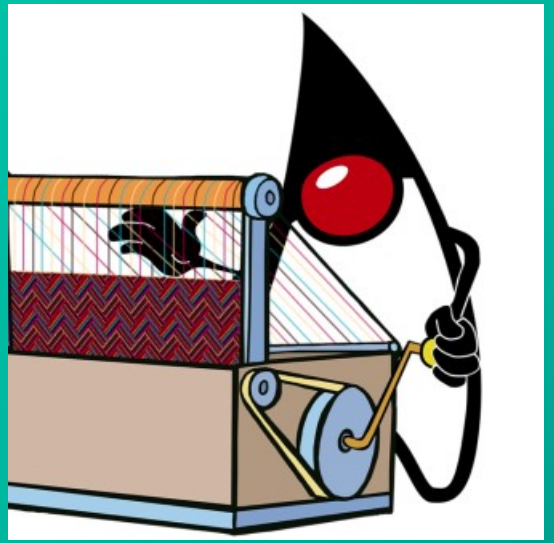
A Dev perspective on Java Loom



Jean-François JAMES

Software Architect

- Focus on Java, Jakarta EE, MicroProfile
- Open Source contributor
- Head of DevRel

Get in touch:

@jefrajames

jean-francois.james@worldline.com

linkedin.com/In/jefrajames

Who are we?

We design payments technology that powers the growth of millions of businesses around the world.

7000+ engineers
in over 40 countries

Managing 28+ billion
transactions per year

€250M spent in R&D
every year

Handling 150+
payment methods

One project, 3 JEPs

Virtual Threads

Lightweight threads

JEP 444
Stable with
Java 21 LTS

Structured Concurrency

Facilitate tasks dev& run on top of Virtual Threads

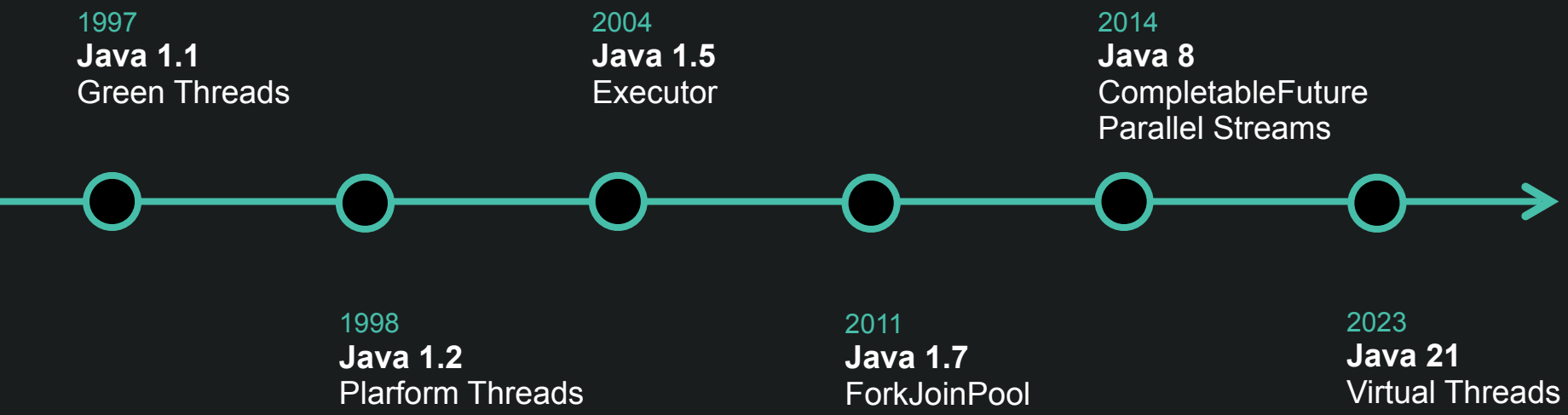
JEP 453
Preview with
Java 21 LTS

Scoped Values

Modernization, optimization of Thread Locals

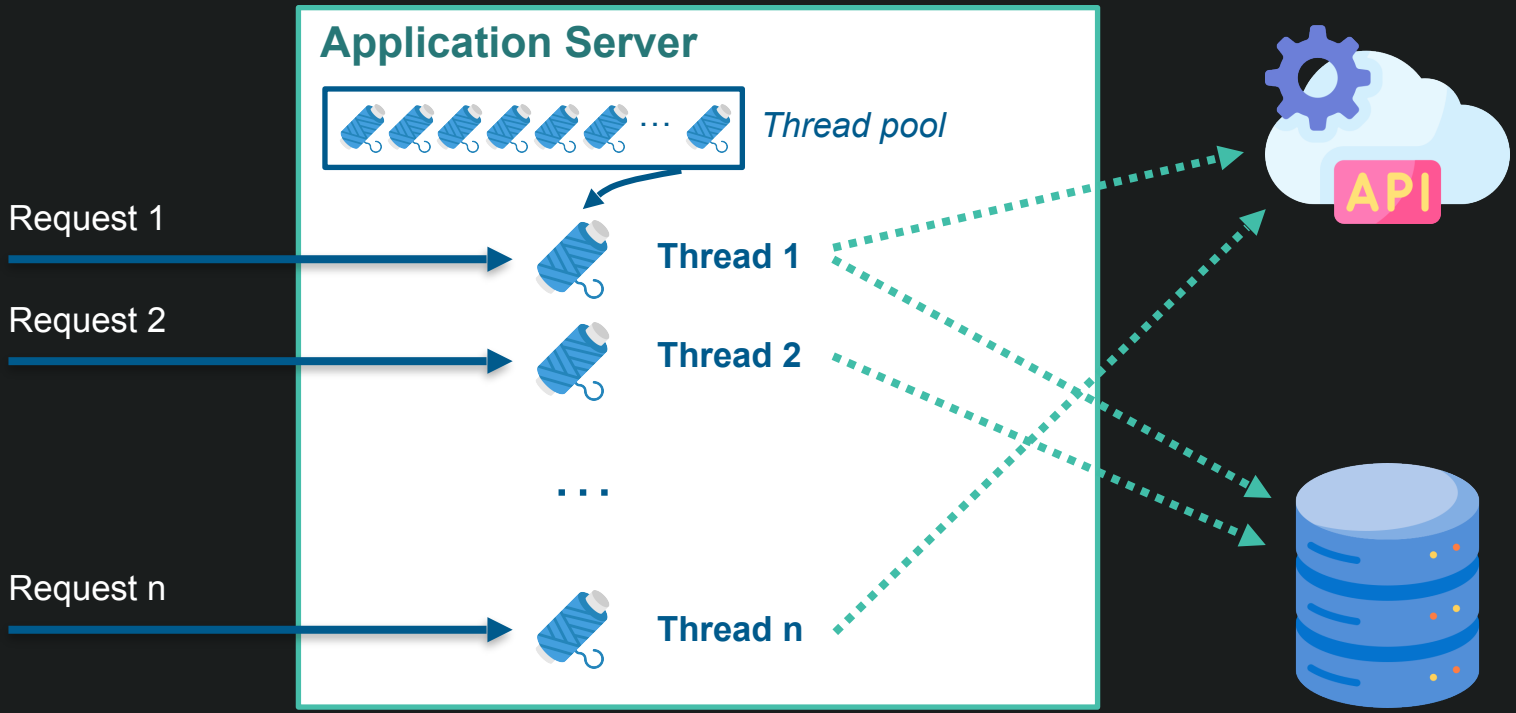
JEP 446
Preview with
Java 21 LTS

Evolution of Java Concurrency



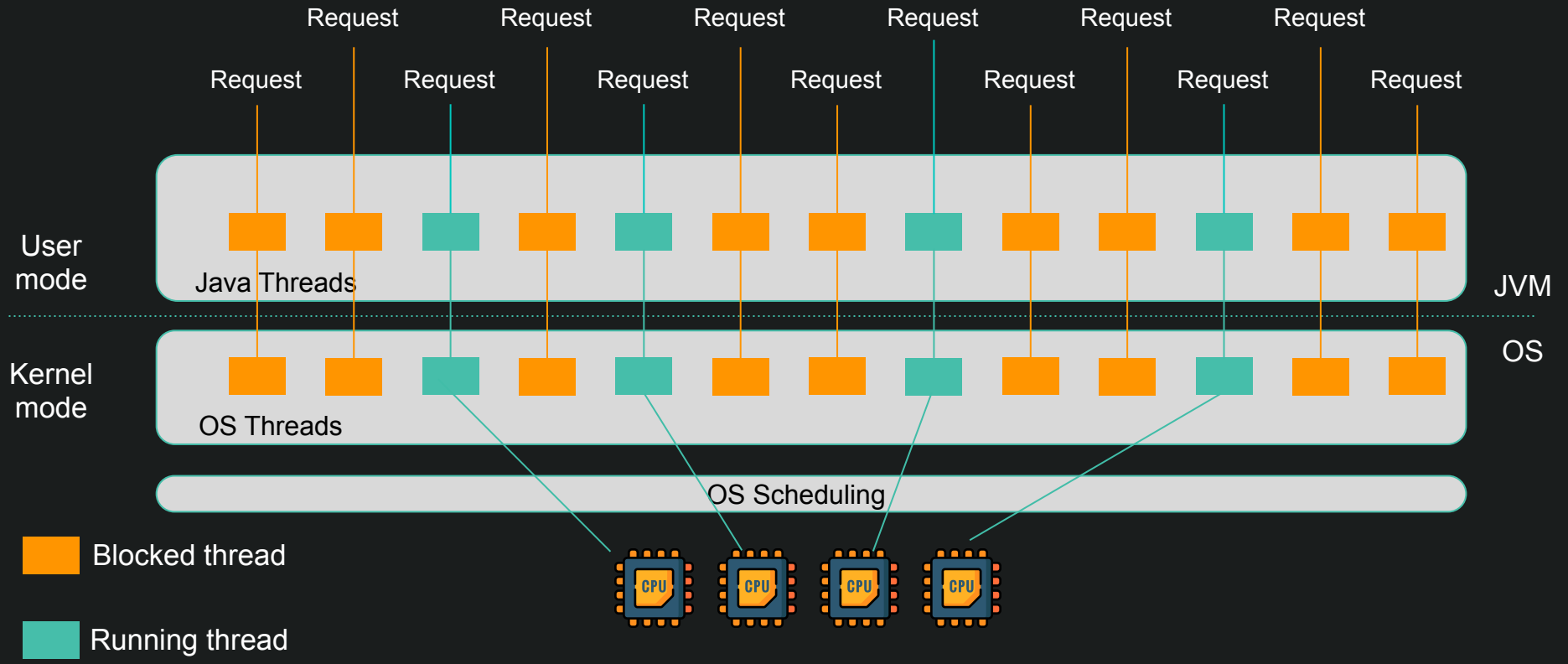
So, what's the problem with our Java apps?

How your application server handles requests?

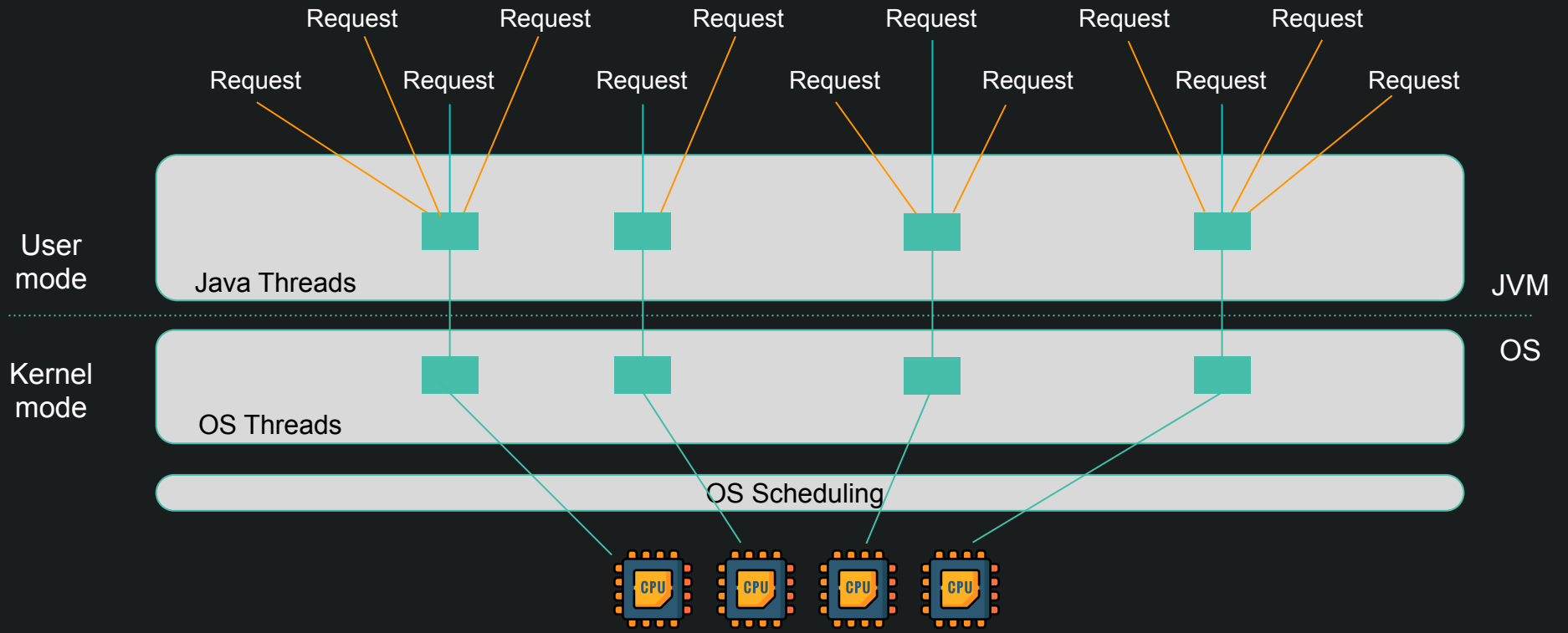


Images: Flaticon.com/Freepik

Thread-per-request



Reactive programming



Reactive vs Imperative

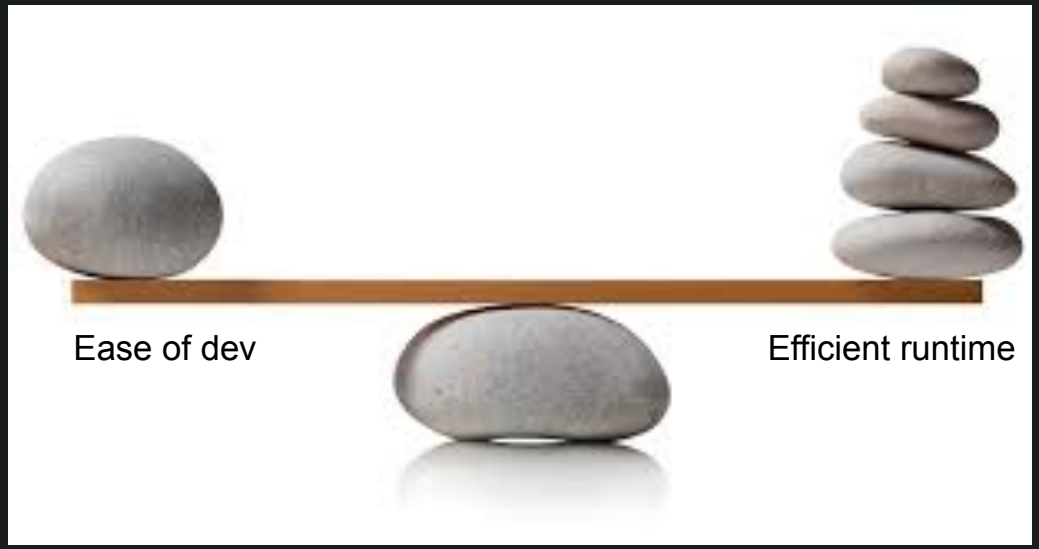
Reactive

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
public Uni<JsonObject> getStats() {
    int mb = 1024 * 1024;
    return Uni.createFrom()
            .item(Runtime.getRuntime())
            .onItem().invoke(() ->
                Log.info("Start building JVM stats the reactive way"))
            .onItem().transform(runtime ->
                Json.createObjectBuilder()
                    .add("availableProcessors", runtime.availableProcessors())
                    .add("usedMemoryMB", (runtime.totalMemory() - runtime.freeMemory()) / mb)
                    .build())
            .onItem().invoke(() ->
                Log.info("Done building JVM stats the reactive way"))
            .onFailure().invoke(e ->
                Log.error("Error building JVM stats the reactive way", e));
}
```

Imperative

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getStats() {
    int mb = 1024 * 1024;
    Runtime runtime = Runtime.getRuntime();
    JsonObject stats = null;
    Log.info("Start building JVM stats the imperative way");
    try {
        stats = Json.createObjectBuilder()
            .add("availableProcessors", runtime.availableProcessors())
            .add("usedMemoryMB", (runtime.totalMemory() - runtime.freeMemory()) / mb))
            .build();
        Log.info("Done building JVM stats the imperative way");
        return Response.ok().entity(stats).build();
    } catch (Exception e) {
        Log.error("Error building JVM stats the imperative way", e);
        return Response.serverError().build();
    }
}
```

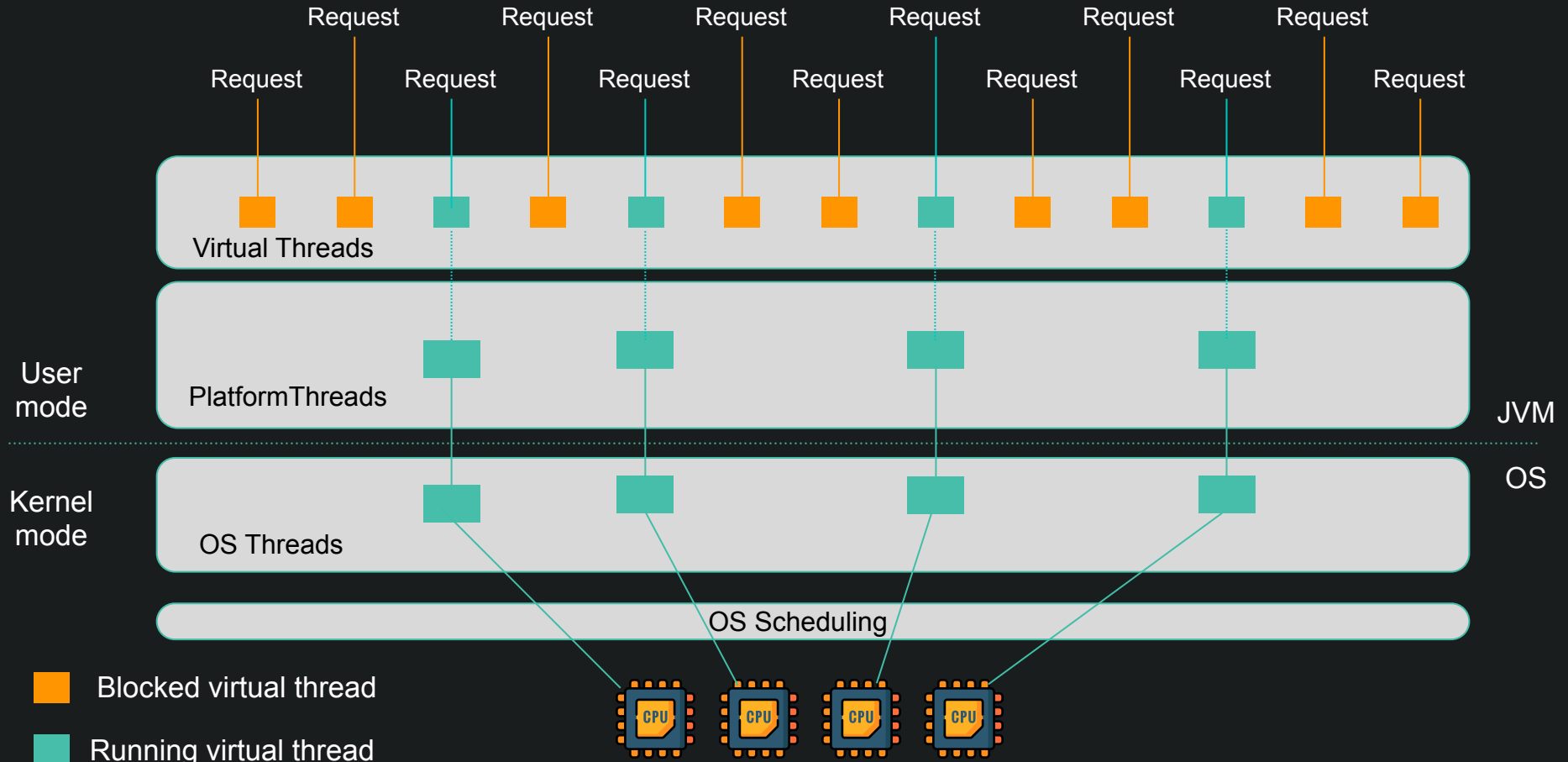
You need to choose!



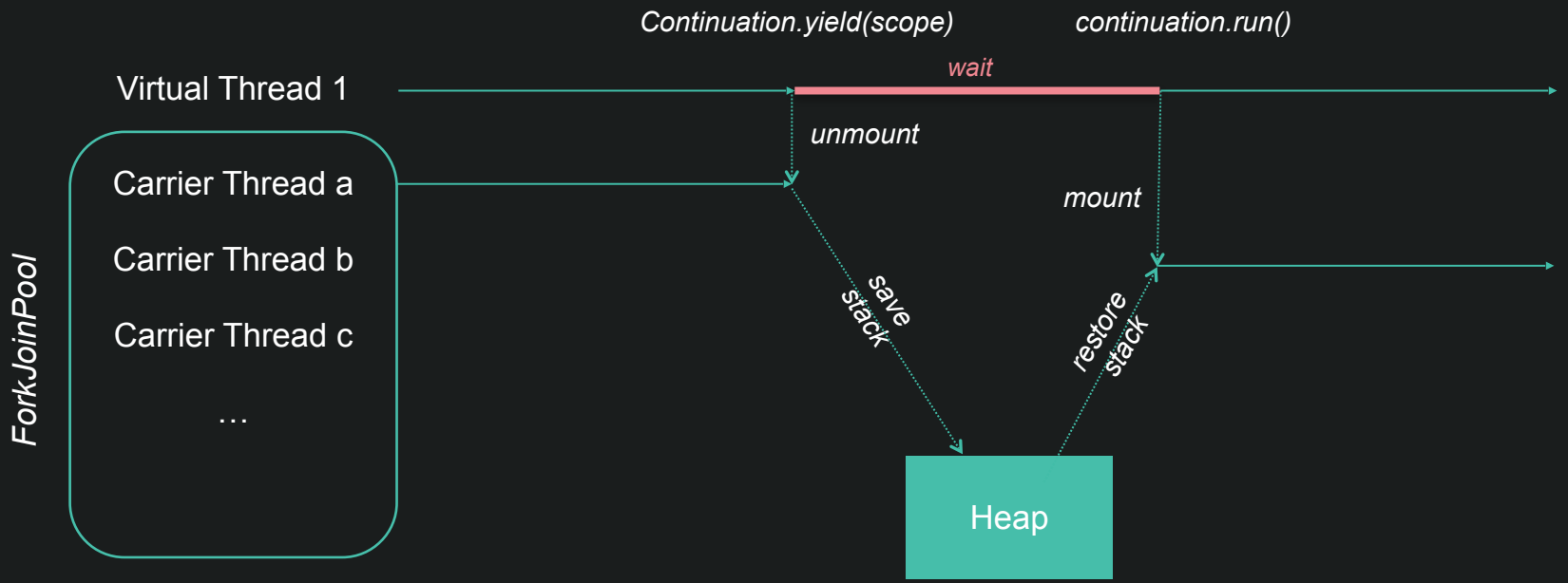
Loom is coming!

Code imperative
AND
Run reactive

Virtual thread-per-request



Inside the JVM: the magic of Continuation



Demo time!

Let’s create millions of threads!

Technical Context

Basic Java SE

Code on GitHub

Measured on MacOS (ARM M1, 8 CPU, 16 G RAM)

Java 21

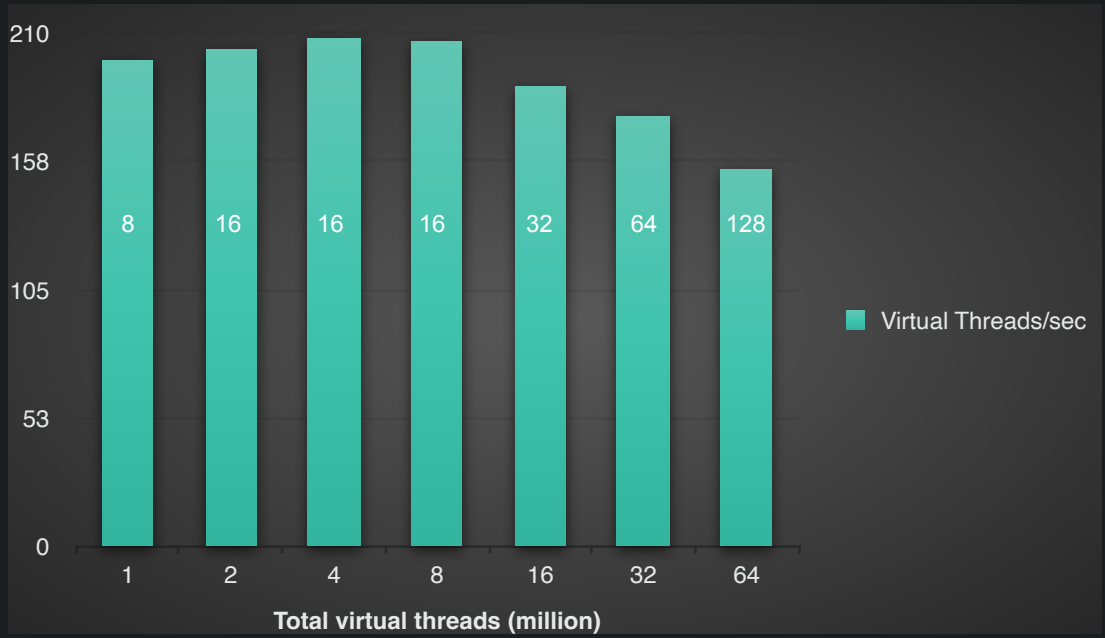
G1 GC (default)

Fixed Heap Size -Xmx=-Xms

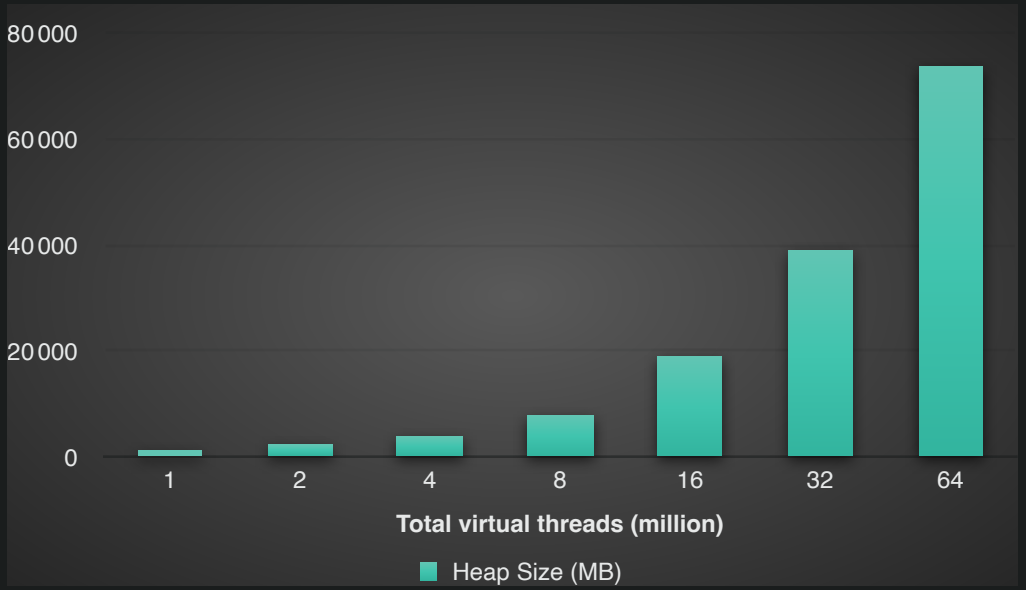
Some code

```
1  public static void main(String[] args) {
2      List<Thread> threads = new ArrayList<>(THREAD_COUNT);
3      var hold = new CountDownLatch(1);
4
5      while (threads.size() < THREAD_COUNT) {
6          CountDownLatch started = new CountDownLatch(1);
7          Thread thread = Thread.ofVirtual().start(() -> {
8              process(pThreads);
9              started.countDown();
10             try {
11                 hold.await();
12             } catch (InterruptedException ignore) {
13             }
14         });
15     }
16 }
```

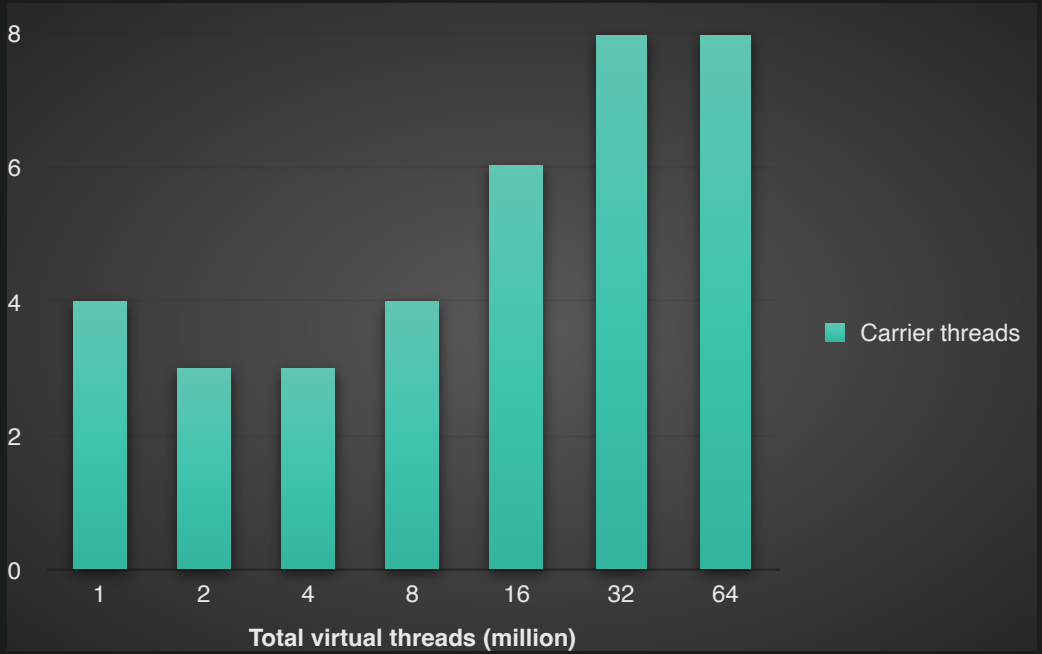
Fast to create?



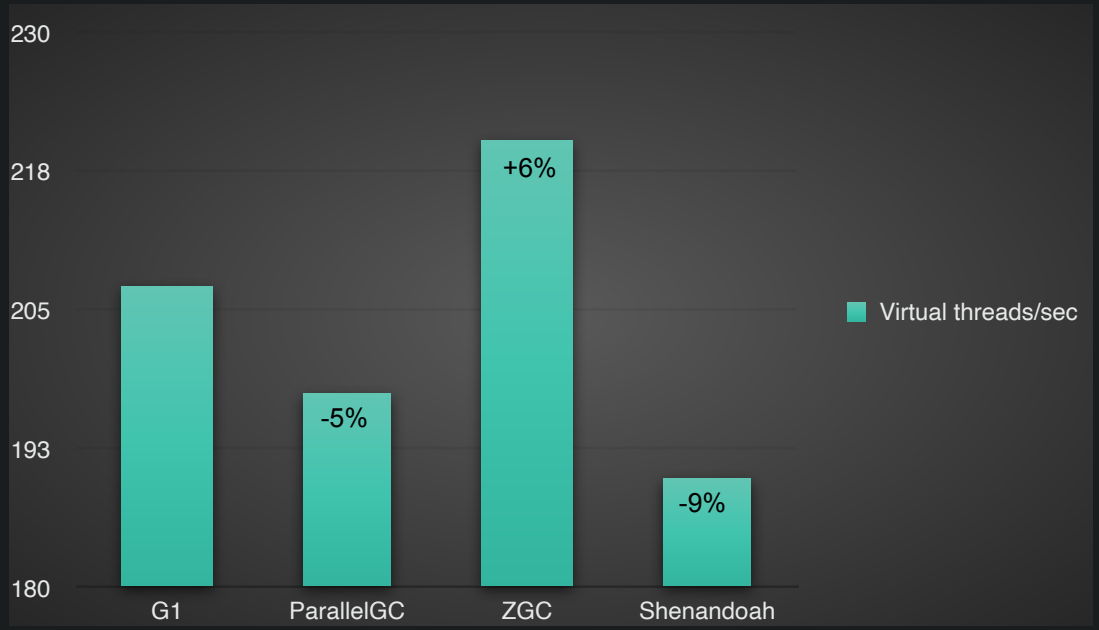
Memory footprint



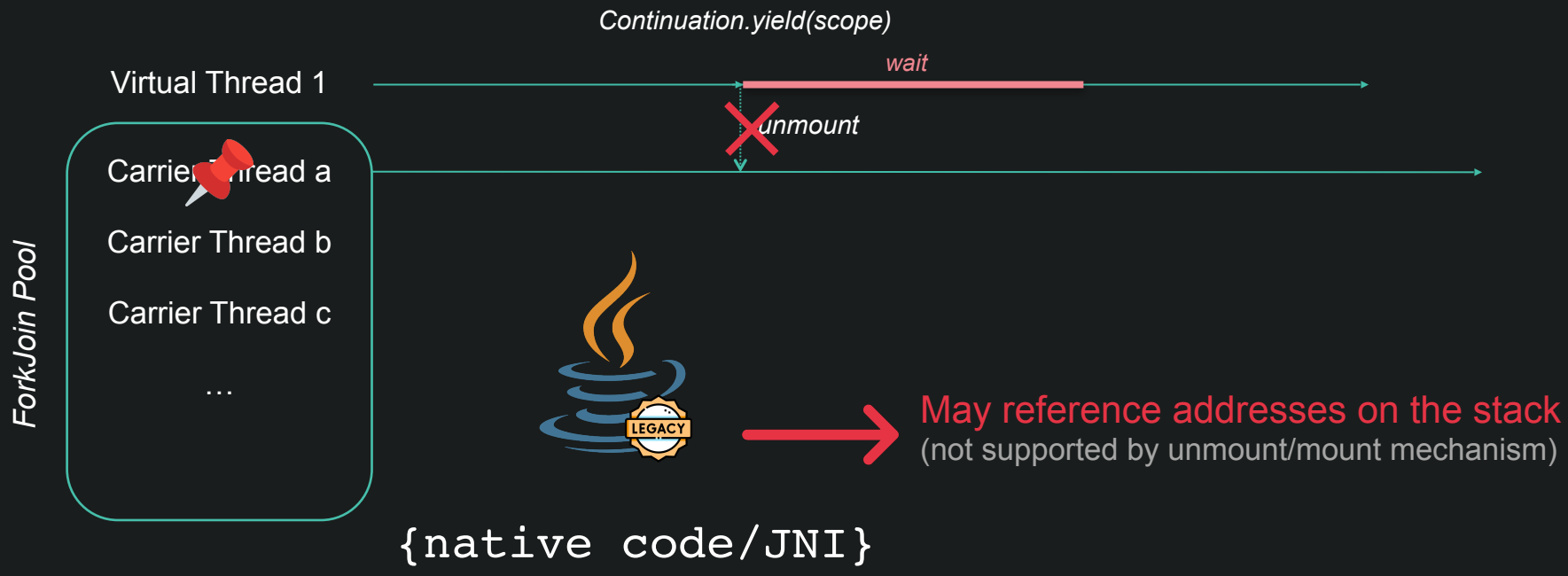
Carrier Threads



Performance per GC



Thread pinning



Observability & monitoring

Pinned Carrier Threads

-Djdk.tracePinnedThreads=short/full

```
Thread[#63,ForkJoinPool-1-worker-2,5,CarrierThreads]
    org.h2.command.Command.executeUpdate(Command.java:252) <== monitors:1
    org.h2.jdbc.JdbcPreparedStatement.executeUpdateInternal(JdbcPreparedStatement.java:209) <== monitors:1
```

Others

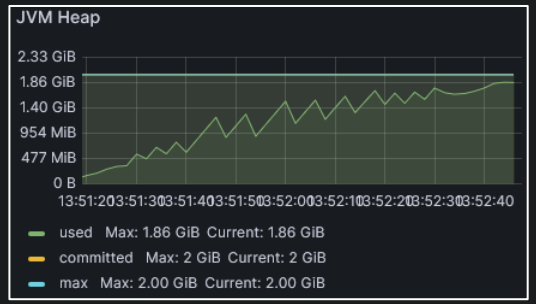
Java Flight Recorder events: start, end, pinned, submit failed
jcmd thread dumps: plain text (verbose), JSON (hierarchical view)
Warning: not yet production-ready

Memory and GC

Memory footprint

GC activity

```
-xlog:gc
```



Configuration

Heap Size -Xmx -Xms

```
Example when trying to create 16 million Virtual Threads with 4g:
[55,637s][info][gc] GC(23) Pause Full (Ergonomics) 3754M->3754M(3925M) 1869,993ms
[57,496s][info][gc] GC(24) Pause Full (Ergonomics) 3754M->3754M(3925M) 1858,765ms
[59,377s][info][gc] GC(25) Pause Full (Ergonomics) 3754M->3754M(3925M) 1880,413ms
[61,236s][info][gc] GC(26) Pause Full (Ergonomics) 3754M->3754M(3925M) 1858,437ms
```

Garbage Collector

Throughput first: SerialGC
Latency first: G1, ZGC, Shenandoah

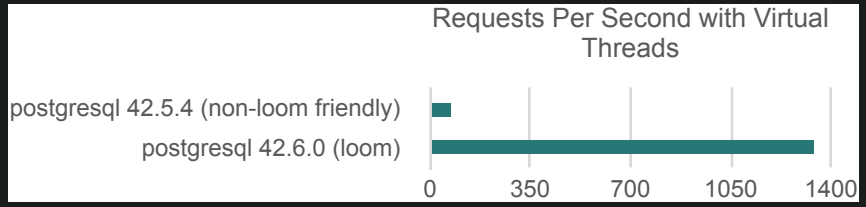
Virtual Thread Scheduler

ForkJoinPool: parallelism, maxPoolSize, minRunnable
Monitoring tool?

Make your code Loom-friendly

Avoid long synchronized blocks/methods

- Replace with ReentrantLock
- Check your dependencies



Thread pool not needed with VT

- Warning: no safety guard with Excecutors.newVirtualThreadPerTaskExecutor()
- Risk of saturation of resources used: outgoing connections
- Use Semaphore to limit the access to resources

Use Thread Locals with care

- Enables to share variables in the context of a Thread
- Design flaws: unbounded lifetime, unconstrained mutability, expensive inheritance
- Not optimal with « millions » of Virtual Threads
- In the mid-term: to be replaced by Scoped Variables

DZone: Pitfalls to avoid when switching to virtual threads

Demo time!

Virtual Thread Adoption

Open-Source from Red Hat
« Supersonic Subatomic Java »
Native Image support

IO Threads

IO Processing
Reactive programming
Can’t be blocked

Worker Threads

Platform Threads
Default

Virtual Threads
@RunOnDefaultThread

Open-Source from Oracle
«Lightweight. Fast. Crafted for Microservices »
Native Image support

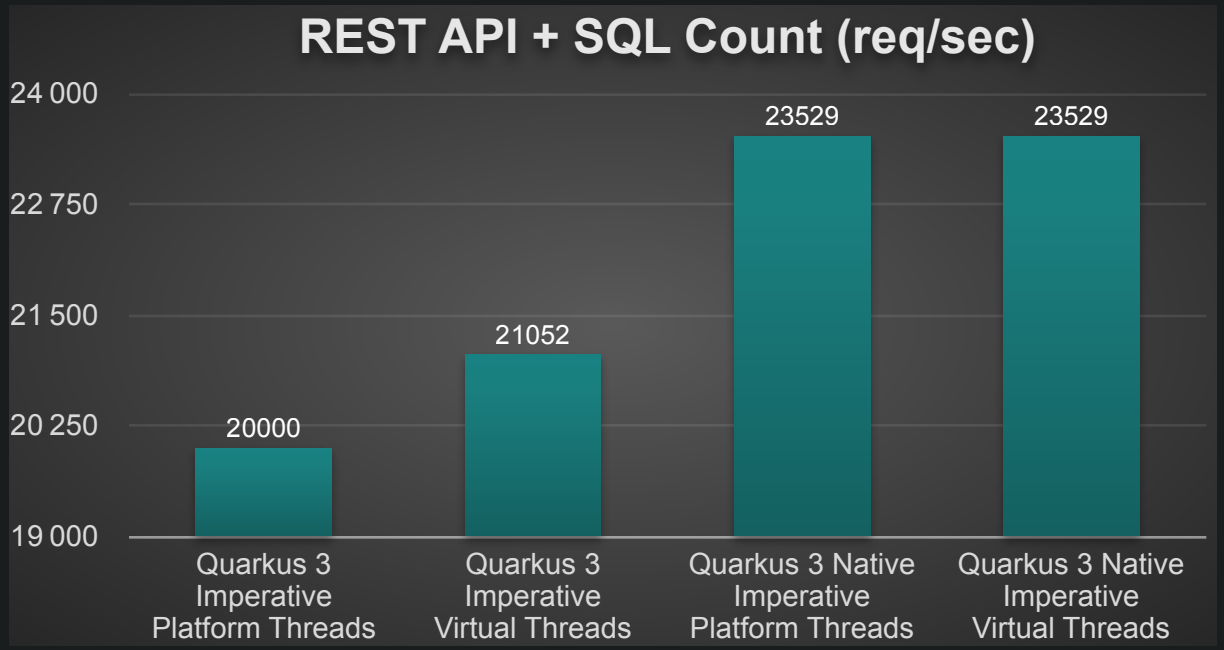
Helidon 3

Production-ready
Java 17
Netty Web Server
Default to Platform Threads
Supports Virtual Threads

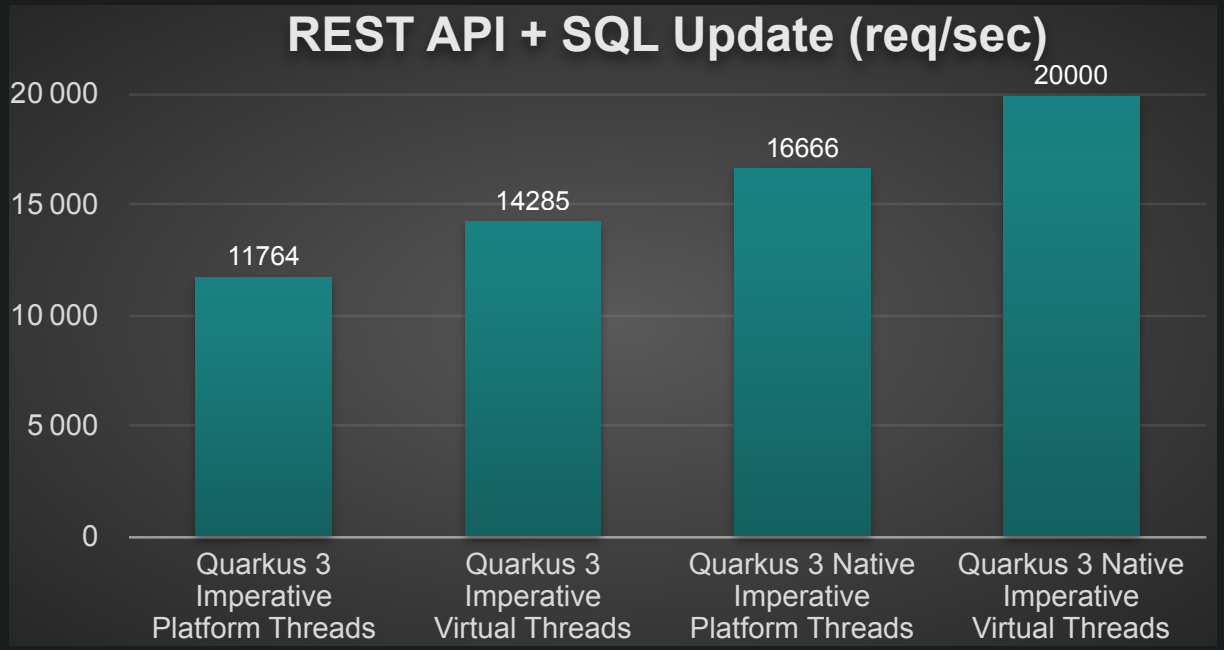
Helidon 4

In development
Java 21
Nima Web Server
Virtual Threads by Design

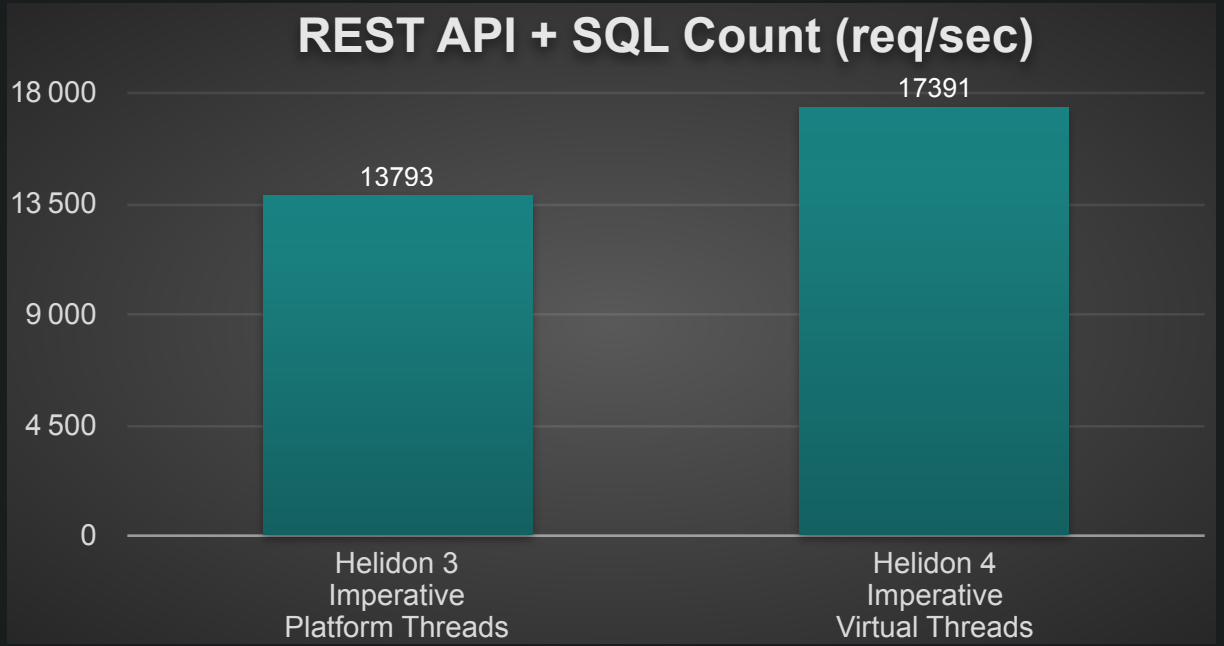
Quarkus Native Performance



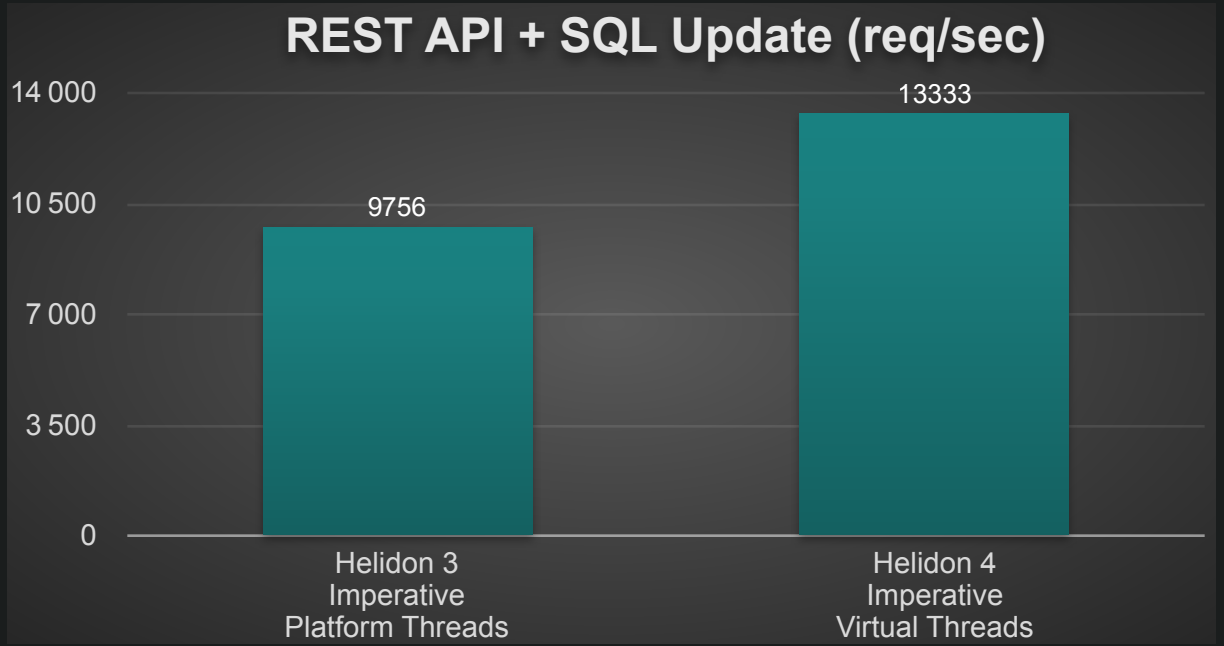
Quarkus Native Performance



Helidon Performance



Helidon Performance



Conclusion

Next steps

Short term-In development

Learn Virtual Threads

Make your code Loom-friendly

Check how your frameworks and libs adopt Virtual Threads

Bench: identify bottlenecks not visible so far!

Re-bench regularly: things are improving fast!

Mid term-Preparing production

Bench

Determine heap size

Select GC algorithm

Configure Virtual Threads scheduling

Check monitoring tool improvement

Long term

Test Structured Concurrency & Scoped Values

Do some feedbacks to the community

Replace Thread Locals by Scoped Values

Thanks for your attention!

Contact:
JF James
Worldline

WORLDLINE

Appendix

Contact:
JF James & David Pequegnot
Worldline

WORLDLINE

Structured Concurrency

- Asbtract away the use of Virtual Threads
- Enable to coordinate tasks running on Virtual Threads in the context of a “scope”
- Built-in coordination strategy: any, all
- Extensible coordination strategy
- No back pressure

Scoped Values

- Thread locals not designed to be shared by “millions” of threads
- Unclear lifecycle: not always cleaned up
- Uncontrolled mutability: can be changed at any time
- Inheritance: risk of high memory footprint
- Bound to a callable (not a Thread)