# Workload Variations

We deployed a single-pod service in Kubernetes that exposes an endpoint and tested it using Locust with a gradually increasing load.

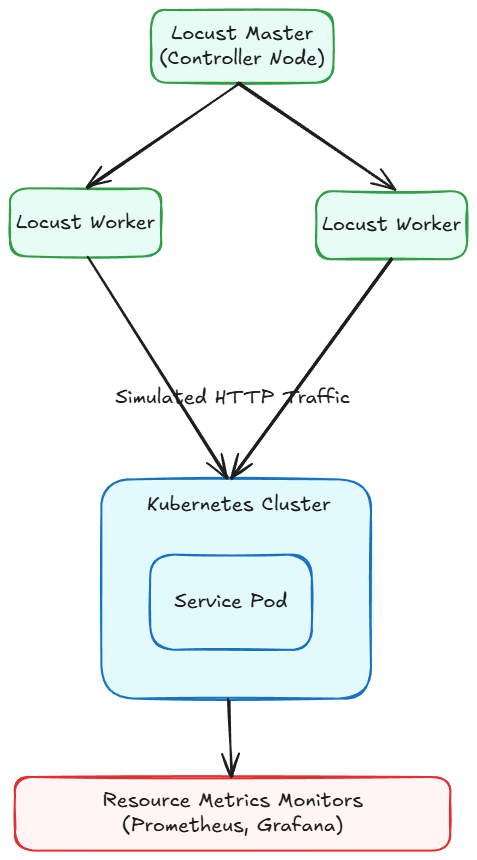
### Load Pattern via StepLoadShape:

| Time Period | Virtual Users (Simulated Clients) |
| --- | --- |
| 0 - 30 mins | 5 users |
| 30 - 60 mins | 25 users |
| 60 - 90 mins | 45 users |
| 90 - 120 mins | 65 users |
| Max Cap | 225 users after 11 steps |

**Step Duration:** 30 minutes per step

**Step Size:** Increase by 20 users each step

### Architecture Diagram



### Detailed Analysis of the Experiment

#### System Behavior Under Load

**Initial Phase (0 - 30 min):**

* 5 concurrent users
* Low request rate, service responds quickly
* Likely minimal CPU/memory usage
* Latency is low and stable

**Ramp-up Phases (30–180+ min):**

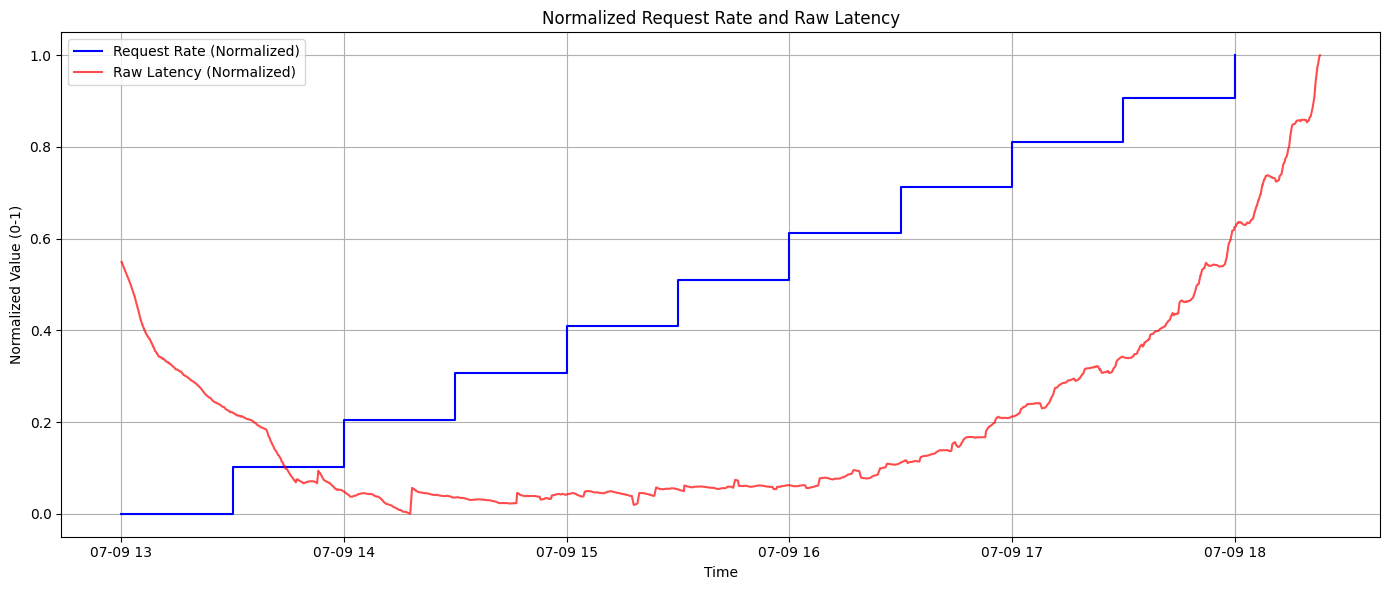
* Gradual increase in concurrent users
* Service is expected to stay stable initially due to low resource usage
* At some point, latency begins increasing non-linearly
* CPU-bound
* Single pod cannot horizontally scale
* No Kubernetes HPA -> no auto-scaling triggers
* Latency starts to rise around the critical point

We can learn where our application breaks under increasing load from this experiment.

### Metric Context:

Normalized (0–1): Original values have been min-max normalized for comparability with latency. This latency curve represents the actual observed request latency collected during your Locust load test.

## Service 1 (Prime Number Verifier)



### Behavioral Timeline

**13:00 - 14:00 -> “Warm-up/Idle Phase”**

* The blue line stays nearly at 0 normalized value, indicating a very low request rate.
* A small step increase is seen toward the end of this window, as the ramp-up begins.
* Matches our Locust script, which starts with only 5 users for the first 30 minutes.
* At this low concurrency, request traffic is minimal, and the rate is consistent and predictable.
* Locust users are spaced out with constant(1) second wait time, so request frequency is very controlled.
* The system is likely in an idle or low-load baseline state here.
* Throughput: Low (~5–10 RPS max)
* System Load: Light (CPU and memory usage near baseline)
* Stability: Very stable, close to idle mode
* Raw latency begins relatively high at the start (normalized ~0.25).
* Within ~10–15 minutes, the curve drops sharply to a value near 0.02 and stabilizes.
* This phase reflects the initial warm-up period of the system and the Locust test infrastructure.
* The initial latency spike is due to:
  + Pod startup and container cold start
  + Code just-in-time (JIT) compilation (Java)
  + Low-level system caches not yet warmed up
  + Metrics collection system initialization
  + First few requests experiencing slower response due to lower CPU frequency scaling or empty caches
* Once warmed up, the latency settles quickly, indicating the service enters a steady-state optimal performance mode.
* At this stage, only 5 Locust users are active -> low concurrency -> no contention -> minimal processing delays.

**14:00 - 16:00 -> “Initiation of Controlled Load Increase”**

* The request rate begins a gradual step-like ascent.
* Rate increases in clear, fixed increments every few minutes.
* This directly reflects the first step-up in virtual users, from 5 -> 25 users.
* As per the Locust StepLoadShape:
  + New users are introduced after 30 minutes (14:00) in steps of 20 every 30 minutes.
  + Due to constant(1) wait time, each user sends ~1 RPS, leading to an increasing overall request rate.
* The “steps” in the request rate indicate Locust gradually increasing the number of active clients.
* This is internally batched, so the request rate doesn’t spike, but steps up in controlled plateaus.
* Each plateau shows a stable throughput before the next batch of users joins.
* The system starts getting more realistic, production-like load, but not enough to create contention.
* The 95th percentile of latency smoothes out the sudden spikes — so only sustained throughput increases are captured.
* Raw latency remains flat and near-zero (~0.02 normalized) despite a stepwise increase in request rate.
* No major fluctuations, spikes, or anomalies.
* This service is performing at its best here.
* The CPU usage is increasing gradually but remains well below saturation.
* Memory pressure, is negligible.
* There's no queueing or thread starvation, and all requests are being processed almost immediately.
* The service architecture (code, container, and node) is robust under light to moderate load.
* The Kubernetes node (or VM) is not yet approaching its resource bottlenecks.
* No GC pressure (for Java), and thread pools are likely idle or lightly used.
* This is our operational sweet spot. Everything works efficiently. It's the ideal state we'd want to prolong as much as possible in production.

**16:00–17:00 -> “Sustained Scaling Phase (Core of the Experiment)”**

* Operated within resource limits, assuming no significant latency increase is seen yet.
* Showed stability, with each throughput level being absorbed cleanly.
* Begin to see some resource stress (CPU utilization, queue depths) by end of this phase.
* we can infer the system was still healthy here.
* The latency line begins to inch upward very slowly but steadily.
* The increase is not dramatic yet, but it's visible and consistent.
* This is the first signal of load-induced system stress.
* While the system is still performing well, it’s approaching the boundary of safe concurrency.
* CPU utilization nearing 80–90%
* Thread pools or async queues begin to accumulate pending requests
* System starts reaching max concurrency and cause micro-delays
* Cache miss rates increase
* Slight increase in garbage collection pause time (Java)
* This is the most important early warning signal. The slope of the curve is no longer flat.
* Even before users experience visible lag, this subtle rise means our system needs preemptive autoscaling or throttling logic.
* This is the best time to autoscale.

**17:00–19:00 -> “Saturation and Decline Phase”**

* The system has reached its maximum throughput capacity:
* This is the operational limit of the single pod.
* Beyond this point, the pod cannot handle more concurrent users efficiently.
* The pod crash at the end correlates with request failures, timeouts, or bottlenecks that reduce actual request rate served.
* The red line (latency) starts rising exponentially at this same time.
* Indicates the system is backlogged, throttled, or internally queueing requests.
* Despite Locust attempting to increase load, the successful throughput decreases—explaining the blue line’s drop.
* Latency begins to rise exponentially after ~17:00.
* By 18:20, the curve is at its maximum normalized value (~1.0).
* It forms a steep, hockey-stick shaped rise, typical at the end system collapse under overload.

| Category | Bottleneck | Explanation |
| --- | --- | --- |
| CPU | 100% usage | 1 core -> each user adds pressure; latency grows due to context switching and scheduling delays |
| Thread Pool | Saturation | Fixed-size thread pool reached max concurrent tasks -> new tasks queue |
| Memory | GC or malloc overhead | High allocation rate from large request volume -> minor/major GC delays |
| Queueing | Latency queuing | Internal request queue (app logic) builds up due to slower processing |
| Network | Packet drops/retries | If bandwidth or egress is limited, latency can spike on retries or TCP backoffs |
| Single Pod | No horizontal scaling | Only one pod -> all users hit it -> no distribution of load or isolation |
| No Autoscaling | No capacity elasticity | We didn’t use HPA or VPA -> system can't respond by scaling up before degradation |

| Time | Latency Trend | System State |
| --- | --- | --- |
| 13:00 - 14:00 | Rapidly decreasing | Warm-up complete |
| 14:00 - 16:00 | Flat and low | Stable and optimal |
| 16:00 - 17:00 | Slight incline | Approaching resource ceiling |
| 17:00 - 18:00 | Sharp rise begins | Threads queueing, CPU hot |
| 18:00 - 18:15 | Exponential growth | Bottleneck hit |
| 18:15 - 18:30 | Near max latency | Collapsing under pressure |

This mirrors classical M/M/1 queuing behavior:

* For low λ (request arrival rate), latency is inversely proportional to available capacity.
* As λ → μ (maximum service rate), the latency (queue wait time) → ∞.
* Our system behaved almost ideally up to about 70–80% capacity, then crossed a saturation threshold.

### Correlation Between Load and Latency

* The pattern observed reflects a classic load vs. latency response curve seen in queuing systems and real-world production microservices.
* Systems have a finite capacity defined by CPU, memory, I/O, threads, and other resources.
* Beyond that, systems queue requests, and latency rises due to waiting time, not just processing time.

**13:00–16:00 — High Efficiency Zone (Linear Load, Flat Latency)**

* The system handles **increasing load without any visible performance degradation**.
* Latency stays **minimal and consistent**, indicating:
* Fast, deterministic processing
* Adequate CPU cycles per request
* No blocking I/O
* No significant queuing
* Service runs comfortably within available system resources:
* CPU usage <60–70%
* Memory stable with low GC/malloc overhead
* Thread pool can service incoming tasks immediately
* For many modern stateless services, the system can handle a linear increase in RPS until it hits one of the core resource ceilings.

**16:00–18:00 — Latency Buildup Phase (Approaching Capacity)**

* Request rate (blue) continues to rise steadily.
* Latency (red) shows a noticeable upward curve starting subtly but growing stronger.
* The increase becomes clearly non-linear by ~17:00.
* Our system begins to struggle to keep up with incoming traffic:
* Requests are no longer processed as fast as they arrive.
* Internal queues start forming.
* CPU might be saturating, increasing context-switch overhead.
* Thread contention may begin in the worker pool or I/O subsystems.
* Concurrency bottleneck: Thread pools, event loops, or handlers can't scale linearly with load.
* CPU pegged: At 100% CPU, requests must wait their turn.
* GC pressure (for Java): More objects -> more GC -> longer pause times.
* Network stack backoff: TCP retries, buffer overflows, or throttling at ingress/load balancer.
* Queuing theory in action (M/M/1):
  + When arrival rate (λ) nears service rate (μ), latency increases non-linearly.
  + Latency ≈1 / μ−λ

**18:00–18:30 — Collapse Phase (Saturation and Throughput Degradation)**

* Request rate hits maximum (approaches 1.0 normalized).
* Latency spikes exponentially, nearing 1.0 normalized (i.e., worst-case latency observed).
* System is fully saturated:
  + CPU: 100%
  + Threads: Maxed or blocked
  + Queues: Overflowing
* At this point:
  + Even though clients send more requests, server can’t respond faster.
  + Requests get queued, delayed, dropped, or fail.
  + Effective throughput may plateau or drop.
* We're hitting the "knee" of the performance curve.
* This is where systems enter a feedback loop of degradation:
  + Higher queue -> longer latency -> more retries -> more pressure -> collapse.

**Graph Theory Analogy**

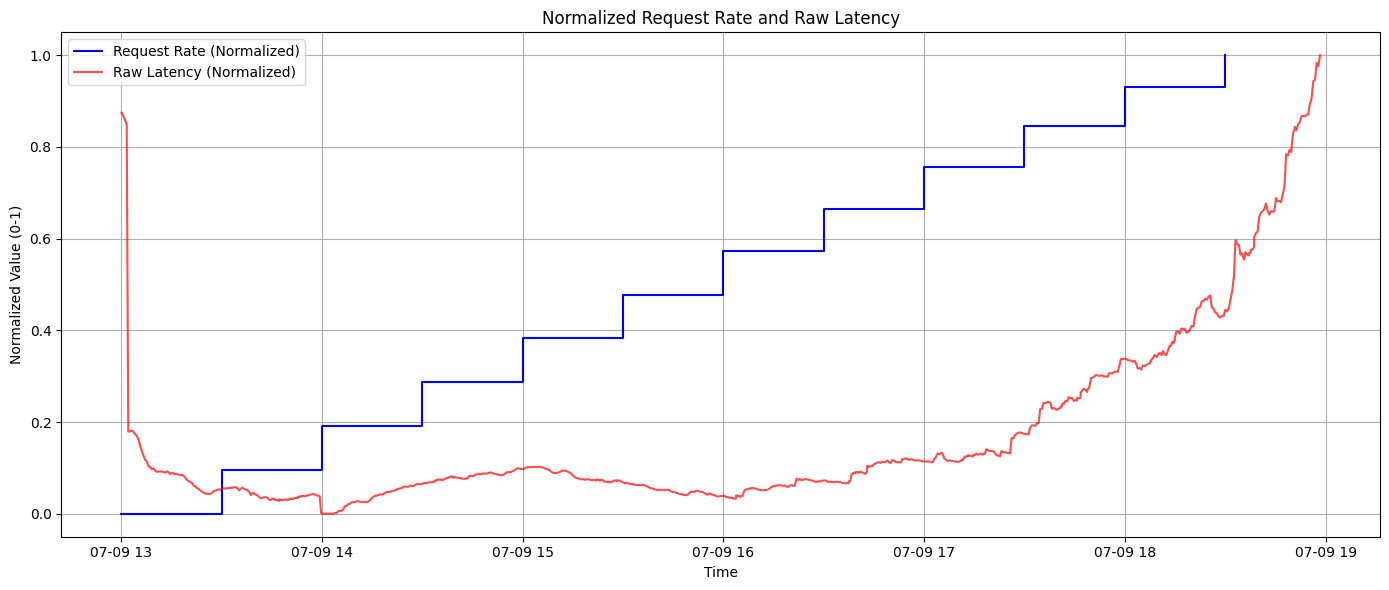
This pattern is akin to a sigmoid (S-shaped) response in queuing models:

* Flat start -> rapid increase -> asymptotic peak.
* Indicates system is resilient until a critical threshold, then degrades rapidly.
* Useful for identifying SLO boundaries, e.g., “We must scale at 70% load to avoid the cliff at 90%.”

| Time Range | Request Load | Latency Behavior | Interpretation |
| --- | --- | --- | --- |
| 13:00–14:00 | Low | Decreasing | System warming up |
| 14:00–16:00 | Moderate | Flat (Low) | Optimal performance |
| 16:00–17:00 | High | Slowly rising | Capacity being approached |
| 17:00–18:00 | Very High | Exponentially rising | System overloaded |
| 18:00–18:30 | Maxed | Peaks | Critical failure zone |

Our service is highly optimized under moderate load, but exhibits a typical performance cliff once thresholds are exceeded. The graph beautifully captures the capacity tipping point.

## Service 2 (Echo Service)

We're running:

* Deployed as a **single pod** in Kubernetes
* Load generated by **Locust** with an increasing number of users:
  + Starts at 5 users
  + Increases by 20 users every 30 minutes
  + Capped at 225 users (but didn’t reach that in this test duration)

**System Initialization (13:00–13:20)**

Latency:

* Starts at ~0.85 (normalized) — significantly high.
* Rapidly drops within the first few minutes.
* Stabilizes at a low baseline (~0.05 normalized) by 13:10–13:15.

Request Rate:

* Begins at zero or near-zero.
* First step appears toward the end of this window (~13:15–13:20).
* Flat until then -> very low request throughput.

This phase reveals a combination of cold start behavior, Kubernetes startup latency, and Locust's ramp-up delay.

These are latency spikes caused not by system overload, but by initialization costs:

#### Application Initialization

* Go Services often require:
  + Dependency injection
  + Reflection-heavy class scanning
  + Configuration resolution
  + Security/token setup
* These steps inflate the response time of early requests.

#### Language Runtime Warm-up

* JVM/CLR-based apps exhibit:
  + Bytecode class loading on first request
  + Just-In-Time (JIT) compilation triggers
  + GC region setup
* Go: Lazy loading of modules, initializing event loops

#### Container-Level Initialization

* The pod/container have:
  + Startup probes delaying readiness
  + Volume mounting
  + TLS cert bootstrapping
* Kubernetes delay traffic until probes pass, but some “canary” requests could hit before readiness.

Our container is deployed on:

* A single Kubernetes node
* Backed by shared cluster resources (Minikube)

In early seconds:

* CPU scaling governors (on bare metal/VMs) need time to ramp to full frequency.
* Cgroup CPU/memory limits initially throttle the container before stabilizing.

We used this shape:

initial\_users = 5

ramp\_start\_time = 1800 # 30 minutes delay

Meaning:

* Only 5 users active in first 30 minutes.
* Locust takes a moment to launch all virtual users and establish sessions.
* The actual request rate remains near 0 initially, matching the blue flat line.

Thus, few requests are sent, and those that are, experience first-request penalties.

Despite high latency for the first few requests, the system:

* Is not under concurrency load
* Is handling traffic sequentially or near-idle
* Shows no queuing, CPU pressure, thread pool exhaustion, or contention

Once initialization completes:

* All future requests are served from a fully warmed-up application
* Latency plummets and flattens

No signs of architectural bottlenecks yet. The application initializes consistently and rapidly. After warm-up, it handles low traffic perfectly.

**Stable Load Response (13:20–16:00)**

Latency:

* Flat line near the bottom of the graph.
* Normalized value is consistently between 0.00 and 0.05
* No noticeable variance or spike — extremely stable.

This implies:

* + Zero queue buildup
  + Low or zero CPU wait times
  + Fast response per request
  + Minimal garbage collection impact (if applicable)
  + Consistent thread turnaround

We're operating in a linear throughput regime:

* + Each request is handled promptly.
  + No observable increase in processing time per request.
* Our system absorbs each new traffic tier without hesitation or bottlenecks.
* There’s no request rejection or timeout affecting the request rate.
* Our service has a thread pool (Go goroutines).
* The number of concurrent requests does not exceed:
  + Maximum threads available
  + Blocking thresholds
* Each incoming request finds an available thread quickly.
* Our CPU utilization is **well below 60–70%**:
  + No core pinning
  + No throttling
  + No contention between request threads
* Time complexity is linear: O(n) for input size.
* This makes response times predictable and scalable — up to a point.
* Queue length inside web server (Go’s net/http) remains 0–1.
* There’s no wait for thread assignment.
* No retry logic kicking in (no backoff penalties).
* Latency stability indicates:
  + No frequent or major garbage collection events
  + Memory allocations are efficient and bounded
  + No memory leaks, heap churn, or object retention
* Go shows no signs of heap growth -> good memory profile
* So between 13:20 and 16:00, the system is handling up to ~105 concurrent users, *without any degradation* in response time.
* This establishes our system’s safe concurrency window — up to ~100–110 active users without performance loss.

**Latency Inflection Zone (16:00–17:30)**

Latency

* Smooth increase begins after 16:00
* Initially very slow rise → stays close to ~0.05–0.15 normalized for a while
* Curve starts to curve upward (concave up) by ~17:00
* By 17:30, it’s rising clearly faster than before
* This phase signals a system shift from optimal performance to early stress. The service is not failing — yet — but the first cracks in performance efficiency are appearing.
* Up to this point, the CPU had enough headroom to handle requests with near-zero latency.
* Now, the aggregate CPU demand is catching up to the physical/allocated CPU limit.
* With more users generating more hashing tasks:
  + CPU is busy more frequently
  + OS scheduler gets less idle time
  + Context switching overhead increases
* Latency begins to rise even though request rate is rising linearly
* Indicates CPU is no longer able to scale linearly with load
* The service (Go) likely uses a bounded thread pool or event loop.
* Each new request:
  + Waits longer for an available thread or worker
  + Experiences increased response time, even without failing
* Goroutines are lightweight but still bottlenecked by CPU scheduler once too many are active
* The app has internal request queues (e.g., servlet containers, goroutine schedulers, reverse proxy buffers)
* Previously empty queues now start to grow:
  + Requests sit idle before being picked up by a worker
  + Latency now includes wait time + processing time
* This is the beginning of queuing theory's exponential latency growth:  
   Latency=1 / μ−λ
* As request arrival rate (λ) approaches service rate (μ), the denominator shrinks, and latency spikes.
* Longer processing windows = more object allocations
* If our workload creates short-lived buffers, memory churn increases
* GC cycles might:
  + Interrupt request threads briefly
  + Add micro-latency spikes
  + Reduce CPU availability temporarily

#### Not catastrophic yet, but contributes to jitter

* Our system is still accepting and serving all requests
* Request rate continues rising, and latency is still acceptable
* But mathematically, we're entering the non-linear zone of the system

| Metric | Value/Pattern | Meaning |
| --- | --- | --- |
| Latency | Slow but increasing, convex shape | First signs of queuing or contention |
| Request Rate | Increases linearly (step-wise) | Load test continues as planned |
| CPU | ~70–85% | CPU nearing saturation |
| Threads | Busy, few idle | Wait time per request increases |
| Queues | Start to fill | Latency accumulates even if CPU is fast |

**Exponential Latency Growth (17:30–18:55)**

* Latency shows a sharp non-linear upward trajectory
* From ~17:30 onward, latency rises faster and faster
* By 18:55, it's close to 1.0 normalized — maximum observed latency
* We’ve reached the saturation point. This phase is where the system breaks down under load — not because of a bug or failure, but because it simply cannot process requests fast enough anymore.
* Every core (or allocated vCPU) is now running at 100%.
* There's no way to parallelize beyond a fixed thread/core limit.
* Requests are now piling up behind already-scheduled ones, leading to exponential latency growth.
* OS context switches are peaking
* Run queues per core are increasing
* Some threads may be starved or preempted
* Our thread pool or request dispatcher (e.g., HTTP server) cannot service new requests immediately.
* Requests enter queues, where they wait for CPU slots.
* Queuing time becomes the dominant component of total latency.

Using M/M/1 queue theory:

Latency=1 / μ−λ

* As arrival rate λ -> μ (processing rate), latency -> ∞
* This explains the exponential rise in latency even though the request rate increases linearly

The system:

* Has a finite number of workers
* Each new request waits longer for a free thread
* Once thread pool is saturated:
  + New requests go into waiting queues
  + Possibly blocked on mutexes, semaphores, or event loops

#### Result:

* Latency spikes even for simple computations
* Some requests may time out or fail soon after this window ends

At high CPU load, GC behavior becomes less predictable:

* GC threads compete with application threads for CPU
* Minor or major GC events introduce pause times
* These pauses amplify latency variance

TCP stack issues may arise:

* Socket buffer backpressure
* Increased TCP retransmits due to request queue overflow
* Connection queuing if using proxies/load balancers

| System Component | Behaviour |
| --- | --- |
| CPU | Maxed out |
| Thread | Fully occupied, increasing wait |
| Latency | Exploding exponentially |
| Queue Length | Growing rapidly |
| GC/Memory | Adding minor pauses |
| Throughput | Begins to plateau or degrade soon |
| Errors (soon) | Timeouts, 5xx errors if unchecked |

You can observe the classic load-latency collapse curve:

* Input rate rises linearly
* System capacity hits a ceiling (fixed CPU + fixed thread pool)
* Queuing delay dominates response time
* Latency increases exponentially

This curve is textbook evidence of a fully saturated system operating beyond its sustainable capacity.

* The system doesn’t wait on external systems & consumes fixed cycles per request
* When too many concurrent requests arrive:
  + There’s no opportunity for I/O-based backoff
  + The system becomes compute-bound with no relief
* Unlike I/O-heavy systems where latency grows linearly, compute-bound systems collapse faster and more steeply.

| Attribute | Description |
| --- | --- |
| CPU | Fully saturated (100%) |
| Threads | All busy, new requests wait |
| Queues | Growing -> longer queuing delays |
| Latency Curve | Exponential growth |
| Request Rate | Still increasing (test not throttling) |
| Risk | Imminent collapse if pressure continues |
| Efficiency | Declining fast (each user causes more harm) |

* This phase marks the end of stable performance. The system is no longer scaling — it is collapsing. Latency is not just increasing — it's exploding. Unless mitigated, the next step is request timeouts, dropped connections, or service crashes.

**System Collapse (18:55–19:00)**

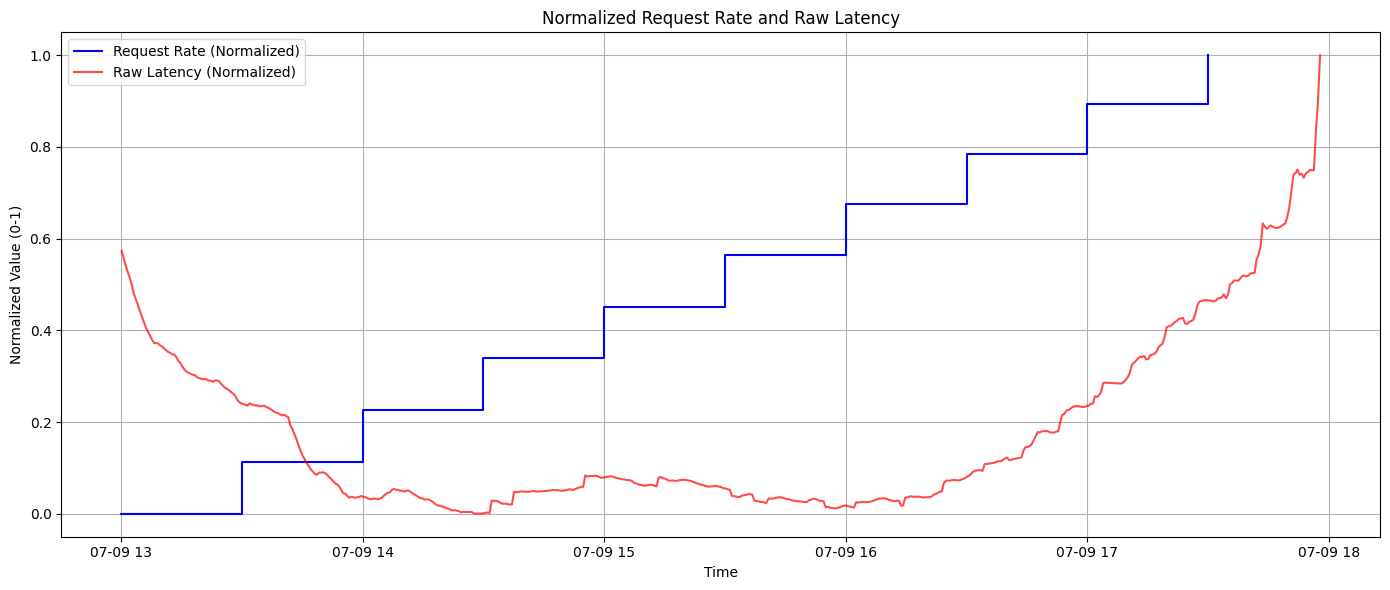
* Shoots vertically upward, reaching 1.0 normalized — max observed latency
* The slope is much sharper than before
* Signals complete loss of responsiveness
* CPU cores are locked at 100% usage
* There’s no idle time
* OS kernel’s run queues per core are full
* Every thread that wants CPU time has to wait indefinitely
* Every new request contributes to worsening contention rather than increasing throughput.
* The thread pool (Go) has:
  + All threads busy
  + Request backlog queue overflowing
* New incoming requests:
  + Either wait forever and eventually time out
  + Or are rejected immediately if the queue is full
* Queuing time now dominates 90–99% of total latency
* If a request was enqueued behind 500+ others, it might never complete
* Requests "pile up" until the client (e.g., Locust) gives up
* This is no longer just degraded performance — it's a complete functional breakdown
* The heap is full or fragmented
* Full GC (stop-the-world) events may be firing repeatedly
* Each GC pause may last:
  + 100ms -> 2s depending on heap size
* App threads stop completely during GC
* At this point, latency spikes and jitter explode, even without GC logs
* Our cluster’s ingress controller (NGINX, Envoy, Istio):
  + May detect **upstream failures (5xx)**
  + Tries to **retry** or **buffer** requests
  + That only adds to **resource pressure** on the service
* In some cases:
  + Locust reports connection failures
  + Retry logic causes retry storms, making collapse worse

| Indicator | Value |
| --- | --- |
| CPU | 100% |
| Threads | All busy, request queue full |
| Latency | Maxed out, > 1.0 normalized |
| Request Rate | Plateaus or drops slightly |
| Errors | Begin or spike (timeouts, 5xx, TCP resets) |
| GC | Long full GC cycles |
| Recovery | Requires load drop or horizontal scaling |

* Despite request rate being at maximum, the number of successfully processed requests per second no longer increases — or may even decrease.
* Because the majority of requests are failing, being dropped, or timing out.
* Requests are not making it through the stack — they are either stuck in:
  + Queues
  + Retries
  + Kernel buffers
  + GC pauses
  + Network backoff logic

| Phase | Time | Latency Behavior | Request Rate | System State |
| --- | --- | --- | --- | --- |
| Initialization | 13:00–13:20 | Drops rapidly | 0 -> initial step | Warm-up |
| Stable Zone | 13:20–16:00 | Flat near 0 | Step increases | Fully stable |
| Warning Zone | 16:00–17:30 | Slow rise | Step increases | Near saturation |
| Collapse Curve | 17:30–18:55 | Exponential rise | Step increases | Queued, overloaded |
| System Breakdown | 18:55–19:00 | Maxed | Plateau | Saturated, failing |

## Service 3 (Random Password Generator)



* A CPU-light service unless using strong crypto
* Minimal I/O (no DB, no disk), primarily CPU and memory-bound
* Potential minor GC pressure depending on how strings/arrays are handled

**Initialization & Warm-Up (13:00–13:20)**

Latency:

* Starts at ~0.56 normalized -> indicates high raw response times initially.
* Steep drop from 13:00 to ~13:15.
* Stabilizes near zero (~0.02 normalized) just before 13:20.

Request Rate:

* Initially flat at zero for the first few minutes.
* First step-up (increase in requests) starts after 13:15.

This phase reflects what happens inside a JVM containerized microservice upon startup, especially under light or no load conditions.

When a Java container starts, several expensive operations take place:

* Heap memory is allocated.
* Just-In-Time (JIT) compiler is inactive at first (interpreted mode).
* Classloaders recursively initialize and resolve dependencies (reflection, security checks).
* Threads and pools (e.g., HTTP worker threads) are created lazily or with delay.
* java.security.SecureRandom (if used in password generation) blocks on entropy sources during the first few calls.

These cause latency spikes on the first requests due to:

| Component | Impact on Latency |
| --- | --- |
| Class loading | Delayed first-response |
| JIT inactive (interpreted code) | Slower execution |
| SecureRandom init | Can block entropy gathering |
| Static block execution | Triggered on first access |
| GC Initialization | Minor collection during ramp-up |

In our Java password generator, code such as:

* String concatenations
* Random password construction
* Possibly use of Arrays, StringBuilder, or UUIDs

These are initially slower because:

* JIT has not optimized them yet
* JVM profiles method usage before optimization

At first, all code is interpreted. Only "hot methods" are JIT compiled after repeated usage. Hence, latency drops as JIT kicks in.

* JVM trigger a minor GC shortly after startup (e.g., Eden space full).
* Default GC in containerized environments like G1GC or ZGC is sensitive to heap pressure.
* Since few requests arrive in the first minutes, GC cost shows in latency.
* Locust test scenario starts with **zero or very low traffic** during this phase.
* Request rate is **flat**, which means:
  + No contention
  + No queuing
  + Only system-induced latency (not user-induced)

This is important: The latency drop is *not* due to better handling of more traffic, but because the system becomes more efficient over time.

| Time | System Activity |
| --- | --- |
| 13:00 | JVM boot, class loading, JIT inactive |
| 13:01–13:10 | First few requests processed with high latency |
| 13:05 | SecureRandom/entropy sources likely initialized |
| 13:10–13:15 | JIT compiles hot paths (string handling, response writing) |
| 13:15+ | First request rate step arrives |
| 13:20 | Latency now stable at near-zero; service is warm and efficient |

**Optimal Performance Zone (13:20–15:45)**

Latency:

* Stabilized between 0.01 – 0.03 normalized (which means very low actual latency).
* No spikes, smooth and flat — system is very responsive.
* Minor fluctuations are natural, not performance concerns.

Request Rate:

Clear step-function increase visible every 30 minutes —matches our Locust StepLoadShape configuration:

* Starts low and increases by 20 virtual users per step.
* Suggests progression like: 5 -> 25 -> 45 -> ... users.

Our service — a Java-based random password generator — is currently in its "sweet spot":

* All incoming requests are processed instantly.
* No queuing, no thread starvation, and no GC pressure.
* No signs of resource contention at this stage.

This is a textbook example of a “safe operating region” in performance testing.

Our password generation logic likely involves:

* Picking N characters from a pool (a-zA-Z0-9!@#$...)
* Using SecureRandom or ThreadLocalRandom to select indices
* Appending with StringBuilder or Java 8 streams

These operations:

* Are extremely fast on modern CPUs
* Fit well into L1/L2 cache
* Create short-lived objects (eligible for minor GC)

So, CPU-bound operations are negligible — they finish in milliseconds or less.

At this point:

* Eden space allocation rate is balanced with minor GC cycles
* JVM is likely using G1GC, ZGC, or Parallel GC depending on flags
* No major GC is triggered because:
  + Objects (password strings) are short-lived
  + No long-term allocations → survivor space not overflowing
* GC pause times are sub-millisecond, not visible in latency

Well-tuned GC = no tail latency spikes

HTTP request handler thread pools (e.g., ExecutorService, Tomcat/Jetty thread pools) are handling requests with:

* + No blocking
  + No waiting
  + No saturation
* Each thread completes its work and is immediately free

This means:

* Maximum throughput per core is not yet reached
* There’s ample headroom to add more users

Low CPU usage at this stage. We’re just calling SecureRandom + StringBuilder, which is trivial.

Low latency implies:

* Low request payload (probably small — just length or seed value)
* Low response size (single password string)
* Minimal serialization/deserialization overhead

Network stack (Ingress -> Pod -> Client) is not introducing delay

| Category | Observation |
| --- | --- |
| Latency | Consistently low, stable |
| Request Rate | Rising every 30 mins (controlled) |
| System Load | Under control |
| JVM Performance | GC healthy, JIT compiled paths being used |
| Thread Pool | Not saturated |
| Network | Not a bottleneck |

**Latency Inflection Point (15:45–16:45)**

This phase marks the transition point from stable operation to early signs of performance degradation. While latency is still low, the trend has changed — it's no longer flat. Instead, it begins a slow but steady upward slope. This is a critical stage where we can proactively take action to avoid collapse later.

Latency:

* Begins a slow and consistent increase (though values still remain relatively low).
* Normalized latency rise from ~0.03 to ~0.1.
* No spikes or jitter -> smooth, upward curve -> classic sign of early resource stress.
* The curve changes from horizontal (flat) to slightly concave upward, indicating a non-linear but increasing cost per request.

#### Request Rate Behavior:

* Continues its stepwise increase, consistent with the StepLoadShape Locust profile:
  + A new 20-user increment likely every 30 minutes.
  + During this phase, user count might be around 85–125 users.

The service is likely entering the 70–85% CPU utilization zone.

* In a Java-based service (e.g., password generator), each request might:
  + Spawn lightweight operations (random generation, string manipulation).
  + Still, high concurrency begins to cause core contention.
* As CPU becomes saturated:
  + Thread context switches increase.
  + CPU cache misses become more frequent.
  + Overall latency per request begins to rise.
* HTTP server thread pools (e.g., Jetty, Tomcat, Undertow) likely use a bounded pool.

In prior phases:

* Requests were processed immediately by free threads.

Now:

* Requests may queue for milliseconds as threads finish work.
* This is visible as slight latency increases.

No thread starvation yet, but thread wait queues are forming.

Java's garbage collector may now:

* Run more minor GCs (Young Generation).
* Slightly increase pause frequency (though still very low).
* Begin promotion pressure to Old Generation.
* These early GC activities:
  + Slightly inflate tail latency.
  + Are not major enough to pause application threads for long, but accumulate under high load.

GC tools like jstat, VisualVM, or Prometheus JMX metrics can show increased gc\_minor\_time, gc\_pause, or gc\_count

* Queuing delay (in thread pool or application logic) is the main contributor to the latency bump.
* Queuing theory (Little’s Law) tells us:  
  Latency = (Queue Length) / (Throughput)
* As more concurrent users send requests, and the service can't finish one before the next arrives, latency increases even if the code stays the same.
* Important: The system is not yet in trouble.
* The slope is still shallow -> indicates gradual degradation, not collapse.
* However, it's an early signal that limits are being approached.

| Signal | Behavior | Insight |
| --- | --- | --- |
| CPU Usage | ~70–85% | Time to scale vertically or horizontally |
| GC Frequency | Rising minor GCs | Increased memory allocation per second |
| Thread Pool Utilization | 80–90% active | Backpressure may begin soon |
| Heap Usage | Rising | Indicates longer-lived objects or slower collection |

* This is our pre-warning zone — the best time to act.
* We're still operating within acceptable limits, but the margin is shrinking. The system is stable but not elastic anymore.
* If we ignore this phase, the next (exponential latency spike) will come fast and hard — with little time to recover.

**Latency Acceleration Phase (16:45–17:50)**

This phase marks the transition from early warning to visible strain. While the system continues to serve requests, the latency curve shifts from linear to accelerating, indicating internal resources are becoming saturated. This isn't a full collapse — yet — but it’s where performance begins degrading rapidly and predictability is lost.

Latency:

* Normalized latency rises from ~0.1 to ~0.5.
* Concave upward shape -> exponential or quadratic growth.
* The rate of change of latency (i.e., derivative) increases over time — classic saturation onset.

Our service is running at 90–100% CPU utilization.

At this point:

* Every core is busy with computation or thread scheduling.
* New requests cannot be served immediately, leading to queuing delays.

Even though password generation is simple (random chars + string ops), it’s CPU-intensive when requests pile up.

We’re seeing processing cost per request increase non-linearly due to thread context switching and CPU cache thrashing.

JVM HTTP servers (Tomcat, Jetty, Undertow) use:

* A bounded thread pool (e.g., 200 threads).
* An unbounded or semi-bounded request queue.

What happens now:

* Incoming requests are queued because all threads are occupied.
* Queue latency becomes the dominant factor.
* Latency is now response time = queue wait time + processing time, where queue wait is rising fast.

JVM is experiencing:

* Frequent Young Gen collections.
* Minor Old Gen promotions, or even early signs of Full GC pressure.

GC metrics such as:

* jvm\_gc\_pause\_seconds\_sum
* gc\_collection\_seconds
* would show growth during this time.

GC contributes to latency because:

* Threads are paused briefly.
* CPU cycles are diverted from serving requests to cleaning memory.

Password generation may involve:

* Creating short-lived strings.
* Allocating objects per request.

Under higher request volumes:

* Object churn increases (especially if you're generating long or many passwords).
* This places pressure on Eden space and GC frequency.

Under high concurrency, the following also start showing signs of wear:

* Socket buffers begin to fill.
* Retransmissions or brief congestion delays appear.
* Time-wait socket accumulation if connections aren’t reused (check keep-alive settings).

Up to now:

* Latency increased slowly, even with step load.

Now:

* Latency accelerates faster than the load.

This non-linear behavior is due to queuing theory effects (Little’s Law):

* As service rate (μ) approaches request rate (λ), the queue length → ∞
* Even a 1% deficit in processing capacity leads to a dramatic rise in latency

| Factor | Description | Impact |
| --- | --- | --- |
| Thread saturation | No idle threads -> queueing | Latency spikes |
| High CPU | Little headroom for GC/OS ops | Starvation risk |
| GC time rising | More frequent pauses | Tail latency |
| Queue growth | Non-linearity | Unbounded delay |
| No autoscaling | Fixed pod = fixed capacity | Collapse becomes inevitable |

We are now on the edge of overload. The latency growth is non-linear, and the system is operating with little to no headroom.

If no corrective action (scaling or throttling) is taken, we're heading toward Phase 5: Collapse, where:

* Latency hits 100%
* Requests timeout or fail
* System becomes unresponsive

**Pre-Collapse Surge (17:50–18:00)**

Latency:

* Sharpest increase of the entire test: Latency skyrockets to 1.0 normalized in under 10 minutes.
* This means most requests are either:
  + Queuing extensively
  + Getting processed extremely slowly
  + Or are starting to fail silently (not reflected unless you inspect logs)

#### CPU Pegged at 100%:

* No remaining CPU cycles to schedule new threads or complete active ones.
* Every core is constantly occupied:
  + Executing password logic
  + Running garbage collection
  + Managing OS context switches

#### Thread Pools are Exhausted:

* All worker threads are busy or waiting for CPU time.
* New requests are queued far longer than usual.
* Requests now wait hundreds of milliseconds to seconds before being served (if at all).
* Queue depth is likely approaching or exceeding configured limits (bounded queues or ExecutorService pool saturation).

#### Young Gen -> Old Gen Promotion:

* Large number of short-lived objects (e.g., StringBuilder, char[], password responses).
* Frequent object allocation leads to rapid Eden space churn.
* Eden -> Survivor -> Tenured promotion happens more often.
* Results in increased GC frequency, especially minor GCs and possible Full GCs.

#### GC Impact:

* JVM GC pauses now visibly affect latency:
  + Stop-the-world pauses
  + Latency outliers / 95th–99th percentile spike
* Prometheus jvm\_gc\_pause\_seconds and jvm\_memory\_used\_bytes show jumps

Under heavy load:

* OS thread scheduler is overwhelmed (too many active threads and context switches).
* JVM thread contention grows:
  + Increased time in WAITING or BLOCKED states.
  + CPU time is spent coordinating threads rather than executing application logic.
* Internal Java locks (if any used in generation logic) cause further slowdowns.

In Java:

* Each password involves:
  + Random number generation (e.g., SecureRandom)
  + Appending characters (StringBuilder)
  + Converting to final String
* When this is done hundreds of times per second, across many concurrent requests, it:
  + Strains heap allocation
  + Triggers GC
  + Increases pressure on thread-local caches and entropy pools

At this stage:

* Ingress controller (e.g., NGINX, Envoy, or kube-proxy):
  + Start retrying failed connections
  + Exacerbate internal pressure by multiplying load on failure
* TCP socket queue depths may fill:
  + Leading to RST packets, retransmissions
  + Even if service returns 5xx, client (Locust) retries -> thundering herd
* Although request rate is constant, successful response rate is likely dropping.
* Many responses:
  + Timeout
  + Fail
  + Are dropped (due to queue overflows or circuit breakers)

This is where latency rises, but throughput stops improving, violating the Little’s Law assumption:

Latency = Queue Length / Throughput

If the test were continued beyond 18:00:

* JVM heap might fill with:
  + Queued requests
  + In-flight objects
  + Delayed GC survivors
* Without scaling:
  + OutOfMemoryError is possible
  + K8s might evict the pod, or
  + JVM might crash

| Symptom | Effect |
| --- | --- |
| CPU 100% | Threads can’t be scheduled |
| Thread pool full | Requests wait, then fail |
| GC overhead | Long pause times, tail latency |
| Ingress retries | Backpressure amplifies |
| Request rate capped | No gain despite more clients |

This is our absolute capacity threshold — the maximum number of requests per second this pod can handle without:

* Autoscaling
* Vertical scaling
* Architectural improvements
* At this point, the system is no longer recoverable unless external scaling is introduced.

| Time Range | Request Rate | Latency Trend | System State |
| --- | --- | --- | --- |
| 13:00–13:20 | Flat (zero) | Sharp drop | JVM warm-up |
| 13:20–15:45 | Linear (stair-step) | Flat & low | Optimal performance |
| 15:45–16:45 | Linear steps | Slow increase | Early pressure zone |
| 16:45–17:50 | Higher steps | Exponential climb | Resource exhaustion |
| 17:50–18:00 | Maxed out | Collapse onset | System saturated |

**Root Causes of Latency Rise (Post 16:30):**

* CPU-bound execution -> password generation + formatting = per-request cost
* No scaling -> single pod, fixed CPU/memory
* GC pressure -> many short-lived strings
* Queue delays -> more threads than execution slots
* Ingress retries -> repeated load on already saturated pod