Reducing Toil Through Automation

A practical guide to identifying and eliminating unnecessary work through strategic automation in engineering organizations

Agenda

What Is Toil?

Google SRE Definition

Work that is:

- Manual
- Repetitive
- Automatable
- Tactical (not strategic)
- Without enduring value
- Scales linearly with service growth

Real-World Examples

- Manually restarting crashed services
- Responding to repetitive alerts
- Performing routine data cleanups
- Creating user accounts
- Running manual deployments

Toil isn't just annoying—it actively prevents teams from doing higher-value work that delivers business impact.

The Hidden Cost of Toil

Toil costs more than you think:

- **Time theft**: Engineers spend 21-40% of their time on toil (Gartner)
- **Morale damage**: Repetitive work contributes to burnout
- **Opportunity cost**: Every hour spent on toil is an hour not spent on innovation
- **Scalability ceiling**: Manual processes create hiring dependencies
- **Technical debt**: Short-term manual fixes often become permanent

Identifying Toil in Your Organization

Time Tracking

Have engineers log time spent on manual, repetitive tasks for 2 weeks. Look for patterns across teams.

Ticket Analysis

Review support tickets and incident reports. Identify frequent, similar issues that consume engineer time.

Team Interviews

Ask: "What tasks do you perform regularly that feel like they don't add value?" or "What would you automate first?"

Process Mapping

Document workflows to identify manual handoffs, approvals, and repetitive steps that could be automated.

Real-World Example: Identifying Toil

Case Study: E-commerce Platform

The operations team at an e-commerce company was struggling to keep up with growth. After analyzing their tasks:

Before Analysis

"We're just busy all the time."

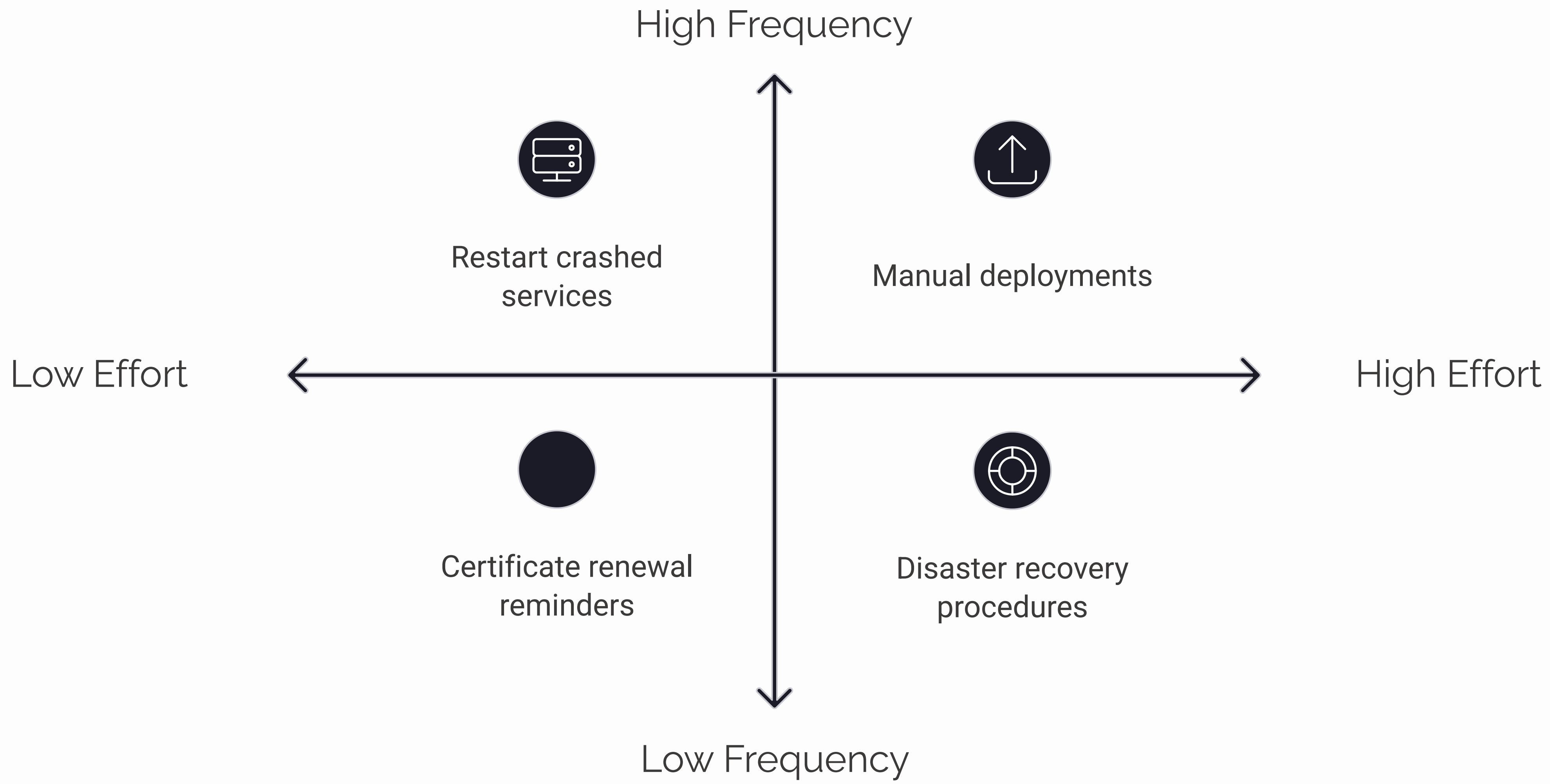
After Analysis

46% of time spent on manual database cleanups, certificate renewals, and user permission changes

This discovery led to a focused automation initiative that reduced toil by 62% in three months.

Toil Assessment Matrix

Prioritize automation efforts by mapping your toil activities on this matrix. Tasks in the high-frequency, low-effort quadrant offer the best initial return on investment. Tasks in the high-frequency, high-effort quadrant may require breaking down into smaller components.



Core Principles of Automation

Automate incrementally

Start with high-value, low-complexity tasks. Build momentum with early wins.

Secure by design

Automated systems should enforce security practices, not bypass them.

Observable automation

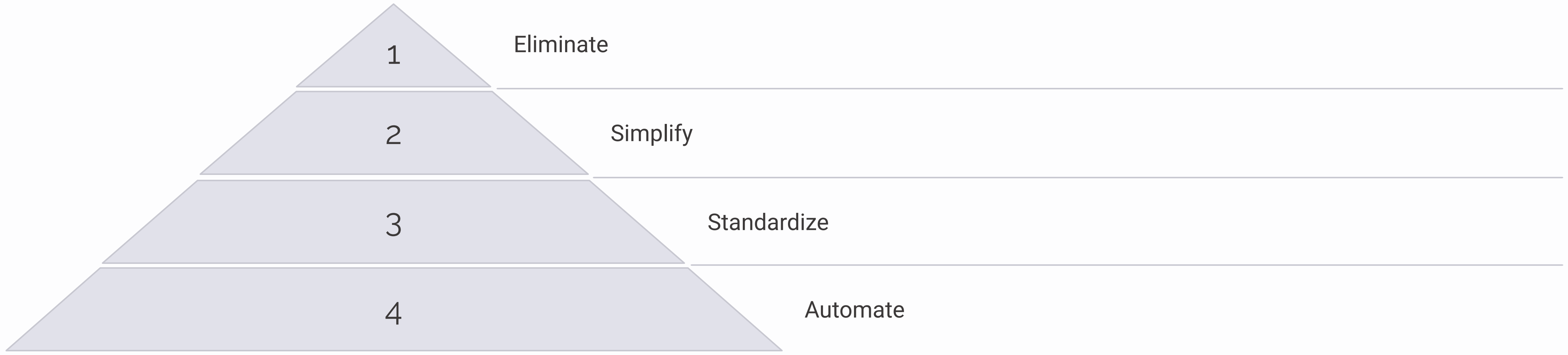
Automated systems must be monitored and produce meaningful logs.

Human-centered

Automation should empower humans, not just replace them.

Effective automation isn't just about technology—it's about a strategic approach that creates lasting, maintainable solutions.

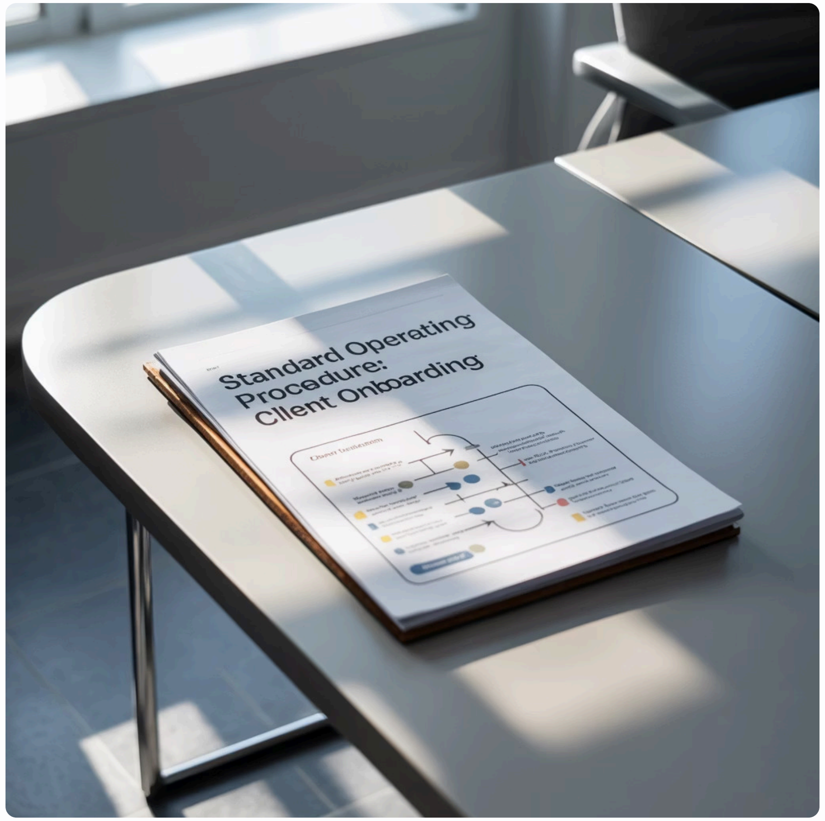
The Automation Hierarchy



Before automating any process, first ask: Can we eliminate this work entirely? Can we simplify it? Can we standardize it? Only then should we invest in automation. Many organizations waste resources automating processes that could be eliminated or simplified.

Example: Instead of automating complex database backups, consider using a managed database service that handles backups automatically.

Runbooks: The First Step Toward Automation



What makes an effective runbook?

- **Clear triggers** that indicate when to use it
- **Step-by-step procedures** with expected outcomes
- **Decision trees** for handling variations
- **Verification steps** to confirm success
- **Links to related resources** and tools
- **Explicit automation candidates** marked for future work

Well-documented runbooks are the foundation for future automation efforts.

From Runbooks to Auto-Remediation

Manual Process

Engineers follow documented steps in a runbook each time an issue occurs

Assisted Process

Tools suggest actions based on runbooks, but engineers execute them

Semi-Automated

System executes predefined actions after human approval

Fully Automated

System detects issues and applies remediation without human intervention

The journey from manual processes to auto-remediation is evolutionary, not revolutionary. Build confidence at each stage before progressing.

Real-World Example: Auto-Remediation

Case Study: Netflix Chaos Monkey

Netflix famously created a suite of tools that deliberately cause failures in their production environment to verify that their auto-remediation systems work correctly.

Their philosophy: "If something hurts, do it more often."

By intentionally causing frequent, small failures, they:

- Built robust auto-healing capabilities
- Eliminated manual recovery procedures
- Reduced the impact of real failures

Automation Tools: Cron Jobs

What are cron jobs?

Scheduled tasks that run at predefined times on Unix-based systems.

Best used for:

- Recurring maintenance tasks
- Scheduled reports generation
- Regular data processing
- Periodic cleanup operations

Common pitfalls:

- **Silent failures**: Scripts crash but no one gets notified
- **Overlapping runs**: New job starts before previous one finishes
- **Wrong timezone**: Job runs at 2am UTC instead of 2am local time
- **No failure tracking**: Can't tell if jobs succeeded or failed

```
# Run backup at 2am every day
0 2 * * * /scripts/backup.sh

# Generate report at 8am on Monday
0 8 * * 1 /scripts/weekly-report.sh

# Check disk space every 30 minutes
*/30 * * * * /scripts/check-disk.sh
```

Modern alternatives: Kubernetes CronJobs, AWS EventBridge Scheduler, or managed scheduling services provide more reliability and observability than traditional cron.

Automation Tools: Scripting

Bash/Shell Scripts

Strengths: Available everywhere, great for system tasks

Weaknesses: Error handling, complex logic

Example use: Log rotation, simple file operations

Python

Strengths: Readability, vast library ecosystem

Weaknesses: Deployment dependencies

Example use: Data processing, API interactions

PowerShell

Strengths: Windows integration, object-oriented

Weaknesses: Less portable to non-Windows

Example use: Windows system administration

JavaScript/Node.js

Strengths: Async operations, web integration

Weaknesses: Callback complexity

Example use: Web scraping, API automation

Event-Based Automation

What is event-based automation?

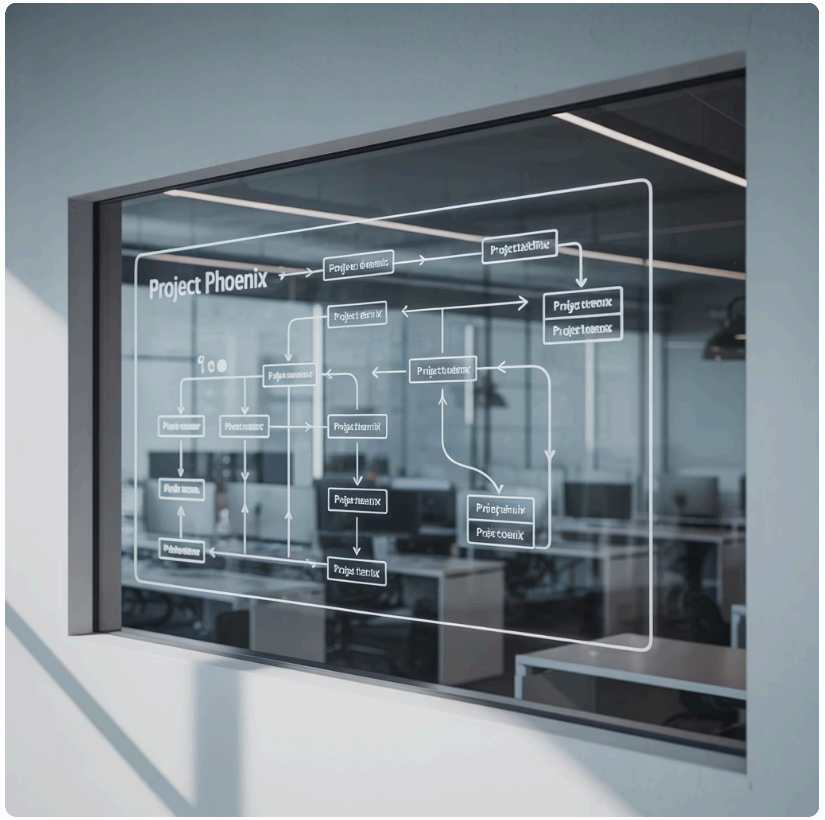
Systems that trigger automated responses based on events rather than schedules.

Advantages over cron/scheduled jobs:

- Immediate response to conditions
- Better resource utilization
- Scales with actual workload
- Reduces polling overhead

Common events:

- File system changes
- System metrics crossing thresholds
- API calls or webhook deliveries
- Message queue notifications



Real-World Example: Event-Based Automation

Case Study: Slack's Auto-Scaling Infrastructure

Slack built an event-driven system that automatically responds to infrastructure events:

Detect

Monitoring system detects high CPU utilization across multiple servers

Event

System generates "scaling needed" event with contextual data

Analyze

Auto-scaler evaluates the event against scaling policies

Act

System provisions new servers and updates load balancers

This approach eliminated manual scaling operations, reducing response time from 30+ minutes to under 3 minutes.

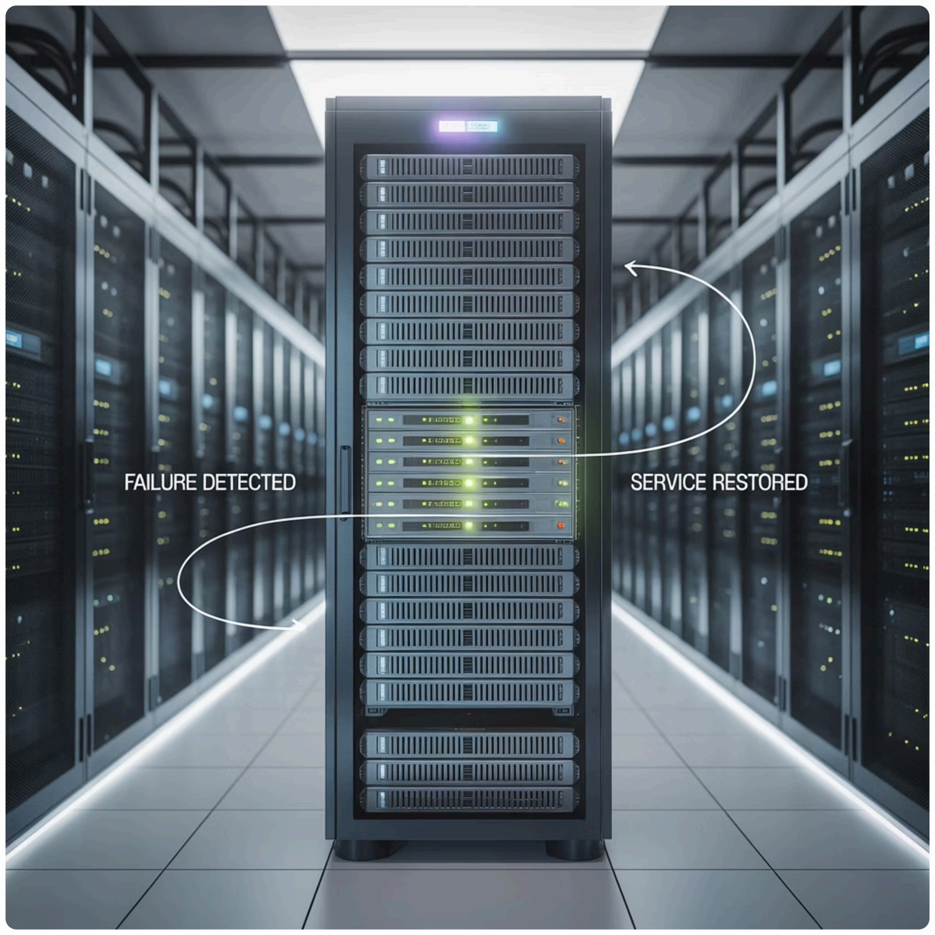
Real-World Example: Automated Service Restarts

Scenario: Web Service Outage

A critical web application service (e.g., Nginx, Apache, or a custom backend) frequently experiences intermittent crashes under heavy load, leading to user-facing downtime.

Manual Runbook (Simplified)

1. **Detect:** Receive alert from monitoring system (e.g., PagerDuty, Prometheus) indicating service is down.
2. **Verify:** SSH into the server and run `systemctl status webapp` to confirm the service state.
3. **Attempt Restart:** If service is inactive, run `sudo systemctl restart webapp`.
4. **Verify Again:** Re-check service status. If active, notify team of recovery.
5. **Escalate:** If service remains down after restart, escalate to on-call engineer for deeper investigation.



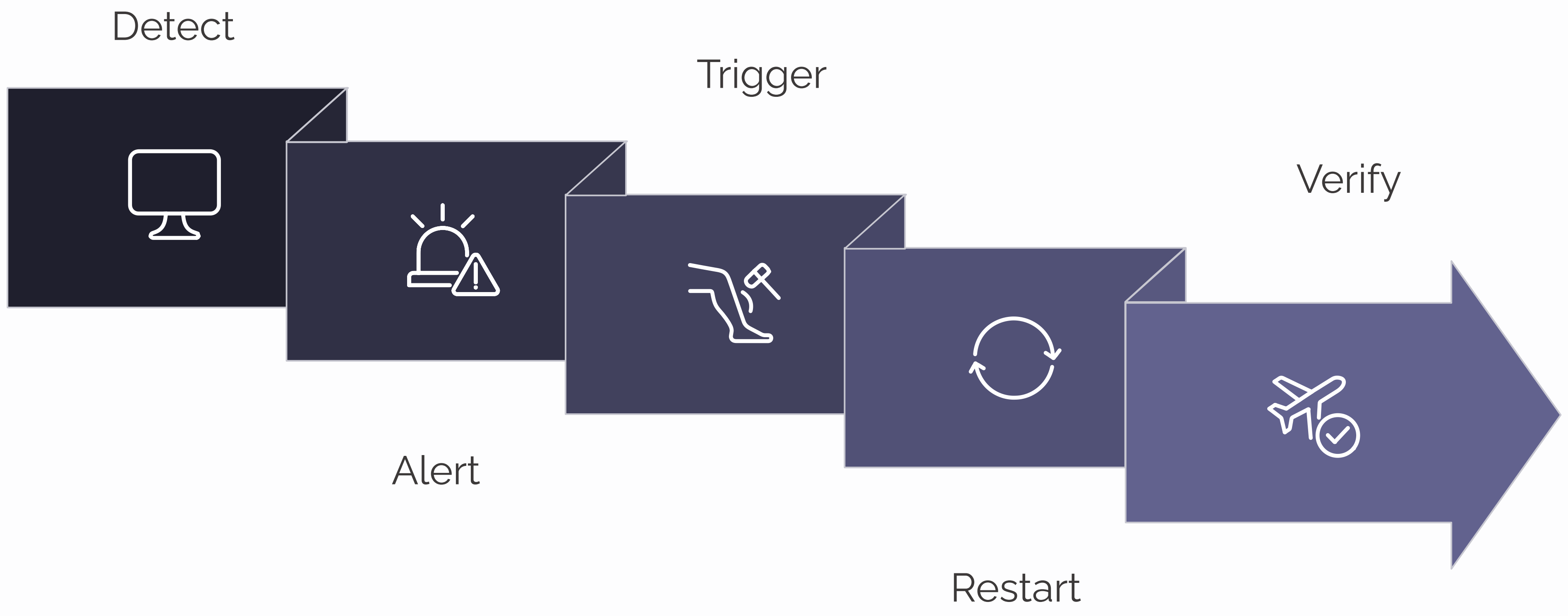
Automated Action

```
# Simplified pseudo-code for an auto-remediation script
if service_status("webapp") == "inactive":
    log_event("Attempting to restart webapp service...")
    execute_command("sudo systemctl restart webapp")
    sleep(10) # Wait for service to start
    if service_status("webapp") == "active":
        send_notification("Webapp service successfully
restarted automatically.")
    else:
        send_alert("Webapp service failed to restart.
Escalating.")
else:
    log_event("Webapp service is running.")
```

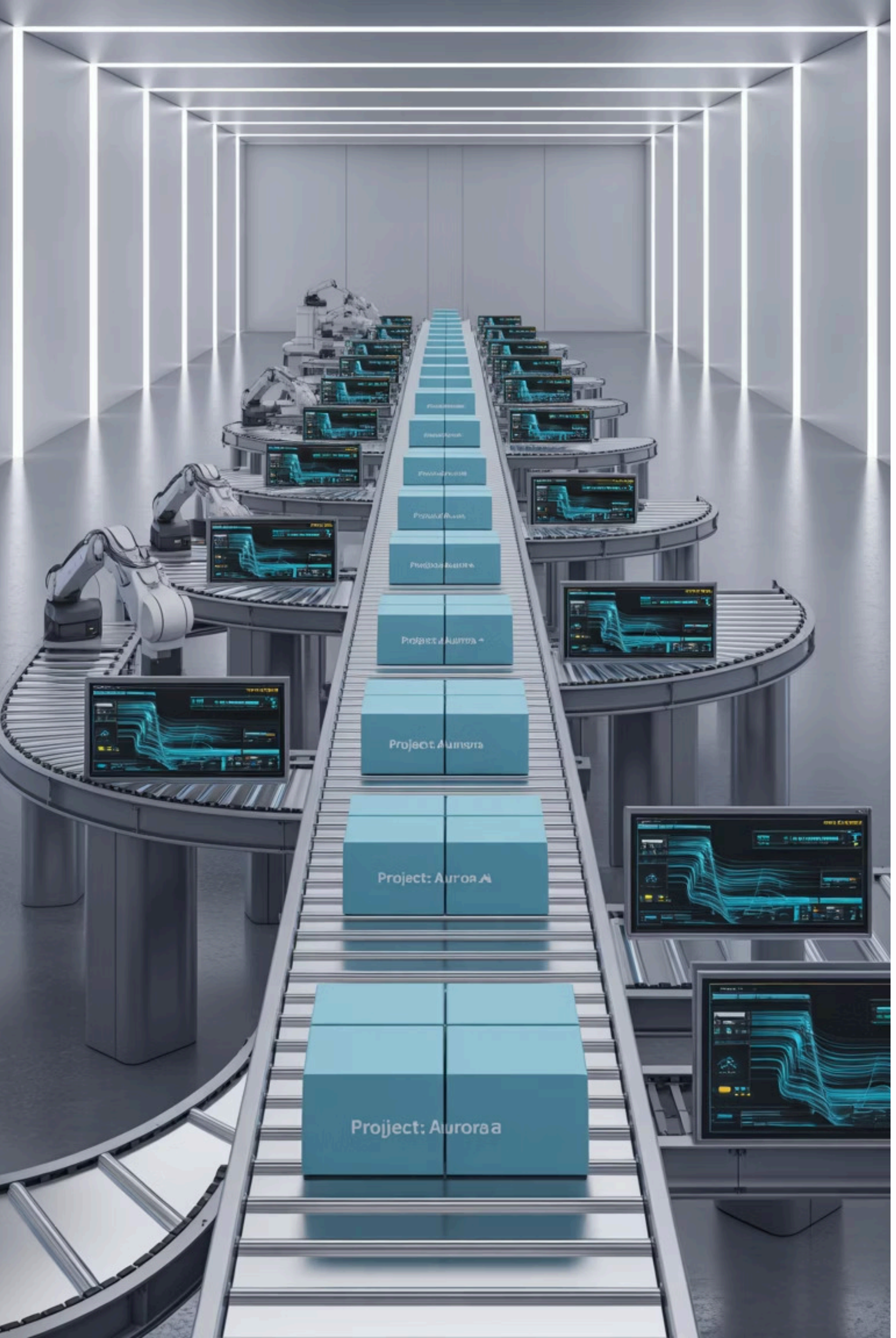
By automating this common runbook, the service can be restored within seconds of failure detection, significantly reducing Mean Time To Recovery (MTTR) and freeing up engineering time.

Real-World Example: Automated Service Restarts

A common source of toil for on-call engineers is manually restarting services that have crashed or become unresponsive. Automating this process significantly reduces operational burden and improves system uptime.



This automated approach ensures faster recovery times, reduces human error during stressful incidents, and allows engineers to focus on more complex, value-adding tasks.



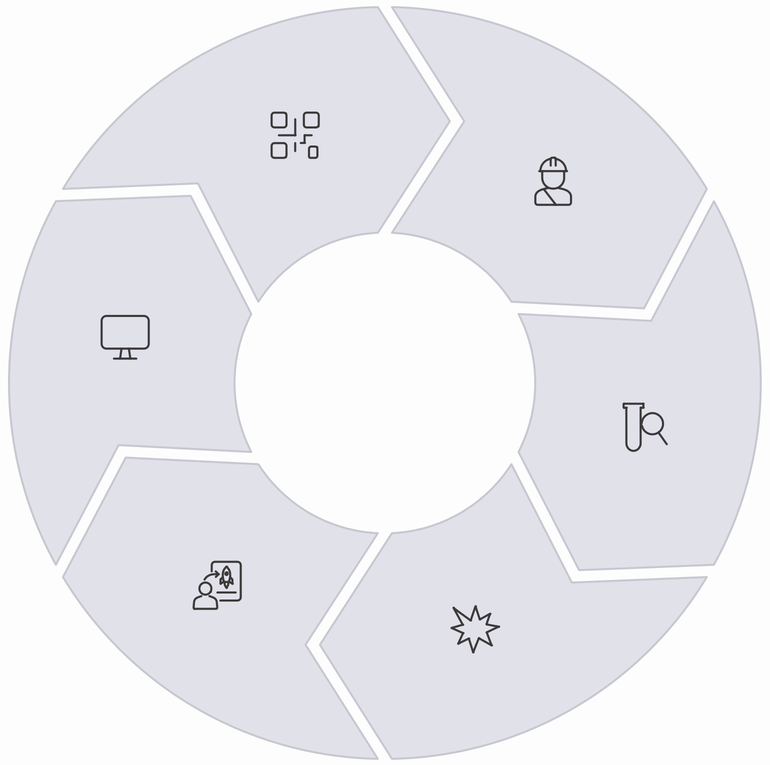
Building Reliable Deployment Pipelines

A reliable deployment pipeline automates the journey from code commit to production deployment, ensuring consistency and quality at each step.

Key components of an effective pipeline:

- **Source control integration** that triggers builds on commits
- **Automated testing** that prevents defective code from progressing
- **Artifact management** for versioned, immutable build outputs
- **Deployment automation** that eliminates manual steps
- **Verification checks** that confirm successful deployment

CI/CD Pipeline Components



Code

Developer commits code to repository

Build

System compiles code and runs static analysis

Test

Automated tests verify functionality

Release

Build artifacts are versioned and stored

Deploy

Code is deployed to target environment

Monitor

System checks deployment health

Each step in the pipeline should be automated, verified, and capable of stopping the process if quality gates aren't met.

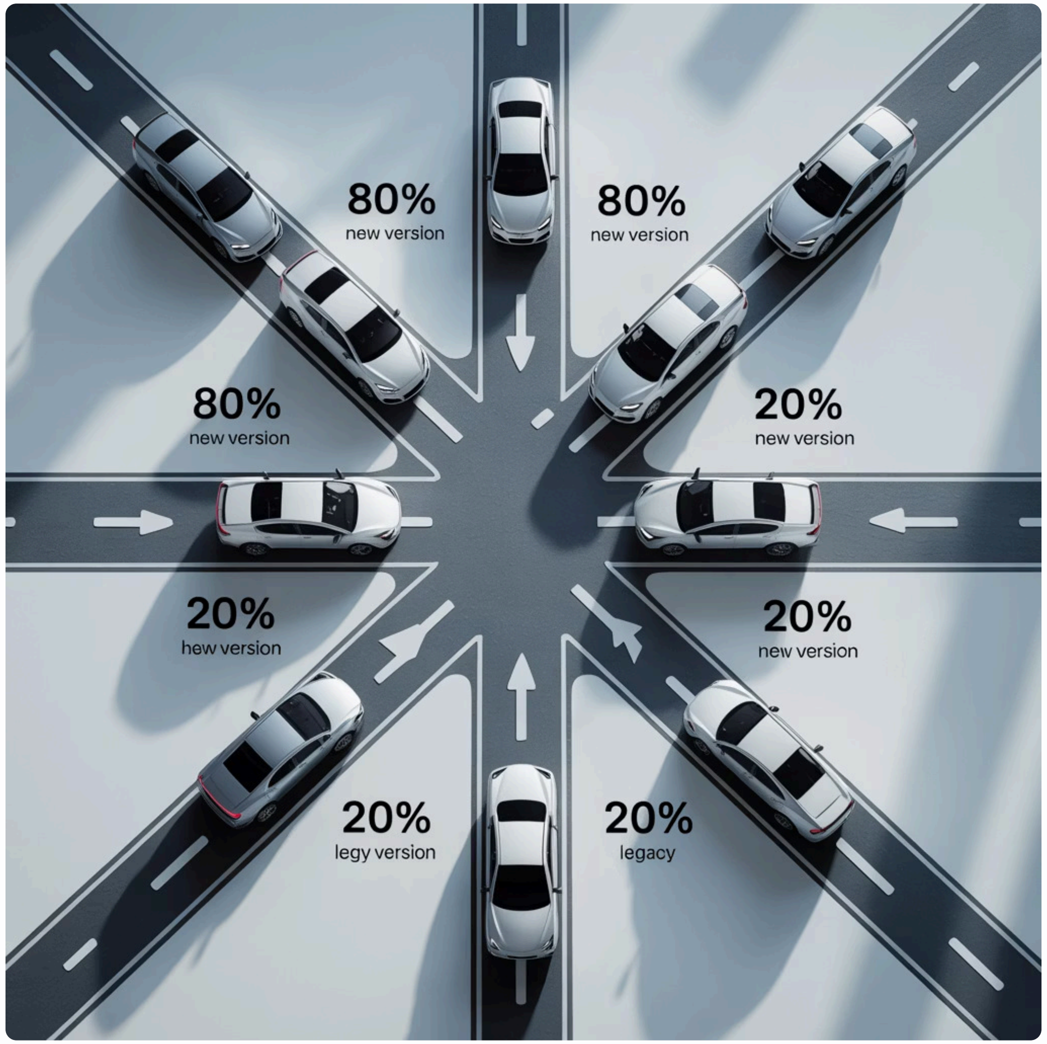
Deployment Strategy: Canary Deployments

What is a canary deployment?

A technique where new code is deployed to a small subset of servers or users before rolling out to the entire infrastructure.

Benefits:

- Early detection of issues with minimal impact
- Real-world validation with actual users
- Ability to quickly revert problematic changes
- Progressive confidence building



Named after the "canary in the coal mine" — miners would bring canaries into mines to detect toxic gases before they affected humans.

Real-World Example: Canary Deployments

Case Study: Google's Production Deployment

Start Small

Deploy to 1% of production servers in a single datacenter

Monitor Closely

Watch key metrics for 30-60 minutes (error rates, latency, CPU usage)

Expand Gradually

If metrics are good, increase to 5%, then 20%, then 50% of servers

Complete Rollout

After passing all verification gates, deploy to 100% of production

Google found that this approach caught 70% of production issues during the early canary phase, affecting only a tiny fraction of traffic.

Deployment Strategy: Blue-Green Deployments

What is Blue-Green Deployment?

You have two identical copies of your app - let's call them Blue and Green. This robust deployment strategy minimizes downtime and risk during updates by ensuring a seamless transition between versions. It's particularly useful for mission-critical applications where uninterrupted service is paramount.

How it works:

1. Blue is live (serving users)
2. Green is empty
3. Deploy new version to Green
4. Test Green thoroughly
5. Switch users from Blue to Green
6. Now Green is live, Blue is backup

Benefits:

- No downtime during updates
- Easy rollback if problems occur
- Safe testing without affecting users

Blue-Green vs Canary: When to Use Each

Blue-Green is better when:

- You need zero downtime deployments
- You want instant rollback capability
- You have sufficient infrastructure resources
- You're deploying major changes that need full testing

Canary is better when:

- You want to minimize risk with gradual rollouts
- You have limited infrastructure resources
- You want real user feedback before full deployment
- You're testing new features with actual traffic

Key difference: Blue-Green switches 100% of traffic at once, while Canary gradually increases traffic to the new version.

Feature Flags

The Problem: You've deployed new code, but you don't want all users to see the new feature immediately. Maybe it's still in testing, or you want to monitor its performance with a small group first.

The Solution: Feature flags allow you to hide or show specific features to different users without changing or redeploying your core application code.

How They Work

Feature flags are simple settings that act like an on/off switch for parts of your application. You embed conditional logic in your code, like this:

```
if (featureFlag.isEnabled('new-dashboard')) {
  showNewDashboard();
} else {
  showClassicDashboard();
}
```

Then, you can toggle the 'new-dashboard' flag on or off for different users or environments, controlling who sees the new experience.

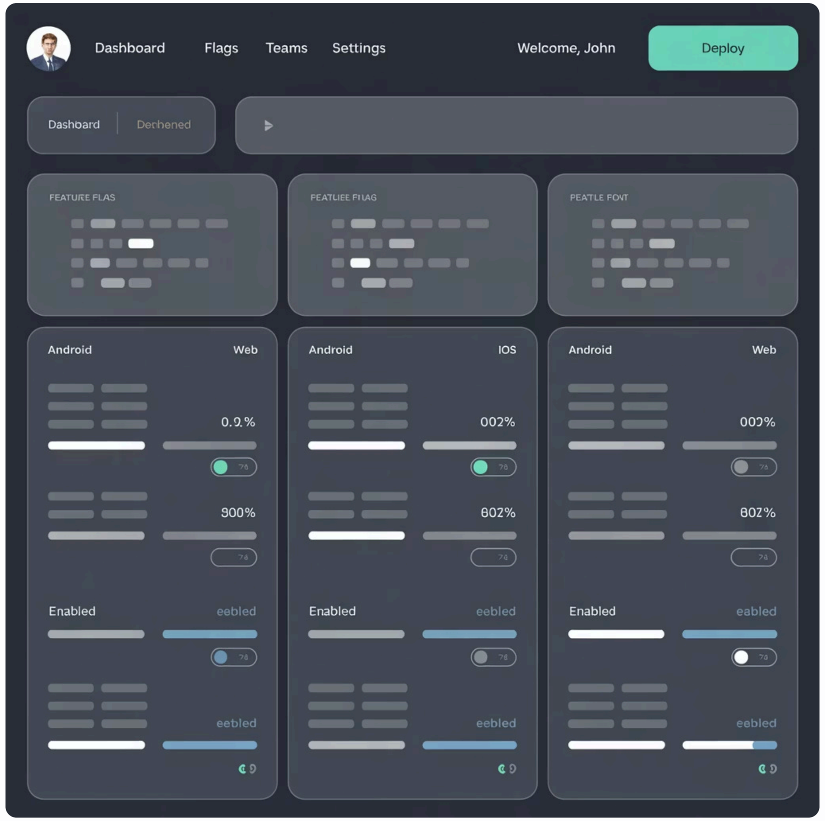
Concrete Benefits

- **Decouple Deployment from Release:** Deploy code at any time, then activate features when they're truly ready, independently.
- **Targeted Testing:** Test new features with specific groups of users (e.g., internal staff, beta testers) before a broader rollout.
- **Instant Kill Switch:** Rapidly disable problematic features in production without an emergency code rollback or new deployment.
- **Phased Rollouts:** Gradually introduce features to your entire user base, monitoring performance and feedback along the way.

Companies like Facebook deploy code daily but might take weeks to slowly enable new features using flags. This gives them very precise control over what users experience.

Real-World Example: Feature Flags

Case Study: Etsy's Feature Flag System



Etsy uses feature flags extensively to control its 250+ deployments per day:

- All new features start behind feature flags
- Features can be targeted to specific user segments
- Engineers can override flags for testing
- Gradual rollouts (1% → 5% → 25% → 100%)
- Automatic rollbacks if metrics degrade

This approach has allowed Etsy to maintain a rapid deployment pace while controlling risk and gathering user feedback early.

Rollback Strategies

Version Rollback

Redeploy the previous known-good version of the application

Database Rollback

Revert database changes using migrations or backups

Feature Flag Disable

Turn off problematic features via configuration

Traffic Shifting

Redirect traffic back to previous infrastructure

Every deployment should have a clearly defined, tested rollback plan. The best time to figure out how to roll back is **before** you need to do it.

Deployment Safety Practices

Before Deployment

- Run pre-deployment checklists
- Verify test coverage
- Review and approve changes
- Check dependency compatibility
- Schedule during low-traffic periods

During Deployment

- Monitor key metrics in real-time
- Implement automatic verification steps
- Have rollback mechanisms ready
- Maintain communication channels
- Use progressive deployment patterns

After Deployment

Run post-deployment verification tests, check for unexpected errors in logs, and confirm all systems are operating normally. Document lessons learned in deployment retrospective.

Integrating Testing into Pipelines

A comprehensive testing strategy integrates different types of tests at different stages of the pipeline:

Fast Feedback Loop

Unit tests run first and provide immediate feedback to developers

Integration Verification

Integration tests confirm components work together correctly

Quality Assurance

End-to-end tests verify complete user workflows

Security Validation

Security scans identify vulnerabilities before production

Real-World Example: Testing Pipeline

Case Study: Amazon's Deployment Testing

1 Code Commit

Developers save their changes

Automated checks for code quality start right away

2 Build Phase

Small, fast tests check individual code parts (all must pass)

We track how much code is tested (must meet our goals)

3 Integration Phase

Tests check if different parts of the system work together

Tests make sure performance hasn't gotten worse

4 Pre-Production

New code is tested with a small group of fake users

We check for security problems and make sure rules are followed

5 Production

Quick automated tests confirm the new code works after it's live

We watch for anything unusual happening in real-time

Measuring Automation Success

75%

Reduction in Toil

Track percentage decrease in time spent on manual tasks

4x

Deployment Frequency

Measure increase in deployment rate

90%

Automation Coverage

Percentage of operational tasks with automation

-60%

Mean Time to Recovery

Reduction in time to resolve incidents

Set clear metrics to track your automation journey. High-performing organizations deploy 208x more frequently and recover from incidents 2,604x faster than low performers (DevOps Research and Assessment).

Implementation Roadmap

1

Month 1: Assessment

Measure current toil, identify high-impact automation targets, and document runbooks for common procedures

2

Month 2-3: Foundation

Implement basic CI/CD pipeline, create initial scripts for common tasks, build monitoring for automated processes

3

Month 4-6: Expansion

Implement feature flags, canary deployments, expand test automation, create self-service tools for common requests

4

Month 7-12: Maturity

Implement auto-remediation, establish deployment safety metrics, create feedback loops for continuous improvement

Key Takeaways

Start with Identification

Use structured approaches to identify and prioritize toil in your organization

Automate Strategically

Follow the elimination → simplification → standardization → automation hierarchy

Build Safety

Implement progressive deployment strategies with robust testing and rollback capabilities

Measure Impact

Track reduction in toil and improvement in key delivery metrics

"Automation isn't about replacing humans; it's about amplifying their impact by freeing them from mundane tasks to focus on creative, high-value work."