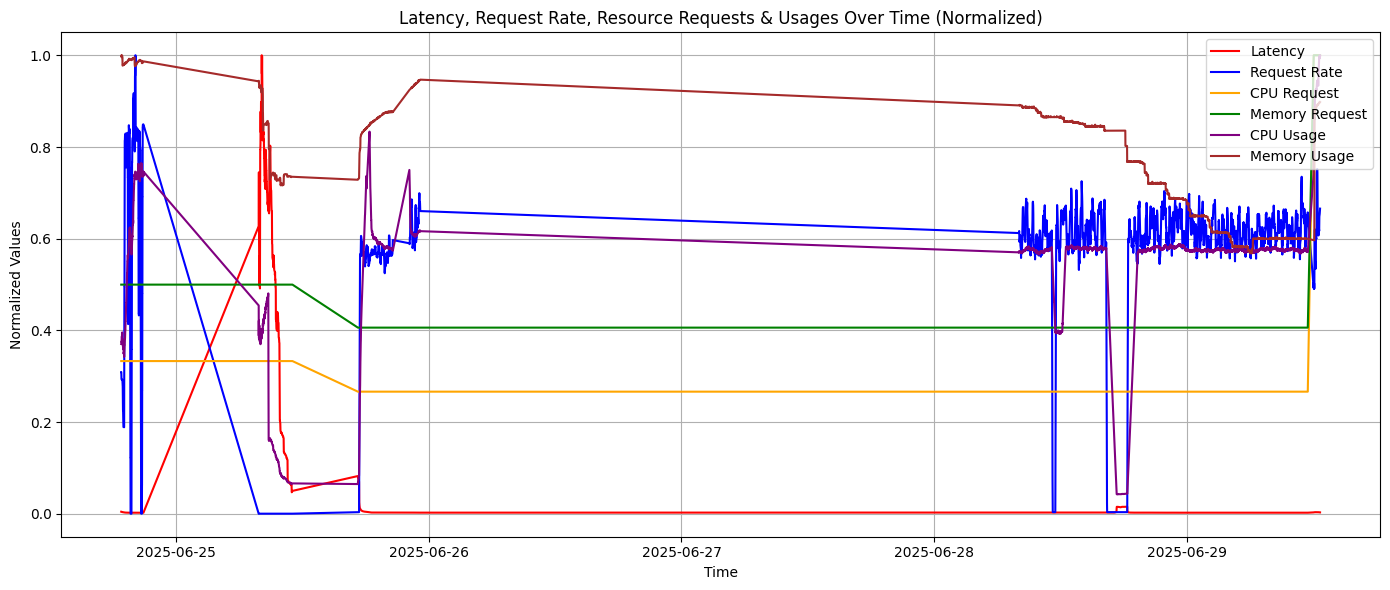
# Service 1



## 1. Observation

Latency:

* June 25: High volatility and sharp latency spikes.
* June 26–29: Latency sharply drops and remains close to zero.

## 2. Interpretation – In-Depth Breakdown

### A. Sharp Latency Spikes on June 25

* These spikes indicate high response times for the application.
* Latency values may be breaching the SLO (Service Level Objective) thresholds intermittently.

|  |  |
| --- | --- |
| Root Cause | Explanation |
| Resource Reduction | CPU/memory request have been reduced, causing queuing or throttling. Check cpu\_throttled\_seconds\_total and memory OOM metrics. |
| Cold Starts or Pod Initialization Delays | New pods were created on demand (e.g., in HPA or VPA scenarios), latency spikes reflect startup delay. |
| Garbage Collection or Go GC Delays | The service is CPU-intensive, latency spikes arise from GC pauses. |
| Reactivity Delay | Scaling process have not reacted quickly enough to load, creating a temporary resource–demand mismatch. |

### B. Latency Drops to ~0 from June 26 to June 29

**Possibilities:**

1. System was Idle

* Ifrequest\_rate ≈ 0, the service was not receiving external load.
* In such cases, latency measurements fall near-zero since:
* No requests are being processed.
* Some systems report default low values or do not record at all.

1. Service Failure or Unavailability

* Latency being near-zero can falsely appear healthy when in fact:
  + The service may have crashed.
  + No pods running, hence no metrics captured.
  + Prometheus scrapes default to zero if no target is available.

1. Aggressive Resource Overprovisioning

* CPU/memory limits may have been increased excessively by an autoscaler or manually.
* This can lead to underutilization and very fast request processing.
* If latency is near-zero with a stable non-zero request rate, this is plausible.

1. Caching or Optimization Mechanism Kicked In

* Caching layers (e.g., Redis, in-memory caches) may have reduced workload drastically.

1. Service is lightweight

* The service can be very lightweight & so the latency will be almost near zero.

|  |  |  |
| --- | --- | --- |
| Behaviour | Most Likely Explanation | Additional Notes |
| Spikes | Resource contention, delayed autoscaling | Indicates stress; tune autoscaler or initial allocation |
| Drop near zero | Lightweight service |  |

### Insights

If we're building a latency-aware autoscaler:

* We must not react to low latency alone — always combine it with:
  + request\_rate
  + cpu\_usage\_pct and memory\_usage\_pct
  + Service health metrics (up, restarts)
* Introduce latency bounds:
  + A latency close to zero with zero usage should trigger scale-down or no-op, not mislead the system into thinking resources are sufficient for future high loads.
* Modeling tip:  
   If we're using this dataset for ML, filter out near-zero request periods or label them as low activity windows so they don’t bias our model.

## Request Rate

* Significant initial spike — a surge in incoming requests.
* Indicates active use or a sudden test/load event.
* Near-zero request rate throughout most of the day.
* Implies the system was light weight, idle, possibly turned off, or not receiving any external/internal requests.
* Frequent, fluctuating activity resumes.
* Pattern resembles non-continuous, possibly batch, or time-triggered request bursts.

### **Layer-by-Layer Technical Breakdown**

**High Initial Request Rate**

|  |  |
| --- | --- |
| Cause | Description |
| Load Testing Initiated | We or a process (e.g., curl, JMeter, Python client) started a high-throughput workload for performance testing. |
| CronJob Executed | A Kubernetes CronJob or scheduled script have kicked off a burst of requests at a specific time. |
| Autoscaling Test or Experiment Triggered | Controlled experiment have started on that day to test latency impact under load. |
| User Traffic or Synthetic Load Generator | External clients or a synthetic generator like Locust, or a Python requests thread pool. |

### Dormant State

|  |  |
| --- | --- |
| Cause | Description |
| Scheduled Traffic | System was intentionally get random traffic. |
| Cron-based Load Generator | CronJobs scheduled only at specific intervals (e.g., every 10 mins). |
| Service Crash or Unavailability | The service might have been down. In such cases, no requests could be processed or recorded. |

If latency dropped to near-zero during this idle time while our autoscaler keeps CPU/memory constant, we might misinterpret this as success. Models should account for low activity intervals.

### Fluctuating Bursts of Activity

* Non-linear, jagged increases and decreases in request rate.
* Appears to be periodic or event-driven, rather than continuous.

|  |  |
| --- | --- |
| Cause | Description |
| CronJob or Scheduled Tasks Every Few Minutes | Jobs \*/10 \* \* \* \* in a Kubernetes CronJob manifest cause this pattern. |
| Dynamic Load Generator Behavior | The load test system (Locust or a custom script) have randomized pause/sleep times, producing natural-looking traffic. |

## Insights

### Key Benefits of This Request Pattern:

Good for Robustness Testing: The fluctuating nature challenges our autoscaler’s adaptability.  
Latency–Load Sensitivity: Useful for training or evaluating ML models that learn to predict latency under varied load.  
Simulation of Real-world Load: Many production workloads behave like this — idle, then sudden bursts.

### Cautions:

* Don’t confuse low latency during zero load as a success metric.
* A dynamic request rate is excellent for online learning models, but we need to filter or tag the idle periods in our dataset to avoid biasing toward no-op actions.
* Ensure latency anomalies are correlated with request rate spikes (not just resource limits).

|  |  |  |
| --- | --- | --- |
| Request Rate Behavior | Interpretation | Action |
| Spike in traffic | Load test or scheduled batch | Tune autoscaler |
| Zero or near-zero | Idle state / cron pause / failure | Filter in modeling |
| Bursty, dynamic | Cron-based or synthetic dynamic workload | Perfect for testing adaptive autoscaling |

## CPU Request & Memory Request – Stepped & Static Behavior

|  |  |
| --- | --- |
| Metric | Behavior |
| CPU Request | Flat segments with a few discrete jumps (step-like) |
| Memory Request | Similarly stepped pattern |
| Step Frequency | Low — only a few changes observed, mostly remaining constant over long intervals |

## Latency vs Request Rate Correlation

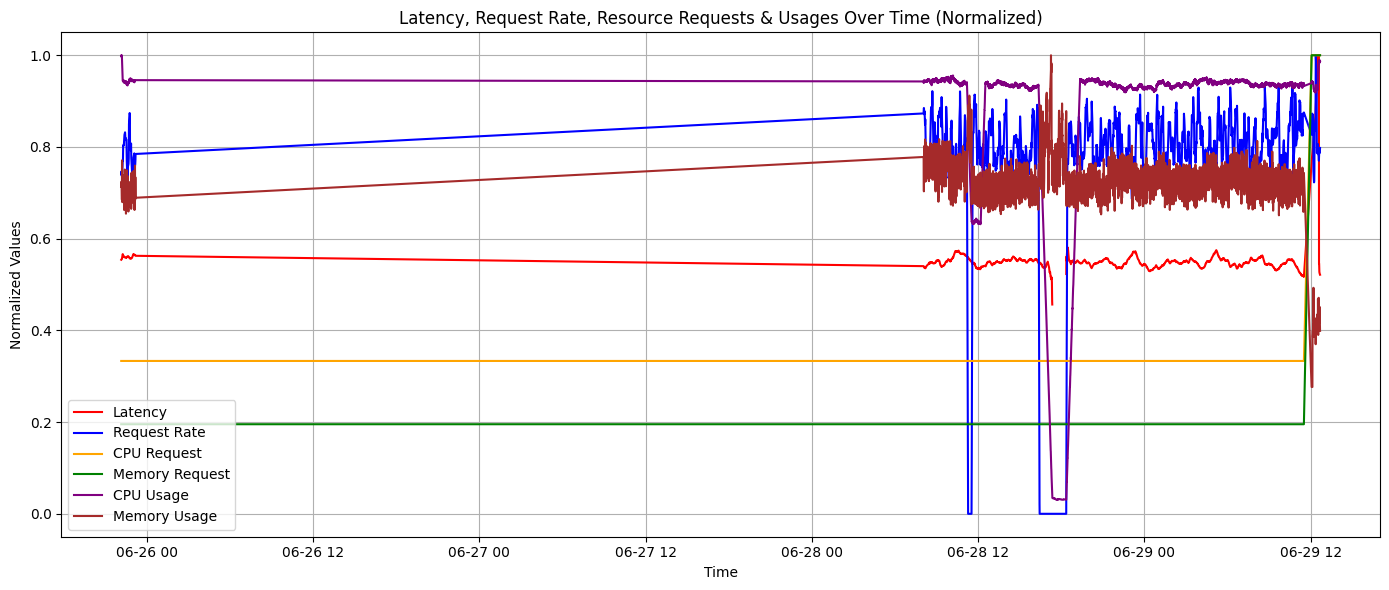
**Observed:**

* High request rate leads to high latency  
   (expected behavior under load)
* Request rate spikes again, but latency remains low  
   (unexpected — suggests the system handled it better

**Interpretation:**

|  |  |
| --- | --- |
| Condition | Explanation |
| Positive correlation | Indicates system under stress; higher request rate → more processing → longer queue time or resource contention. |
| No correlation | Either:   * System scaled up (e.g., higher CPU/memory requests or limits) * Service logic improved (e.g., better caching, batching) * Resource bottlenecks removed (e.g., throttling reduced, garbage collection tuned) |

# Service 2



## Request Rate — Intermittent but High Volume

### Observed Behavior

|  |  |
| --- | --- |
| Phase | Pattern Description |
| 1 | A small, sharp burst in request rate — short duration, low magnitude. |
| 2 | High-frequency, dense, noisy pattern — continuous, high-volume traffic for several hours. |
| 3 | Complete and abrupt drop in request rate to zero. |
| 4 | Traffic resumes, fluctuates with noise, then gradually tapers off. |

### Phase 1 - Initial Low Burst

* The first scheduled test, warming up a synthetic load generator or executing a small CronJob.
* A "sanity check" workload to verify system health.
* Alternatively, a manual test run or script.

### Phase 2 - High-Frequency Load

* A scheduled, high-throughput test.
* Load pattern is dense and noisy, with many small fluctuations per unit time, indicating frequent requests per second.

Very typical of:

* Synthetic workload generators (e.g., JMeter, Python requests with thread pools, wrk, Locust).
* Periodic CronJobs submitting HTTP/gRPC/API requests.
* Performance experiments or ML reward collection loops.

### Phase 3 - Sudden Drop to Zero

An abrupt drop to zero usually does not occur naturally in a real workload. This strongly indicates a failure, pause, or disruption in the system.

|  |  |
| --- | --- |
| Cause | Description |
| Load Generator Crash / Pause | Our cron-based generator have stopped due to error or timeout. |
| Deployment Restart / Pod Eviction | If the application pods were restarted or evicted (e.g., OOMKilled, Evicted, CrashLoopBackoff), incoming requests will fail → request rate = 0. |
| Networking Failure | Temporary DNS failure, Kubernetes CNI issue, or ingress controller crash. |
| Scheduled Pause | If the test was intentionally designed to pause after X time, this is expected. |

### Phase 4: Resumption → Tapering

**Interpretation:**

* Load resumed after the outage. This suggests:
  + Generator recovered automatically (e.g., retry or restart policy).
  + Infrastructure (DNS/network) recovered.
  + Manual restart of job or pod.
* Gradual tapering  indicates:
  + Test completion, or
  + End of working hours, or
  + Cron-based job expiration, e.g., every hour for N mins.

|  |  |
| --- | --- |
| Behavior | Explanation |
| Fast recovery → generator resilience | Your system can recover gracefully from downtime. |
| Same noisy pattern returns | Generator resumed same mode — high-rate, multithreaded/looped structure |
| Gradual tapering | Indicates either generator ramp-down logic or a planned test closure |

Latency Behavior: Flat, Then Mildly Reactive

|  |  |
| --- | --- |
| Phase | Latency Pattern |
| Flat Period | Latency remains very low and steady across all time points. |
| Reactive Phase | Latency starts to fluctuate mildly, though without sharp spikes. |
| Notable Dips | Sudden **d**ownward dips in latency, seemingly out of sync with other signals |

### Phase 1: Flat Latency Under Low Load

Latency remained flat and very low during long idle or low-load periods.

**Interpretation:**

* The system is handling traffic well within its resource envelope.
* causes:  
    
  + Very low request rate → no queue buildup.
  + Overprovisioned CPU/memory → system has more than enough headroom to serve each request immediately.
  + Service is "warmed up" → runtime optimizations like JVM JIT, connection pool readiness, etc., are in place.

**Technical Implications:**

* The flat latency here is expected.
* A latency-aware autoscaler might incorrectly assume this is "optimal baseline", but it’s only true under low stress.