



Reliability Engineering: Error Budgets & Infrastructure as Code

A practical guide to building reliable systems while maintaining development velocity

Your Journey Through Reliability Engineering

1 SLIs & Error Budgets

Understanding reliability metrics

2 Implementing Error Budget Policies

Creating accountability

3 Infrastructure as Code

Managing infrastructure reliably

4 Best Practices

Real-world implementation

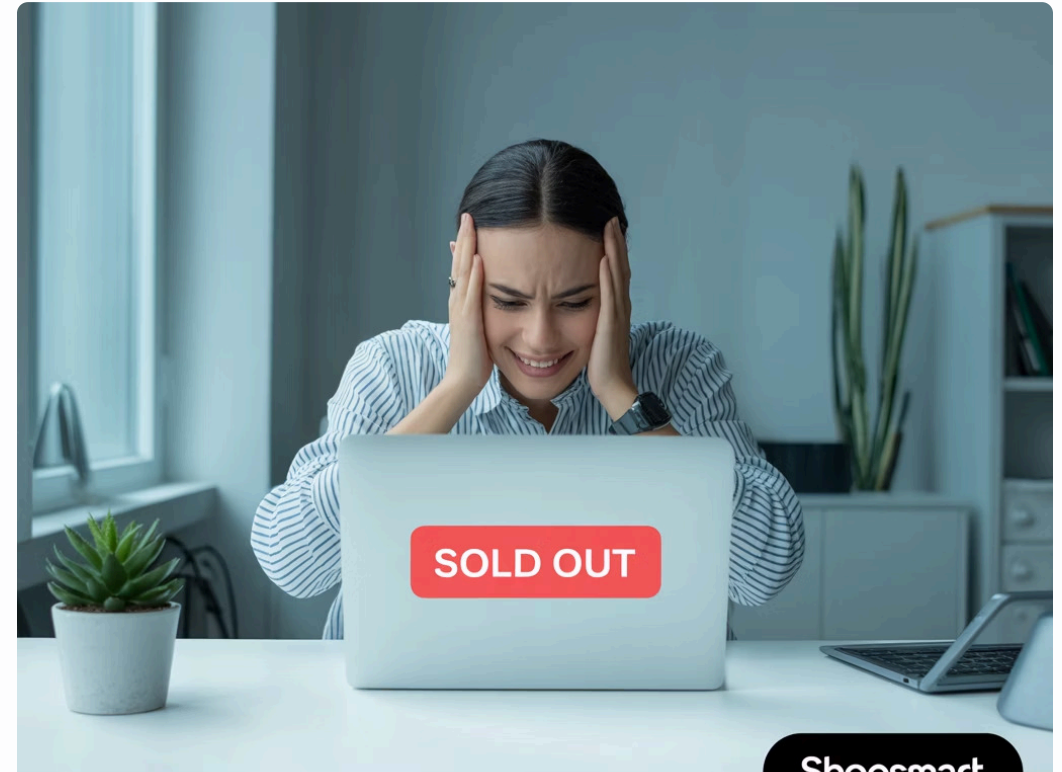
By the end of this presentation, you'll understand how to quantify reliability and manage infrastructure changes safely.

The Reliability Challenge

Imagine you're running an e-commerce website during Black Friday. Your site experiences slowdowns, causing frustrated customers to abandon their shopping carts.

How do you:

- Define what "reliable" means?
- Balance new features vs. stability?
- Ensure infrastructure changes don't break things?



Shopsmart

Don't miss out.
Set alerts

These are the exact challenges we'll address today.

Part 1: Service Level Indicators

The Foundation of Reliability Metrics

What Are Service Level Indicators (SLIs)?

Definition

A quantitative measure of service performance that matters to users.

Formula: $SLI = \text{Good Events} / \text{Valid Events}$

Common Examples

- Availability: % of successful requests
- Latency: % of requests faster than threshold
- Throughput: requests processed per second
- Error rate: % of error-free responses

SLIs translate vague notions like "the service is slow" into precise measurements like "95% of requests complete in under 200ms."

Real-World SLI Example: Coffee Shop



Think of a coffee shop where customers expect:



To be served within 5 minutes of entering



To receive the correct order



Coffee to be at the right temperature

The SLI might be: "Percentage of customers who receive correct orders within 5 minutes."

Choosing the Right SLIs

Good SLIs focus on the user experience, not internal metrics that users don't care about.

User-Centric


Measures aspects users actually notice
(page load time vs. CPU usage)

Measurable

Can be consistently tracked and quantified

Actionable

When it degrades, you know what to investigate

 **Pro Tip:** Start with 2-5 SLIs that directly impact user satisfaction. Too many SLIs create noise and confusion.

From SLIs to SLOs and SLAs

SLI (Service Level Indicator)

The metric itself: "Request latency is 150ms"

SLO (Service Level Objective)

The target for the SLI: "99% of requests will complete in under 200ms"

SLA (Service Level Agreement)

The contract with consequences: "If more than 1% of requests exceed 200ms in a month, customers get a 10% refund"

SLIs measure performance, SLOs set expectations, and SLAs define the business impact of missing those expectations.

Part 2: Error Budgets

Quantifying Acceptable Imperfection

What Is an Error Budget?

An error budget is the allowed amount of failure in a service based on its SLO.

If your SLO is 99.9% availability:

- Your error budget is 0.1%
- That's about 43 minutes of downtime per month
- Once exhausted, feature work stops until reliability improves



Your "budget" for acceptable failures

Calculating an Error Budget

1

Define your SLO

Example: 99.95% of requests should be successful

2

Calculate your error budget

$100\% - 99.95\% = 0.05\%$ of requests can fail

3

Determine time window

For a 30-day month: 0.05% of 43,200 minutes = 21.6 minutes of allowed downtime

4

Track consumption

Monitor how quickly you're using your error budget

Real-World Example: Movie Streaming Service

Consider a streaming service with:

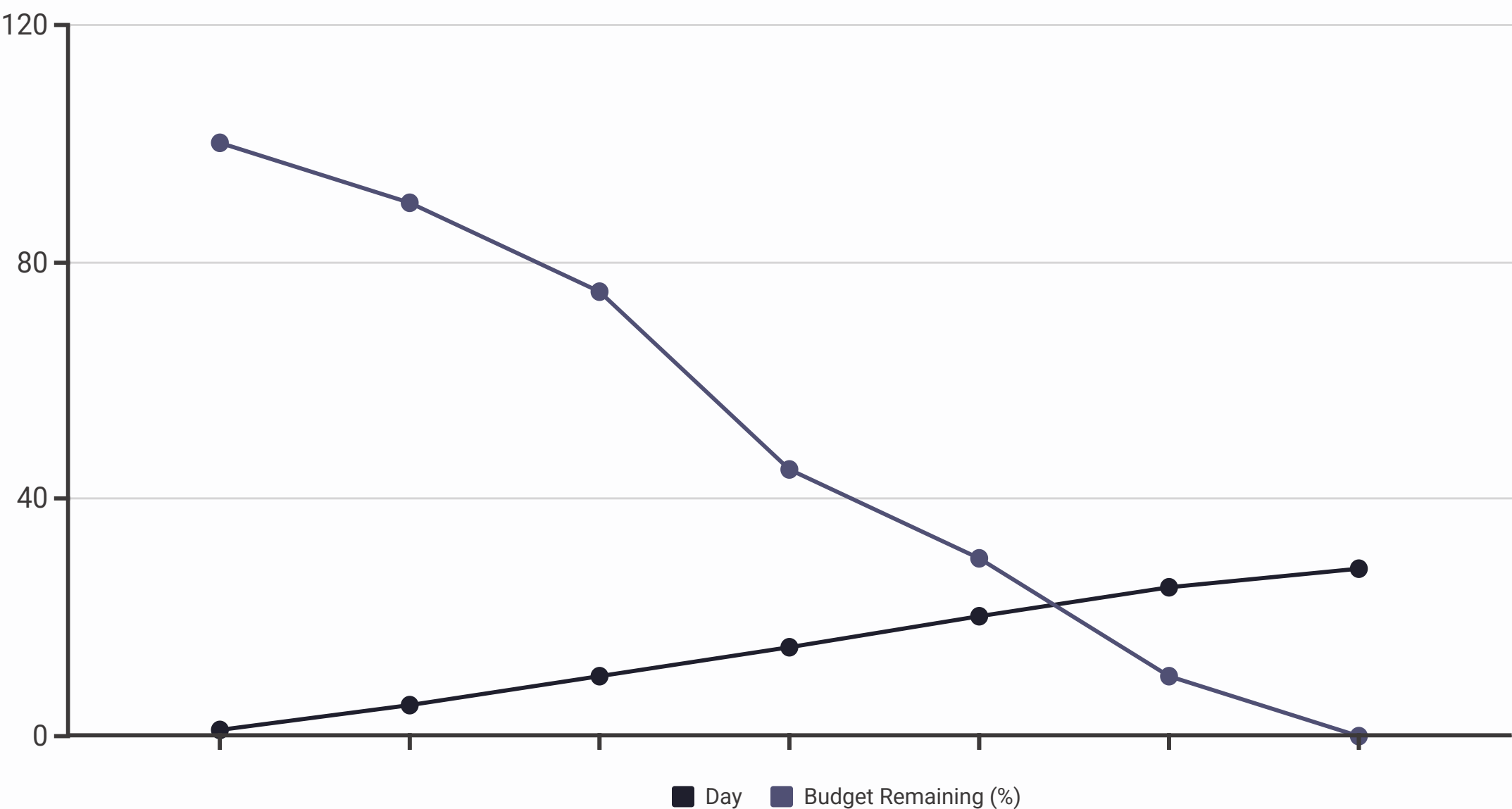
- 10 million daily active users
- SLO: 99.9% of streams start within 3 seconds
- Error budget: 0.1% (10,000 slow starts per day)

During a new feature rollout, 15,000 users experience slow starts, exceeding the daily budget by 5,000 users.



Visualizing Error Budget Consumption

This chart illustrates the daily consumption of our movie streaming service's error budget over a month. Recall from the previous card that our Service Level Objective (SLO) allows for 0.1% of streams to start slowly, which translates to a daily error budget of 10,000 slow starts (based on 10 million daily active users). The line on the chart indicates the percentage of the monthly error budget remaining.



As the chart shows, our error budget was steadily consumed over the month. However, a significant drop occurred on Day 15: a problematic new feature deployment led to 15,000 users experiencing slow starts. This single event consumed 1.5 days' worth of our daily budget, severely accelerating our budget depletion. Consequently, the entire monthly error budget was exhausted by Day 28. This early depletion means that for the remaining two days of the month, the team would need to immediately halt new feature rollouts, prioritize bug fixes, and focus solely on improving reliability to prevent further breaches and restore service quality.

Part 3: Error Budget Policies

Creating Accountability for Reliability

What Is an Error Budget Policy?

An error budget policy is a set of rules that dictate what happens when error budgets are consumed.

Clear Guidelines

Defines automatic actions when budget thresholds are crossed

Shared Responsibility

Agreed upon by both development and operations teams

No Finger-Pointing

Based on objective measurements, not subjective blame

The policy creates a feedback loop where reliability problems automatically slow down feature development.

Sample Error Budget Policy

Budget at 75%

Warning issued to teams

Review upcoming feature launches

Budget at 50%

Launch reviews become more stringent

Some non-critical launches may be postponed

Budget at 25%

Only essential bug fixes and security patches allowed

Form reliability task force

Budget Exhausted

Feature freeze - all work focuses on reliability

Incident review with executive visibility

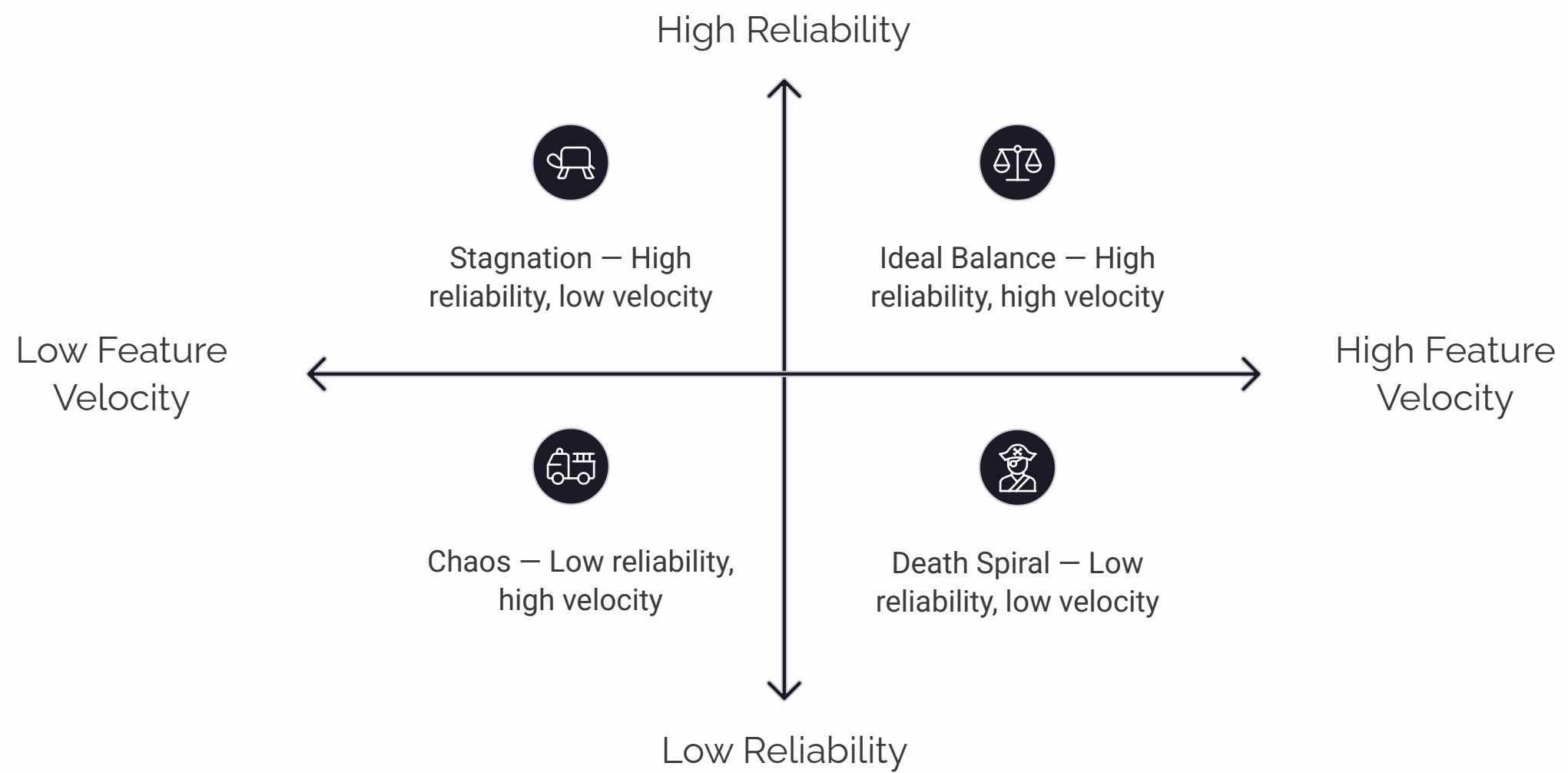
Real-World Example: Retail Website

A retail website implements an error budget policy:

1. After a database migration burns 60% of the monthly error budget, the policy automatically postpones a UI redesign launch
2. Engineering teams shift focus to improving database performance
3. Within a week, they identify and fix query optimization issues
4. The error budget recovers through natural expiration of the measurement window
5. The UI redesign proceeds the following month with minimal impact



Balancing Feature Velocity with Reliability



Error budgets create a self-correcting system: When reliability is good, teams move faster on features. When reliability suffers, they automatically shift focus to stability.

Part 4: Infrastructure as Code

Building Reliable Systems Through Automation

What Is Infrastructure as Code?

Infrastructure as Code (IaC) is the practice of managing infrastructure through code files rather than manual processes.

Key benefits:

- Consistency across environments
- Version control for infrastructure
- Automated deployment and testing
- Self-documenting infrastructure



The Manual Infrastructure Problem

Without Infrastructure as Code, teams face multiple challenges:



Snowflake Servers

Each server becomes unique and irreproducible



Undocumented Changes

Configuration drifts without record of what changed



Slow Recovery

Disaster recovery becomes lengthy and error-prone



Tribal Knowledge

System knowledge lives only in people's heads

Real-World Example: The 2 AM Database Crash



Without IaC:

1. Production database crashes at 2 AM
2. On-call engineer logs in, tries to remember configuration
3. Manually rebuilds server with different settings
4. Service restored but mysterious performance issues appear
5. Takes days to identify the misconfiguration

With IaC:

1. Run automated recovery script
2. Exact configuration reapplied
3. Service restored in minutes, not hours

Popular Infrastructure as Code Tools



Terraform

Declarative tool for provisioning infrastructure across multiple cloud providers



Ansible

Procedural configuration management tool that works without agents



CloudFormation

AWS-specific service for modeling and provisioning resources



Pulumi

Infrastructure as actual code using familiar programming languages

Each tool has strengths in different scenarios - many organizations use a combination.

IaC Example: Creating a Web Server with Terraform

```
provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "web" {  
  ami          = "ami-0c55b159cbfafa1f0"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "WebServer"  
    Environment = "Production"  
  }  
}  
  
resource "aws_security_group" "web" {  
  name = "web-server"  
  
  ingress {  
    from_port = 80  
    to_port   = 80  
    protocol  = "tcp"  
    cidr_blocks = ["0.0.0.0/0"]  
  }  
}
```

This Terraform code:

1. Specifies AWS as the provider
2. Creates a t2.micro EC2 instance
3. Sets identifying tags
4. Creates a security group allowing HTTP traffic

You can deploy this exact configuration to multiple environments and track changes with version control.

Managing Infrastructure Changes Safely



Version Control

Store infrastructure code in git repositories alongside application code



Pull Requests

Review infrastructure changes before applying them



Automated Testing

Validate changes in test environments first



Deployment Pipeline

Automate the application of approved changes

These practices help prevent outages caused by misconfigurations and ensure changes are predictable.

Infrastructure as Code Best Practices

Keep It DRY

Use modules and reusable components to avoid duplication

Immutable Infrastructure


Replace entire resources rather than modifying them in place

Small Changes

Make incremental changes rather than big bang deployments

Secure Storage

Never commit secrets to your code repositories

 **Pro Tip:** Apply the same code quality standards to infrastructure code as you do to application code.

How IaC and Error Budgets Work Together

Infrastructure as Code makes it easier to respond to error budget consumption by enabling rapid, tested, and consistent improvements to your infrastructure.

Real-World Success Story: E-commerce Platform

An e-commerce company struggling with reliability implemented:

1. SLIs based on checkout flow success rate and page load times
2. 99.95% SLO with error budget policy
3. Infrastructure as Code for all production systems

Results after 6 months:

- Deployment frequency increased by 35%
- Mean time to recovery decreased from 3 hours to 15 minutes
- Customer complaints about site stability dropped by 80%



Key Takeaways

Define What "Good" Means

SLIs and SLOs provide objective measurements of reliability

Quantify Acceptable Risk

Error budgets create a common language between development and operations

Automate Infrastructure

Infrastructure as Code reduces human error and speeds recovery

Start Simple

Begin with a few critical SLIs and gradually expand your reliability practices

Next Steps

Immediate Actions

1. Identify your most important user journeys
2. Define 2-3 SLIs that directly measure user experience
3. Set initial SLOs (aim for realistic, not perfect)
4. Create a simple error budget policy
5. Start small with Infrastructure as Code (even for a single service)

Resources

- [Google SRE Book: Service Level Objectives](#)
- [Terraform Documentation](#)
- [Implementing SLOs](#)

