
AWS
re:Invent
DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

ARC403

# Try again: The tools and techniques behind resilient systems

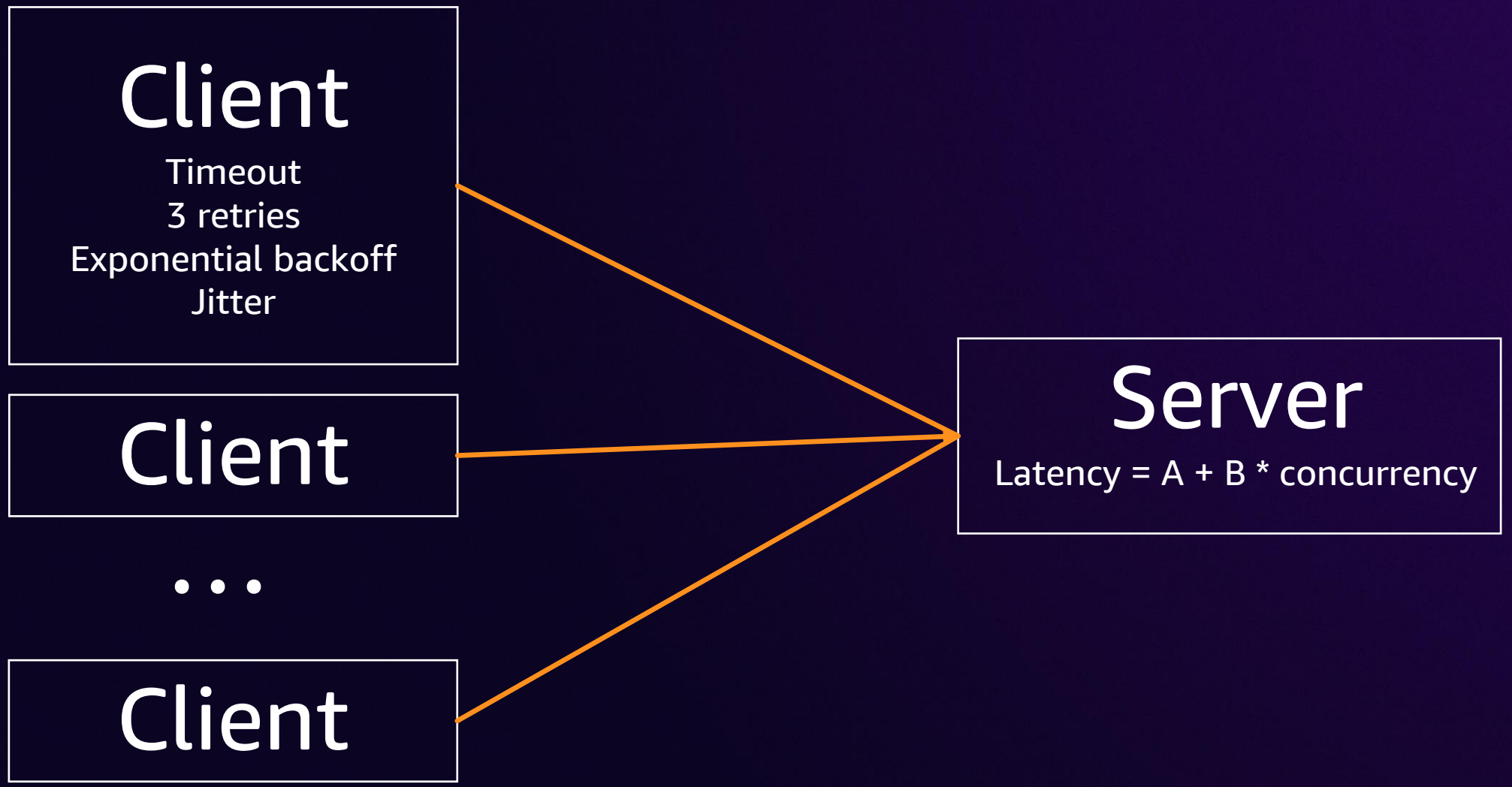
**Marc Brooker**

VP/Distinguished Engineer
Amazon Web Services

# Agenda

- The trouble with retries, and what to do instead
- Breaking circuits
- Erasure coding vs. the tail
- A theory of stable systems
- Simulation, a numerical method in hiding

# Rethinking retries



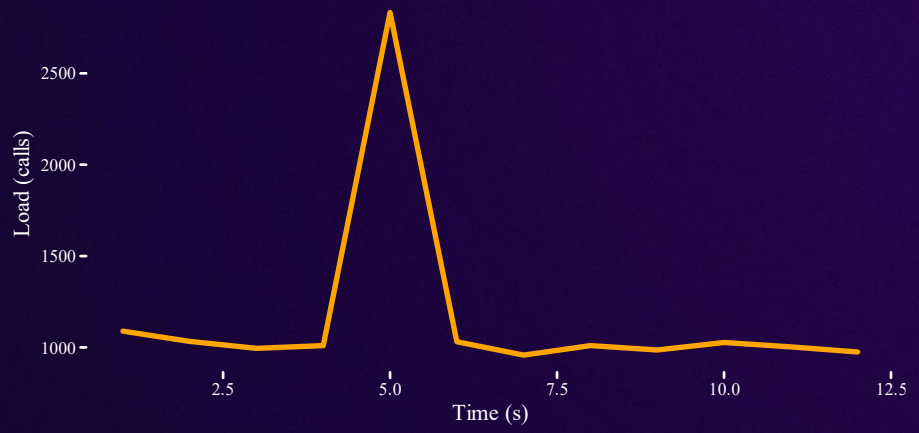
Client

Timeout
3 retries
Exponential backoff
Jitter

Client

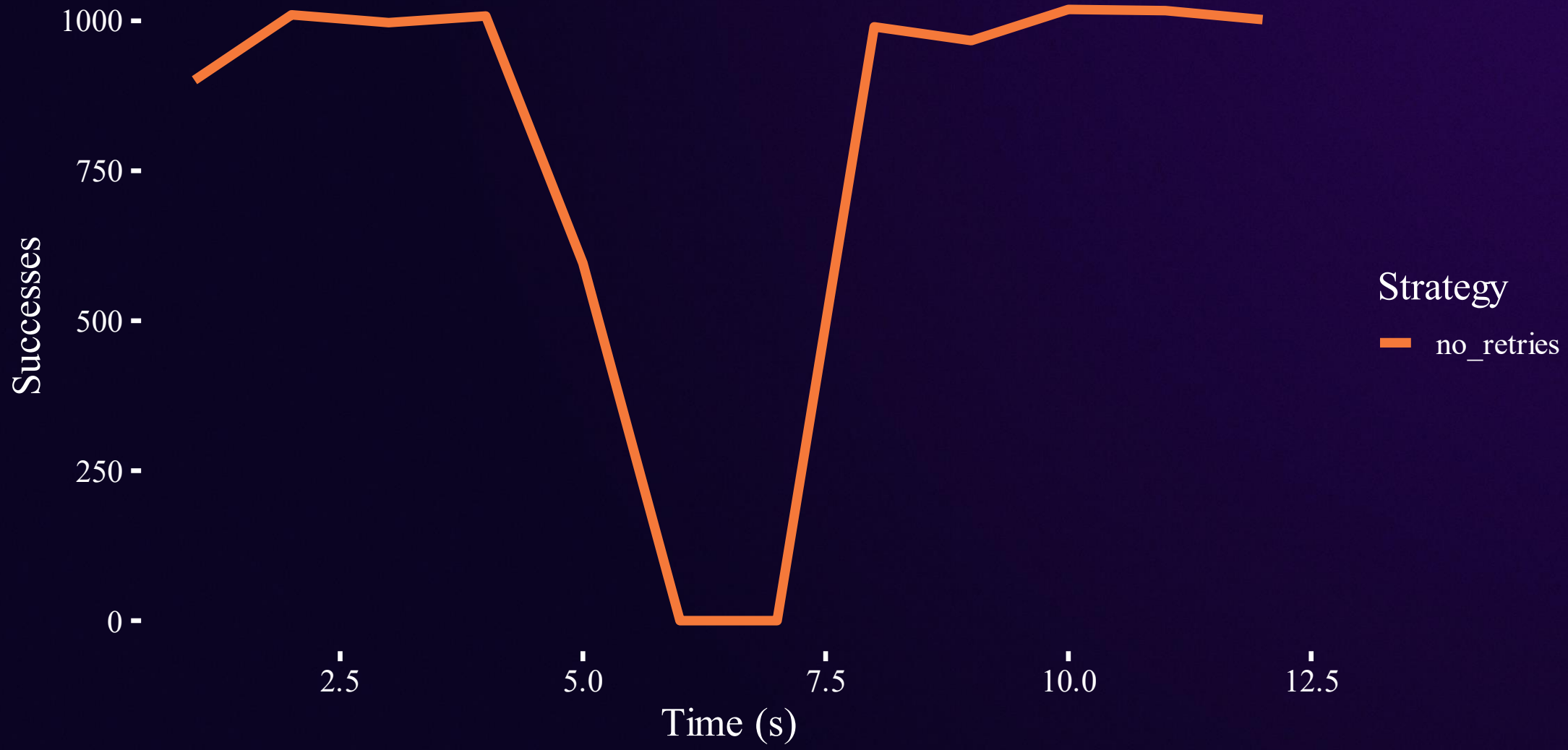
• • •

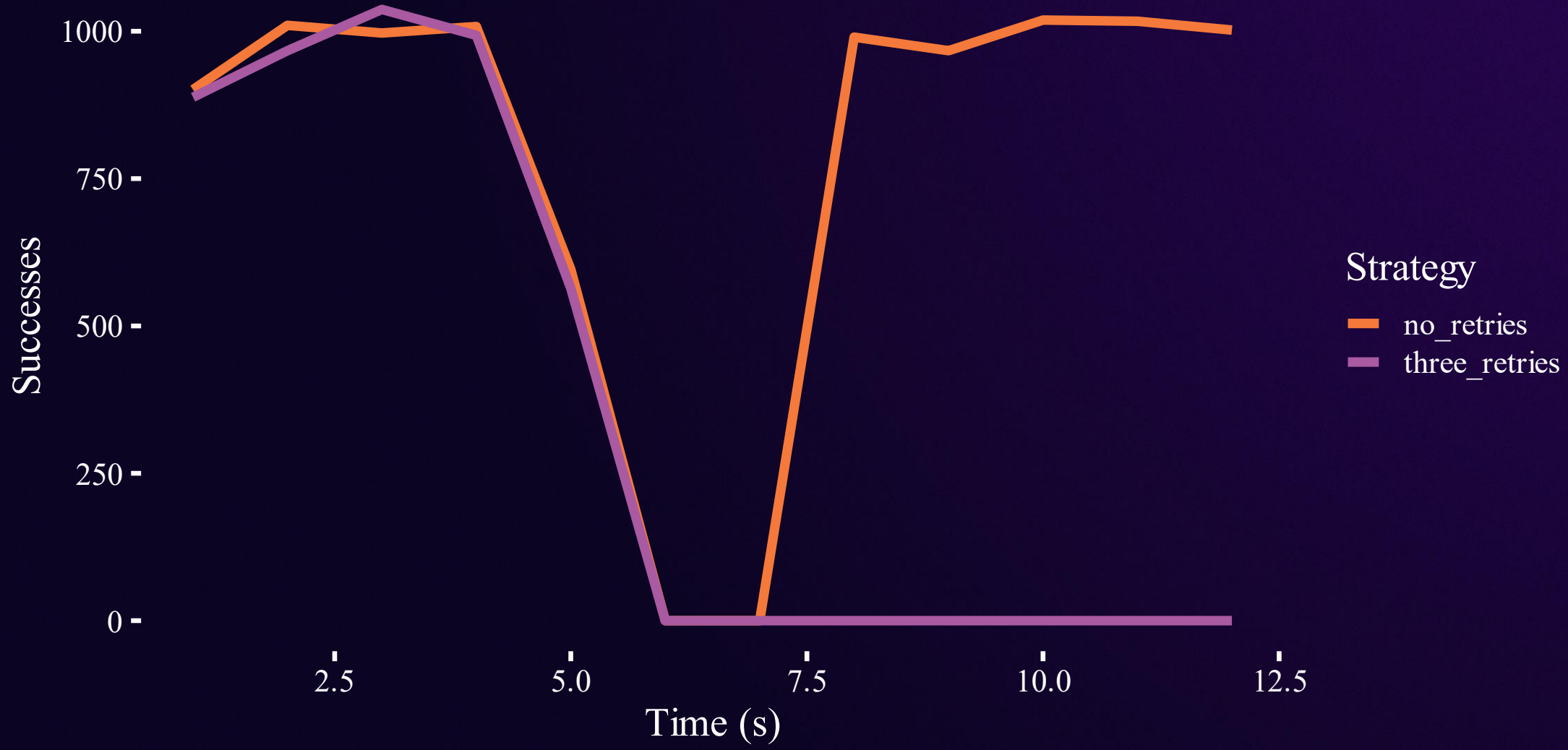
Client




Server

Latency = A + B * concurrency





## Transient failures

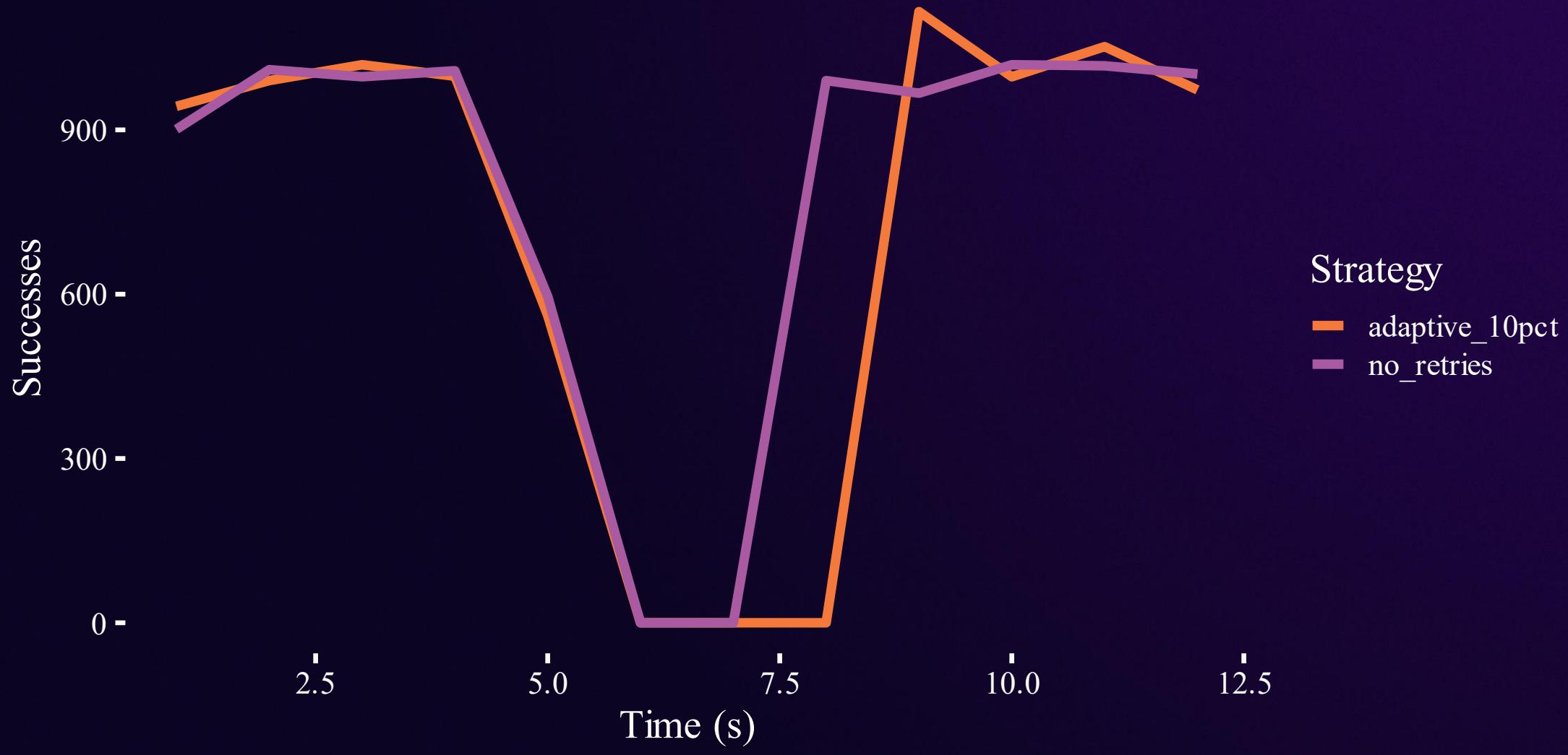
Common in redundant systems, caused by individual component failures

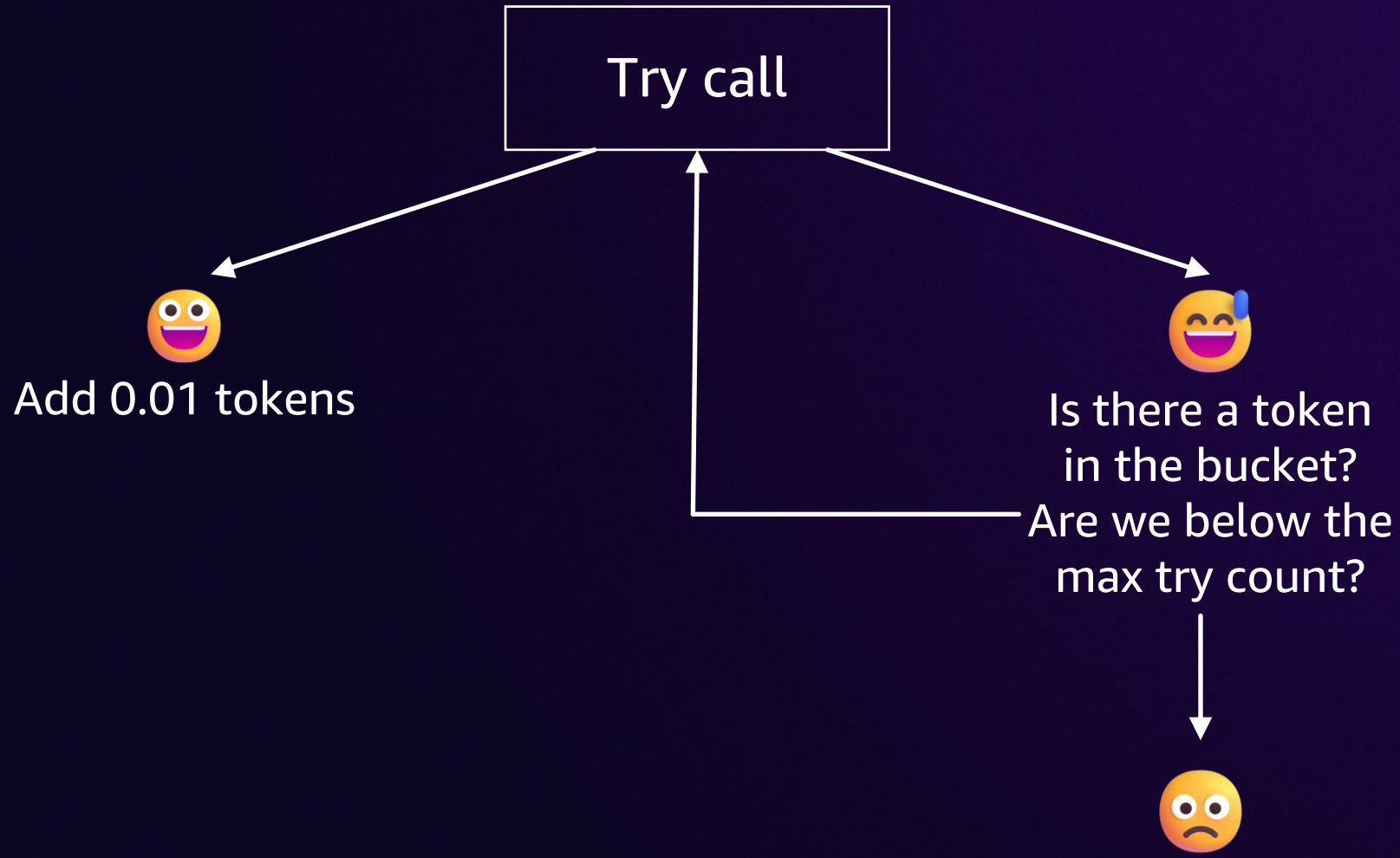
Retries help

## Systemic failures

Caused by load, correlated behavior, software bugs, operational issues, and so on

Retries harm





🙈 **No retries:** Great for systemic, no help for transient

♻️ **3 retries:** Terrible for systemic, great for transient

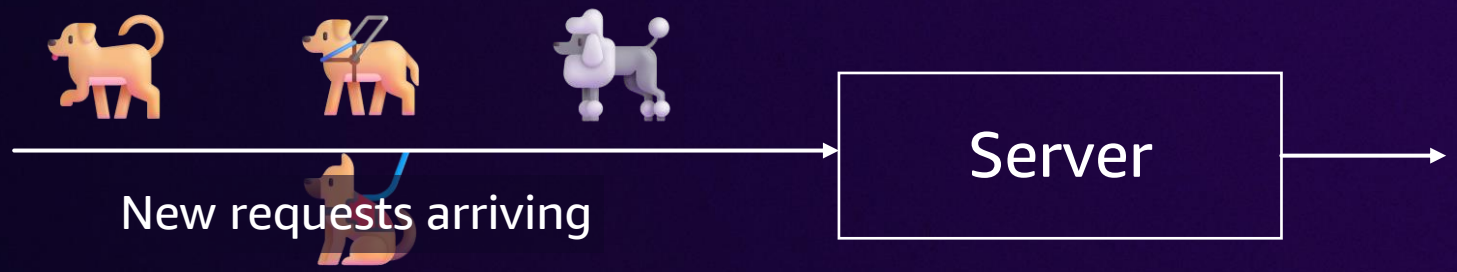
🪣 **Adaptive:** Great for systemic, great for transient

# A tiny change in algorithm avoids whole classes of **metastable** system behavior

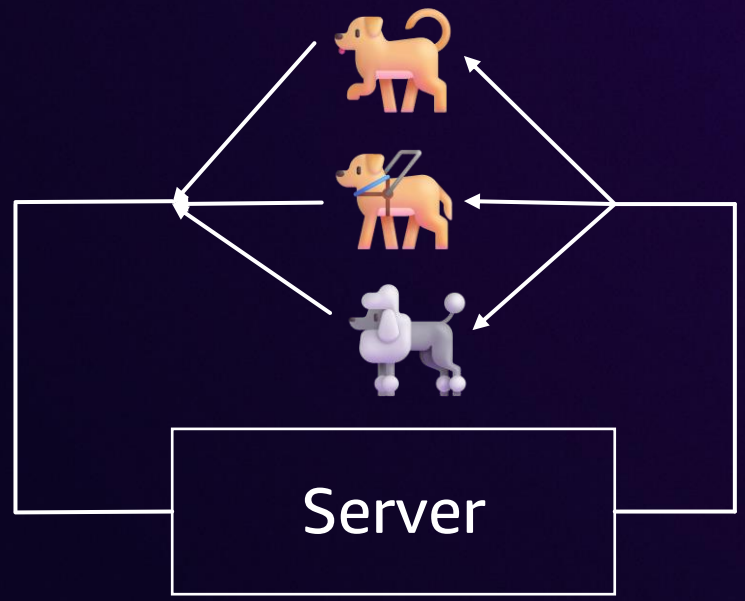
# Built into the AWS SDKs!

# What about backoff and jitter?

# Open system




# Closed system






Adapted from *Open versus Closed: A Cautionary Tale* by Open Versus Closed: A Cautionary Tale by Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter

Backoff is very effective in closed systems . . .

Mostly ineffective in open systems . . .

**Jitter is always a good idea**

# What about circuit breakers?

## Reducing harvest

Remove non-essential functionality (such as UI elements) to preserve availability

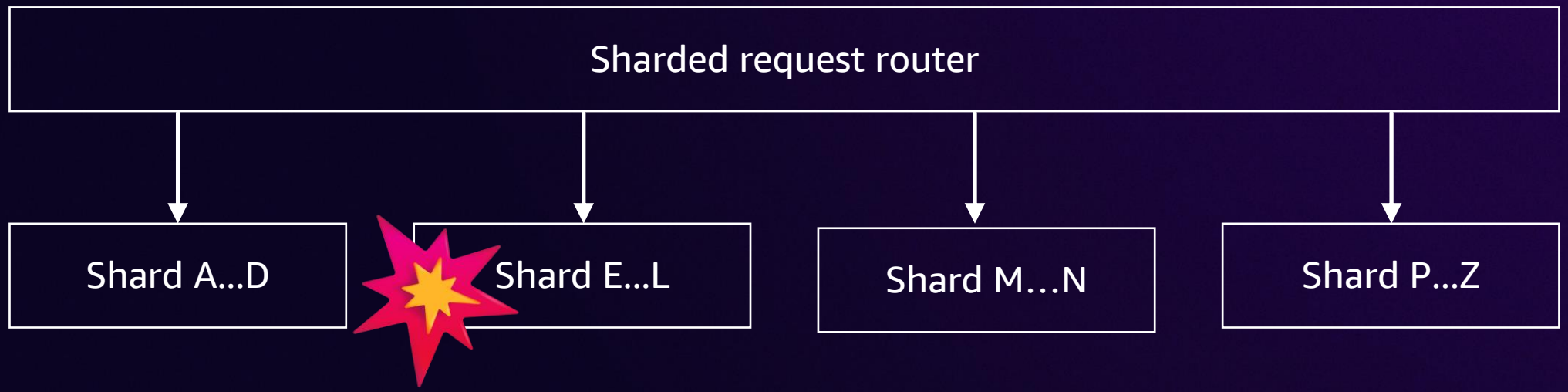
## Rejecting load

Sacrificing availability to reduce traffic to a downstream system, hoping to reduce contention

Circuit breakers can allow systems to avoid congestion and congestive collapse.

But . . .

# Reduce reliability of sharded systems



# Amazon DynamoDB: ATC'22

## Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service

*Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog*
*Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam ,Tim Rath*
*Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, Akshat Vig*
*dynamodb-paper@amazon.com*
Amazon Web Services

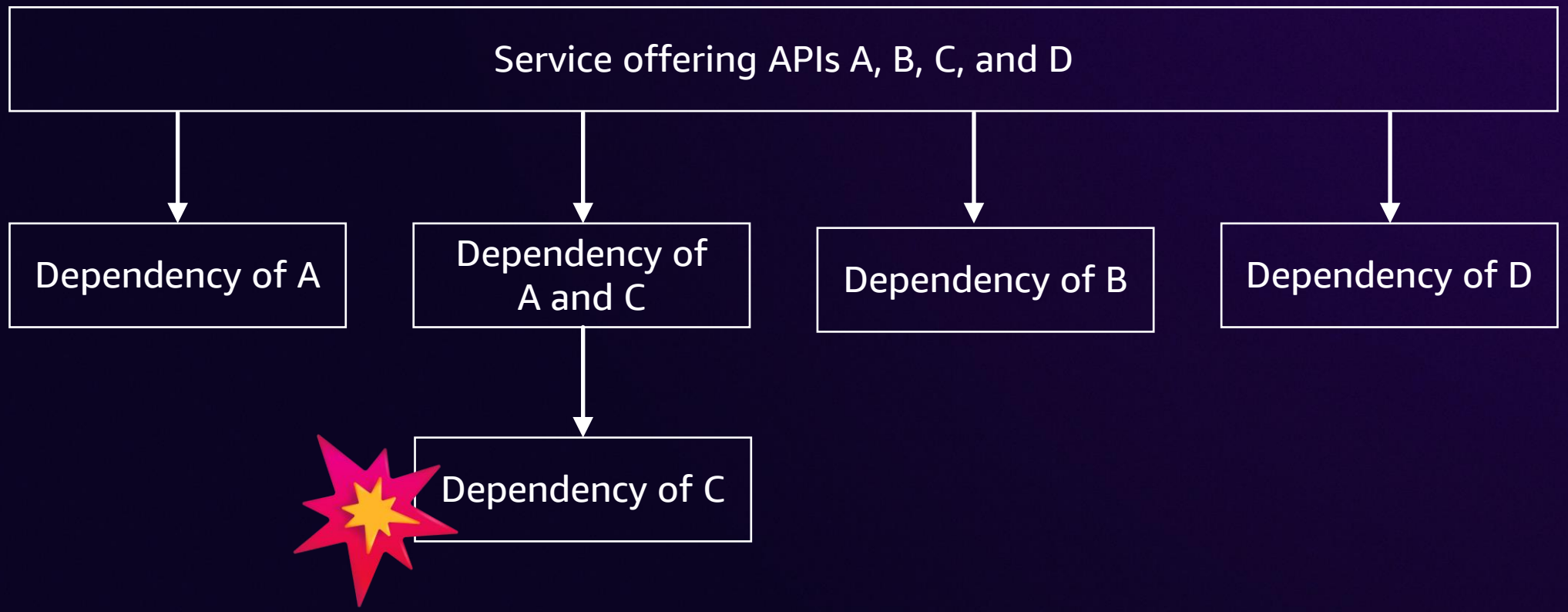
### Abstract

Amazon DynamoDB is a NoSQL cloud database service that provides consistent performance at any scale. Hundreds of thousands of customers rely on DynamoDB for its fundamental properties: consistent performance, availability, durability, and a fully managed serverless experience. In 2021, during the 66-hour Amazon Prime Day shopping event, Amazon systems - including Alexa, the Amazon.com sites, and Amazon fulfillment centers, made trillions of API calls to DynamoDB, peaking at 89.2 million requests per second, while experiencing high availability with single-digit millisecond performance. Since the launch of DynamoDB in 2012, its design and implementation have evolved in response to our experiences operating it. The system has successfully dealt with issues related to fairness, traffic imbalance across partitions, moni-

on its ability to serve requests with consistent low latency. For DynamoDB customers, consistent performance at any scale is often more important than median request service times because unexpectedly high latency requests can amplify through higher layers of applications that depend on DynamoDB and lead to a bad customer experience. The goal of the design of DynamoDB is to complete *all* requests with low single-digit millisecond latencies. In addition, the large and diverse set of customers who use DynamoDB rely on an ever-expanding feature set as shown in Figure 1. As DynamoDB has evolved over the last ten years, a key challenge has been adding features without impacting operational requirements. To benefit customers and application developers, DynamoDB uniquely integrates the following six fundamental system properties:

*DynamoDB is a fully managed cloud service.* Using the DynamoDB API, applications create tables and read and write

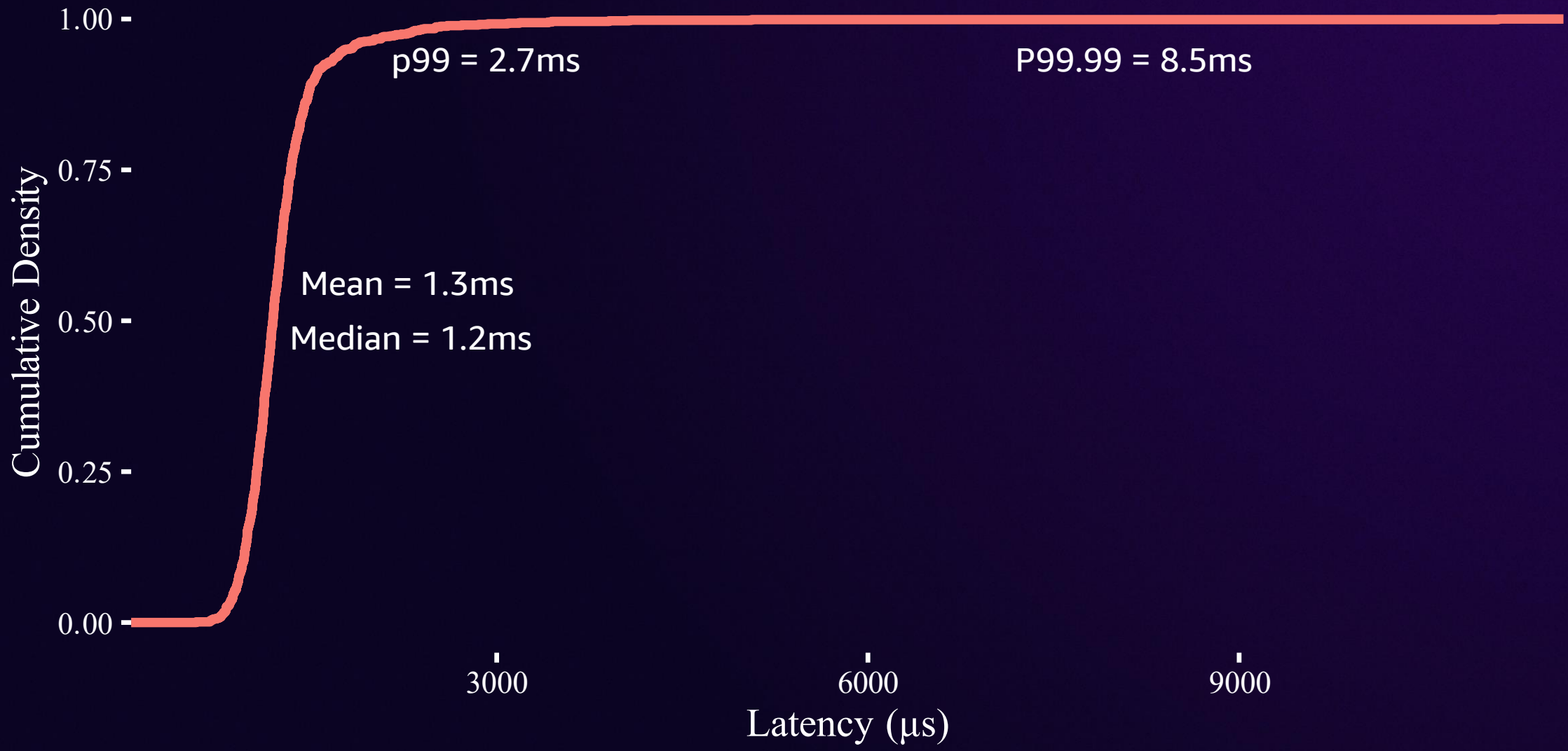
# Expand blast radius in SOA and microservices

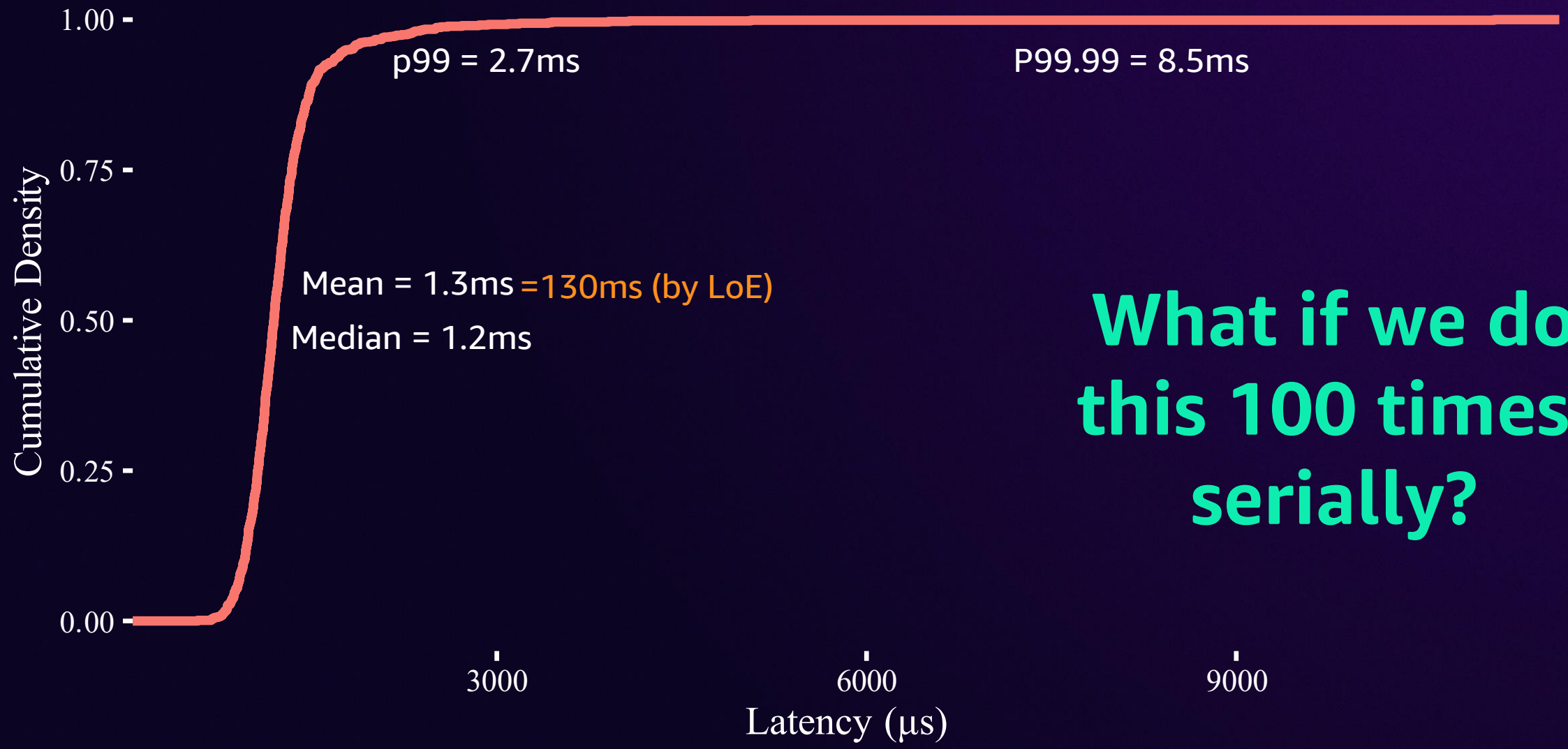


# Circuit breaker lessons

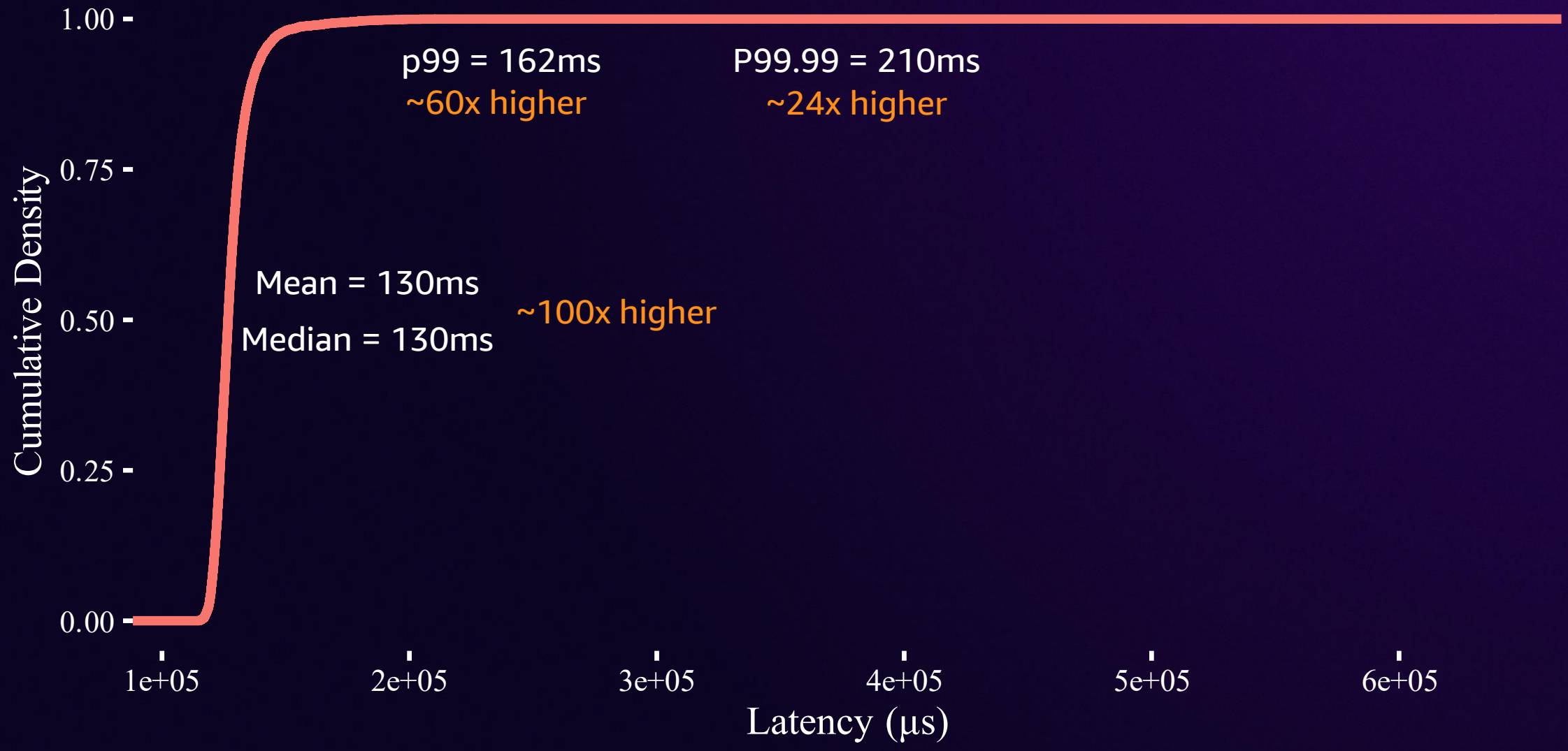
- Avoid binary on/off circuit breakers
- Prefer algorithms like AIMD, which adapt to downstream capacity
- Avoid high blast-radius circuit breakers
- Successful circuit breakers might need to know internal details of the downstream 🤢

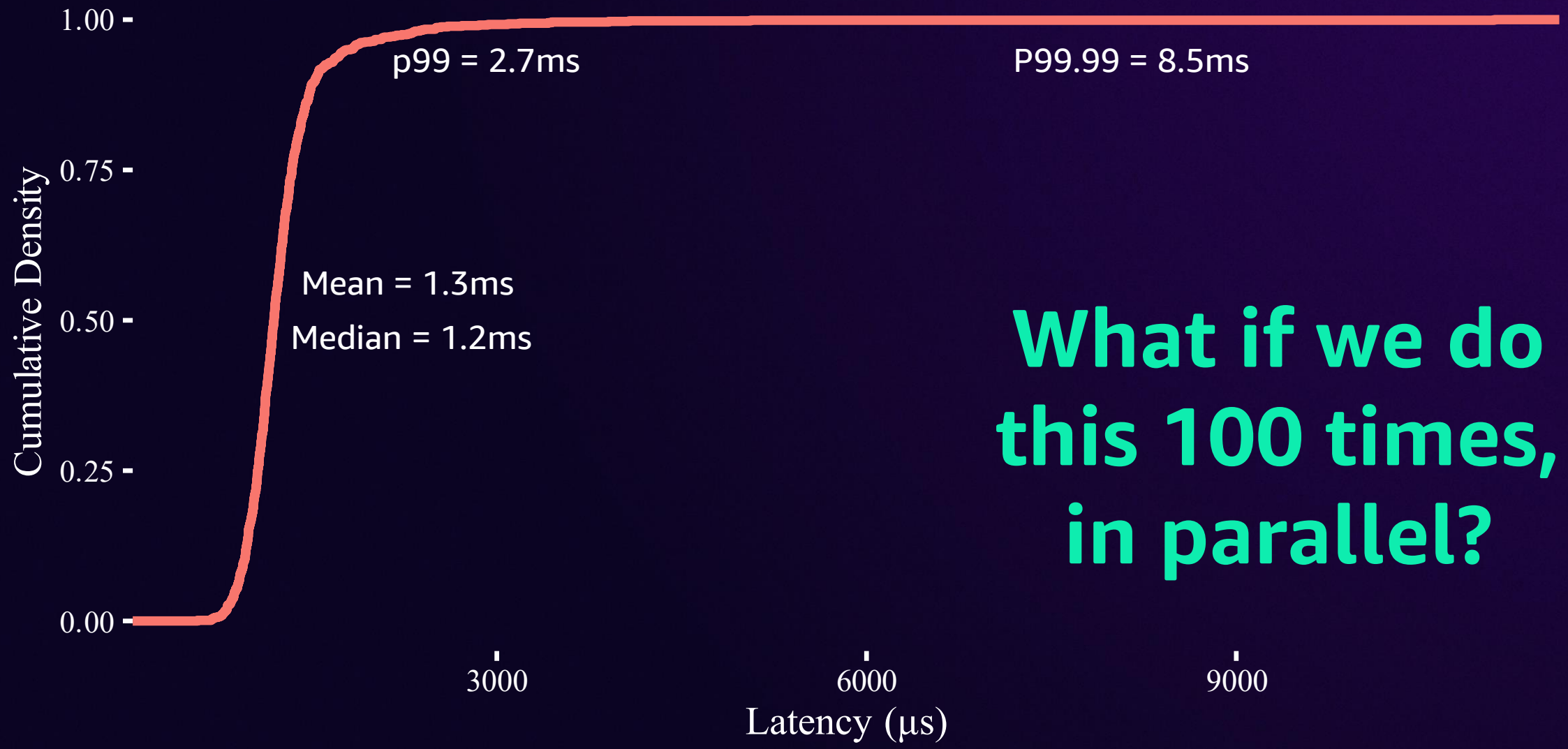
# Tackling the tail






**What if we do this 100 times, serially?**






# What if we do this 100 times, in parallel?

# Tail tool: First response of 2

- Send two requests
- Use the first response

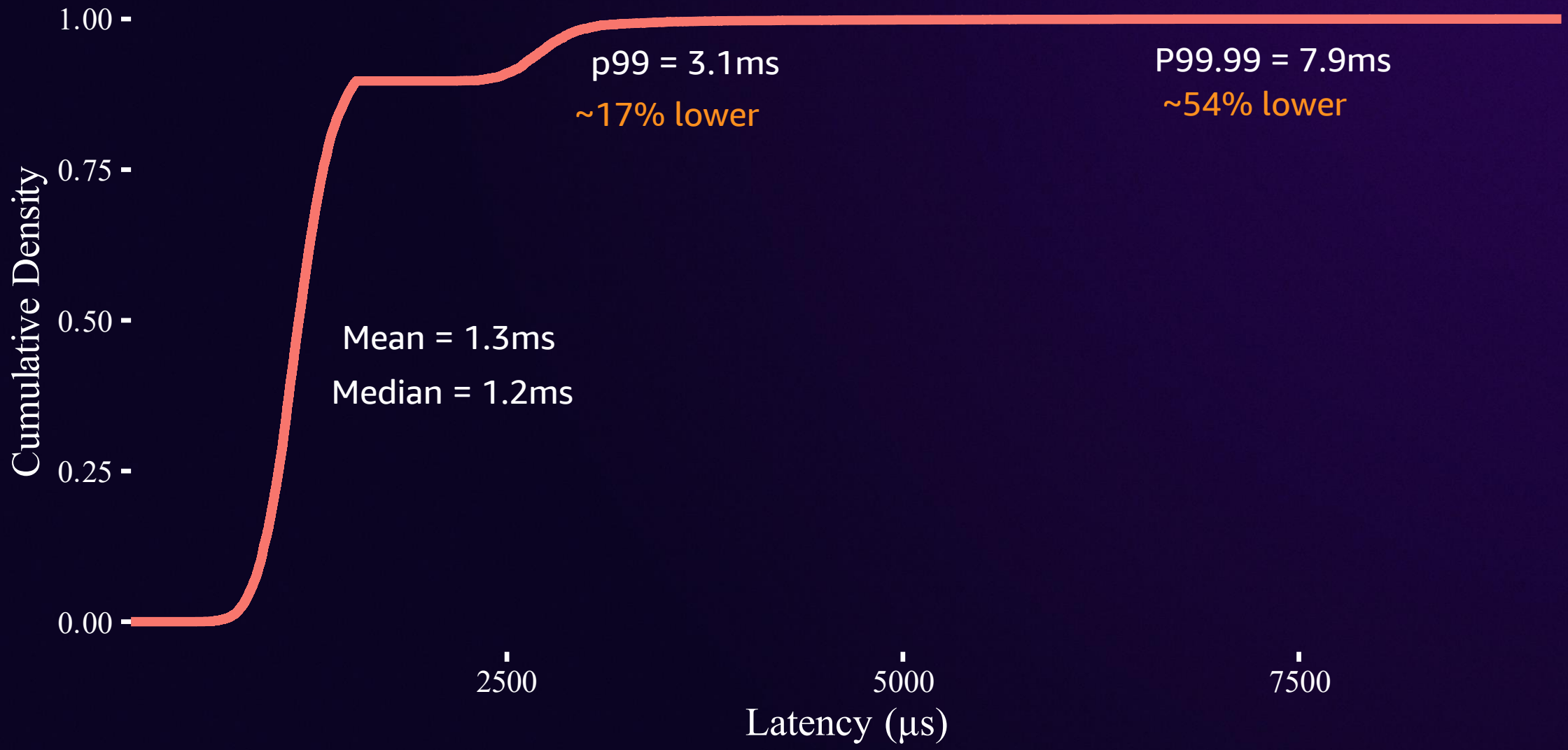
# Tail tool: First response of 2

- Send two requests
- Use the first response

- **Cost:** Double the throughput, double the request volume!

# Tail tool: Hedging

- Send one request
- If it doesn’t come back soon, send another
- Use the first result



# Tail tool: Hedging

- Send one request
- If it doesn’t come back soon (> p90), send another
- Use the first result

**Cost:**

- ~10% extra request rate and throughput
- 2x extra storage

**Introduces metastable failure modes!**

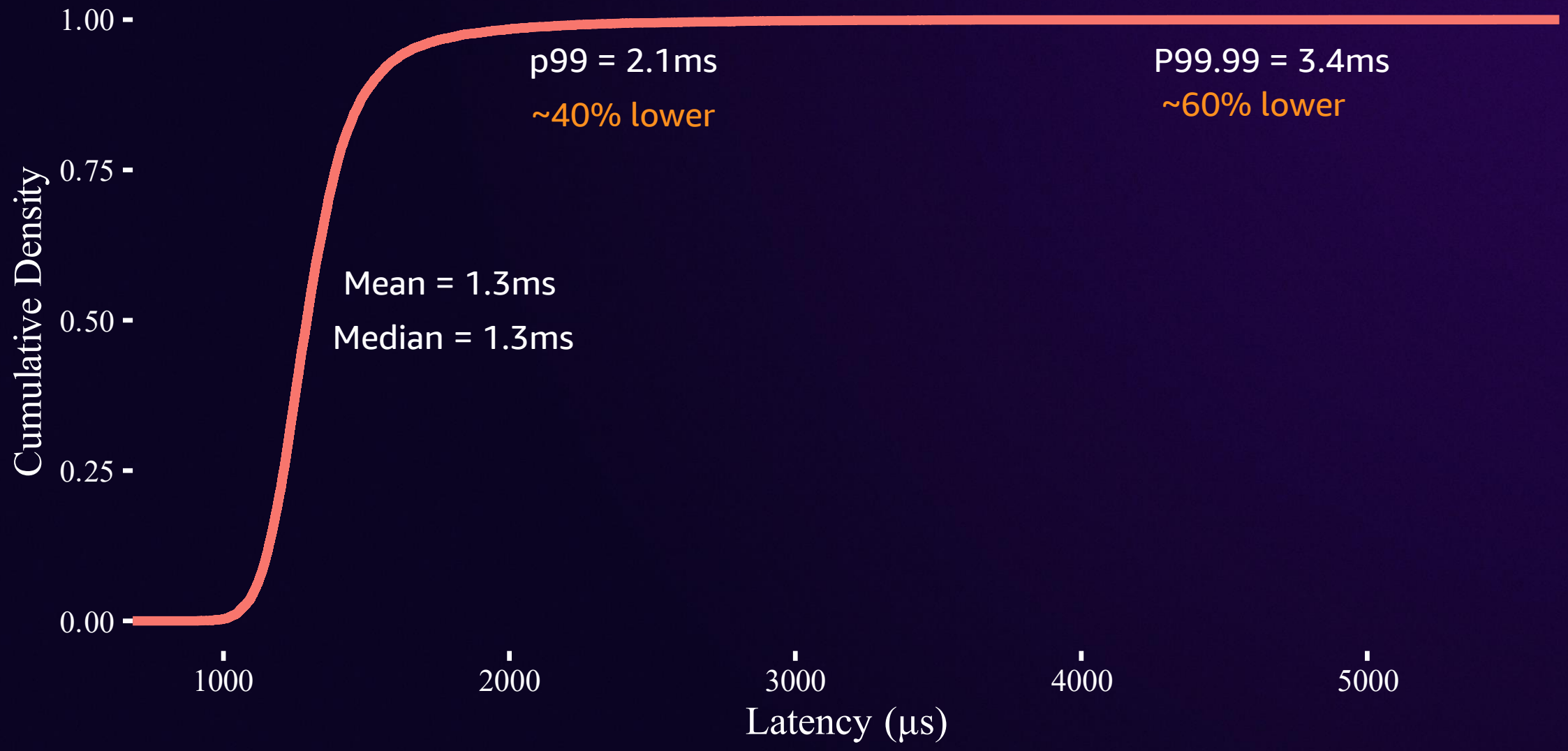
# Tail tool: Erasure coding

- Send *N* requests
- Reassemble result from first *k*

**Cost:**

- *N* times the request rate
- *N/k* times the bandwidth
- N/k extra storage

**Constant work!**



# On-demand container loading in AWS Lambda

Best paper!

## On-demand Container Loading in AWS Lambda

Marc Brooker
*Amazon Web Services*

Mike Danilov
*Amazon Web Services*

Chris Greenwood
*Amazon Web Services*

Phil Piwonka
*Amazon Web Services*

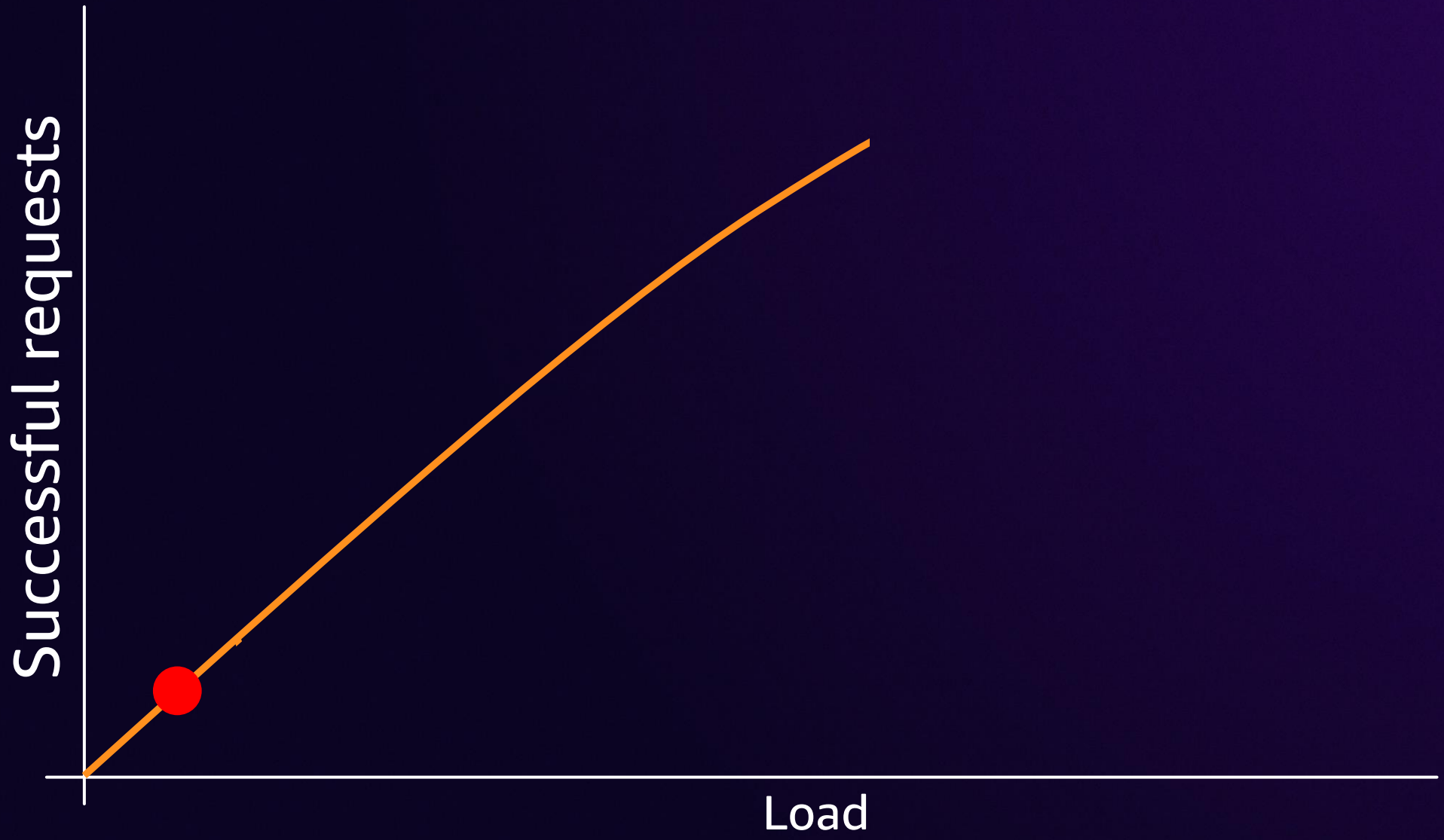
**Abstract**

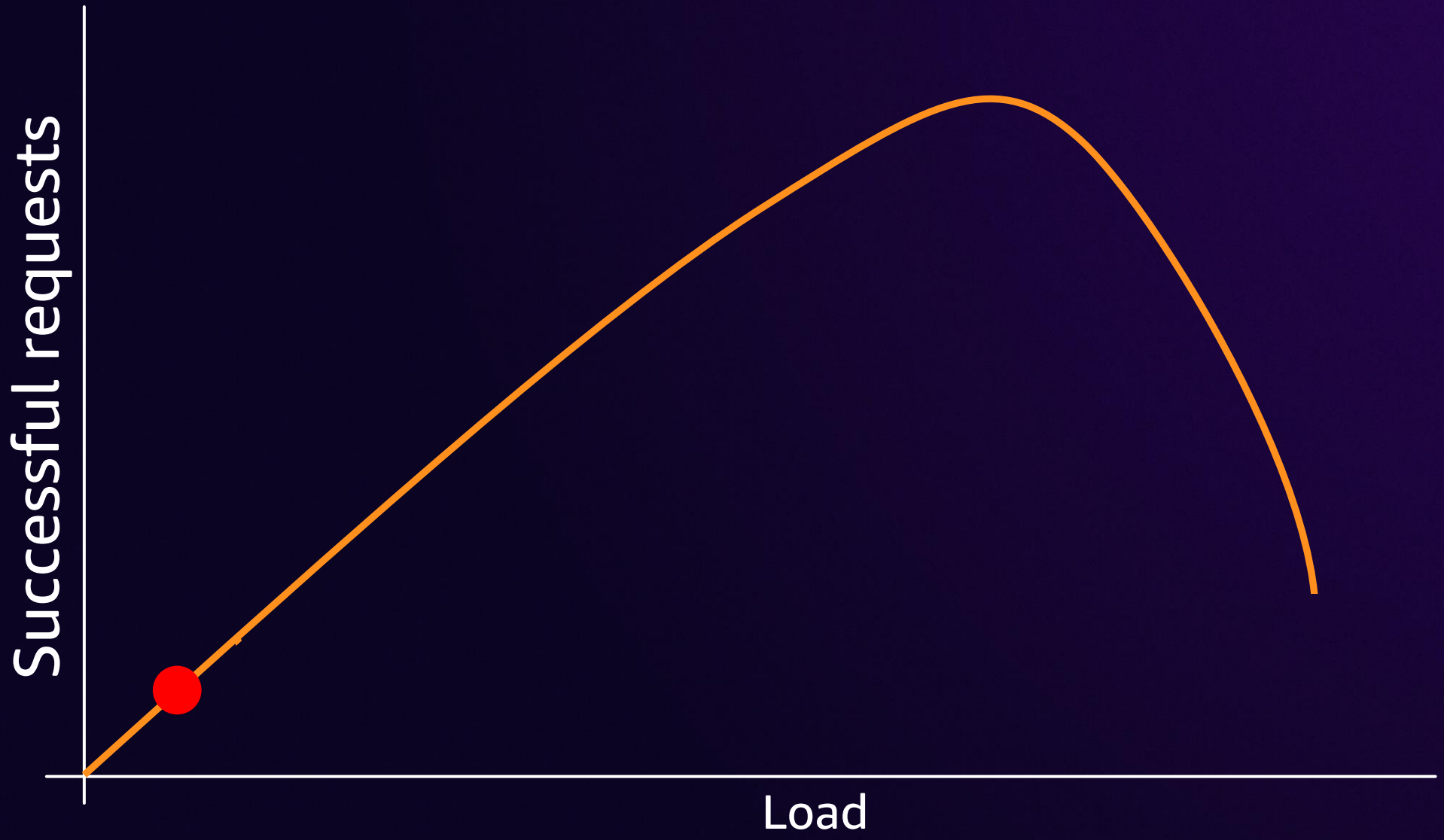
AWS Lambda is a serverless event-driven compute service, part of a category of cloud compute offerings sometimes called Function-as-a-service (FaaS). When we first released AWS Lambda, functions were limited to 250MB of code and dependencies, packaged as a simple compressed archive. In 2020, we released support for deploying container images as large as 10GiB as Lambda functions, allowing customers to bring much larger code bases and sets of dependencies the most important metrics that determine the customer experience in FaaS systems. When we launched AWS Lambda, we recognized that reducing data movement during these cold starts was critical. Customers deployed functions to Lambda in compressed archives (*.zip* files), which were unpacked as each function instance was provisioned. As Lambda evolved, and customers increasingly looked to deploy more complex applications, there was significant demand for larger deployments, and the ability to use container tooling (such as *Docker*) to create and manage these deployment images. Customers

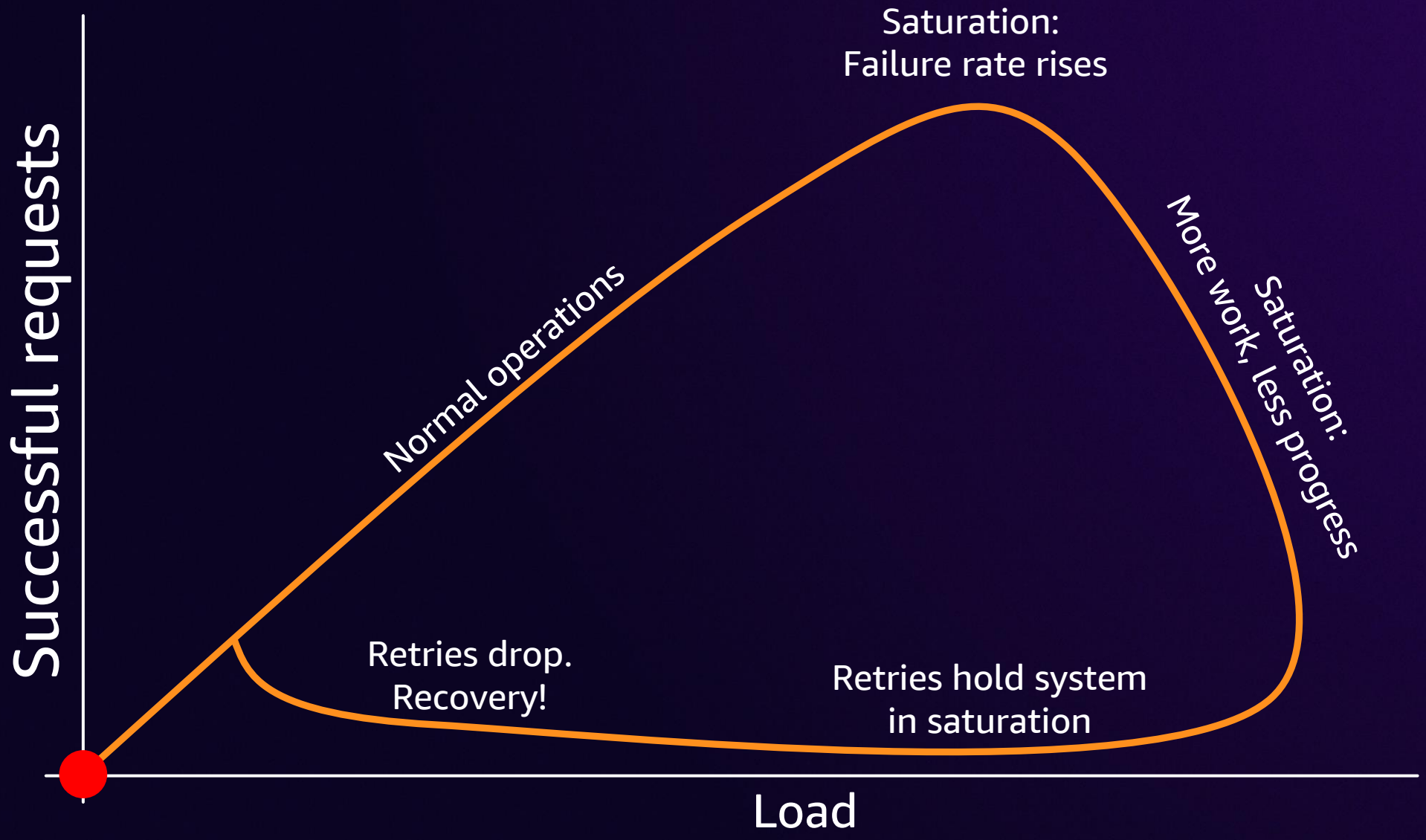
# Erasure coding in AWS Lambda

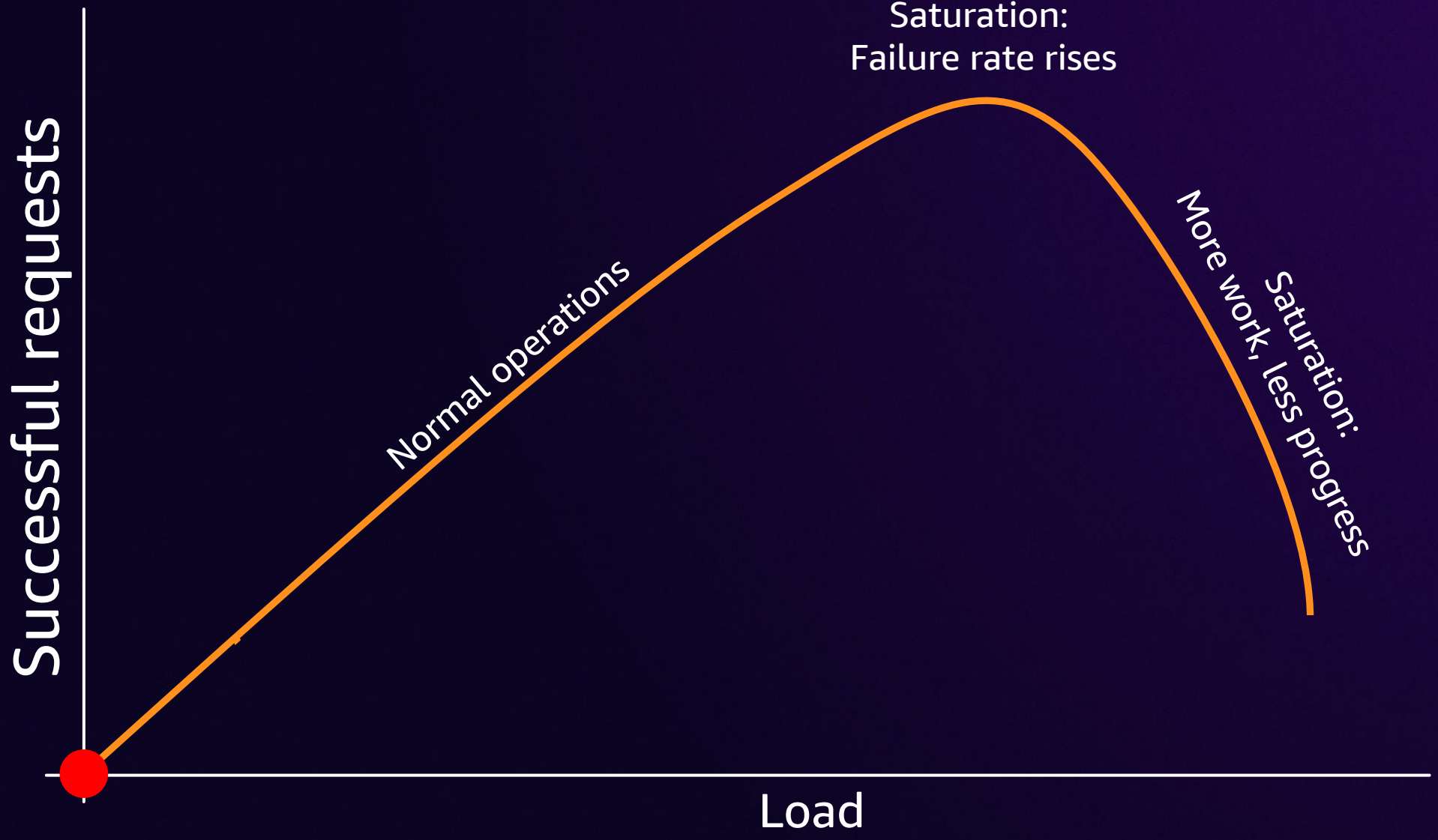
- Significant tail latency benefits
- Resilience benefits
  - Also handles a failed server, or an in-progress deployment
- Constant work
  - Fast and slow paths do exactly the same amount of work
- Lambda uses a simple 4-of-5 code (just XOR!)
- But much more sophisticated codes exist

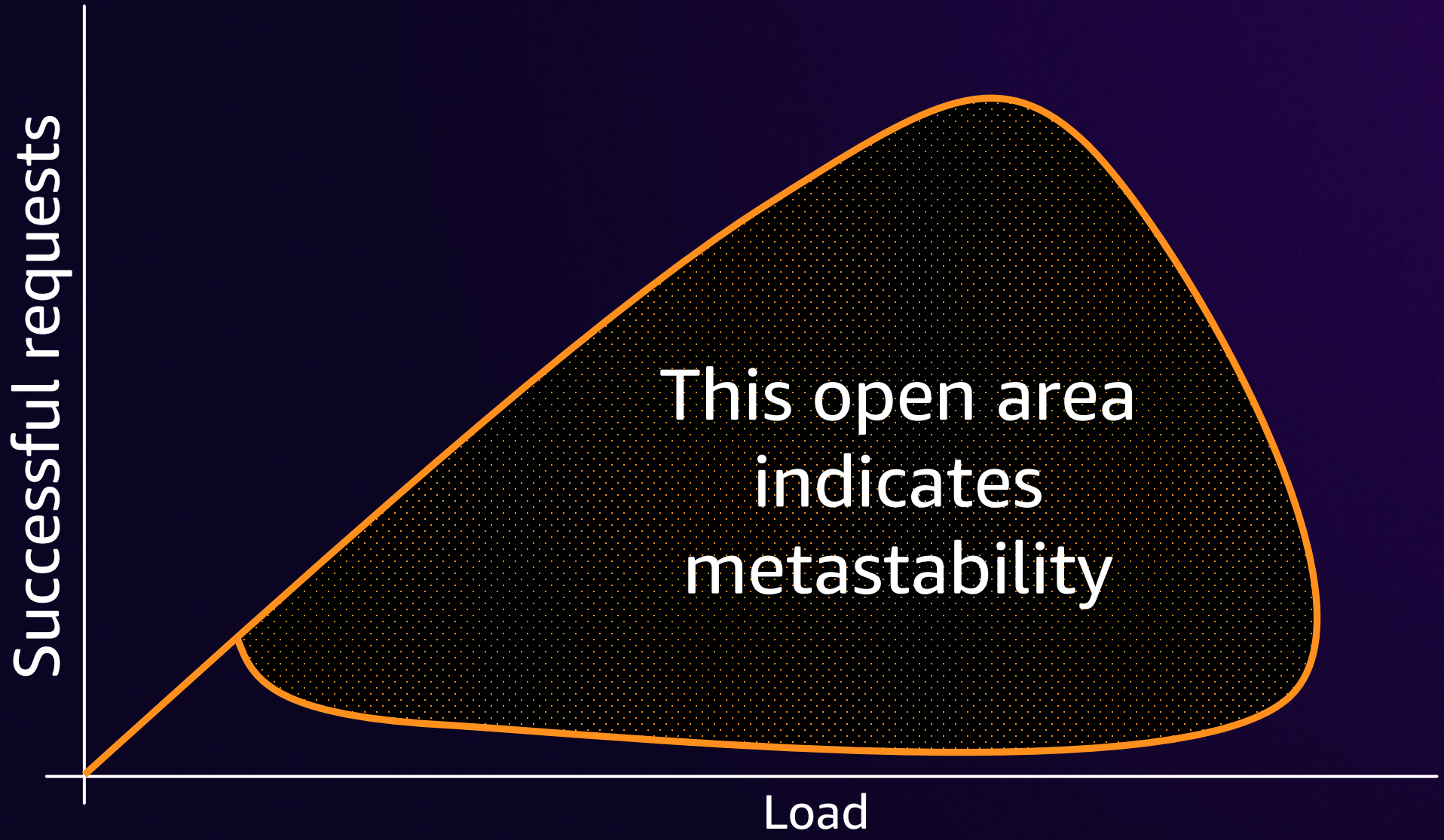
# What is metastability, anyway?











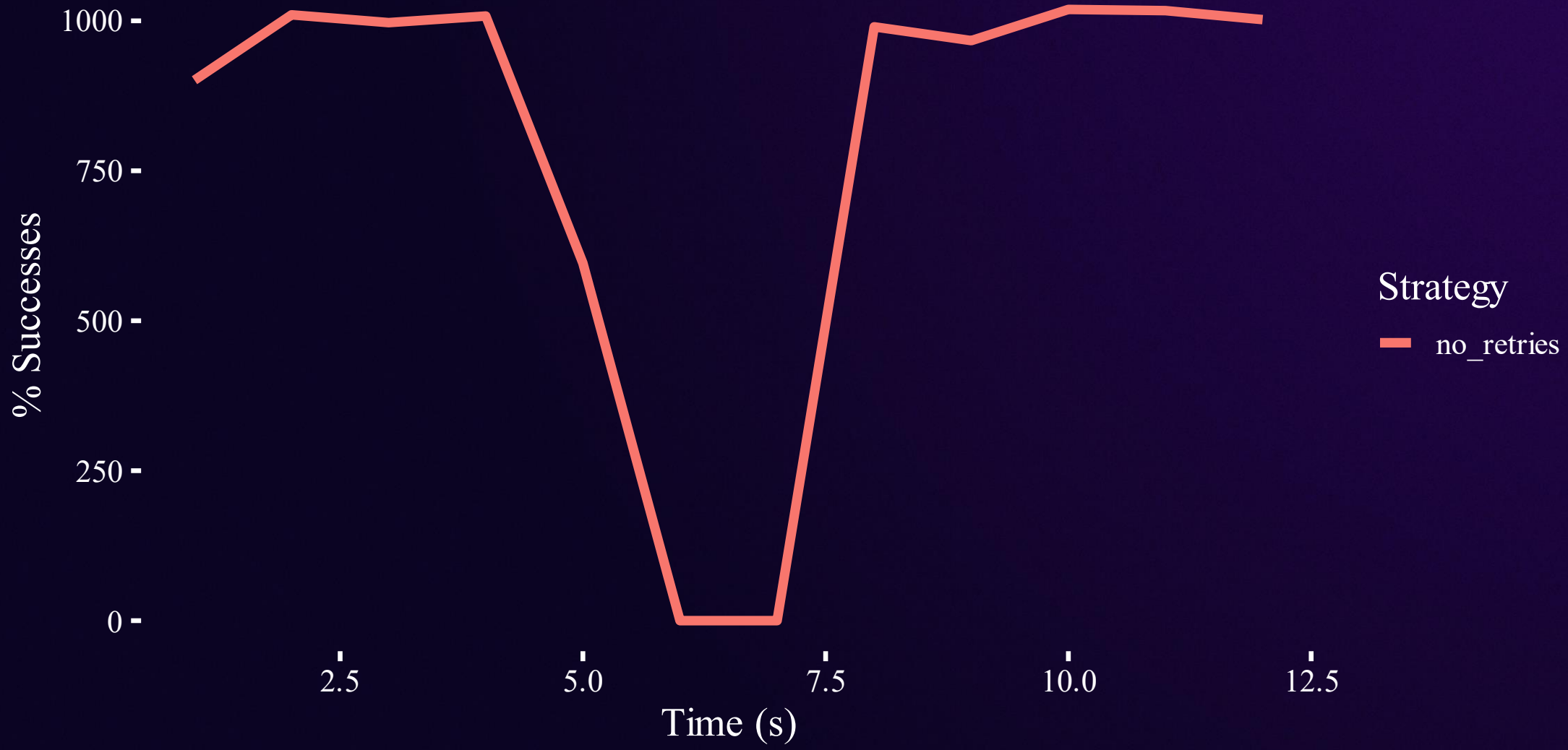
# Simulation: Understanding behavior

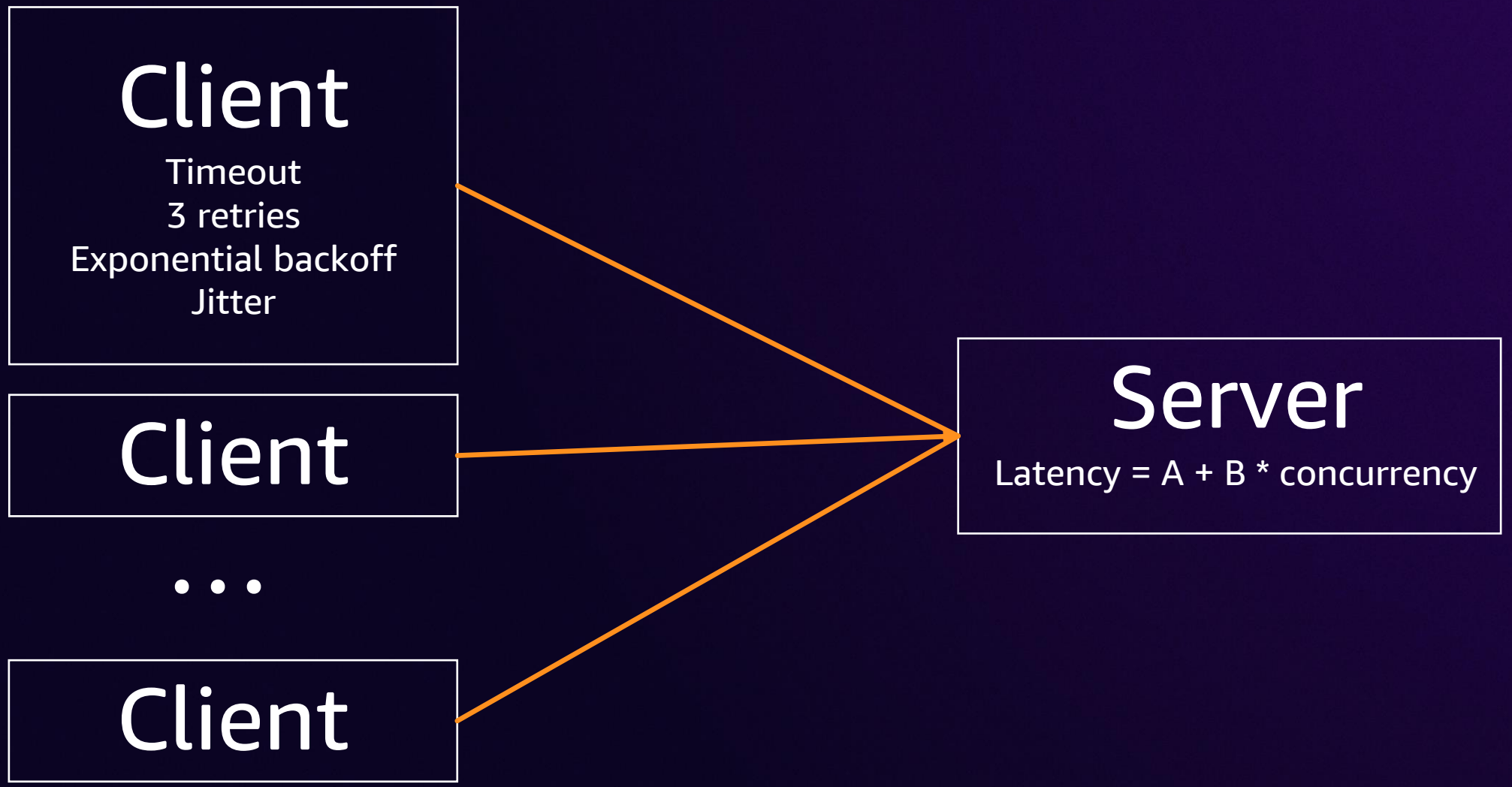
# Resampling: A statistical method

- Use real measured data
  - Real samples
  - eCDF from real samples
- Pick 5 samples from the data
- Write down the $4^{th}$ best
- Do that a bunch of times, and measure the result

# Resampling: A statistical method

- No need to:
  - Choose a distribution
  - Fit a distribution
  - Implement goodness-of-fit tests
- Add, multiply, best-of-N, and so on, without lots of complexity





# Simple simulations

- This one is <150 lines of Python
- Directly models the system
  - A client class, a server class, and so on
- Easy to review
  - Few statistical assumptions, little advanced math
- It’s a deeply mathematical method that doesn’t feel like math 😀

# Another example: Nudge

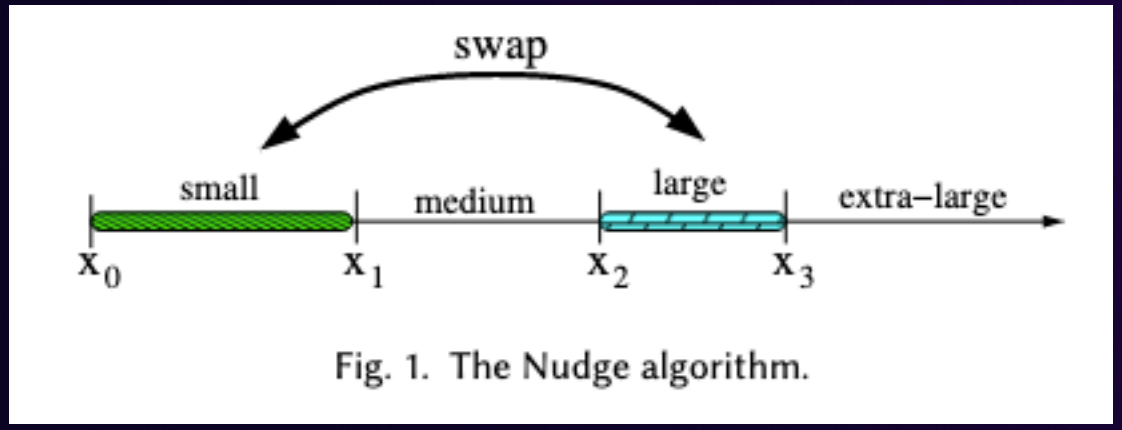
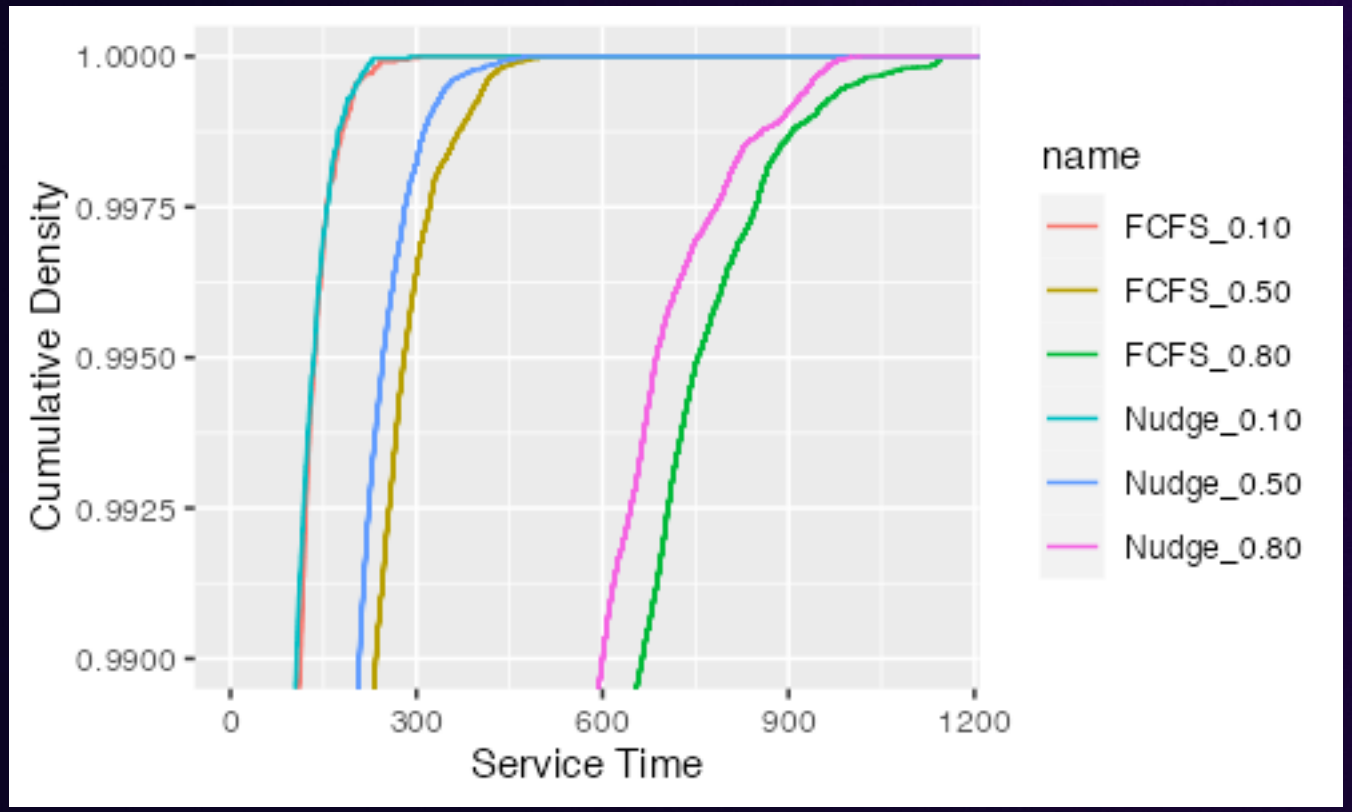



Fig. 1. The Nudge algorithm.



Grosof et. Al., *Nudge: Stochastically Improving upon FCFS,* 2021

# Would Nudge work in my systems?






https://github.com/mbrooker/simulator_example/tree/main/nudge

# Build the simplest simulations using Amazon Bedrock!

“Write a simulator that estimates how likely two arrays of N random numbers chosen from the range 0 to k are to have at least one number in common, in the R language.

Consider both the cases that the random numbers are chosen uniformly, and chosen from a Zipf distribution.

Plot the results for k=10000 and N between 1 and 100...”

**To Sonnet 3.5 on Amazon Bedrock**

# Takeaways

# Retries

- Jitter is good
- Retries can introduce **tipping point** failures
- Using a token bucket avoids this failure mode
- Backoff isn’t very effective in **open** systems

# Circuit breakers

- Can be a very powerful tool
- Be careful in sharded systems
- Be careful in service architectures

# The tail

- Do more work; get a flatter tail
- Prefer techniques that do constant work
- Or use a token bucket to limit additional work
- Erasure coding is great, if you can get it

# Simulations and statistics

- Simulations are a powerful mathematical tool
- Simulations look like code
- You should write more simulations

# Thank you!

Please complete the session survey in the mobile app

**Marc Brooker**
mbrooker@amazon.com
𝕏 @marcjbrooker