
aws
SUMMIT
SAN FRANCISCO | APRIL 20–21, 2022

OPN302

Enhancing Java memory management with generational Shenandoah

Kelvin Nilsen (he/him)
Principal Software Engineer
AWS

Agenda

Pauseless Shenandoah GC

Generational GC concepts

Shenandoah generations

Qualifying use cases

Getting started with generational Shenandoah

Configuration options

Pauseless Shenandoah GC

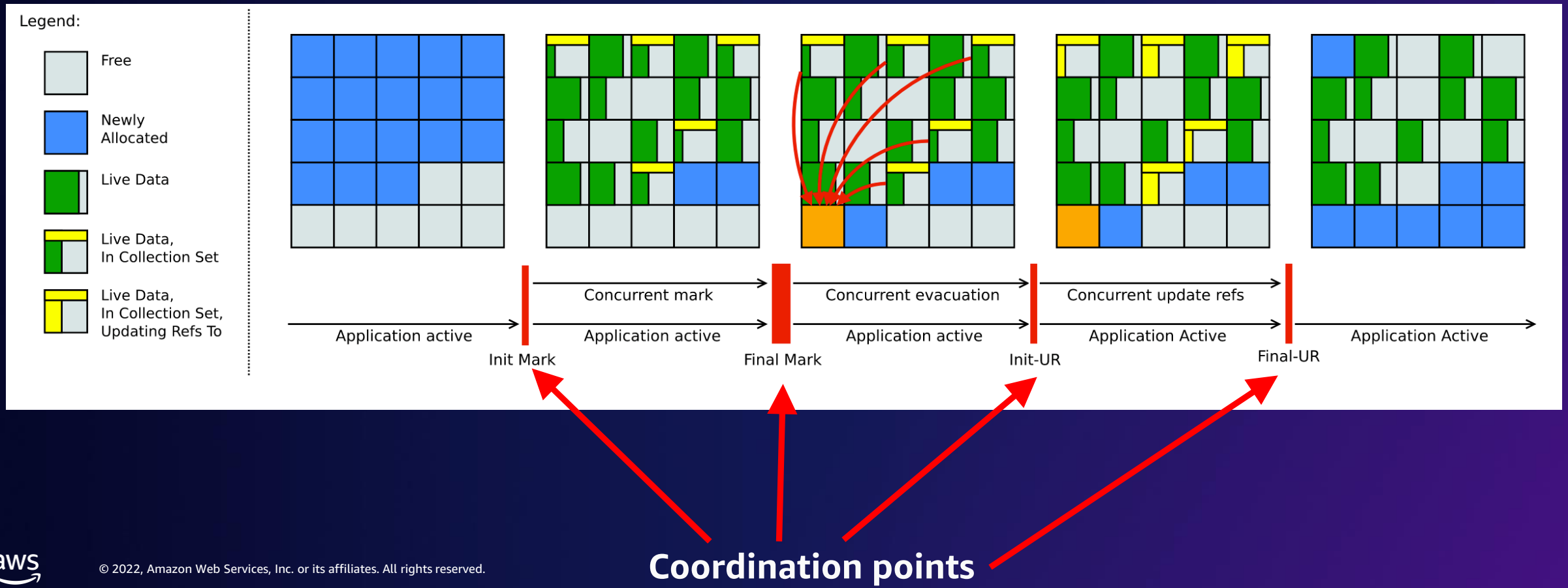
- Traditional (serial, parallel) GC stops the world for seconds or minutes at a time to collect all young or full
- Mostly concurrent (CMS, G1) GCs allow Java threads to run while GC is marking live memory but stop the world for less frequent compaction efforts
- CMS is deprecated in JDK9, eliminated in JDK14

Pauses in G1 GC

- Periodically compacts the entire young-generation (limits size of young-generation to satisfy – XX:MaxGCPauseMillis=200)
- Following rare concurrent marking, some number of subsequent young-generation collections also compact some number of old-gen heap regions (so-called mixed evacuations)
- A stop-the-world full GC is required if:
  - We encounter an evacuation failure, or
  - We encounter an allocation failure during concurrent marking

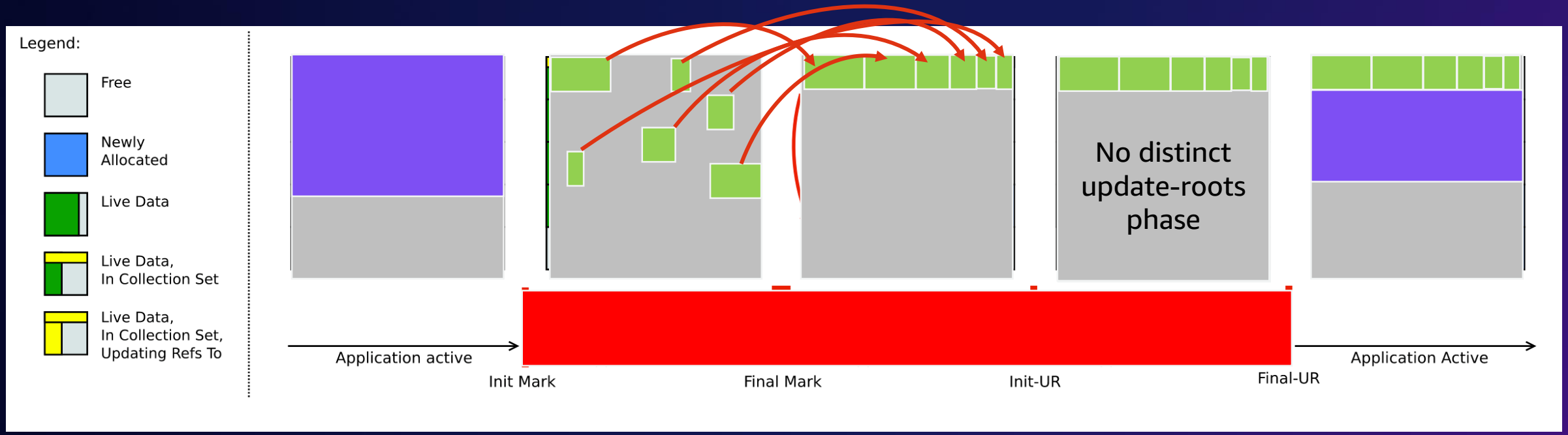
Pauses in Shenandoah GC

We describe Shenandoah GC as pause-less because there is no stop-the-world phase (except brief coordination points)



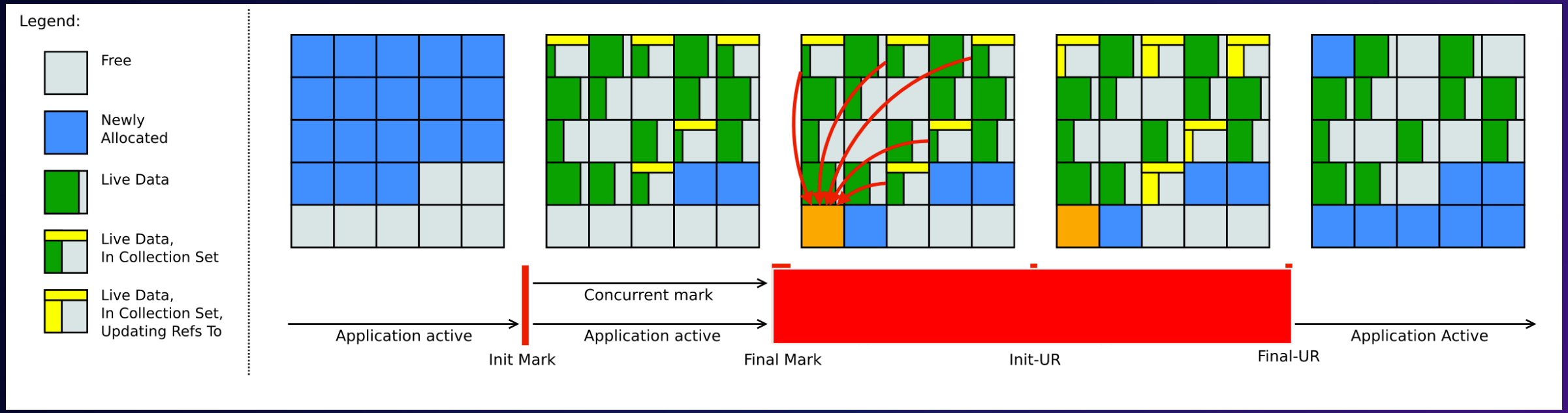
Shenandoah GC compared to parallel GC

- Application stalled throughout mark, evacuate, update
- Higher application throughput because no coordination overhead



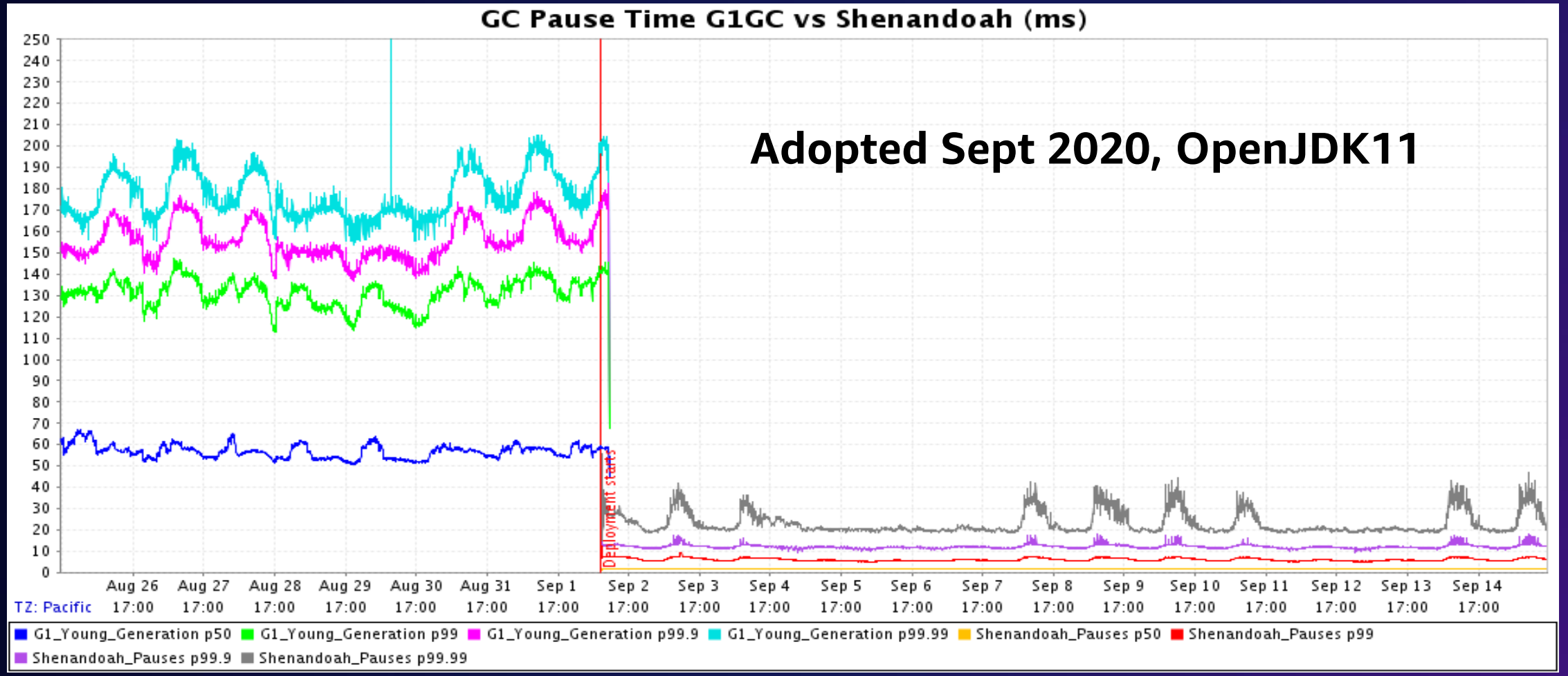
Shenandoah GC compared to G1 GC

Global marking is concurrent, but evacuation and updating is stop-the-world

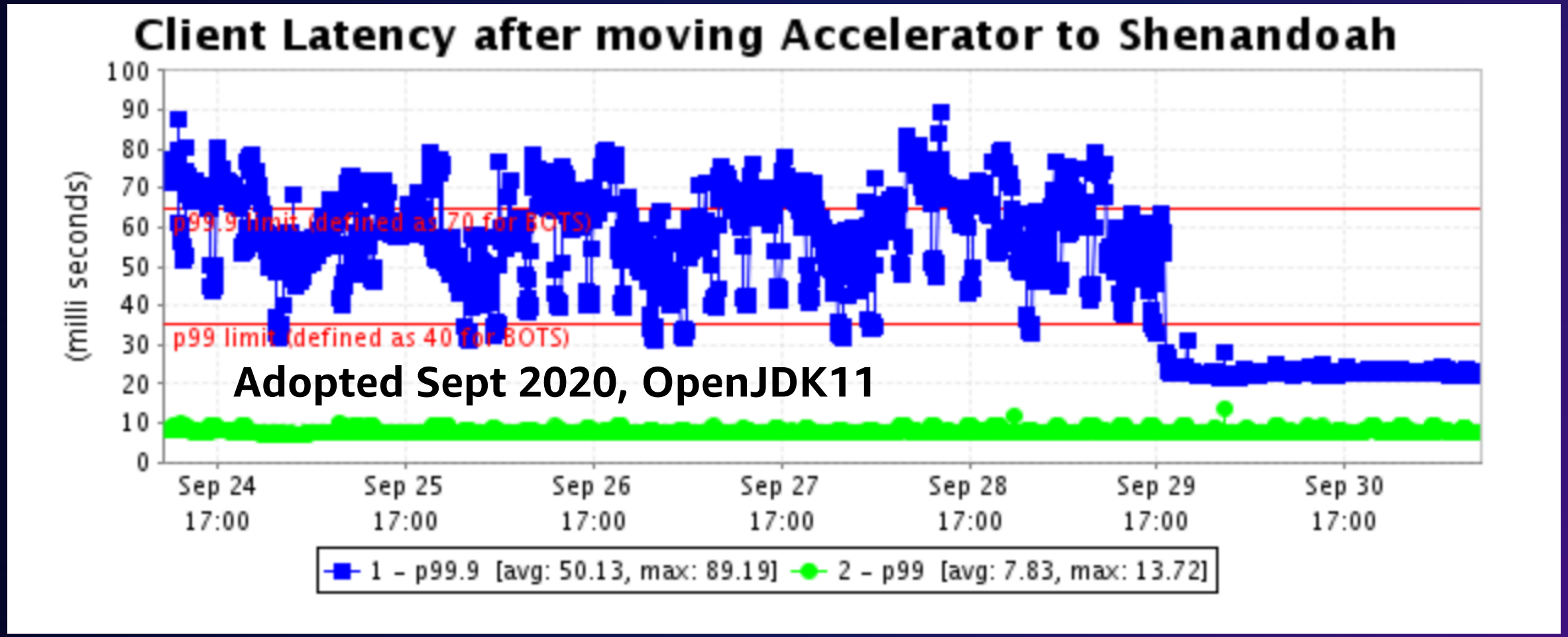


200 ms typical bound on this pause

AWS service adoption of Shenandoah



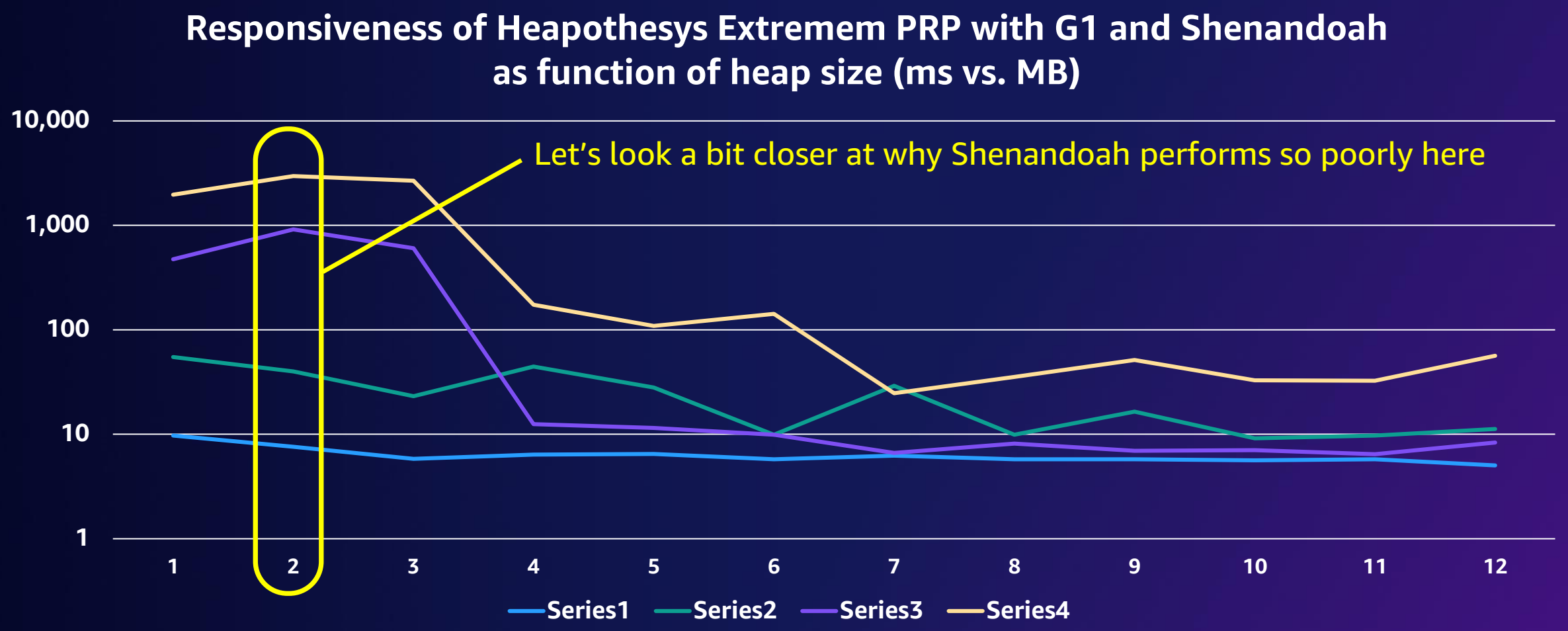
Another AWS service adoption of Shenandoah



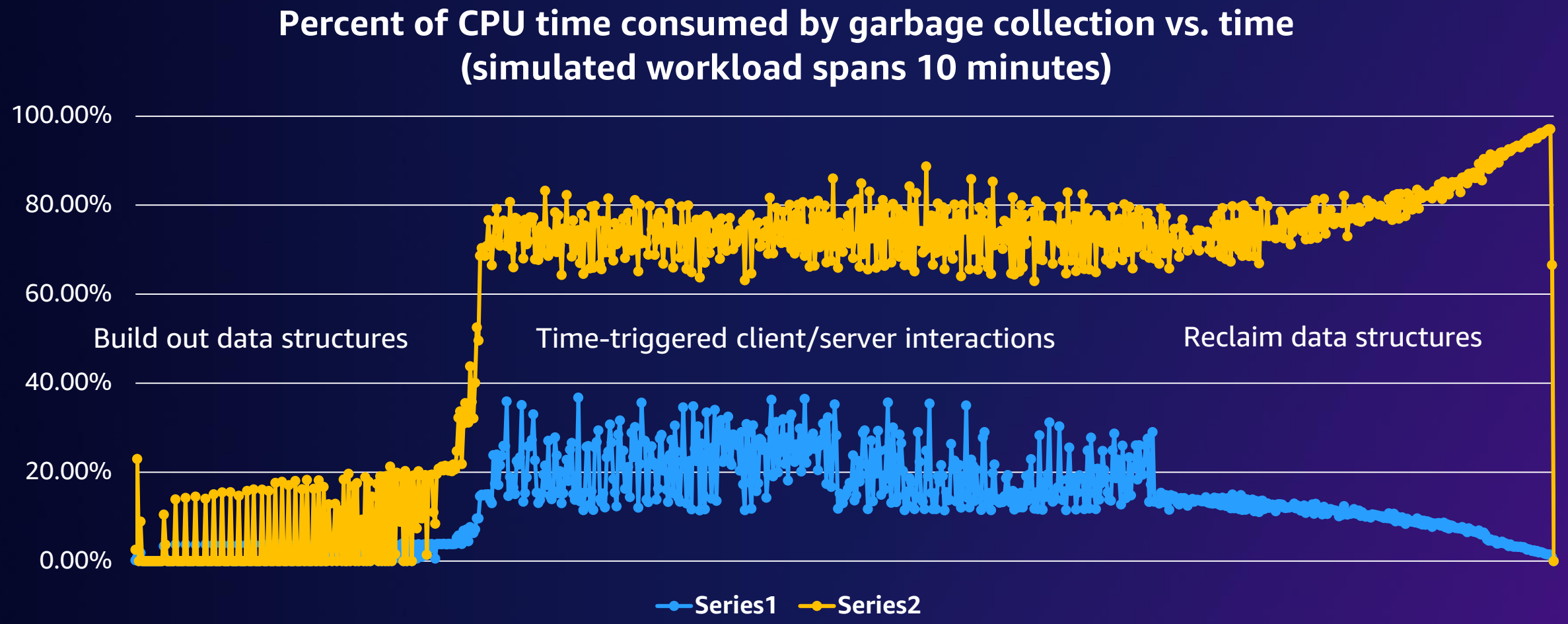
Additional pauses in Shenandoah GC

- A stop-the-world degenerated GC occurs when concurrent GC fails:
  - Due to allocation or evacuation failure (application allocates faster than GC replenishes)
  - Degenerated GC stops the world to finish what concurrent GC started
- A stop-the-world full GC occurs if too many back-to-back degenerated GCs fail to resolve problem
  - Reclaims all garbage and compacts even the humongous regions
  - Often runs in less time than a degenerated GC even though it is doing more work
- Planned improvements to generational Shenandoah
  - Use pacing and heuristics to avoid out-of-memory triggers for degenerated GC
  - Implement concurrent compaction of humongous regions to avoid need for full GC

Strenuous workload: Shenandoah vs. G1 GC



768 MB heap: 381 degenerated GC of 3,763 GC



Shenandoah GC strengths

- Concurrent Shenandoah effectively bounds GC pauses to less than 10 ms when lightly loaded
  - Low allocation rates (e.g., 512 MB/s or lower) even with high (e.g., 90%) memory utilization
  - High allocation rates (e.g., 4 GB/s or higher) if memory utilization is low (e.g., below 25%)
- This motivates generational mode of Shenandoah
  - Young generation has high allocation rate and very low memory utilization
  - Old generation has very low allocation (promotion) rate and high memory utilization

The generational hypothesis

- Typical Java applications allocate lots of very short-lived objects
- Most objects die young
  - The lifetime of a single method
  - Or the lifetime of a single client interaction
- Smaller percentage of objects live much longer
  - Indexes and databases used by all transactions
  - Cached results used by multiple transactions
- Generational GC focuses efforts on recently allocated objects
  - This is where we reclaim the largest amount of garbage with the lowest effort

Adding generations to Shenandoah

- Young-gen and old-gen marking run concurrently (with application and with each other)
- Young-gen typically has higher allocation rate
  - Replenish free pool by running young-gen collections more frequently at higher priority than old-gen collections
- The start of an old-gen collection piggybacks on the root scan performed by a young-gen collection
  - Following root scan, old-gen threads mark reachable old-gen objects and young-gen threads mark reachable young-gen objects
  - Young-gen marking runs at higher priority than old-gen marking because it is most urgent to replenish the allocation pool

Adding generations to Shenandoah

- Young-gen marking is followed by evacuation and updating references, all running at higher priority than old-gen marking
- When old-gen marking completes, we assemble a list of old-gen collection set candidates
  - During subsequent young-gen collections, each evacuation phase includes a subset of the old-gen collection set candidates in its collection set; these are mixed evacuations
  - After all collection set candidates have been processed, we can begin another concurrent old-gen marking effort
- Shenandoah mixed evacuations resemble G1 mixed evacuations
  - But Shenandoah evacuations are concurrent; G1 evacuations are stop-the-world
  - Shenandoah limits size of collection set in order to sustain pace at which allocation pool is replenished
  - G1 limits size of collection set in order to honor MaxGCPauseMillis

Qualifying use cases

- Does your service value consistent response time latencies below 100 ms?
  - If not, you might be better served by G1 or parallel GC
- Does your service require a combination of short-lived and long-lived data with a high allocation rate for objects that die young?
  - If not, you might be better served (or served equally well) by traditional Shenandoah
- Does your service value frugal use of memory and CPU cores?
  - If not, you may be able to configure traditional Shenandoah to satisfy your target transaction rates and response-time latencies

Generational Shenandoah target audience

- Have high allocation rates
- Maintain nontrivial amounts of long-lived data
- Value consistent client response times below 100 ms
- Value efficient use of memory
- Value efficient use of core processors
- Value efficient use of electricity for power and cooling

Getting started with generational Shenandoah

- Development source: https://github.com/openjdk/shenandoah
- Experimental OpenJDK executable preview releases: https://github.com/corretto/corretto-17/tree/generational-shenandoah

Generational Shenandoah is under development

- Ongoing efforts to:
  - Improve performance
  - Reduce variation in timeliness
  - Improve ease of use and reliability (automate self-configuration and adaptive behavior)
- Early experimentation by users helps us identify where further work is required
- Manual static configuration is currently required to make the best use of this technology

Service memory needs

- Your service memory needs are approximated by:
  - Allocation rate
  - Total live memory footprint
  - How much of the live memory is ephemeral vs. enduring?
  - What is the maximum lifetime of the ephemeral memory (e.g., transaction completion time)
  - What is the rate of churn within “enduring memory”? (e.g., cached values become obsolete)
- We are working on techniques to automatically detect these characteristics of a JVM workload
  - Until this is completed, you’ll have to approximate and configure manually

Understanding your memory needs

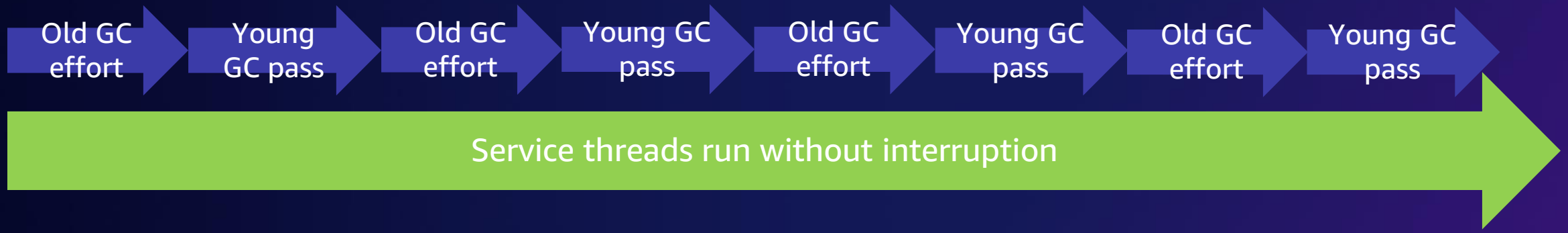
- Analyze the GC logs when running your service with other GC
- ParallelGC
  - From consecutive young-gen collection reports, you can determine allocation rate, ephemeral live-memory needs, and promotion rate
  - From less common full GC, you can determine total live-memory usage
- G1 GC
  - Harder to know from G1 GC log what your true memory needs are because G1 GC is intentionally lazy
  - You can still obtain allocation rates, the size of young-gen chosen by G1 GC, and promotion rates
- Generational Shenandoah
  - Logs report allocation rates, ephemeral live-memory needs, and promotion rates
  - From less common full GC, you can determine total live-memory usage

Concurrent GC pacing

- With concurrent GC, there is a race between the GC and the application service threads
  - The application continues to allocate, while …
  - The GC tries to replenish the allocation pool before it becomes exhausted
- Floating garbage
  - GC only promises to find the garbage that existed at the moment GC begins
    - Anything that becomes garbage following the start of GC may not be reclaimed (will not be reclaimed if concurrent GC uses a SATB barrier)
    - Anything allocated during the current GC pass is considered live until start of next GC pass
    - Floating garbage is larger when concurrent GC runs longer

Well-tuned generational Shenandoah

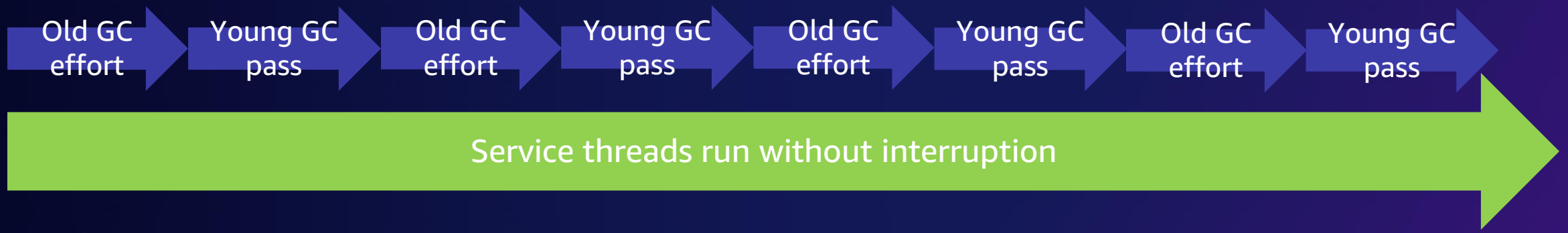
- GC is running “continually” in the background



- Old GC effort requires more total time because typical workload has much more live memory in old-generation
- Young GC is higher priority because it is urgent to refresh the allocation pool
- Old GC effort is frequently interrupted by young GC passes

Well-tuned generational Shenandoah

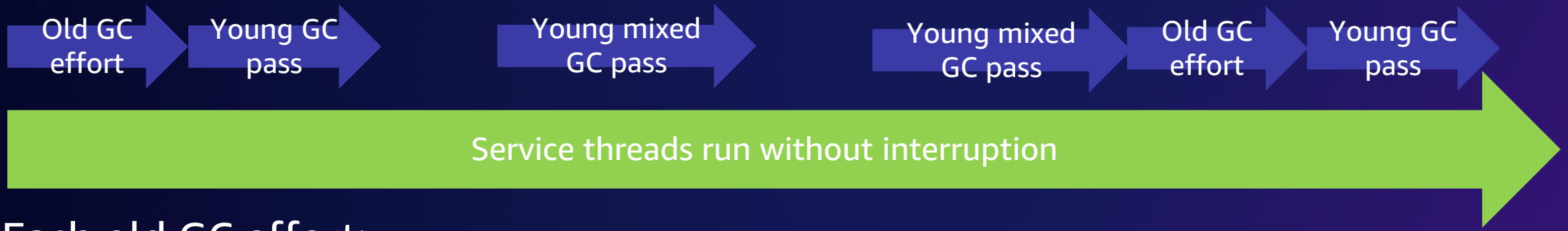
- GC is running “continually” in the background



- Each young GC pass:
  - Reclaims dead objects allocated during preceding young GC pass and old GC effort
  - Preserves as live the ephemeral live plus the floating garbage allocated during this young GC pass
  - Promotes some small percentage of the preserved memory into old-gen

Well-tuned generational Shenandoah

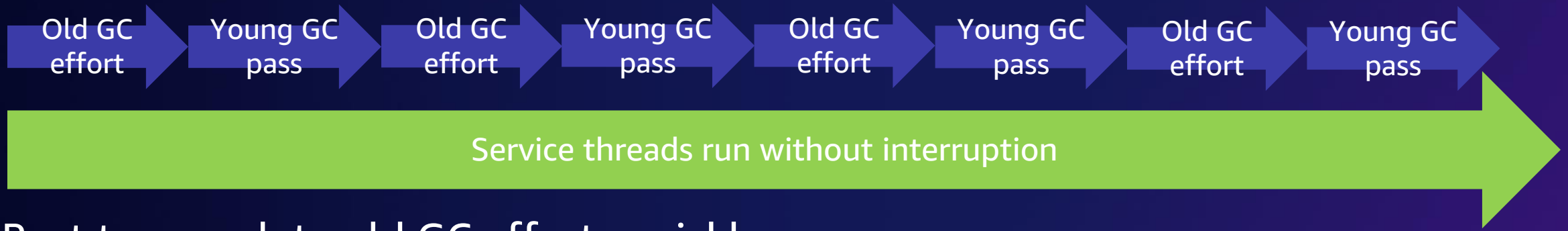
- GC is running continually in the background



- Each old GC effort:
  - Works to identify live memory residing in old generation
  - After old-gen memory is fully marked, constructs old collection set candidates and feeds these to subsequent mixed evacuations
  - A young mixed effort takes more time than a traditional young effort
  - Old GC efforts do not resume until mixed evacuations have processed all old collection set candidates

Well-tuned generational Shenandoah

- GC is running continually in the background



- Best to complete old GC efforts quickly:
  - Not so much urgency to replenish old allocation pool because promotion rate is low
  - But, old-gen can hold lots of old floating garbage if we take minutes to complete
- Both young- and old-gen efforts may not start immediately following completion of previous GC effort (to improve GC efficiency and yield CPU)

Configuring generational Shenandoah

- Step 1: Make sure old-gen is large enough to hold your enduring memory (caches, databases, etc.)
  - Add safety buffer to hold objects that were accidentally promoted
  - Add a working buffer to support churn and floating garbage within enduring memory
  - Example: total heap size 32g, old-gen size 8g: -Xms32g –Xmx32g –XX:NewSize=24g
- Guidance:
  - If you configure old-gen too large, you force too frequent young-gen collections
  - If configured too small, objects that have reached tenure age will not be promoted, requiring frequent young-gen GC to repeatedly process enduring data
  - Advice: Don’t overdue analysis – tune empirically, using theory to guide understanding of tradeoffs

Configuring generational Shenandoah

- Step 2: Configure young-gen size
  - Make young-gen large enough to hold live ephemeral memory plus the floating garbage created during young-gen pass, plus a working buffer to hold allocations that accumulate during old GC efforts
- Guidance:
  - Consult GC log or metrics to figure out how long it takes to perform concurrent young-gen
  - Adjusting the number of concurrent GC threads will change time required to perform concurrent young collection
  - Larger young-gen size lets GC run less frequently, reclaiming more garbage with same effort
  - Smaller young-gen size is more frugal: smaller host memory, lower power, and cooling bills

Configuring generational Shenandoah

- Step 3: Configure promotion behavior
  - Set the tenure age so as to promote objects that have lived longer than the “ephemeral age” for your application:
    - -XX:InitialTenuringThreshold=7
    - -XX:ShenandoahAgingCyclePeriod=1
- Guidance:
  - Every application is different
  - Judge based on knowledge of configured young-gen frequency and expected maximum duration of ”transactions”
  - Data that spans multiple transactions probably belongs in old generation

Other configuration options

- -XX:ThreadPriorityPolicy=0 –XX:+UseThreadPriorities –XX:ConcGCThreads=N
  - Denotes that N GC threads get higher priority than Java threads
- Watch for degenerated and full GC events in GC logs
  - If you see these, your system may be underprovisioned or misconfigured
  - Or there may be more work for the generational Shenandoah developers to do to reduce dependency on these
  - Share your experiences with Amazon Corretto team and we will provide guidance (for now)

Coming attractions

- We’re working to:
  - Improve ShenandoahHeuristics to more effectively trigger the starts of GC efforts so as to avoid allocation failures during GC
  - Improve ShenandoahPacing to make it work better with generational mode
    - This boosts GC priority and deprioritizes Java threads in order to avoid allocation failures
  - Improve overall efficiency so that GC is less likely to lose the race against allocating service threads
  - Auto-tune and adapt configuration of generational Shenandoah

Call to action

- If you run a service that needs efficient memory management and values the absence of long execution pauses:
  - Give generational Shenandoah a test drive
  - Share your findings and workloads with the Corretto team
    - Open a ticket at https://github.com/corretto/corretto-jdk/issues
    - Help us understand what you are looking for and what you are finding
- Your early efforts to evaluate generational Shenandoah will help us make sure the technology best serves your needs

Learn in-demand AWS Cloud skills

AWS Skill Builder

Access **500+ free** digital courses and Learning Plans

Explore resources with a variety of skill levels and **16+** languages to meet your learning needs

Deepen your skills with digital learning on demand

Train now

AWS Certifications

Earn an industry-recognized credential

Receive Foundational, Associate, Professional, and Specialty certifications

Join the **AWS Certified community** and get exclusive benefits

Access **new** exam guides

Thank you!

Kelvin Nilsen

@kdnilsen on Twitter