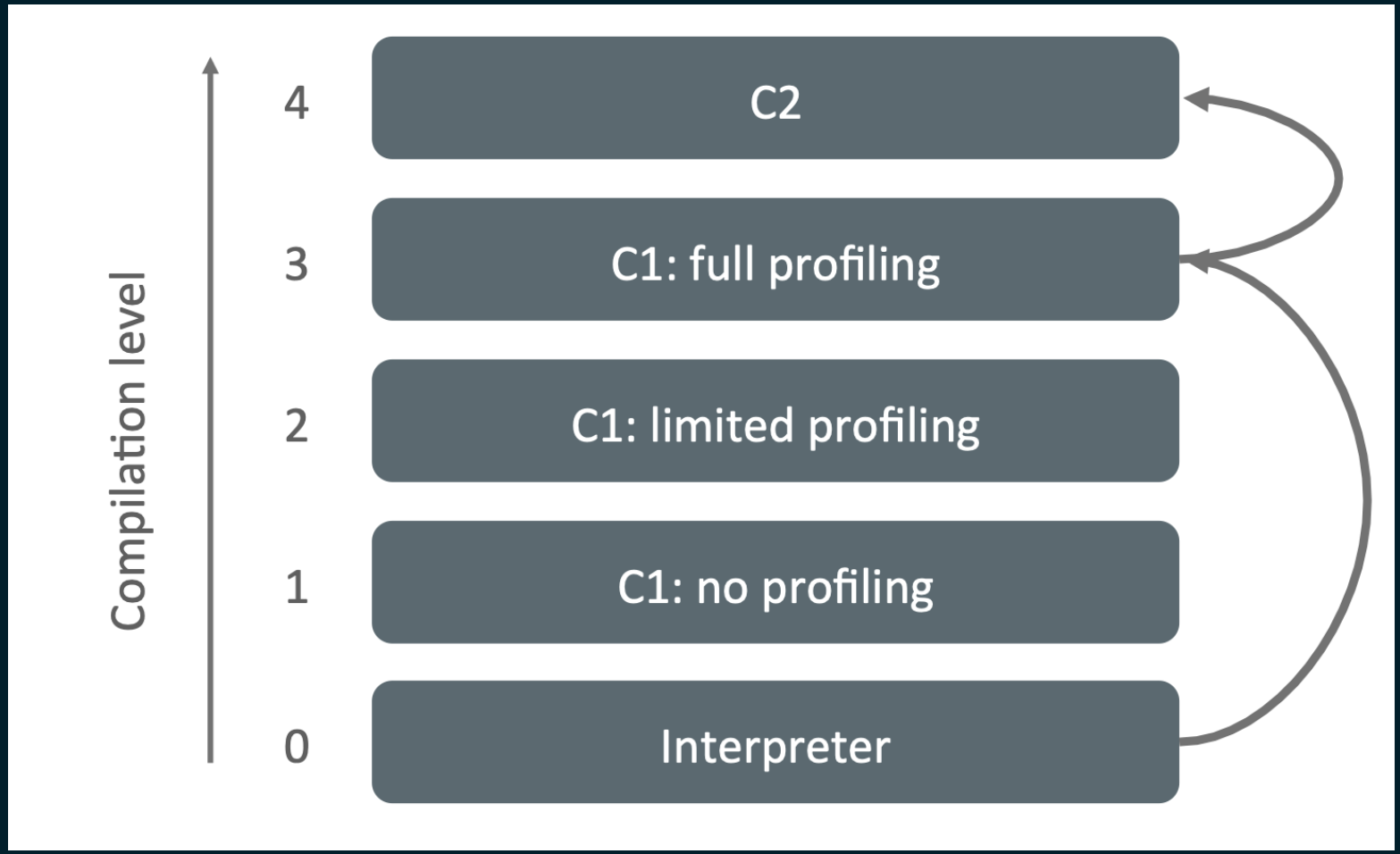
Plot implies convergence, this isn't always the case



From: *The Java HotSpot VM – Under The Hood*, Tobias Hartmann, 2020

Compilation tiers

Introducing numbers on top of acronyms



From: *The Java HotSpot VM – Under The Hood*, Tobias Hartmann, 2020

Which tier do we use for AOT compiled methods?

Some of them… All of them… None of them?

AOT compiled code is divided into two sets:

AOT Preload code (AP):

- Can handle uninitialized classes
- Is immediately applicable once the preloaded classes are installed

Fully optimized AOT code (A):

- Uses AOT profiles to generate the best possible code
- Assumes all required classes are initialized.
- Can only be installed once the class initialization dependencies are satisfied.

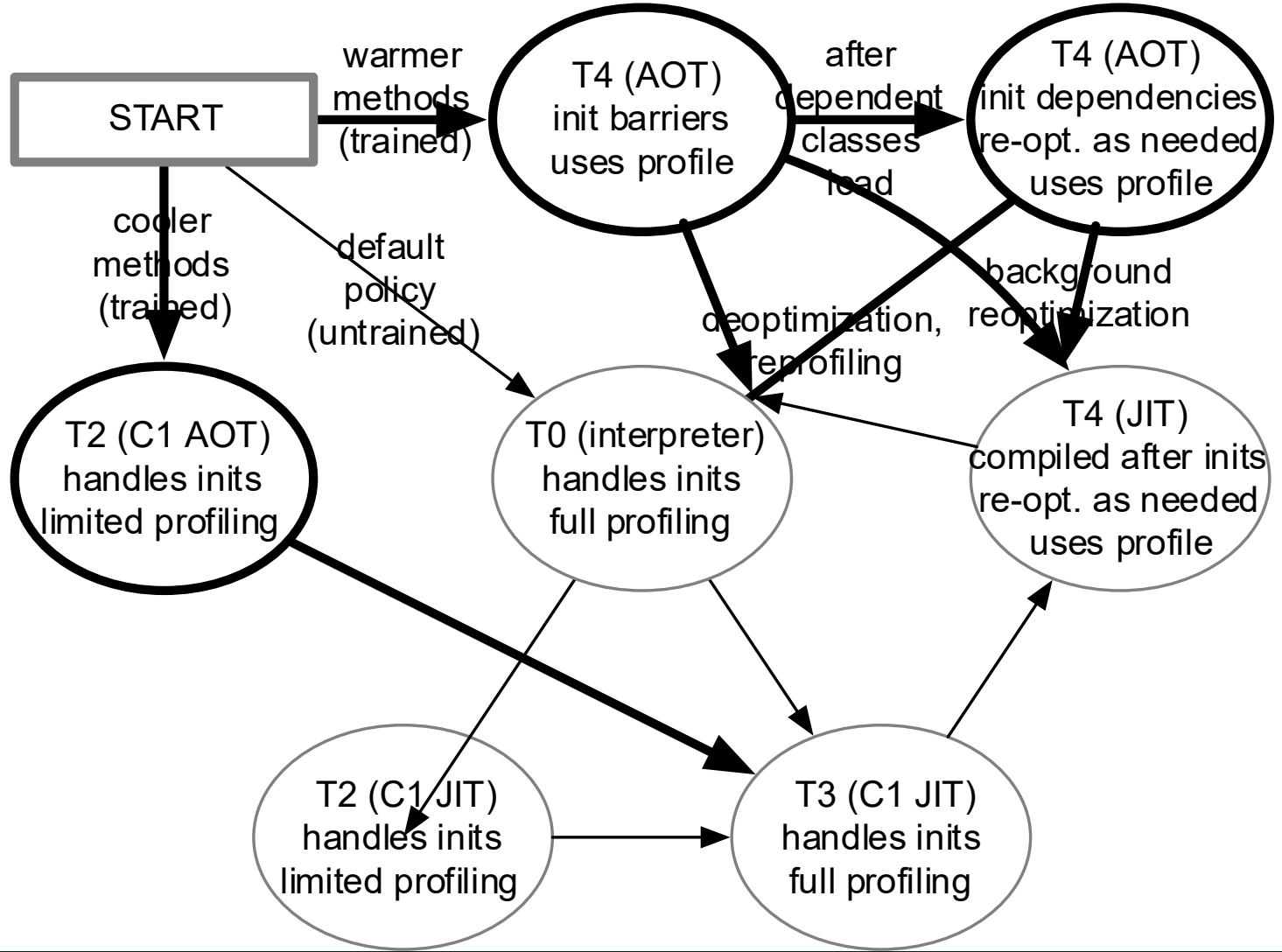
The hottest methods compiled with C2 during training get both AP and A code.

The warm methods compiled with C1 during training only get AP code.

Even the best AOT code for the hottest methods should eventually be replaced by new JIT code

- There is speculation AOT code can’t do (yet): object identities, trusted final fields, @Stable variables, some machine particulars
- ”A” code needs a recompilation hook to trigger a JIT replacement compilation

JVM execution modes and transitions,
as extended by Project Leyden.
Note: Heavy lines mark new modes & transitions.
Rare T1 states are omitted.



If Class initialization is the primary reason for two kinds of AOT code, why not *just* initialize all the classes early?

Each initialized class must be treated very carefully!

Cached state may be invalidated immediately

```
class Updater {

  static long lastUpdated = getUpdateTime();

  public static void update() {
   long time = System.currentTimeMillis();
   if (time - lastUpdated < valid_period()) {
     do_update();
     lastUpdated = time;
   }
}
```

Each initialized class must be treated very carefully!

Invariant: Training runs and deployment runs should produce consistent results

```
class A {
  static final String myString
         = StringSetter.get();
}
```

```
class B {
  static final String myString
         = StringSetter.get();
}
```

```
class StringSetter {
    static final int pick = Random.nextInt(4);

    public static String get() {
      return switch(pick) {
                  0 -> “T”;
                  1 -> ”B”;
                  2 -> “X”;
                  3 -> “W”;
      };
    }
}
```

Each initialized class must be treated very carefully!

Invariant: Training runs and deployment runs should produce consistent results

```
class A {
  static final String myString
         = StringSetter.get();
}
```

“W”

```
class B {
  static final String myString
         = StringSetter.get();
}
```

“T”

```
class StringSetter {
    static final int pick = Random.nextInt(4);

    public static String get() {
      return switch(pick) {
                0 -> “T”;
                1 -> ”B”;
                2 -> “X”;
                3 -> “W”;
        };
    }
}
```

Shifting Class initialization is still an open question.
So far, we've gotten great results without needing to go there.

The good news

Our job is to manage the complexity of:

- Collecting profiles, and using them at the right time
- Precompiling methods and handling the transitions between AP, A, and JIT compiled code
- Detecting and falling back to the safer alternatives when the AOTCache is invalidated

Our job is to make your application start and warmup faster.

Your job is to:

- Figure out how to train your application
  - Canary deployment
  - Integration test
  - Something else?
- Figure out how to deploy the AOTCache with your application

Reap the continued benefits as improve the AOTCache

What’s next for Leyden?

Deliver the Ahead-of-time compiled code JEP

- Keep tuning the policy and heuristics for transitions between AOT and JIT code
- Identify and close any remaining performance gaps between AOT and JIT code

Improve the portability of the AOTCache between different micro-architectures and deployment options

- AOT compiled code targets the equivalent of a “native” setting today. A more “portable” option needs to be developed
- Portable mode would allow the cached code to be applicable across more hardware generations
- Portability is not just code. It’s also heap sizes, garbage collectors, etc.

Enable User-defined ClassLoaders to cache their classes as well

- Work underway to allow URLClassloader-subclasses to get some AOTCache classloading benefits
- Current approach provides benefits transparently, but more can be done by explicitly opting in (code changes required)

Support “iterative training” modes where an AOTCache is both an input and output for a given run

- Allows frameworks to train what they know best, and users to train on top of that with their application

Invest in training your application today

The payoff in faster startup and warmup is good today

And will just keep getting better release-on-release


ORACLE