# Springboot Production/Deployment Issues Scenario

Spring Boot Basics

├── @SpringBootApplication

│ ├── Combines @Configuration, @ComponentScan, @EnableAutoConfiguration

├── Auto Configuration

│ └── spring.factories → loads default configs

├── Application Properties

│ ├── application.properties / application.yml

│ ├── Profile-based configs (dev, prod)

│ └── External overrides via command line

**⚙️ 2. DevTools + Hot Reload**

swift

CopyEdit

DevTools

├── Auto Restart

│ ├── Monitors: src/main/java + src/main/resources

│ └── Exclude: META-INF, logs, static

├── LiveReload

├── Disabled in production

**🔍 3. Actuator + Observability**

bash

CopyEdit

Spring Boot Actuator

├── /actuator/health → Check DB, Redis, etc.

├── /actuator/metrics → GC, memory, HTTP stats

├── /actuator/beans → Loaded Spring Beans

├── /actuator/env → Environment properties

├── /actuator/threaddump → Thread-level issues

**💾 4. JVM + GC + Memory**

sql

CopyEdit

JVM & Memory

├── -Xms / -Xmx → Initial/Max Heap

├── GC Overhead → Full GC, memory leak

├── Heap Dump → Use VisualVM, MAT

├── GC Logs → -Xlog:gc

├── Metaspace → Class metadata

├── OOMKilled → Container memory exceeded

**⛓ 5. Threading + Async**

pgsql

CopyEdit

Thread Management

├── @Async → Needs @EnableAsync + Executor config

├── RejectedExecutionException → Pool full

├── Blocking Threads → Thread starvation, app freeze

├── Thread Leak → Threads not shutting down

├── Metrics → jvm.threads.live / jvm.threads.peak

**📦 6. Containers + Kubernetes**

pgsql

CopyEdit

Spring Boot in Docker/K8s

├── Readiness Probe → Ready to receive traffic

├── Liveness Probe → App still alive

├── SIGTERM → Triggers Spring shutdown hooks

├── terminationGracePeriodSeconds → Graceful shutdown

├── CrashLoopBackOff → Startup failures or health check fails

├── Exit Code 137 → OOMKilled

**🔁 7. Retry, Circuit Breaker, Timeouts**

mathematica

CopyEdit

Fault Tolerance

├── Retry (Spring Retry, Resilience4j)

│ └── maxAttempts, backoff

├── Circuit Breaker

│ ├── Fails fast after threshold

│ └── fallback methods

├── Timeouts

│ ├── Feign / RestTemplate → connectTimeout, readTimeout

**🛑 8. Common Errors & Crashes**

pgsql

CopyEdit

Error Scenarios

├── Port Already in Use → Change server.port

├── NoClassDefFoundError → Missing dependency

├── StackOverflowError → Recursive calls

├── IllegalStateException → Circular dependency

├── Too many open files → File leak

├── 500/503/504 → Downstream failure, timeout, unavailability

**🔎 9. Startup + Performance**

nginx

CopyEdit

Startup Optimization

├── Lazy Init → spring.main.lazy-initialization=true

├── Bean Loading Logs → debug=true

├── Startup Steps → Spring Boot 2.4+ (startup actuator endpoint)

├── Limit Component Scan → scanBasePackages

**🎯 10. Monitoring + Metrics**

vbnet

CopyEdit

Monitoring

├── Micrometer → Metrics collection framework

├── Exports to → Prometheus, Grafana

├── Key Metrics:

│ ├── jvm.memory.used

│ ├── jvm.gc.pause

│ ├── http.server.requests

**🧹 11. Cleanup + Shutdown**

Graceful Shutdown

├── @PreDestroy → Clean up on shutdown

├── DisposableBean → Custom logic

├── ApplicationListener<ContextClosedEvent>

├── Important for:

│ ├── Releasing threads

│ ├── Closing DB/Kafka connections

│ └── Flushing logs

**✅ Spring Boot – Master Series: JVM + GC + Crash + Downtime MCQs**

**🔧 Subset 1: Q1–Q25 with Deep Explanation**

**Q1. Your Spring Boot app goes down intermittently under high load. Logs show:**

java.lang.OutOfMemoryError: GC overhead limit exceeded.  
What should you do?

**A.** Increase -Xmx memory  
**B.** Tune GC settings  
**C.** Analyze memory leak via heap dump  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
This error indicates that the GC is spending too much time collecting but freeing very little memory. This often happens due to:

* Memory leaks
* Insufficient heap (-Xmx)
* Improper GC algorithm  
  **Solution:** Start with heap dump analysis → fix memory leak → then adjust GC or heap if needed.

**Q2. JVM crashed due to memory leak in production. What’s the first step to investigate?**

**A.** Delete logs  
**B.** Restart the app  
**C.** Analyze heap dump using VisualVM or Eclipse MAT  
**D.** Increase CPU  
✅ **Answer:** C

📘 **Explanation:**  
Heap dumps capture the state of memory — objects, references, and sizes. You can find which objects are not released and causing leaks.

**Q3. App is running, but /actuator/health returns "DOWN".**

What could be the cause?

**A.** Actuator misconfiguration  
**B.** DB or external dependency is unhealthy  
**C.** Port conflict  
**D.** Docker crashed  
✅ **Answer:** B

📘 **Explanation:**  
Even if the app is up, Spring Boot reports "DOWN" if any health contributor (like DB, Redis, etc.) is failing.  
Use /actuator/health for details or /health/{component} (like /health/db).

**Q4. Which Actuator endpoint gives thread-level visibility?**

**A.** /actuator/threaddump  
✅ **Answer:** B

📘 **Explanation:**  
/actuator/threaddump shows:

* Live thread states
* Locks
* Waiting threads  
  Useful for diagnosing **deadlocks**, **slow I/O**, or **blocked threads**.

**Q5. App becomes unresponsive; CPU = 100%. What’s your approach?**

**A.** Check GC  
**B.** Take thread dump  
**C.** Check load metrics  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
High CPU can be caused by:

* GC overuse
* Infinite loops
* Thread starvation  
  Use top/htop and thread dumps to locate the cause.

**Q6. Logs clean, but service is stuck — what to check?**

**A.** GC logs  
**B.** Deadlocks  
**C.** Open file handles  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
Logs may not show deep runtime problems. Use OS + JVM tools to analyze GC, open file limits, deadlocks, or native memory leaks.

**Q7. What does frequent Full GC indicate?**

**A.** All good  
**B.** Low memory + high pressure on old generation  
✅ **Answer:** B

📘 **Explanation:**  
Frequent Full GC means your heap is overused. Full GC pauses app completely → leads to latency → user timeouts.

**Q8. You get NoClassDefFoundError during runtime. What's likely wrong?**

**A.** Class not on runtime classpath  
✅ **Answer:** B

📘 **Explanation:**  
This happens when:

* Class was available during compile-time but not packaged into the JAR.
* Dependency missing from the final build.

**Q9. Error: Port already in use. What’s the fix?**

**A.** Change port  
**B.** Kill process  
✅ **Answer:** D (A or B)

📘 **Explanation:**  
Use lsof -i :8080 or netstat to find the PID, then kill or configure server.port in application.properties.

**Q10. Logs stop; service unreachable. First checks?**

**A.** Disk full  
**B.** JVM crash logs  
**C.** Container out-of-memory  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
No logs = likely system-level issue like:

* Disk full (app can't write)
* Pod OOM killed
* JVM crashed

**Q11. GC overhead limit exceeded — meaning?**

**A.** GC runs too much but frees <2% heap  
✅ **Answer:** B

📘 **Explanation:**  
This means JVM is spending 98%+ time on GC but reclaiming <2%. JVM throws this to prevent infinite GC loops.

**Q12. java.lang.StackOverflowError usually caused by?**

**A.** Infinite recursion  
✅ **Answer:** A

📘 **Explanation:**  
Happens when methods call each other endlessly — common in incorrectly wired beans or recursion bugs.

**Q13. App restarts constantly in Docker/K8s. Likely cause?**

**A.** Failed health checks  
✅ **Answer:** C

📘 **Explanation:**  
Kubernetes restarts pods when:

* livenessProbe fails
* App throws exception  
  Check health probe configs and pod logs.

**Q14. API gives no response, logs clean. What could be happening?**

**A.** Thread blocking  
**B.** GC freeze  
**C.** Load balancer config  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
Silent failures need full-stack visibility — logs + JVM metrics + networking.

**Q15. You get:**

Tomcat connector in failed state — cause?

**A.** Port conflict or crash  
✅ **Answer:** D (A or C)

📘 **Explanation:**  
Tomcat couldn’t bind port or start server threads. Could be:

* Port already in use
* Resource issue (no file handles)

**Q16. What does -Xmx1024m do?**

**A.** Sets max heap memory  
✅ **Answer:** B

📘 **Explanation:**  
This JVM flag sets the max heap size (1 GB here). Monitor -Xmx and -Xms carefully in production.

**Q17. Need to check what objects are causing memory leak. What to take?**

**A.** Heap dump  
✅ **Answer:** B

📘 **Explanation:**  
Thread dump = for blocked code  
**Heap dump = memory footprint**

**Q18. Best tools for Spring Boot memory issues?**

**A.** VisualVM, JConsole, Eclipse MAT  
✅ **Answer:** D

📘 **Explanation:**  
These tools help analyze heap usage, GC cycles, and memory leaks.

**Q19. K8s logs show: OOMKilled. What does it mean?**

**A.** App exceeded memory limit  
✅ **Answer:** B

📘 **Explanation:**  
Kubernetes terminates containers that exceed memory limits. Happens a lot with Spring apps having low Xmx.

**Q20. API timeout, no error logs. Where to start?**

**A.** Check thread pool  
**B.** GC  
**C.** Downstream timeouts  
**D.** All of the above  
✅ **Answer:** D

📘 **Explanation:**  
Timeouts with no logs = app stuck in IO, downstream, or thread starvation. Use **thread dump** + **metrics**.

**Q21. You suspect memory leak. What type of object should you look for in heap dump?**

**A.** Static maps, caches, sessions  
✅ **Answer:** A

📘 **Explanation:**  
Long-lived objects often cause leaks. Watch out for:

* Map<SessionId, UserData>
* Custom caches
* Streams never closed

**Q22. Spring Boot actuator endpoint to check memory usage?**

**A.** /actuator/metrics/jvm.memory.used  
✅ **Answer:** A

📘 **Explanation:**  
You can see memory used by heap, non-heap, metaspace, etc.

**Q23. JVM restart loop in logs. What's a stable way to prevent it?**

**A.** Use graceful shutdown and health checks  
✅ **Answer:** A

📘 **Explanation:**  
Configure readinessProbe, gracePeriod, and limit auto-restart on health check failures.

**Q24. Which log level gives performance + crash visibility in prod?**

**A.** INFO  
**B.** WARN  
**C.** ERROR  
**D.** INFO + ERROR  
✅ **Answer:** D

📘 **Explanation:**  
Keep INFO for business ops and ERROR for failures. Avoid DEBUG/TRACE in prod unless needed.

**Q25. You want to catch app crashes and auto-restart it. What’s one good approach?**

**A.** Use process manager (PM2, systemd) or K8s liveness probe  
✅ **Answer:** A

📘 **Explanation:**  
Use infra like:

* systemd on Linux
* livenessProbe in Kubernetes
* supervisord

**Subset 2 – Q26–Q50: JVM + GC + Crash + Downtime + No Response Issues (with Deep Explanations)**

**Q26. After a memory-intensive job runs, your Spring Boot app starts freezing for seconds. What could be the cause?**

**A.** Minor GC  
**B.** Full GC pauses due to heap saturation  
**C.** DB connections not released  
**D.** Wrong profile active

✅ **Answer:** B

📘 **Explanation:**  
When heap fills, the JVM triggers **Full GC**, which stops all threads until memory is cleaned. Use GC logs or JDK Flight Recorder to analyze.

**Q27. Your app gives random OutOfMemoryError: Metaspace. What likely causes this?**

**A.** Too many dynamic class loads (e.g. proxies, reflection)  
**B.** Heap memory low  
**C.** App not deployed as JAR  
**D.** Wrong DB pool config

✅ **Answer:** A

📘 **Explanation:**  
Metaspace stores class metadata. Memory leaks due to excessive proxy class generation (like in Spring AOP, Hibernate) can fill metaspace.

**Q28. Your service receives requests but never responds, even after retries. Logs are silent. Which is most useful?**

**A.** Thread dump  
**B.** Access logs  
**C.** DevTools  
**D.** Swagger UI

✅ **Answer:** A

📘 **Explanation:**  
Silent hangs are almost always due to blocked threads, stuck IO, or thread starvation. A thread dump shows what threads are doing (e.g., stuck in socket read).

**Q29. You see frequent PermGen space errors. Which Java version are you using?**

**A.** Java 11+  
**B.** Java 8  
**C.** Java 6 or 7  
**D.** Java 17

✅ **Answer:** C

📘 **Explanation:**  
PermGen was replaced by Metaspace in Java 8+. If you see this error, it means the app is using an **older JVM** (not suitable for production anymore).

**Q30. After many deployments, your container hits "too many open files". What is the likely cause?**

**A.** File handles not closed  
**B.** Leak in FileInputStream or DB connections  
**C.** OS file limit too low  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Each opened file, socket, and JDBC connection consumes a file descriptor. Not closing streams/sockets leads to exhaustion.  
Check with: ulimit -n and lsof -p PID.

**Q31. You're using a large object cache (e.g. Guava). App slows over time. What could be the cause?**

**A.** Cache size grows without limit  
✅ **Answer:** A

📘 **Explanation:**  
If your cache has no eviction policy (maximumSize, expireAfterWrite), it becomes a memory leak. Guava, Caffeine, and even manual maps can be dangerous if unchecked.

**Q32. What is the best way to monitor garbage collection behavior?**

**A.** -verbose:gc flag  
**B.** GC logs with timestamps  
**C.** Use JVisualVM or JFR  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Use JVM flags like:

bash

CopyEdit

-XX:+PrintGCDetails -Xlog:gc\*:file=gc.log

For rich visual insights: **JDK Flight Recorder**, **VisualVM**, **Grafana dashboards** (via Micrometer).

**Q33. You get java.lang.IllegalStateException: Failed to load ApplicationContext. What's the usual cause?**

**A.** Missing Bean  
**B.** Misconfigured Spring context  
**C.** Missing properties file  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Spring fails to start context when:

* @Autowired fails
* Required beans not present
* Missing application.properties
* Profiles not loaded

**Q34. JVM is killed suddenly without logs. Where else can you look?**

**A.** hs\_err\_pid\*.log file  
✅ **Answer:** A

📘 **Explanation:**  
When JVM crashes due to **native memory, SIGKILL, core dumps**, it leaves crash logs like:

lua

CopyEdit

hs\_err\_pid1234.log

Check /tmp, app root, or container filesystem.

**Q35. A service is stuck during startup on cloud. Health probe keeps failing. What do you check?**

**A.** Readiness and liveness probe thresholds  
✅ **Answer:** A

📘 **Explanation:**  
Sometimes the app takes time to initialize (DB warmup, cache preload). If your readinessProbe.initialDelaySeconds is too low, the pod is marked as failed and restarted repeatedly.

**Q36. You want to know if Spring Boot is ready to receive traffic. What Actuator endpoint helps?**

**A.** /actuator/health/readiness  
✅ **Answer:** A

📘 **Explanation:**  
When Spring Boot Actuator + Kubernetes is used, readiness probes are mapped to this endpoint. Only when it's UP should the service receive requests.

**Q37. You're seeing long delays in REST responses. GC logs show repeated full GCs every few seconds. What can you try?**

**A.** Increase heap  
**B.** Switch to G1GC  
**C.** Analyze GC logs for memory pressure  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Repeated Full GCs → app keeps pausing → latency increases. Use a modern GC like G1 or ZGC and adjust heap sizing.

**Q38. A containerized app fails with Exit Code 137. What does it mean?**

**A.** Port not found  
**B.** App crashed due to OOM  
**C.** Missing properties  
**D.** Timeout

✅ **Answer:** B

📘 **Explanation:**  
Exit code 137 = terminated with **SIGKILL**, often due to **memory limits** breached. Check pod memory usage.

**Q39. You want to trigger heap dump automatically on OutOfMemoryError. Which flag do you use?**

**A.** -XX:+HeapDumpOnOutOfMemoryError  
✅ **Answer:** A

📘 **Explanation:**  
Use with:

bash

CopyEdit

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/dumps/

**Q40. You're tuning GC. Which flag gives summary of GC events in human-readable format (JDK 11+)?**

**A.** -Xlog:gc  
✅ **Answer:** A

📘 **Explanation:**  
In JDK 9+, use unified logging:

bash

CopyEdit

-Xlog:gc\*:file=gc.log:time

**Q41. Which of the following tools can show heap growth over time in a running JVM?**

**A.** VisualVM  
**B.** JConsole  
**C.** Micrometer + Grafana  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
All of these can monitor live heap, metaspace, GC frequency, etc. Use in staging or with remote access in production.

**Q42. How do you configure a pod to get graceful shutdown time during JVM termination?**

**A.** Set terminationGracePeriodSeconds in deployment  
✅ **Answer:** A

📘 **Explanation:**  
This allows Spring Boot app to clean up, flush logs, release resources before Kubernetes SIGKILLs it.

**Q43. Service is slow on first hit but fast after. What might be the reason?**

**A.** Lazy loading of classes/beans  
**B.** JVM JIT warmup  
**C.** Hibernate first connection  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Cold start latency is common in Spring Boot due to bean creation, proxy generation, DB warmup. Can be optimized using **ApplicationRunner pre-warm** logic.

**Q44. Which JVM flag can cause long GC pauses in high-throughput services?**

**A.** -XX:+UseSerialGC  
✅ **Answer:** A

📘 **Explanation:**  
Serial GC is stop-the-world and **single-threaded** — terrible for servers. Use **G1GC** or **ZGC** in production.

**Q45. Your app hangs when many threads are active. What should you check?**

**A.** Thread pool size  
**B.** Blocking code (e.g. RestTemplate)  
**C.** Use of @Async without Executor config  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
By default, Spring uses a small async thread pool. If unconfigured, @Async tasks can pile up and hang.

**✅ Subset 3 – Q46 to Q75: JVM, GC, Threading, Service Failures, Timeout, Container & Prod Troubleshooting**

Each question continues with real-world issues, root-cause thinking, and Spring Boot-specific explanations.

**Q46. Your Spring Boot app is using @Async, but async tasks never run. Why?**

**A.** No @EnableAsync annotation  
✅ **Answer:** A

📘 **Explanation:**  
Spring requires @EnableAsync to scan and execute async methods. Also ensure a thread pool executor is configured or the default will be too small.

**Q47. You have heavy database load and the app freezes. What thread pool is most likely overloaded?**

**A.** HTTP Worker Pool  
**B.** Spring Scheduler  
**C.** Connection pool (e.g., HikariCP)  
**D.** JVM Garbage Collector

✅ **Answer:** C

📘 **Explanation:**  
Database connections are pooled. If all connections are in use and not released (e.g., due to slow queries), threads block → app freezes.

**Q48. What could happen if your thread pool size is too small for scheduled jobs?**

**A.** Tasks are dropped  
**B.** Tasks are delayed or queued  
**C.** App becomes idle  
**D.** Scheduler shuts down

✅ **Answer:** B

📘 **Explanation:**  
Spring’s @Scheduled tasks run in a default pool (1 thread). Configure TaskScheduler with a proper pool size to avoid bottlenecks.

**Q49. You hit /actuator/metrics/jvm.threads.live and see a growing thread count. What’s the likely issue?**

**A.** Memory leak  
**B.** Thread leak — threads not shut down after execution  
**C.** GC issue  
**D.** REST call delay

✅ **Answer:** B

📘 **Explanation:**  
Threads created dynamically (e.g., from custom pools or Executors.newThread) need to be shut down. Monitor with /metrics/jvm.threads.\*.

**Q50. App freezes with no exceptions in logs. Which log level might give better diagnostics?**

**A.** DEBUG for org.springframework  
**B.** TRACE for root  
**C.** INFO for web  
**D.** WARN for jvm.memory

✅ **Answer:** A

📘 **Explanation:**  
Enable DEBUG for internal Spring and web packages to inspect bean lifecycle, slow filters, failed autowiring, etc.

**Q51. You are using HikariCP. Your app throws Connection pool is exhausted. Which config helps?**

**A.** spring.datasource.hikari.maximum-pool-size  
✅ **Answer:** A

📘 **Explanation:**  
Increase the pool size or reduce connection usage. Use /actuator/metrics/hikaricp.connections to monitor real-time stats.

**Q52. Your microservice is slow because another service is down. What is the best way to avoid system-wide failure?**

**A.** Use Circuit Breaker (e.g., Resilience4j)  
✅ **Answer:** A

📘 **Explanation:**  
Circuit breakers prevent cascading failures by short-circuiting calls to known-failing services and fallback immediately.

**Q53. How can you avoid retrying on non-retriable errors like 400 or 404?**

**A.** Use @Retryable(include=...)  
**B.** Use conditional retry logic in code  
**C.** Use Resilience4j’s retry config with retryOnResult()  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Retries should only occur on transient errors (timeouts, 500). Retrying on 404 or 400 wastes resources.

**Q54. Your logs show: java.net.SocketTimeoutException: Read timed out. What does it indicate?**

**A.** Connection failed  
**B.** DNS resolution failed  
**C.** Server did not respond in time  
**D.** Request was malformed

✅ **Answer:** C

📘 **Explanation:**  
The client made a connection but did not receive a response within the timeout. Increase readTimeout or optimize the backend.

**Q55. Which of these helps diagnose which beans are created at startup?**

**A.** Enable debug logs  
**B.** Use /actuator/beans  
**C.** Use ApplicationRunner logging  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
All options give insight into bean creation. /actuator/beans shows the structure, while DEBUG logs show creation order.

**Q56. Your container logs show SIGTERM but no cleanup happens. Why?**

**A.** No @PreDestroy or graceful shutdown handler is defined  
✅ **Answer:** A

📘 **Explanation:**  
To intercept shutdown events, Spring needs @PreDestroy, DisposableBean, or shutdown hooks (e.g., for releasing connections or saving state).

**Q57. You want to handle JVM shutdown cleanly in Spring Boot. What’s best?**

**A.** Implement DisposableBean  
**B.** Use @PreDestroy  
**C.** Use ApplicationListener<ContextClosedEvent>  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Spring offers multiple shutdown hooks. Use all if needed — for thread pools, DB, Kafka, cache, etc.

**Q58. A critical API fails with 503 Service Unavailable. What’s the likely cause?**

**A.** Service down  
**B.** Health probe failure  
**C.** Circuit breaker opened  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
503 = service temporarily unavailable. It could be due to:

* Manual restart
* Health check failure
* Infra scaling issues

**Q59. A memory-intensive task slows down everything else. What architecture pattern helps?**

**A.** Run the task in a worker queue (event-driven)  
✅ **Answer:** A

📘 **Explanation:**  
Heavy workloads should be moved to async processing (e.g., RabbitMQ, Kafka). Keep your APIs responsive.

**Q60. You get Too many open files in logs. How to view file handle usage?**

**A.** lsof -p <pid>  
✅ **Answer:** A

📘 **Explanation:**  
lsof lists all open files per process — can include logs, DB sockets, HTTP clients, etc.

**Q61. In Docker, Spring Boot stops responding under load. You suspect memory limit. How to verify?**

**A.** docker stats  
**B.** Enable /actuator/metrics/jvm.memory.used  
**C.** Add -XX:+PrintGCDetails  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
All tools give memory usage from different perspectives: container, JVM, and logs.

**Q62. You notice sudden memory drops every 5–10 minutes. What's likely?**

**A.** GC cycle running (especially Full GC)  
✅ **Answer:** A

📘 **Explanation:**  
Use /actuator/metrics/jvm.gc.pause to track. Drop patterns = GC behavior. Tune GC or analyze memory pressure.

**Q63. A single service is slow, but other services are fast. What’s a quick way to confirm?**

**A.** Use distributed tracing (e.g., Sleuth + Zipkin)  
✅ **Answer:** A

📘 **Explanation:**  
Tracing tells you which microservice is slow and what method is causing latency.

**Q64. How do you track request-response latency in Spring Boot?**

**A.** Actuator + Micrometer timers  
✅ **Answer:** A

📘 **Explanation:**  
Spring Boot + Micrometer provides metrics like http.server.requests for latency, status codes, and endpoints.

**Q65. If a Spring Boot service keeps restarting in Kubernetes, what should you first check?**

**A.** Logs and readiness probe  
✅ **Answer:** A

📘 **Explanation:**  
K8s restarts pods that fail readiness. If app is slow to start, increase probe delay or fix startup config issues.

**Q66. In production, you want to identify GC bottlenecks. Which metric is best?**

**A.** /actuator/metrics/jvm.gc.pause  
✅ **Answer:** A

📘 **Explanation:**  
Micrometer tracks the time GC takes. Spikes indicate GC is stalling the app.

**Q67. What causes RejectedExecutionException in @Async tasks?**

**A.** Thread pool is full or shutdown  
✅ **Answer:** A

📘 **Explanation:**  
Configure ThreadPoolTaskExecutor with corePoolSize, maxPoolSize, and queueCapacity to avoid overflow.

**Q68. You’re using Feign Client and it causes app to hang. What's missing?**

**A.** Read + connect timeouts in config  
✅ **Answer:** A

📘 **Explanation:**  
Without timeouts, HTTP clients may hang indefinitely on dead endpoints. Always define feign.client.config.default.connectTimeout.

**Q69. How to limit how many requests your service handles in parallel?**

**A.** Use thread pool + rate limiter  
✅ **Answer:** A

📘 **Explanation:**  
Limit via:

* API gateway
* Resilience4j rate limiter
* Controller level semaphores

**Q70. App memory usage is stable, but latency increases under load. What could be the cause?**

**A.** Thread starvation  
**B.** Blocking I/O  
**C.** DB contention  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Latency often comes from **non-memory issues** — queue backlog, thread limits, or overloaded database.

**Q71. You want to profile production performance for a short time. What tool helps?**

**A.** JFR (Java Flight Recorder)  
✅ **Answer:** A

📘 **Explanation:**  
JFR is a low-overhead profiler built into the JVM. Use with -XX:StartFlightRecording.

**Q72. Your app is underperforming after enabling SSL. What could cause it?**

**A.** SSL handshake latency  
**B.** No HTTP2  
**C.** Blocking filters  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
SSL adds overhead. Use connection pooling + HTTP2 + async filters to improve performance.

**Q73. What happens if you keep creating new threads manually (e.g., new Thread().start())?**

**A.** Potential thread leak  
✅ **Answer:** A

📘 **Explanation:**  
Use thread pools. Manual threads = no cleanup, unbounded growth, app crash risk.

**Q74. Your thread pool has coreSize=10, maxSize=20, queueSize=50. What happens after 70 requests?**

**A.** 10 core threads handle initial  
**B.** 50 are queued  
**C.** 10 more threads are created  
**D.** 71st request is rejected

✅ **Answer:** D

📘 **Explanation:**  
10 core + 50 queue + 10 max = 70. The 71st is rejected if no CallerRunsPolicy is set.

**Q75. What's the most accurate way to analyze Spring Boot startup time?**

**A.** spring.main.log-startup-info=true  
**B.** Use ApplicationListener<ApplicationReadyEvent>  
**C.** Use Spring Boot Startup Actuator endpoint (2.3+)  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Spring Boot now provides startup actuator endpoint, but logging + events also help track exact timings.

**✅ Subset 4 – Q76 to Q100: Final 25 Real-World Spring Boot Prod Issue MCQs (with Explanations)**

**Q76. Your service is deployed in a Kubernetes cluster and keeps restarting with CrashLoopBackOff. What do you check?**

**A.** Pod logs  
**B.** Readiness and liveness probes  
**C.** Resource limits  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
CrashLoopBackOff usually means the app is crashing after every restart. Reasons:

* Wrong health probe config
* OOMKilled due to low memory limits
* App startup failure

**Q77. Spring Boot app freezes during shutdown. What might be the cause?**

**A.** Blocking code in @PreDestroy  
✅ **Answer:** A

📘 **Explanation:**  
Improper shutdown logic can hang the app (e.g., waiting for an async task to finish). Use timeout and proper Executor.shutdown() logic.

**Q78. In a production app, you want to delay health probe until cache is fully loaded. What’s the right technique?**

**A.** Use ReadinessState.REFUSING\_TRAFFIC until ready  
✅ **Answer:** A

📘 **Explanation:**  
Spring Boot lets you control readiness state programmatically using ApplicationAvailability to prevent premature traffic routing.

**Q79. You want to check how much time is spent in GC over a period. Which metric is useful?**

**A.** /actuator/metrics/jvm.gc.pause  
✅ **Answer:** A

📘 **Explanation:**  
Micrometer records total GC pause time — useful to correlate with latency spikes.

**Q80. When does Spring Boot trigger its graceful shutdown?**

**A.** On SIGTERM signal (e.g., container shutdown)  
✅ **Answer:** A

📘 **Explanation:**  
Spring listens for SIGTERM and invokes shutdown hooks like @PreDestroy, DisposableBean.

**Q81. How can you reduce Spring Boot startup time in large applications?**

**A.** Remove unnecessary auto-config  
**B.** Use lazy initialization  
**C.** Profile slow beans with StartupStepLogger  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Spring Boot 2.4+ supports startup profiling. Also use spring.main.lazy-initialization=true to improve cold start time.

**Q82. Which tool lets you continuously profile a JVM in production without downtime?**

**A.** JDK Flight Recorder (JFR)  
✅ **Answer:** A

📘 **Explanation:**  
JFR is low-overhead and built into JDK 11+. It records heap, threads, GC, and method traces.

**Q83. You're getting BlockedThreadDetectionException in a reactive app. What does it mean?**

**A.** Blocking call detected in a non-blocking thread  
✅ **Answer:** A

📘 **Explanation:**  
In WebFlux/Reactor apps, blocking operations like JDBC, file IO cause thread starvation and defeat reactivity.

**Q84. A slow startup is caused by large classpath scanning. What can help?**

**A.** Use @SpringBootApplication(scanBasePackages = "...")  
✅ **Answer:** A

📘 **Explanation:**  
Limit the scan to only required packages to reduce bean discovery overhead.

**Q85. You get BeanInstantiationException: Circular reference. What’s the best solution?**

**A.** Use constructor injection + @Lazy on one side  
✅ **Answer:** A

📘 **Explanation:**  
Circular dependencies can be broken using @Lazy or restructuring logic into a third class.

**Q86. Your application takes up 2 GB RAM but only uses 500MB heap. What could be the cause?**

**A.** Native memory usage (e.g., threads, direct buffers)  
✅ **Answer:** A

📘 **Explanation:**  
Tools like ps, top, and JFR show total memory, not just heap. Native threads, Netty buffers, or logging buffers can cause high usage.

**Q87. You observe high gc.live.data.size. What does this indicate?**

**A.** Retained heap size is growing  
✅ **Answer:** A

📘 **Explanation:**  
Live data size = amount of heap retained after GC. If it keeps increasing, there may be a memory leak or objects aren't released.

**Q88. What can cause thread starvation in Spring Boot?**

**A.** Long-running tasks in a small pool  
**B.** Blocking database queries  
**C.** Synchronous calls in @Async methods  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Always match thread pool size to app load. Use separate pools for I/O and computation.

**Q89. Your app is deployed in Docker but shows timezone-related issues. What’s missing?**

**A.** Set TZ environment variable in Dockerfile  
✅ **Answer:** A

📘 **Explanation:**  
Docker containers use UTC by default. Set the timezone explicitly if your app is sensitive to local time.

**Q90. A critical queue (e.g., Kafka) is unavailable, and your app crashes. How can you prevent this?**

**A.** Add Circuit Breaker and retry logic  
**B.** Delay startup using a custom health indicator  
**C.** Fail fast with meaningful logs  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Use custom health indicators to check downstream availability. Prevent startup if required dependencies are down.

**Q91. Your Spring Boot app is leaking memory slowly. What JVM tool can help find it?**

**A.** Eclipse MAT  
**B.** VisualVM  
**C.** JFR  
**D.** All of the above

✅ **Answer:** D

📘 **Explanation:**  
Each has a different strength:

* **MAT**: deep heap dump analysis
* **VisualVM**: live monitoring
* **JFR**: continuous profiling

**Q92. If you want to simulate CPU or memory stress in dev/test, what tool is useful?**

**A.** stress Linux command  
✅ **Answer:** A

📘 **Explanation:**  
Example: stress --cpu 2 --vm 2 --vm-bytes 512M  
Helps test how your app behaves under pressure.

**Q93. In a reactive app, using blocking code causes which major issue?**

**A.** Blocks event loop → causes backpressure and timeouts  
✅ **Answer:** A

📘 **Explanation:**  
Reactor/Netty runs on non-blocking I/O threads. Blocking them stops request processing.

**Q94. You want your app to shutdown after processing in-flight requests. What helps?**

**A.** Use SpringApplication.setRegisterShutdownHook(true)  
✅ **Answer:** A

📘 **Explanation:**  
Spring provides lifecycle hooks to gracefully finish work before stopping.

**Q95. You observe this message in logs: Full GC (System.gc()). What might be the cause?**

**A.** Manual or 3rd-party library triggering System.gc()  
✅ **Answer:** A

📘 **Explanation:**  
Avoid calling System.gc() — it forces Full GC and pauses the app. Use -XX:+DisableExplicitGC to block it.

**Q96. What JVM metric shows how often GC is being triggered?**

**A.** /actuator/metrics/jvm.gc.memory.promoted  
✅ **Answer:** A

📘 **Explanation:**  
Memory promoted from young gen to old gen = frequency of GC. High frequency may mean heap pressure.

**Q97. You want to tune GC in production. What are safe defaults in modern JDK 11+?**

**A.** G1GC with reasonable -Xmx, -Xms, and GC logs  
✅ **Answer:** A

📘 **Explanation:**  
Use G1GC as it balances latency and throughput. Log GC with:

bash

CopyEdit

-Xlog:gc\*:file=gc.log:time,level,tags

**Q98. What’s the impact of high CPU + low memory usage in Spring Boot?**

**A.** Likely thread contention or infinite loop  
✅ **Answer:** A

📘 **Explanation:**  
High CPU with low memory usually indicates code issues (tight loops, unbounded retry, thread leak).

**Q99. You want to stop Spring Boot startup if DB is down. What can help?**

**A.** Set spring.datasource.initialize=false and fail-fast  
✅ **Answer:** A

📘 **Explanation:**  
Spring retries DB connection by default. Set it to fail on startup if required resources are unavailable.

**Q100. How do you reduce Docker image size for Spring Boot?**

**A.** Use layered JAR or jlink/jib/paketo buildpacks  
✅ **Answer:** A

📘 **Explanation:**  
Layered JAR separates dependencies from app code, reducing rebuilds. Use spring-boot:build-image or jib.