Comparative Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Service | CPU-only Reduction Latency Trend | Memory-only Reduction Latency Trend | Combined CPU & Memory Reduction Latency Trend |
| Prime Verifier (Java) | The latency increases gradually but clearly with each CPU limit reduction. The pattern is strongly correlated with CPU capacity, suggesting this service is CPU-bound. Even small reductions lead to noticeable latency shifts. | Shows mild to moderate latency increase depending on the stage of reduction. The memory sensitivity is lower unless limits fall near critical GC or heap thresholds. | Latency becomes highly sensitive, reacting quickly to combined drops. Due to JVM being stressed on both GC and thread management. Latency spikes are early and sharper than either axis alone. |
| Echo (Go) | Latency remains nearly flat despite CPU limit reductions. This implies the echo service is lightweight, well-optimized, & asynchronous enough not to be throttled by CPU. | No major latency effects observed. Even significant memory cuts do not affect response times. This reflects Go’s minimal memory footprint and low heap churn. | The service shows resilience even with both limits reduced. Due to minimal computation and no significant memory allocation pressure. Latency is very stable. |
| Hash Generator (Java) | There is a rapid and severe latency spike following the first few CPU limit reductions. This implies the service is CPU-intensive, especially under moderate to high load. | Initially, the latency is relatively unaffected. However, as memory limits continue to drop, the system enters a volatile phase where GC or heap constraints cause spikes. | With both CPU and memory reduced, latency increases sharply and irregularly, indicating a highly sensitive and unstable execution pattern. JVM overhead becomes dominant. |
| Password Generator (Java) | Latency increases with each CPU limit drop, in a predictable spike pattern. Each spike appears after a specific threshold, indicating step-wise performance degradation. | Shows spiky latency patterns followed by brief plateaus. Memory limits affect performance in bursts, related to specific allocation sizes in password logic. | Surprisingly, combined reductions result in sudden latency dips after steep drops, due to runtime adaptation (smaller GC loads, tighter memory reuse). |

Detailed Interpretation

**Prime Verifier**

* CPU-bound workload involving intense numeric computation.
* JVM performance is highly dependent on CPU availability; as such, latency increases in a predictable way when CPU is reduced.
* Memory alone does not impact latency as strongly, unless heap pressure or GC tuning thresholds are breached.
* When both CPU and memory are reduced, GC tuning, CPU throttling, and scheduling overheads compound, creating a nonlinear latency jump.

**Echo**

* Echo is minimalistic and stateless, with minimal computation & memory allocation.
* Neither CPU nor memory reductions significantly impact it - especially under a constant, low-to-moderate load.
* Demonstrates ideal resilience, potentially due to Go’s efficient runtime, lightweight goroutines, and absence of GC stalls common in JVM.

**Hash Generator**

* Hashing is CPU-intensive, especially in Java, where object creation and cryptographic operations also use heap memory.
* CPU reduction causes immediate performance degradation due to slow hashing.
* Memory cuts initially don’t hurt much, but as GC overhead rises, latency becomes volatile.
* In the combined case, latency sharply increases, and the system becomes unstable and noisy - classic JVM under dual pressure.

**Password Generator**

* Uses random number generation, character manipulation, and temporary objects.
* CPU reductions cause regular, predictable latency spikes - tied to contention and scheduling delays.
* Memory reductions cause bursty latency, reflecting interactions with object pooling or garbage collection.
* Combined reductions cause an unexpected dip in latency after steep reduction, due to fewer objects, lower GC footprint, & runtime optimizations kicking in.

Final Observations

* CPU limits consistently influence latency more significantly than memory across all services - except in very tight memory conditions.
* Go-based services are inherently more resilient due to lightweight runtime and better memory efficiency.
* Java services show complex, nonlinear behaviors due to GC, JIT compilation, and thread scheduling - all sensitive to both CPU and memory.
* Combined resource reductions do not always compound latency - in some cases, they trigger efficient adaptations, reducing overheads unexpectedly.