AWS re:Invent

DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

NFX301

How Netflix handles sudden load spikes in the cloud

Rob Gulewich
(he/him)
Principal Software Engineer
Netflix

Ryan Schroeder
(he/him)
Staff Software Engineer
Netflix

Joseph Lynch
(he/him)
Principal Software Engineer
Netflix

Manju Prasad
(she/her)
Sr. Solutions Architect
Amazon

Rob Gulewich
Principal Software Engineer
Netflix Platform

Ryan Schroeder
Staff Software Engineer
Netflix Reliability

Joseph Lynch
Principal Software Engineer
Netflix Data Platform

Agenda

01 Problem: Load spikes

02 Solution: Predict and plan

03 Solution: React quickly

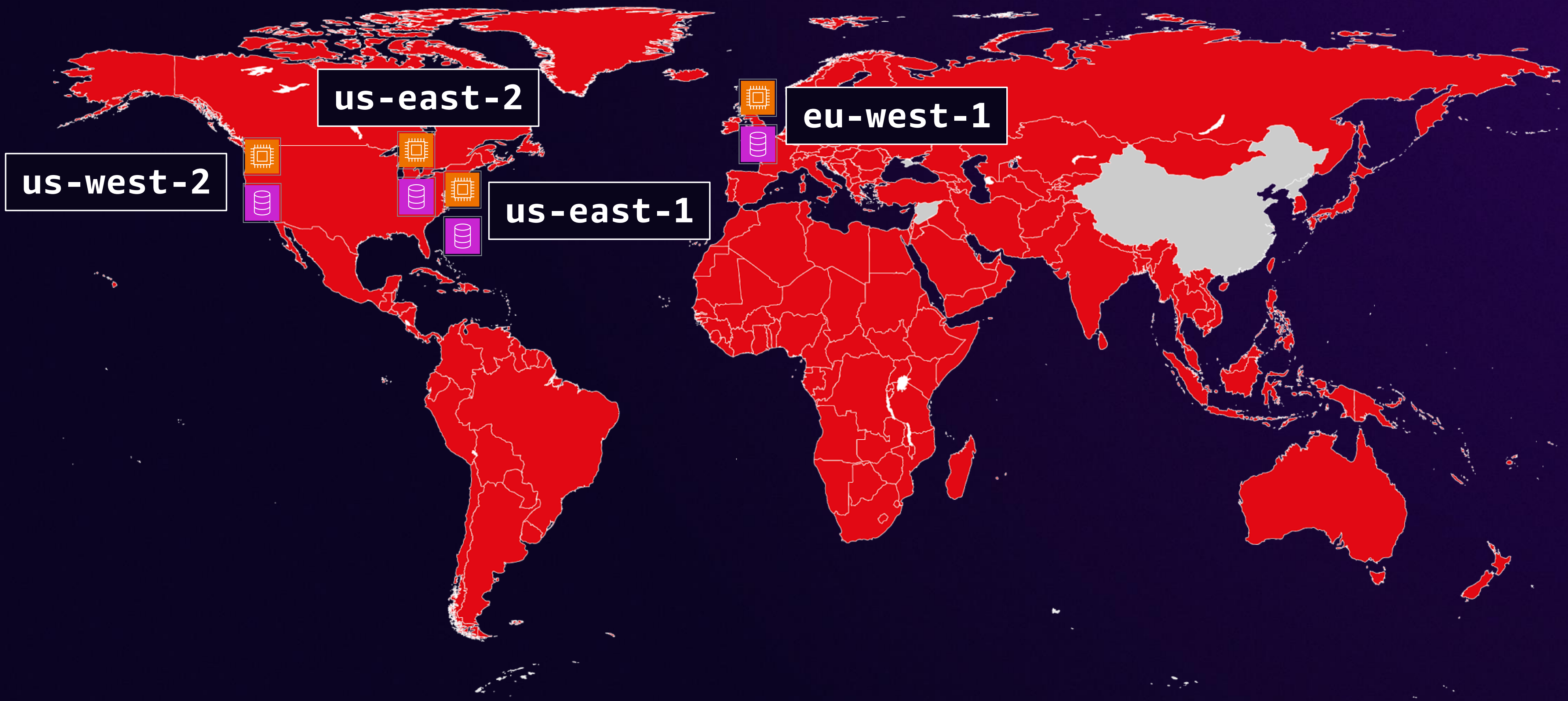
04 Solution: Stay available

05 Experiment: Test resilience

06 Conclusions and wrap up

01: Problem
Load spikes at Netflix

Netflix cloud topology



Gradual traffic increases are the norm

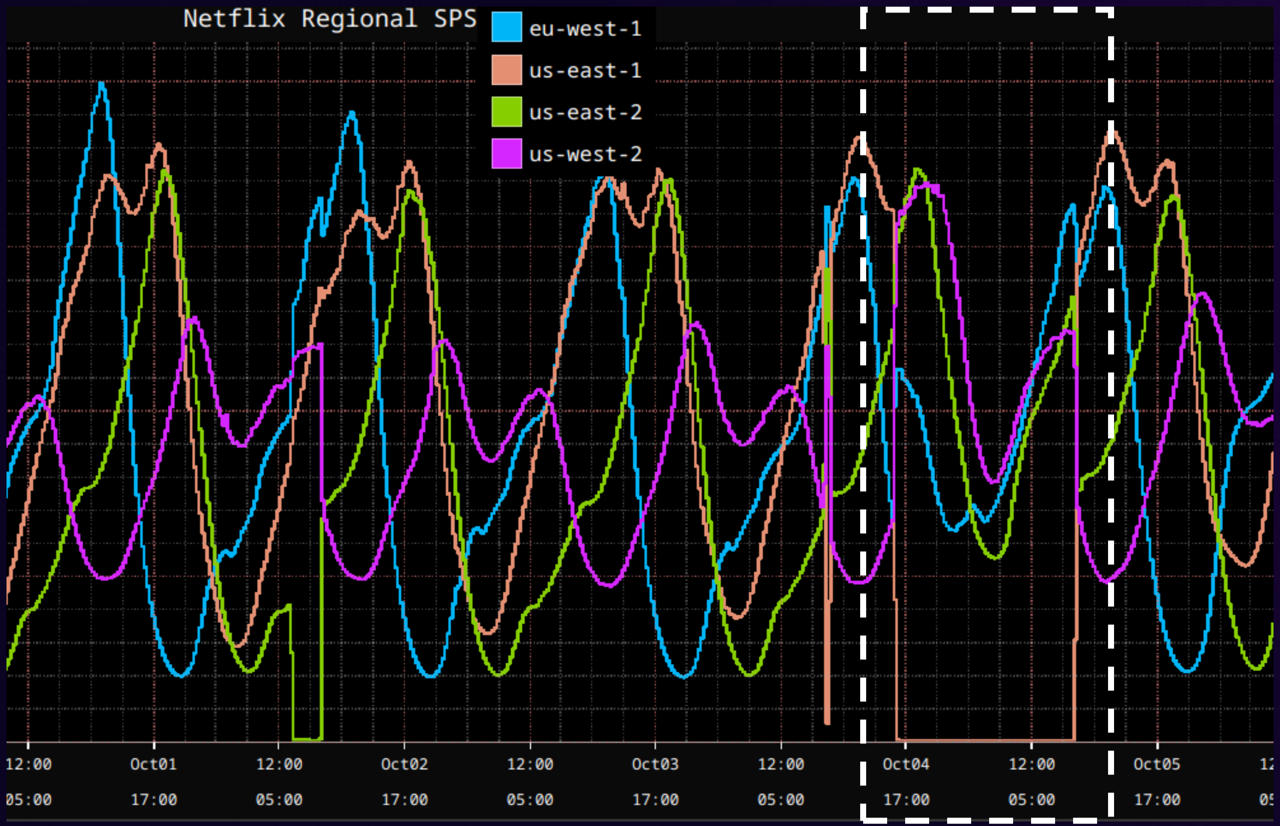
~10x peak trough

Starts per second (SPS)

24-hour periodicity

Traffic phase shifts

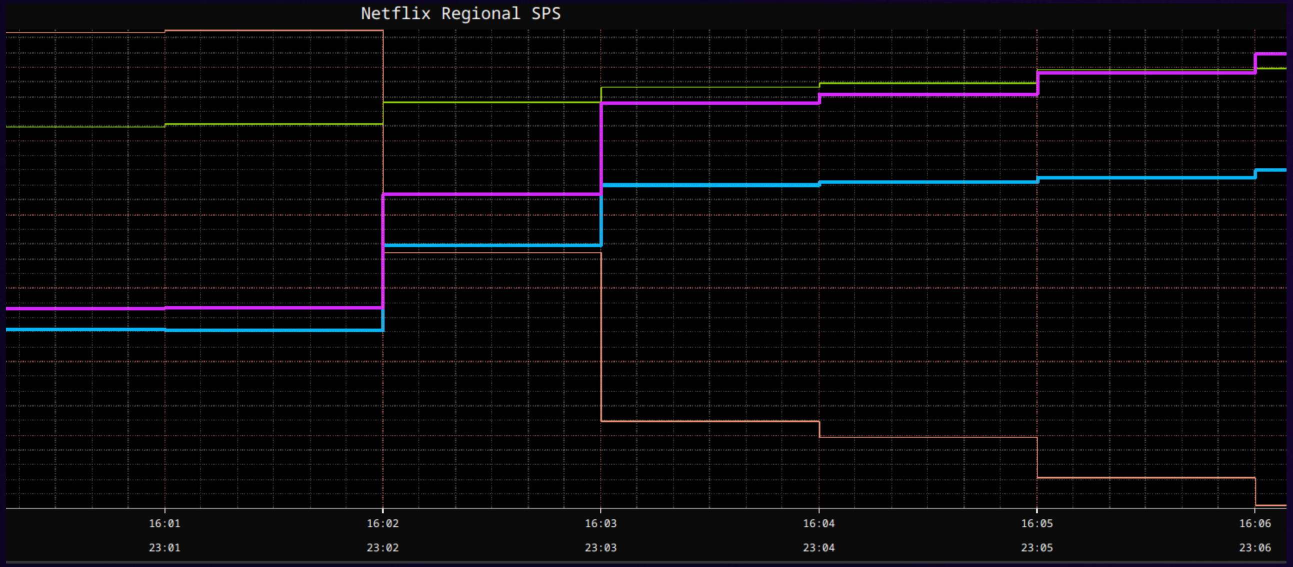
Load spikes are common

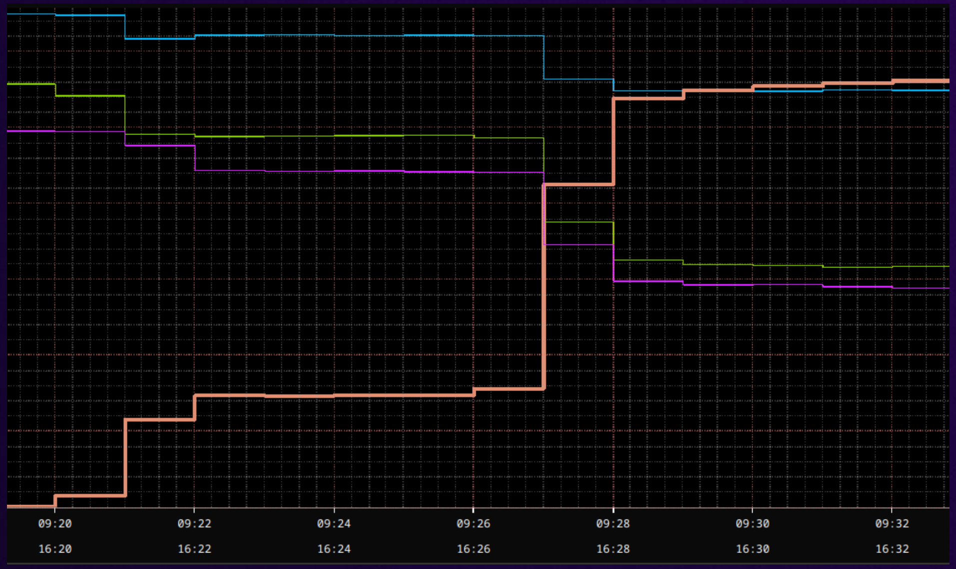


Failovers due to:

- Regular practice
- Bad software deployment
- Regional impairment

Fast failover is a load spike





On Evacuation

Up to ***2x traffic to saviors (variable) in 1 minute***

Long tail traffic takes ~5 minutes

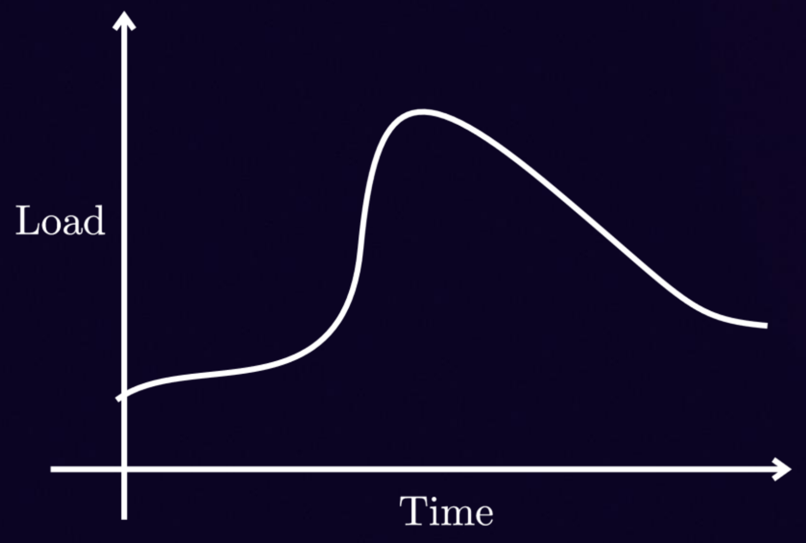
Intelligent steering to minimize spike

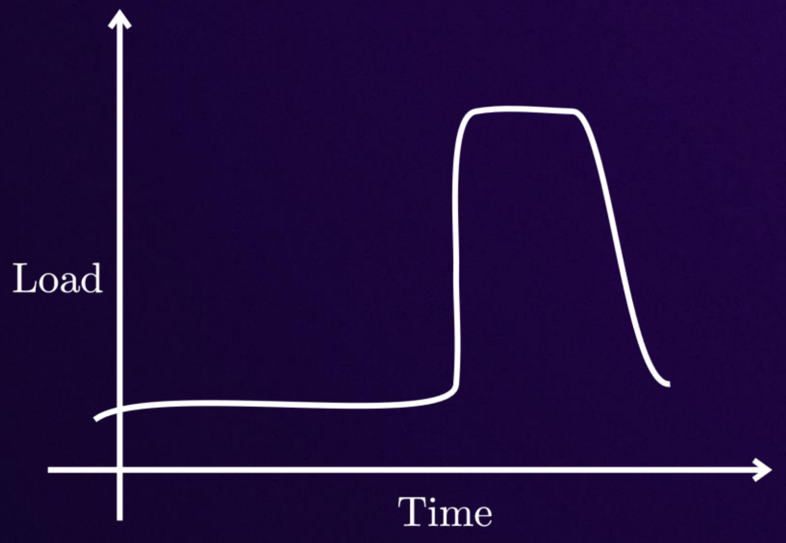
On Restore

Up to ***100x traffic to evacuated***

Reintroduce traffic over 5-10 minutes

Sources of unexpected traffic surges





Long surges

Title launches
External events (soccer matches, other sites down)

Short spikes

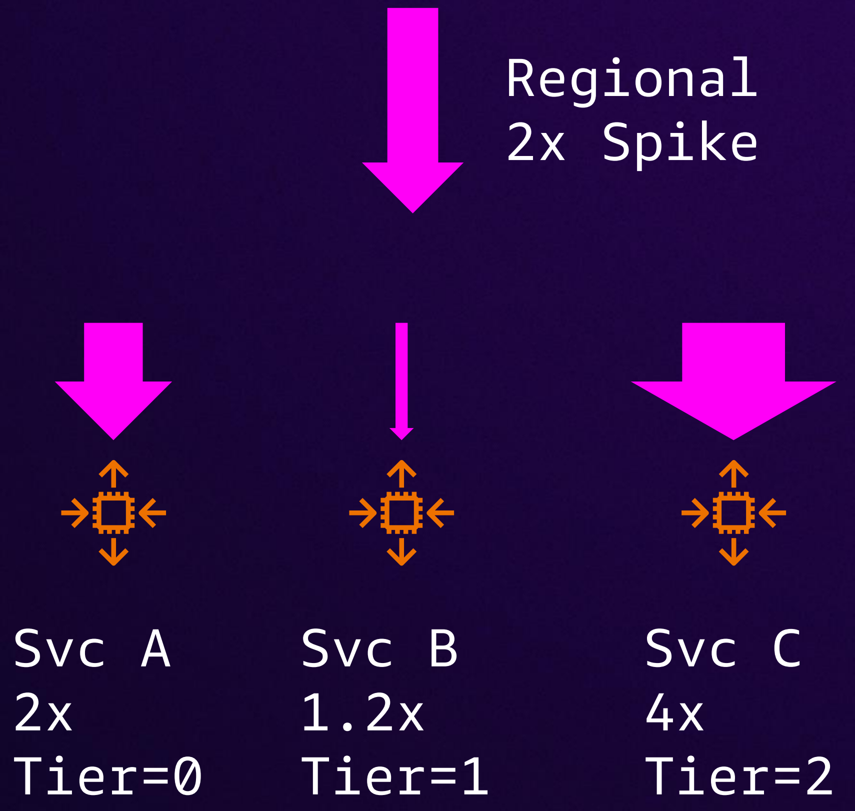
Retry storms
Device bugs

Thousands of microservices – Complex downstream call graph

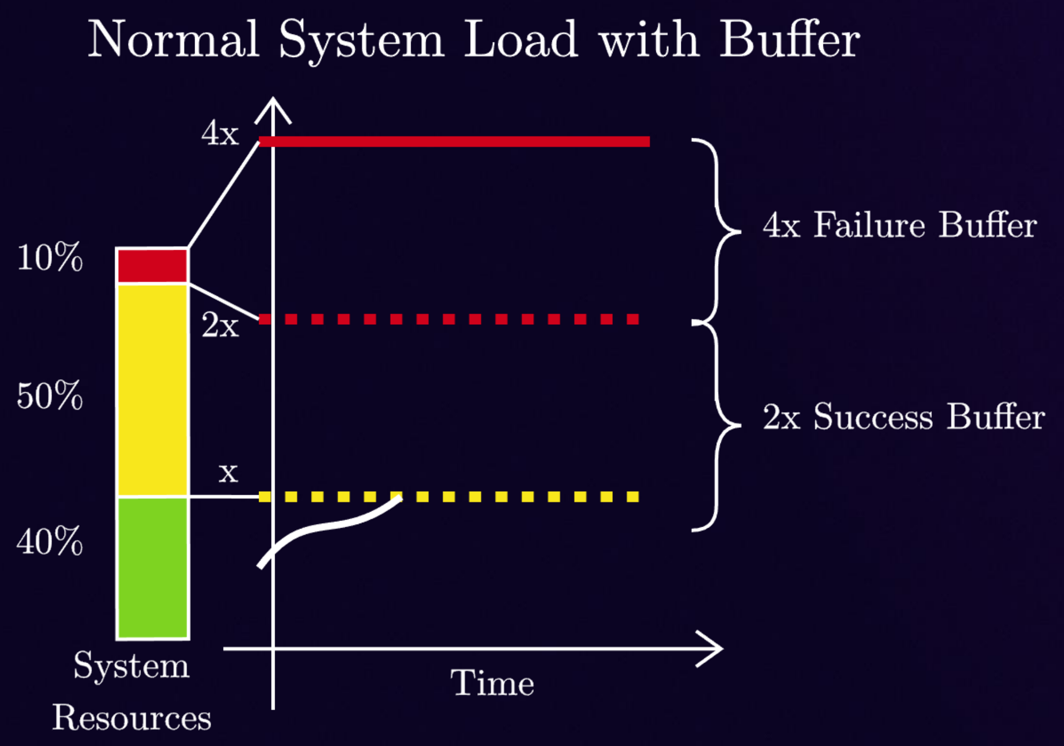
Microservice-Specific Regional Demand

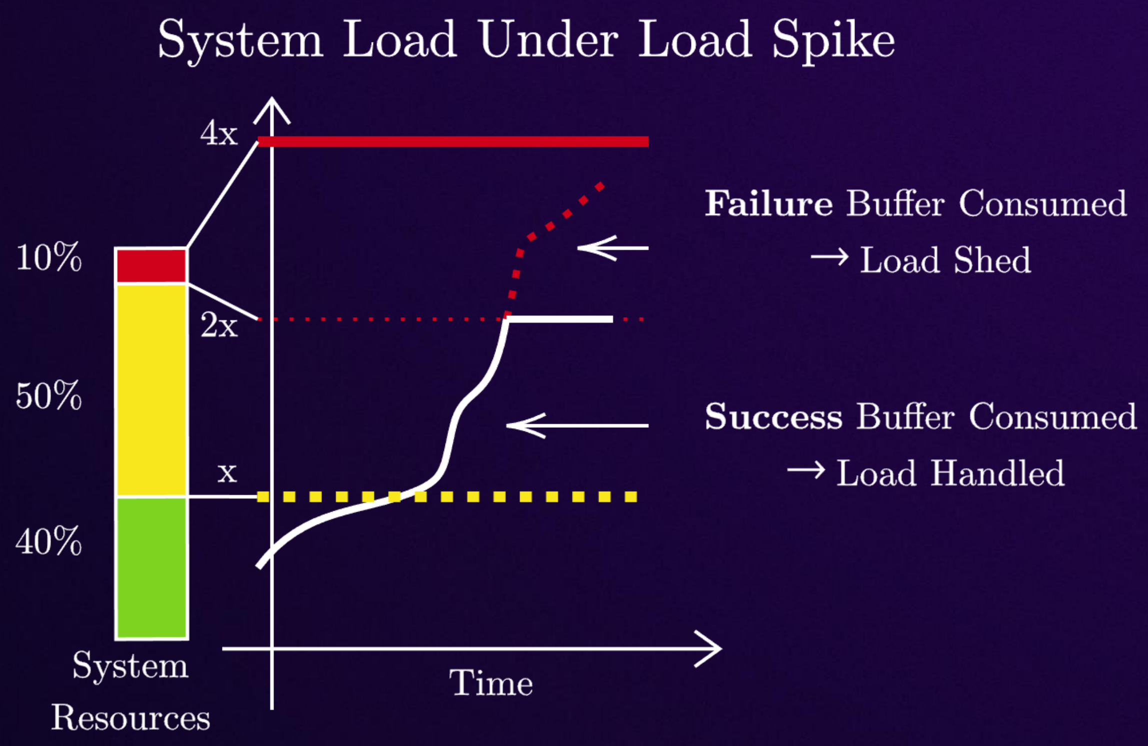
Because of service decomposition, we understood that using a proxy demand metric like SPS wasn't tenable and we needed to transition to microservice-specific demand. Unfortunately, due to the diversity of services, a mix of Java (Governator/Springboot with Ribbon/gRPC, etc.) and Node (NodeQuark), there wasn't a single demand metric we could rely on to cover all use cases. To address this, we built a system that allows us to associate each microservice with metrics that represent their demand.

Blog post:



Thousands of microservices – Different headroom

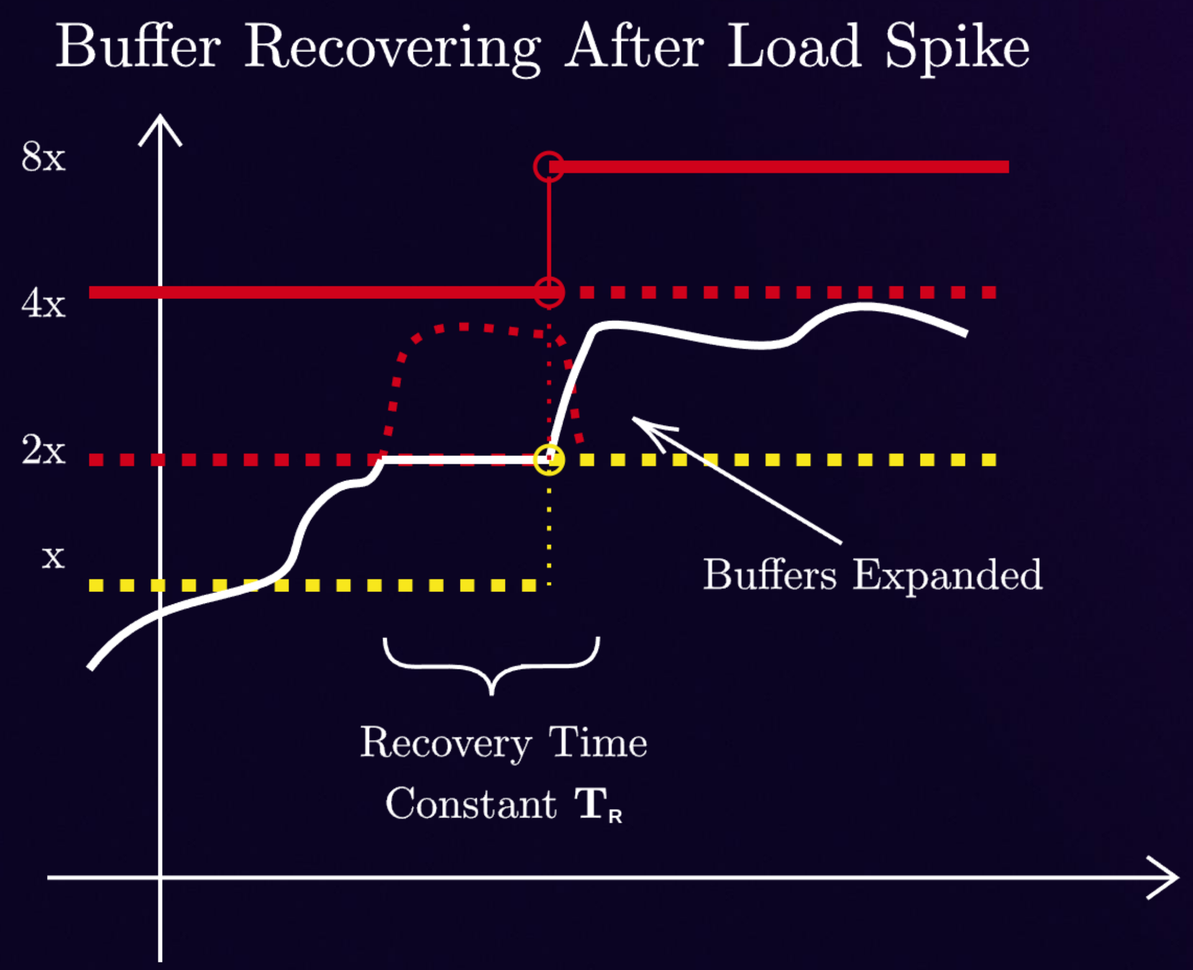




Every service operates with two key **Buffers**

Success buffer Headroom before errors (bad)
Failure buffer Headroom before congestive collapse (very bad)

Our business is evolving



Changing business needs:

- More frequent big title launches
- More global launches

Goals:

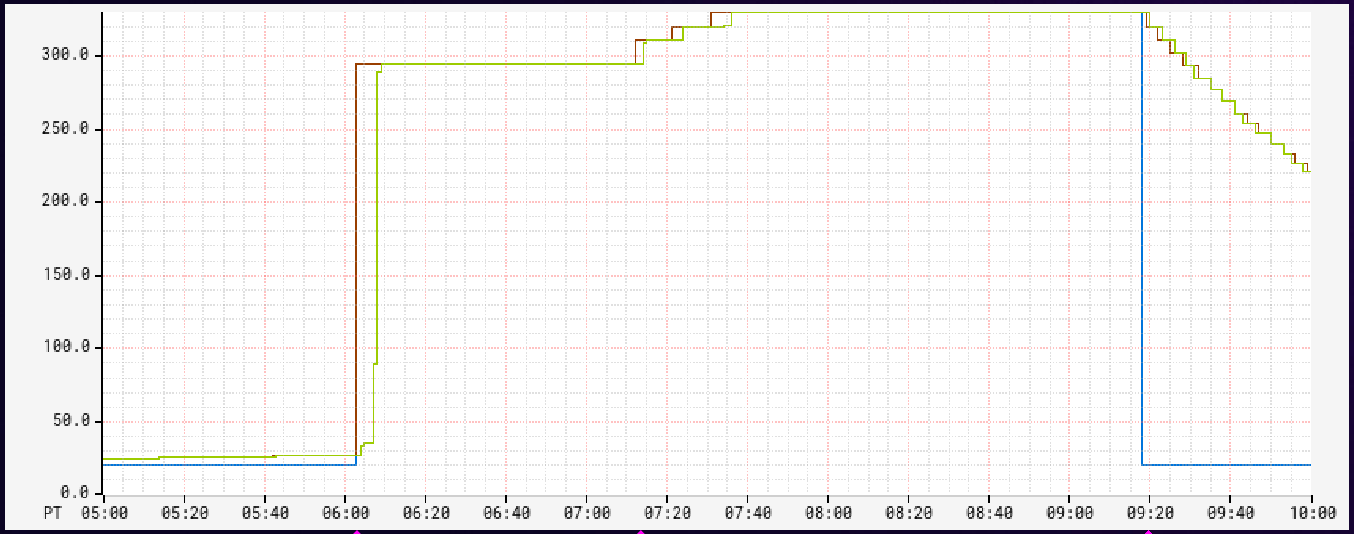
- Reduce time to recover
- Use regional failover less as the primary remediation
- Build resiliency assuming load spikes are the norm

02: Capacity plan

Load is often predictable

Scale on a schedule

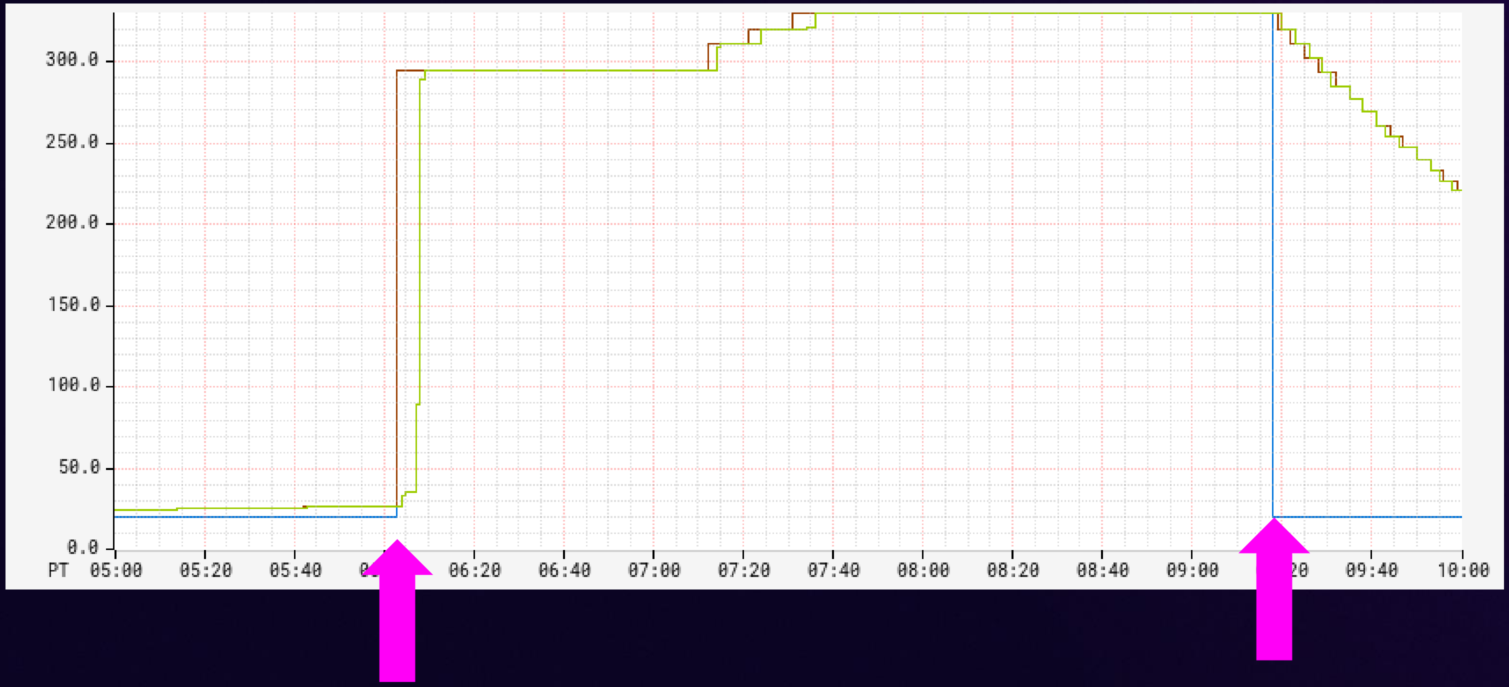
- If we know when traffic is going to arrive, prescale services beforehand to match the predicted load
- Autoscaling is designed for reactive scaling of individual services



Prescale **Spike** **Downscale**

Prescaling

- Use the failover system to scale up the entire streaming fleet
- Maps regional SPS to RPS per instance and calculates new min. sizes



Autoscaling Group:

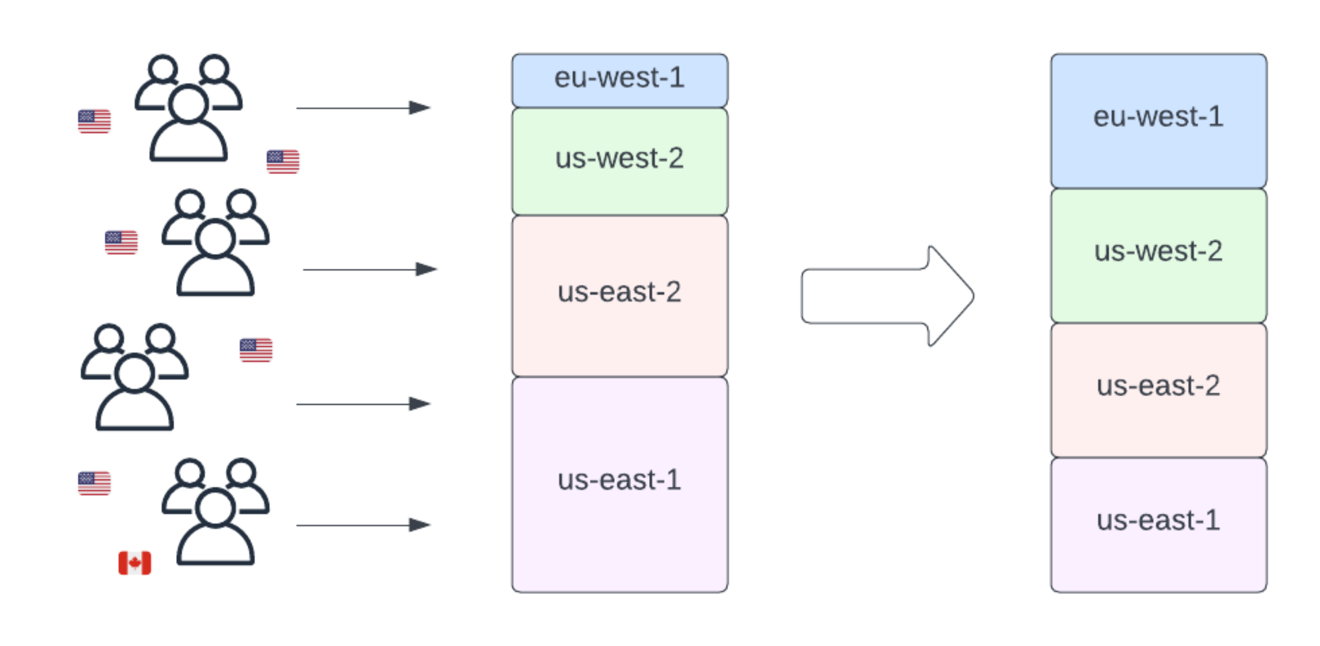
- Blue: min
- Brown: desired
- Green: instances up

Increase mins to match expected spike

Decrease mins to normal levels

Shape on a schedule

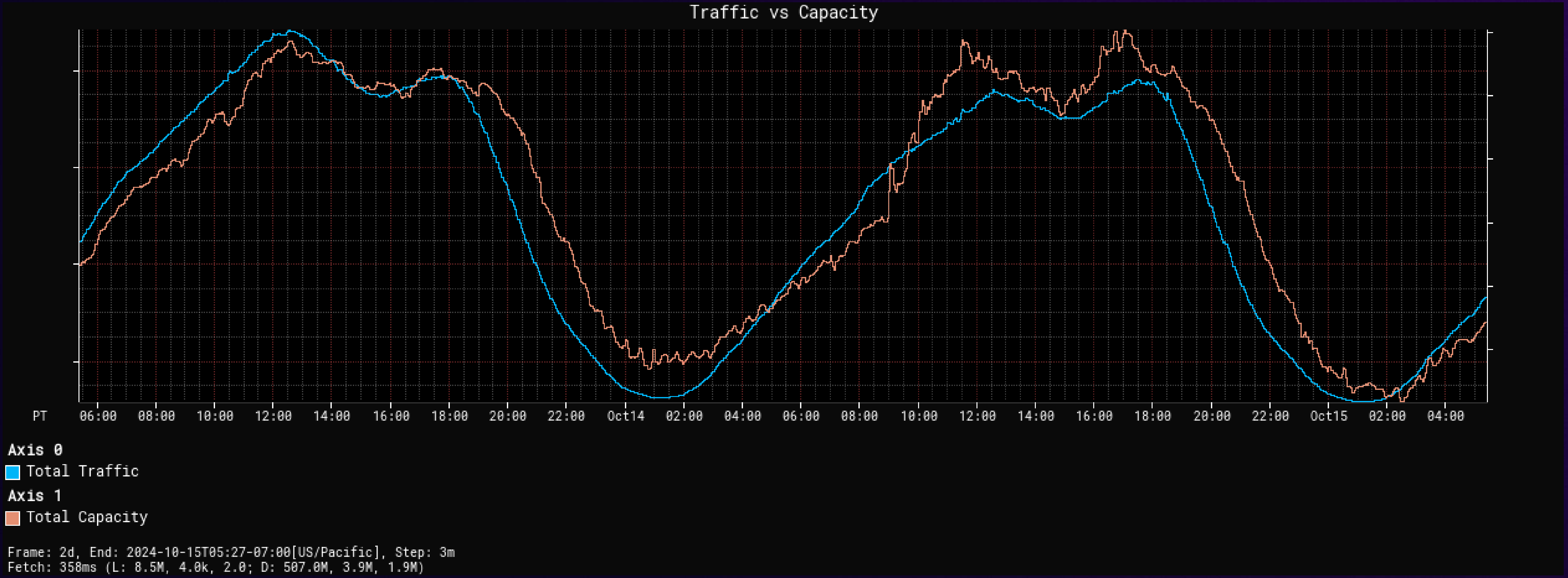
- Some title launches are centered in a specific geography
- For large launches, we can proactively steer users to other regions to balance global capacity usage



03: Scale out of trouble

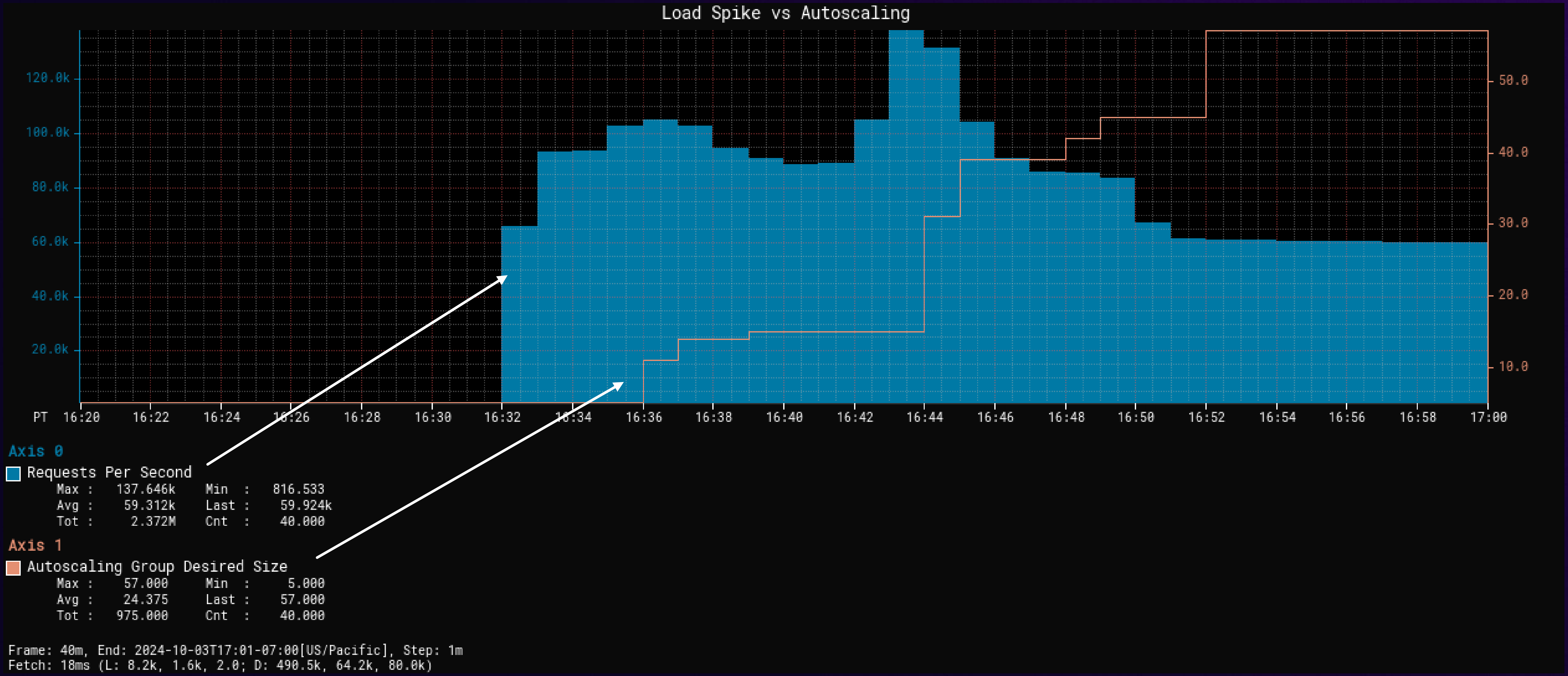
Predictions are often wrong

Autoscaling during steady state

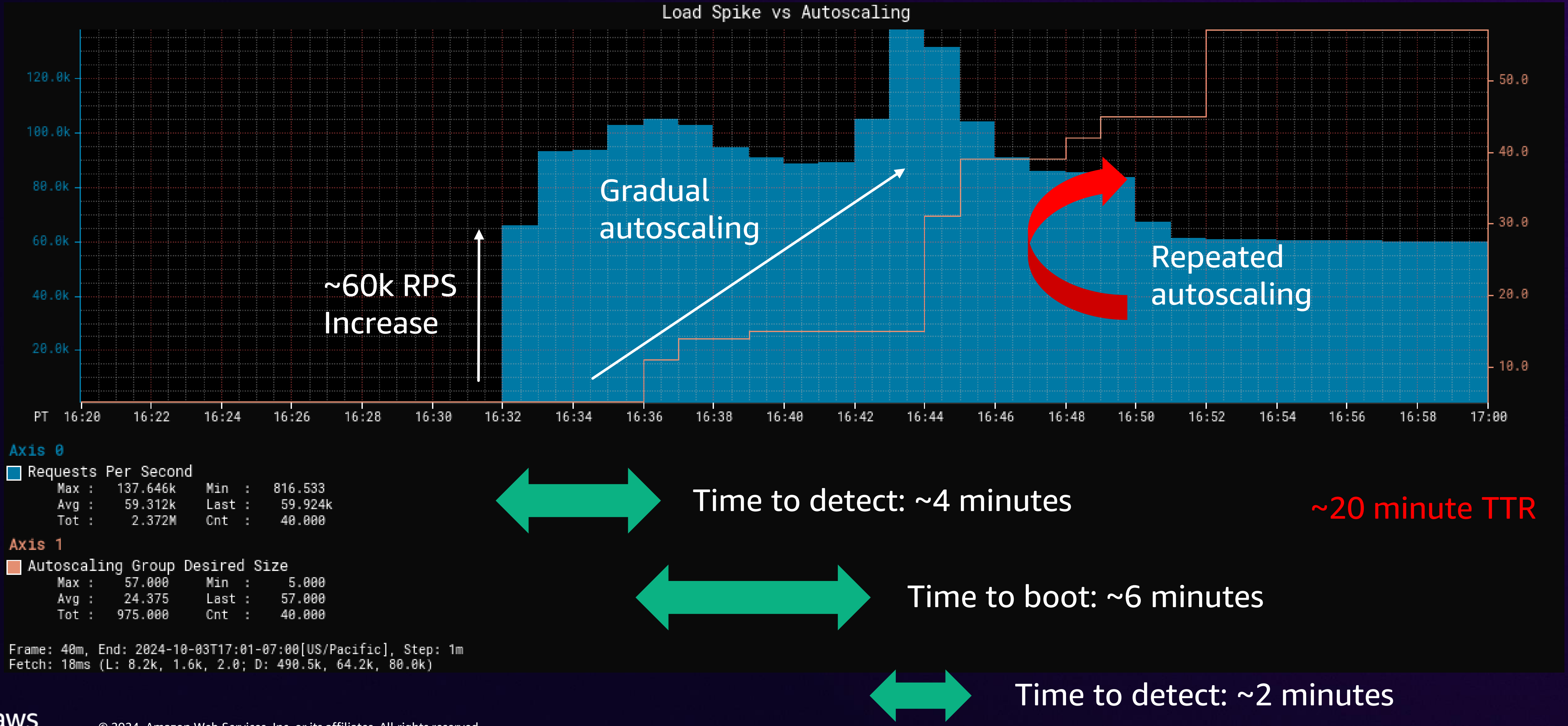


Autoscaling during steady state

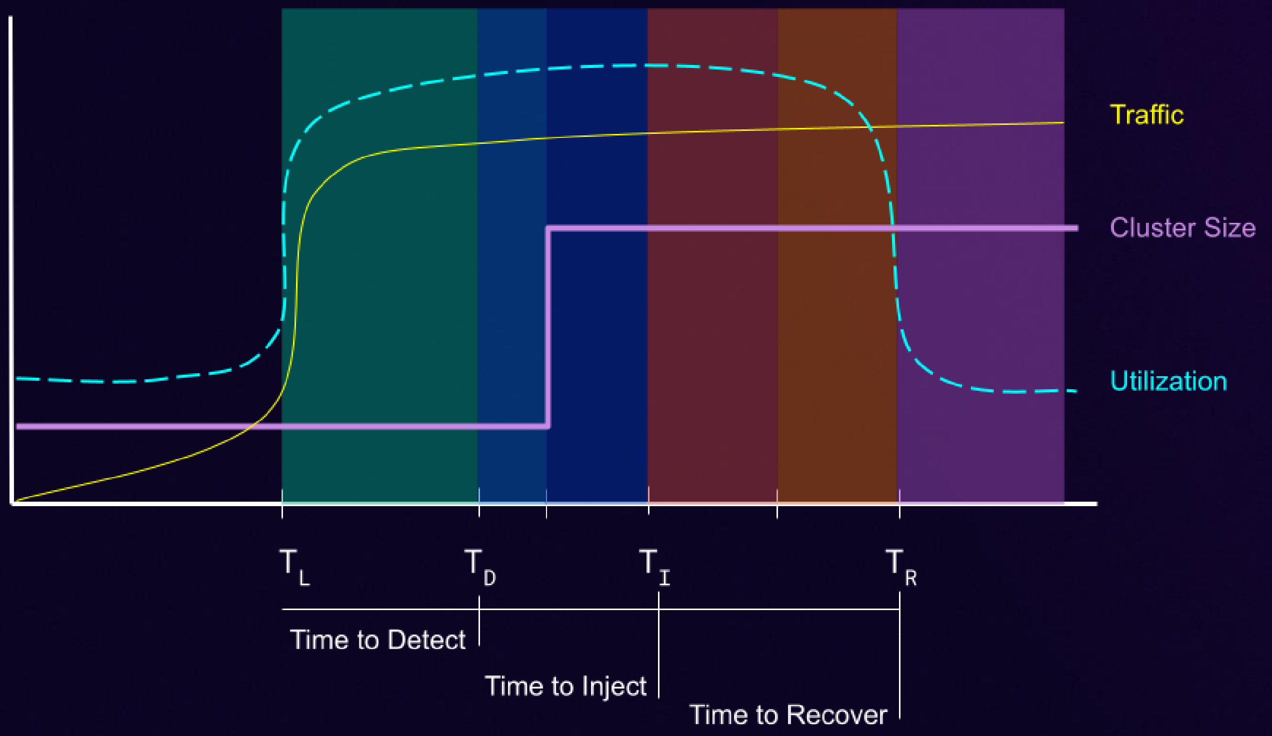
Autoscaling during load spikes



Autoscaling during load spikes



Components of time-to-recovery



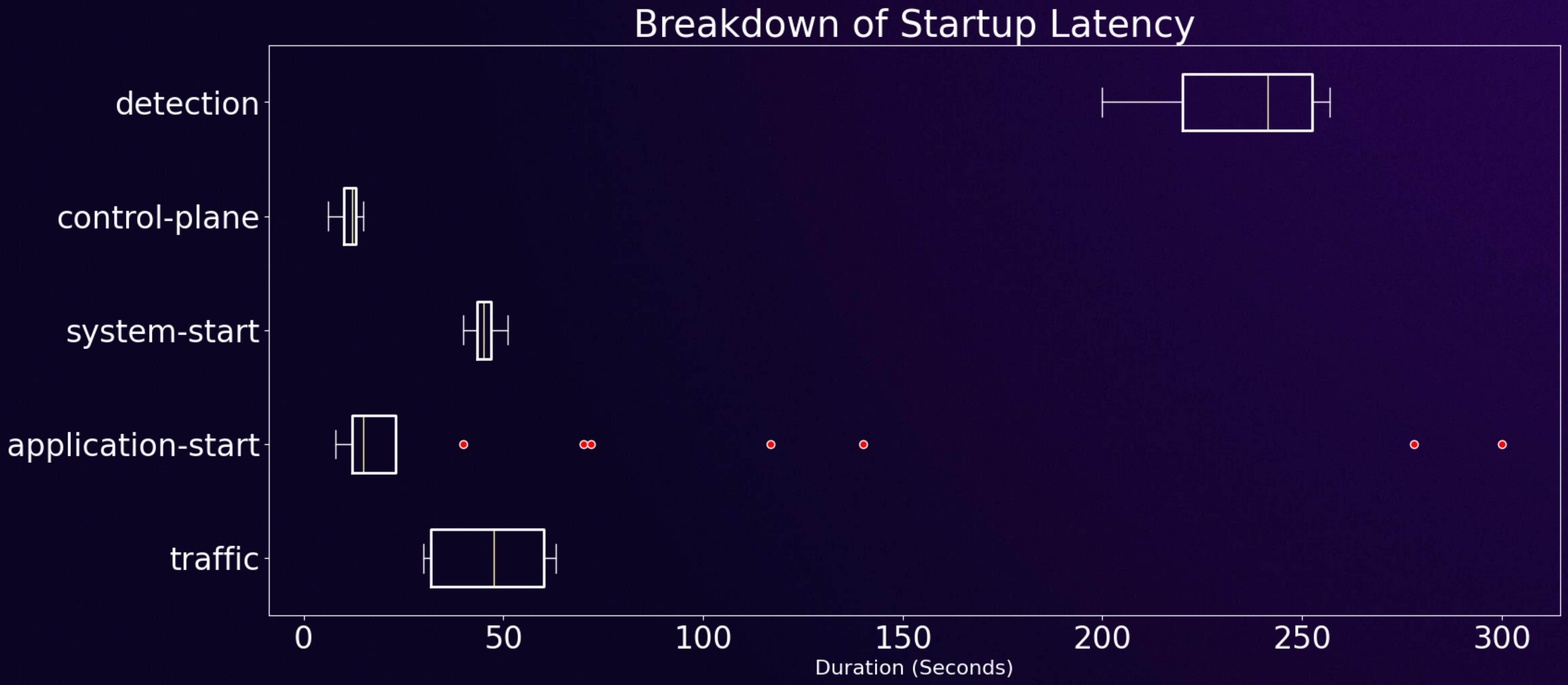
Traffic spikes at T_L causing Utilization to increase. The Cluster Size increases only after delays for Detection and Control Plane . After a delay for OS Startup, we reach the point usable capacity is injected T_I. Utilization remains high until Application Startup and Load Balancing delays allow new capacity to take traffic - then we Recover at T_R.

Stage	Description
Detection	Scaling alarm triggers
Control plane	Hardware online
System startup	Kernel and base systemd units started
Application startup	Microservice started
Traffic	Traffic arrives

Experimentation setup

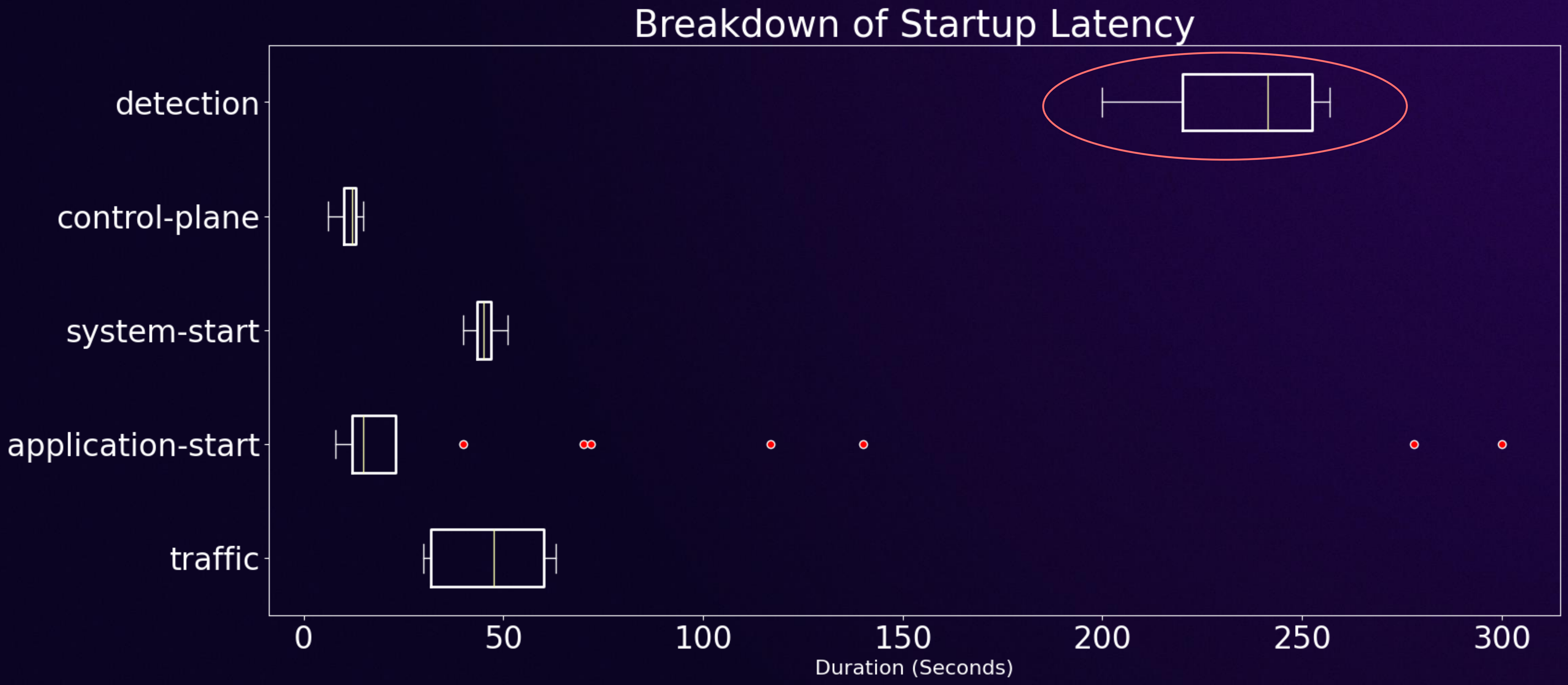
- Synthetic load generation
- Baseline vs. experiment comparison
- Variations of scaling policy configurations

Time-to-recovery dominating factors



Detection > App Startup > System Startup > Hardware Startup

Time-to-recovery dominating factors



Detection > App Startup > System Startup > Hardware Startup

Detection – Scaling on RPS

CPU target tracking is nice for gradual changes, but doesn't provide enough information for 10x spikes

- Typical CPU target is around 50% utilization
- At 2x RPS, CPU is 100%
- At 10x RPS, CPU is also 100%

During load shedding, CPU does not reflect actual workload

CPU: 0% - 100%

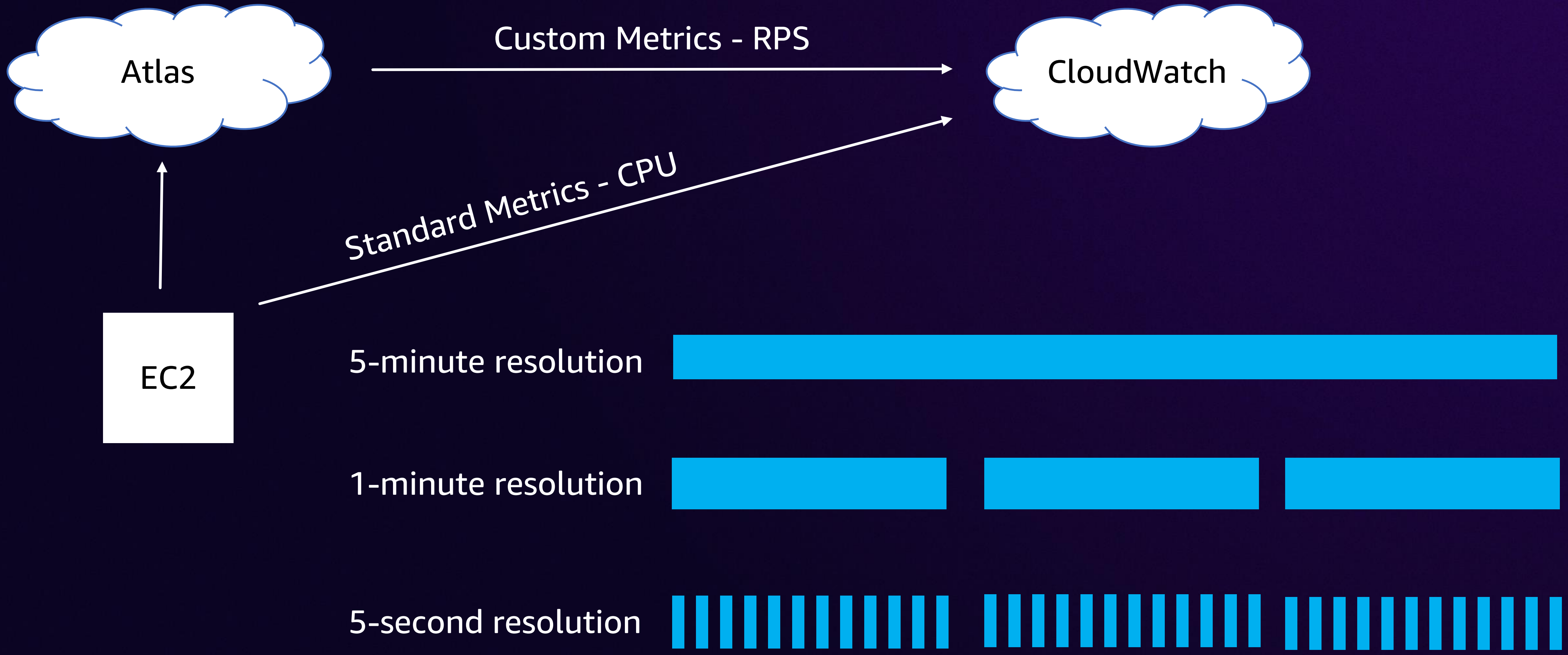
RPS: 0 - Infinity

Detection – Scaling on RPS

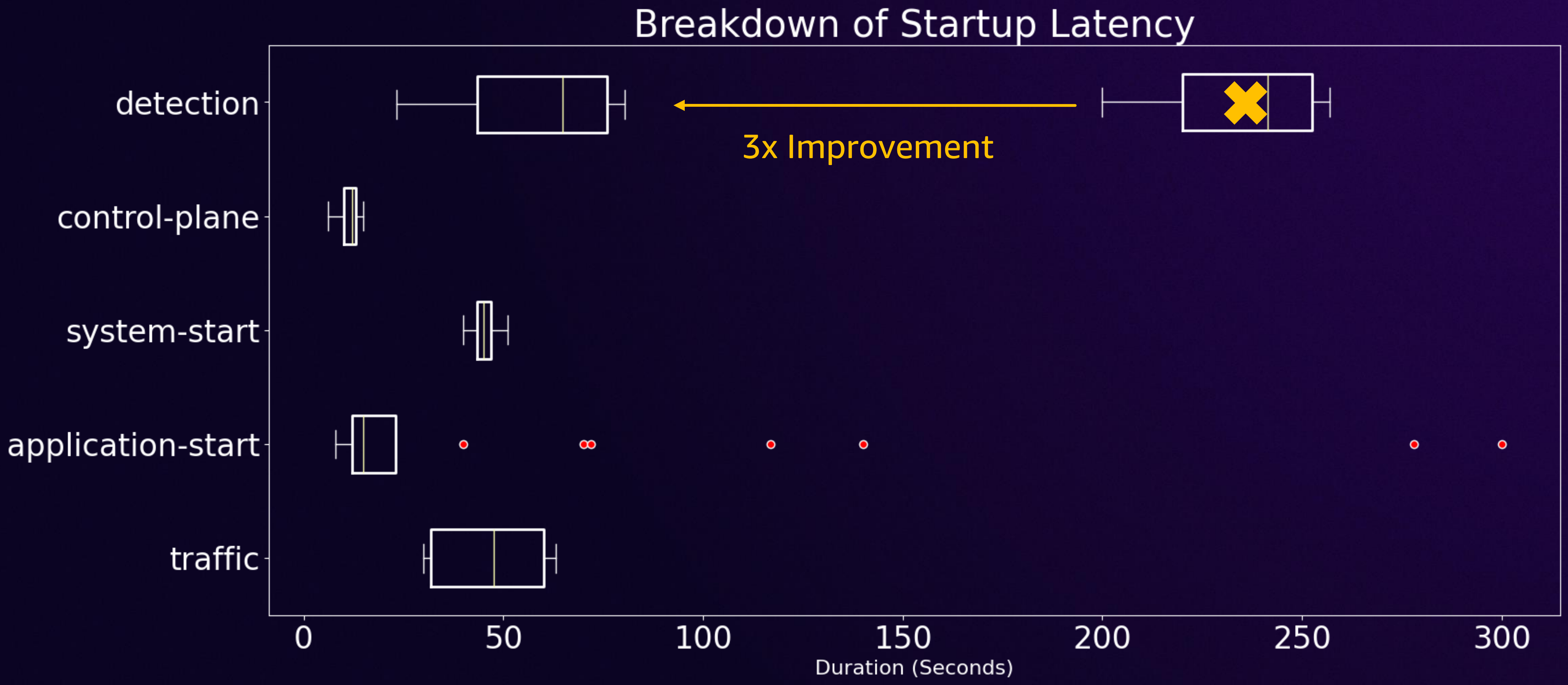
Add RPS "hammer" policy – one shot to success

- Bad – 2x scales
- Good – exactly what you need
- Bad – scale way too much

Detection – Higher resolution metrics

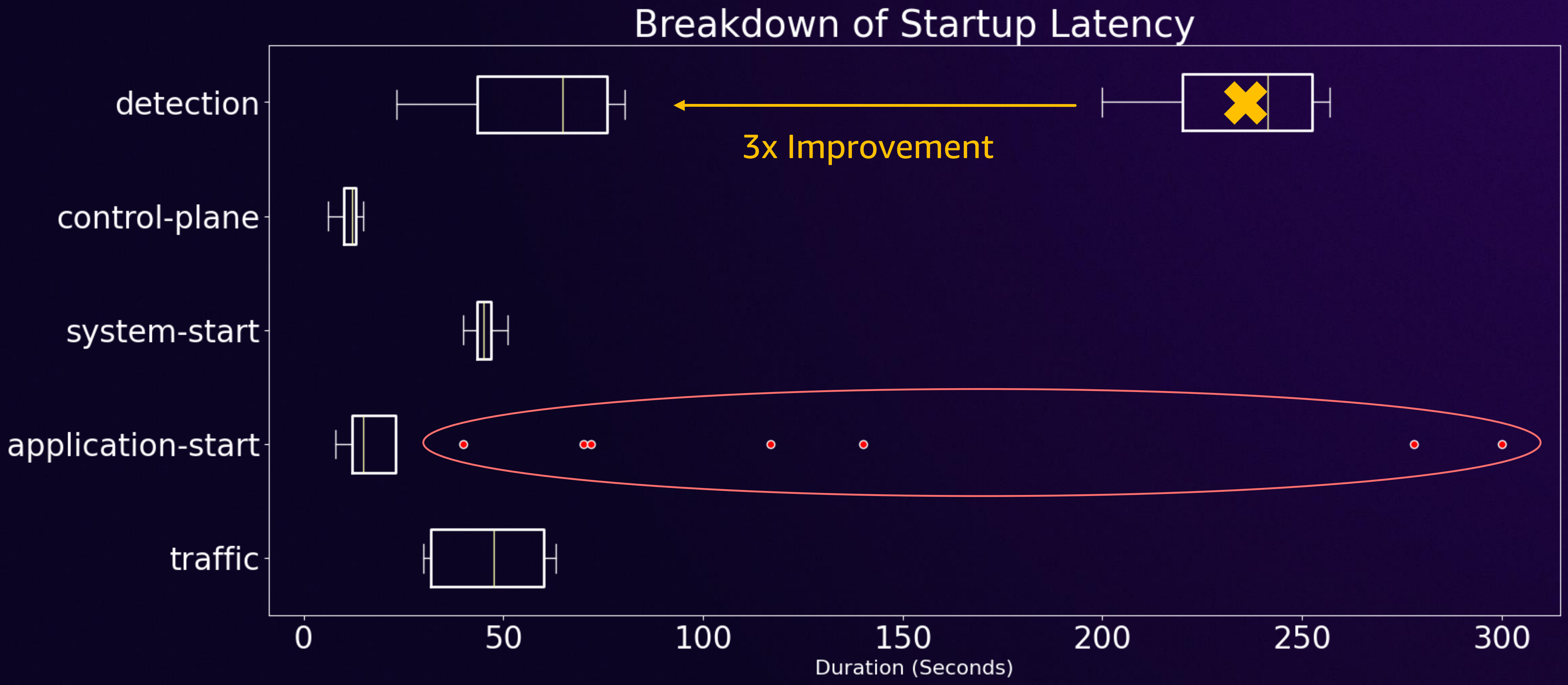


Time-to-recovery dominating factors



Detection > App startup > System startup > Hardware startup

Time-to-recovery dominating factors

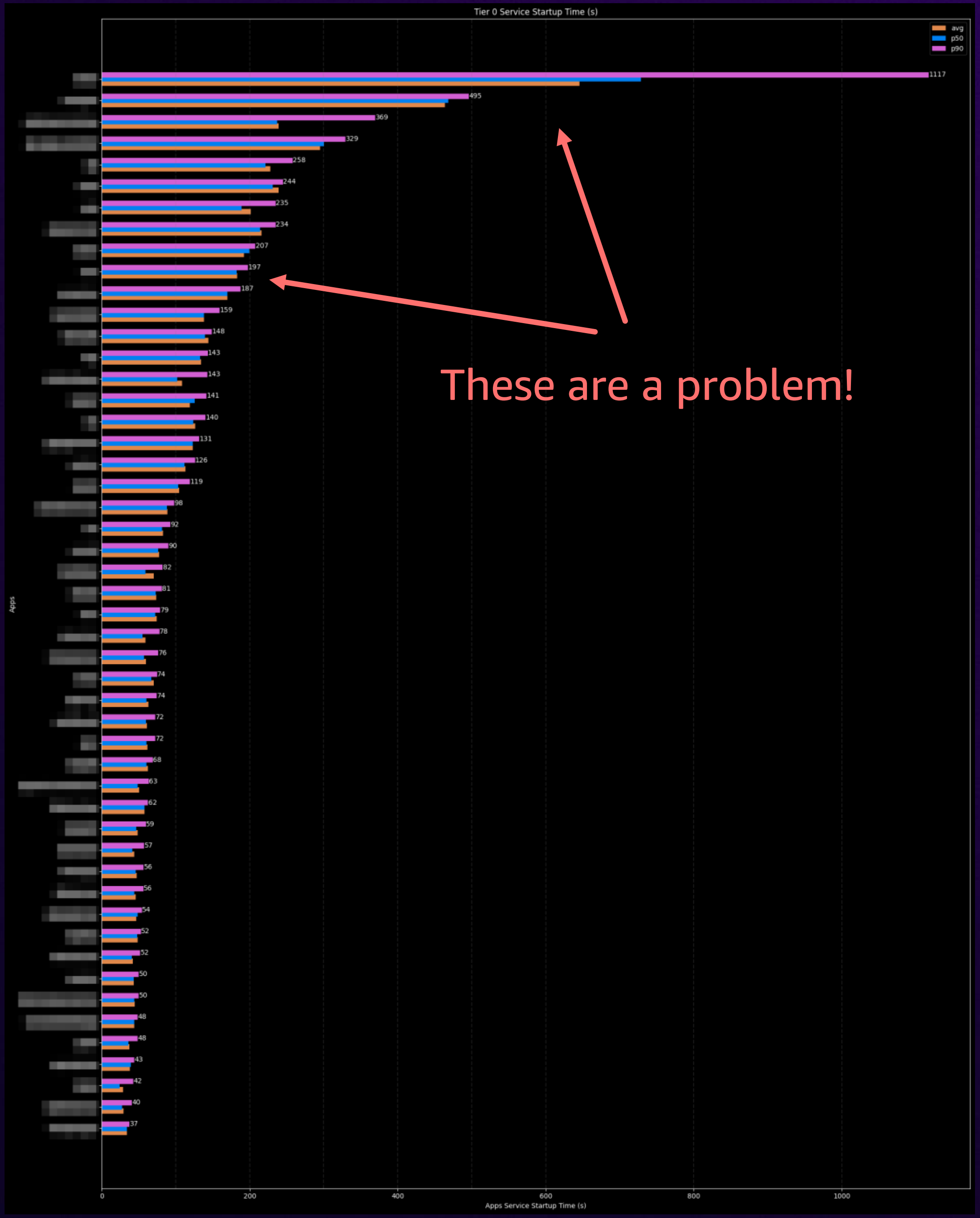


Detection > **App startup** > System startup > Hardware startup

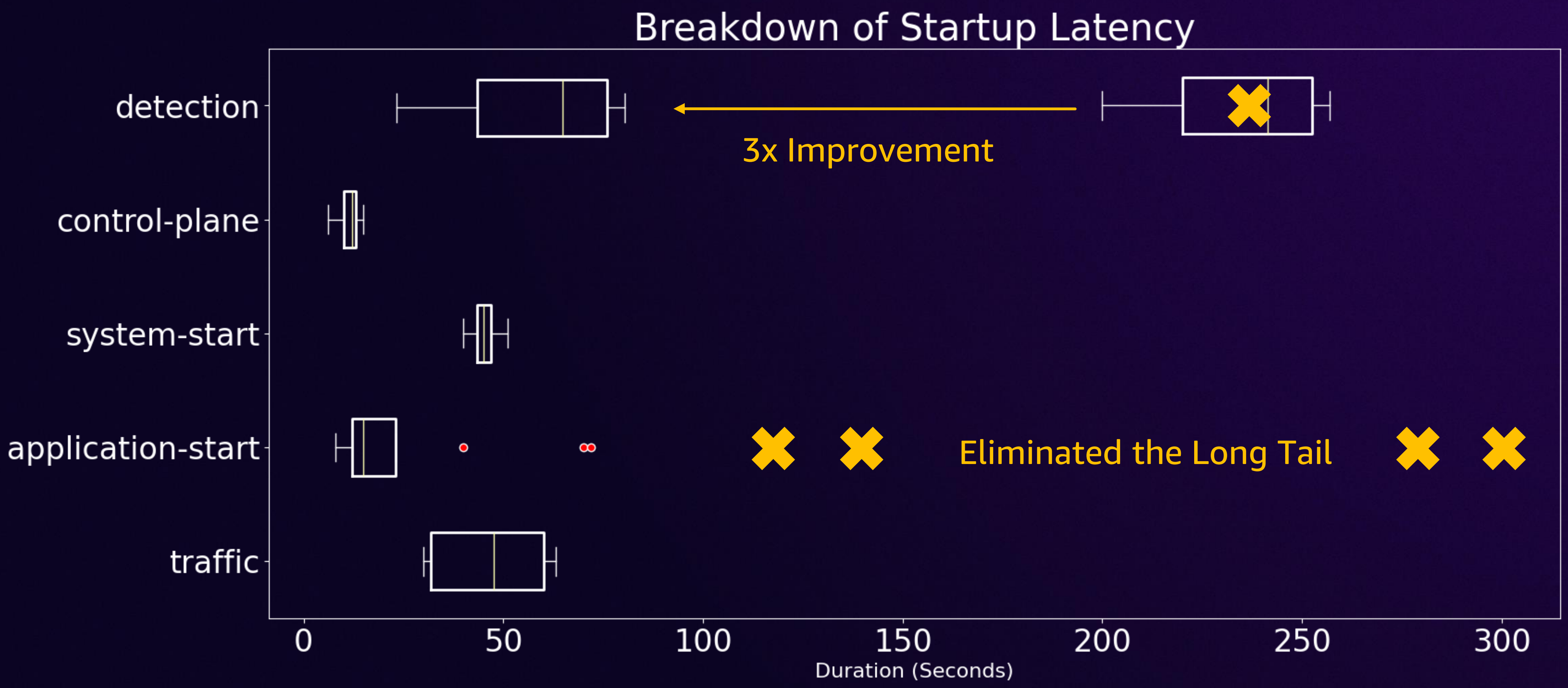
App startup has a long tail

Long tail of startup delay. Vast majority under a minute.

Worst offender took **p90 of 18 *minutes*** to start!

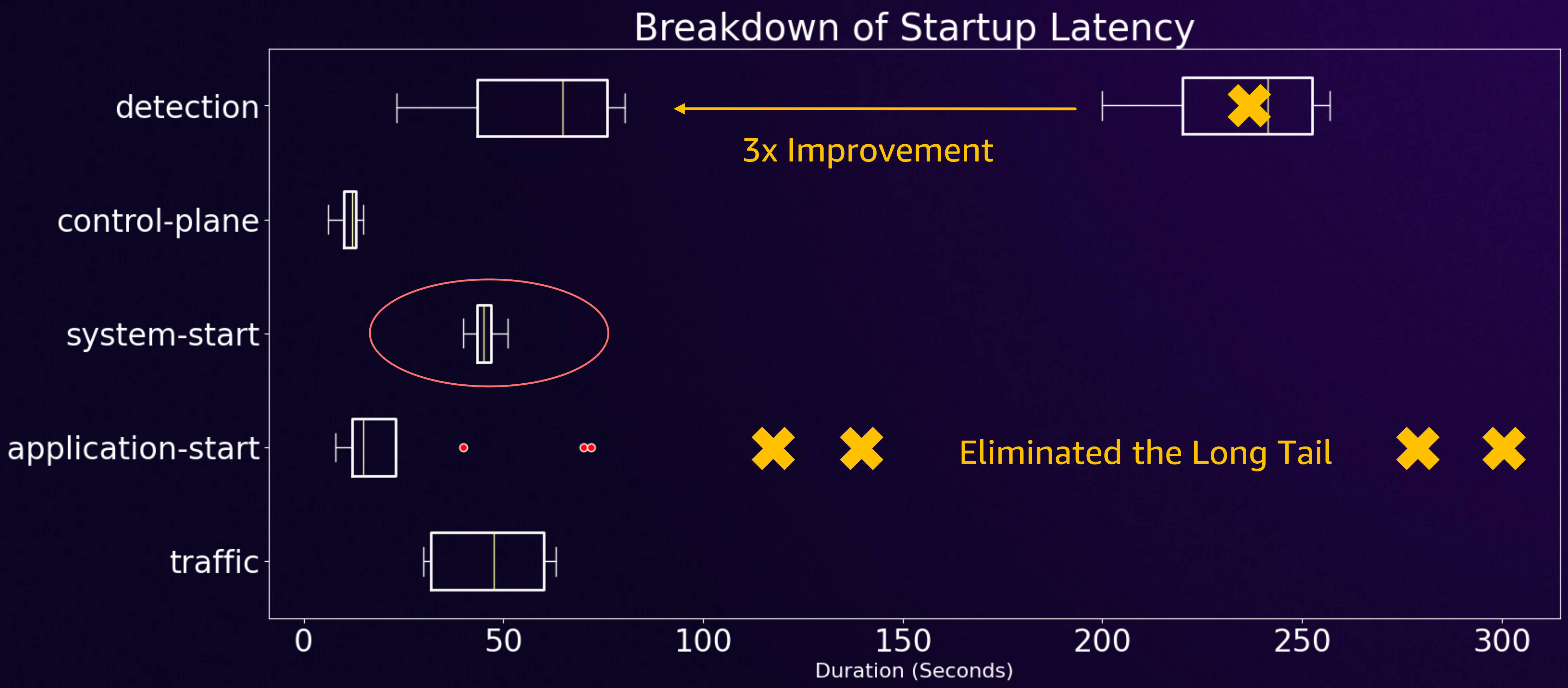


Time-to-recovery dominating factors



Detection > **App startup** > System startup > Hardware startup

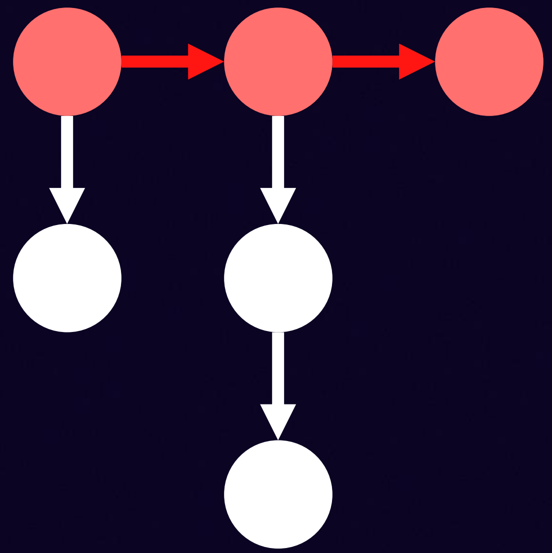
Time-to-recovery dominating factors



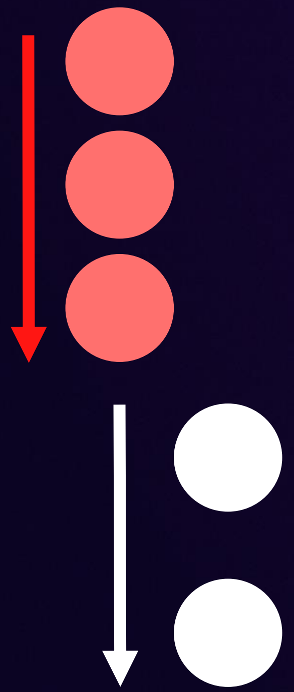
Detection > App startup > **System startup** > Hardware startup

Start system faster – systemd-analyze

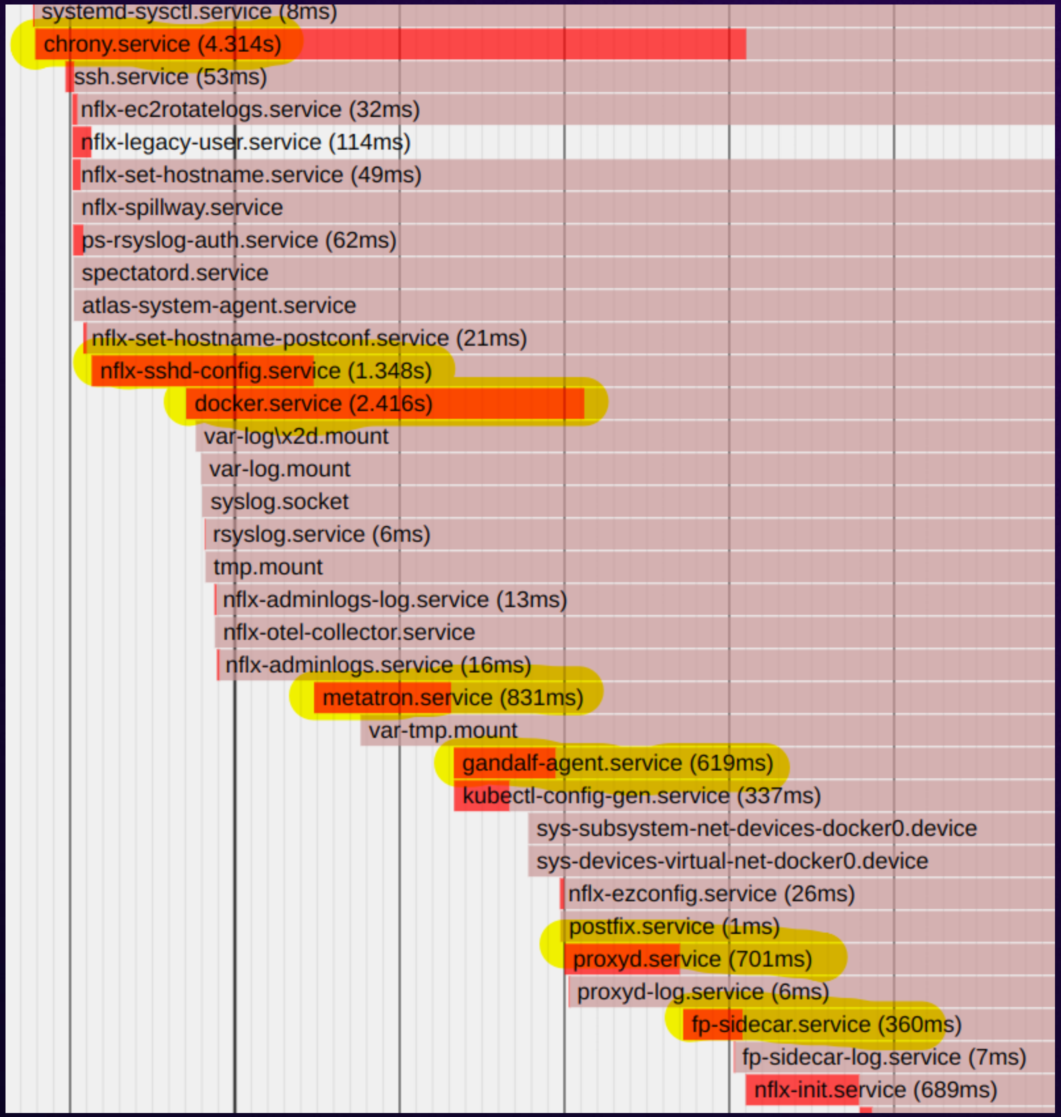
FIND UNIT CHAINS AND MAKE THEM PARALLEL



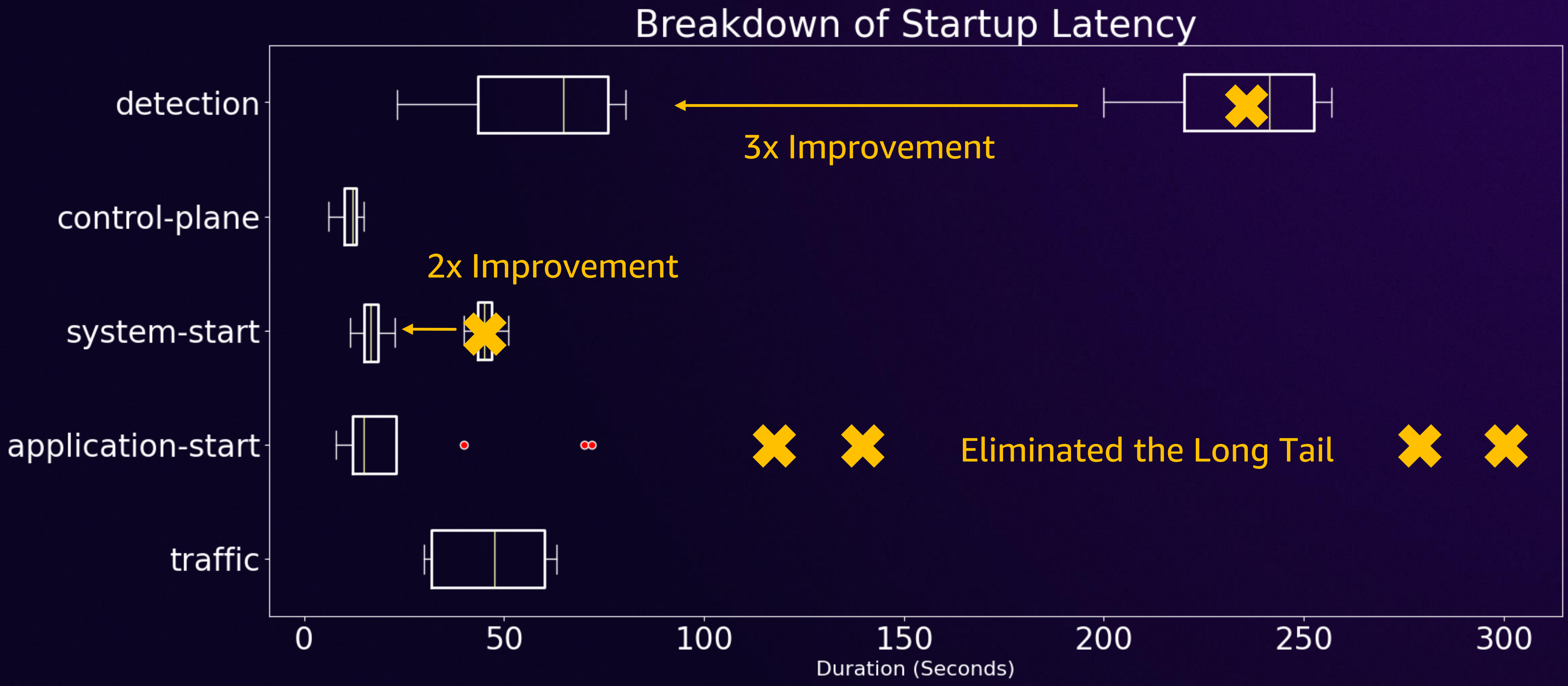
Sequential - ***Slow***



Parallel - ***Fast***

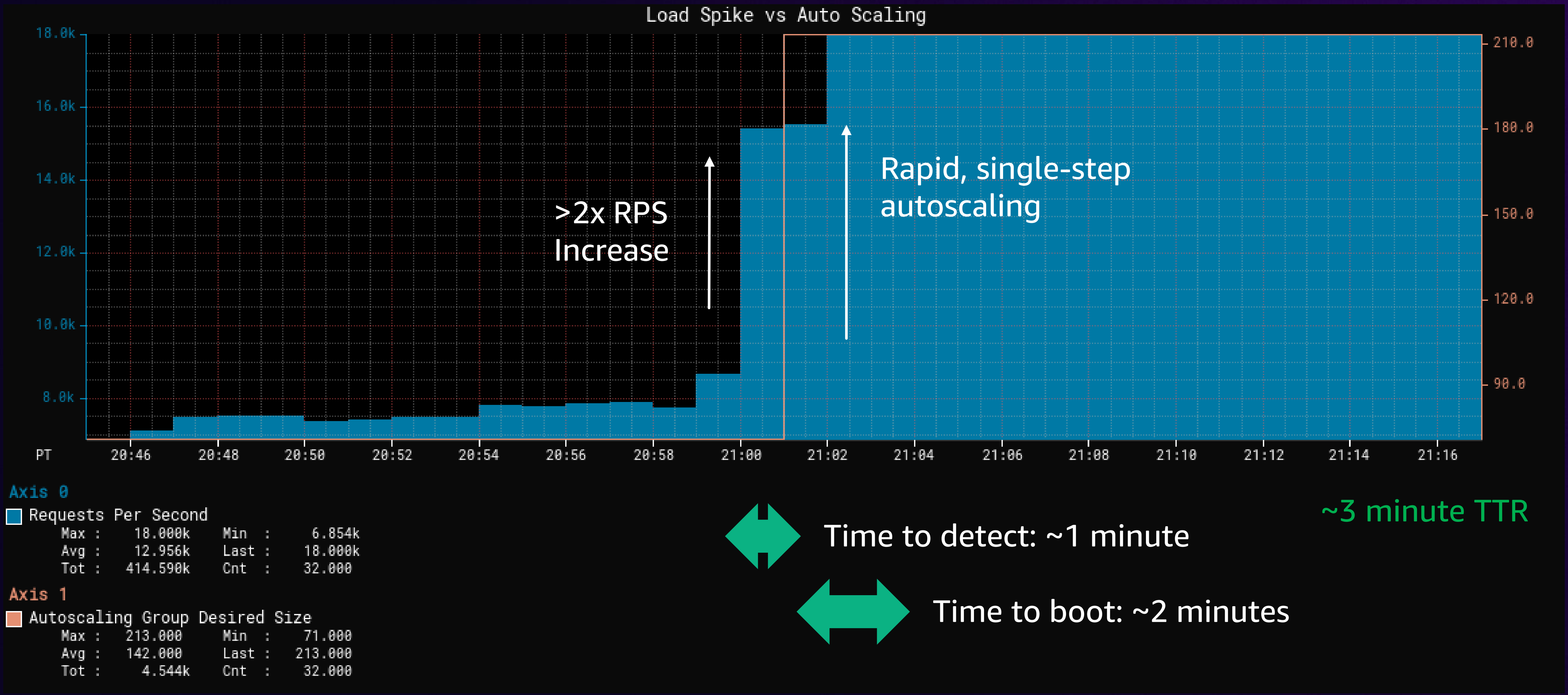


Time-to-recovery dominating factors



Detection > App startup > **System startup** > Hardware startup

Results



Load Spike vs Auto Scaling
18.0k
16.0k
14.0k
12.0k
10.0k
8.0k
210.0
180.0
150.0
120.0
90.0
PT
20:46 20:48 20:50 20:52 20:54 20:56 20:58 21:00 21:02 21:04 21:06 21:08 21:10 21:12 21:14 21:16
>2x RPS Increase
Rapid, single-step autoscaling
Axis 0
Requests Per Second
Max : 18.000k Min : 6.854k
Avg : 12.956k Last : 18.000k
Tot : 414.590k Cnt : 32.000
Axis 1
Autoscaling Group Desired Size
Max : 213.000 Min : 71.000
Avg : 142.000 Last : 213.000
Tot : 4.544k Cnt : 32.000
Time to detect: ~1 minute
Time to boot: ~2 minutes
~3 minute TTR

04: Stay available
Techniques to stay up

Build shared criticality nomenclature

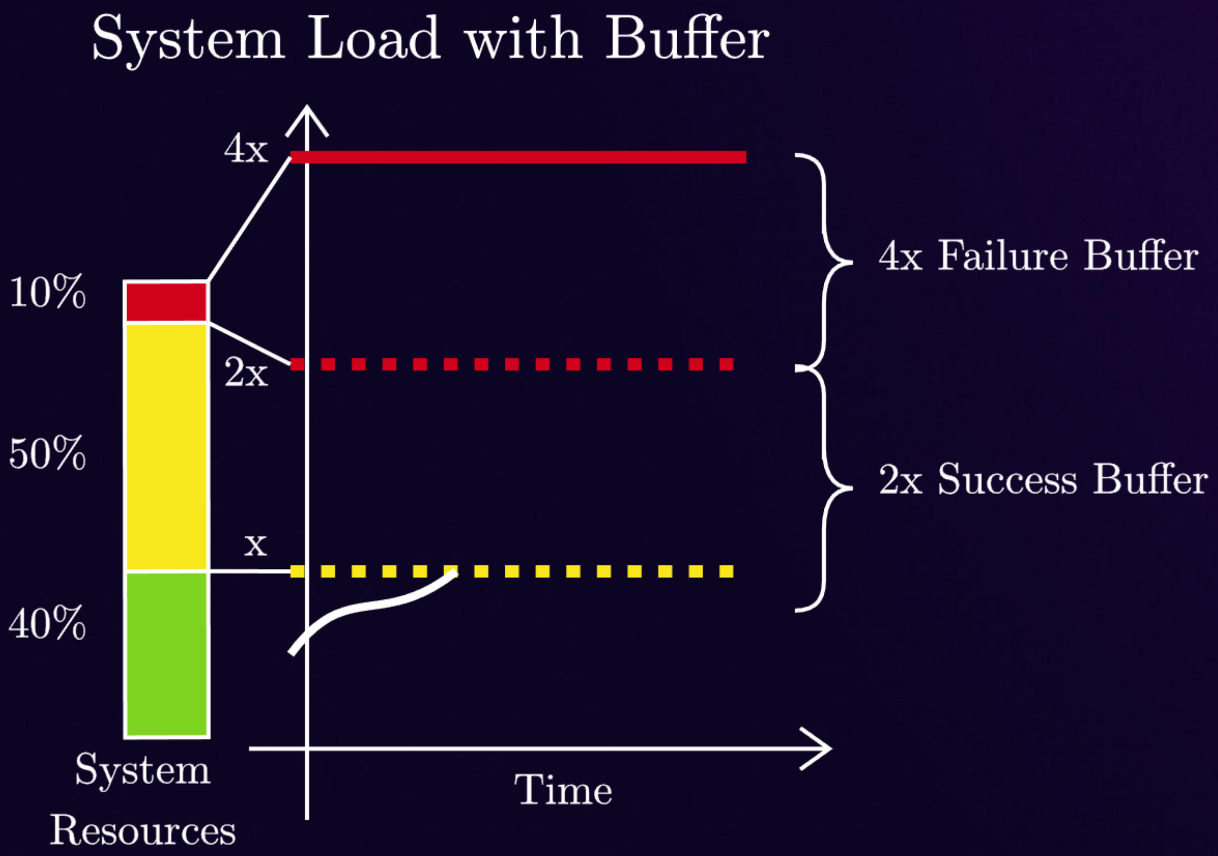
Define some tags, and start tagging

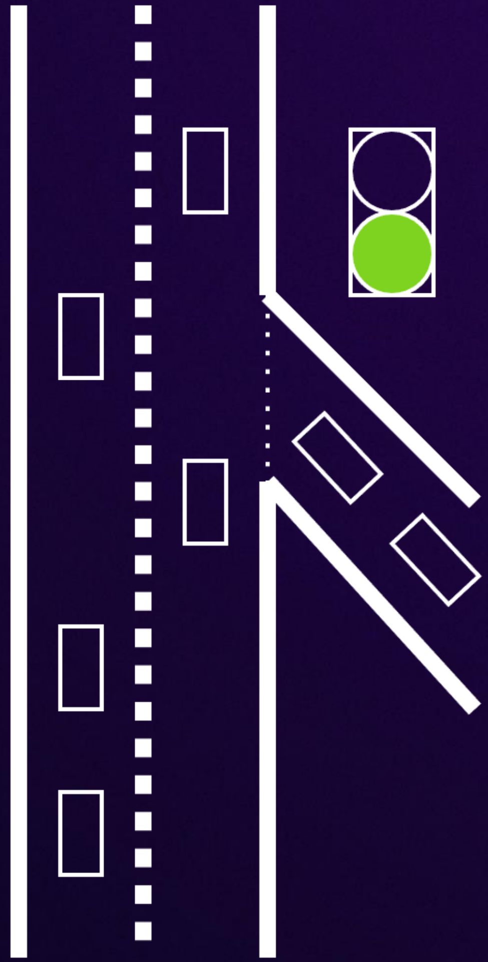
Less debating, more tagging

Tag	Values	Reason	Consequence
Tier	`int:= {0, 1, 2, 3}`	Spend $$ on what is important	Buffer, testing requirements
Business domain	`List[str] := {"svod", "gaming", "studio"…}`	Different domains scale differently	Deployment modalities, buffer, …
Lifecycle	`Str := {"alpha", "beta", "ga", "deprecated", "eol"}`	Do not waste time on deprecated apps	Exclude early/late from requirements

aws

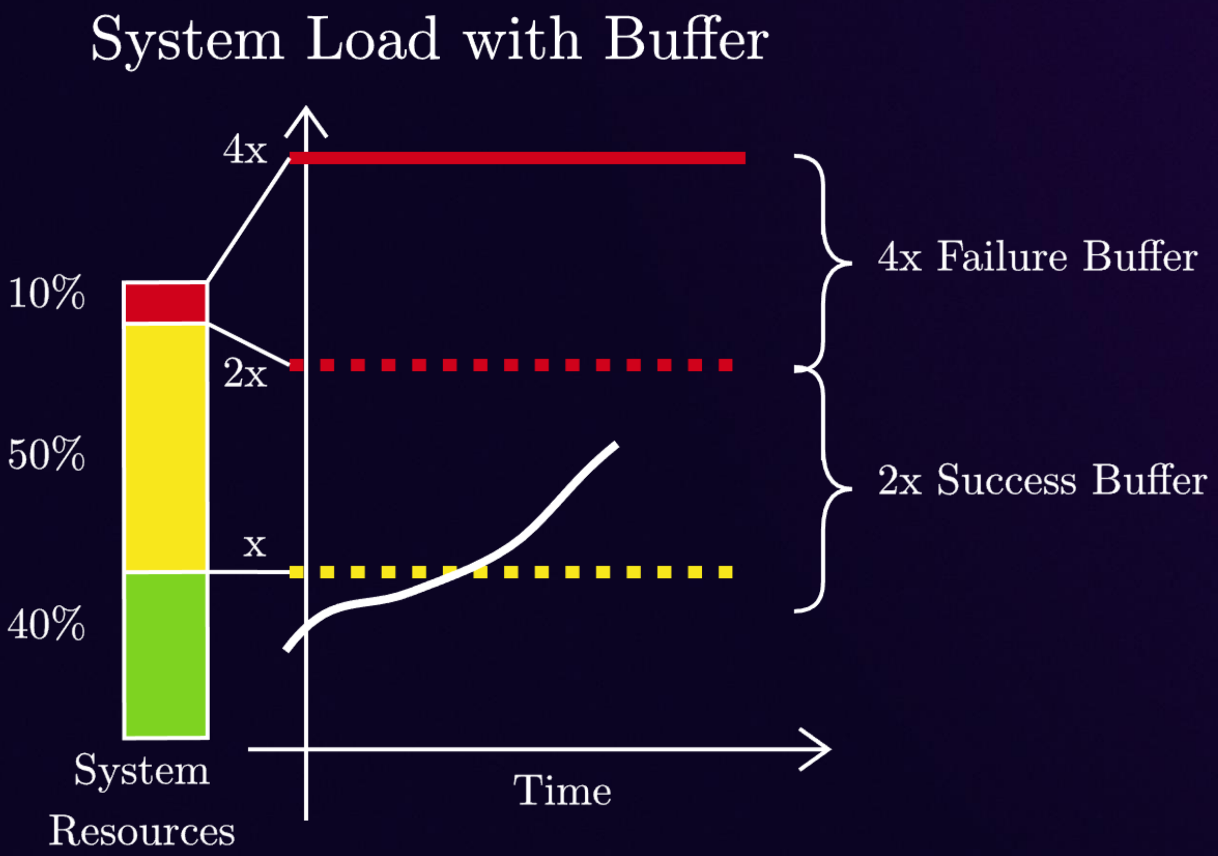
Load begins

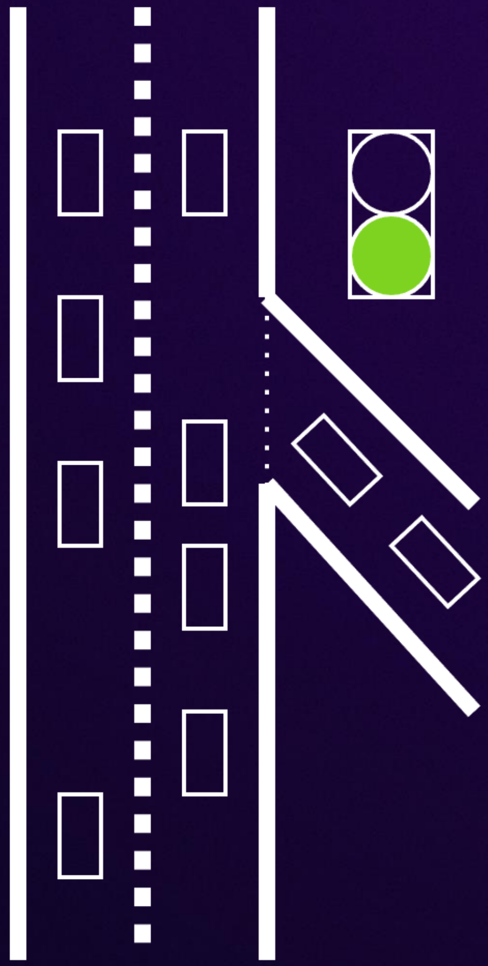




$$T_{transit} = 10m$$

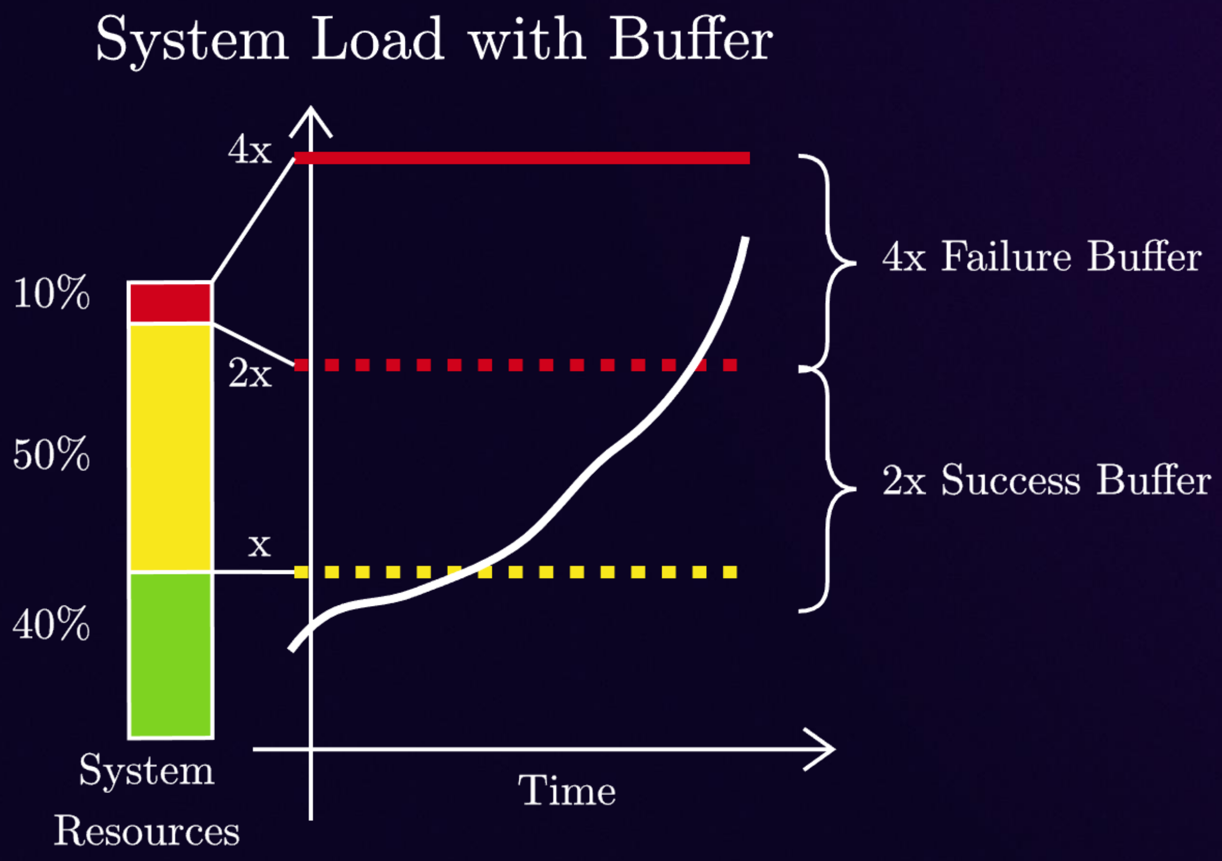
Load grows

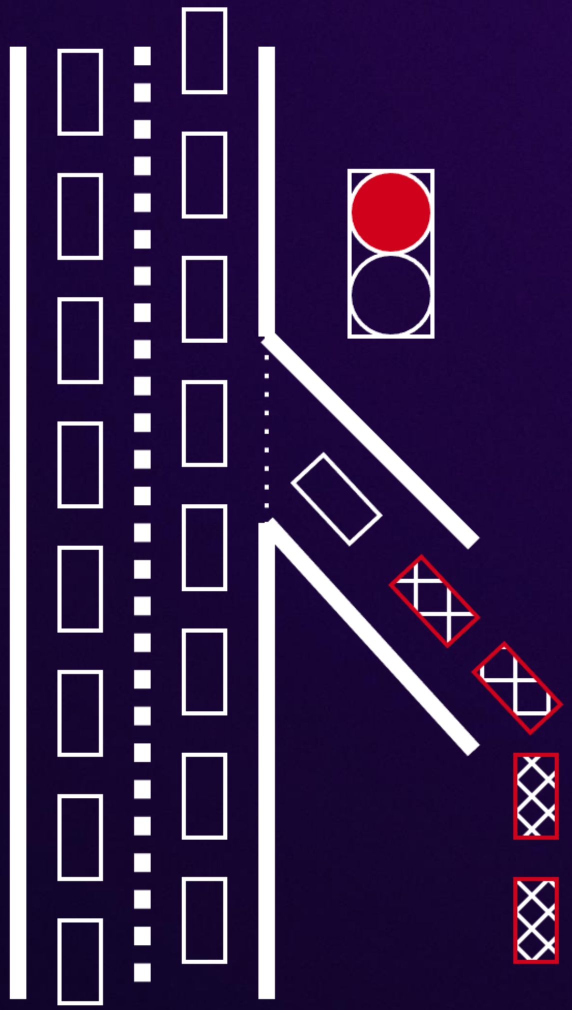




$$T_{transit} = 15m$$

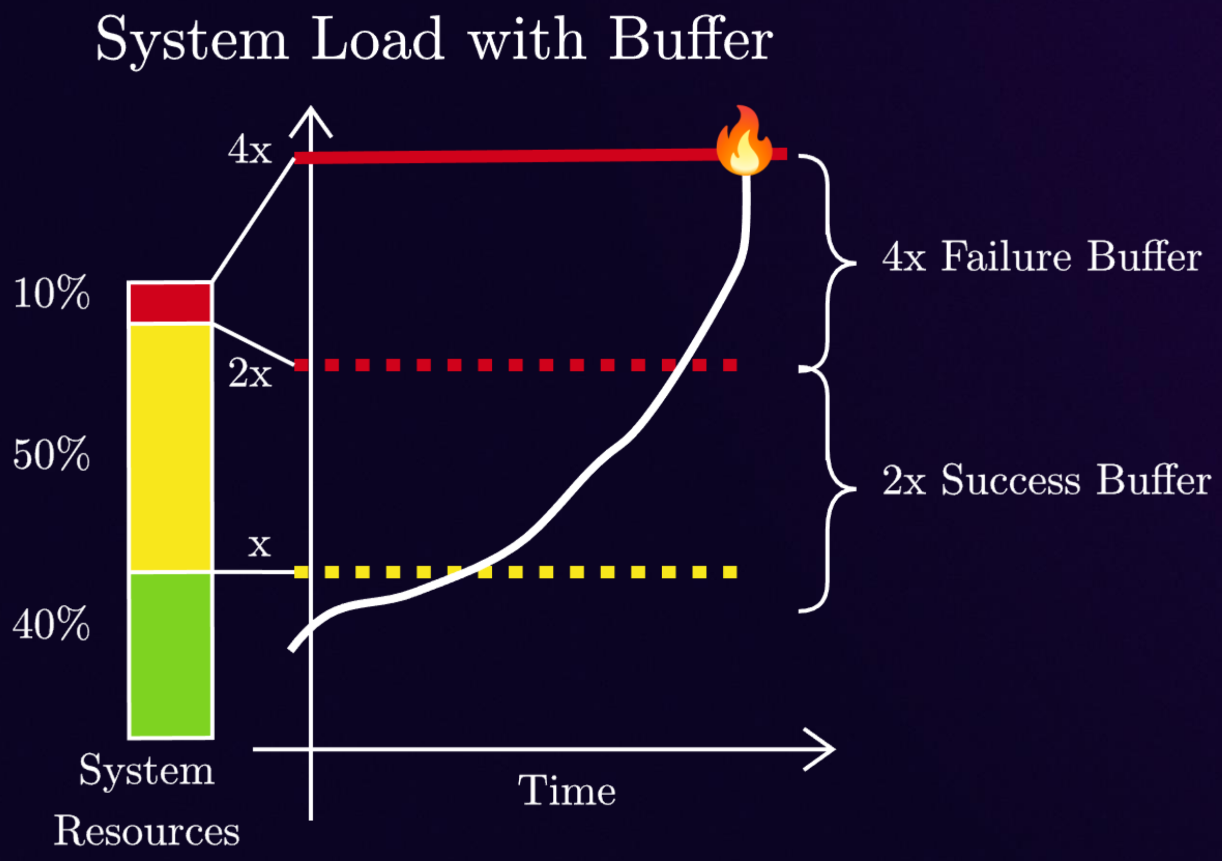
Load sheds

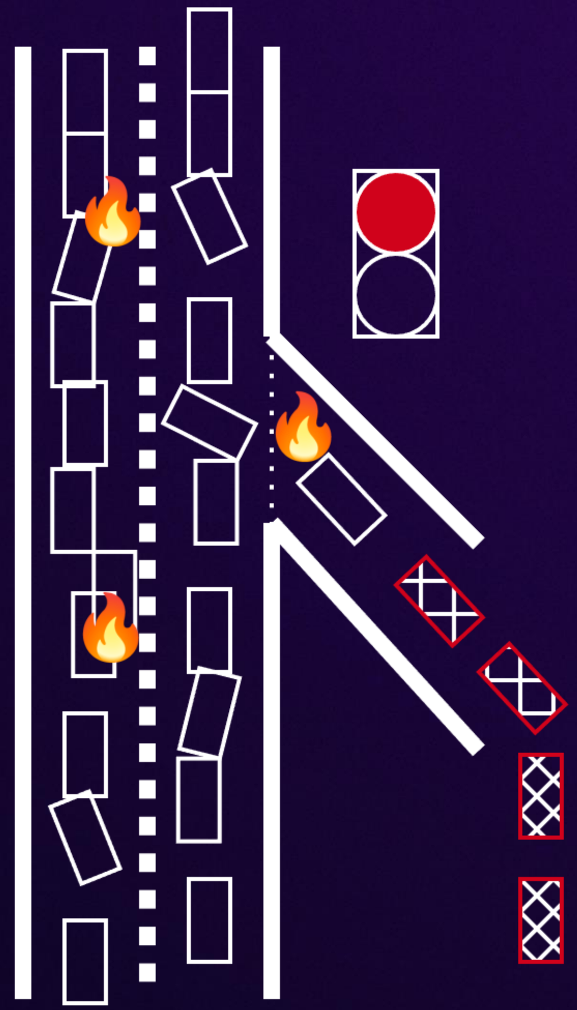




$$T_{transit} = 30m$$

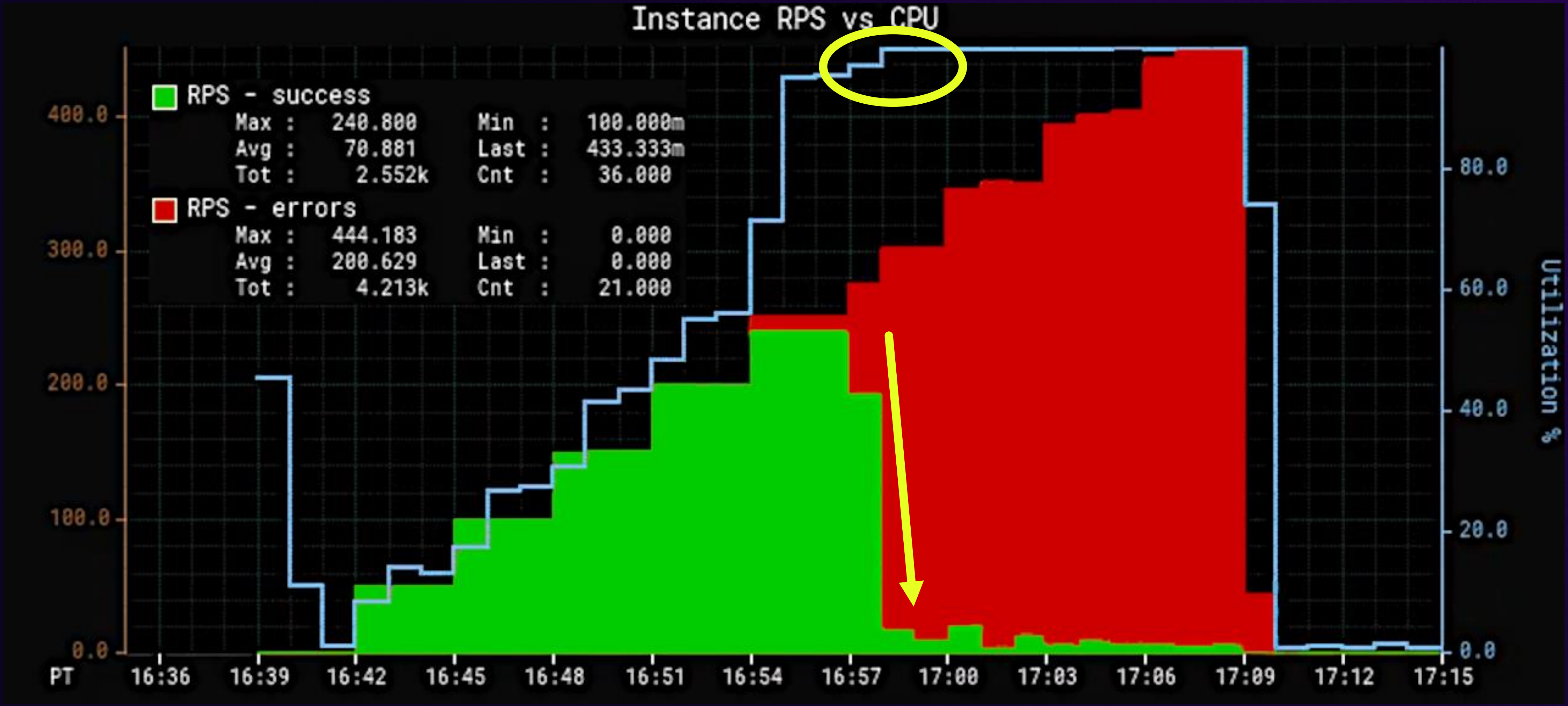
Congestive failure – very *bad*





$T_{transit} = \infty$

Congestive failure



Stay up – Prioritized shedding

Success Buffer

Prioritized CPU Shedding

Failure Buffer

Unprioritized CPU Shedding

Blog post:

Enhancing Netflix Reliability with Service-Level Prioritized Load Shedding

Applying Quality of Service techniques at the application level

Netflix Technology Blog · Follow
Published in Netflix TechBlog · 12 min read · Jun 24, 2024

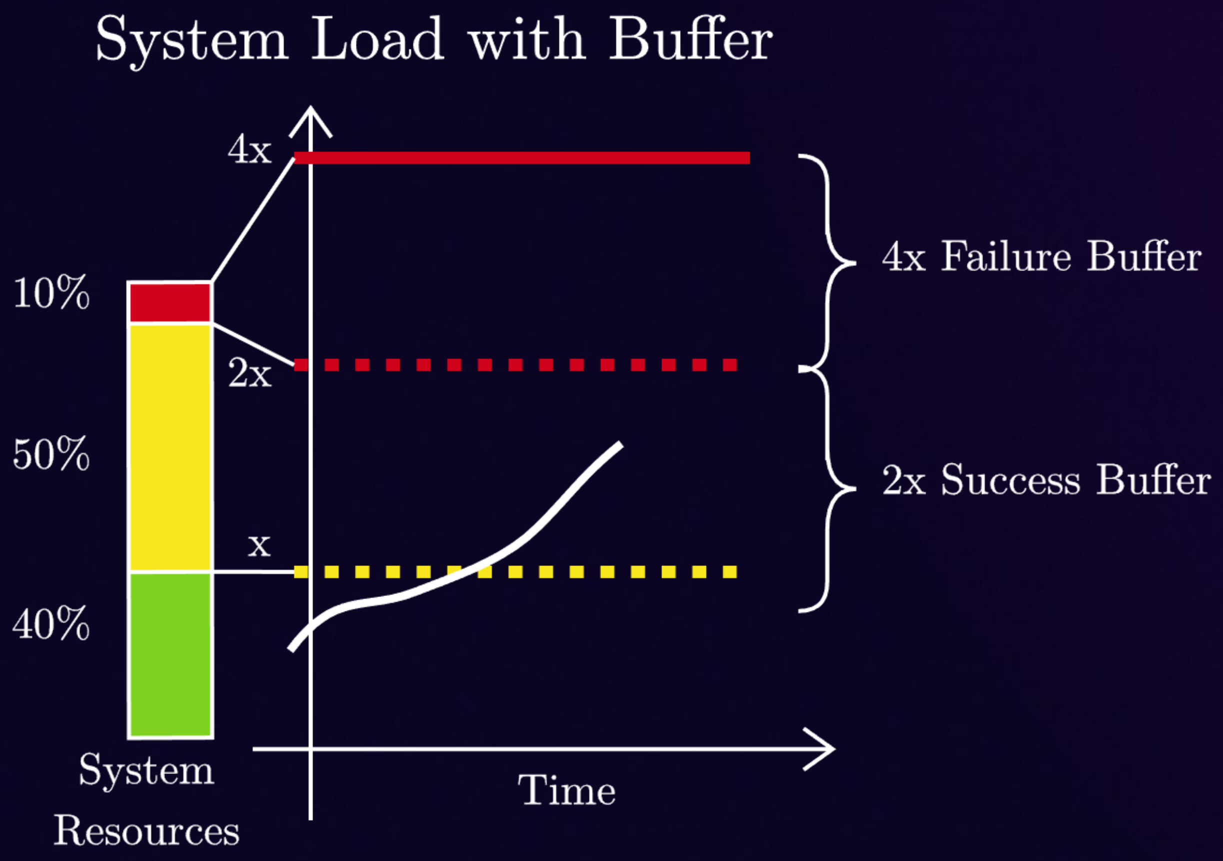
337 4

Anirudh Mendiratta, Kevin Wang, Joey Lynch, Javier Fernandez-Ivern, Benjamin Fedorka

Introduction

In November 2020, we introduced the concept of prioritized load shedding at the API gateway level in our blog post, Keeping Netflix Reliable Using Prioritized Load Shedding. Today, we're excited to dive deeper into how we've extended this strategy to the individual service level, focusing on the video streaming control plane and data plane, to further enhance user experience and system resilience.

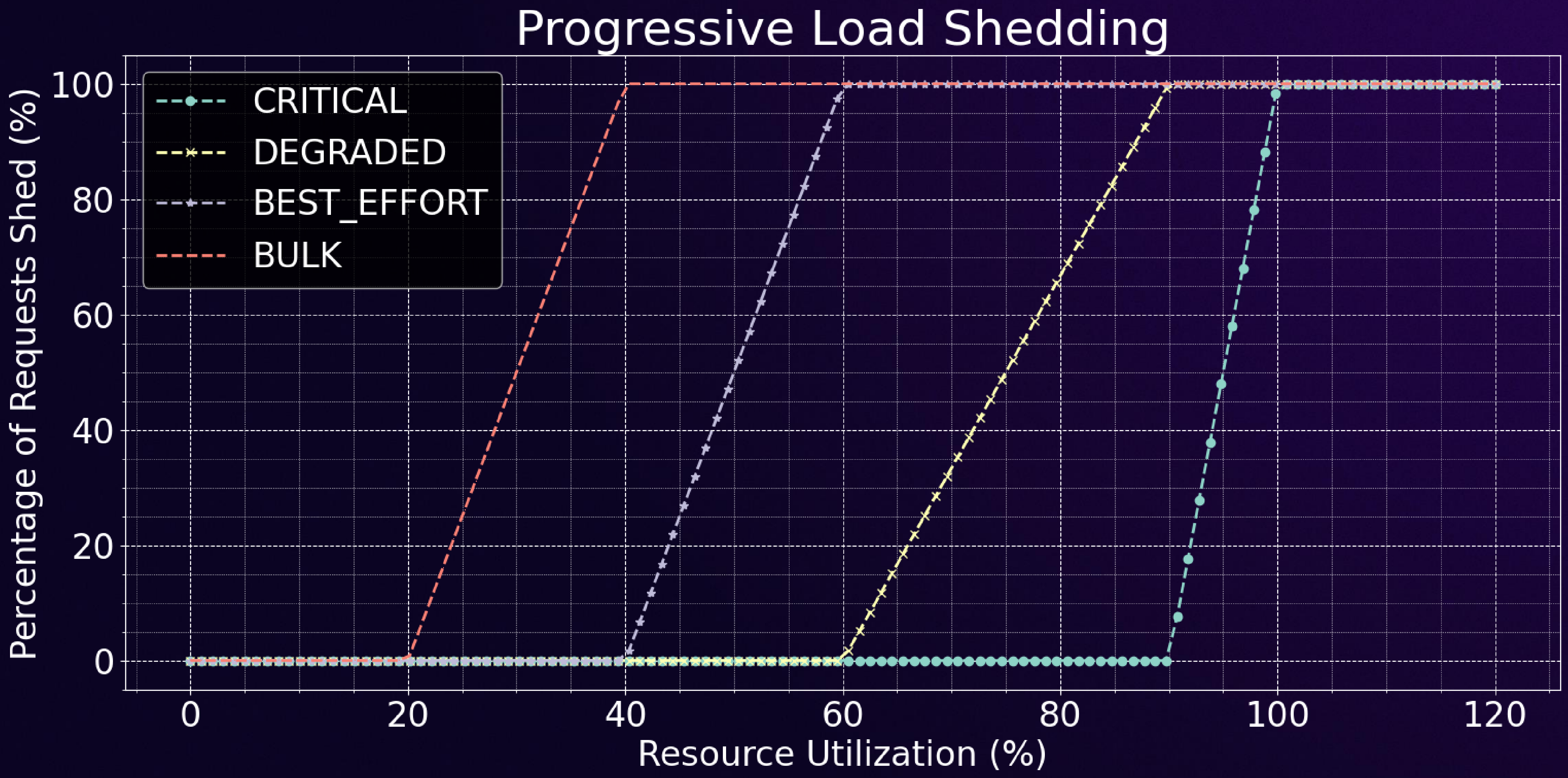
Stay up – Allocate buffers



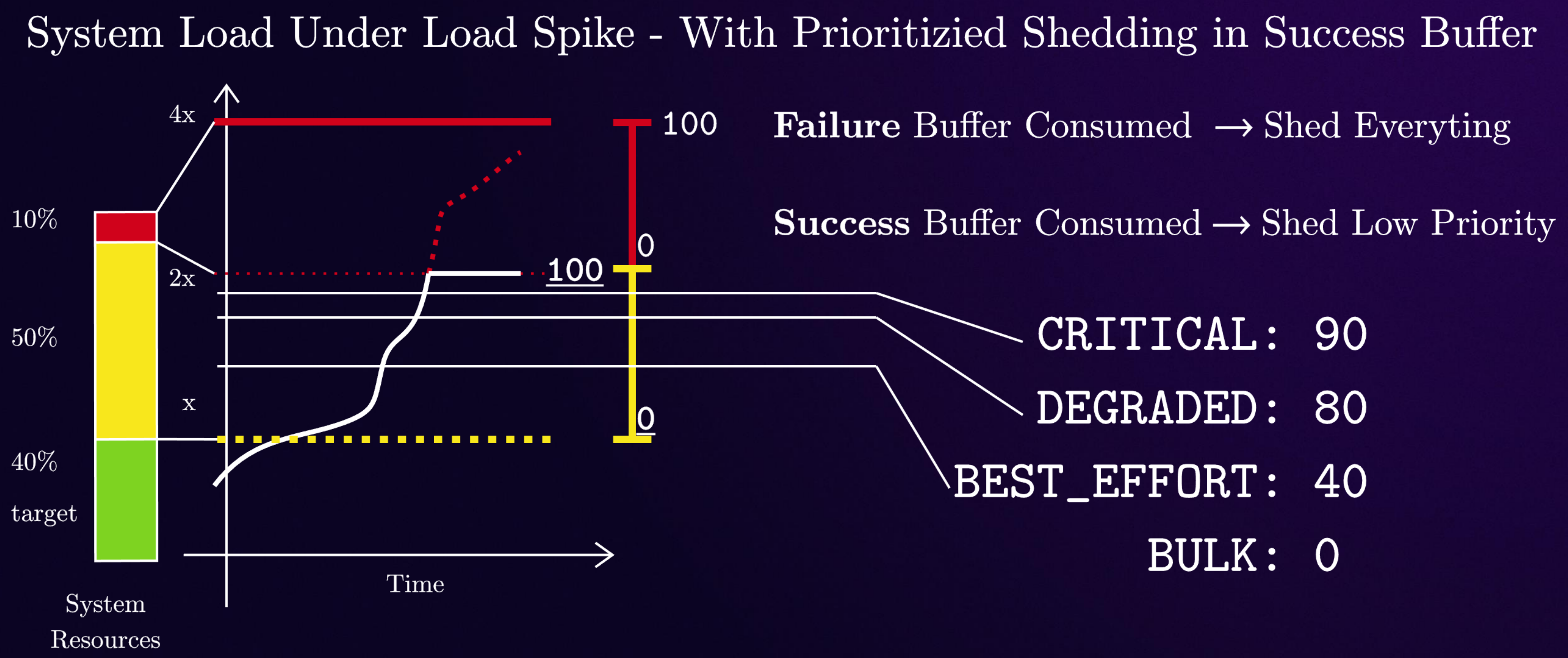

```
resource:
  utilization:
    cpu:
      target: 40
      max: 90
```

$$Buffer_{success} \propto T_{startup}$$

Stay up – Define priority buckets



Stay up – Allocate buffers



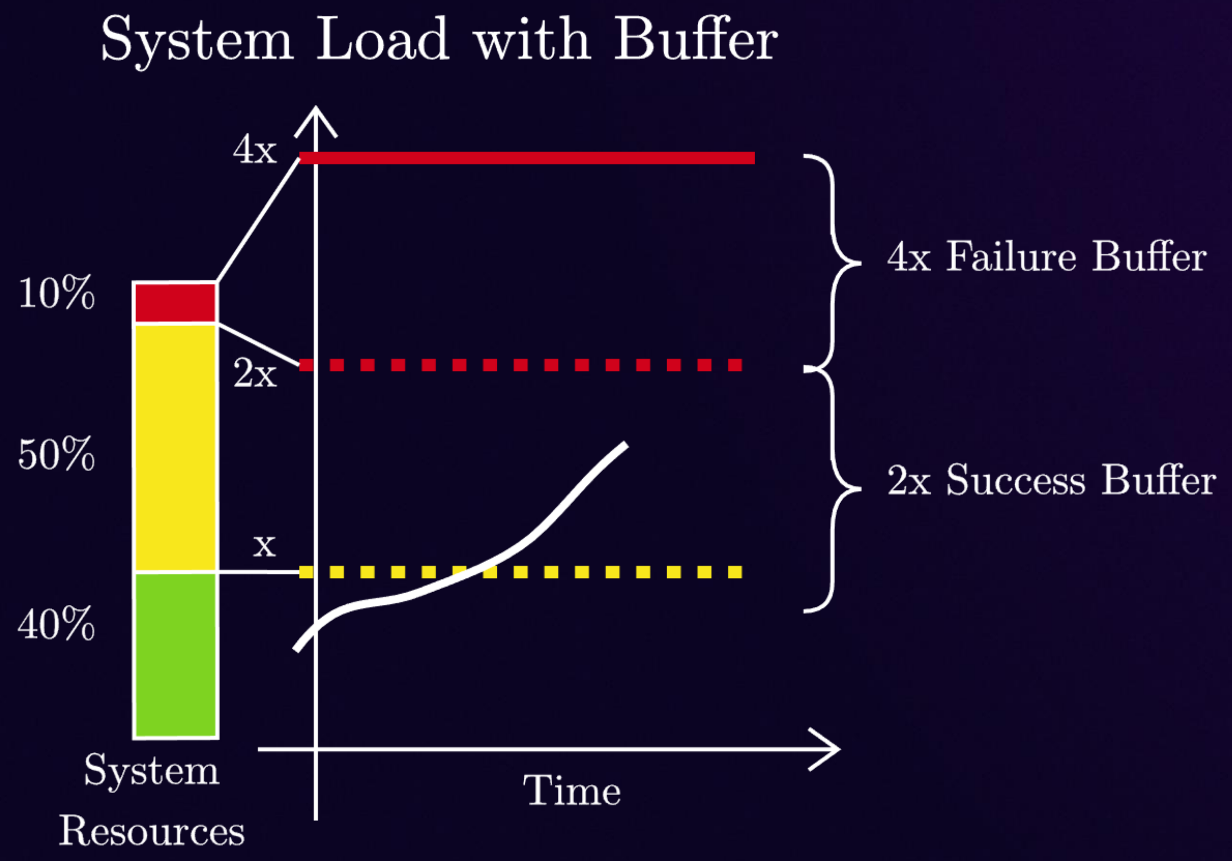
Stay up – Prioritize requests

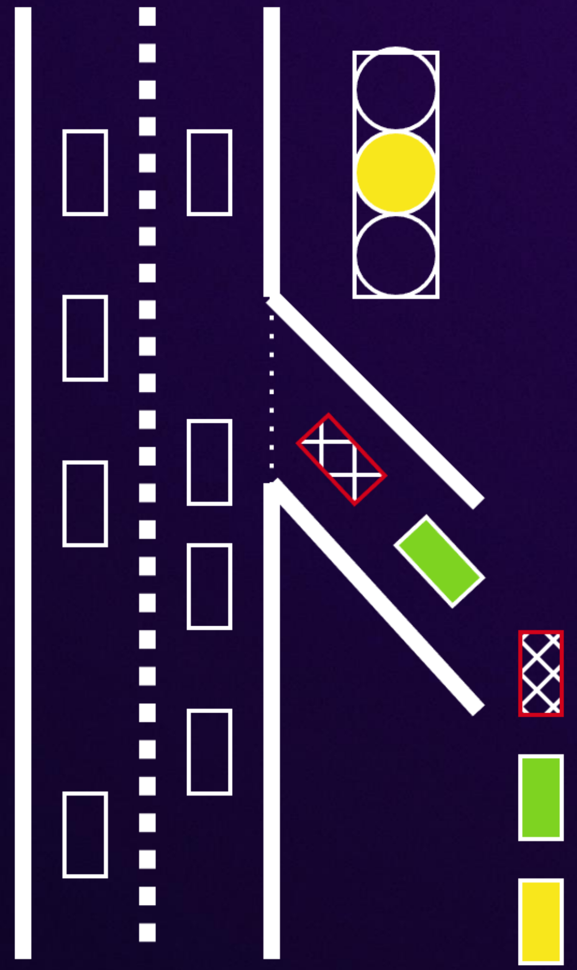
```
return context -> {
  Request req = context.getRequest();
  // Prioritize a particular path
  if (req.getPath().startsWith("/critical-play-url")) {
    return PriorityBucket.CRITICAL;
  }

  // Deprioritize background requests
  if (req.getParams().contains("background")) {
    return PriorityBucket.DEGRADED;
  }

  // Take the client device priority
  return getClientPriority(context.getHeaders());
}
```

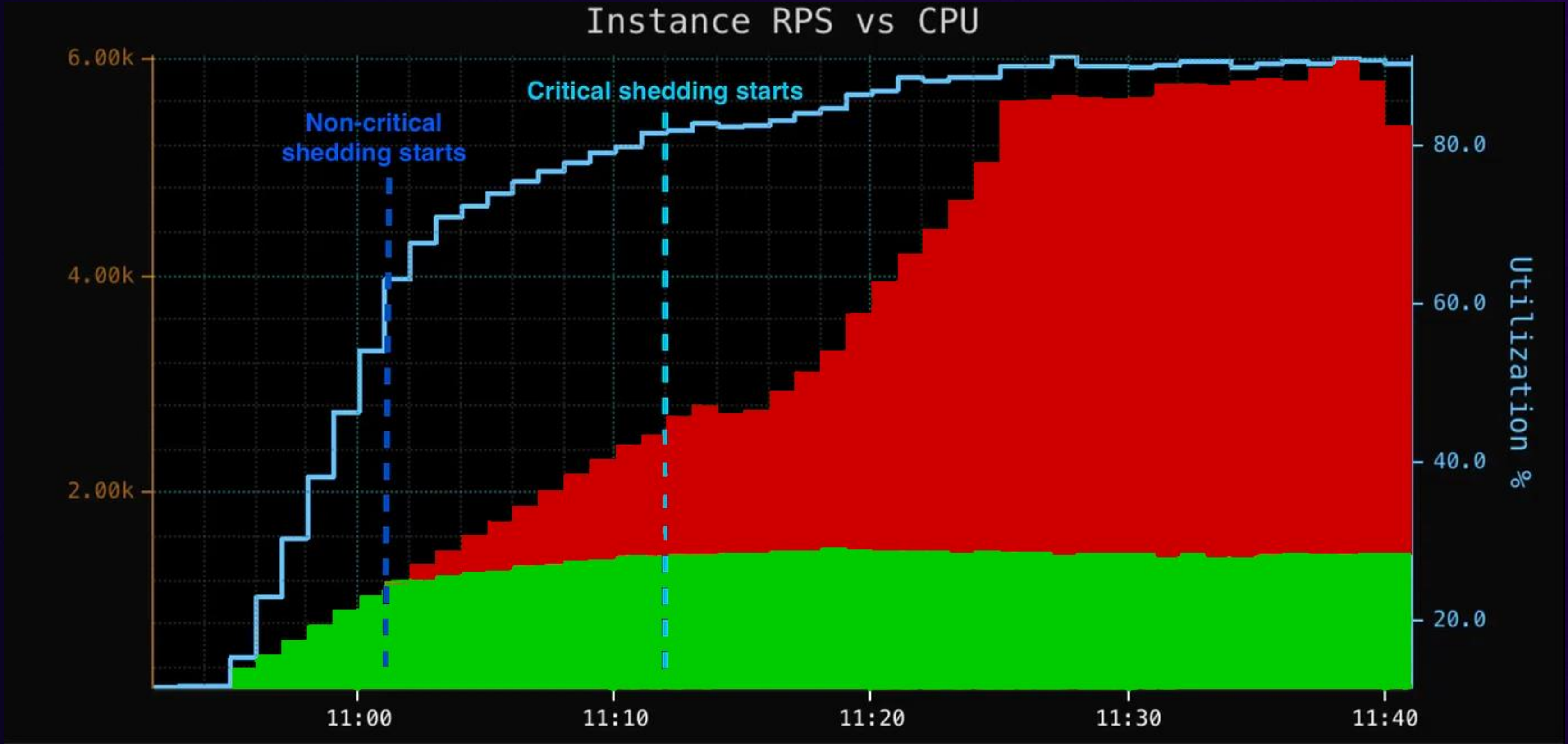
Stay up – Prioritized load shedding





$$T_{transit} = 15m$$

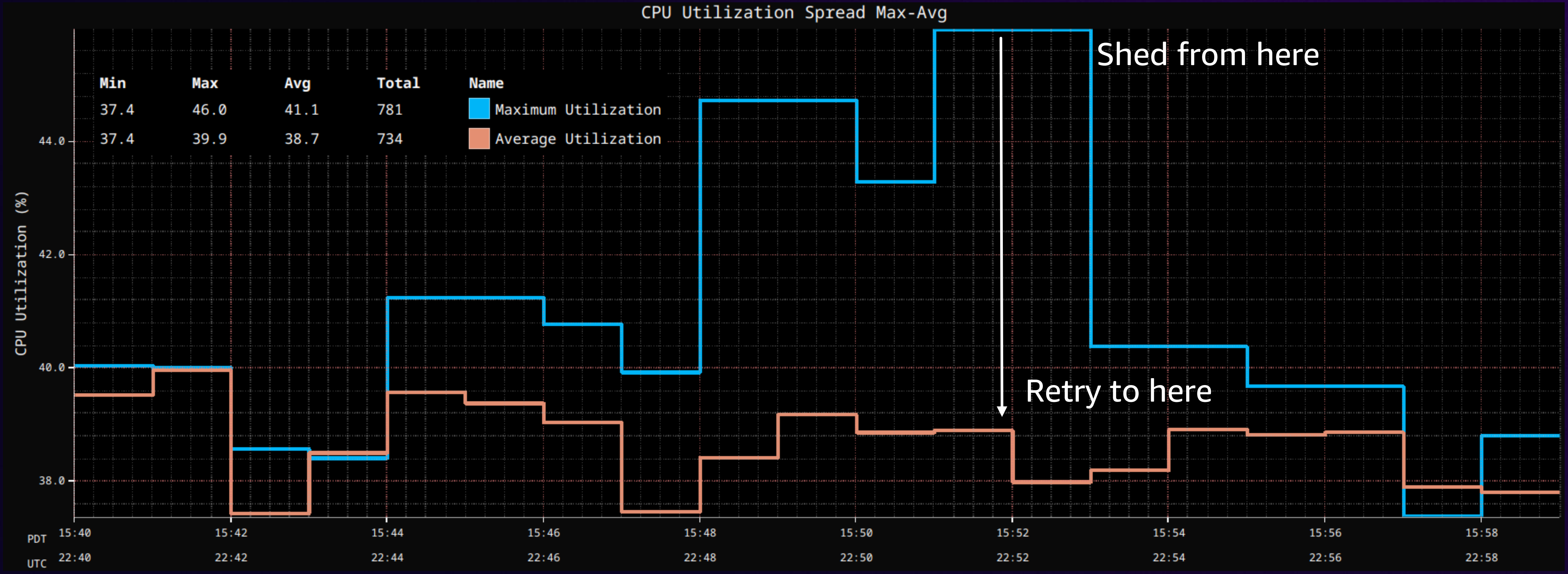
Shed the right load



Show All | Hide All | Filter:

Min	Max	Avg	Total	Name
201	1.48k	1.24k	60.7k	RPS – success
375m	4.59k	2.49k	102k	RPS – errors
11.5	91.0	73.5	3.60k	CPU utilization

Are retries a good idea?



Retry sparingly

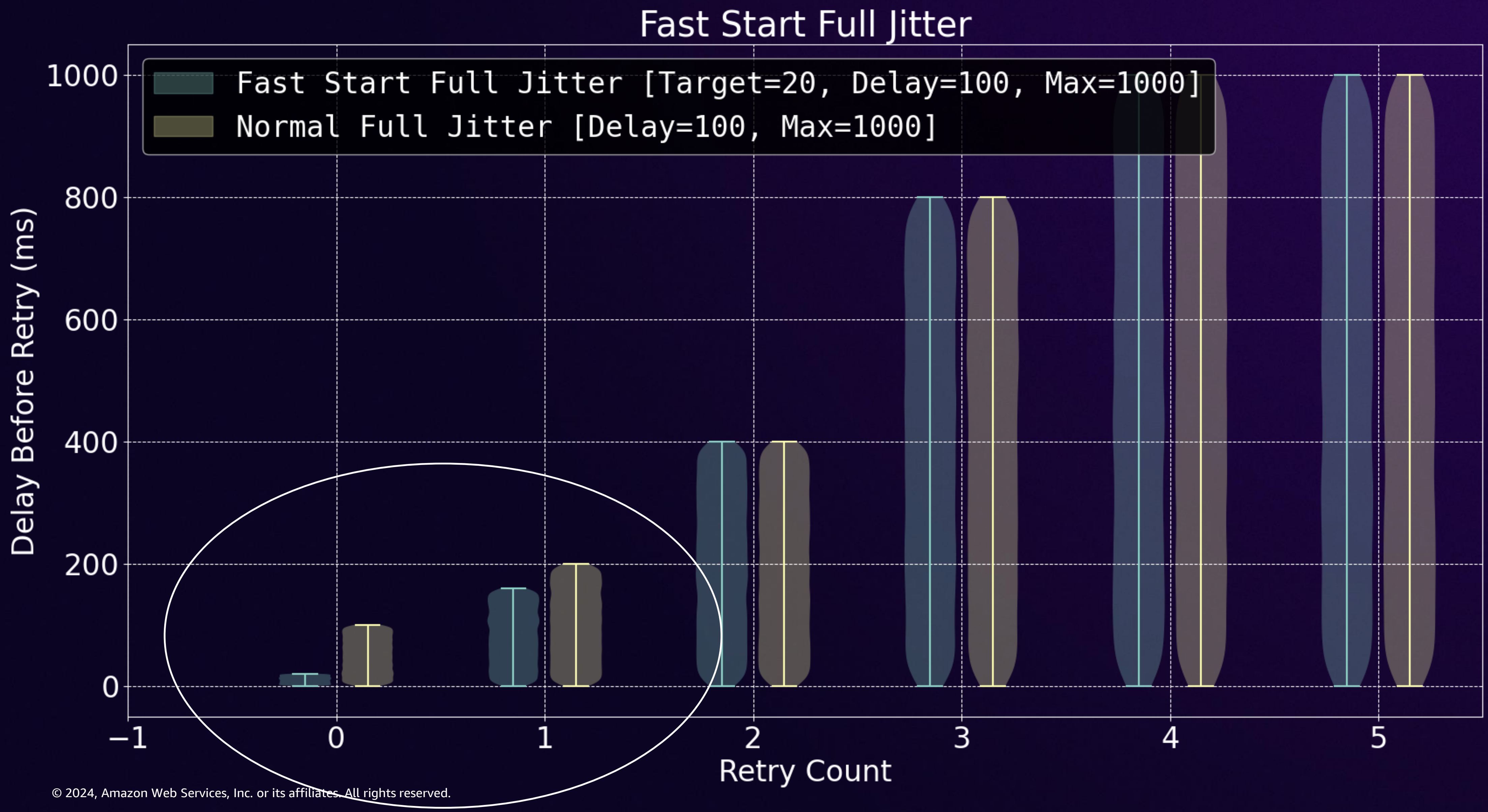
```
# Full jitter exponential backoff on shedding only
interceptor.retry.default.maxRetries     = 2
                    i.r.d.statuses       = UNAVAILABLE
                    i.r.d.backoffPolicy = exponential
                    i.r.d.backoffPolicy.jitterMode     = full
                    i.r.d.backoffPolicy.targetMillis   = 20
                    i.r.d.backoffPolicy.delayMillis    = 100
                    i.r.d.backoffPolicy.maxDelayMillis = 1000
```

$$\textit{let } R = \text{retry \#} \in [0, 1, 2, \ldots \text{retry}_{max} - 1]$$

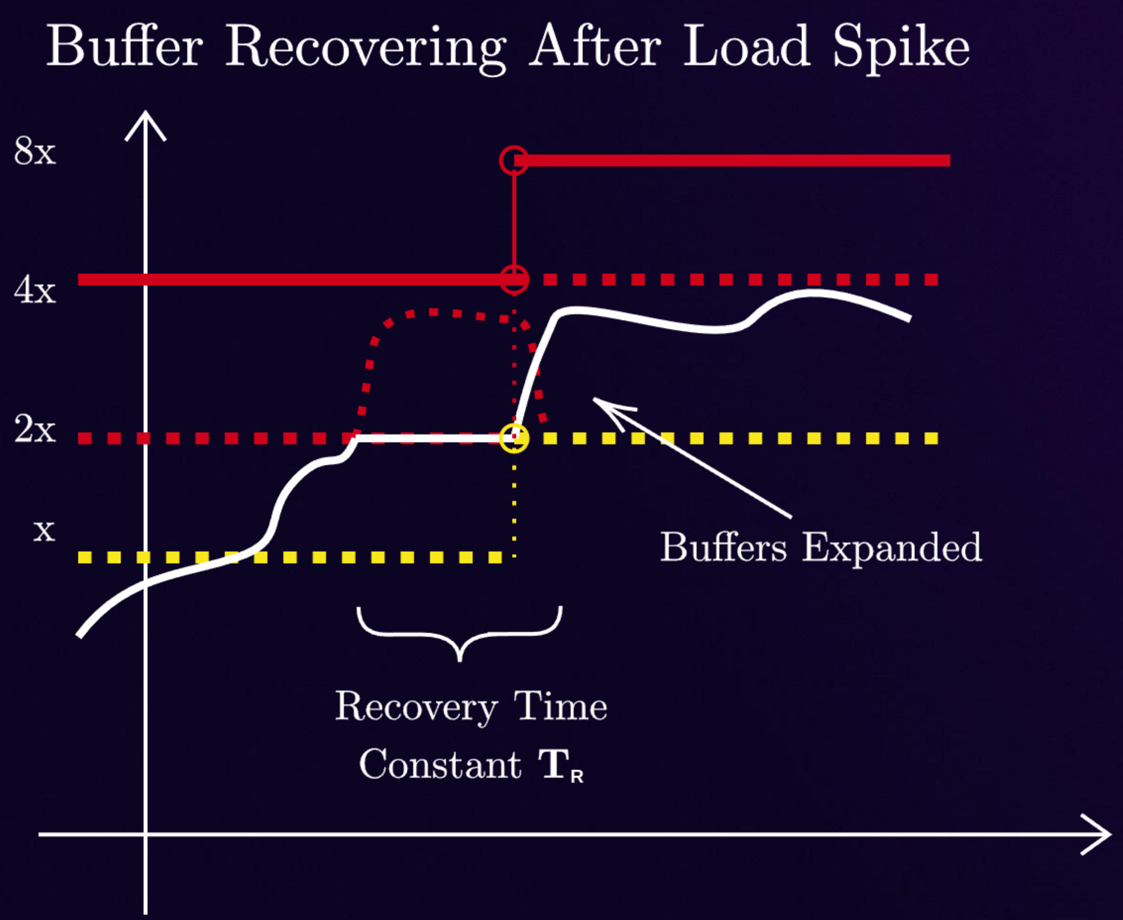
$$\text{base(R)} = \min\left(\text{delay}, \text{target} \times (R + 1)^2\right)$$

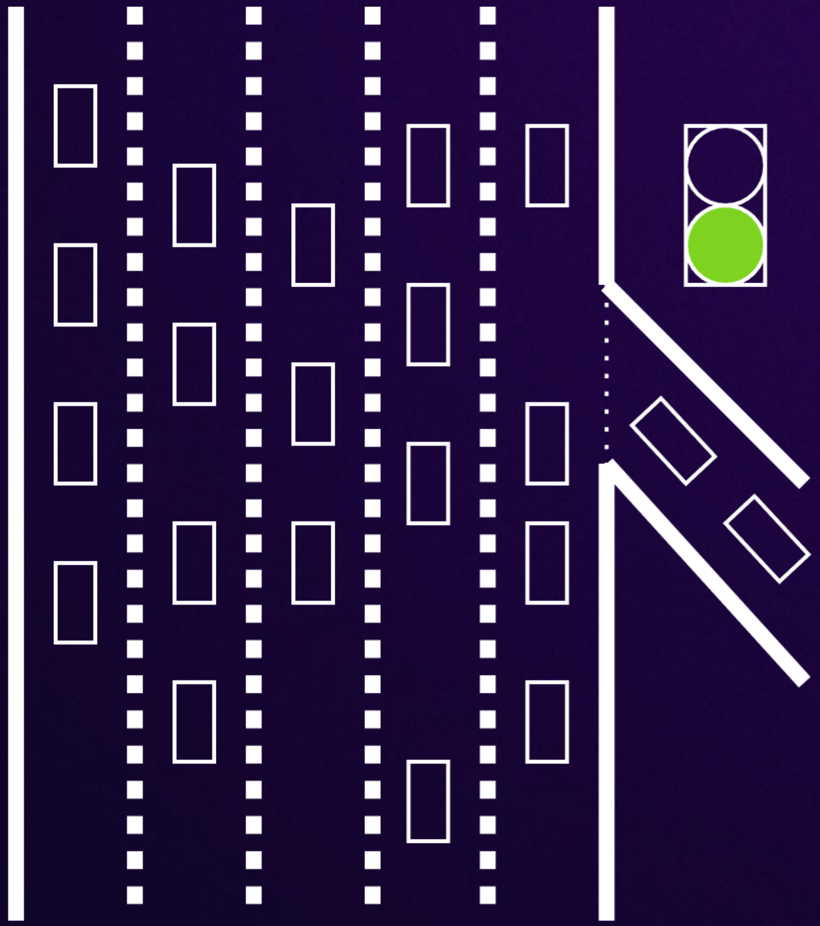
$$\text{retry(R)} = \text{rand}\left[0, \min\left\{\text{delay}_{\text{max}}, \text{base(R)} \times 2^{\text{R}}\right\}\right)$$

Fast start full jitter



More capacity is the real solution





$$T_{transit} = 15m$$

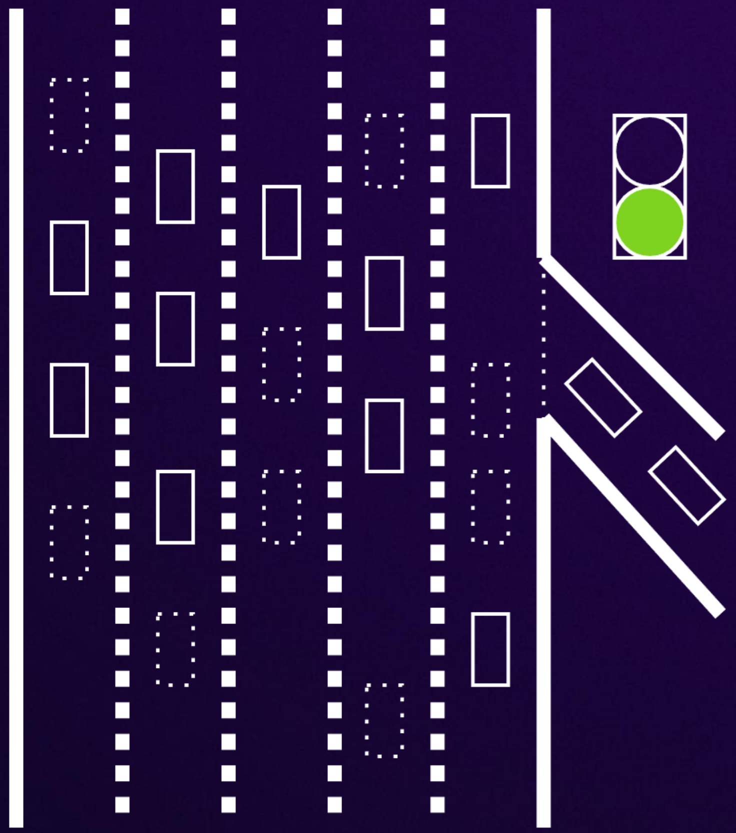
What about IO bound services?

CPU CAPACITY IS NECESSARY, BUT NOT SUFFICIENT!

Most services talk to other services

Async calls don't take much CPU

Latency???



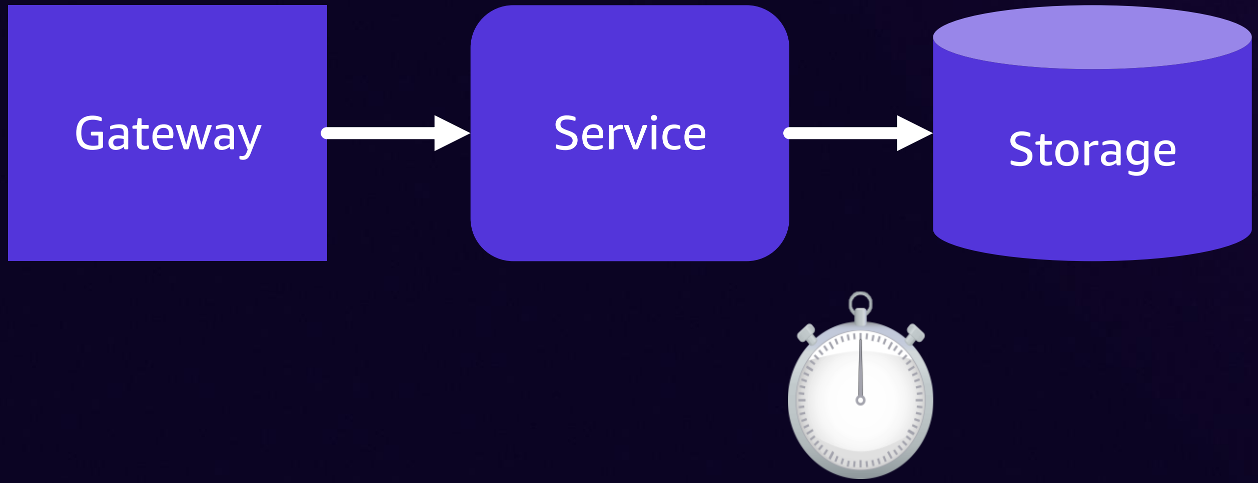
$T_{transit} = 15m$

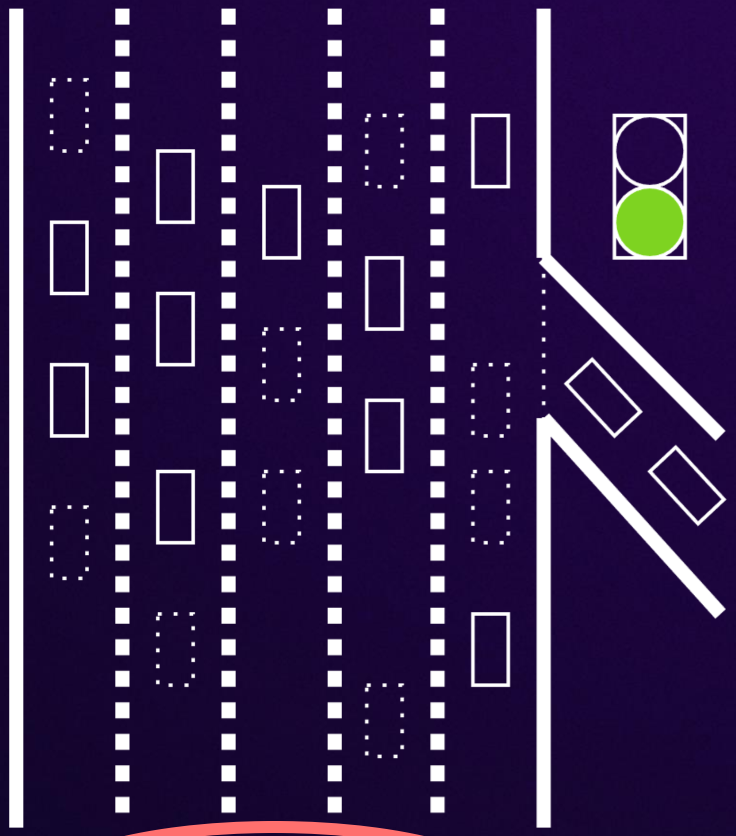
What about IO bound services?

CPU CAPACITY IS NECESSARY, BUT NOT SUFFICIENT!

Most services talk to other services

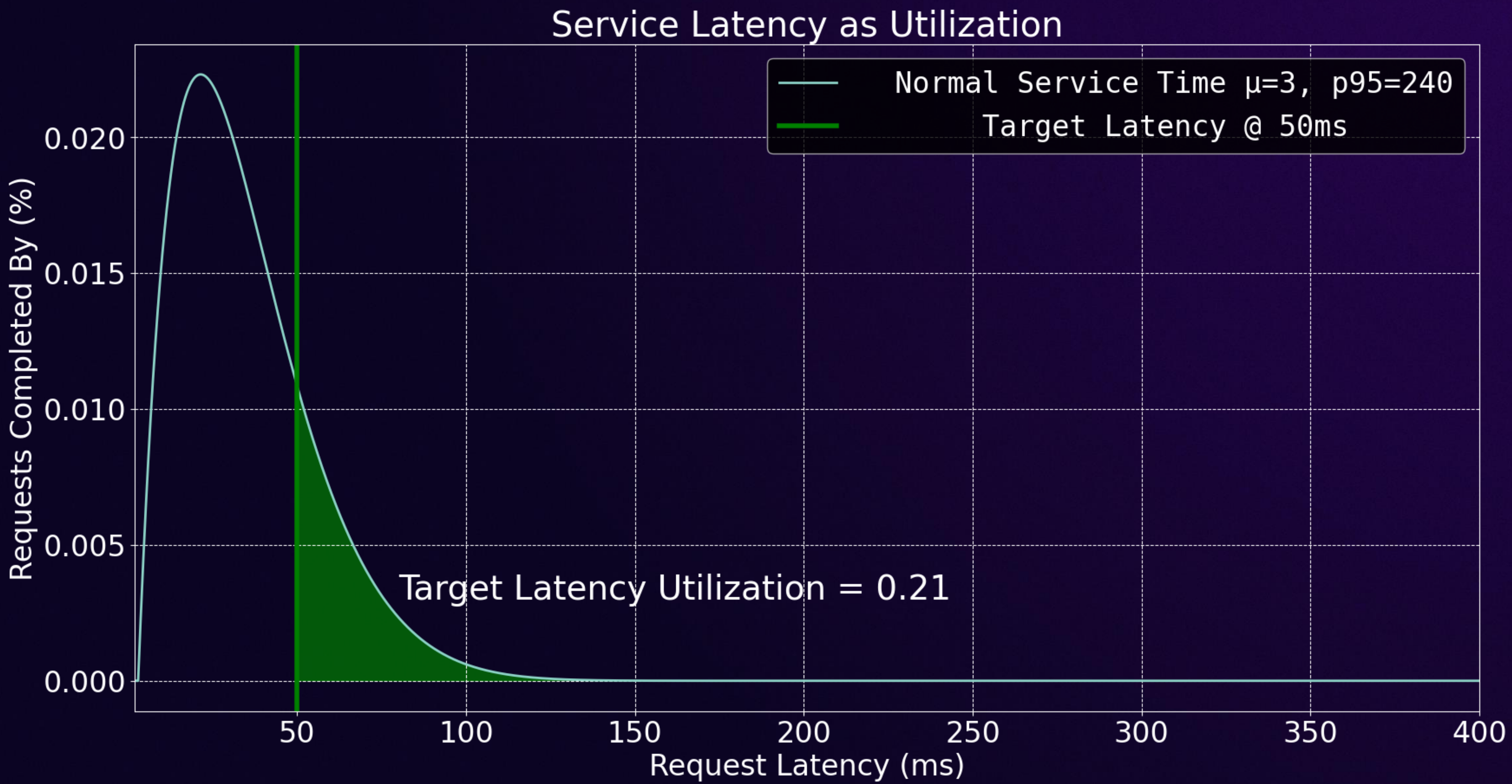
Async calls don't take much CPU



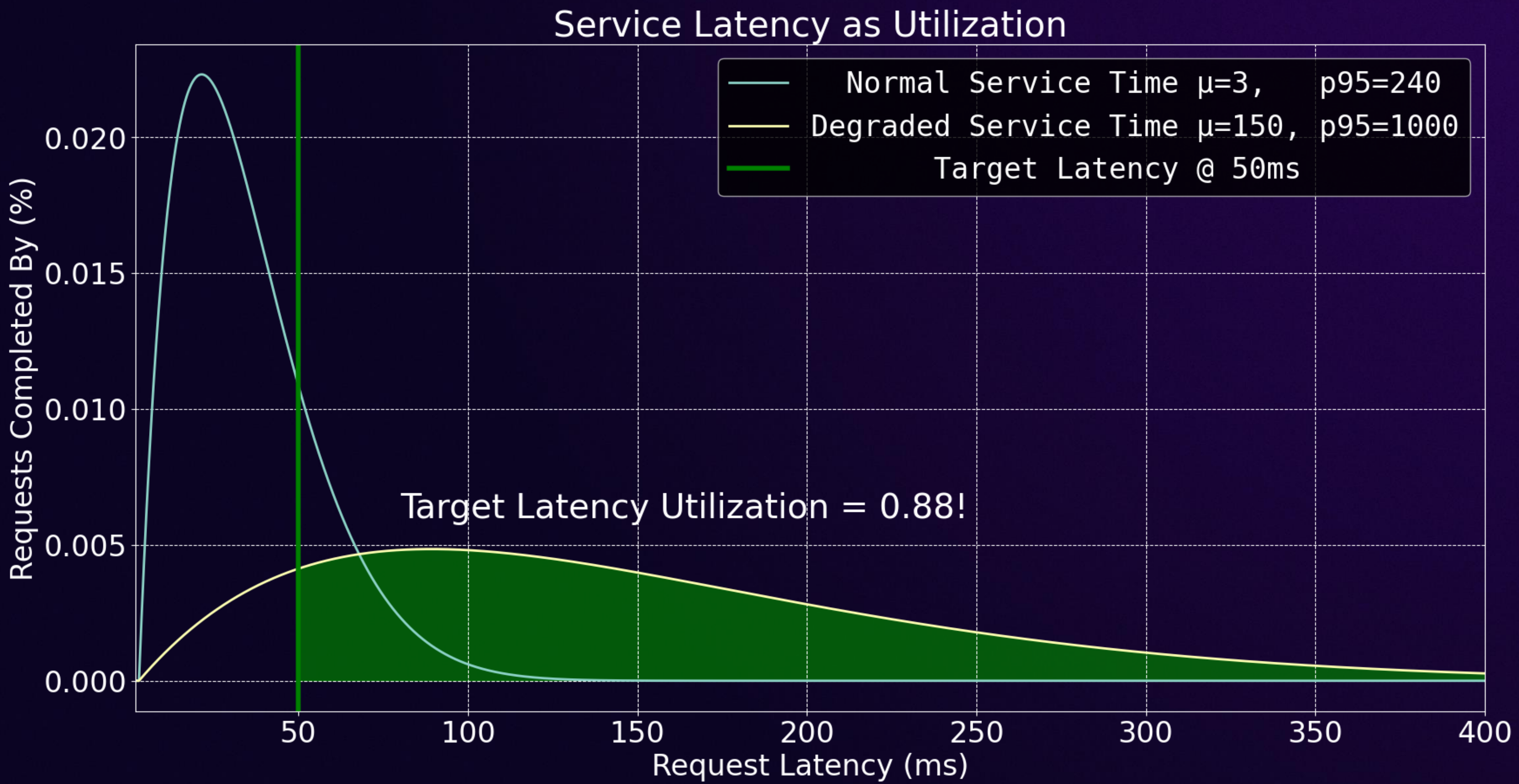


$$T_{transit} = 15m$$

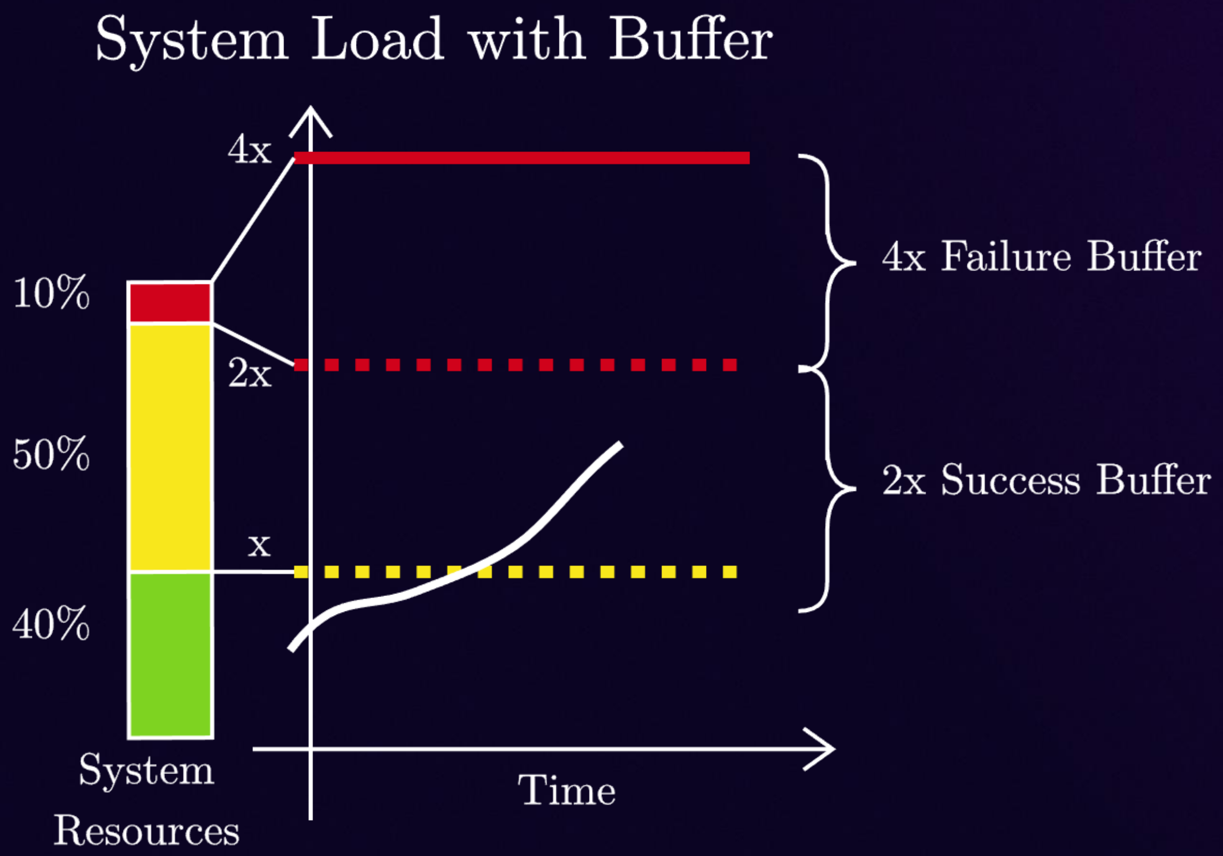
Measure latency as utilization



Measure latency as utilization

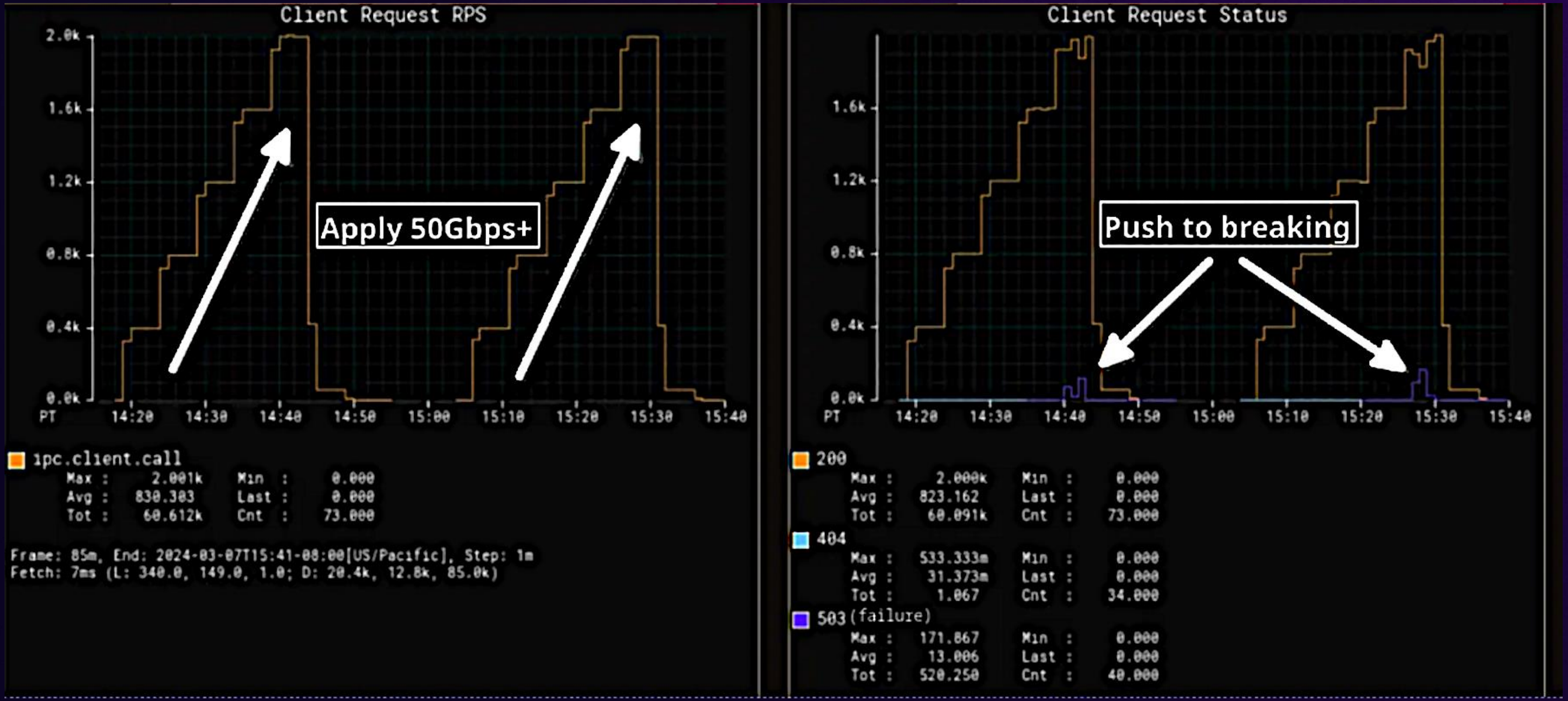


Stay up – Allocate IO success buffer

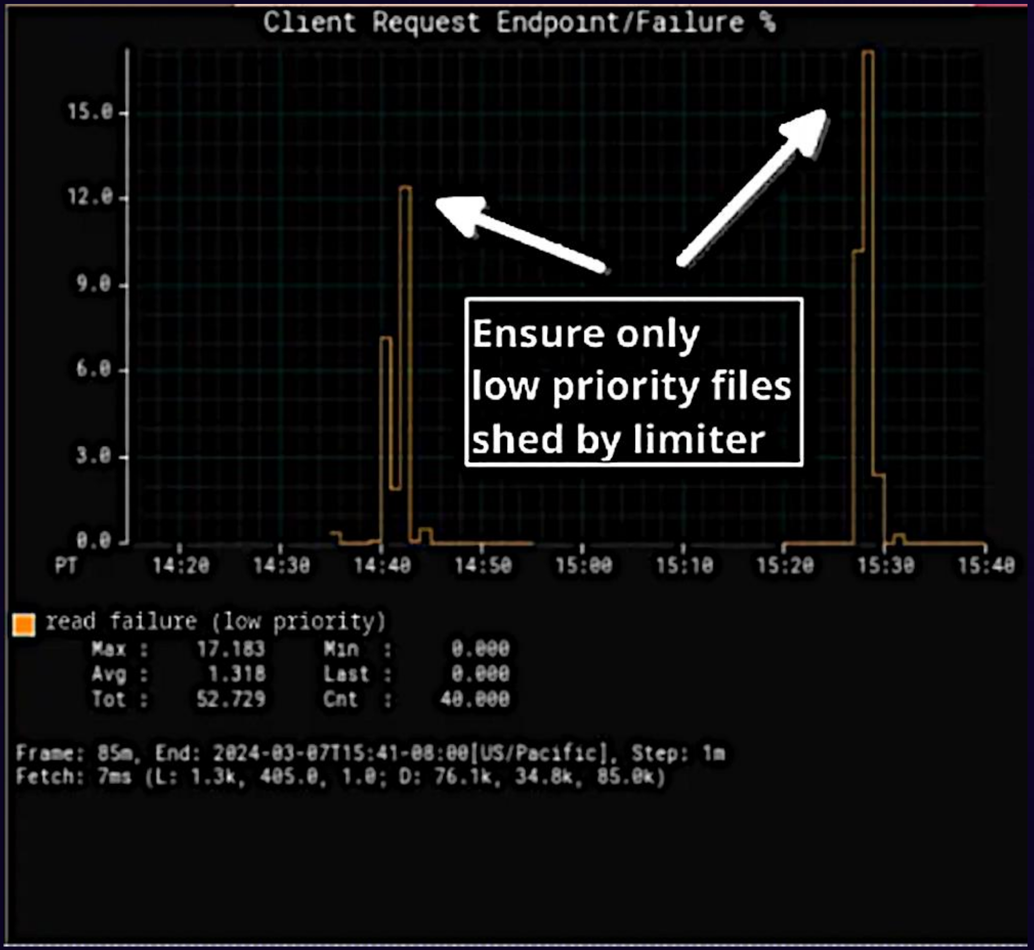



```yaml
resource:
  utilization:
    kv-slo:
      target: 40
      max: 80
  limiter:
    kv-slo:
      enabled: true
      utilization:
        source: kv-slo
        buffer: success
```

Stay up – Add IO limiters



Stay up – Add prioritized IO limiters



Never shed

- High-priority writes

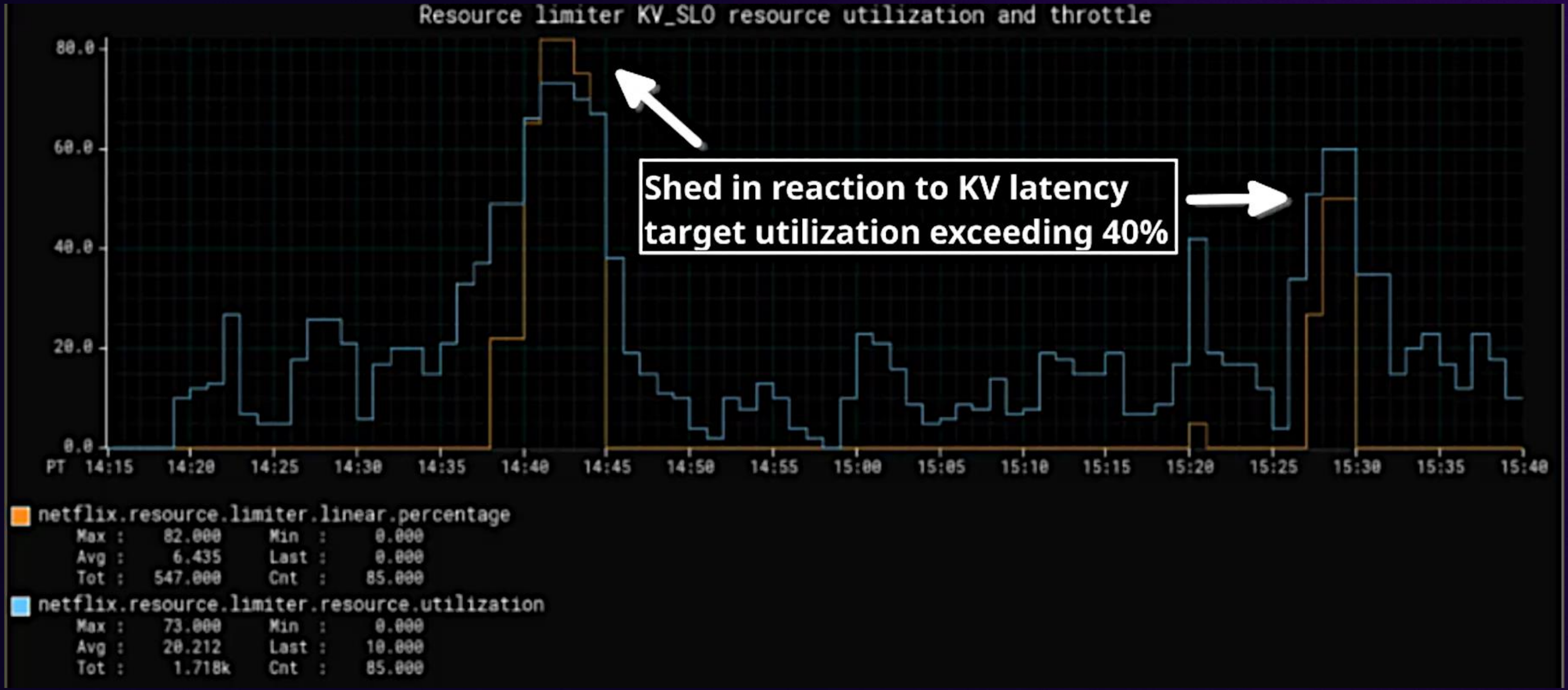
At 80% *max* shed

- High-priority reads

At 40% *target* utilization shed

- Low-priority reads

Stay up – Add IO limiters



Stay up – Prioritized shedding

Success buffer shedding

Prioritized **[CPU]**

Prioritized **[Latency target]**

Failure buffer shedding

Unprioritized **[CPU]**

[Latency timeout]

Generic IO based load-shedding

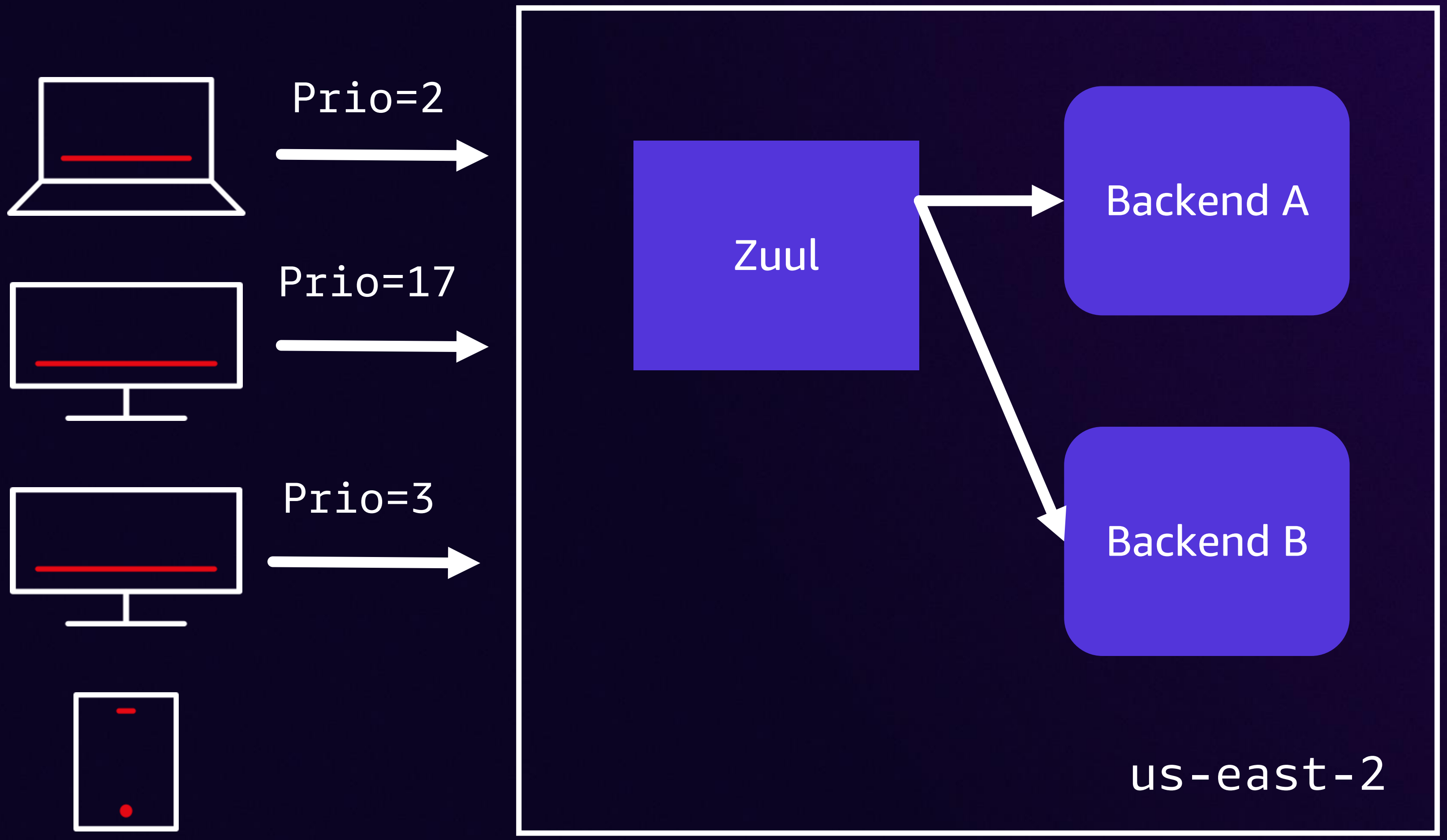
Some services are not CPU-bound but instead are IO-bound by backing services or datastores that can apply back pressure via increased latency when they are overloaded either in compute or in storage capacity. For these services we re-use the prioritized load shedding techniques, but we introduce new utilization measures to feed into the shedding logic. Our initial implementation supports two forms of latency based shedding in addition to standard adaptive concurrency limiters (themselves a measure of average latency):

1. The service can specify per-endpoint target and maximum latencies, which allow the service to shed when the service is abnormally slow regardless of backend.
2. The Netflix storage services running on the Data Gateway return observed storage target and max latency SLO utilization, allowing services to shed when they overload their allocated storage capacity.

These utilization measures provide early warning signs that a service is generating too much load to a backend, and allow it to shed low priority work before it overwhelms that backend. The main advantage of these techniques over concurrency limits alone is they require less tuning as our services already must maintain tight latency service-level-objectives (SLOs), for example a p50 < 10ms and p100 < 500ms. So, rephrasing these existing SLOs as utilizations allows us to shed low priority work early to prevent further latency impact to high priority work. At the same time, the system *will accept as much work as it can* while maintaining SLO's.

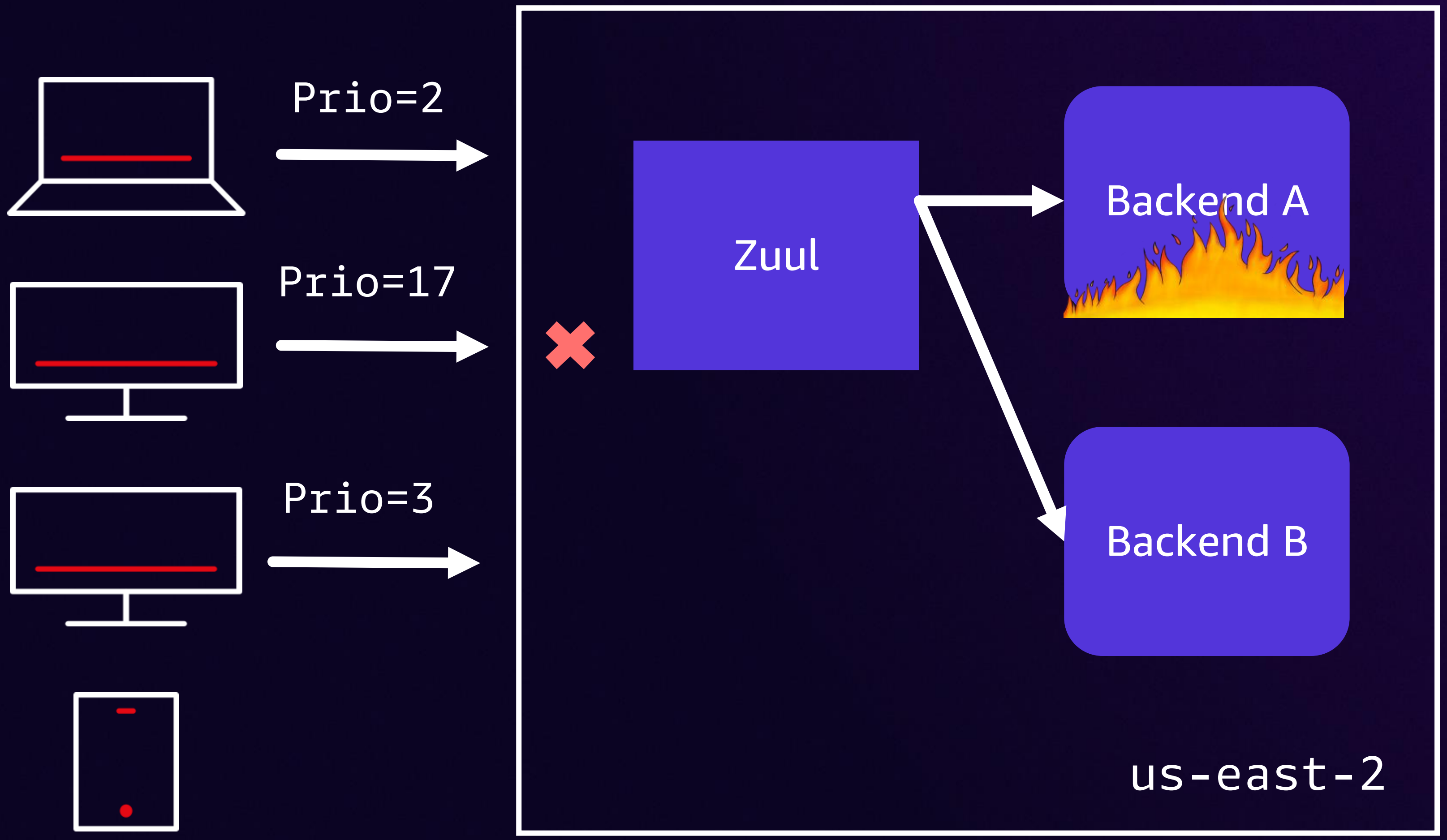
https://netflixtechblog.com/enhancing-netflix-reliability-with-service-level-prioritized-load-shedding-e735e6ce8f7d

Fallbacks!



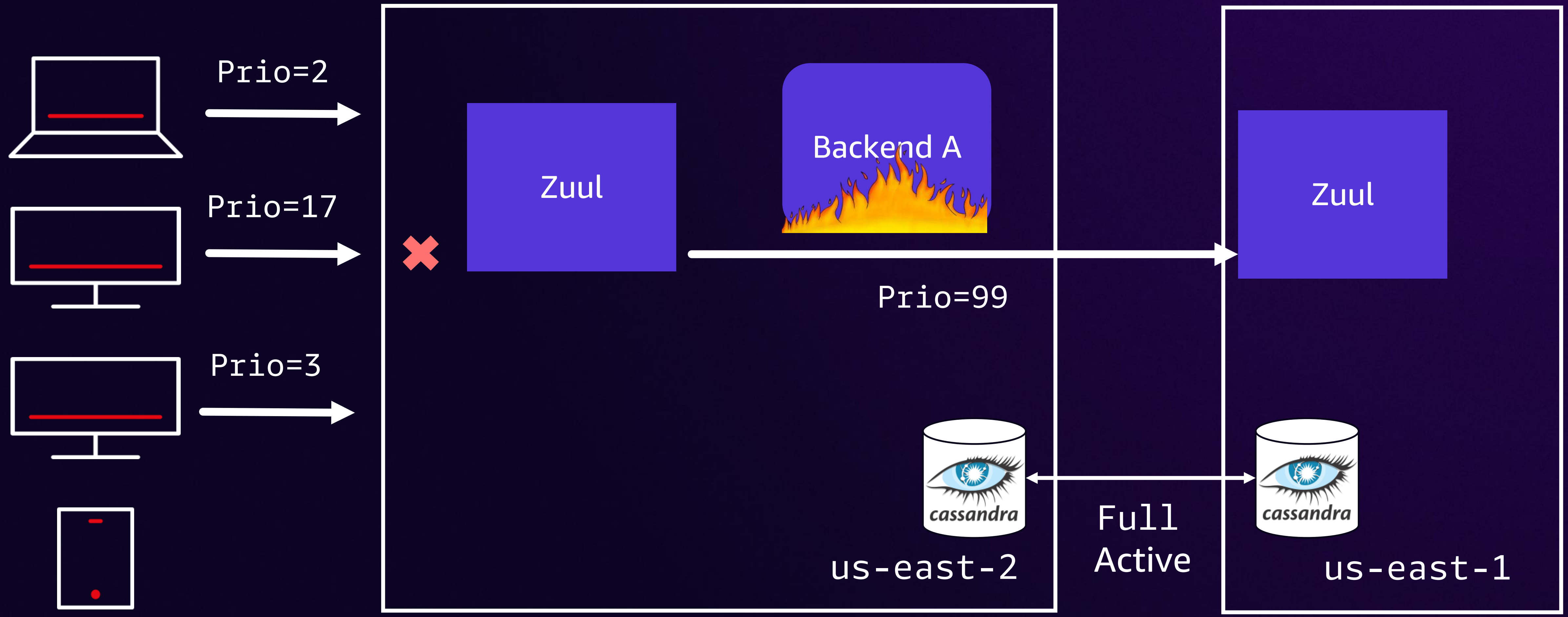
https://netflixtechblog.com/keeping-netflix-reliable-using-prioritized-load-shedding-6cc827b02f94

Fallbacks!



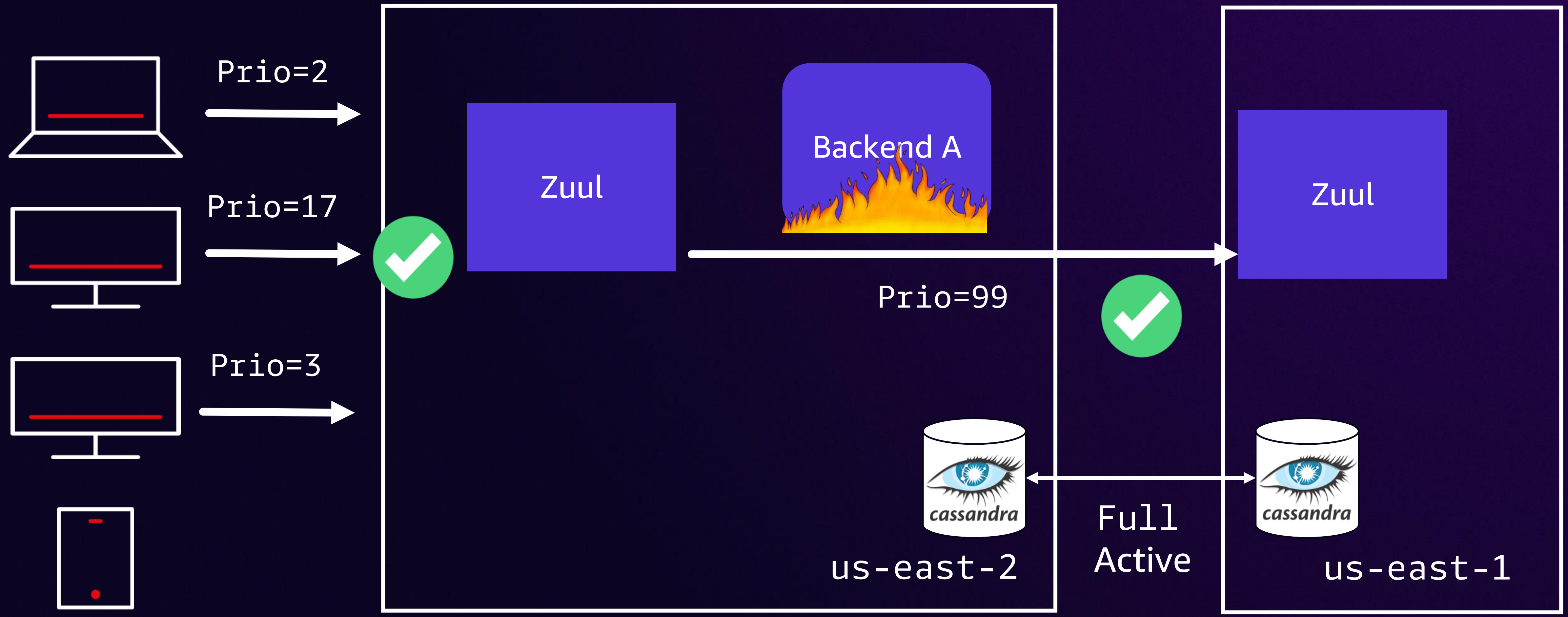
https://netflixtechblog.com/keeping-netflix-reliable-using-prioritized-load-shedding-6cc827b02f94

Fallback and shift



https://netflixtechblog.com/keeping-netflix-reliable-using-prioritized-load-shedding-6cc827b02f94

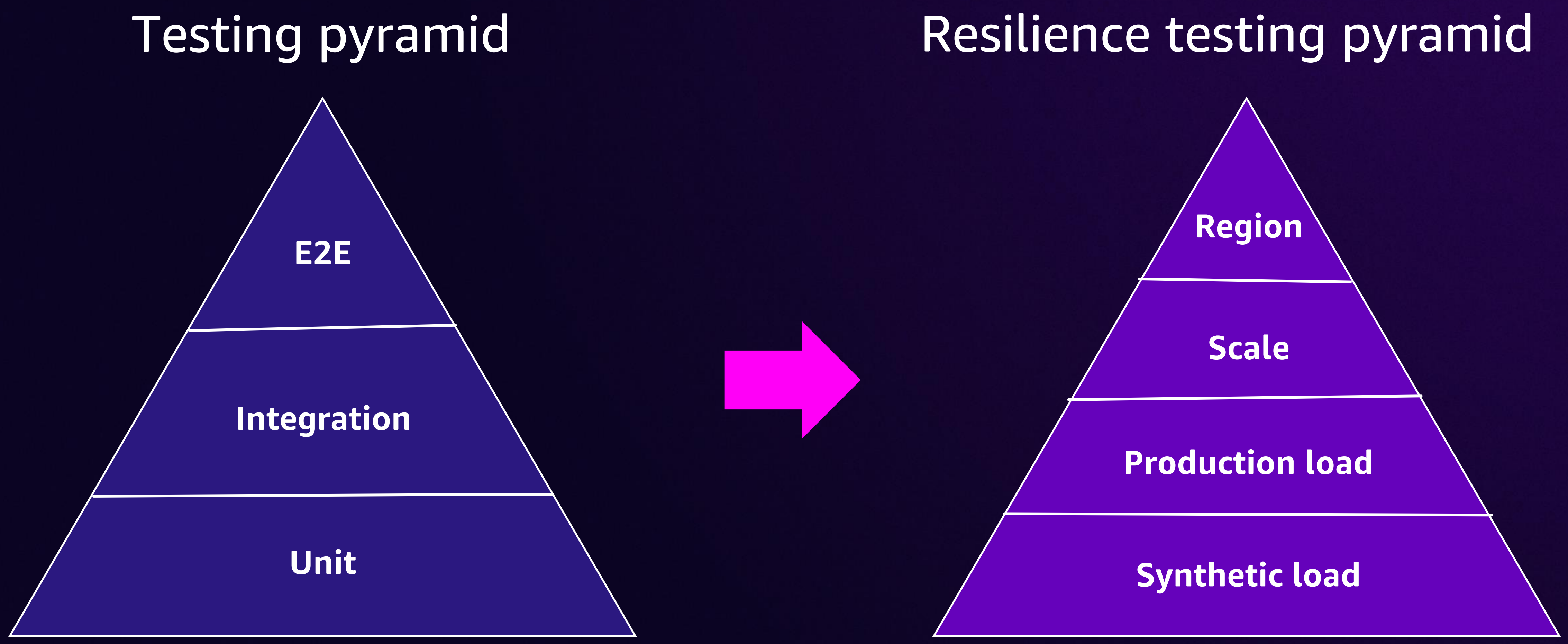
Fallback and shift



https://netflixtechblog.com/keeping-netflix-reliable-using-prioritized-load-shedding-6cc827b02f94

05: Resilience testing
Validating the techniques

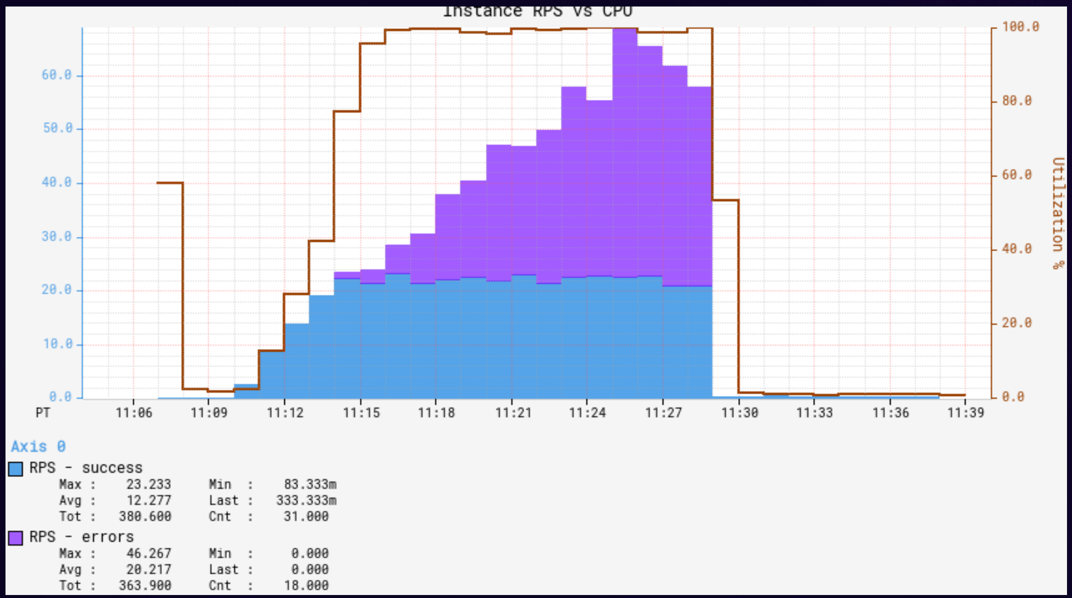
The resilience testing pyramid

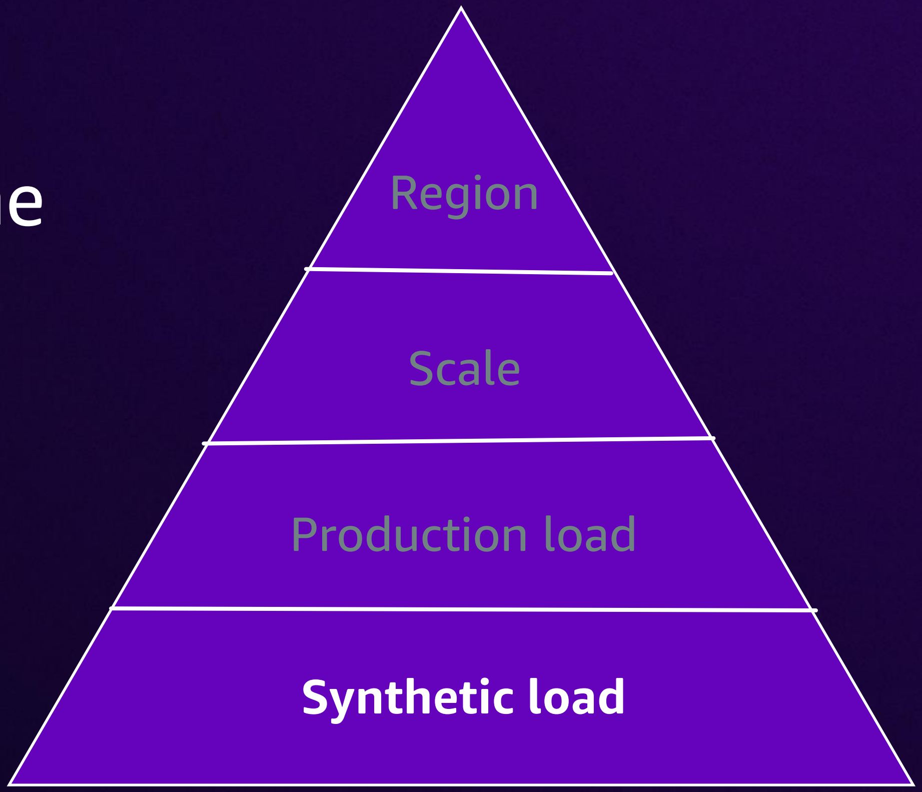


Service-level synthetic load testing

Use synthetic traffic to test an application in isolation

Find bottlenecks in application code and tune load shedding configs



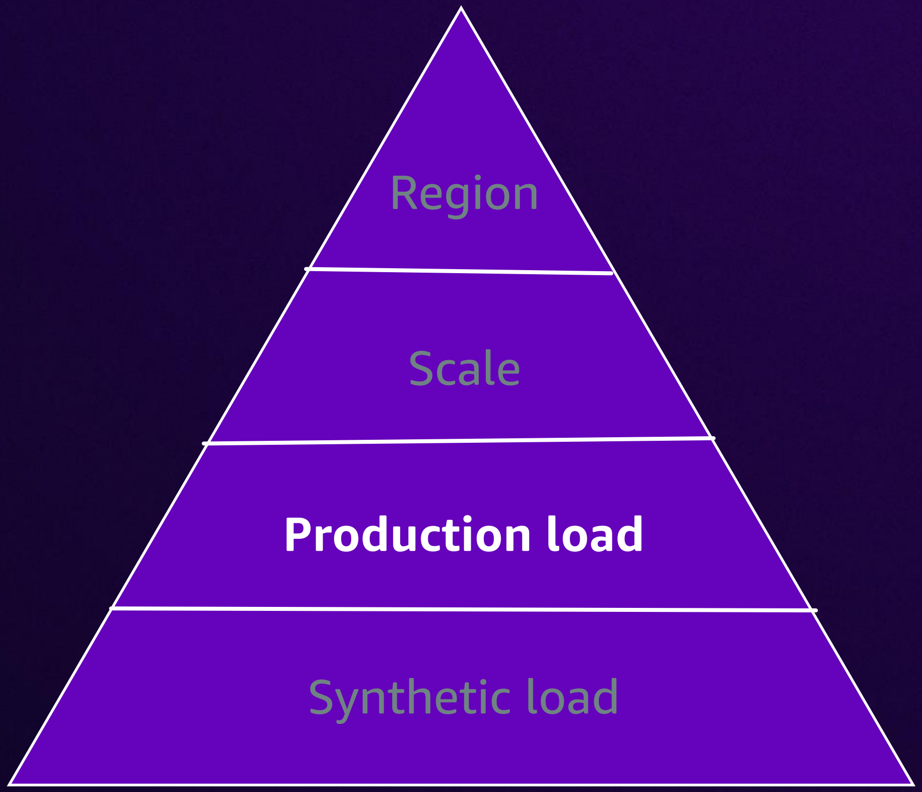


Production load testing

Autoscaling squeeze test through our Chaos Automation Platform

Introduces a load spike to a service to test how load shedding and autoscaling behave

Tests the actual production config with real traffic



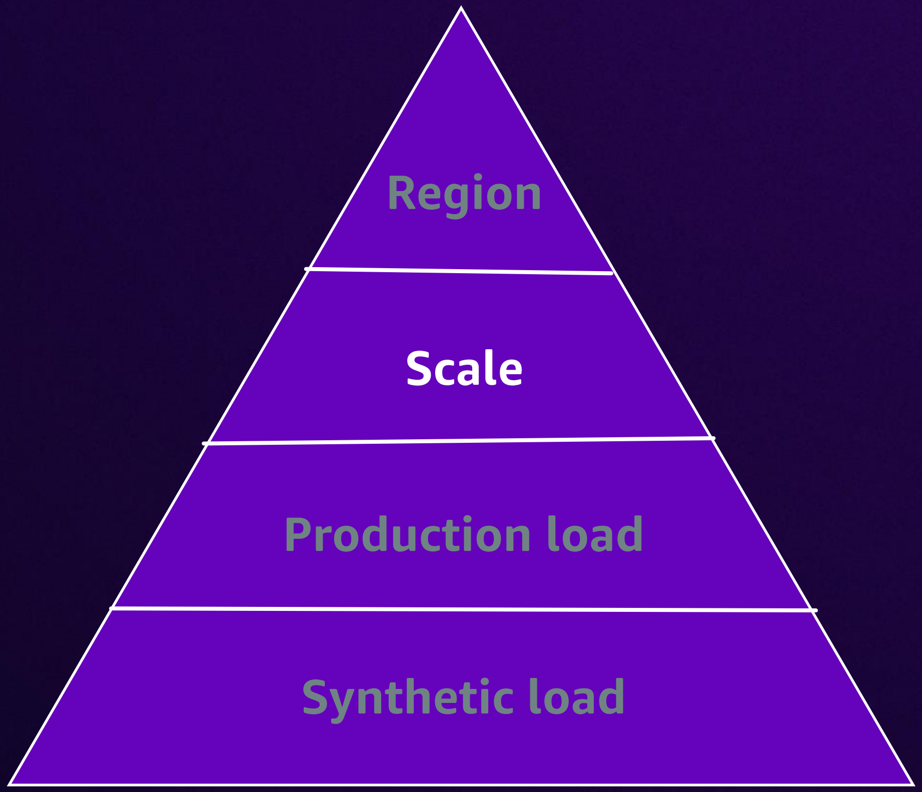
Region scale

Move all global traffic into a single region

Uses regional failover tooling

Finds issues only seen at scale: load that scales with:

- # of instances
- # of RPS



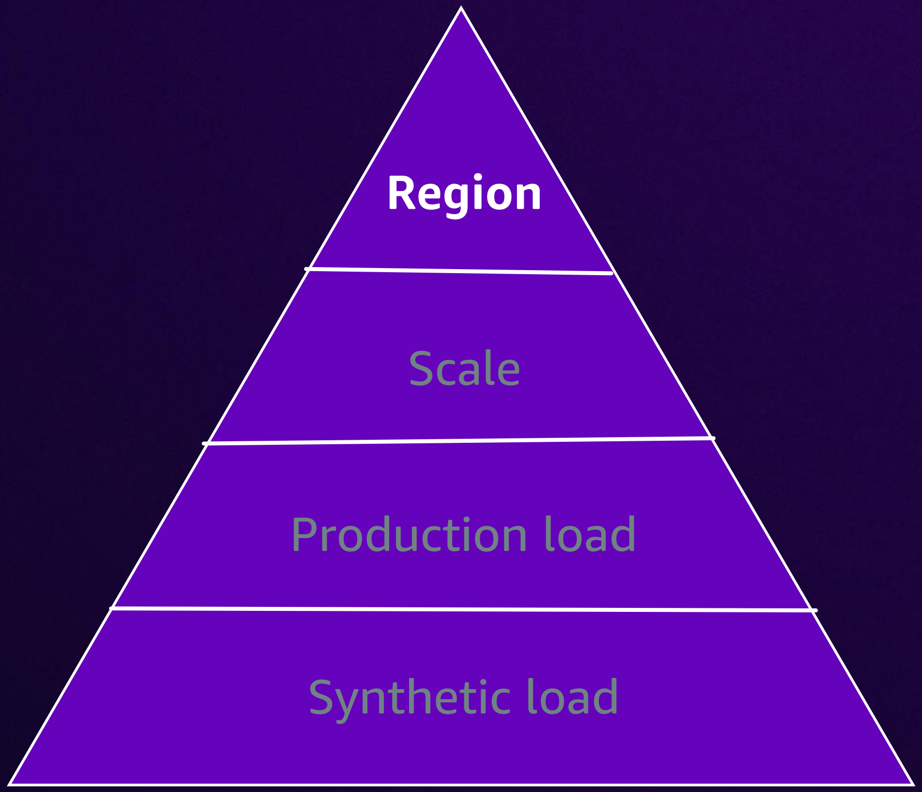
Region load testing

E2E tests that simulate user behavior

Uses synthetic traffic against the production Netflix API

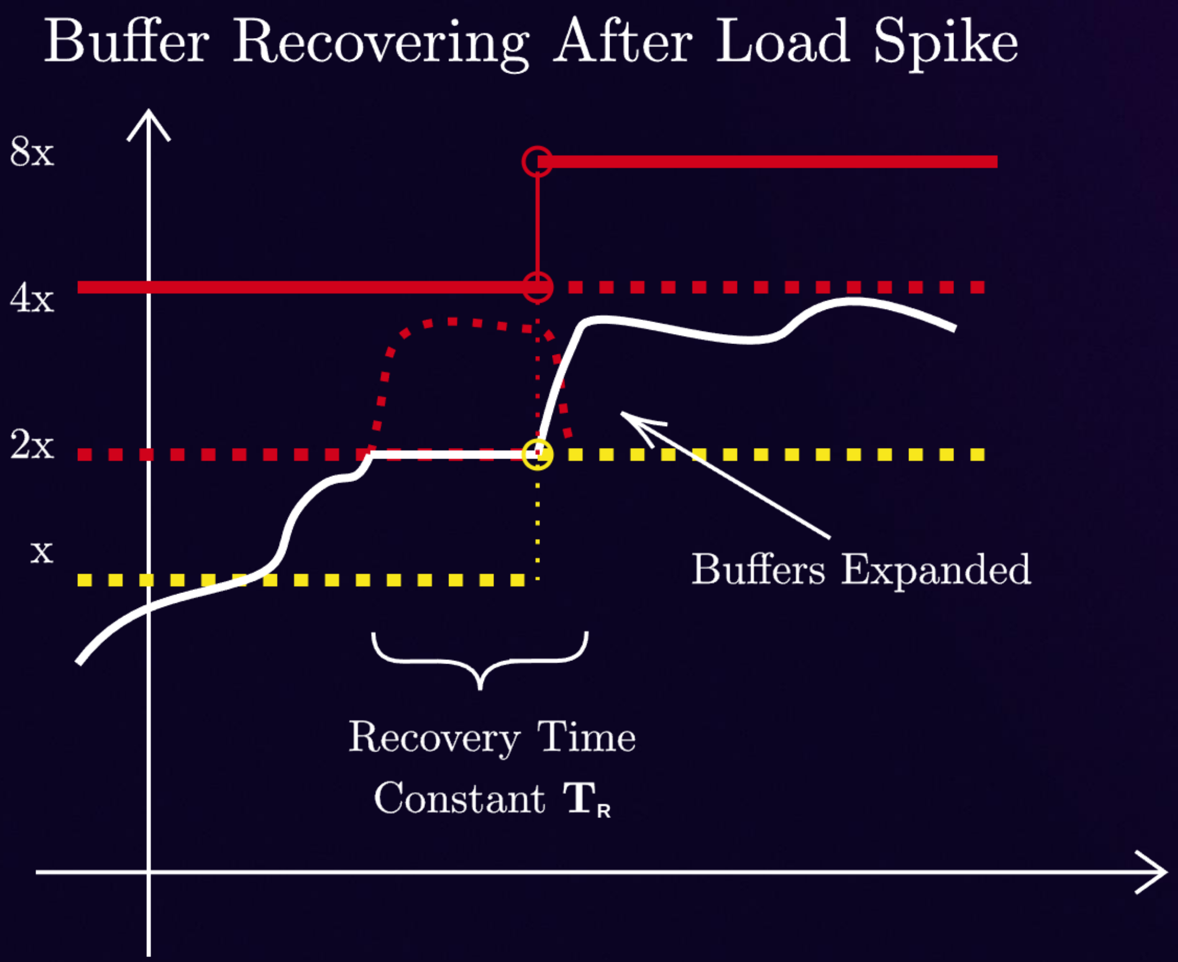
Simulate title launches and failure scenarios

Lets us test scales that are even bigger than our current global peak



06: Conclusions and wrap-up

Measuring success



Goals:

- Reduce time to recover
- Use regional failover less as the primary remediation
- Build resiliency assuming load spikes are the norm

Takeaways

Handling load spikes is a mix of proactive and reactive mechanisms: investing in both is important!

Use your existing compute resources to answer only the most important requests. Fail quickly when overloaded.

Test. In Prod. As often as possible.

Thank you!

Please complete the session survey in the mobile app

Rob Gulewich
rgulewich@netflix.com

Ryan Schroeder
rschroeder@netflix.com

Joseph Lynch
josephl@netflix.com
jolynch.github.io/