Response

1. Interview-Style Opening

"When selecting a Garbage Collector in modern Java (Java 21 or the latest Java 25), the choice is no longer just 'G1 vs. Parallel'. The landscape has shifted dramatically with the maturation of **Generational ZGC** and **Generational Shenandoah**. For most enterprise applications, I look at three main options: **G1** for a balance of throughput and latency, **ZGC** for ultra-low latency, and **Parallel GC** if raw throughput is the only metric that matters."

2. High-Level Overview (The Big Three)

In Java 25 (LTS), the ecosystem of Garbage Collectors has evolved to handle massive heaps (Terabytes) with sub-millisecond pauses.

1. **G1 GC (Garbage-First) - The Default**
    - **Best For:** General-purpose applications, microservices, and heaps from 4GB to 32GB.
    - **Status:** Default since Java 9. In Java 25, it has improved region pinning and ergonomic defaults.[1]
    - **Mechanism:** Splits the heap into regions (Eden, Survivor, Old) and cleans the "most garbage-dense" regions first.
2. **Generational ZGC (The Game Changer)**
    - **Best For:** Low-latency/Real-time systems (Trading, Gaming) and massive heaps (up to 16TB).
    - **Status:** Production-ready in Java 21, **Default Mode** for ZGC in Java 25.[2] [3]
    - **Mechanism:** Unlike the old ZGC, the *Generational* version separates Young and Old objects. This allows it to collect young objects (which die fast) very cheaply, solving the throughput issue of the original ZGC.
    - **Key Stat:** Max pause time < 1ms, regardless of heap size.
3. **Generational Shenandoah**
    - **Best For:** Applications requiring low latency but running on Red Hat distributions or specific OpenJDK builds.
    - **Status:** Became a production feature in Java 25.[4]
    - **Mechanism:** Similar to ZGC (concurrent compaction), but now uses generations to avoid scanning the entire heap for every cycle.

3. Visual Comparison Table

I often use this table to decide which collector to use for a new service:

Feature	**Parallel GC**	**G1 GC** (Default)	**Generational ZGC** (Java 21+)
Primary Goal	Max Throughput	Balance	Min Latency
Pause Times	Long (Stop-the-World)	Controlled (e.g., 200ms)	Micro (< 1ms)
Throughput	Highest	High	High (improved in Gen ZGC)
Heap Size	Small to Medium	Medium to Large	Large to Massive (TB+)
CPU Usage	Low overhead	Medium	Higher (Concurrent threads)

Note: "Throughput" means the percentage of time the CPU spends running your code vs. running the GC.

4. Java Code (Configuration)

We don't "code" the GC, but we configure it via JVM flags. Here is how I would configure a **Low-Latency Payment Service** running on Java 25.

```
# Dockerfile or Startup Script

java \
# 1. Enable Generational ZGC (Explicitly, though default for ZGC in Java 25)
  -XX:+UseZGC \
  -XX:+ZGenerational \
# 2. Set Heap Size (ZGC loves memory, give it headroom)
  -Xms4G -Xmx4G \
# 3. Tuning (Optional: ZGC is mostly self-tuning)
  -XX:SoftMaxHeapSize=3G \
  -jar payment-service.jar
```

For a **Batch Processing Job** (where latency doesn't matter, but finishing fast does):

```
java \
  -XX:+UseParallelGC \
  -Xms2G -Xmx2G \
  -jar batch-processor.jar
```

5. How to Explain This to the Interviewer

"In my last project, we migrated our user-facing API from **G1** to **Generational ZGC** when we upgraded to Java 21.

With G1, we were seeing occasional 'Stop-the-World' pauses of 200-300ms during high load, which caused p99 latency spikes.

By switching to `-XX:+UseZGC -XX:+ZGenerational`, those pauses dropped to under **1 millisecond**. The trade-off was a slight increase in CPU usage (about 10%) because ZGC runs concurrently with the application threads, but since we had spare CPU capacity, the smoother user experience was worth it.

However, for our nightly ETL jobs, we stuck with **Parallel GC** because it simply processes more records per second than any concurrent collector."

6. Edge Cases and Follow-Up Questions

Q: When should you NOT use ZGC?
A: If you are strictly constrained on CPU (e.g., a small container with 0.5 vCPU). ZGC needs concurrent threads to keep up with allocation. If the CPU is saturated, the GC falls behind, and you hit an 'Allocation Stall,' which freezes the app entirely.

Q: What about Epsilon GC?
A: That's a 'No-Op' collector. It allocates memory but never cleans it. I only use it for performance testing (benchmarking code without GC interference) or extremely short-lived lambda tasks. [2]

Q: Does Java 25 still support CMS?
A: No, CMS (Concurrent Mark Sweep) was removed in Java 14. You must use G1, ZGC, or Shenandoah instead.

⁂

1. https://dev.java/learn/jvm/tool/garbage-collection/java-specifics/
2. https://www.linkedin.com/posts/amitesh-kumar-ray_java25-garbagecollector-jvm-activity-7382233440433254401-ulL6
3. https://andrewbaker.ninja/2025/12/03/deep-dive-pauseless-garbage-collection-in-java-25/
4. https://inside.java/2025/10/20/jdk-25-performance-improvements/
5. https://foojay.io/today/the-ultimate-10-years-java-garbage-collection-guide-2016-2026-choosing-the-right-gc-for-every-workload/
6. https://www.morling.dev/blog/lower-java-tail-latencies-with-zgc/
7. https://www.oracle.com/in/java/technologies/javase/25-relnote-issues.html
8. https://www.geeksforgeeks.org/java/types-of-jvm-garbage-collectors-in-java-with-implementation-details/
9. https://blog.jetbrains.com/idea/2025/09/java-25-lts-and-intellij-idea/
10. https://www.perfectscale.io/blog/java-optimization