FluxNinja

Mastering Chaos: Achieving Fault Tolerance with Observability-Driven Prioritized Load Shedding

Building fault-tolerant, performant and cost-efficient applications with the ***Aperture*** open source project

Harjot Gill
CEO, FluxNinja

@harjotsgill
@harjotsgill

Hardik Shingala
Engineer, FluxNinja

@HDKShingala
@hardik-shingala

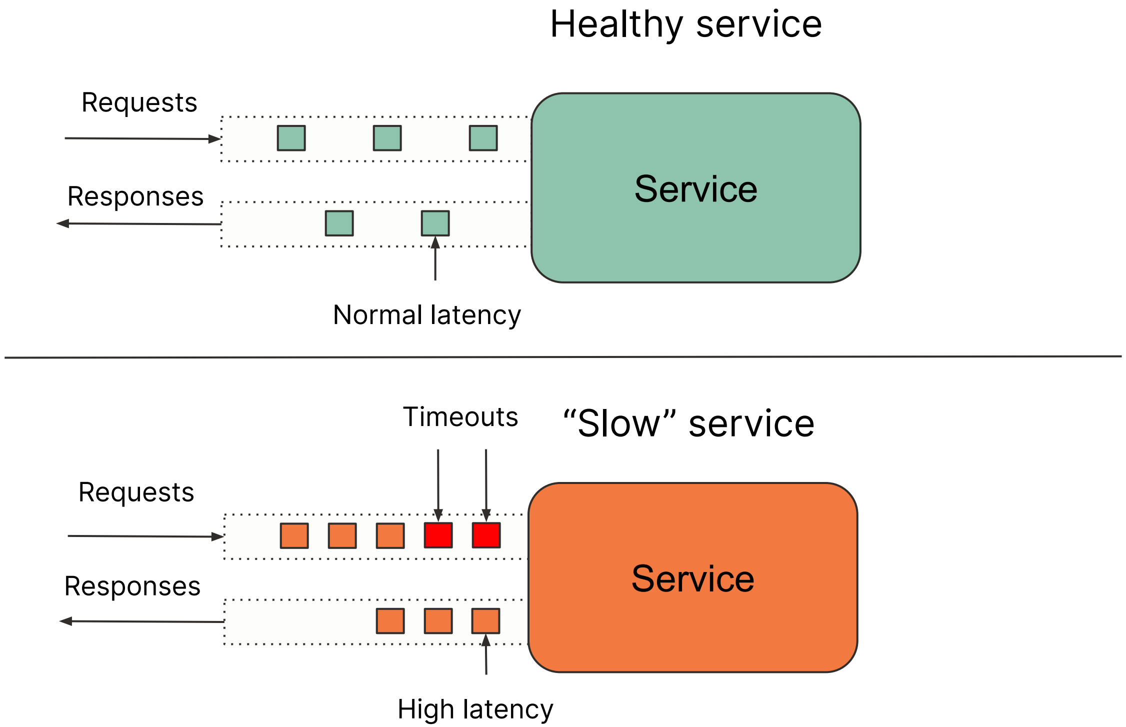
Introduction

- **Harjot Gill**
  - Co-founder and CEO @ FluxNinja
    - Founded in 2021
    - Based in the San Francisco Bay Area
    - Announced Aperture open source project in late 2022
  - Dedicated 10+ years building tooling for DevOps and SREs
  - Previously, Co-founder and CEO @ Netsil (Acquired by Nutanix in 2018)
    - Microservices observability start-up, spin-off from University of Pennsylvania
    - Pioneered low-friction API observability: stream-processed packets to reconstruct APIs
    - Mapping complex microservices applications
- **Hardik Shingala**
  - Software Engineer @ FluxNinja
  - 5+ years of experience in cloud native infrastructure products

Metastable failures

Little's law conundrum: The inevitability of overloads

Little's law and overloads



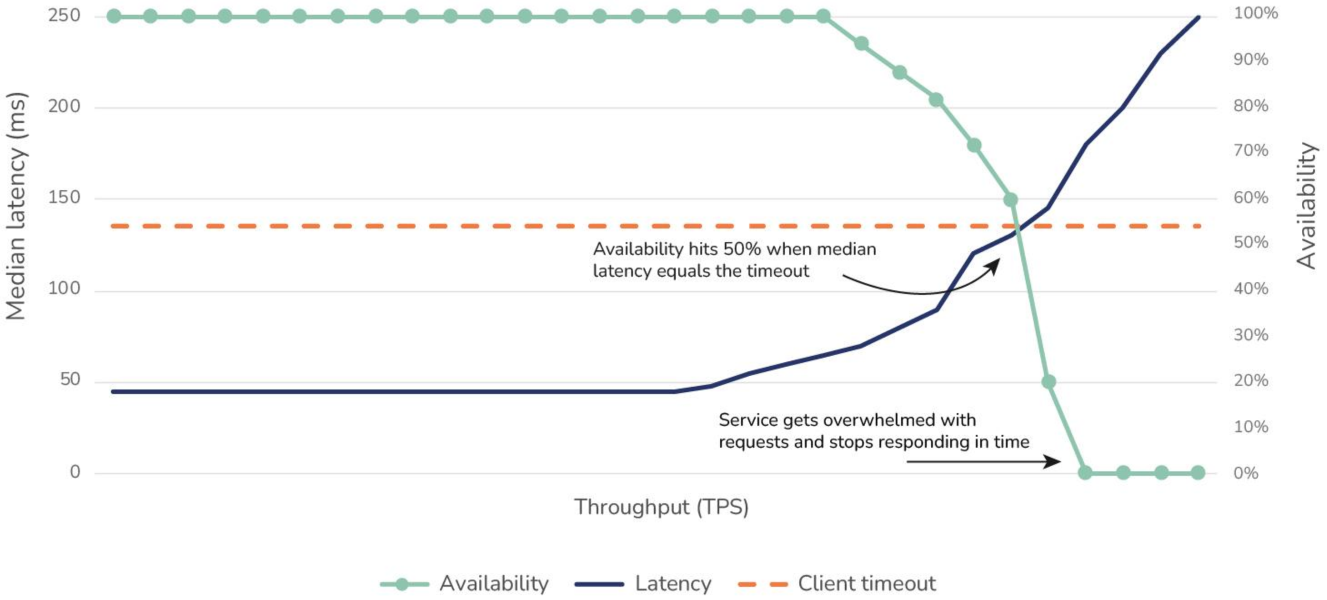
Little's law

L = λW

L = Requests in-flight
λ = Average Throughput
W = Average Response time

Every service has an inherent concurrency limit. For a service to remain ***stable****, concurrent requests must be limited*

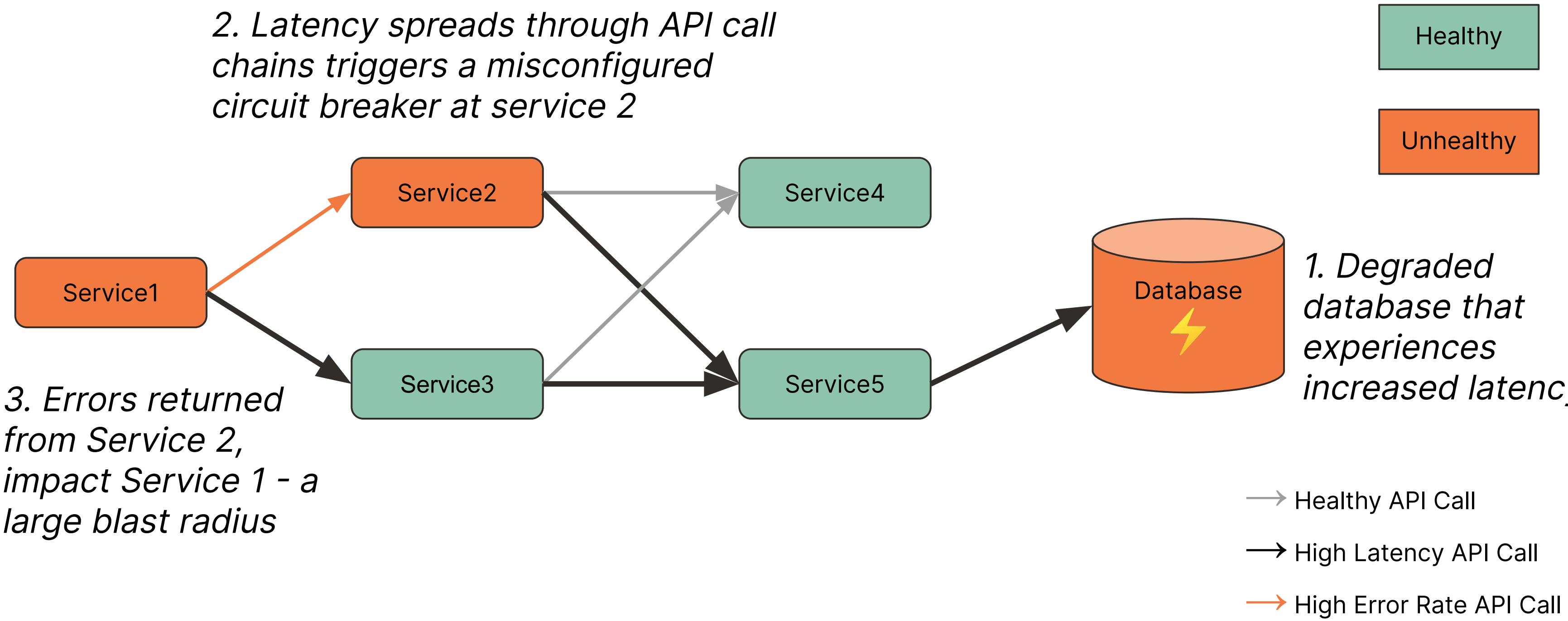
Availability degrades rapidly



An overload on a service often kicks-off a chain reaction causing an application wide outage...

Cascading failure

2. Latency spreads through API call chains triggers a misconfigured circuit breaker at service 2

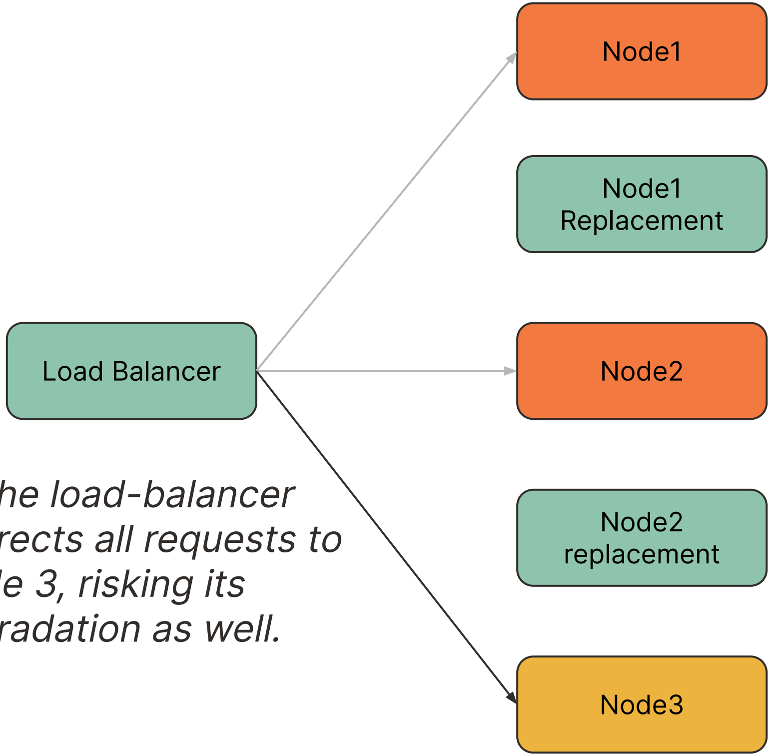


1. Degraded database that experiences increased latency.

3. Errors returned from Service 2, impact Service 1 - a large blast radius

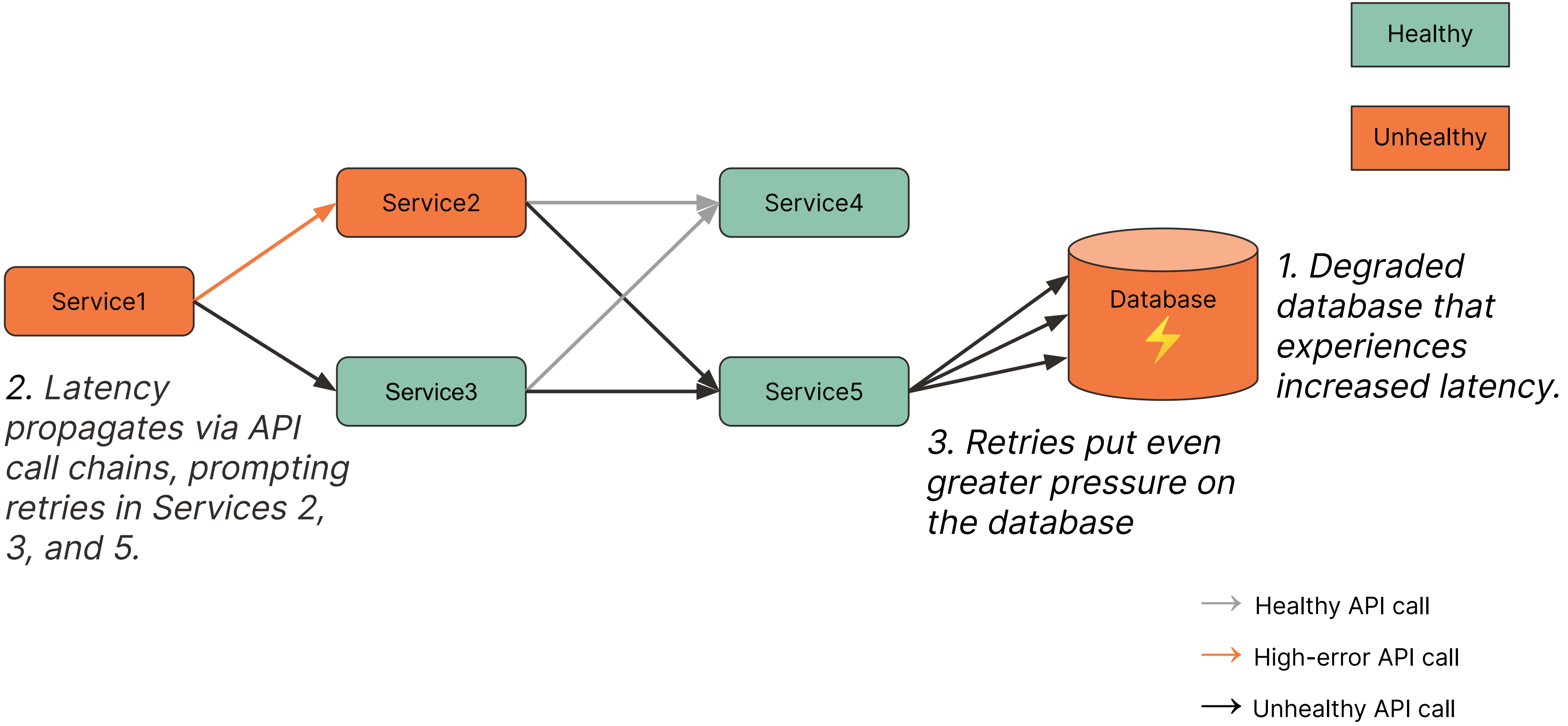
Death spiral

1. *Nodes 1 and 2 degrade and are replaced by new nodes which are not ready for traffic.*

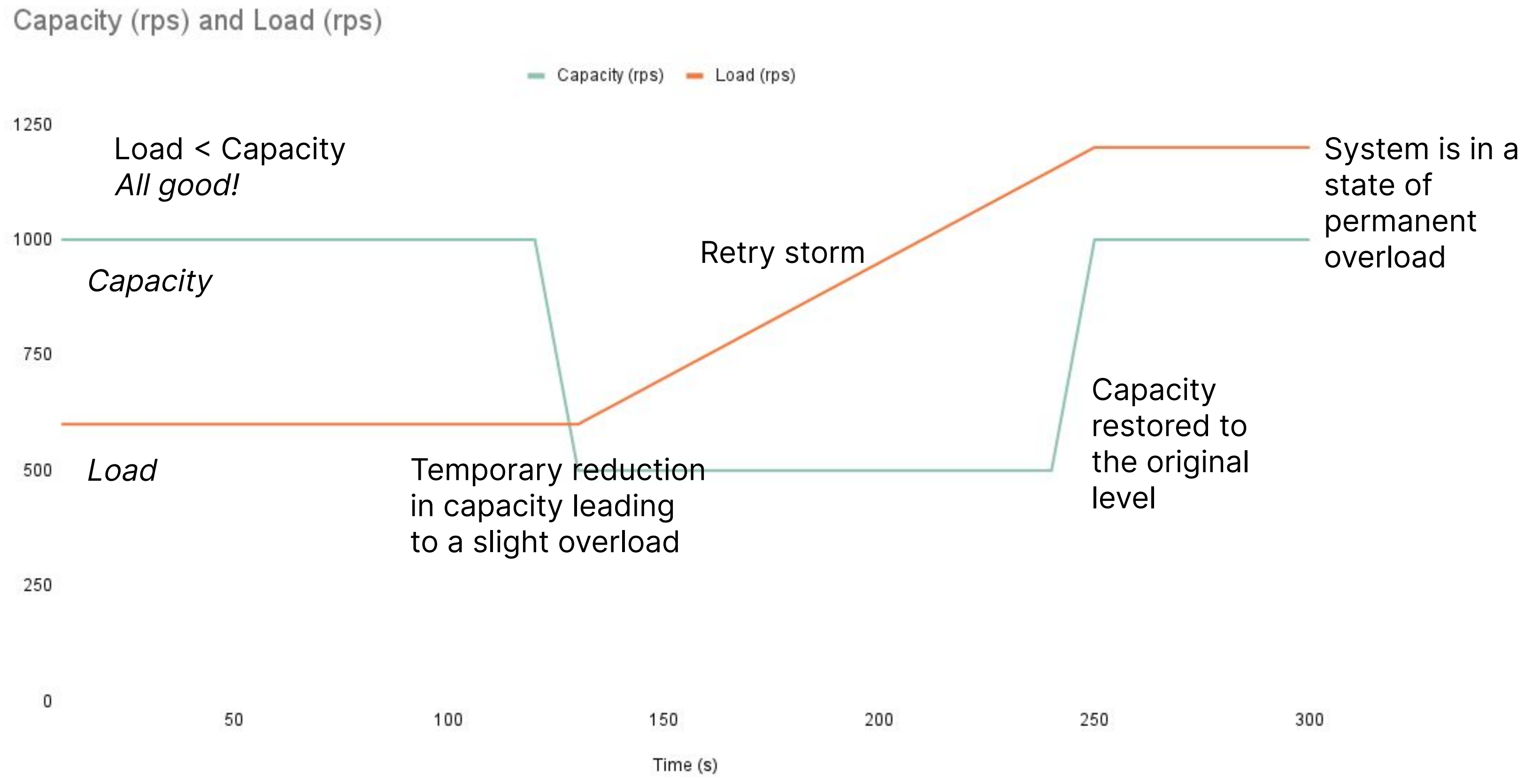


2. *The load-balancer redirects all requests to Node 3, risking its degradation as well.*

Retry storm

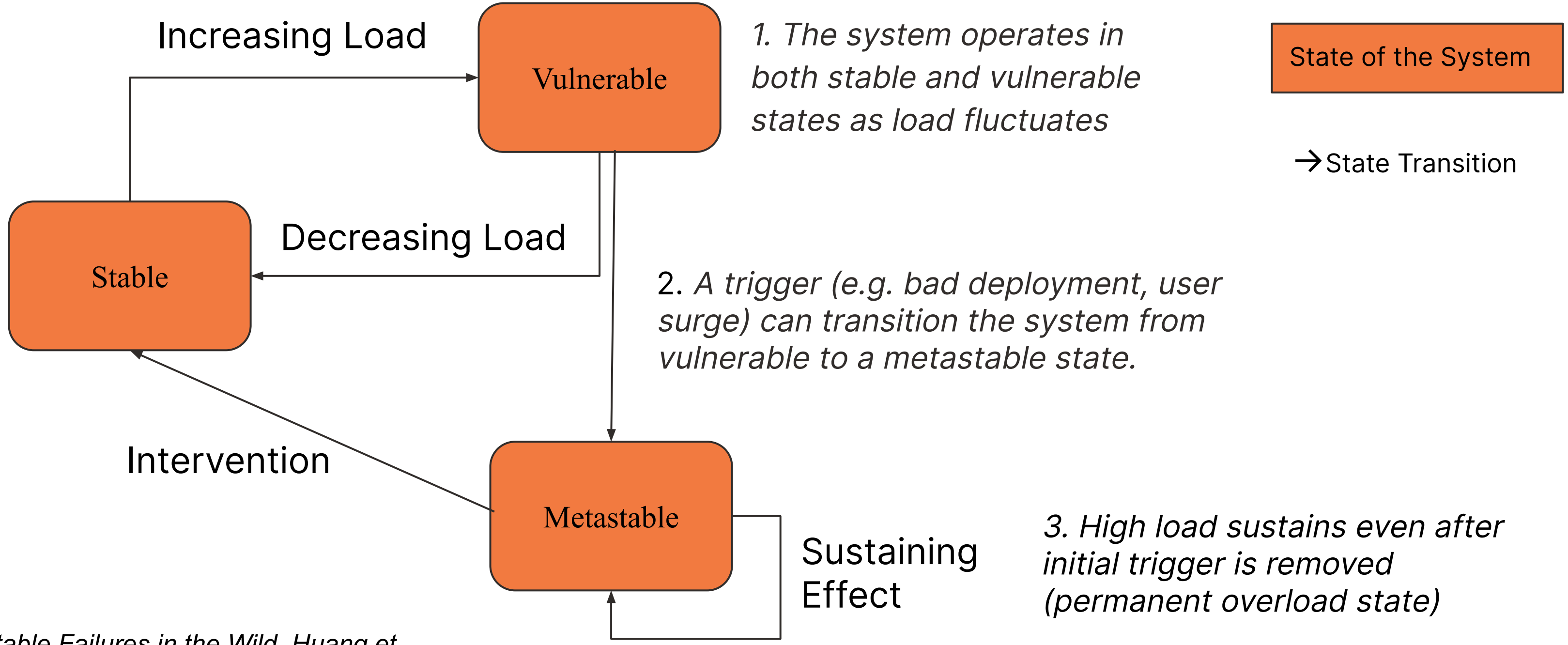

Healthy
Unhealthy
Service1
Service2
Service3
Service4
Service5
Database
1. Degraded database that experiences increased latency.
2. Latency propagates via API call chains, prompting retries in Services 2, 3, and 5.
3. Retries put even greater pressure on the database
Healthy API call
High-error API call
Unhealthy API call

Retry storm: permanent overload



Metastable failures

FluxNinja



1. The system operates in both stable and vulnerable states as load fluctuates

2. *A trigger (e.g. bad deployment, user surge) can transition the system from vulnerable to a metastable state.*

3. High load sustains even after initial trigger is removed (permanent overload state)

Metastable Failures in the Wild, Huang et al.

Common triggers

FluxNinja

- Insufficient capacity allocation
- Service upgrades that introduce performance-regressions due to bugs
- Unexpected traffic spikes during new product launches or sales promotions
- Slowdowns in upstream services or third-party dependencies
- Retry storm after a temporary failure
- Cache failure leading to higher load on database
- Subset of servers going offline causing excess load on remaining servers

Metastable failures are unpredictable, yet very common in modern applications

Mitigation strategies

Building indestructible applications

Local countermeasures are ineffective

Circuit breaking

- Typically implemented in service proxy (e.g. Envoy)
- Localized view between service instances (e.g. error rates)
- Rejects all requests when it “trips”
- Hard to configure the “tripping” threshold as some services are more tolerant to errors
- Client-side technique - does not offer service protection

Static rate-limiting

- Typically implemented as a per-user limit
- Does not offer service protection as the per-user limit is not per-service limit

Reactive auto scaling

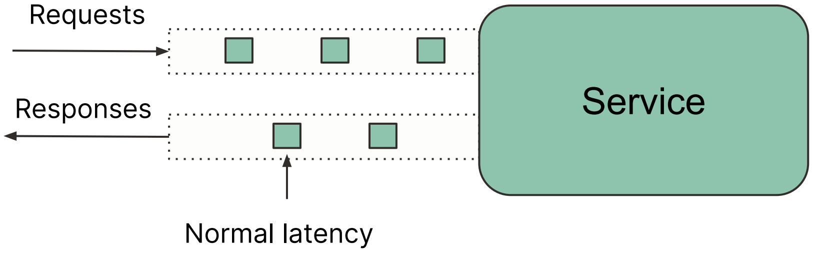
- Typically scale workers based on resource consumption (e.g. CPU or memory)
- Can be slow as services need time to warm-up, do discovery, establish database connections and so on
- Bottleneck typically shifts elsewhere
- Expensive to absorb transient traffic spikes

Local countermeasures are often slow, inadequate and ineffective

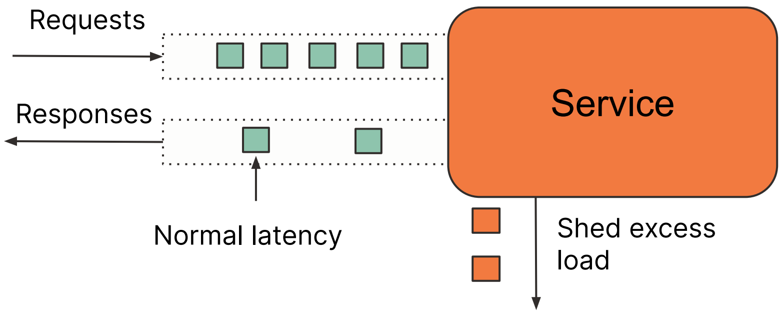
Mitigation with adaptive load shedding

FluxNinja

Normal load



Overload



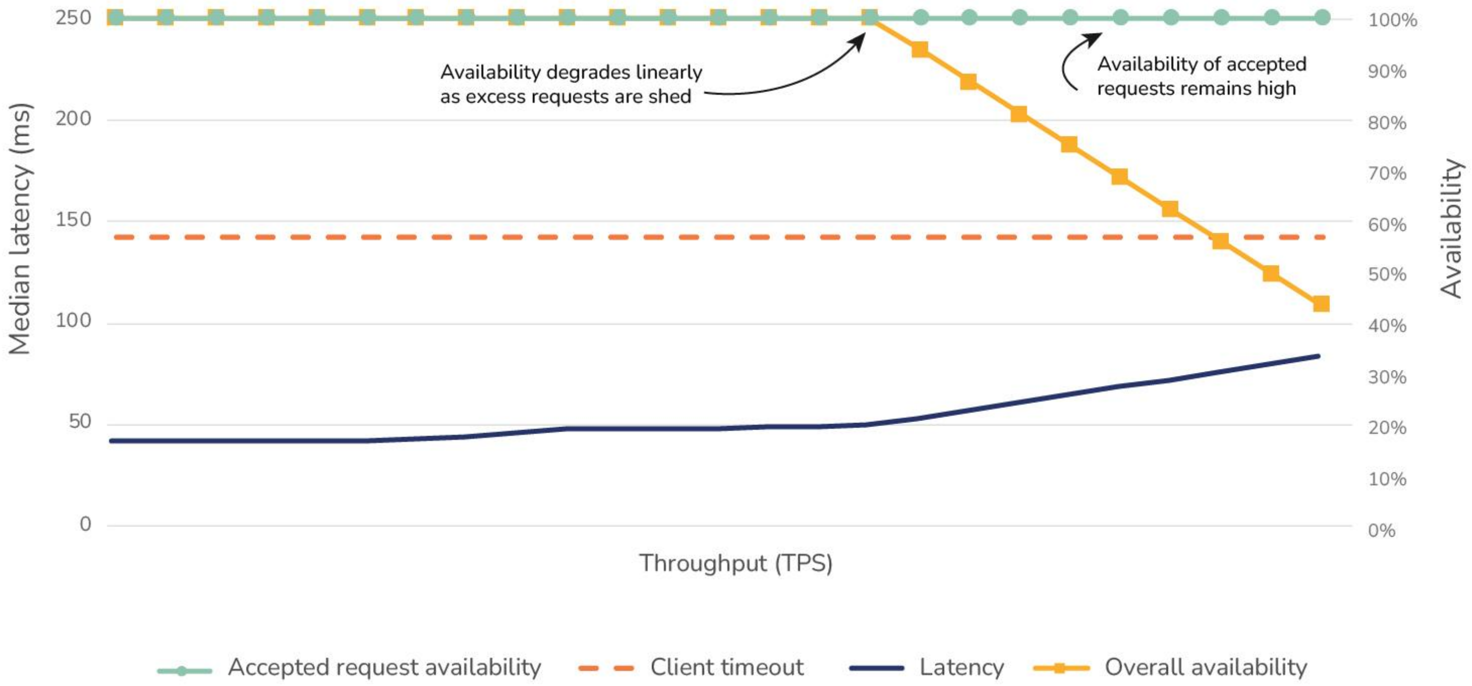
Little's law

$L = \lambda W$

L = Requests in-flight
λ = Average Throughput
W = Average Response time

*Service remains **stable** by shedding excess load*

Availability degrades gracefully



Requirements for adaptive load shedding

- Determining the ideal load in a constantly changing environment
  - Setting the limit too low can result in rejected requests and wasted capacity
  - Setting the limit too high can lead to slow and unresponsive servers
- Observability: Real-time, global visibility into the state of the entire system
  - Detect overload at databases but load shed at the gateway services
- Controllability: Continuously tracking and correcting system state variables
  - PID controller based closed-loop system
  - Congestion control and active queue management algorithms: TCP BBR, AIMD (Additive increase, multiplicative decrease), CoDel
- Interaction with other control systems with similar goals:
  - Auto scaling
  - Load balancing

Requirements for prioritization

- Optimize user experience and business value: prioritize on attributes such as API endpoints, user types, origin service
- Prioritization and fairness algorithms
  - Token and leaky buckets
  - Network schedulers: weighted-fair queueing
  - Probabilistic dropping
- Estimating the cost (tokens) of admitting different types of requests
  - Tokens = Estimated latency?
  - Tokens = Query complexity?

Global load management with Aperture

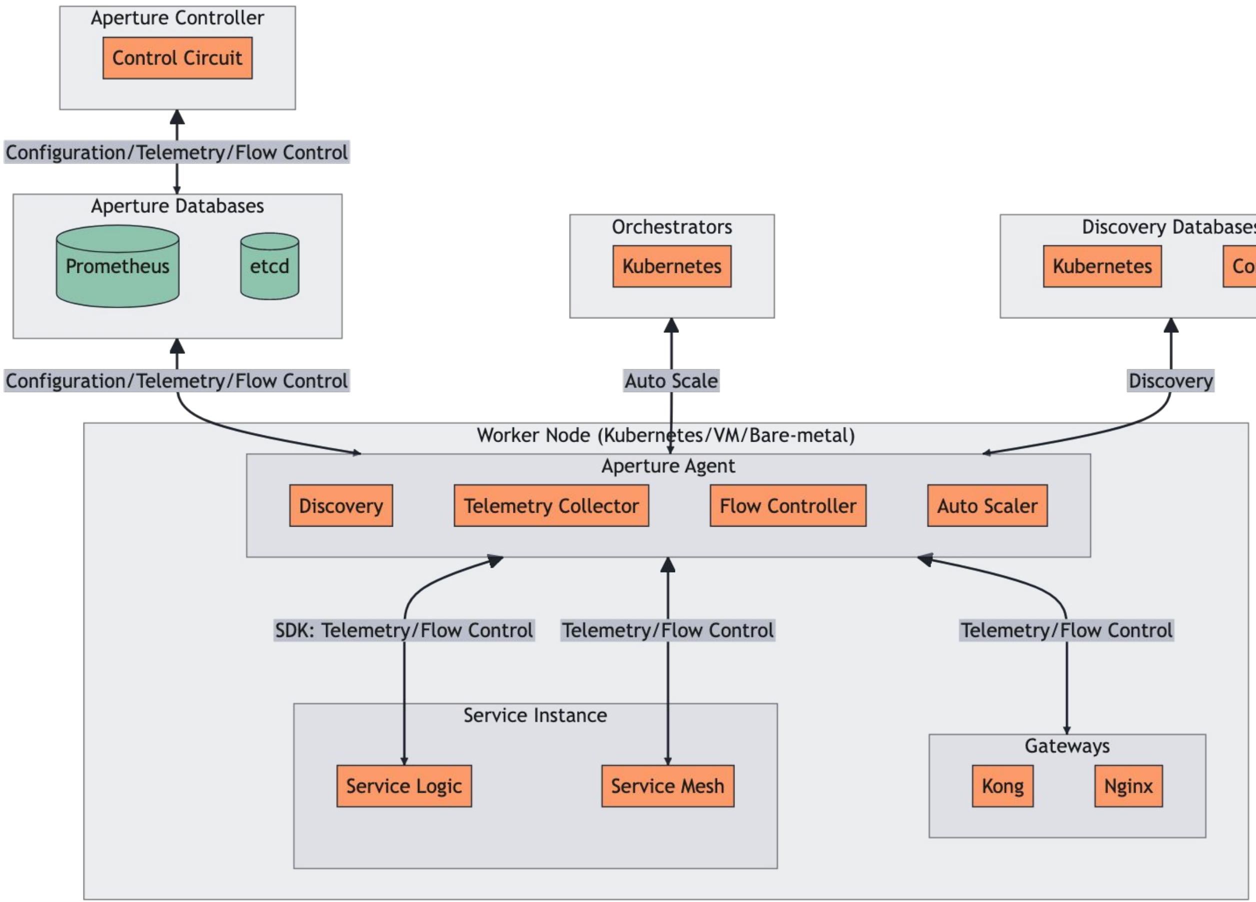
Controlling the flux: Observability meets Controllability

Aperture overview

FluxNinja

- Open source platform for observability-driven load management
- Programmable through declarative policy language expressed as a control circuit graph
- Common policies are packaged as high-level “blueprints”
  - Load scheduling & workload prioritization
  - Quota enforcement
  - Load ramping
  - Auto scaling
- Layered on top of existing stack
  - SDKs: Java, Go, Python etc.
  - Service Mesh: Istio etc.
  - API Gateways and proxies: Nginx, Kong

Aperture architecture

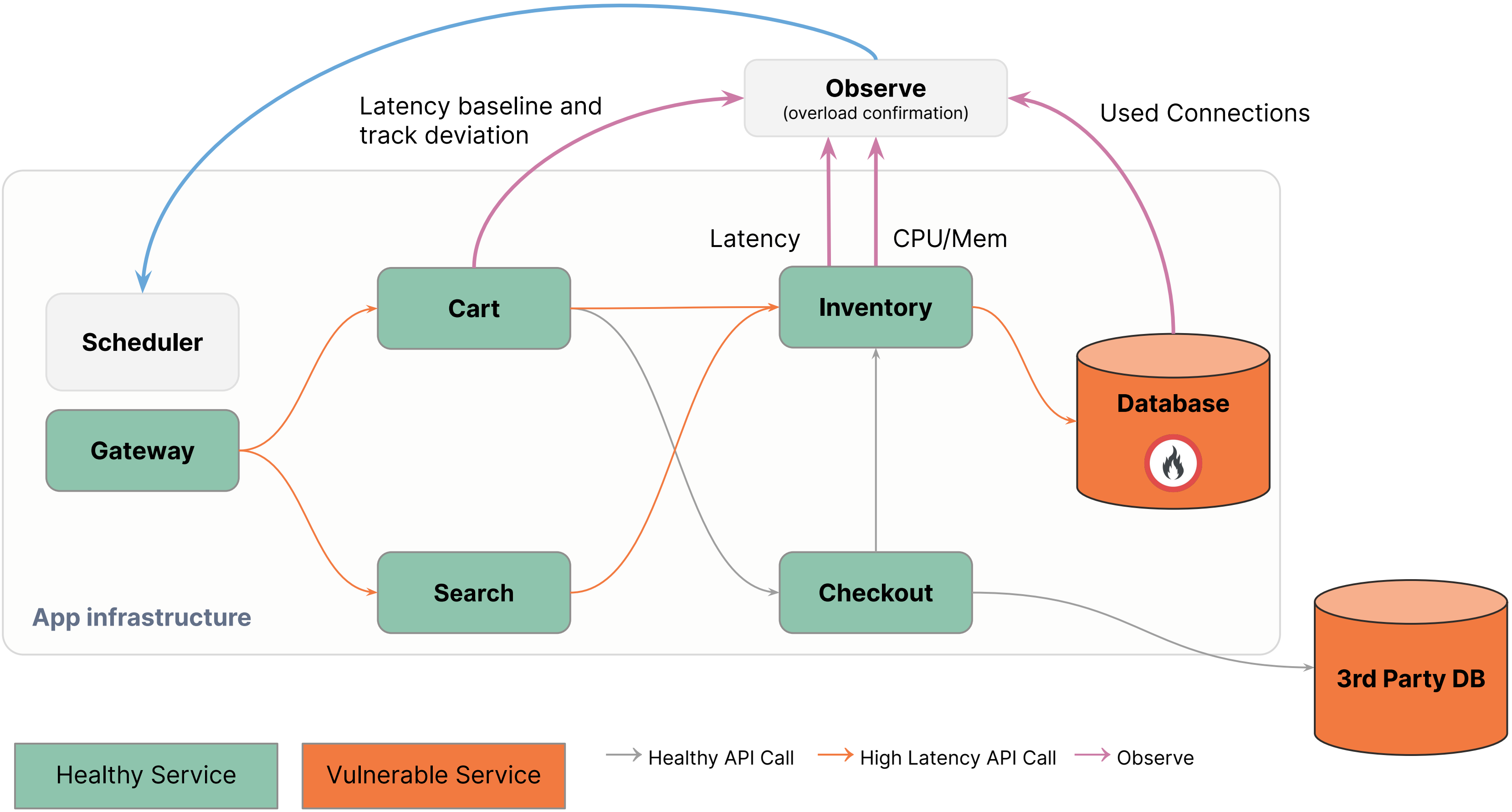


FluxNinja

Adaptive load scheduler

Service protection based on feedback loop

Observability-driven approach

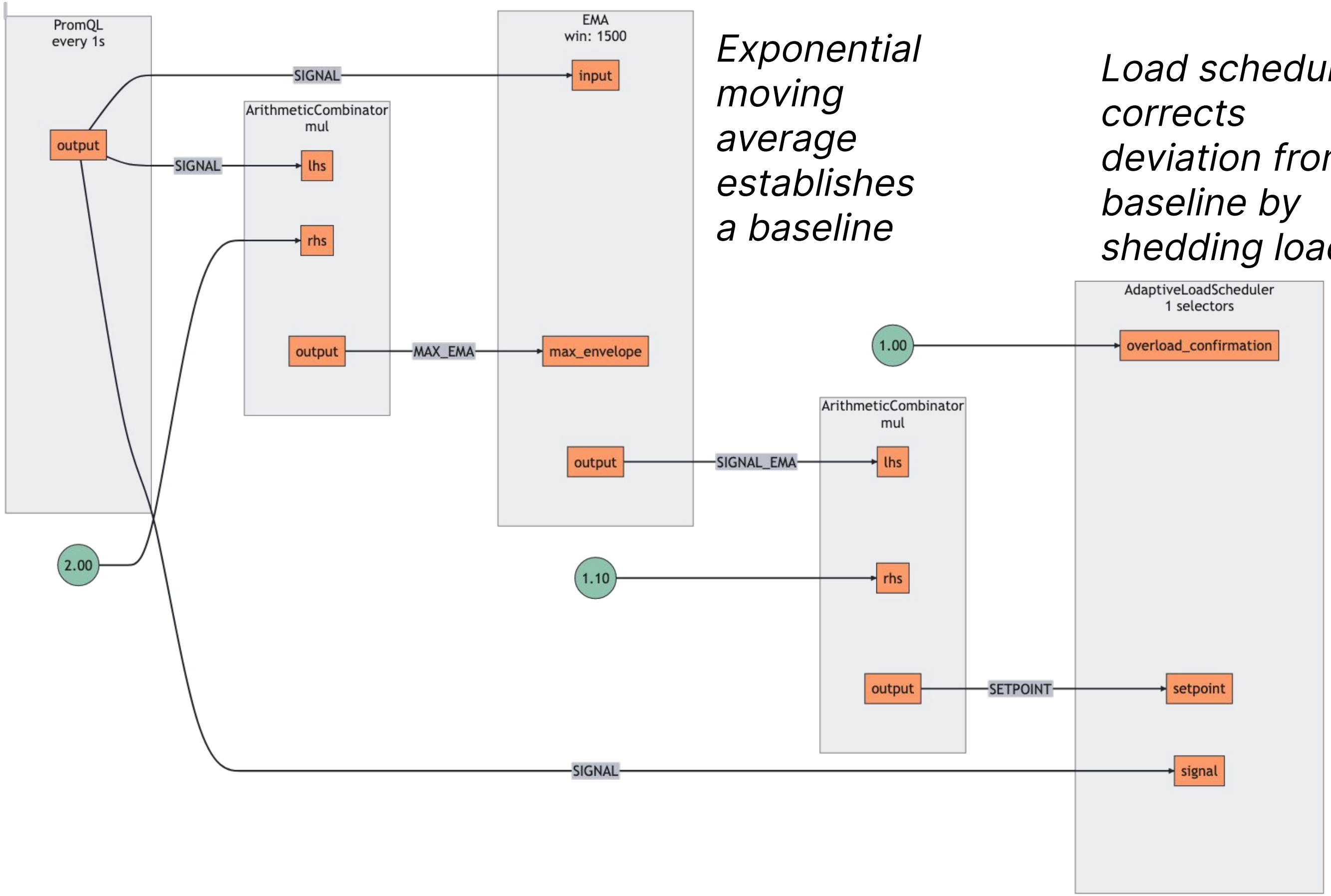


Adaptive load scheduling policy

Service latency is queried periodically

Exponential moving average establishes a baseline

Load scheduler corrects deviation from baseline by shedding load



FluxNinja

Load scheduler policy component

```yaml
circuit:
  components:
    - flow_control:
        adaptive_load_scheduler:
          in_ports:
            setpoint:
              signal_name: SETPOINT
            signal:
              signal_name: SIGNAL
          out_ports:
            desired_load_multiplier:
              signal_name: DESIRED_LOAD_MULTIPLIER
            observed_load_multiplier:
              signal_name: OBSERVED_LOAD_MULTIPLIER
          parameters:
            load_scheduler:
              scheduler:
                workloads:
                  - label_matcher:
                      match_labels:
                        user_type: guest
                    parameters:
                      priority: 50
                  - label_matcher:
                      match_labels:
                        user_type: subscriber
                    parameters:
                      priority: 200
              selectors:
                - control_point: ingress
                  service: service1-demo-app.demoapp.svc.cluster.local
```

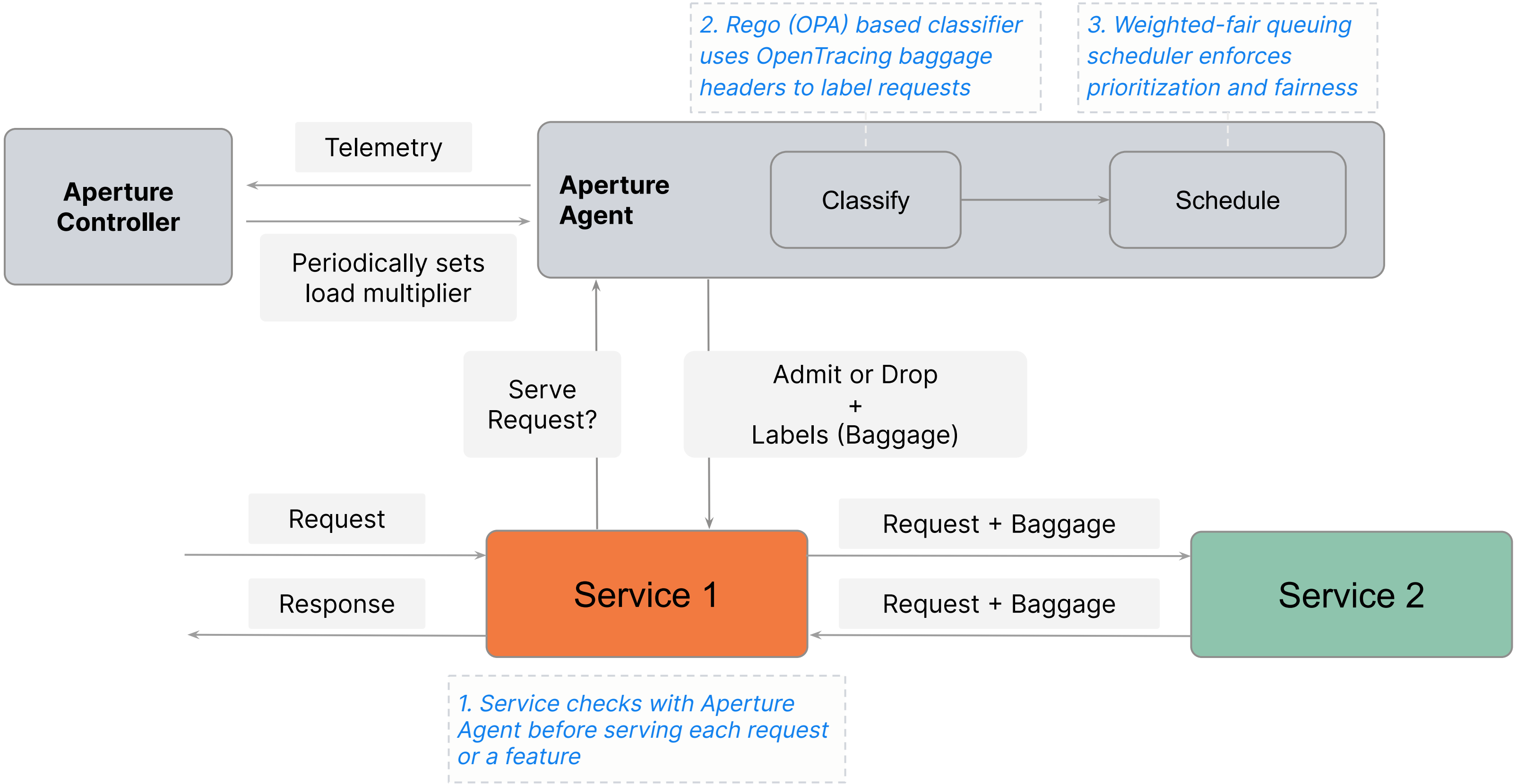
Policy is expressed as a control "circuit" composed of components

Signals flow between components through ports and the circuit is evaluated periodically

Workloads are defined by matching labels and assigning priorities

Selectors determine agents where this scheduler will be configured

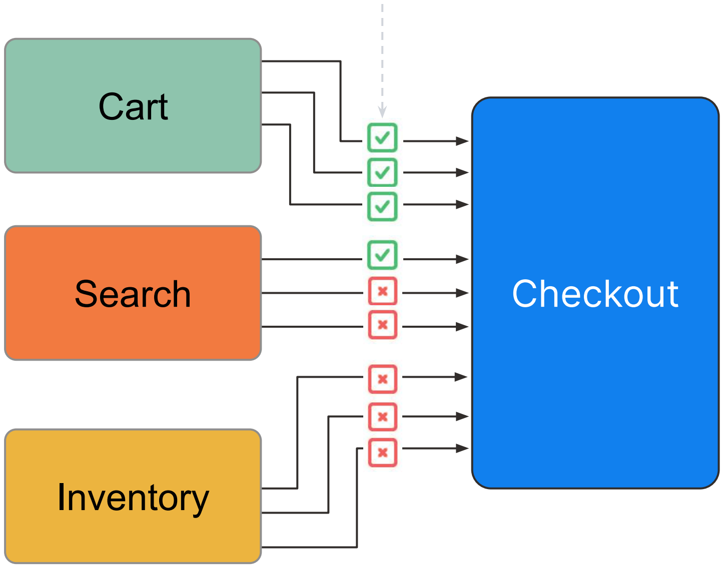
Adaptive load scheduler insertion

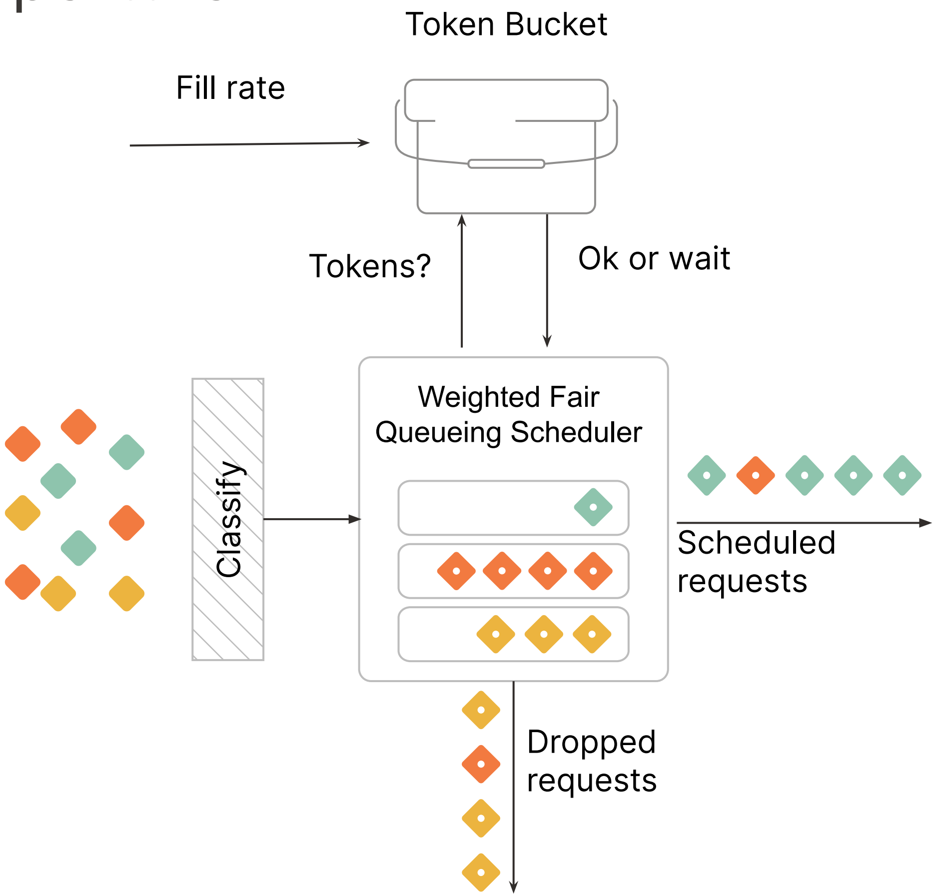


FluxNinja

Workload prioritization with Aperture

Client Service	Priority
Cart	255
Search	150
Recommendations	10



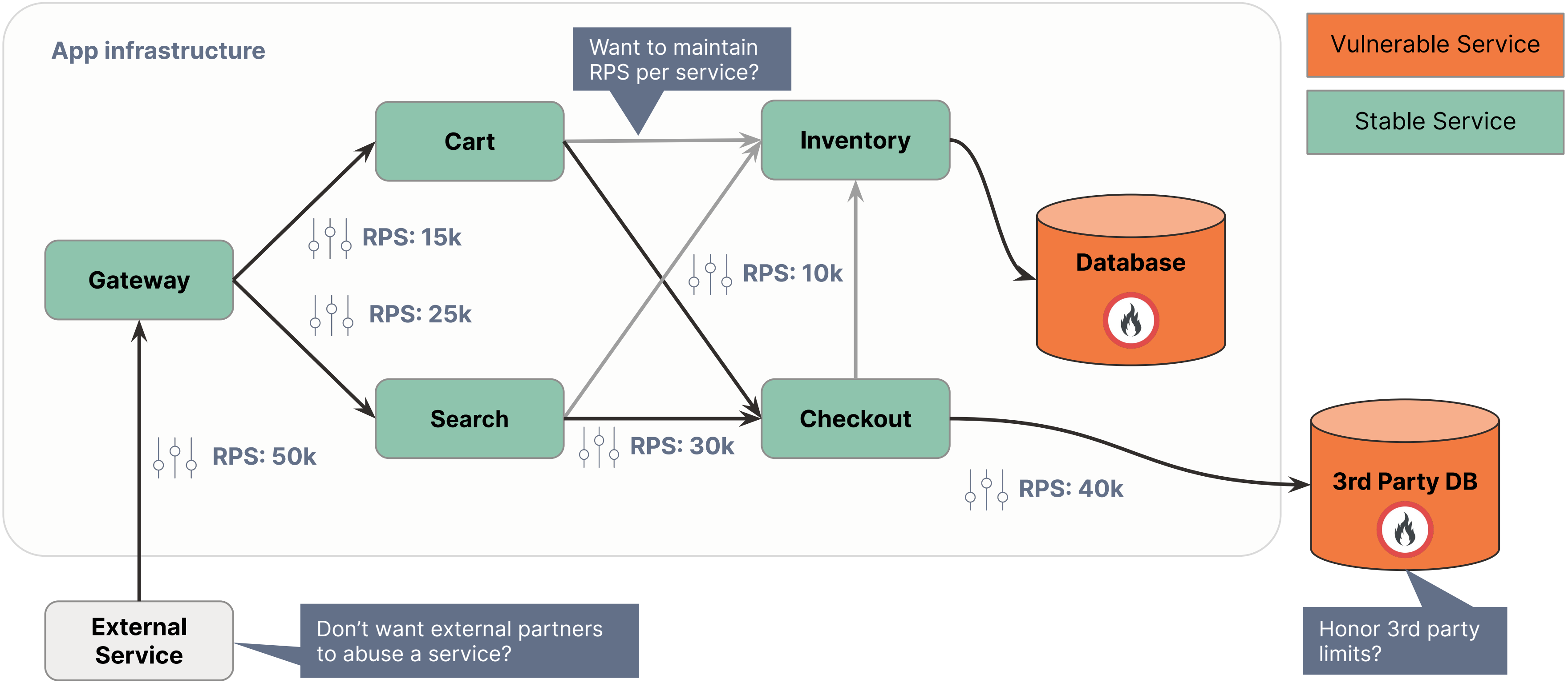


Global quotas

Enforcing precise limits

Global quotas

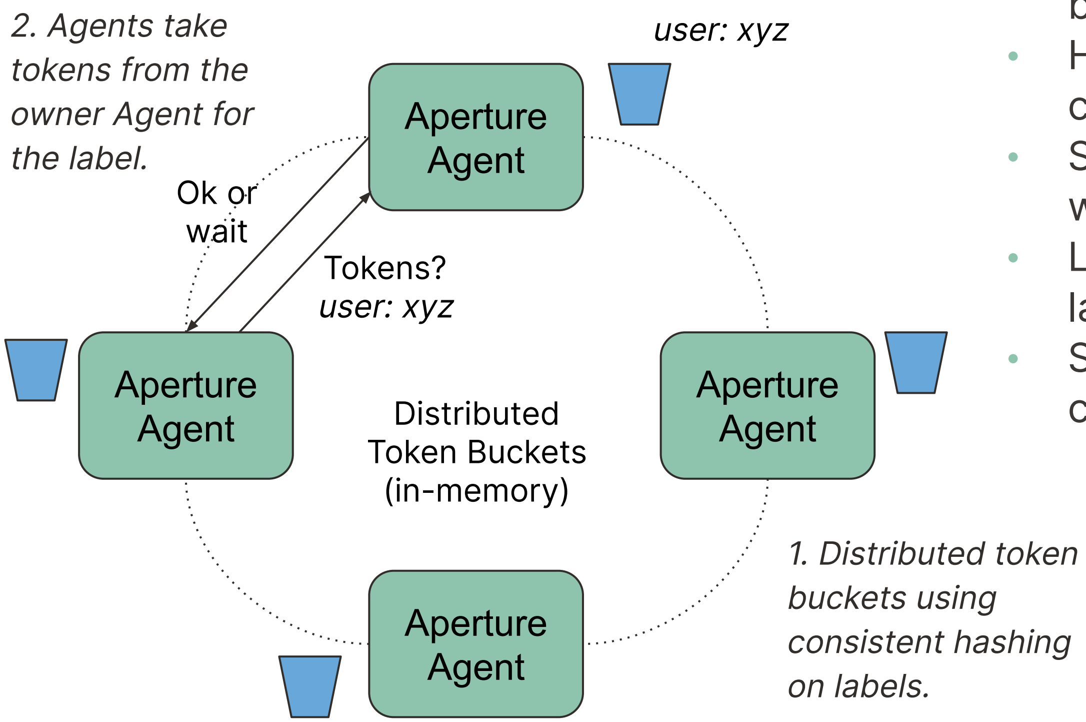
FluxNinja



Global quotas

- Service protection
  - When max capacity is known (load testing)
  - Allocate/enforce exact quotas (rps) with other services
- Managing external API rate limits
  - External services such as OpenAI, GitHub, DynamoDB etc. have rate limits. Clients must honor the limit in order to prioritize requests
  - Control costs by preventing accidental overuse
- Preventing abuse
  - Rate-limit external clients based on per-user or per-device quotas

Global quotas in Aperture



- Aperture provides consistent-hashing based global token buckets
- High performance compared to centralized Redis based system
- Smooth load compared to fixed window rate limiting
- Lazy sync (optional) for even lower latencies
- Schedule (prioritize) requests when capacity is reached

Quota scheduler policy component

FluxNinja

```yaml
circuit:
  components:
    - flow_control:
        quota_scheduler:
          in_ports:
            bucket_capacity:
              constant_signal:
                value: 500
            fill_amount:
              constant_signal:
                value: 25
          rate_limiter:
            interval: 1s
            label_key: http.request.header.api_key
          scheduler:
            workloads:
              - label_matcher:
                  match_labels:
                    http.request.header.user_type: guest
                parameters:
                  priority: 50
              - label_matcher:
                  match_labels:
                    http.request.header.user_type: subscriber
                parameters:
                  priority: 200
          selectors:
            - control_point: ingress
              service: service1-demo-app.demoapp.svc.cluster.local
```

Quotas are expressed as -

- *Bucket capacity (for allowing bursts) - e.g. 500 requests*
- *Fill amount and interval - e.g. 25 request per second*
- *Label key - Buckets are created for each key/value pair, e.g. users, services, API keys*

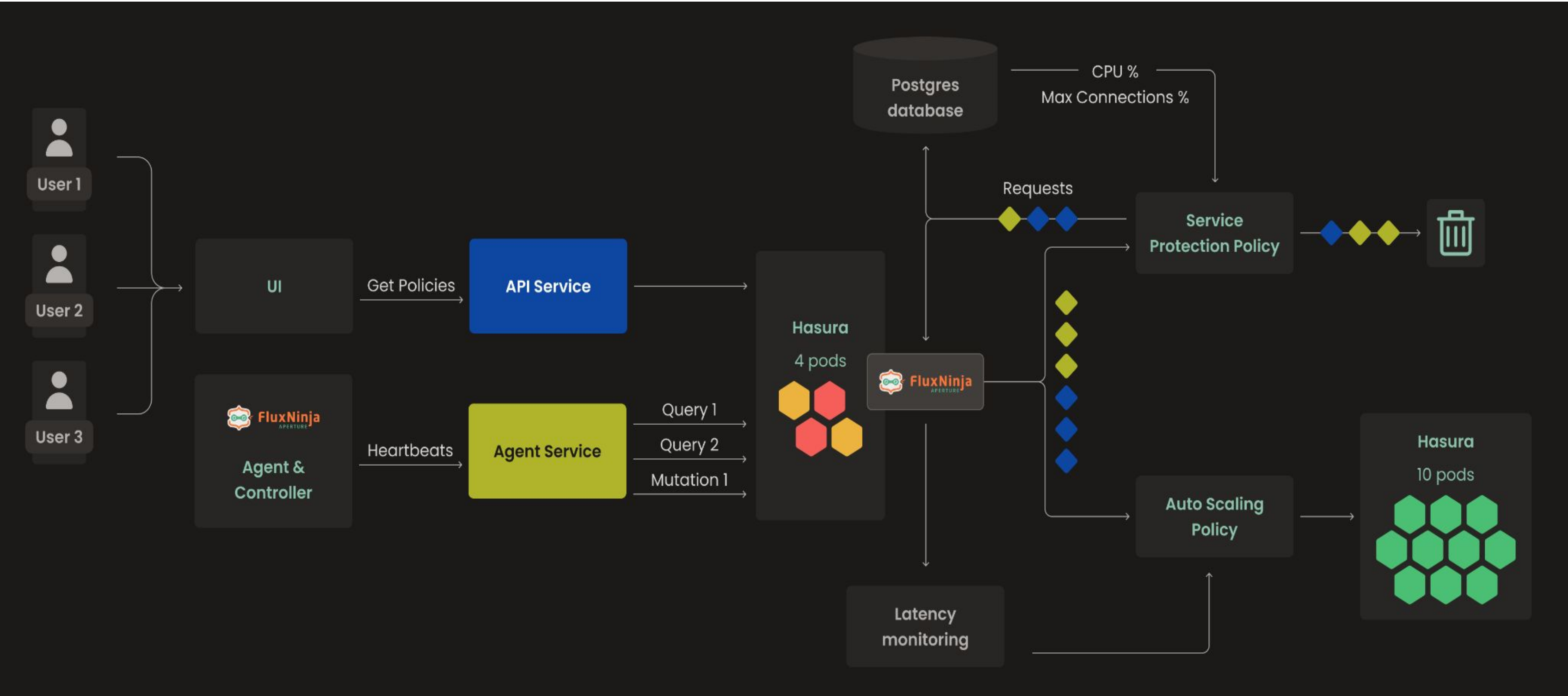
Workloads are defined by matching labels and assigning priorities

Selectors determine agents where this scheduler will be configured

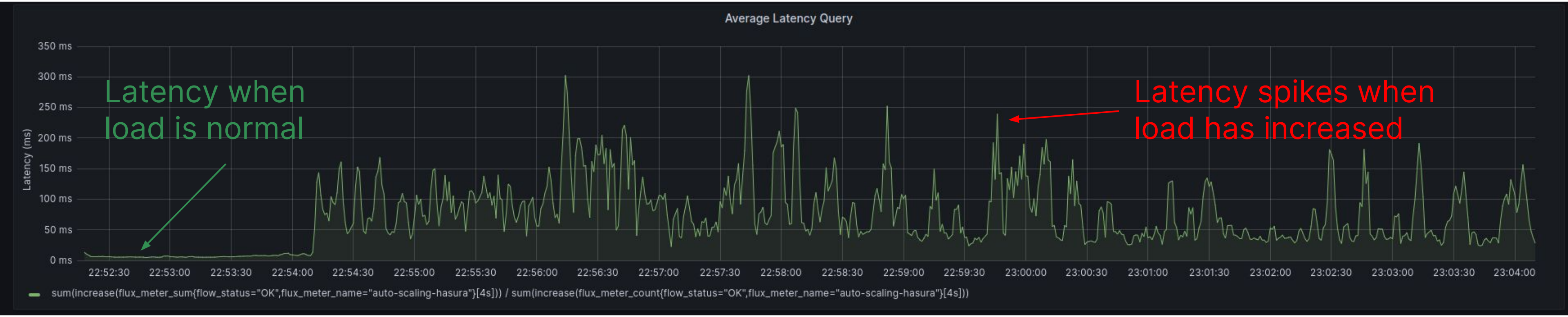
Aperture in FluxNinja ARC

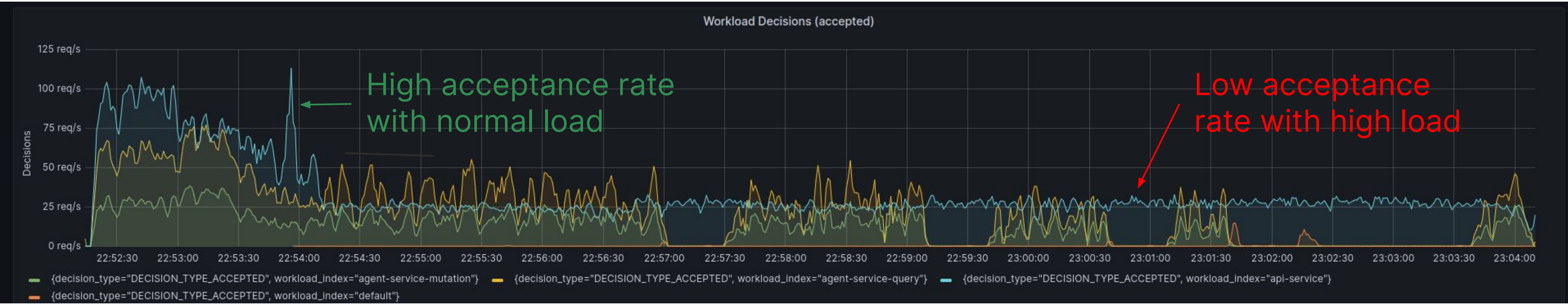
Protecting PostgreSQL by scheduling GraphQL APIs

Protecting PostgreSQL



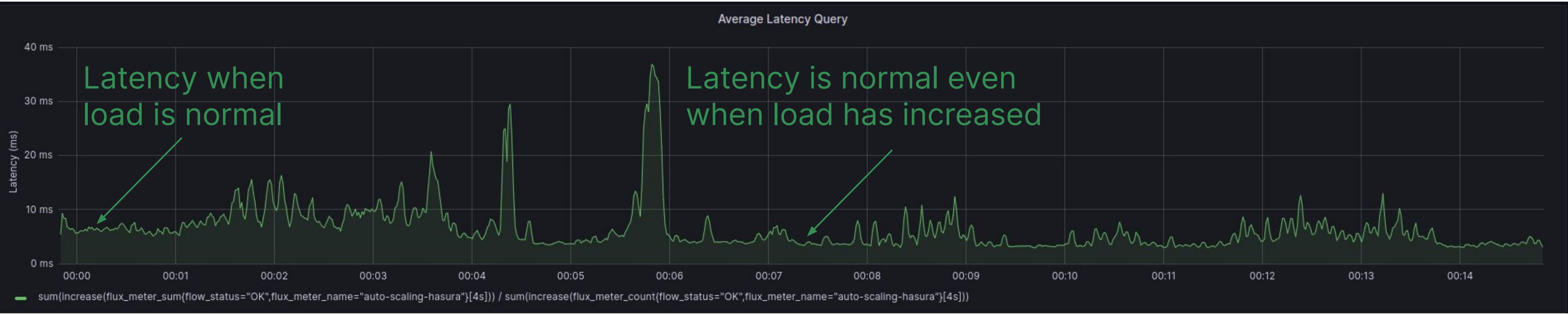
Without Aperture

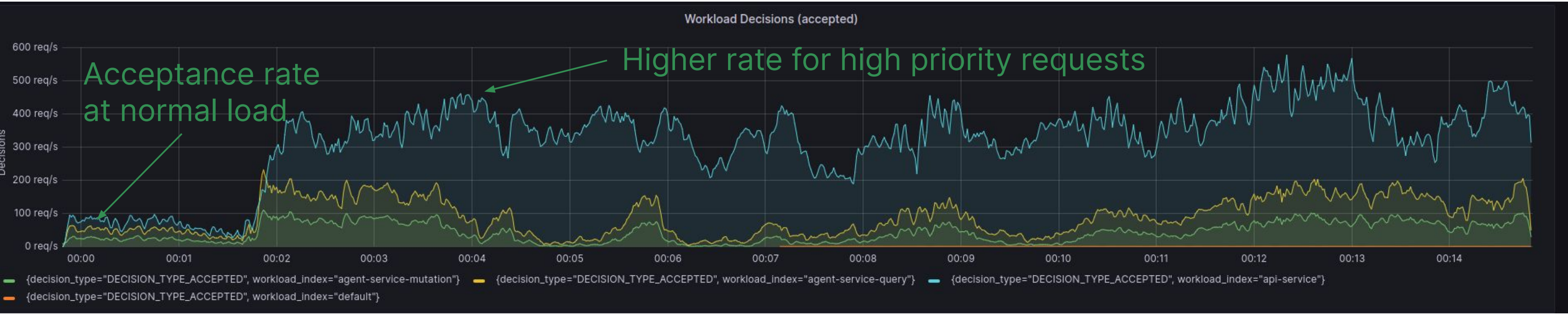




FluxNinja

With Aperture





Q & A

- Aperture project on GitHub: https://github.com/fluxninja/aperture
- Aperture Docs: https://docs.fluxninja.com/docs
- Early access to FluxNinja ARC: https://app.fluxninja.com/sign-in