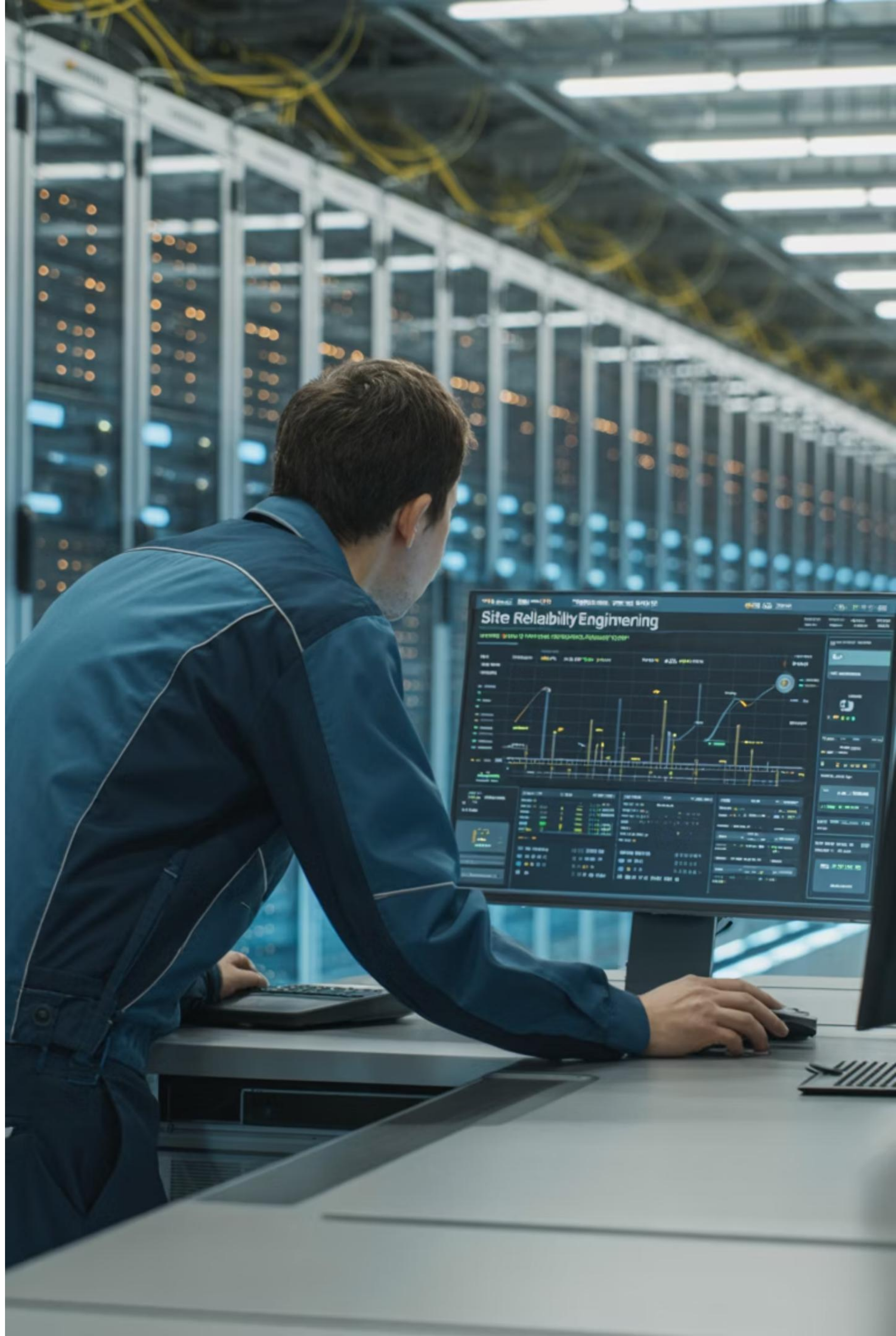
Introduction to SRE and Systems Thinking

A practical guide for engineers and engineering managers



Let's Start With a Story...

Imagine it's 3 AM. Your e-commerce site is down during a major sale. Customer complaints are flooding in. Your team is scrambling to find the issue while executives demand answers. Sound familiar?

This is the world *without* proper SRE practices. Let's explore how SRE can transform this scenario.

What is Site Reliability Engineering?

> "SRE is what happens when you ask a software engineer to design an operations team."

— Ben Treynor Sloss, Google VP and founder of SRE

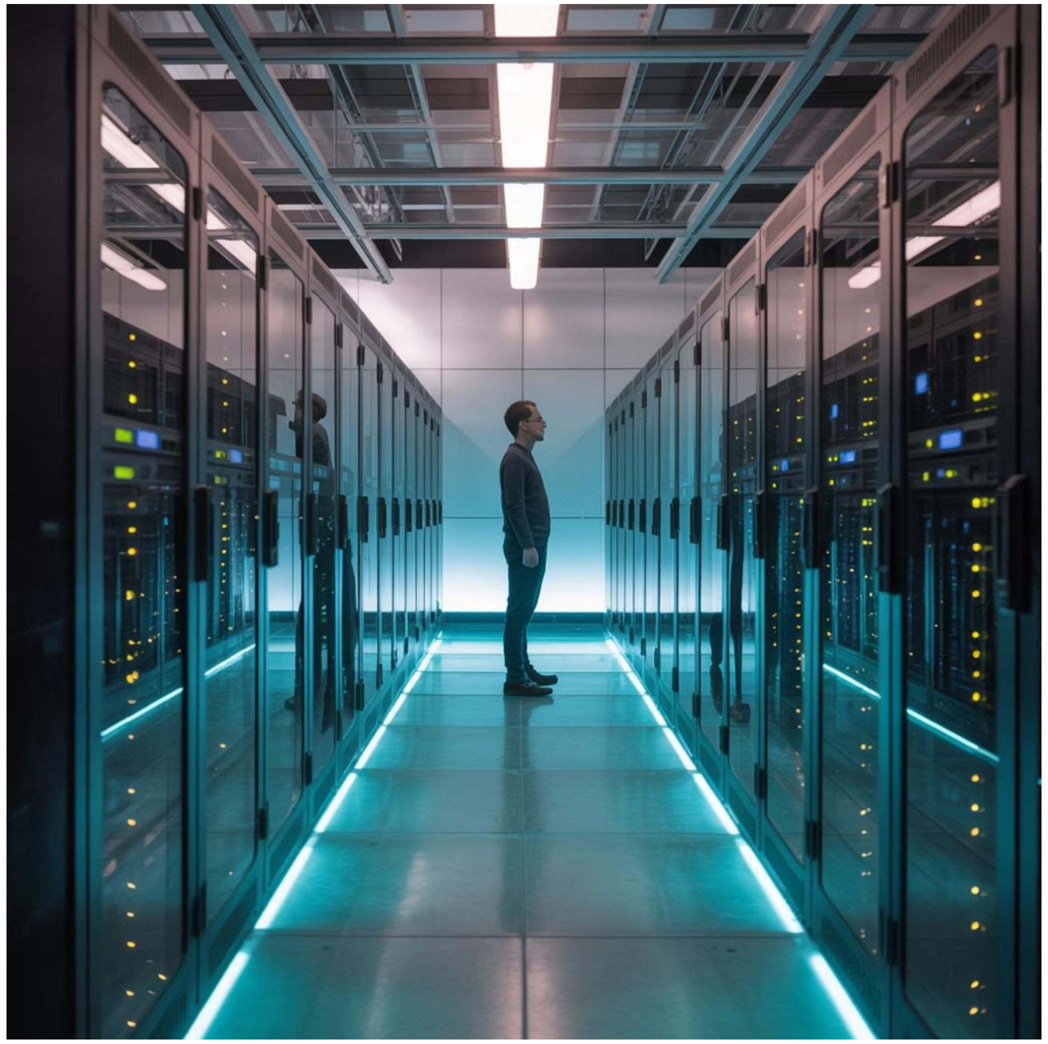
SRE applies software engineering principles to infrastructure and operations. It's about creating scalable and reliable software systems by treating operations as a software problem.

The Birth of SRE: Google's Model

In 2003, Ben Treynor Sloss was hired at Google to lead a team of engineers responsible for keeping Google's production systems running smoothly.

Before this, Google's approach to reliability was reactive rather than proactive. The creation of the SRE team changed that fundamentally.

Google's SRE approach has since influenced how many tech companies manage their operations and reliability concerns.



SRE in the Real World: Netflix's Chaos Monkey

Netflix embraced SRE principles with their famous "Chaos Monkey" - a tool that randomly terminates instances in production to test system resilience.

This approach seems counterintuitive: deliberately breaking your system? Yet it ensures Netflix engineers build services that can withstand failures, improving overall reliability.

This exemplifies SRE's proactive approach to preventing outages rather than just reacting to them.

SRE vs. Traditional IT Operations

Traditional IT Operations

- Manual processes and workflows
- Reactive approach to problems
- Separate operations and development teams
- Success measured by uptime (often aiming for "100%")

Site Reliability Engineering

- Automation and engineering solutions
- Proactive approach with preventative measures
- Engineers handle both development and operations
- Success measured by SLOs and error budgets

SRE vs. DevOps: Similarities and Differences

Both SRE and DevOps aim to bridge the gap between development and operations, but they approach this goal differently. DevOps is a broader cultural and professional movement, while SRE provides a specific implementation of engineering practices.

Core SRE Principles

Embrace Automation

Automate repetitive tasks to reduce toil and human error

Balance Reliability & Features

Use error budgets to make data-driven decisions about risk

Measure Everything

Define and track SLIs, SLOs, and SLAs

Reduce Toil

Eliminate manual, repetitive work that doesn't add long-term value

The Toil Trap: A Real-World Example

Meet Sarah, a systems engineer at an e-commerce company. Every week, she spends 15 hours manually checking server logs, restarting services, and updating configurations across hundreds of servers.

This is **toil** - manual, repetitive work that scales linearly with service growth. Over a year, Sarah spends nearly 800 hours on these tasks instead of improving the system.

SRE solution: Sarah writes scripts to automate these tasks, reducing her weekly toil to just 2 hours and freeing time for engineering work that prevents future problems.

Reliability Metrics: SLIs, SLOs, and SLAs

1

Service Level Indicators (SLIs)

Quantitative measures of service level

Example: API response time (how fast your app responds) (Latency)

2

Service Level Objectives (SLOs)

Target values for service level measured by SLIs

Example: 95% of API requests complete within 200ms

3

Service Level Agreements (SLAs)

Explicit or implicit contracts with users including consequences of missing SLOs

Example: If API response time exceeds 200ms for more than 10% of requests, customers get service credits

SLIs in Practice: The User Perspective

> "The most meaningful and useful SLIs directly measure user experience."

Latency

How long does it take to respond to a request?

Error Rate

What percentage of requests fail?

Availability

Is the service responding to requests?

Throughput

How many requests can be handled?

Good SLIs measure what users actually experience, not just what's easy to measure internally.

Setting Realistic SLOs: The 9s of Availability

99.9%	99.99%	99.999%
Three Nines	Four Nines	Five Nines
8.76 hours of downtime per year	52.56 minutes of downtime per year	5.26 minutes of downtime per year

Higher availability comes with exponentially increasing costs. Most services don't need five nines! Choose SLOs based on user needs and business requirements, not arbitrary perfection.

SLO Example: E-commerce Checkout Flow

Business Context

An online retailer notices that when their checkout process takes longer than 2 seconds, customers abandon their carts. For every additional 100ms of delay, they lose 1% of potential sales.

SLO Definition

SLI: Checkout completion time (from "Place Order" click to confirmation page)

SLO: 95% of checkout processes complete within 2 seconds over a rolling 30-day period

Why this matters: Keeps cart abandonment low and protects revenue

Error Budgets: Permission to Fail

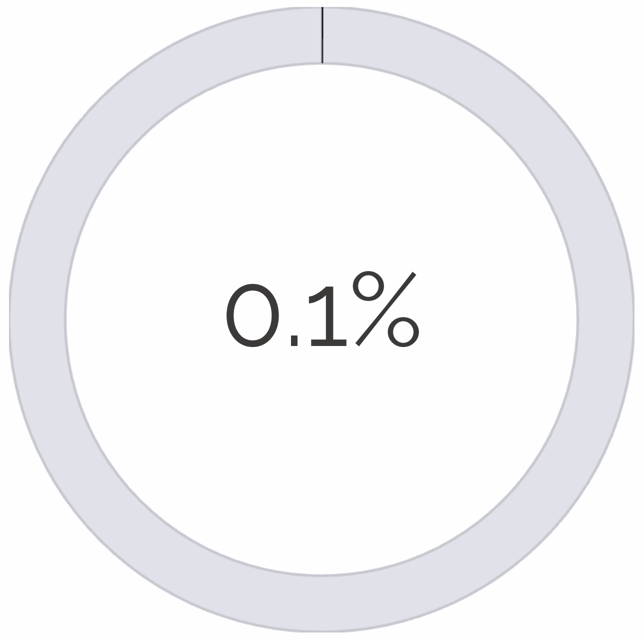
If your SLO is 99.9% availability, you have a 0.1% error budget - the amount of unreliability you're willing to accept to balance innovation with stability. Lets say you promised you client an SLA of 99.5%

Here error budget : 100% - SLO : 0.1% (1,000 failed requests out of 1,000,000)

Buffer for engg teams : | SLA -SLO | : 99.5 - 99.9 : 0.4%

Error budgets align incentives between development teams (who want to ship features) and operations teams (who want stability).

When the error budget is healthy, teams can take more risks. When it's depleted, they focus on reliability improvements.



Monthly Error Budget

About 43 minutes of allowed downtime per month at 99.9% availability

Error Budgets in Action: Google's Approach

Google uses error budgets to balance reliability and innovation:

- A product with an SLO of 99.99% uptime has a downtime budget of 52.56 minutes per year
- If the service is meeting its SLO, developers can deploy new features
- If too many outages occur and the budget is exhausted, feature launches pause while the team focuses on reliability improvements

This creates a self-correcting system that naturally balances innovation and stability.

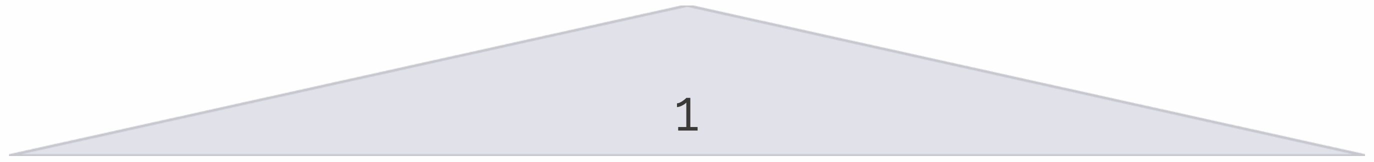
Systems Thinking: Seeing the Bigger Picture

"A system is more than the sum of its parts; it's the product of their interactions."

Systems thinking means understanding how different components interact with each other, rather than just examining each part in isolation.

In complex systems, problems often emerge from interactions between components, not from individual components themselves.

The Reliability-First Mindset



A reliability-first mindset shifts from "How do we add this feature?" to "How do we add this feature while maintaining reliability?"

This perspective values both innovation and stability, recognizing that unreliable systems ultimately deliver less value regardless of their features.

The Illusion of Complete Control

Traditional operations often assumes perfect control over systems. SRE acknowledges the reality: **complex systems will fail in unexpected ways.**

Instead of trying to prevent all failures (impossible), SRE focuses on building resilient systems that can recover quickly when failures inevitably occur.

This shift in perspective – from prevention to resilience – is fundamental to effective SRE practice.

Case Study: The Domino Effect

What Happened

In 2017, Amazon S3 experienced an outage when a team member made a small typo in a command. This minor error cascaded into a major outage affecting numerous sites and services across the internet.

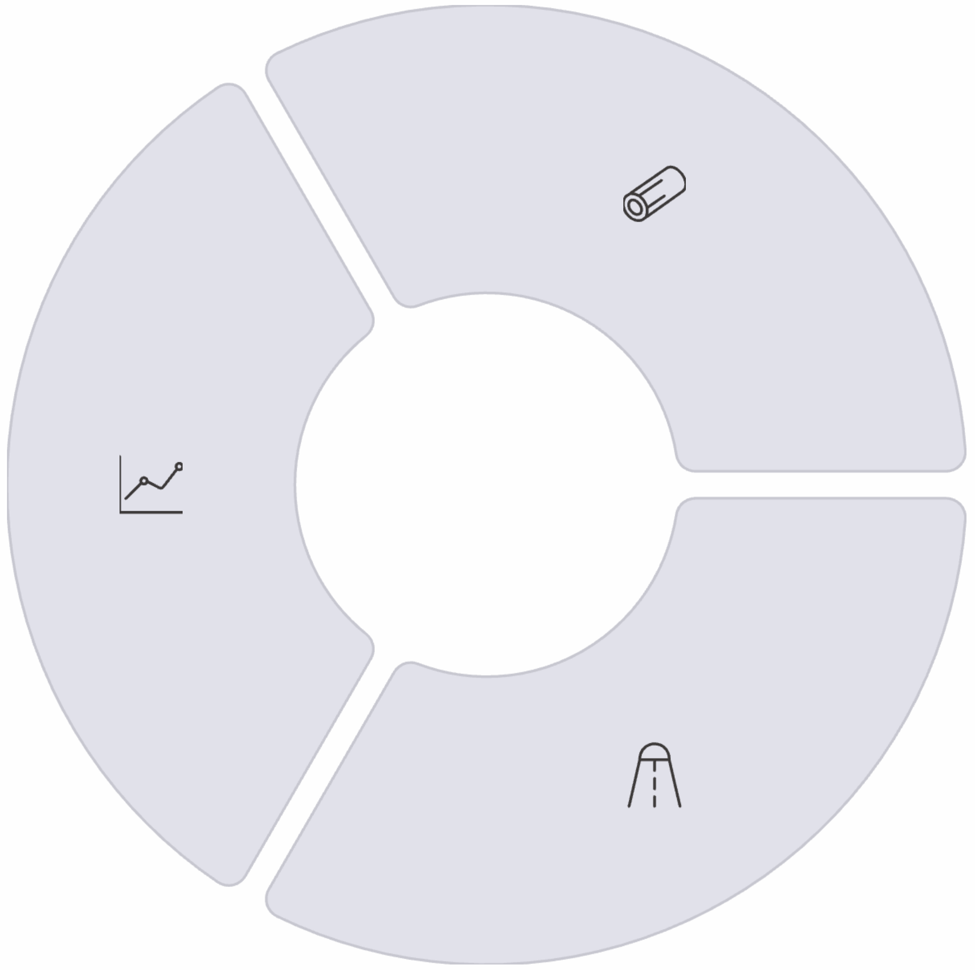
Systems Thinking Perspective

The issue wasn't just the typo – it was how the system responded to that typo. A systems thinking approach examines not just what failed, but how the components interacted to amplify the failure.

Amazon subsequently improved their system to prevent this type of cascading failure.

Observability: Beyond Simple Monitoring

The Three Pillars of Observability



Metrics

Numerical data points measured over time (CPU usage, request count)

Logs

Timestamped records of discrete events

Traces

Records of request paths through distributed systems

Observability is about gaining insights into what's happening inside your systems without having to predict every possible failure mode in advance.

Monitoring vs. Observability

Monitoring

Tells you **when** something is wrong

- Based on predefined thresholds
- Focused on known failure modes
- Often uses dashboards and alerts

Observability

Helps you understand **why** something is wrong

- Enables exploration of unknown issues
- Provides context and correlation
- Allows asking new questions of the system

Good SRE practice requires both monitoring and observability to maintain reliable systems.

The Golden Signals of Monitoring

Latency

Time taken to serve a request

- Distinguish between successful and failed requests

Traffic

Demand placed on your system

- Often measured in requests per second

Errors

Rate of failed requests

- Explicit failures and implicit failures (wrong answer, too slow)

Saturation

How "full" your system is

- Focus on constrained resources (memory, I/O, etc.)

RED Method: A Practical Monitoring Approach

The RED Method, popularized by Tom Wilkie, offers a simple framework for monitoring microservices:

- **Rate:** Requests per second
- **Errors:** Number of failed requests per second
- **Duration:** Distribution of request latencies

These three metrics give you a comprehensive view of your service's behavior from the user's perspective, which aligns perfectly with SRE principles.

The USE Method: Resource Monitoring

Utilization

Percentage of time the resource is busy

Example: CPU utilization at 85%

Saturation

Degree to which the resource has extra work

Example: Queue depth, wait time

Errors

Count of error events

Example: Network interface errors

Developed by Brendan Gregg at Netflix, the USE Method complements the RED Method by focusing on resource health rather than request flow.

Practical Example: Monitoring a Web Service

RED Metrics (Service Health)

- Request rate: 1,000 requests/second
- Error rate: 0.1% (1 error/second)
- Duration: P95 latency of 250ms

USE Metrics (Resource Health)

- CPU utilization: 45%
- Memory usage: 70% with no swap
- Network saturation: 30% of bandwidth

Together, these metrics provide a comprehensive view of both the service behavior and the underlying resources.

Black Box vs. White Box Monitoring

Black Box Monitoring:

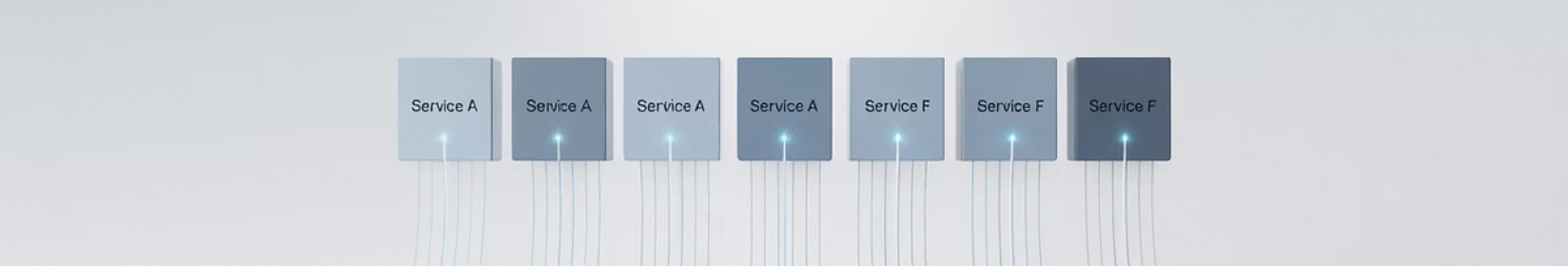
- Tests the system from the outside, like a user would
- Focuses on symptoms and user experience
- Example: HTTP health checks that test if your website responds with 200 OK

White Box Monitoring:

- Looks inside the system at internal metrics
- Focuses on causes and system internals
- Example: CPU usage, memory consumption, database query times

Example scenario: Your website is slow- Black box monitoring: "Website response time is 5 seconds (bad user experience)"- White box monitoring: "Database CPU is at 95%, query queue has 500 pending requests (internal cause)"

Effective monitoring strategies combine both approaches: black box monitoring to understand the user experience, and white box monitoring to diagnose internal issues.



Distributed Tracing: Following the Request Path

In microservice architectures, a single user request might touch dozens of services. Distributed tracing allows you to follow that request through the entire system.

Each service adds span information (timing, metadata) to a trace ID that follows the request, creating a complete picture of the request's journey.

Tools like Jaeger, Zipkin, and OpenTelemetry make this possible by standardizing how traces are collected and visualized.

Real-World Example: Debugging with Traces

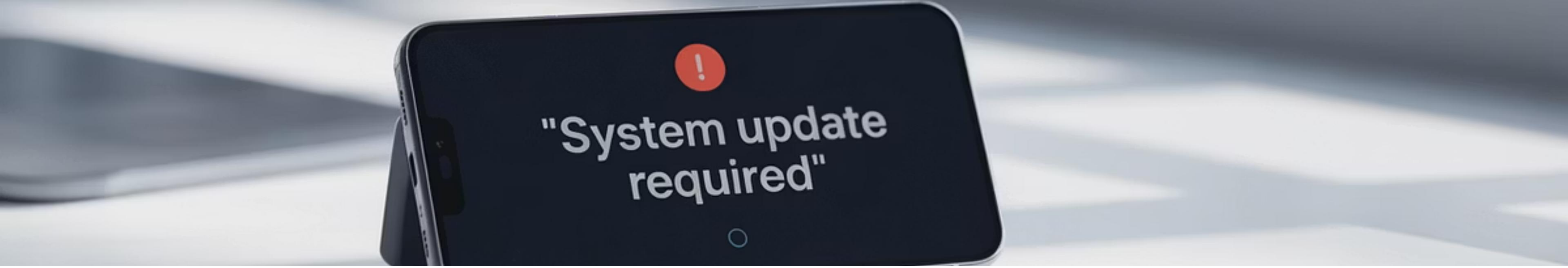
The Problem

Users report that the checkout process is occasionally slow. Metrics show intermittent latency spikes, but don't reveal the cause.

The Solution

Distributed tracing reveals that during these slowdowns, the payment validation service is taking much longer than normal, but only when it needs to communicate with a specific third-party API.

Further investigation shows that the third-party service is occasionally rate-limiting requests. Adding retries with exponential backoff solves the issue.



Building Effective Alerts

> "Every alert should be actionable, and someone should be taking action."

Good alerts have three key characteristics:

- **Actionable:** Tells you what needs to be done
- **Relevant:** Matters to the service's health
- **Timely:** Provides enough time to respond before users are impacted

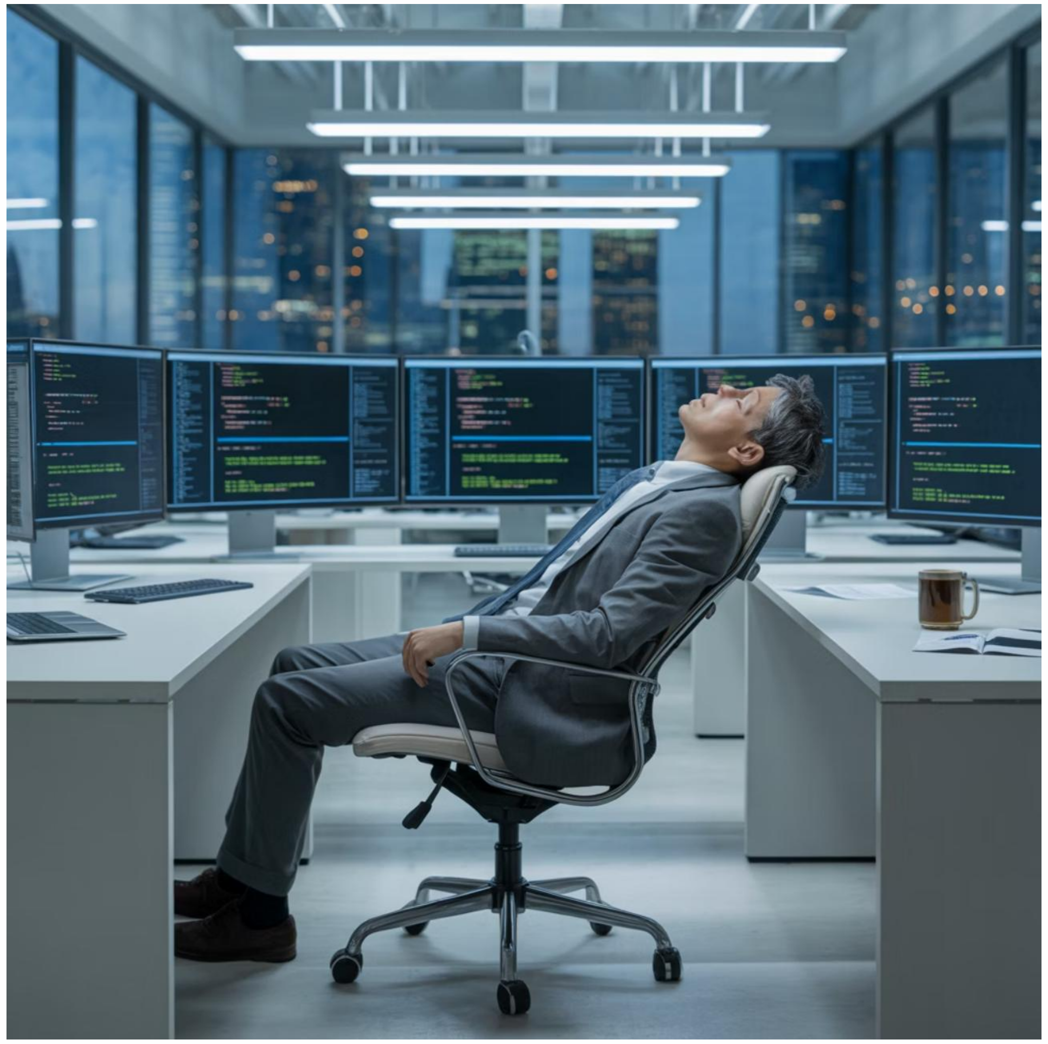
Alert Fatigue: The Boy Who Cried Wolf

When teams receive too many alerts, especially false positives, they start to ignore them – even the important ones.

This "alert fatigue" is dangerous: it leads to missed incidents and slower response times.

Studies show that after 10-12 alerts in a shift, response effectiveness drops dramatically.

SRE addresses this by creating fewer, more meaningful alerts tied directly to user impact.



From Bad to Good: Alert Evolution

Bad Alert

"CPU utilization > 80% on web-server-03"

- Not tied to user impact
- Not actionable (what should I do?)
- Based on a threshold that may not matter

Better Alert

"High latency detected on checkout service, 95th percentile > 2s (SLO: 500ms)"

- Tied to user experience
- Shows violation of a specific SLO
- Implies clear action: investigate the checkout service

Symptom-Based vs. Cause-Based Alerting

Cause-Based Alerts

"Database CPU usage > 90%"

- Focuses on potential causes
- Often results in alert noise
- Requires predicting failure modes

Shifting to

Symptom-Based Alerts

"Search latency exceeding SLO"

- Focuses on user-visible symptoms
- Reduces alert volume
- Catches unknown failure modes

SRE prefers symptom-based alerting because it aligns with what actually matters: the user experience.

Paging vs. Ticketing: Different Response Levels

Page (Immediate Response)

- User-impacting, needs urgent attention
- Requires human intervention now
- Examples: Service down, SLO breach

Ticket (Scheduled Response)

- Not immediately user-impacting
- Can be handled during business hours
- Examples: Capacity planning, non-critical bugs

⚠ Paging alerts should be rare and important. If everything is urgent, nothing is.

Google's SRE teams aim for no more than 1-2 paging incidents per engineer per on-call shift.

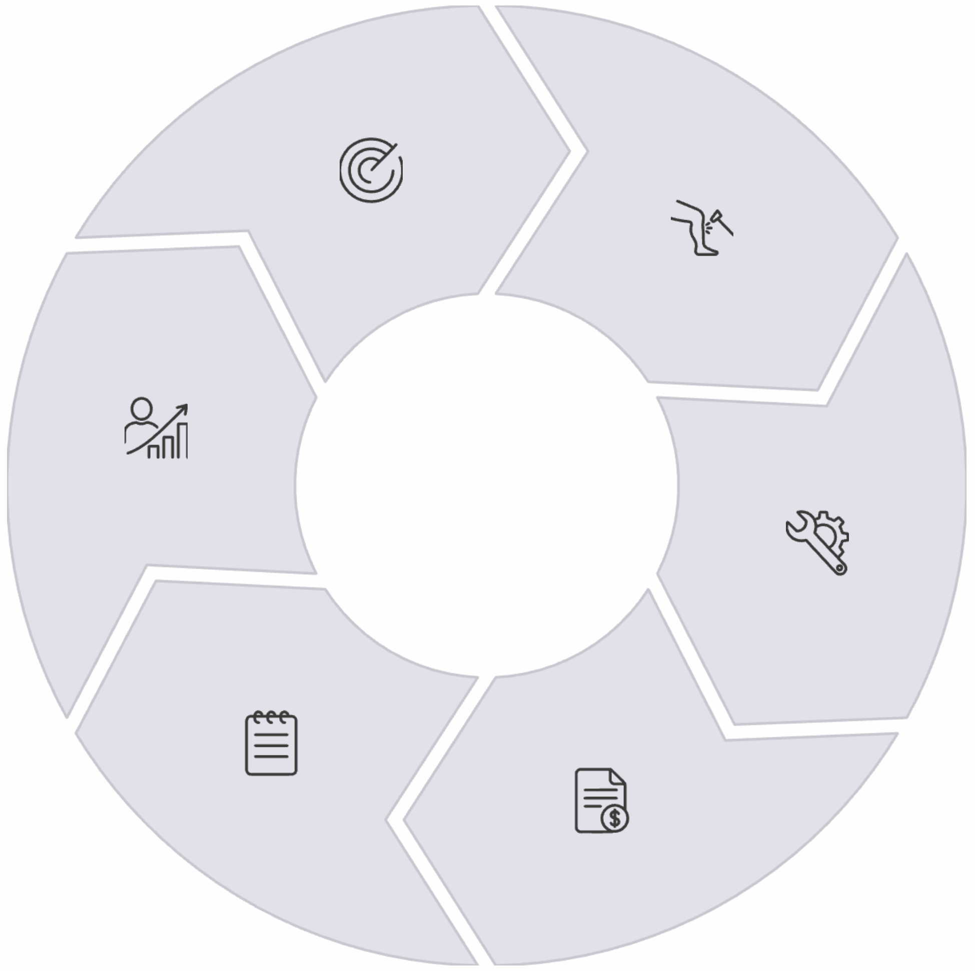
Incident Response: When Things Go Wrong

Even with the best prevention, incidents will occur. Effective incident response minimizes the impact and helps prevent recurrence.

> "Hope is not a strategy. Luck is not a factor. Fear is not an option."

Preparation and process are key to successful incident management. Let's explore how SRE approaches this challenge.

The Incident Response Lifecycle



Detection

Alerting identifies the incident

Response

Team assembles and begins mitigation

Improvement

Systems are hardened against recurrence

Mitigation

Service is restored to users

Documentation

Incident is documented in postmortem

Analysis

Root causes are investigated

Incident Roles and Responsibilities

Incident Commander (IC)

The decision maker who coordinates the response

- Assigns tasks and tracks progress
- Ensures everyone has clear responsibilities
- Makes critical decisions when needed

Communications Lead

Manages internal and external communications

- Updates stakeholders on progress
- Shields technical responders from interruptions
- Ensures consistent messaging

Technical Lead

Directs technical investigation and mitigation

- Guides troubleshooting efforts
- Evaluates proposed solutions
- Makes technical recommendations to the IC

Clear roles prevent confusion and ensure efficient resolution during incidents.

Real-World Incident: The GitHub DDoS Attack

The Incident

In 2018, GitHub experienced the largest DDoS attack ever recorded at that time, with 1.35 terabits per second of traffic.

The Response

GitHub's SRE team:

1. Quickly identified the attack pattern
2. Activated their DDoS mitigation service (Akamai)
3. Rerouted traffic through scrubbing centers
4. Restored service in just 8 minutes

Their preparation and practiced response plan minimized the impact despite the unprecedented scale.

Practical Incident Response: The First 5 Minutes

Acknowledge the Alert

Let the team know you're investigating

Assess Impact

How many users are affected? Is this a total outage or partial degradation?

Check Recent Changes

Was there a recent deployment or configuration change?

Look for Related Symptoms

Check dashboards for other anomalies

Communicate Initial Findings

Share what you know with the team

Severity Levels: Not All Incidents Are Equal

SEV-1: Critical

Complete service outage affecting all users

Example: Main website returns 500 errors for everyone

SEV-2: Major

Significant functionality impaired or subset of users affected

Example: Checkout process failing for 20% of customers

SEV-3: Minor

Limited impact, non-critical functionality affected

Example: Product images loading slowly, but site is usable

Severity levels help teams prioritize response efforts and determine appropriate escalation paths.

Triage Decision Tree

Is the service completely down for all users?

YES: SEV-1 - Page immediately, assemble incident team

Example: API returning 500 errors for 100% of requests = SEV-1

NO: Are users experiencing significant impact?

YES: SEV-2 - Page during business hours, investigate promptly

Example: Checkout slow for 20% of users = SEV-2

NO: SEV-3 - Create ticket, handle during normal work hours

Example: Product images loading slowly = SEV-3

This simplified decision tree helps on-call engineers quickly determine the appropriate response level for an incident.

Incident Management Tools

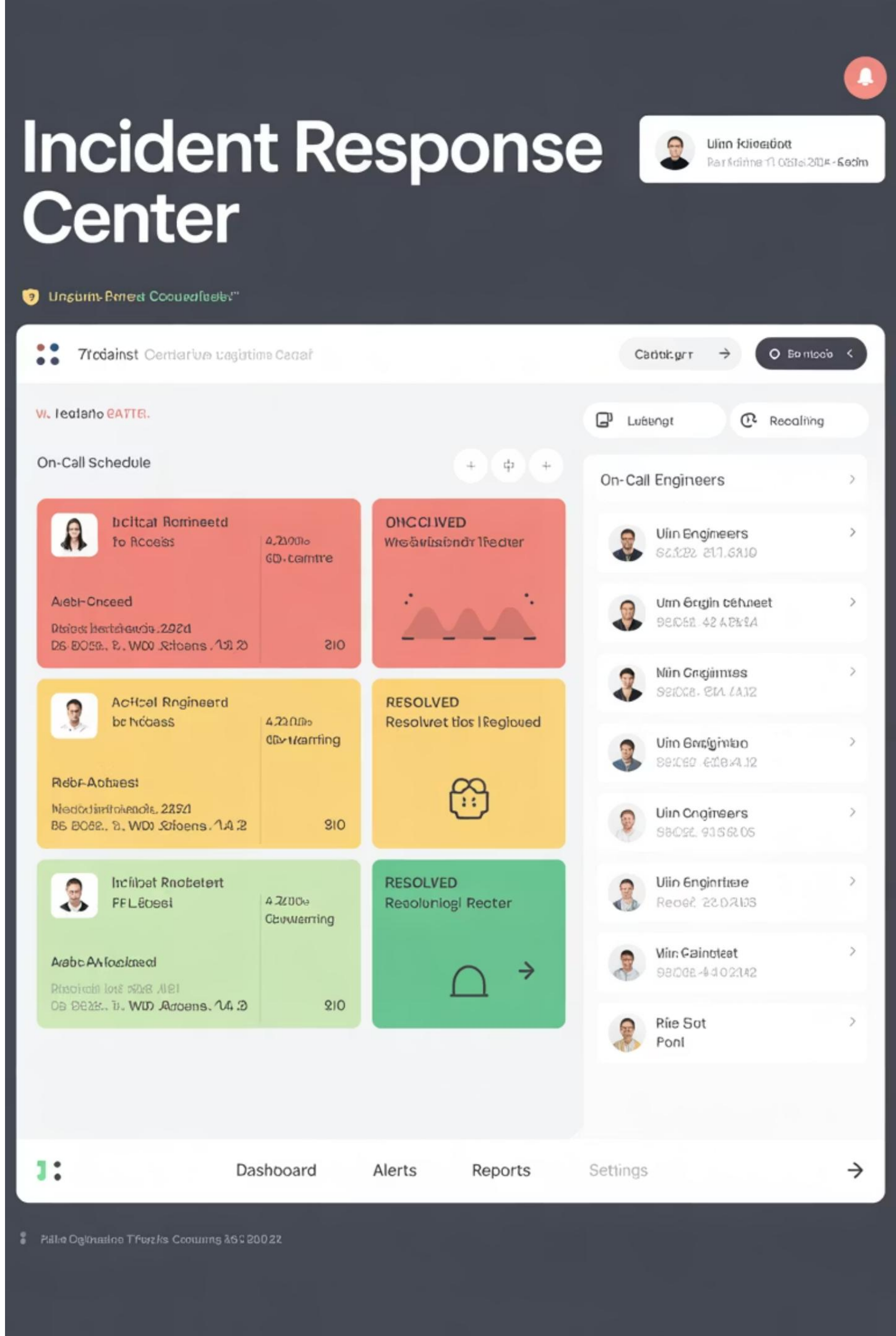
Communication

- Dedicated chat rooms (Slack, Teams)
- Video conferencing for complex incidents
- Status pages for external communication

Coordination

- Incident management platforms (PagerDuty, OpsGenie)
- Runbooks and playbooks
- Shared documents for real-time notes

The right tools make coordination seamless during high-stress incidents.



Case Study: Stripe's Payment Outage

The Incident

During a routine database migration, Stripe experienced an unexpected issue that caused payment processing failures for 23 minutes.

Response Done Right

- Clear incident roles established immediately
- Regular status updates to affected customers
- Transparent public postmortem published after resolution
- Concrete improvements implemented to prevent recurrence

The incident was notable not for its technical complexity, but for the excellent execution of their incident response process.

Incident Commander Best Practices

Stay High-Level

Focus on coordination, not diving into technical details

Communicate Clearly

Regular updates with clear, concise information

Delegate Effectively

Assign specific tasks to specific people

Document Everything

Track decisions, actions, and timeline

The incident commander role is about coordination and leadership, not technical heroics.

Blameless Postmortems: Learning from Failure

> "Failure is inevitable in complex systems. Learning from failure is optional."

A postmortem (or post-incident review) is a written record of an incident, its impact, causes, and the actions taken to prevent recurrence.

The blameless approach focuses on identifying systemic issues rather than individual mistakes. This creates psychological safety, encouraging honesty and deeper learning.

Key Elements of an Effective Postmortem

Incident Summary

Brief overview including duration, impact, and severity

Timeline

Chronological sequence of key events

Root Cause Analysis

What happened and why (may have multiple causes)

Impact Assessment

Quantified effect on users and business

Action Items

Specific, assigned tasks to prevent recurrence

Lessons Learned

Key takeaways for the organization

From Blame to Systems Thinking

1. **Blaming Language**

   "John didn't follow the deployment checklist, causing the outage."

2. **Blameless, Systems-Focused Language**

   "The deployment proceeded without all checklist items completed. Our process allowed this to happen because the checklist verification was manual and performed during a high-stress period."

Blameless doesn't mean without accountability. It means viewing human actions in the context of the systems they operate within.

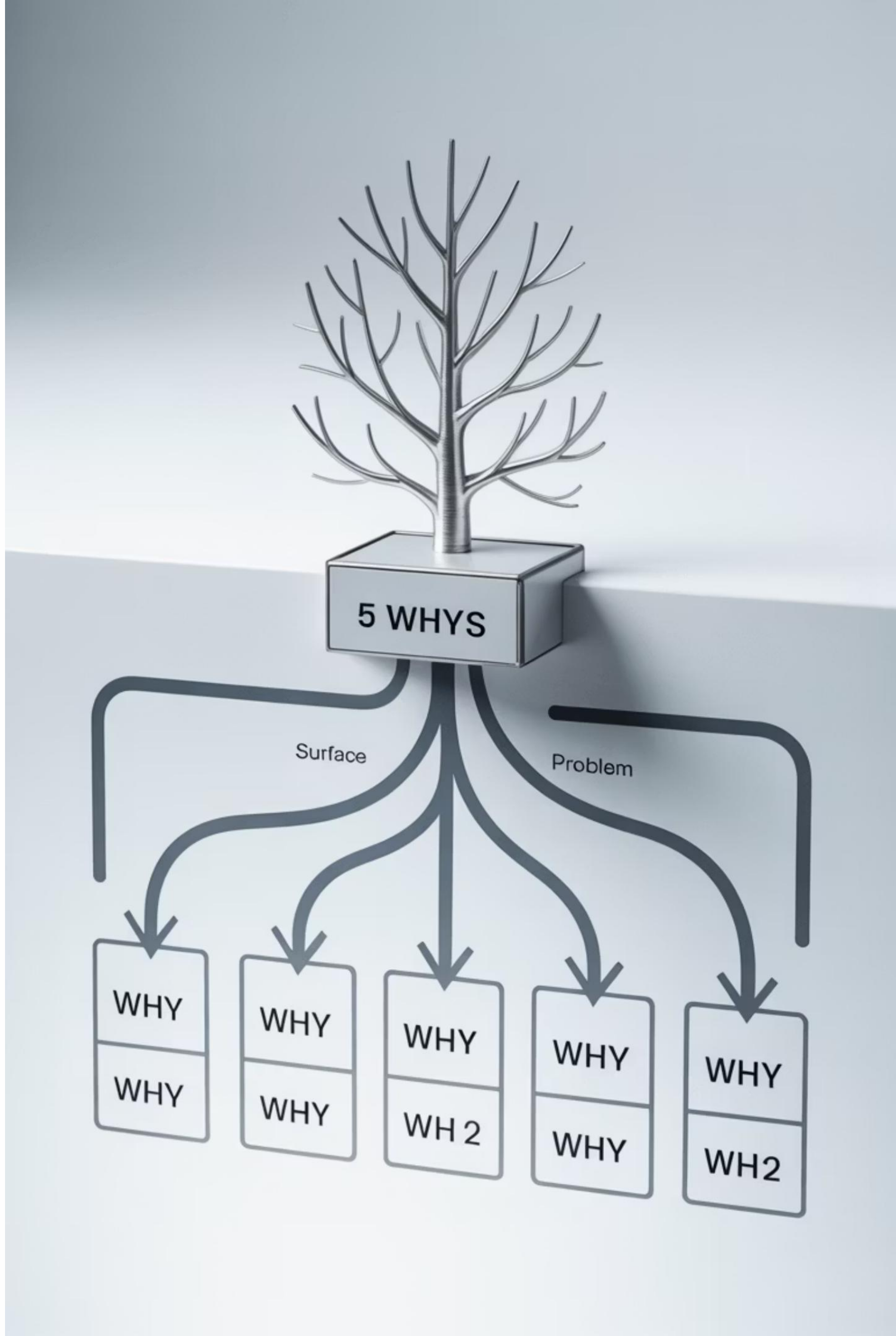
People generally make reasonable decisions based on the information they have at the time. Understanding their context is key to preventing similar issues.

The 5 Whys: Finding Root Causes

The 5 Whys is a simple but powerful technique for discovering the root cause of a problem:

1. Start with the problem statement
2. Ask "Why did this happen?"
3. For each answer, ask "Why?" again
4. Continue until you reach systemic causes (usually 5 levels deep)

This helps move beyond superficial fixes to address underlying issues that could cause future incidents.



5 Whys Example: Database Outage

Problem

The database server crashed, causing a 30-minute outage

Why?

The database ran out of disk space

Why?

Transaction logs filled the disk

Why?

Log rotation failed to run

Why?

The cronjob was accidentally deleted during a server update

Why?

There was no change management process or verification of critical maintenance tasks

The root cause isn't "disk space" – it's the lack of process controls that allowed critical maintenance to be disrupted.

Learning from Near Misses

The aviation industry dramatically improved safety by studying "near misses" – incidents that could have been catastrophic but weren't.

SRE adopts this approach by tracking and analyzing near misses in our systems:

- Automated recovery that prevented user impact
- Issues caught in staging before production
- Problems fixed just before they became critical

These contain valuable lessons without the cost of actual outages.

Building a Postmortem Culture

Psychological Safety

Create an environment where people feel safe sharing mistakes

Celebrate Learning

Recognize and reward thorough, honest postmortems

Share Widely

Distribute lessons across teams to multiply the learning

Follow Through

Track and complete action items from postmortems

Over time, these practices create an organizational culture that learns and improves from incidents rather than hiding or ignoring them.

Implementing SRE in Your Organization

Start Small

Begin with one service or team

- Define SLIs and SLOs
- Implement basic monitoring
- Start tracking toil

Build Practices

Develop core operational procedures

- On-call rotations
- Incident response process
- Blameless postmortems

Expand Scope

Apply SRE principles more broadly

- More services and teams
- Deeper observability
- Error budgets for feature development

Common SRE Implementation Pitfalls

Cargo Culting

Blindly copying Google's practices without understanding the underlying principles

Creating an SRE Silo

Treating SRE as a separate team that "owns reliability" instead of a shared responsibility

Unrealistic SLOs

Setting perfectionistic SLOs (like 100% availability) that drive wrong behaviors

Tool Obsession

Focusing on tools and technologies rather than cultural and process changes

SRE for Smaller Organizations

You don't need to be Google-sized to benefit from SRE principles. Smaller organizations can adapt SRE by:

- Focusing on the highest-impact practices first (SLOs, postmortems)
- Embedding SRE principles within existing engineering teams
- Using managed services to reduce operational overhead
- Simplifying processes while maintaining the core principles

Start small, measure results, and expand gradually as you demonstrate value.

Measuring SRE Success

↓60%	↑99.9%	↓40%	↑25%
Toil Reduction	SLO Compliance	MTTR	Deployment Frequency
Less time spent on repetitive operational tasks	Consistently meeting reliability targets	Faster incident resolution times	More frequent, smaller changes

Effective SRE implementation should show measurable improvements in both reliability metrics and engineering productivity.

The Future of SRE

AIOps Integration

AI/ML for anomaly detection, root cause analysis, and predictive insights

Chaos Engineering

Deliberately injecting failures to build resilience

Service Meshes

Infrastructure layer for reliability, observability, and security

SRE as Business Function

Reliability as a competitive advantage, not just an engineering concern

As systems become more complex, SRE practices will continue to evolve to meet new challenges.

Continuous Learning Resources

Google SRE Books

The original SRE books from Google, available free online

Industry Books

"Implementing Service Level Objectives" by Alex Hidalgo

"Seeking SRE" by David N. Blank-Edelman

Conferences

SREcon, DevOps Days, and Chaos Conf

Online Courses

LinkedIn Learning, Coursera

Key Takeaways

1

Reliability is an Engineering Problem

Apply software engineering principles to operations

2

Define and Measure What Matters

SLIs, SLOs, and error budgets provide data-driven reliability

3

Embrace Systems Thinking

Look beyond isolated components to understand interactions

4

Build Observable Systems

Metrics, logs, and traces provide insight into complex systems

5

Learn from Failures

Blameless postmortems turn incidents into improvement opportunities

Thank You!