When Things Go Wrong

KEEPING HIGH-TRAFFIC SYSTEMS RUNNING

When Things Go Wrong

A comprehensive guide to maintaining system reliability
under high traffic and failure conditions

How do we keep the system stable while it's failing?

The Promise of Reliability

A Real Incident Story

PART 1 — How systems actually fail under load

It starts quietly. A database query that usually takes 50ms now takes 200ms.

Your monitoring shows everything is "green" — error rates are low, CPU looks fine.

But requests are starting to **queue up**. Connection pools are filling.

This is how most outages actually begin.

Fan-out + Latency Multiplication

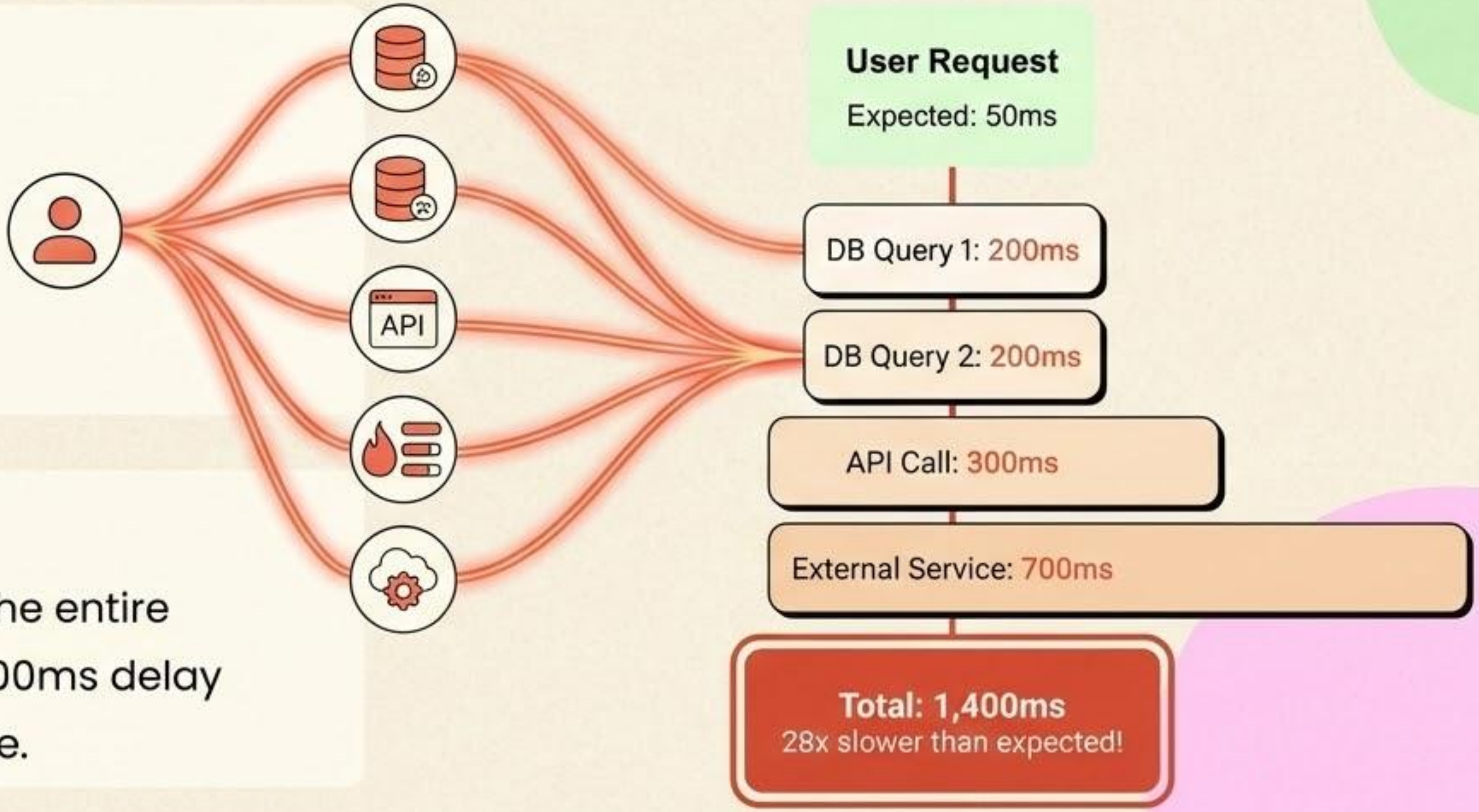
In modern systems, one user action can trigger multiple calls

Single user request triggers:

- 3 database queries
- 2 API calls to other services
- 1 cache lookup
- 1 external service call

The multiplication effect:

If one dependency gets slow, the entire request chain slows down. A 200ms delay becomes a 1.4s user experience.



The Retry Storm

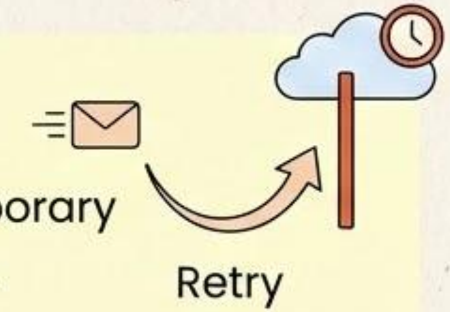
The system sees slowness.

Developers did what felt reasonable: **"add retries."**

But here's the problem...

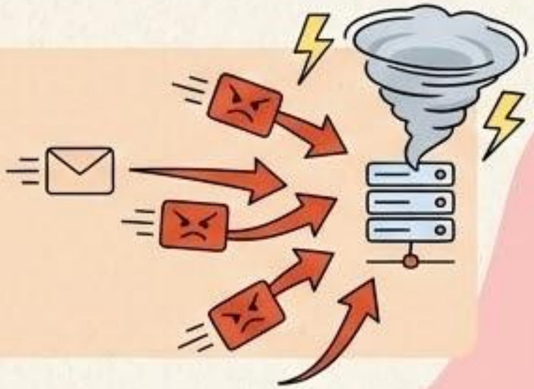
What We Think Happens:

Retries help recover from temporary failures and improve reliability.



What Actually Happens:

Retries multiply load at the **worst possible moment.**

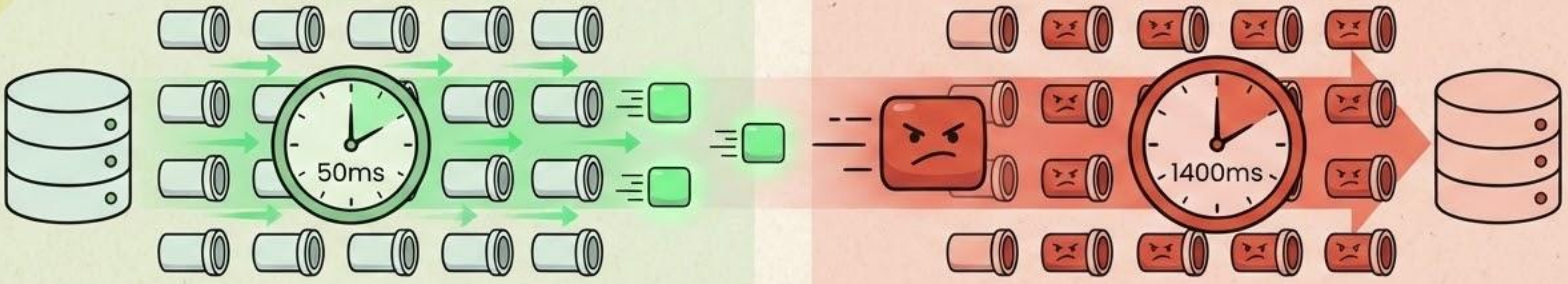


RETRIES DON'T REDUCE LOAD

This is called a **retry storm**

Connection Pool Exhaustion

Then you hit the next failure mode



Before (Normal):

20 connections × 50ms = 400 requests/second capacity

After (Slow):

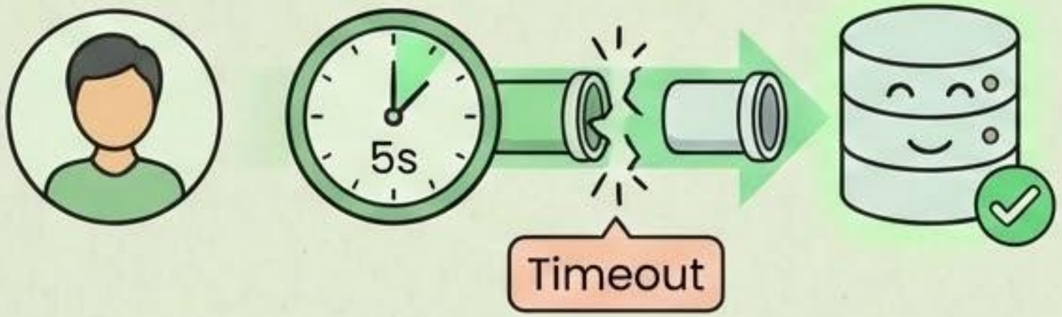
20 connections × 1400ms = 14 requests/second capacity

YOUR SERVICE IS NOW FROZEN

All connections are held by slow requests

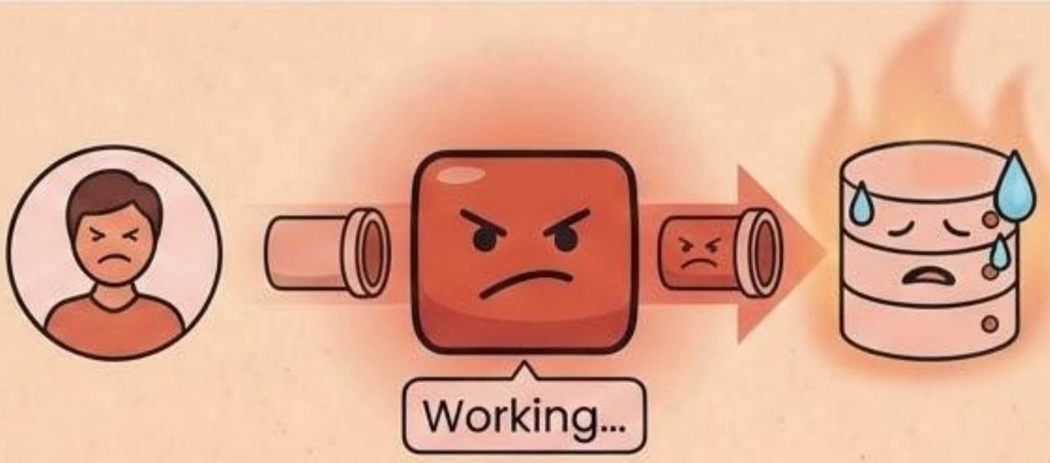
The Silent Killer: Timeouts

At this point, timeouts start firing



What We Expect:

Request times out after 5 seconds, connection is freed, system recovers.



What Actually Happens:

Client gives up, but server keeps working. Database connection stays busy.

YOUR RESOURCES ARE SUFFOCATING

The work continues, but the client is gone.

The Goal During an Incident

PART 2 — How to stop the bleeding (what actually works)

Your goal is NOT root cause analysis.

Your goal is **STABILIZATION**.

Stop the bleeding

Shed load, limit work

Degrade gracefully

Turn off non-essential features

Stop unnecessary work

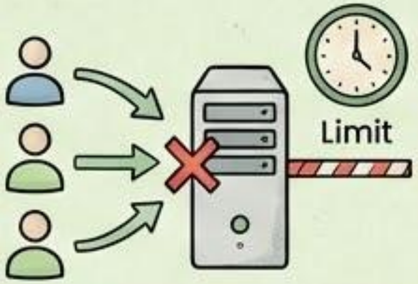
Cancel background jobs

Step 1: Put Hard Limits on Work

First thing: limit concurrency

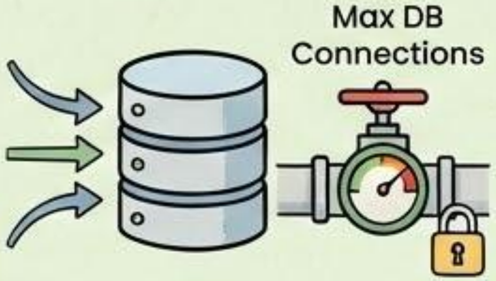
Concurrent Requests

Limit concurrent requests per endpoint to prevent overload



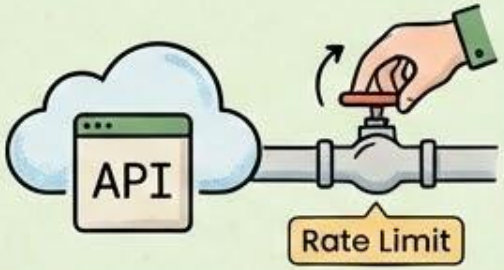
Database Queries

Limit concurrent DB queries per service



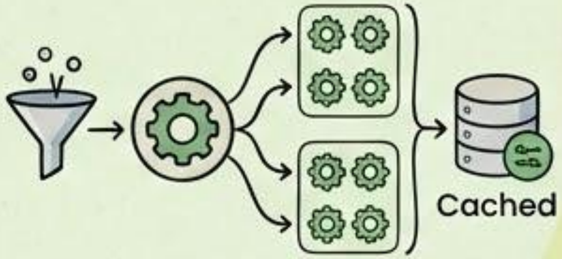
Third-party Calls

Limit concurrent calls to third-party providers



Fan-out Control

Limit fan-out by batching or caching



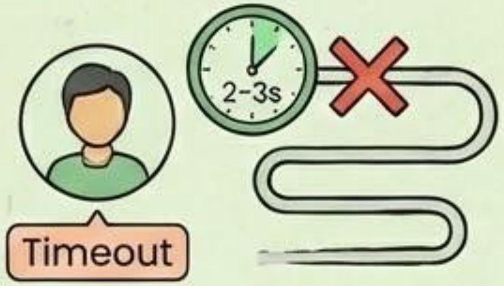
HARD LIMITS PREVENT CASCADING FAILURES

Step 2: Timeouts + Cancellation

Next: timeouts, but done properly

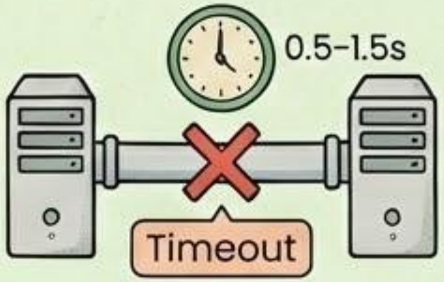
Client Timeout

Maybe 2-3s for a user request



Service-to-Service

Maybe 500ms-1.5s depending on endpoint



Database Timeout

Tight limits

Background Jobs

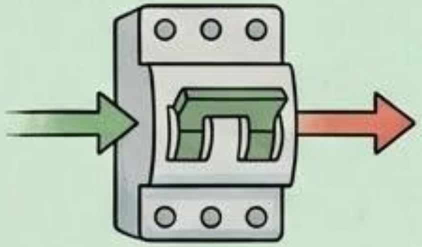
Different limits

CRITICAL: TIMEOUTS MUST INCLUDE CANCELLATION.

When the client gives up, the server must stop working too.

Step 3: Circuit Breakers

Now let's talk about circuit breakers

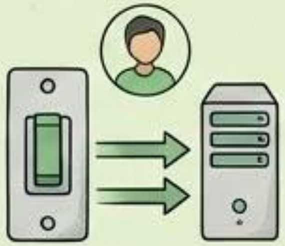


A circuit breaker stops calling a failing service.

When error rate exceeds threshold, it fails fast instead of waiting.

CLOSED

Normal operation. Requests pass through.

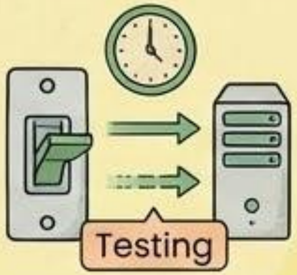


OPEN

Service is failing. Requests fail immediately.

HALF-OPEN

Testing recovery. Limited requests allowed.



CIRCUIT BREAKERS PREVENT RETRY STORMS

They give failing services time to recover

Step 4: Graceful Degradation

The part that separates mature systems from fragile ones

When dependencies fail, your system should still work — just with reduced functionality.

Recommendations Engine Down?

Show popular items instead

Search Service Slow?

Use cached results or simpler search

Payment Service Failing?

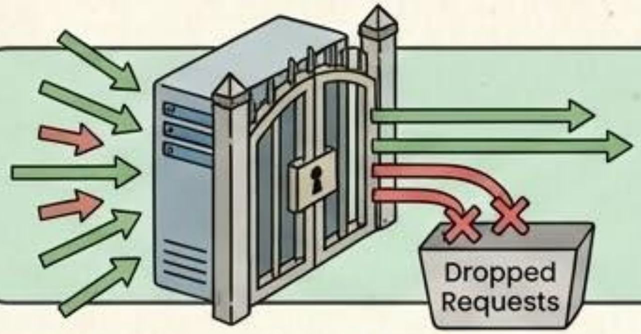
Queue orders for later processing

Analytics Down?

Skip tracking, keep core features working

Step 5: Load Shedding

Being intentional about failure

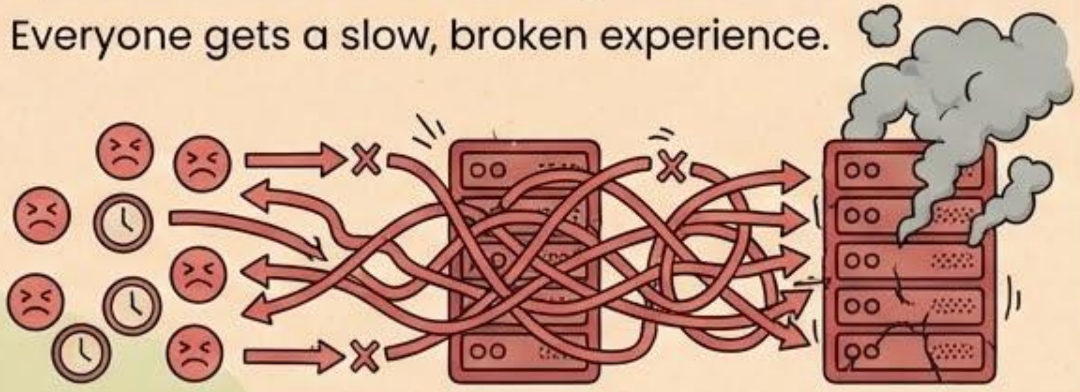


Load shedding means **deliberately dropping some requests** to keep the **system stable** for everyone else.

Without Load Shedding

System tries to handle all requests.

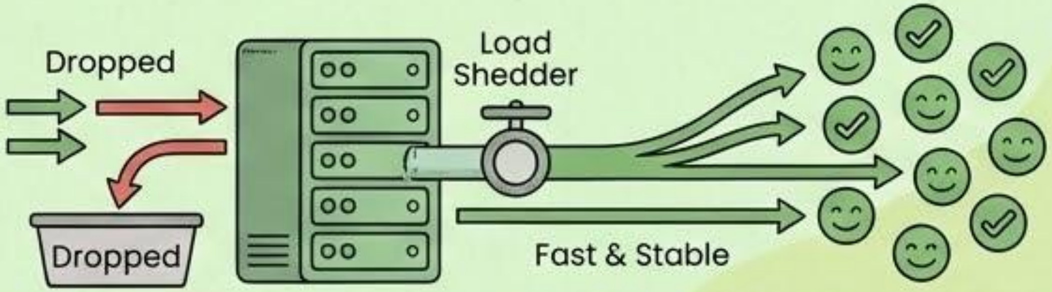
Everyone gets a slow, broken experience.



With Load Shedding

Drop 20% of requests.

80% of users get a fast, working experience.

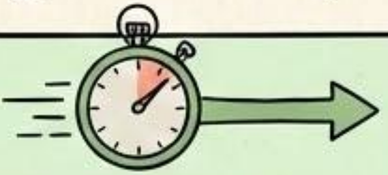


The 4 Things to Watch

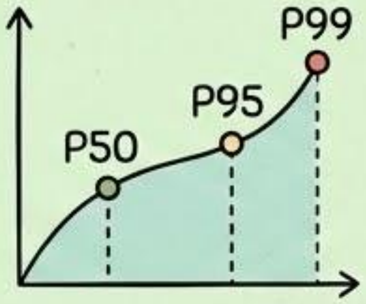
PART 3 — Observability that actually helps (not just dashboards)

In a high-traffic system, you track four things like your life depends on it:

LATENCY

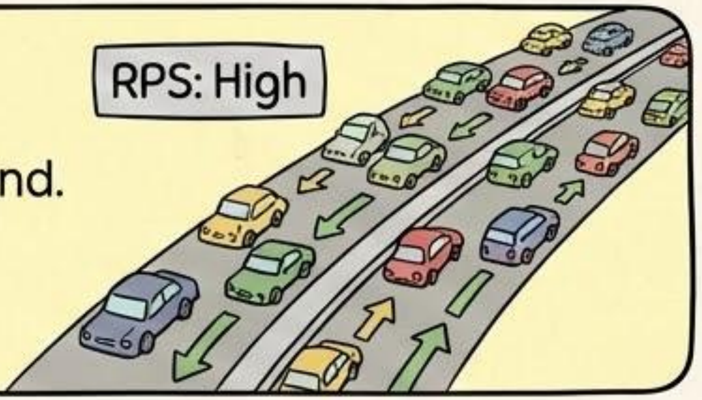


Percentiles, not averages.
P50, P95, P99 tell the real story.



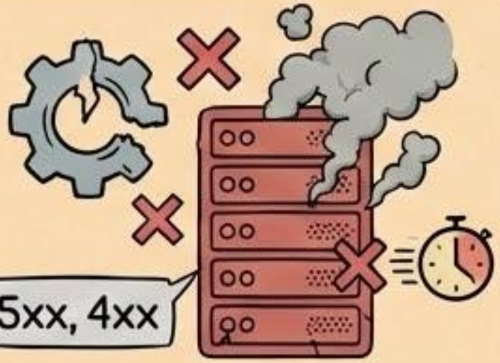
TRAFFIC

Requests per second.
Know your normal patterns.



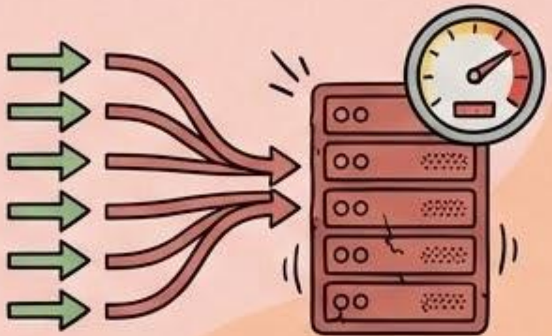
ERRORS

Error rates and types.
5xx vs 4xx vs timeouts.



SATURATION

Queues, pools, CPU, DB connections.
This is often the first sign of trouble.



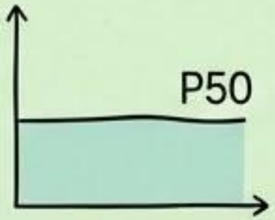
What a Quiet Incident Looks Like

Here’s how quiet incidents show up

The warning signs that most teams miss:

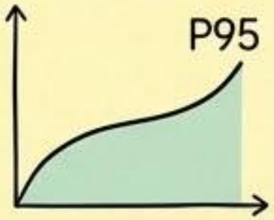
P50 is stable

Median response time looks normal



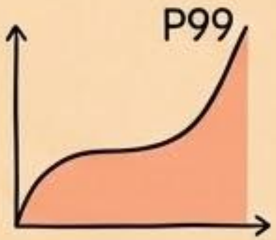
P95 creeps up a little

95th percentile starts to increase

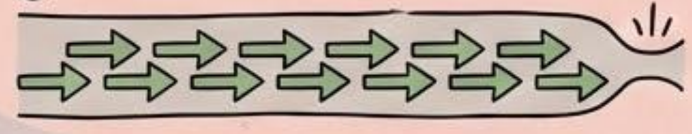


P99 spikes

Worst-case latency jumps dramatically



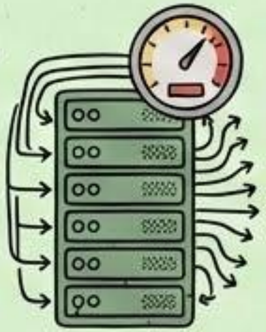
Queue depth slowly grows



Requests start backing up

DB connections approach max

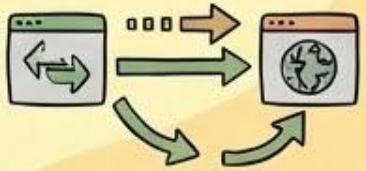
Connection pool getting exhausted



Retries increase

System trying to compensate

Attempts: 1, 2, 3, 4...

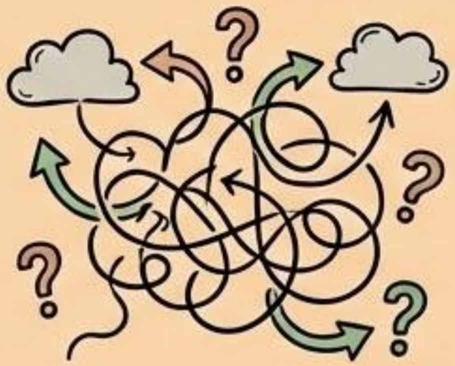


Traces and Correlation IDs

When things are failing, logs alone won't save you

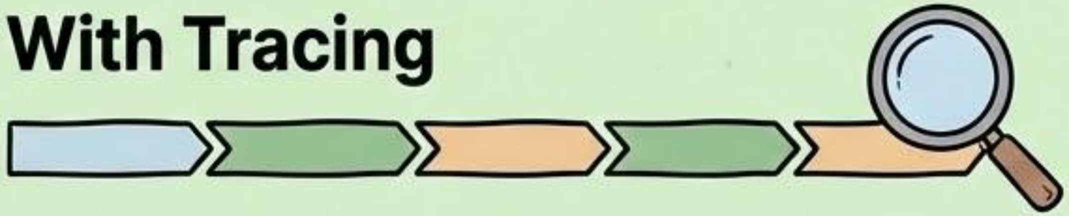
You need the ability to take one request and ask: "Where did the time go?"

Without Tracing



- Something is slow
- Check all the services
- Look at all the databases
- Hours of investigation

With Tracing



- Request ABC123 spent 2.3s in the payment service
- The payment service spent 2.1s waiting for database query XYZ
- Minutes of investigation

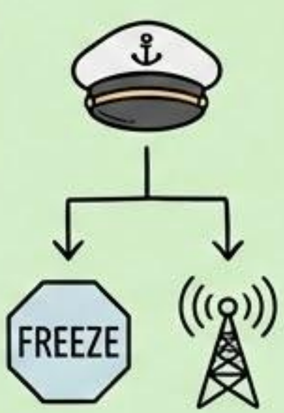
Incidents Fail Socially First

PART 4 — Incident response that doesn't make it worse

Most outages are prolonged by messy human responses, not technical problems.

What Goes Wrong

- Too many people "helping"
- Conflicting commands
- Panic deployments
- No clear decision maker



Calm Incident Response

- Name an incident commander
- Freeze all changes
- Stabilize first, investigate later
- Communicate clearly and often

The Incident Response Playbook

1. Stabilize → 2. Restore Service → 3. Investigate Root Cause

The Reliability Mindset

PART 5 — Designing systems that survive traffic

If you want reliability, you don't start during the outage.

You start before the outage.

Design Principles

- Idempotency
- Queues for async work
- Bulkheads
- Rate limiting
- Safe defaults

Operational Practices

- Chaos engineering
- Load testing
- Runbooks
- Game days
- Post-mortems

Cultural Elements

- Blameless culture
- Learning from failures
- Shared responsibility
- Continuous improvement
- Psychological safety

Final Takeaway

CLOSING — End it clean and memorable

Let me end with this.

High-traffic systems don't break because engineers are careless.

They break because complexity compounds faster than intuition.

To survive, you need to:

Put hard limits on work • Use smart timeouts • Degrade gracefully

Monitor saturation • Plan for failure