Google Cloud

Next '24

A java developer walks into a serverless bar

Proprietary

Spotify

Mohammed Aboullaite

Senior Engineer,
Spotify

Agenda


JAVA
Serverless Bar

Evolution of software architecture

1980s - Early 2000s
Monolithic architecture dominates software development.

Late 1990s - 2010s
Shift towards SOA for more modular applications.

Early 2010s
Microservices architecture gains popularity for its flexibility and scalability.

Mid 2010s
Containerization and orchestration tools like Docker and Kubernetes become critical for microservices deployment.

Late 2010s - Present
Serverless computing introduces a new level of infrastructure abstraction

“

Everything in software architecture is a tradeoff

Fundamentals of Software Architecture
by Mark Richards and Neal Ford

Architecture tradeoffs

- Performance (Execution latency, Start-up time, ...)
- Efficiency (Resource consumption, Storage, ...)
- Cost-Effectiveness
- Scalability & Resiliency
- Sustainability & Maintainability

~~12~~ 15 factor apps

One Codebase, One Application	Design, Build, Release, and Run	Configuration, Credentials, and Code
API First	Logs	Disposability
Dependency Management	Backing Services	Environment Parity
Administrative Processes	Port Binding	Stateless Processes
Concurrency	Telemetry	Authentication and Authorization

https://developer.ibm.com/articles/15-factor-applications/

What Serverless means

Operational Model

- No Infra Management
- Managed Security
- Pay only for usage

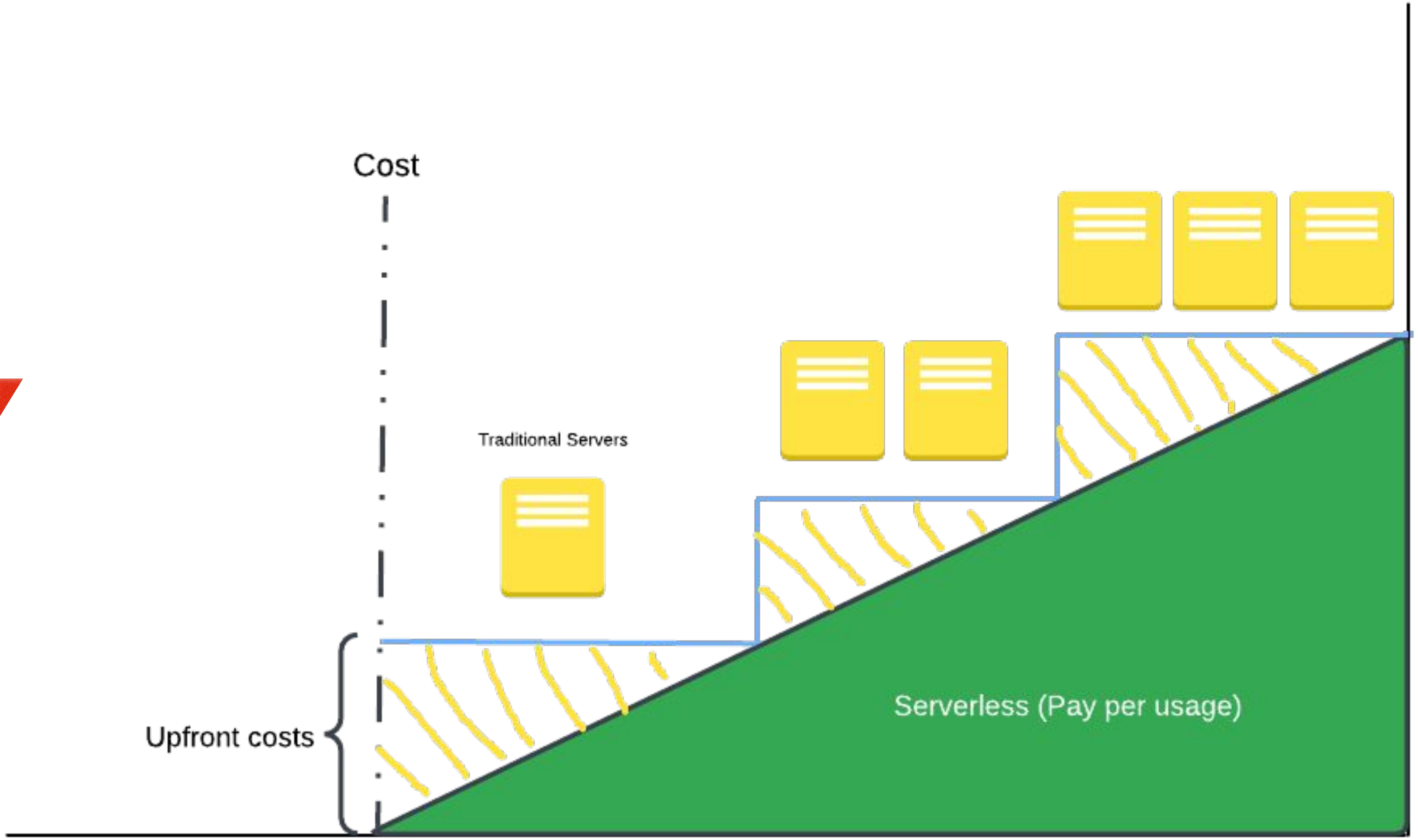
Programming Model

- Service-based
- Event-driven
- Stateless

The Serverless Promise

- Simplicity (Managed runtime environments)
- Cost Efficiency (Pay-Per-Usage, No idle costs, ...)
- Speed (Maintenance-Free Operations)
- Flexibility (Auto-scaling, dynamic adaptation, ...)
- Simplicity (Focusing on product)

Cost savings with serverless



https://www.mongodb.com/blog/post/understanding-costs-serverless-architecture-save-money

Google Cloud Serverless platform

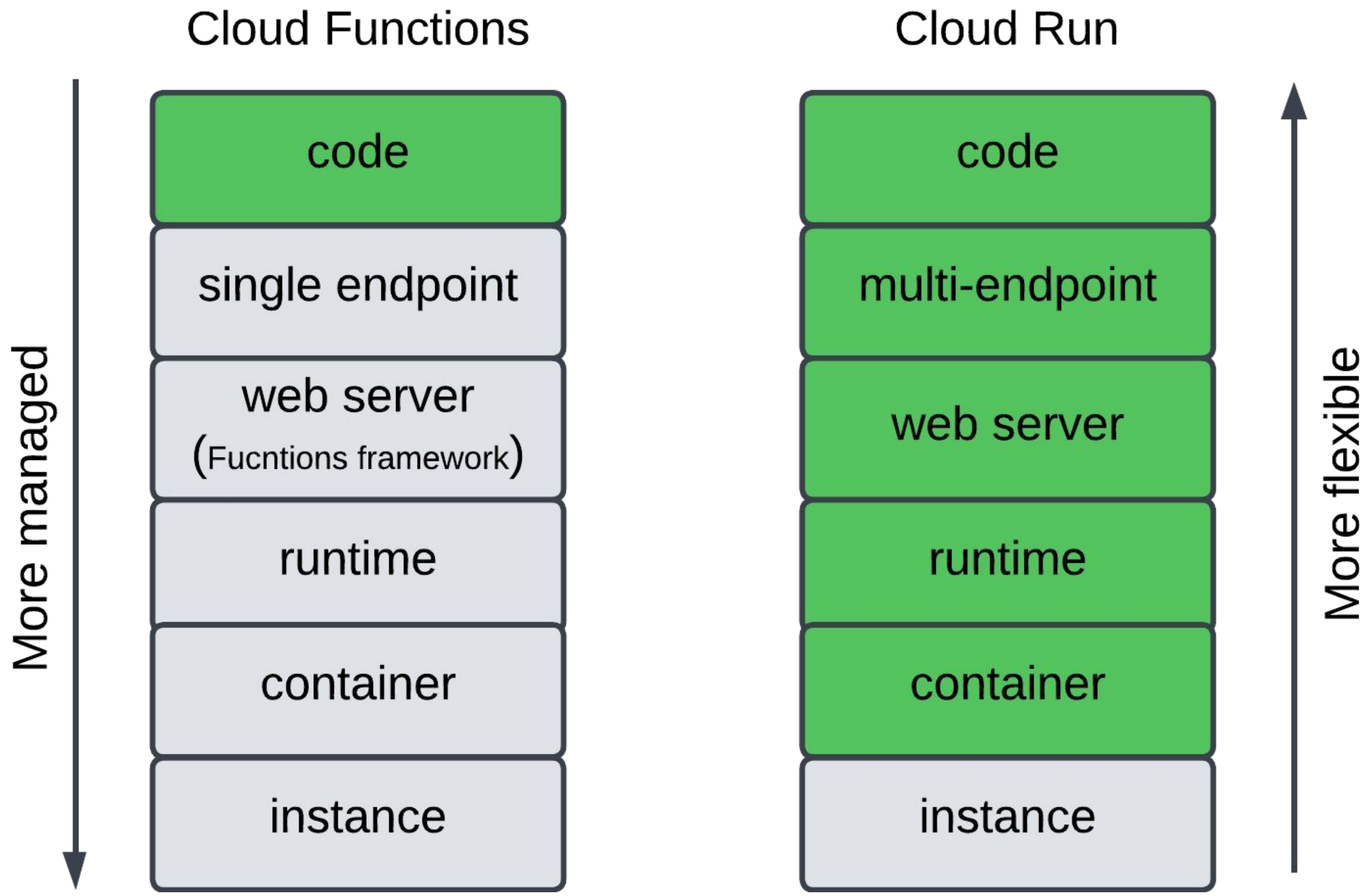
Cloud functions

Focus on executing individual functions in a serverless environment.

Cloud Run

Centered around containerization, allowing you to run entire containers

Which one to pick



https://cloud.google.com/blog/products/serverless/cloud-run-vs-cloud-functions-for-serverless?hl=en

Serverless Compute

Functions

Cloud run

Serverless database

Alloydb

Bigtable

Serverless data

pub/sub

storage

Java in a serverless world

The good part

- Robust ecosystem
- Performance
- Mature Tooling and Development Support
- Continuous Evolution
- Security

The tricky part

- Cold Start
- Resource Constraints

Does startup time matters?

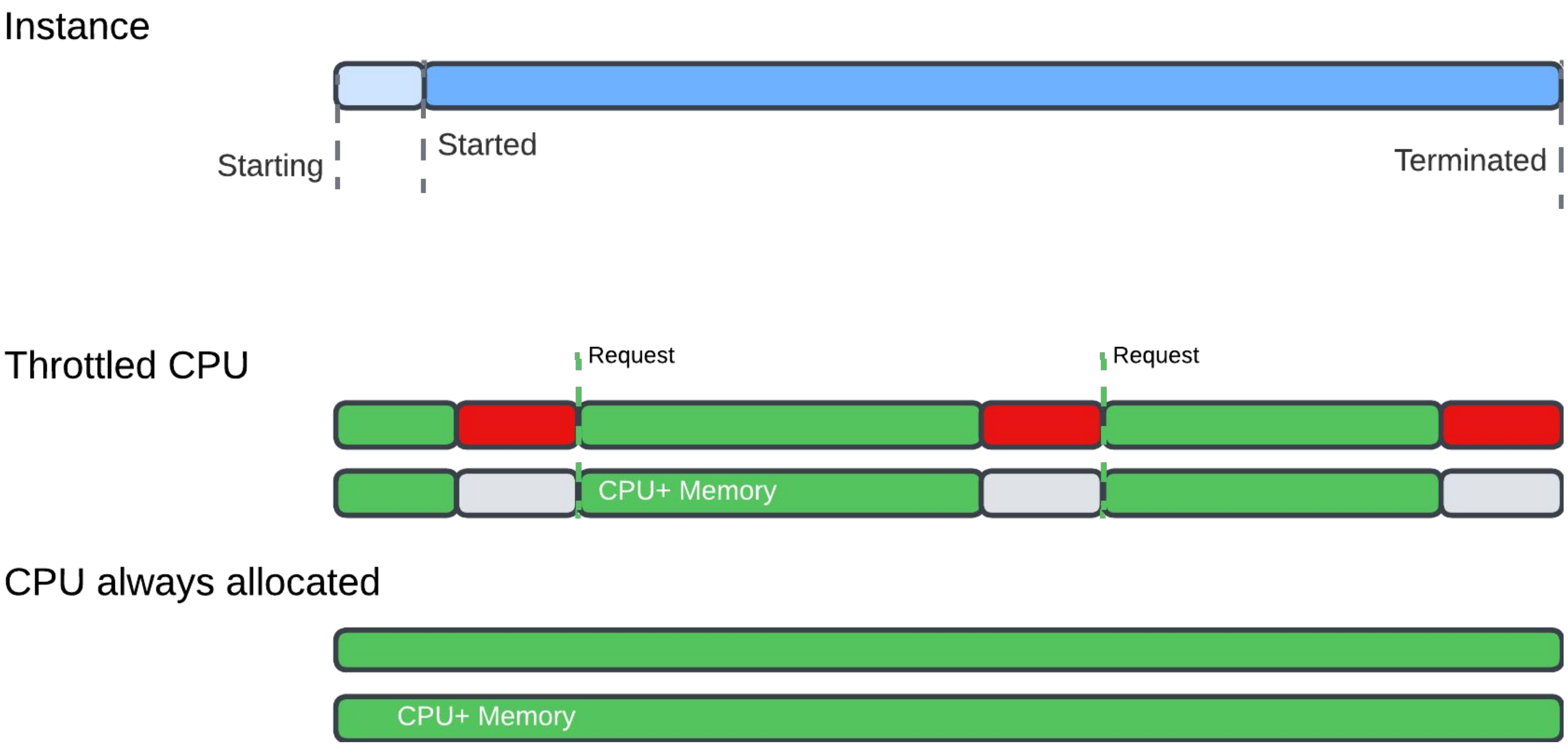
A faster startup time improves auto-scaling's responsiveness and aids in quickly launching new instances. Additionally, it enhances system resilience against sudden increases in user activity, effectively preventing system slowdowns or failures.

Mitigations

- CPU allocation & Background execution
- Request Concurrency per instance
- Container instance autoscaling
- Health checks
- CPU Boost - Google specific optimization
- Latest Java versions
- Class Data Sharing (CDS)
- JVM optimizations
- Coordinated Restore at Checkpoint (CRaC)
- Ahead-of-Time (AOT) compilation with GraalVM

Cloud Runtime Optimizations

CPU allocation



https://cloud.google.com/run/docs/configuring/cpu-allocation

CPU allocation: Traffic patterns considerations

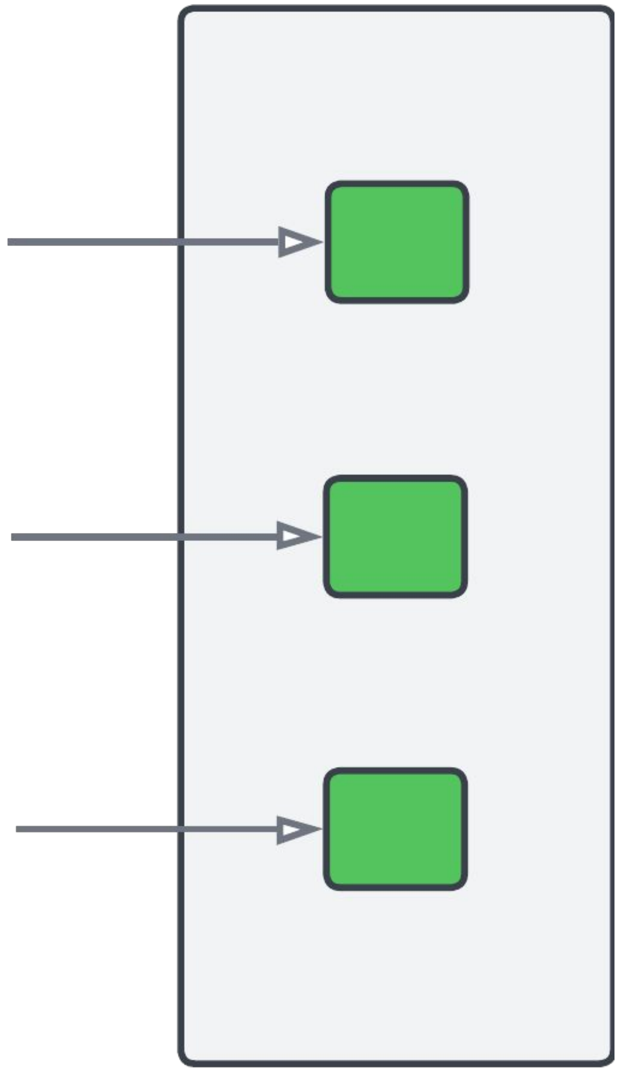
- **CPU only allocated during request processing** is recommended for applications experiencing intermittent or unpredictable traffic patterns.

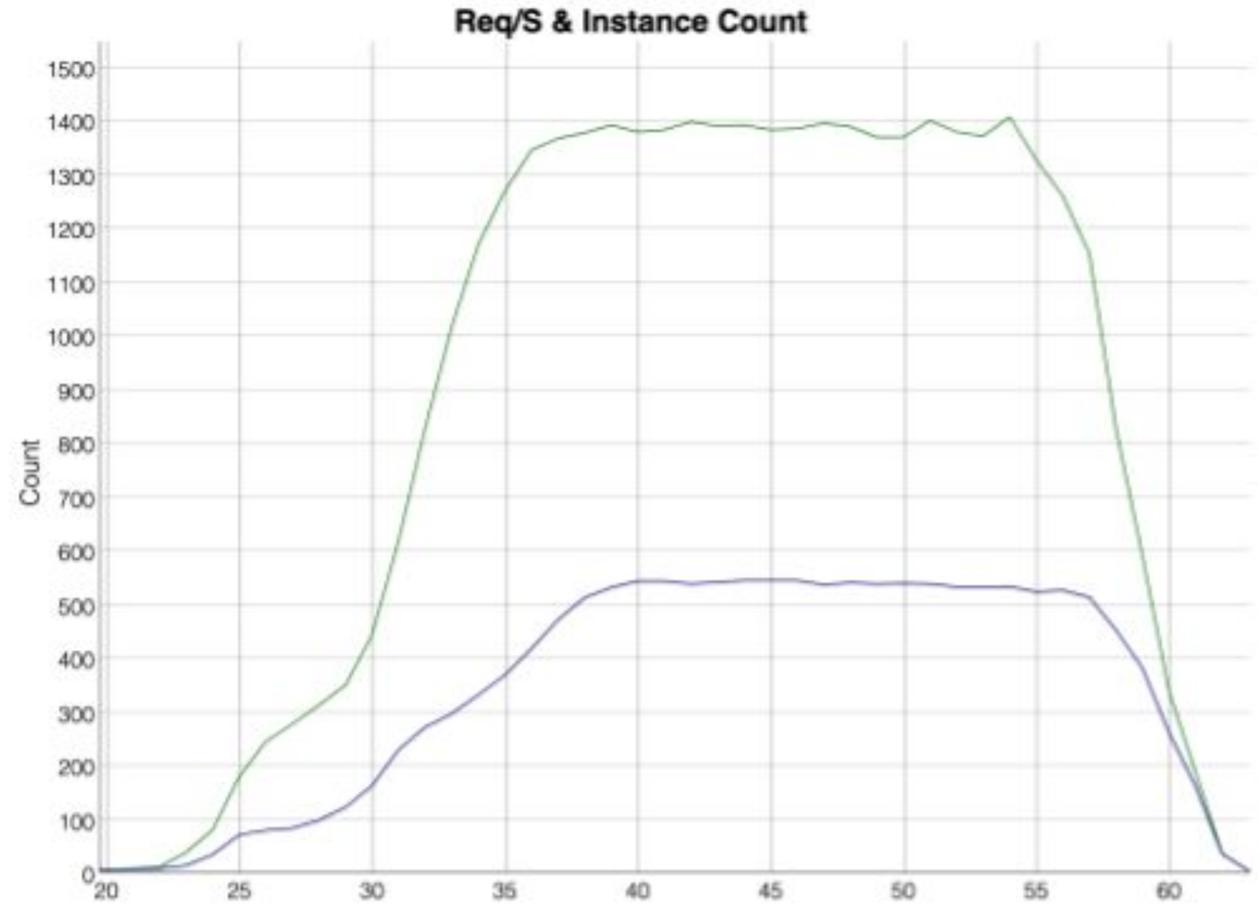
- **CPU always allocated** is beneficial for services with consistent traffic flows, slowly varying:

  - Distributed tracing or metrics senders apps, periodically sending data
  - Java threads, Kotlin coroutines
  - App frameworks relying on built-in scheduling/timing
  - Pull subscribers for messaging systems
  - JDBC Connection Pool management.

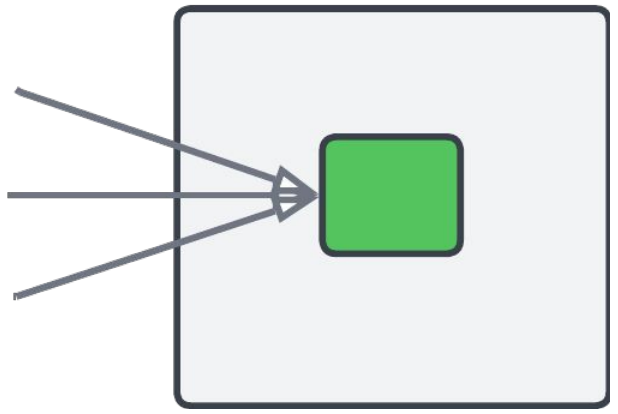
Concurrency in Serverless

Concurrency = 1

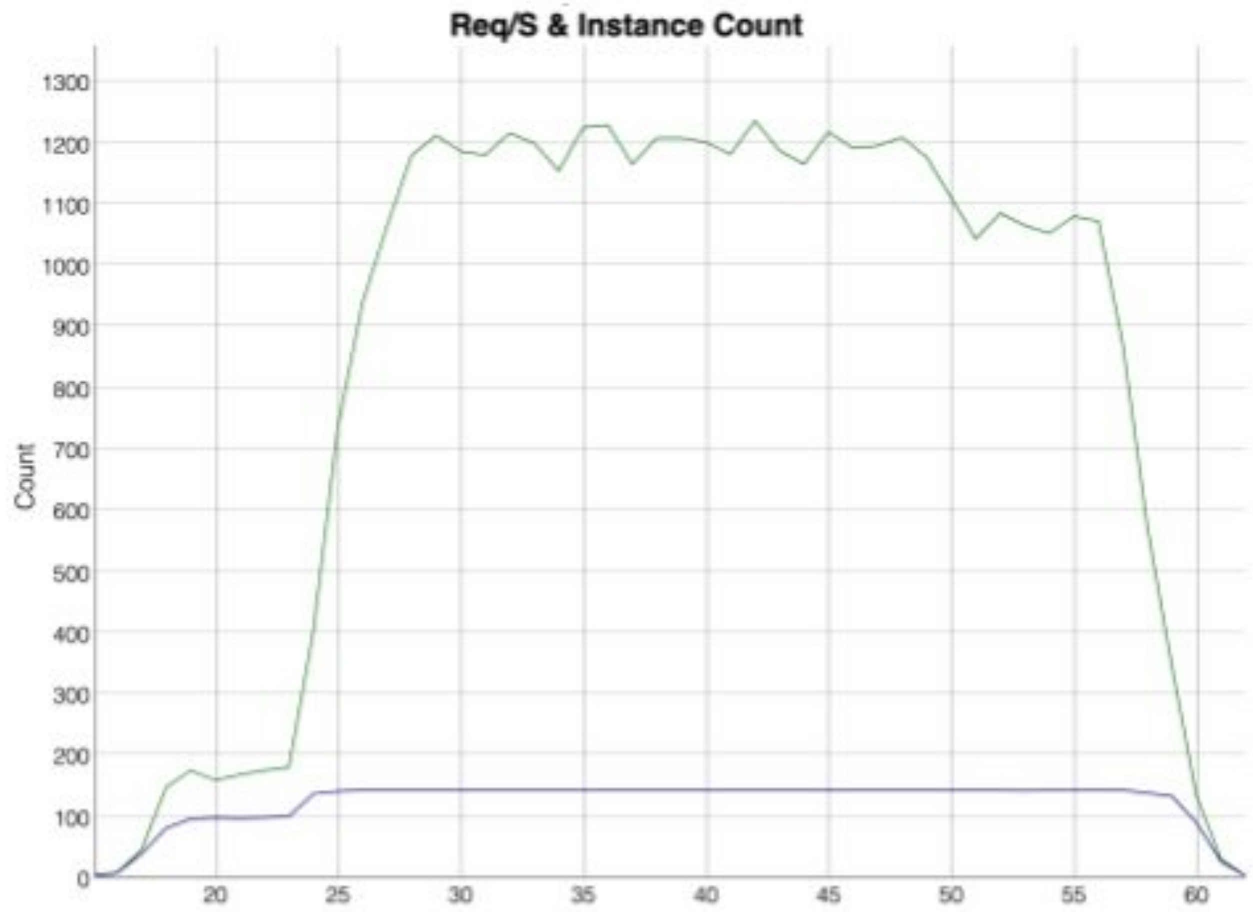




Concurrency = 80

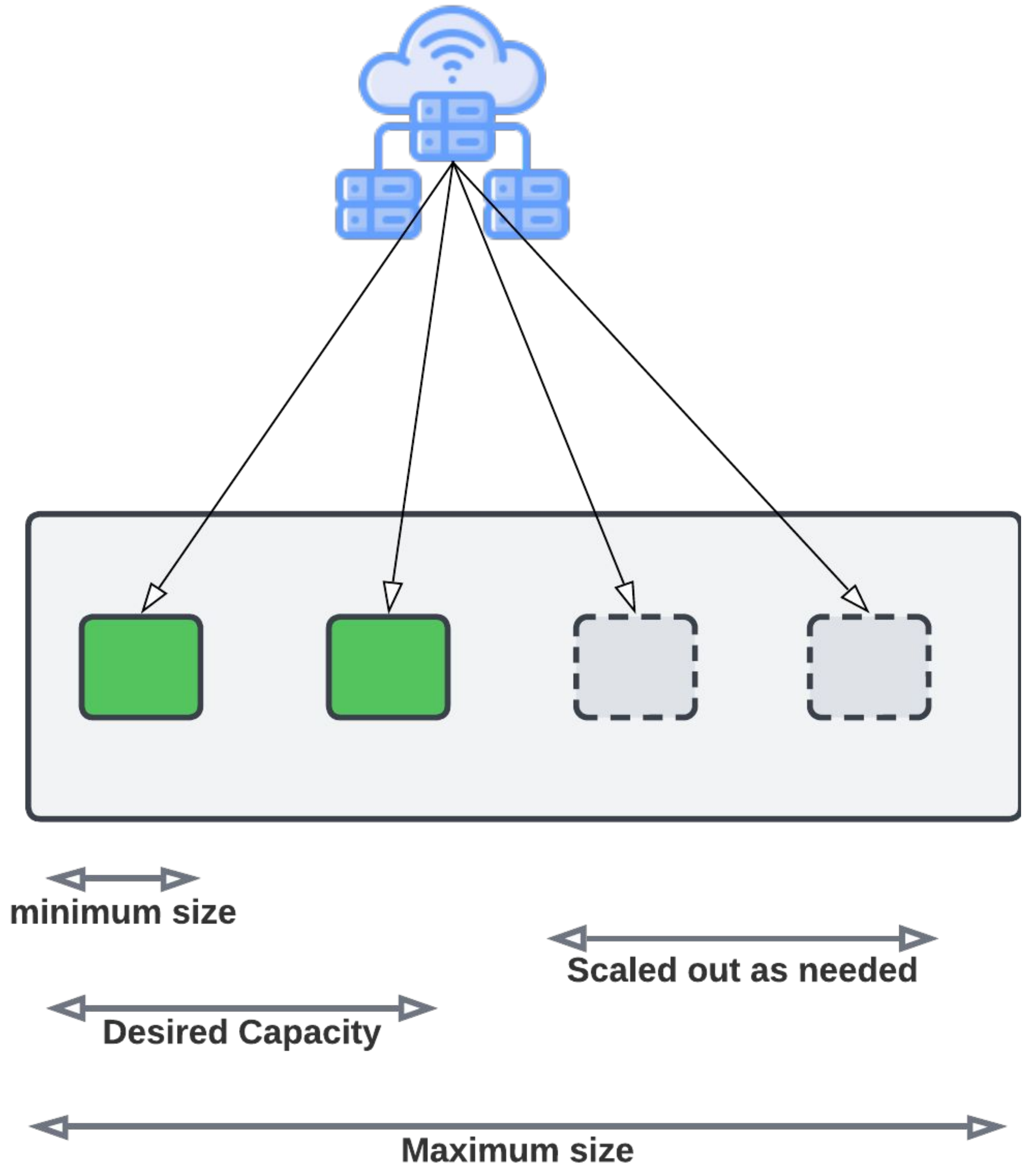


Proprietary



https://cloud.google.com/run/docs/about-concurrency

Autoscaling

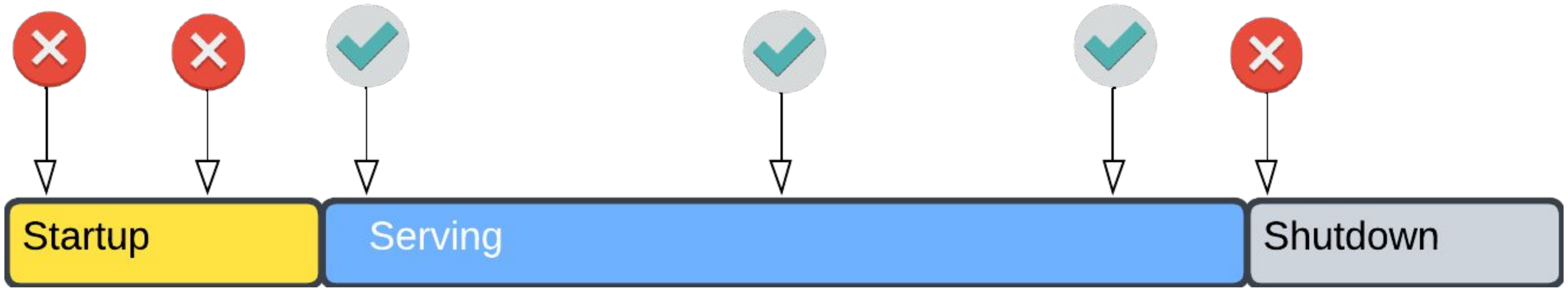


Autoscaling factors:

- CPU Utilization: Aims for 60% CPU usage over 1-minute intervals.
- Request Concurrency: Assesses against max concurrency in the last minute.
- Instance Caps: Respects max and min instance settings.

https://cloud.google.com/run/docs/about-instance-autoscaling

Health checks



Startup probe

Help ascertain when a container is ready to receive traffic. Particularly beneficial for slow-starting containers, ensuring they are not prematurely terminated before becoming operational.

Liveness probe

Determine when to restart a container, aiding in catching deadlocks where a service is running but unable to progress, thereby enhancing service availability in the presence of bugs.

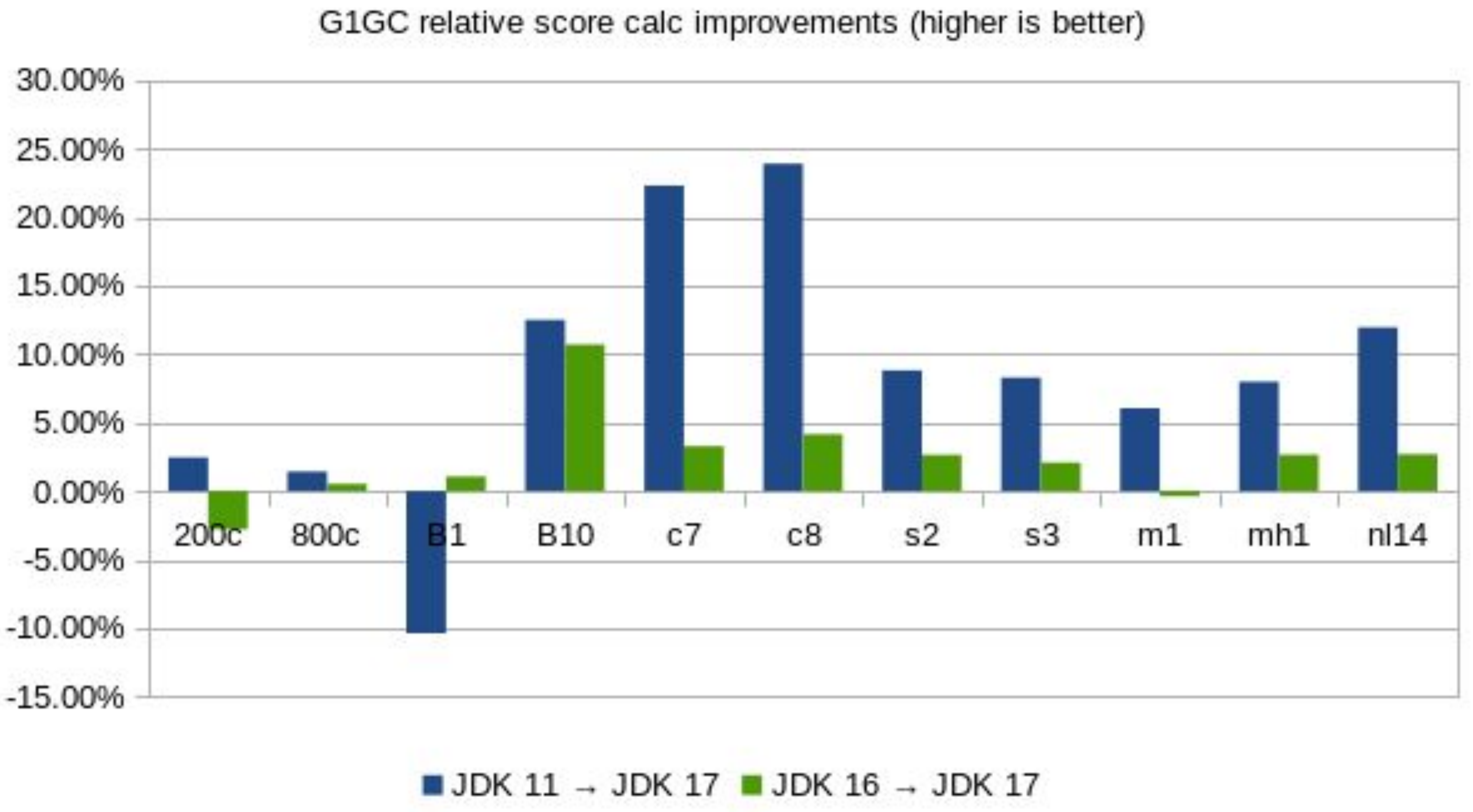
https://cloud.google.com/run/docs/configuring/healthchecks

Startup CPU Boost

https://cloud.google.com/run/docs/configuring/services/cpu

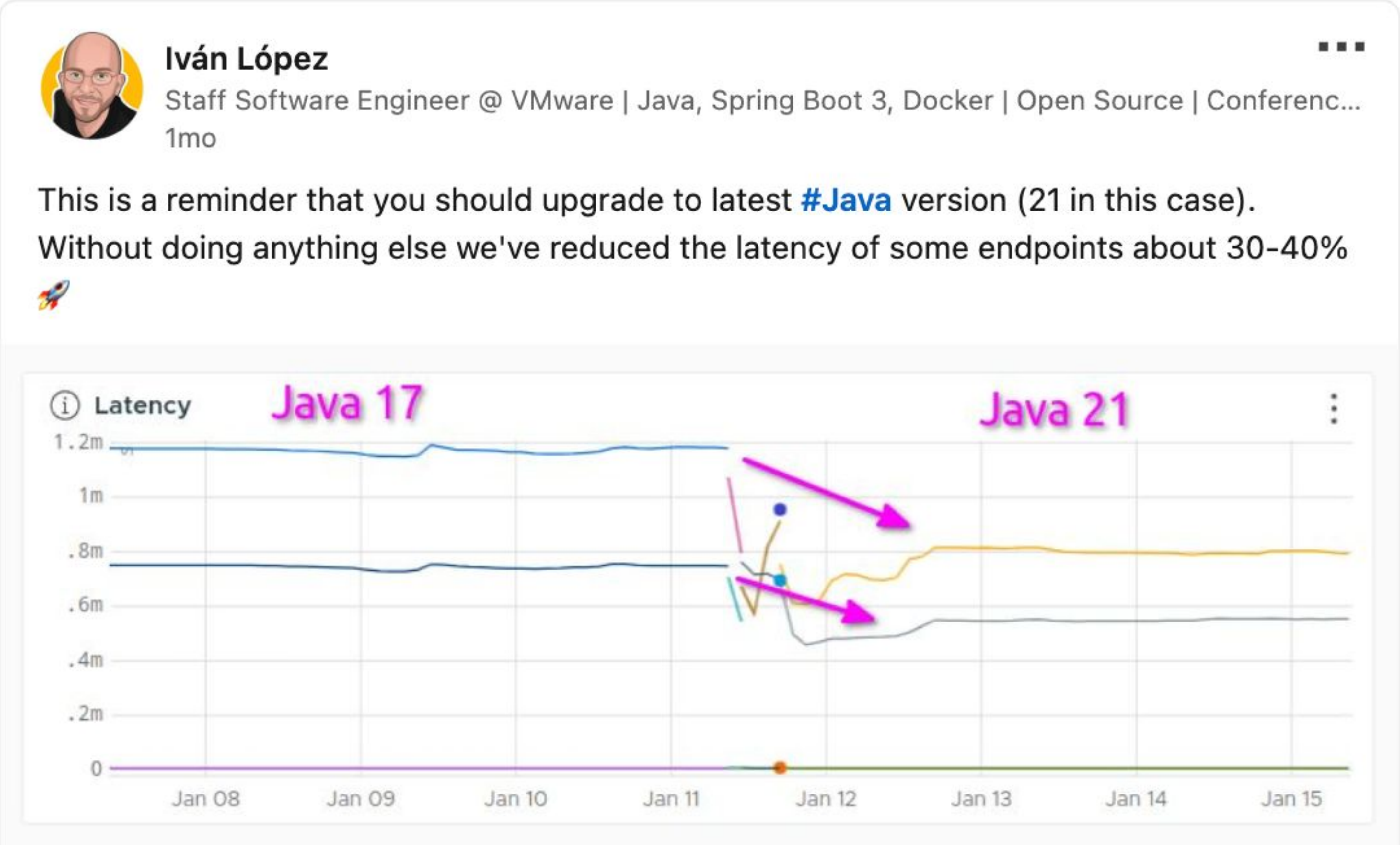
Java-specific optimizations

Latest is better



https://www.optaplanner.org/blog/2021/09/15/HowMuchFasterIsJava17.html

Latest is better



Know your JVM ergonomics

Hardware Resources

JVM assumes it has full access to the host's physical resources (CPU, memory), which it uses to make decisions on resource allocation.

Consistent Performance

JVM presumes the underlying hardware will provide consistent performance, which is used to optimize JIT compilation and GC behavior.

Exclusivity

JVM often assumes it's the only significant process running on a machine, thus it optimizes as if it has all resources to itself.

JVM ergonomics: GC

- Default GC in
  - HotSpot JVM / OpenJDK (Java 11 or later)
    - Defaults to SerialGC or G1GC if no GC is specified.

  - Java 8
    - Uses SerialGC or ParallelGC by default.

- Default GC when ressources are
  - Up to 1791 MB of memory
    - Default GC: SerialGC

  - 2 or more processors & <=1792 MB of memory
    - Default GC: G1GC

JVM ergonomics: Memory

- Maximum Heap Size:
  - Memory Available: Up to 256 MB
    - Default Heap: 50% of memory
  - Memory Available: 256 MB to 512 MB
    - Default Heap: ~127MB
  - Memory Available: More than 512 MB
    - Default Heap: 25% of memory
- Initial Heap Size
  - Set to 1/64th of available memory.

Tune your JVM

Core JVM Container Support

-XX:+UseContainerSupport

Auto-adjusts to container limits (Enabled by default).

Memory Management

-XX:InitialRAMPercentage,
-XX:MaxRAMPercentage,
-XX:MinRAMPercentage

Configure heap size.

CPU Core Adjustment

-XX:ActiveProcessorCount

Specifies the number of CPU cores for the JVM to use.

Garbage Collection Tuning

-XX:ParallelGCThreads,
-XX:ConcGCThreads

Limits GC parallel & GC concurrent phase threads to match container cores.

Debugging and Monitoring

-XX:+PrintFlagsFinal

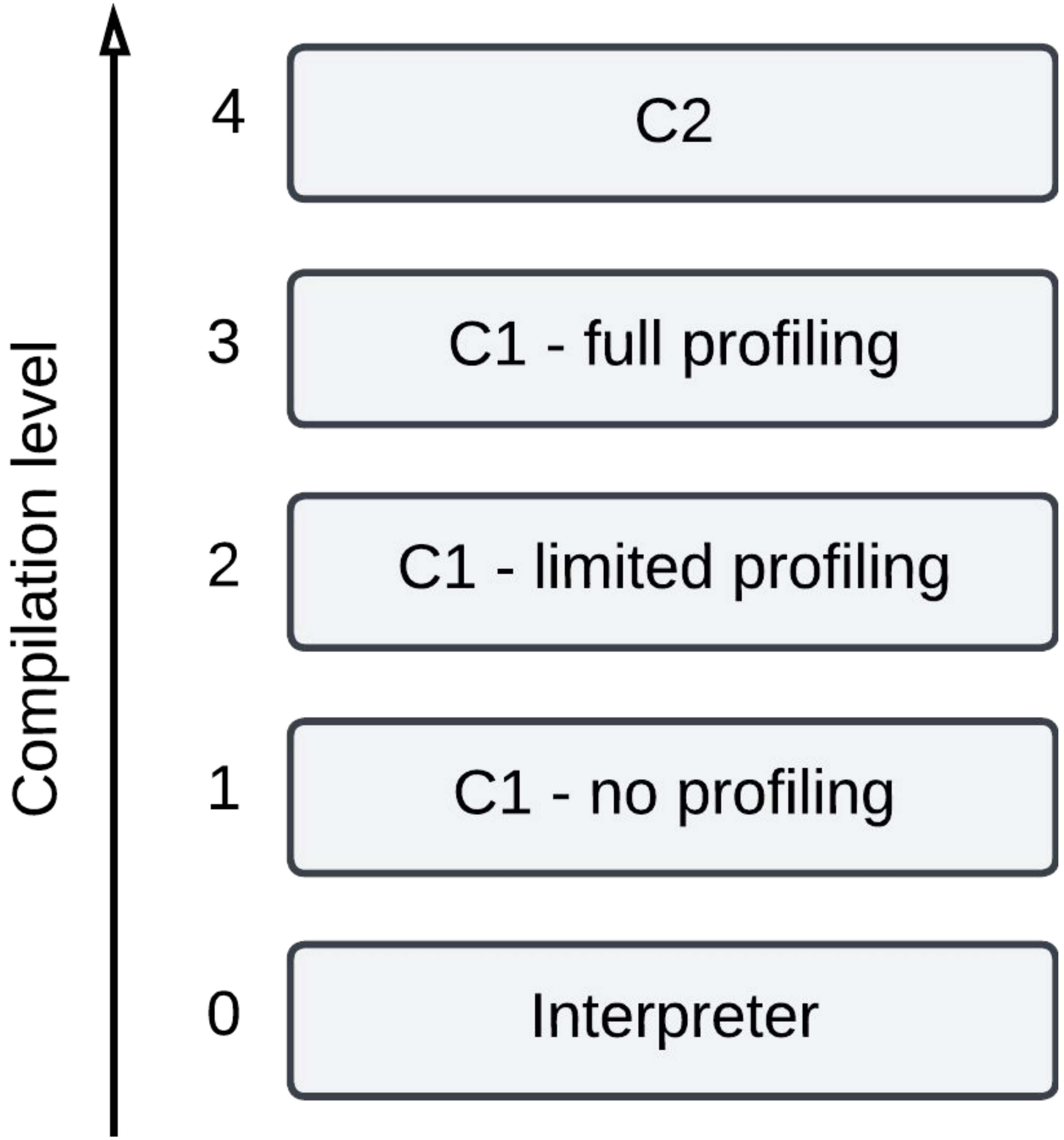
Outputs final JVM flag values for verification.

Pick the right GC

Factors	SerialGC	ParallelGC	G1GC	ZGC	ShenandoahGC
Number of cores	1	2	2	2	2
Multi-threaded	No	Yes	Yes	Yes	Yes
Java heap size	<4 GBytes	<4 GBytes	>4 GBytes	>4 GBytes	>4 GBytes
Pause	Yes	Yes	Yes	Yes (<1 ms)	Yes (<10 ms)
Overhead	Minimal	Minimal	Moderate	Moderate	Moderate
Tail-latency Effect	High	High	High	Low	Moderate
JDK version	All	All	JDK 8+	JDK 17+	JDK 11+
Best for	Single-core small heaps	Multi-core small heaps or batch workloads with any heap size	Responsive in medium to large heaps (request-response/DB interactions)	Responsive in medium to large heaps (request-response/DB interactions)	Responsive in medium to large heaps (request-response/DB interactions)

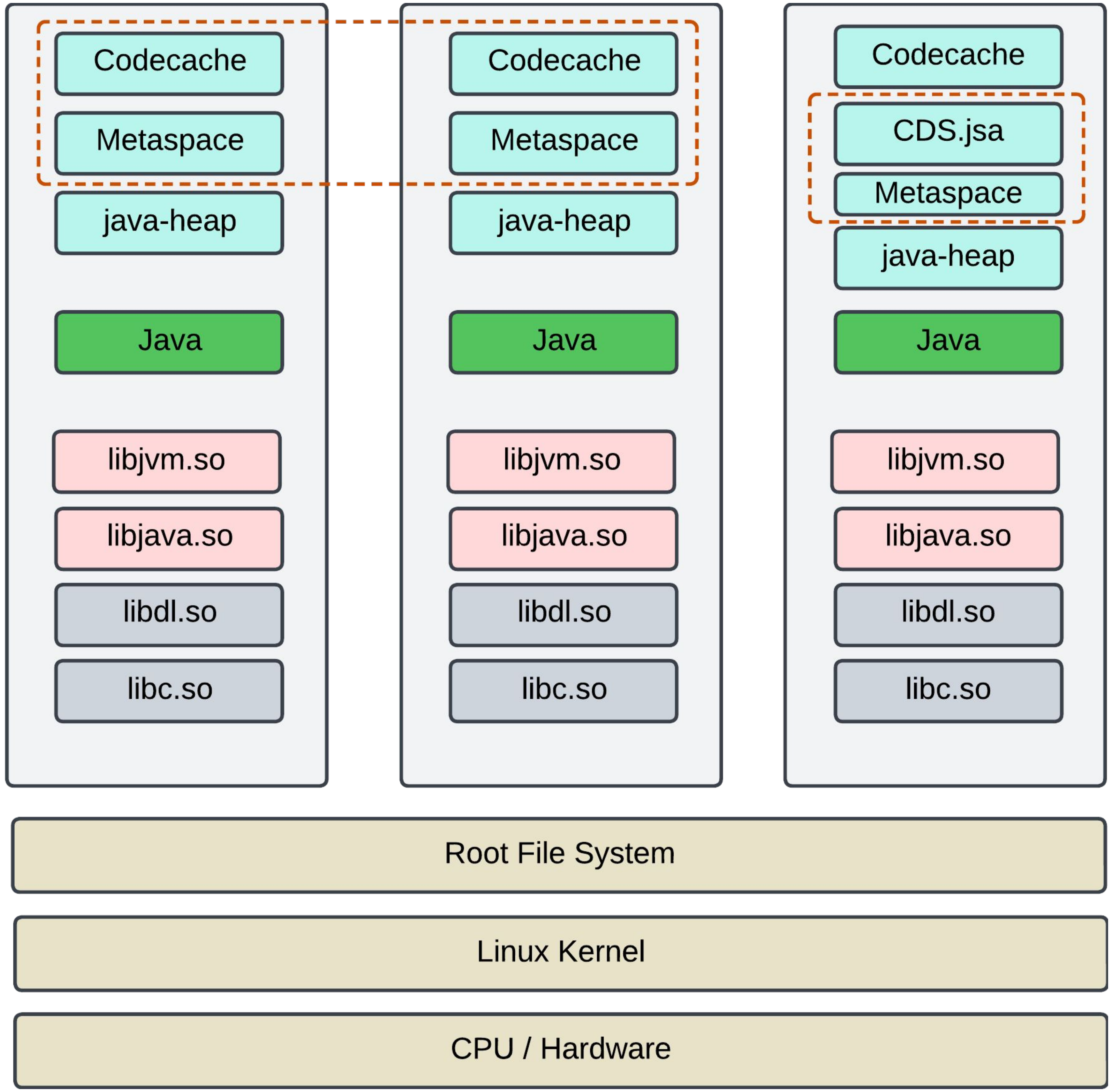
https://learn.microsoft.com/en-us/azure/developer/java/containers/overview

Tiered Compilation



- Setting **-XX:TieredStopAtLevel=1** instructs the JVM to use only the C1 only and disable C2
- By stopping at the first tier, the JVM avoids the overhead of further optimization, which is unnecessary for short-running applications.
- It will slow down the JIT later at the expense of the saved startup time

Class Data Sharing (CDS)



Class Data Sharing

How It Works

- **Archive Creation:** Generate a CDS archive file containing class data.
- **Runtime Usage:** Use -XX:SharedArchiveFile to point to the CDS file for faster class loading.

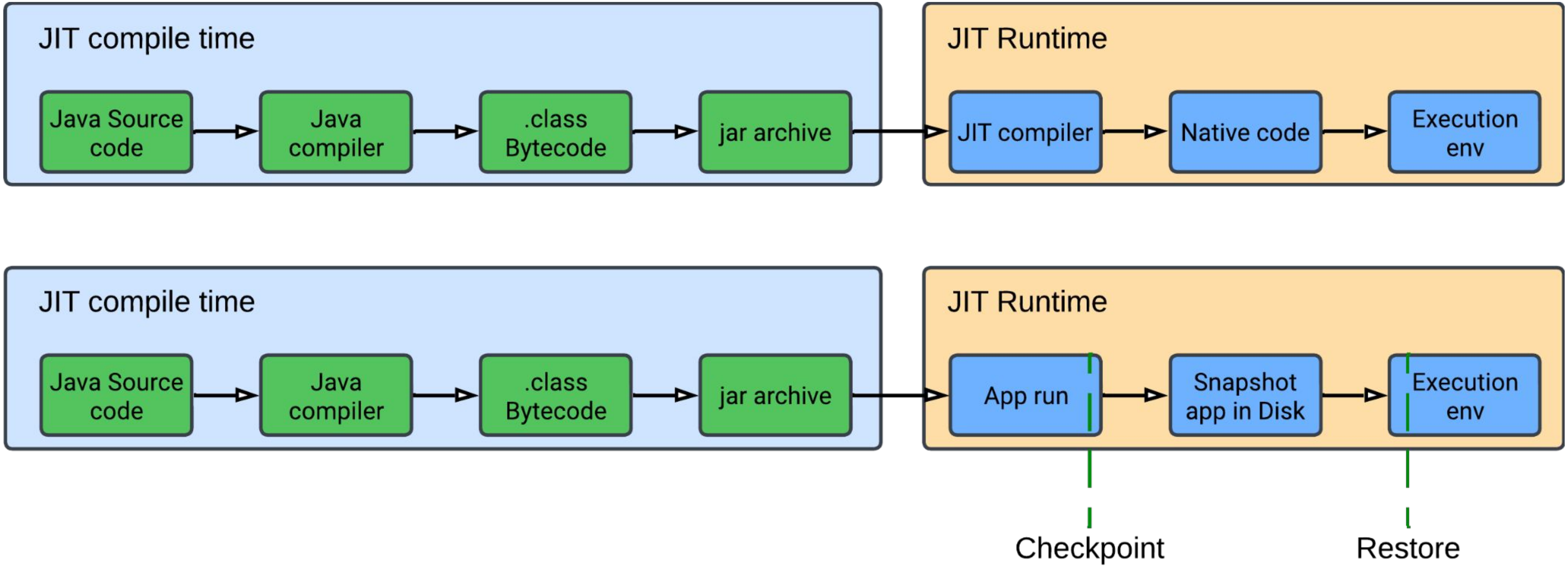
Archive Types

- **Static Dumps:** Created with -Xshare:dump and -XX:SharedClassListFile.
- **Dynamic Dumps:** Generated with -XX:ArchiveClassesAtExit (since JDK 13), simplifies the process by not requiring a class list.

Considerations

- **Consistency:** The JDK used for creating and running the archive must be identical.
- **Compatibility:** Static dumps work without the default JDK CDS archive; dynamic dumps do not.

Coordinated Restore at Checkpoint (CRaC)



CRaC: The good part

fast startup time

Project CRaC can significantly reduce the time it takes for an application to start up.

Peak performance from first request

CRaC allows applications to operate at peak efficiency from the very first request, particularly when checkpointing a fully warmed-up image.

Easy developer onboarding

Same developer experience, jvm based.

CRaC: tradeoffs

Checkpointing Requirement

Need to checkpoint/store the Java application state upfront, which might add complexity to the deployment process.

Performance Variability

Peak performance is dependent on when the checkpoint is taken within the application's lifecycle, requiring careful timing.

Resource Management

Necessitates closing and reopening files, connection pools, and sockets upon restore, which may introduce latency.

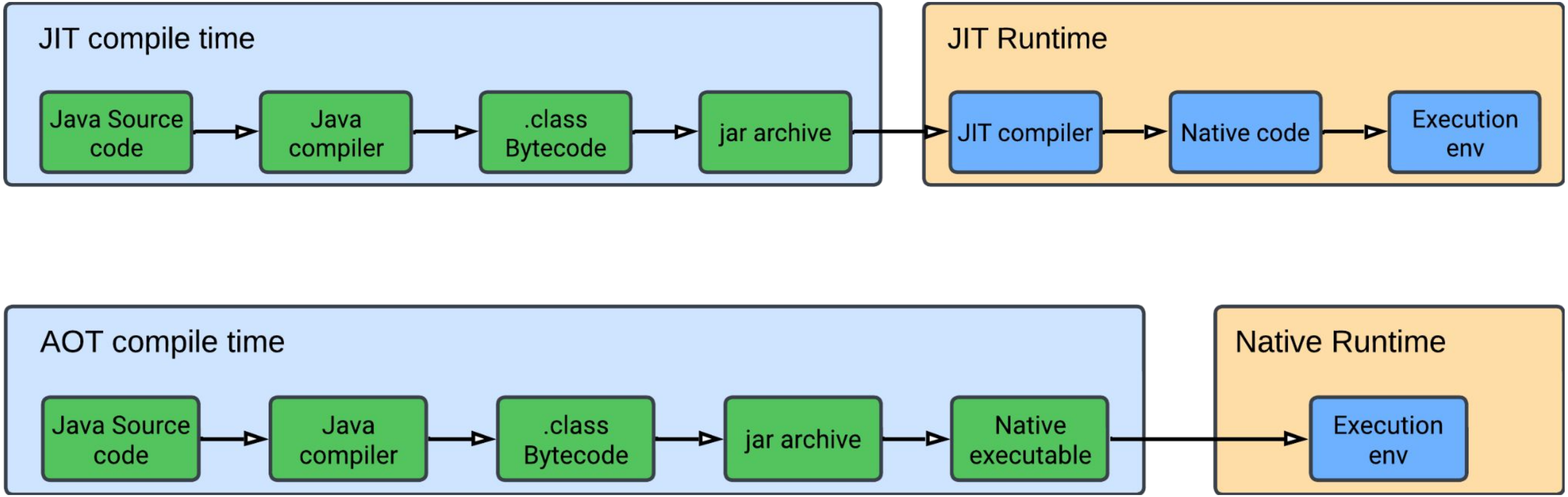
OS Dependency

Limited to Linux runtime environments due to dependency on the CRIU (Checkpoint/Restore In Userspace) technology.

Data Security Concerns

Sensitive data may be at risk of being leaked in snapshots if not properly secured or handled.

Ahead of time compilation with Graal VM



AOT: The good part

Fast Startup Time

Native images compiled with GraalVM initialize instantly, enabling rapid application startup

Peak Performance from First Request

GraalVM optimizes code at build time, providing immediate high performance at runtime without the need for JIT warm-up.

Lower CPU and Memory Usage

GraalVM reduces runtime CPU and memory overhead on startup

Smaller Attack Surface

Smaller binaries with fewer dependencies minimize the attack surface, enhancing application security.

AOT: tradeoffs

Slow compile time

AOT compilation with GraalVM can be time-consuming as it involves thorough analysis and optimization of the codebase.

Closed-World Assumptions with AOT

AOT requires all code paths to be known at compile time, limiting dynamic features typically used in Java applications.

Additional Metadata for 3rd Party Libraries

Some libraries may need extra metadata to work with GraalVM's native images, adding to the complexity of development and build processes.

Future sneak-peek: Project Leyden

- An innovative OpenJDK project aiming to improve Java application's startup/warmup time and reduce footprint.
- Focuses on selective computation shifting and constraining, ensuring 'meaning preservation'.
- Currently undergoing early-stage experiments by the Java Platform Group.
- Promising 'early' optimization observed with a 15% startup improvement, achieved by combining Class Data Sharing (CDS) with Spring AOT.

https://openjdk.org/projects/leyden/notes/03-toward-condensers

https://github.com/openjdk/leyden/tree/premain/test/hotspot/jtreg/premain/javac_new_workflow

Demo

Security

Security considerations

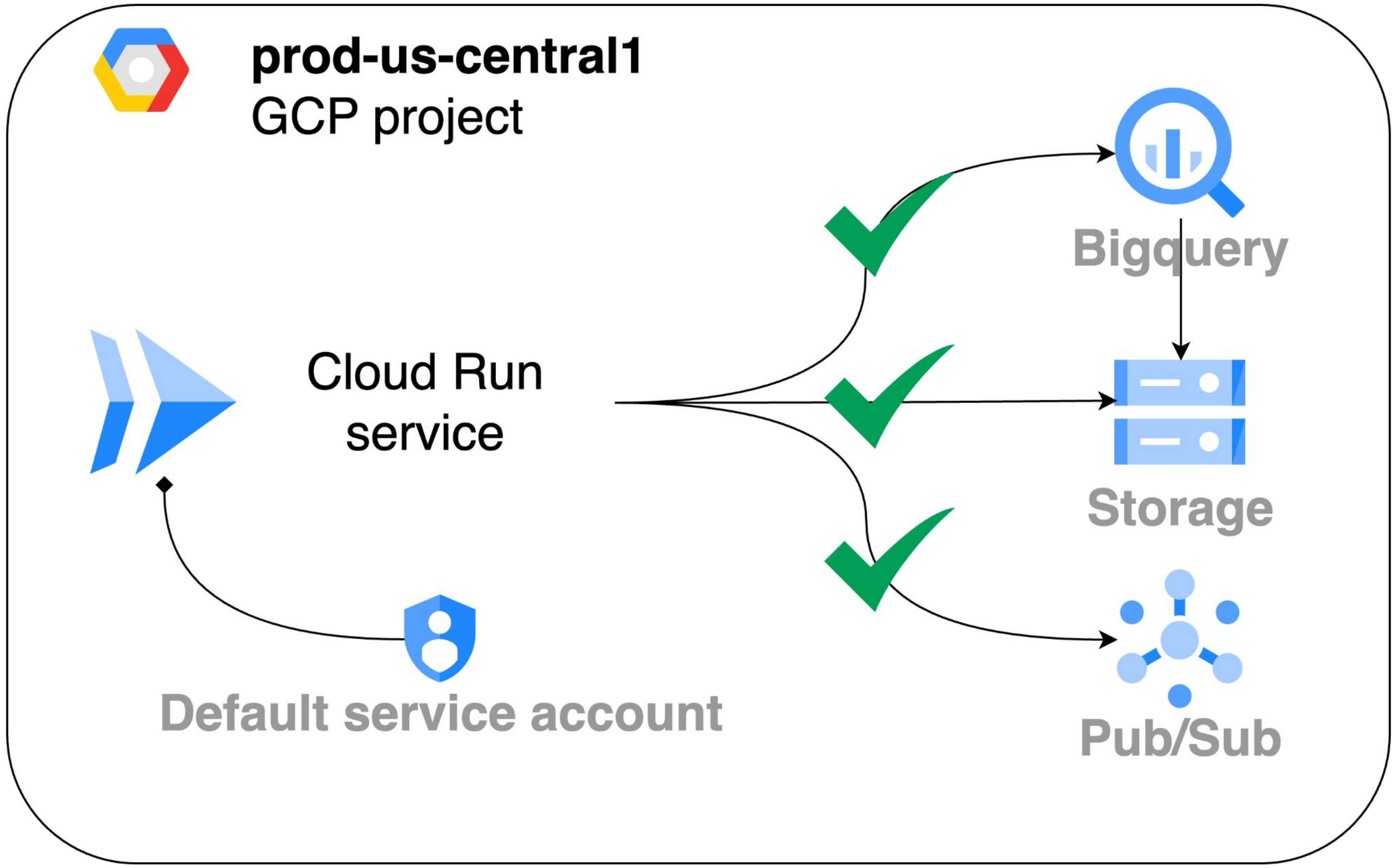
- Your code is your responsibility
- Keep a secret
- Establish access controls and permissions
- Get visibility into your functions
- Automate security controls for function code

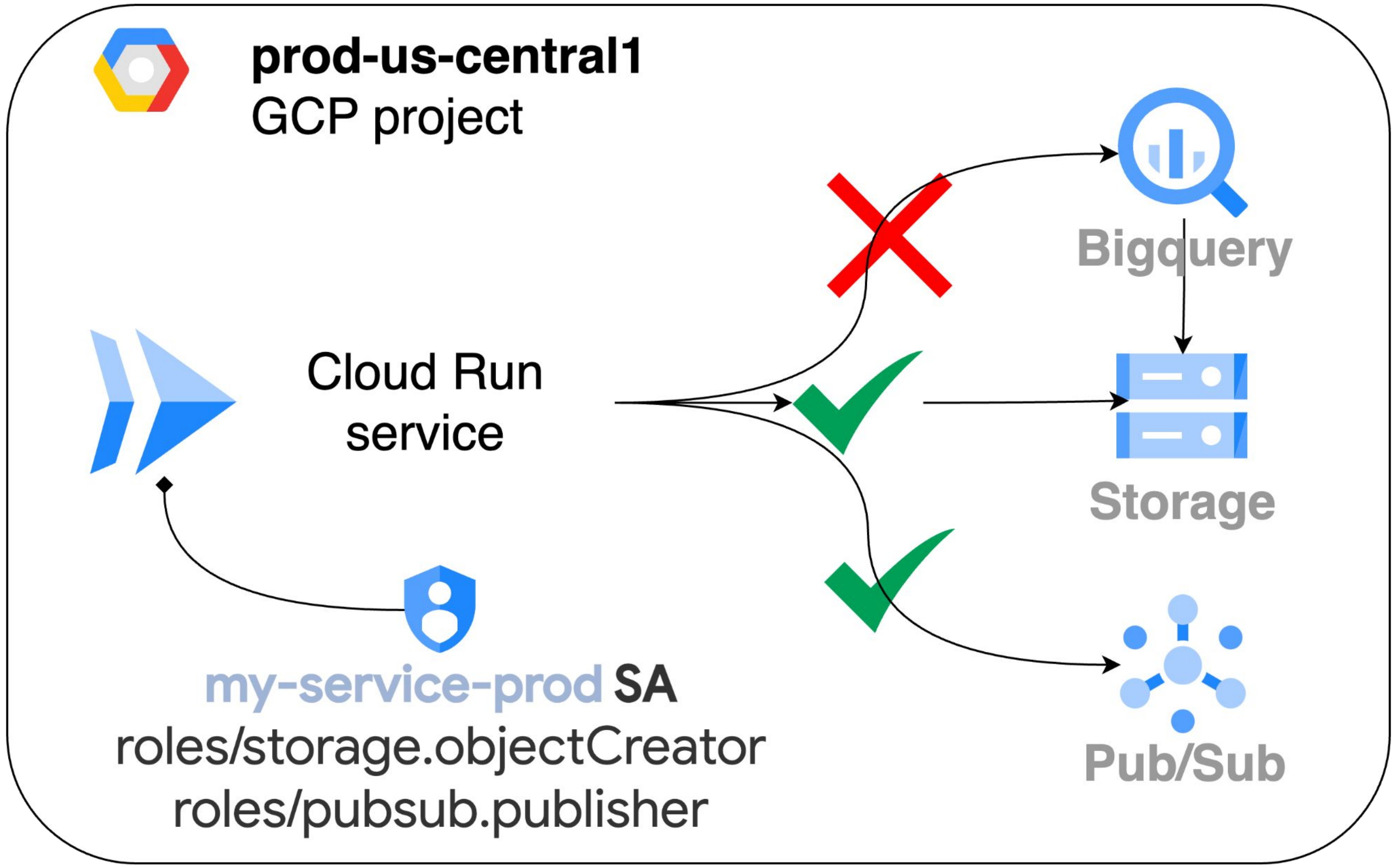
Secret Manager

- Mount secrets as volumes for real-time access to the latest version from Secret Manager, ideal for secret rotation
- Pass secrets via environment variables.
- Opt for pinning the version for stability instead of using the latest version

https://cloud.google.com/run/docs/configuring/services/secrets

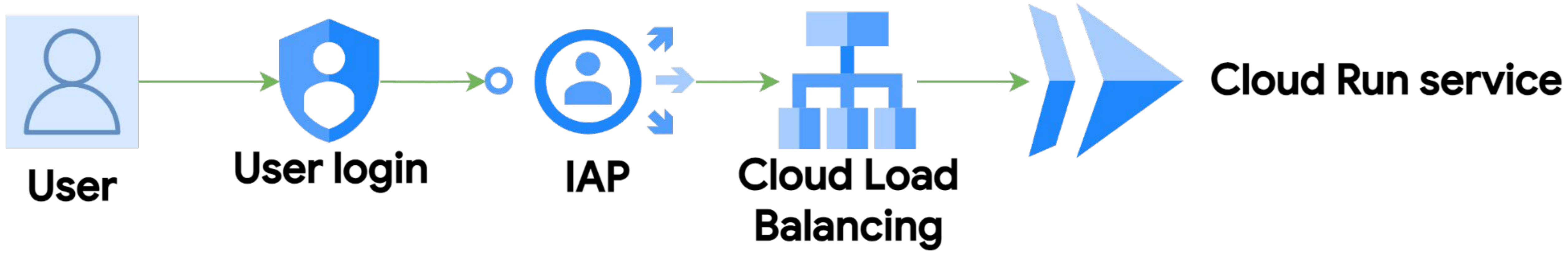
Service Accounts





GCP Identity aware proxy

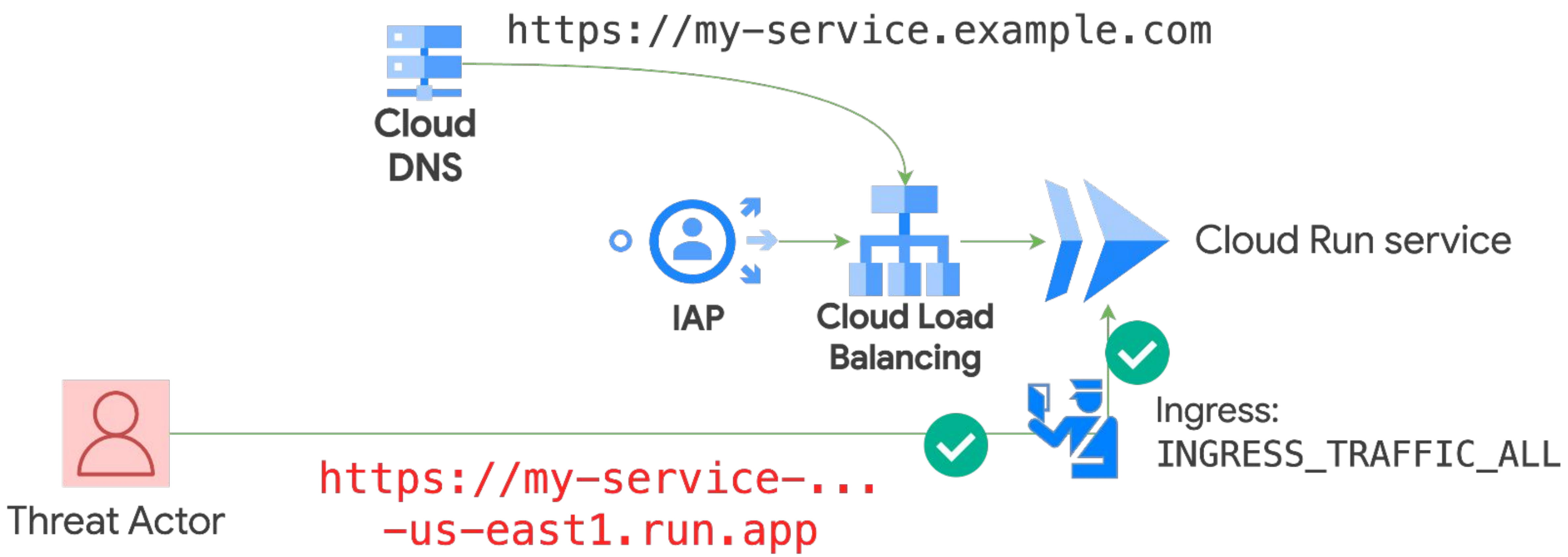
- Adds login page to the web app or the API
- Integrate with Google Workspace & GCP IAM
- AuthN & AuthZ
- Cloud Run, App Engine, GKE, Compute engine, ...



https://cloud.google.com/iap/docs/enabling-cloud-run

GCP Cloud Run ingress policy

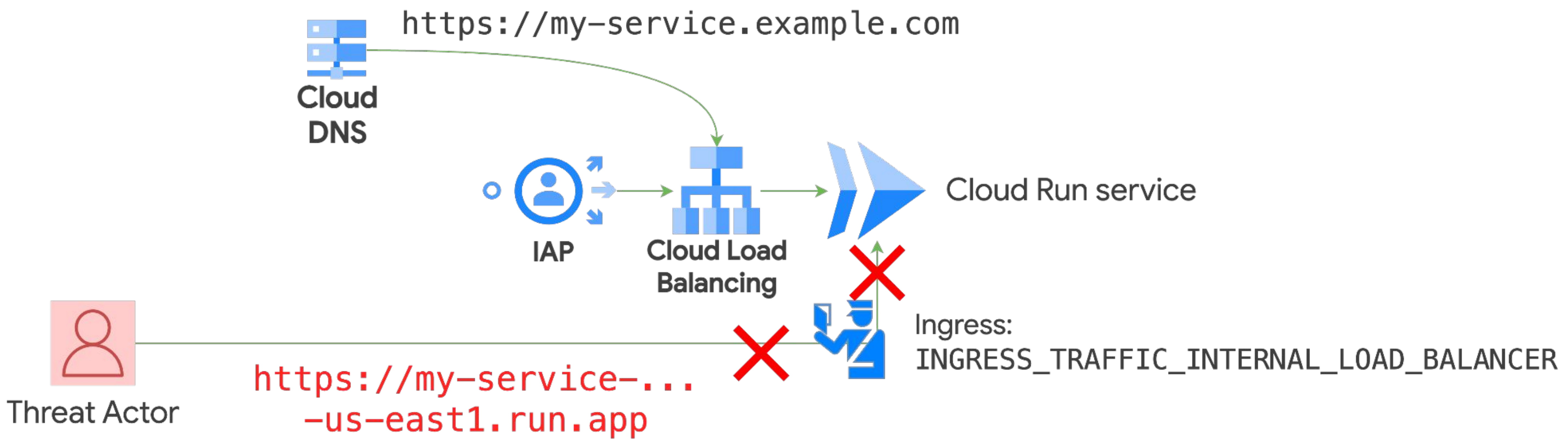
Ingress controls access to the URL of the Cloud Run service itself:
https://<serviceName>-<projectHash>-<region>.run.app



GCP Cloud Run ingress policy

3 options for ingress:

- INGRESS_TRAFFIC_ALL <- **default**
- INGRESS_TRAFFIC_INTERNAL_ONLY
- INGRESS_TRAFFIC_INTERNAL_LOAD_BALANCER



CI / CD

Choose your adventure!

Key takeaways

Serverless doesn't solve all architectural constraints, but it's advantageous for the right use case.

- Java <3 containers & Serverless
- Use the latest Java versions for both speed, security & DevX
- Your code is your responsibility
- Google cloud has a big community, and so is Java.
- Know your tools

Ready to build what's next?

Tap into **special offers** designed to help you **implement what you learned** at Google Cloud Next.

Scan the code to receive personalized guidance from one of our experts.

Or visit **g.co/next/24offers**

Thank you