IDEO-Lab

DevOps Guide

Terraform - GitLab CI/CD - Production Incidents - SRE Runbooks

English edition	Based on the 3-part IDEO-Lab DevOps HTML guide
Audience	Junior to senior DevOps engineers, platform teams, SREs and cloud engineers
Objective	Provide practical procedures, mental models, production checklists and incident discipline

This document is designed as a practical field guide: concise enough for onboarding, detailed enough for production reviews, and structured enough to be used as a runbook companion.

Preface

This English edition consolidates the IDEO-Lab DevOps material into a printable reference manual. It keeps the spirit of the original guide: visual thinking, practical examples, tables, operational procedures and a strong focus on production reality.

The guide is not a theoretical encyclopedia. It is meant to help teams build a reliable DevOps culture around infrastructure as code, GitLab CI/CD, state management, incident response, security and continuous improvement.

How to read this guide

- Use the Terraform chapters to understand how infrastructure changes move safely from code to production.

- Use the GitLab chapters to design predictable, auditable and protected delivery pipelines.

- Use the incident chapters to practice detection, qualification, mitigation, rollback, restoration and post-mortems.

- Use the annexes as templates for risk scoring, go/no-go meetings, runbooks and training paths.

Operating principles

Principle	Meaning in production
Everything critical should be traceable	A reviewer must understand who changed what, why, when and how it was validated.
Automation is not a substitute for discipline	CI/CD must enforce checks, approvals, environment separation and rollback readiness.
Observe before acting	Most incidents become worse when teams restart or apply changes before reading facts.
Small reversible changes win	The best production change is understandable, reviewed and recoverable.

Document Map

The document follows the same major journey as the original 3-part guide: foundations, advanced delivery, and production operations.

Part	Focus	Main outcomes
Part 1	Foundations	Terraform basics, state, GitLab CI/CD, environments, secrets, deployment, observability and incidents.
Part 2	Operational maturity	Runbooks, roadmap, cheat sheet, Terraform design, import/drift, testing, FinOps, MR practices, templates and approvals.
Part 3	Production excellence	Pipeline debugging, deployment strategies, capacity, DB/network incidents, IAM, supply chain, Kubernetes, Linux hardening, training, glossary and annexes.

Target reader

- A new DevOps engineer learning how infrastructure really behaves in production.

- A developer moving toward platform or SRE responsibilities.

- A team lead standardizing Terraform, GitLab and incident practices.

- A candidate preparing for infrastructure DevOps interviews.

Table of Contents

1.1 - DevOps Infrastructure Overview .... page 6

1.2 - Terraform Fundamentals .... page 8

1.3 - Terraform Modules .... page 10

1.4 - Terraform State .... page 12

2.1 - GitLab CI/CD for Infrastructure .... page 14

2.2 - Environments .... page 16

2.3 - Secrets and CI Security .... page 18

2.4 - Infrastructure Deployment .... page 20

3.1 - Observability .... page 22

3.2 - Production Incidents .... page 24

3.3 - DevOps Runbooks .... page 26

3.4 - DevOps Roadmap .... page 28

4.1 - DevOps Cheat Sheet .... page 30

1.5 - Advanced Terraform Design .... page 32

1.6 - Import and Drift .... page 34

1.7 - Terraform Testing .... page 36

1.8 - Cost and FinOps .... page 38

2.5 - Infrastructure Merge Requests .... page 40

2.6 - GitLab CI Templates .... page 42

2.7 - Approvals and Protections .... page 44

2.8 - Pipeline Debugging .... page 46

3.5 - Deployment Strategies .... page 48

3.6 - Capacity and Performance .... page 50

3.7 - Database Incidents .... page 52

3.8 - Network Incidents .... page 54

4.4 - IAM Deep Dive .... page 56

4.5 - CI/CD Supply Chain .... page 58

4.6 - Kubernetes for DevOps .... page 60

4.7 - Linux Hardening .... page 62

5.5 - Training New DevOps Engineers .... page 64

5.6 - Advanced Glossary .... page 66

6.1 - Practical Annexes .... page 68

End-to-End DevOps Flow

```
Developer / Platform Engineer
    |
    v
Git branch -> Merge Request -> CI checks -> Terraform plan artifact
    |              |              |              |
    |              |              |              +-- cost, security and drift signals
    |              |              +-- fmt, validate, lint, test
    |              +-- human review and approval
    v
Protected apply -> Post-checks -> Observability -> Incident readiness
    |
    +-- if failure: rollback, mitigation, RCA, runbook update
```

Core review gates

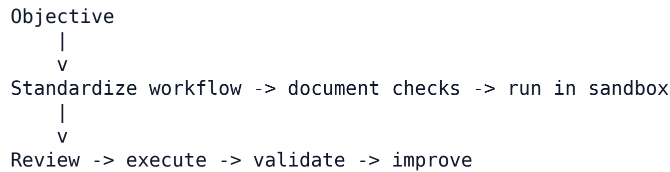
Gate	Question	Expected evidence
Plan	What will change?	Create/update/delete summary, no unexpected destroy.
Security	Are permissions and secrets safe?	Protected variables, least privilege, no exposed secrets.
Operations	Can we recover?	Rollback steps, monitoring, owner and post-checks.
Cost	Will this change spend money?	Infracost or cloud estimate, tags and budget owner.
Communication	Who needs to know?	Change note, go/no-go decision, incident channel if needed.

1.1 - DevOps Infrastructure Overview

Part 1 - English operational chapter

DevOps Infrastructure Overview is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. The infrastructure DevOps role: automate, make reliable, trace and secure production platforms. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Git	Supports devops infrastructure overview by adding automation, evidence, traceability or control.
CI/CD	Supports devops infrastructure overview by adding automation, evidence, traceability or control.
Monitoring	Supports devops infrastructure overview by adding automation, evidence, traceability or control.
Runbooks	Supports devops infrastructure overview by adding automation, evidence, traceability or control.
Change tickets	Supports devops infrastructure overview by adding automation, evidence, traceability or control.

1.1 - DevOps Infrastructure Overview: Production Use

Production example

A team uses devops infrastructure overview to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

1.2 - Terraform Fundamentals

Part 1 - English operational chapter

Terraform Fundamentals is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Providers, resources, variables, outputs, plan/apply and the Terraform lifecycle. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

What good looks like

- Keep modules small, versioned and documented with clear inputs and outputs.

- Require terraform fmt, validate and plan in CI before any review.

- Never apply production from a laptop when a protected pipeline exists.

- Pin provider versions and keep upgrade windows explicit.

Tools commonly involved

Tool / area	How it helps
Terraform CLI	Supports terraform fundamentals by adding automation, evidence, traceability or control.
tflint	Supports terraform fundamentals by adding automation, evidence, traceability or control.
tfsec/checkov	Supports terraform fundamentals by adding automation, evidence, traceability or control.
Terratest	Supports terraform fundamentals by adding automation, evidence, traceability or control.
Atlantis/GitLab	Supports terraform fundamentals by adding automation, evidence, traceability or control.

1.2 - Terraform Fundamentals: Production Use

Production example

A platform team introduces a new load balancer module. The MR shows the module version, variables, outputs and the exact plan. Reviewers verify no unrelated resource will be destroyed, the security group is not public by mistake, and the module can be reused by staging and production.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
terraform fmt -check
terraform validate
terraform plan -out=tfplan
terraform show -no-color tfplan
terraform apply tfplan
```

Anti-patterns

- Using one huge root module for every environment.

- Changing provider versions and functional resources in the same MR.

- Using count where stable for_each keys are required.

- Relying on implicit dependencies that reviewers cannot see.

1.3 - Terraform Modules

Part 1 - English operational chapter

Terraform Modules is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Split network, compute, security, database and application components cleanly. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

What good looks like

- Keep modules small, versioned and documented with clear inputs and outputs.

- Require terraform fmt, validate and plan in CI before any review.

- Never apply production from a laptop when a protected pipeline exists.

- Pin provider versions and keep upgrade windows explicit.

Tools commonly involved

Tool / area	How it helps
Terraform CLI	Supports terraform modules by adding automation, evidence, traceability or control.
tflint	Supports terraform modules by adding automation, evidence, traceability or control.
tfsec/checkov	Supports terraform modules by adding automation, evidence, traceability or control.
Terratest	Supports terraform modules by adding automation, evidence, traceability or control.
Atlantis/GitLab	Supports terraform modules by adding automation, evidence, traceability or control.

1.3 - Terraform Modules: Production Use

Production example

A platform team introduces a new load balancer module. The MR shows the module version, variables, outputs and the exact plan. Reviewers verify no unrelated resource will be destroyed, the security group is not public by mistake, and the module can be reused by staging and production.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
terraform fmt -check
terraform validate
terraform plan -out=tfplan
terraform show -no-color tfplan
terraform apply tfplan
```

Anti-patterns

- Using one huge root module for every environment.

- Changing provider versions and functional resources in the same MR.

- Using count where stable for_each keys are required.

- Relying on implicit dependencies that reviewers cannot see.

1.4 - Terraform State

Part 1 - English operational chapter

Terraform State is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Remote backend, locking, drift, environment isolation and state security. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

```
Change request
     |
     v
Code update -> fmt/validate -> plan
     |
     v
Review plan -> security/cost checks -> approval
     |
     v
Apply -> post-checks -> drift watch
```

What good looks like

- Use one remote backend per environment and enable locking.

- Treat state as sensitive because it can contain IDs, topology and sometimes secrets.

- Never edit state manually without a backup and a peer review.

- Run drift checks on a schedule and before large production changes.

Tools commonly involved

Tool / area	How it helps
S3/GCS/Azure backend	Supports terraform state by adding automation, evidence, traceability or control.
DynamoDB or backend lock	Supports terraform state by adding automation, evidence, traceability or control.
terraform state	Supports terraform state by adding automation, evidence, traceability or control.
driftctl/checkov	Supports terraform state by adding automation, evidence, traceability or control.

1.4 - Terraform State: Production Use

Production example

An existing production database was created manually months ago. The team imports it into Terraform state, runs a refresh-only plan, documents drift, and only then starts managing safe attributes from code.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
terraform state list
terraform state show <resource>
terraform import <address> <id>
terraform plan -refresh-only
terraform force-unlock <lock-id>  # only after proof
```

Anti-patterns

- Local state for shared infrastructure.

- Manual console changes with no drift follow-up.

- Force-unlocking without proving no apply is running.

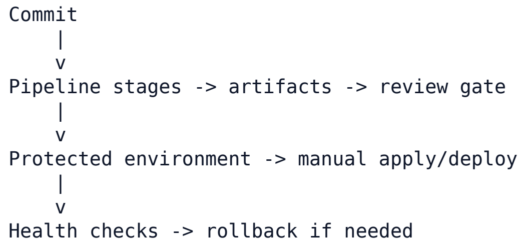
- Committing state files or plan files with sensitive values.

2.1 - GitLab CI/CD for Infrastructure

Part 1 - English operational chapter

GitLab CI/CD for Infrastructure is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. fmt, validate, plan, review, apply and controlled rollback stages. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Commit
    |
    v
Pipeline stages -> artifacts -> review gate
    |
    v
Protected environment -> manual apply/deploy
    |
    v
Health checks -> rollback if needed
```

What good looks like

- Separate fmt, validate, plan, review and apply stages.

- Keep production apply manual and protected with approvals.

- Publish the plan as an artifact and review it before approval.

- Use templates and extends to reduce duplication across repositories.

Tools commonly involved

Tool / area	How it helps
GitLab CI	Supports gitlab ci/cd for infrastructure by adding automation, evidence, traceability or control.
GitLab Runners	Supports gitlab ci/cd for infrastructure by adding automation, evidence, traceability or control.
Protected variables	Supports gitlab ci/cd for infrastructure by adding automation, evidence, traceability or control.
Artifacts	Supports gitlab ci/cd for infrastructure by adding automation, evidence, traceability or control.
Review Apps	Supports gitlab ci/cd for infrastructure by adding automation, evidence, traceability or control.

2.1 - GitLab CI/CD for Infrastructure: Production Use

Production example

A GitLab pipeline fails in the plan stage. Instead of retrying blindly, the engineer checks runner tags, protected variables, the working directory and the backend credentials. The root cause is a production variable not available from an unprotected branch.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
gitlab-runner verify
printenv | sort | sed 's/=.*$/=<redacted>/'
ls -la $CI_PROJECT_DIR
cat .gitlab-ci.yml
terraform plan -out=tfplan
```

Anti-patterns

- Copy-pasting pipeline jobs across repositories.

- Allowing production jobs from unprotected branches.

- Not storing the plan as an artifact.

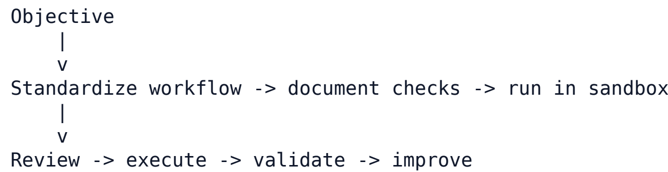
- Using runners with excessive privileges for normal jobs.

2.2 - Environments

Part 1 - English operational chapter

Environments is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Dev, staging and production with protected variables, branches, approvals and separation. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Git	Supports environments by adding automation, evidence, traceability or control.
CI/CD	Supports environments by adding automation, evidence, traceability or control.
Monitoring	Supports environments by adding automation, evidence, traceability or control.
Runbooks	Supports environments by adding automation, evidence, traceability or control.
Change tickets	Supports environments by adding automation, evidence, traceability or control.

2.2 - Environments: Production Use

Production example

A team uses environments to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

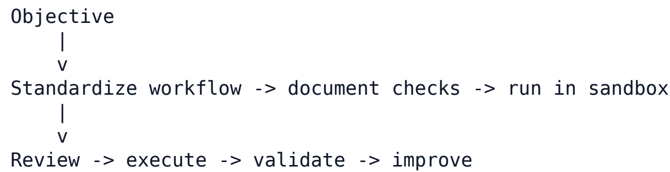
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

2.3 - Secrets and CI Security

Part 1 - English operational chapter

Secrets and CI Security is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Masked variables, scopes, short-lived credentials and least-privilege policies. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Use least privilege and short-lived credentials wherever possible.

- Keep production variables protected, masked and environment-scoped.

- Prefer OIDC or assume-role flows over long-lived static keys.

- Log privileged actions and define break-glass access explicitly.

Tools commonly involved

Tool / area	How it helps
Vault	Supports secrets and ci security by adding automation, evidence, traceability or control.
Cloud IAM	Supports secrets and ci security by adding automation, evidence, traceability or control.
OIDC	Supports secrets and ci security by adding automation, evidence, traceability or control.
Secrets Manager	Supports secrets and ci security by adding automation, evidence, traceability or control.
Audit logs	Supports secrets and ci security by adding automation, evidence, traceability or control.

2.3 - Secrets and CI Security: Production Use

Production example

A CI token is suspected to be exposed in logs. The team treats it as compromised, revokes it, rotates credentials, checks audit logs and adds a guard preventing secret echo in future jobs.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
aws sts get-caller-identity
aws iam simulate-principal-policy ...
trivy image myapp:tag
cosign verify image
printenv | grep -E 'TOKEN|KEY|SECRET' # never print values
```

Anti-patterns

- Long-lived cloud keys in CI variables.

- Wildcard IAM policies without justification.

- Secrets printed in logs or stored in artifacts.

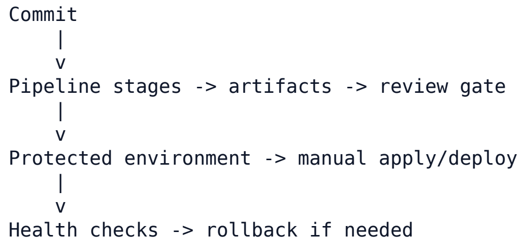
- Break-glass access with no audit trail.

2.4 - Infrastructure Deployment

Part 1 - English operational chapter

Infrastructure Deployment is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. A clean workflow: change request, review, plan, production window, apply and verification. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Make deployment units small, observable and reversible.

- Define pre-checks and post-checks before the production window.

- Keep rollback steps visible in the merge request or change ticket.

- Use canary or blue/green for risky user-facing changes.

Tools commonly involved

Tool / area	How it helps
GitLab Environments	Supports infrastructure deployment by adding automation, evidence, traceability or control.
Helm	Supports infrastructure deployment by adding automation, evidence, traceability or control.
Nginx/LB	Supports infrastructure deployment by adding automation, evidence, traceability or control.
Feature flags	Supports infrastructure deployment by adding automation, evidence, traceability or control.
Rollback plan	Supports infrastructure deployment by adding automation, evidence, traceability or control.

2.4 - Infrastructure Deployment: Production Use

Production example

A team uses infrastructure deployment to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

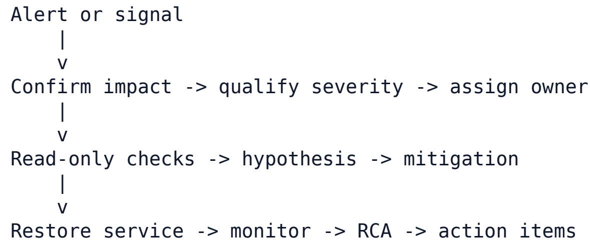
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.1 - Observability

Part 1 - English operational chapter

Observability is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Logs, metrics, alerts, traces and dashboards to operate production confidently. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Dashboards must exist before the incident, not after it.

- Alert on user impact and saturation, not only on host availability.

- Link runbooks directly from alerts.

- Correlate deployments, logs, metrics and traces in one timeline.

Tools commonly involved

Tool / area	How it helps
Prometheus	Supports observability by adding automation, evidence, traceability or control.
Grafana	Supports observability by adding automation, evidence, traceability or control.
ELK/OpenSearch	Supports observability by adding automation, evidence, traceability or control.
Jaeger	Supports observability by adding automation, evidence, traceability or control.
Alertmanager	Supports observability by adding automation, evidence, traceability or control.

3.1 - Observability: Production Use

Production example

A team uses observability to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
kubectl get events --sort-by=.lastTimestamp
kubectl logs deploy/app --tail=200
curl -s /metrics | head
journalctl -p warning -n 100
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

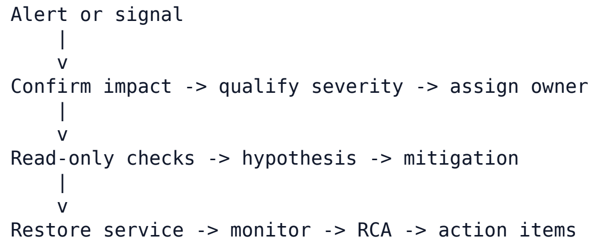
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.2 - Production Incidents

Part 1 - English operational chapter

Production Incidents is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Qualify, mitigate, restore, communicate, analyze and prevent recurrence. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Confirm the impact before changing production.

- Assign an incident commander for severe incidents.

- Mitigate before searching for a perfect root cause.

- Capture a timeline and convert it into action items.

Tools commonly involved

Tool / area	How it helps
PagerDuty/Opsgenie	Supports production incidents by adding automation, evidence, traceability or control.
Grafana	Supports production incidents by adding automation, evidence, traceability or control.
Kibana	Supports production incidents by adding automation, evidence, traceability or control.
Status page	Supports production incidents by adding automation, evidence, traceability or control.
Post-mortem template	Supports production incidents by adding automation, evidence, traceability or control.

3.2 - Production Incidents: Production Use

Production example

The API error rate jumps from 0.3% to 12%. The team opens a war room, assigns an incident commander, rolls back the last release, watches latency and 5xx metrics, then documents a post-mortem with action items.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

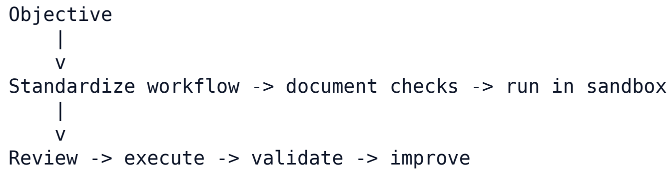
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.3 - DevOps Runbooks

Part 2 - English operational chapter

DevOps Runbooks is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Clear procedures for rollback, restart, saturation, database, network and blocked pipelines. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Confluence/Notion	Supports devops runbooks by adding automation, evidence, traceability or control.
Git runbooks	Supports devops runbooks by adding automation, evidence, traceability or control.
Grafana links	Supports devops runbooks by adding automation, evidence, traceability or control.
Shell snippets	Supports devops runbooks by adding automation, evidence, traceability or control.
Incident bot	Supports devops runbooks by adding automation, evidence, traceability or control.

3.3 - DevOps Runbooks: Production Use

Production example

A team uses devops runbooks to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

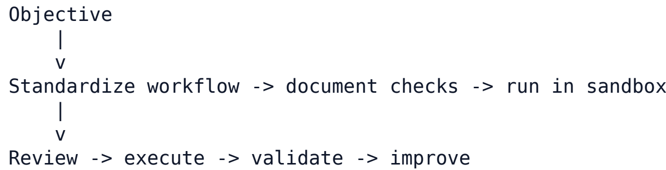
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.4 - DevOps Roadmap

Part 2 - English operational chapter

DevOps Roadmap is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. A pragmatic roadmap for a DevOps team moving from manual operations to platform engineering. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Git	Supports devops roadmap by adding automation, evidence, traceability or control.
CI/CD	Supports devops roadmap by adding automation, evidence, traceability or control.
Monitoring	Supports devops roadmap by adding automation, evidence, traceability or control.
Runbooks	Supports devops roadmap by adding automation, evidence, traceability or control.
Change tickets	Supports devops roadmap by adding automation, evidence, traceability or control.

3.4 - DevOps Roadmap: Production Use

Production example

A team uses devops roadmap to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

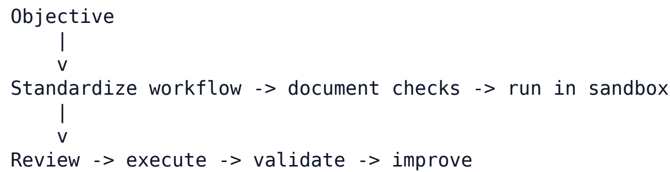
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

4.1 - DevOps Cheat Sheet

Part 2 - English operational chapter

DevOps Cheat Sheet is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Terraform, GitLab, pipeline debugging and production verification commands. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Objective
    |
    v
Standardize workflow -> document checks -> run in sandbox
    |
    v
Review -> execute -> validate -> improve
```

What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Git	Supports devops cheat sheet by adding automation, evidence, traceability or control.
CI/CD	Supports devops cheat sheet by adding automation, evidence, traceability or control.
Monitoring	Supports devops cheat sheet by adding automation, evidence, traceability or control.
Runbooks	Supports devops cheat sheet by adding automation, evidence, traceability or control.
Change tickets	Supports devops cheat sheet by adding automation, evidence, traceability or control.

4.1 - DevOps Cheat Sheet: Production Use

Production example

A team uses devops cheat sheet to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

1.5 - Advanced Terraform Design

Part 2 - English operational chapter

Advanced Terraform Design is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Naming, locals, validations, count/for_each, lifecycle and dependencies. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

What good looks like

- Keep modules small, versioned and documented with clear inputs and outputs.

- Require terraform fmt, validate and plan in CI before any review.

- Never apply production from a laptop when a protected pipeline exists.

- Pin provider versions and keep upgrade windows explicit.

Tools commonly involved

Tool / area	How it helps
Terraform CLI	Supports advanced terraform design by adding automation, evidence, traceability or control.
tflint	Supports advanced terraform design by adding automation, evidence, traceability or control.
tfsec/checkov	Supports advanced terraform design by adding automation, evidence, traceability or control.
Terratest	Supports advanced terraform design by adding automation, evidence, traceability or control.
Atlantis/GitLab	Supports advanced terraform design by adding automation, evidence, traceability or control.

1.5 - Advanced Terraform Design: Production Use

Production example

A platform team introduces a new load balancer module. The MR shows the module version, variables, outputs and the exact plan. Reviewers verify no unrelated resource will be destroyed, the security group is not public by mistake, and the module can be reused by staging and production.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
terraform fmt -check
terraform validate
terraform plan -out=tfplan
terraform show -no-color tfplan
terraform apply tfplan
```

Anti-patterns

- Using one huge root module for every environment.

- Changing provider versions and functional resources in the same MR.

- Using count where stable for_each keys are required.

- Relying on implicit dependencies that reviewers cannot see.

1.6 - Import and Drift

Part 2 - English operational chapter

Import and Drift is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Attach existing resources, detect drift and bring infrastructure back under IaC control. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

What good looks like

- Use one remote backend per environment and enable locking.

- Treat state as sensitive because it can contain IDs, topology and sometimes secrets.

- Never edit state manually without a backup and a peer review.

- Run drift checks on a schedule and before large production changes.

Tools commonly involved

Tool / area	How it helps
S3/GCS/Azure backend	Supports import and drift by adding automation, evidence, traceability or control.
DynamoDB or backend lock	Supports import and drift by adding automation, evidence, traceability or control.
terraform state	Supports import and drift by adding automation, evidence, traceability or control.
driftctl/checkov	Supports import and drift by adding automation, evidence, traceability or control.

1.6 - Import and Drift: Production Use

Production example

An existing production database was created manually months ago. The team imports it into Terraform state, runs a refresh-only plan, documents drift, and only then starts managing safe attributes from code.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
terraform state list
terraform state show <resource>
terraform import <address> <id>
terraform plan -refresh-only
terraform force-unlock <lock-id>  # only after proof
```

Anti-patterns

- Local state for shared infrastructure.

- Manual console changes with no drift follow-up.

- Force-unlocking without proving no apply is running.

- Committing state files or plan files with sensitive values.

1.7 - Terraform Testing

Part 2 - English operational chapter

Terraform Testing is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Local validation, test plans, sandbox, Terratest and module tests. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

```
Change request
     |
     v
Code update -> fmt/validate -> plan
     |
     v
Review plan -> security/cost checks -> approval
     |
     v
Apply -> post-checks -> drift watch
```

What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
terraform validate	Supports terraform testing by adding automation, evidence, traceability or control.
Terratest	Supports terraform testing by adding automation, evidence, traceability or control.
sandbox	Supports terraform testing by adding automation, evidence, traceability or control.
policy tests	Supports terraform testing by adding automation, evidence, traceability or control.
mock providers	Supports terraform testing by adding automation, evidence, traceability or control.

1.7 - Terraform Testing: Production Use

Production example

A team uses terraform testing to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

1.8 - Cost and FinOps

Part 2 - English operational chapter

Cost and FinOps is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Understand infrastructure cost impact before apply, using tags and cost gates. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model

What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
Infracost	Supports cost and finops by adding automation, evidence, traceability or control.
cloud tags	Supports cost and finops by adding automation, evidence, traceability or control.
budgets	Supports cost and finops by adding automation, evidence, traceability or control.
showback	Supports cost and finops by adding automation, evidence, traceability or control.
rightsizing	Supports cost and finops by adding automation, evidence, traceability or control.

1.8 - Cost and FinOps: Production Use

Production example

A team uses cost and finops to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

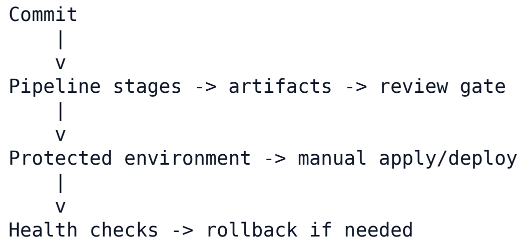
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

2.5 - Infrastructure Merge Requests

Part 2 - English operational chapter

Infrastructure Merge Requests is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Document a Terraform MR so it can be read, reviewed and approved safely. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Separate fmt, validate, plan, review and apply stages.

- Keep production apply manual and protected with approvals.

- Publish the plan as an artifact and review it before approval.

- Use templates and extends to reduce duplication across repositories.

Tools commonly involved

Tool / area	How it helps
GitLab CI	Supports infrastructure merge requests by adding automation, evidence, traceability or control.
GitLab Runners	Supports infrastructure merge requests by adding automation, evidence, traceability or control.
Protected variables	Supports infrastructure merge requests by adding automation, evidence, traceability or control.
Artifacts	Supports infrastructure merge requests by adding automation, evidence, traceability or control.
Review Apps	Supports infrastructure merge requests by adding automation, evidence, traceability or control.

2.5 - Infrastructure Merge Requests: Production Use

Production example

A GitLab pipeline fails in the plan stage. Instead of retrying blindly, the engineer checks runner tags, protected variables, the working directory and the backend credentials. The root cause is a production variable not available from an unprotected branch.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
gitlab-runner verify
printenv | sort | sed 's/=.*$/=<redacted>/'
ls -la $CI_PROJECT_DIR
cat .gitlab-ci.yml
terraform plan -out=tfplan
```

Anti-patterns

- Copy-pasting pipeline jobs across repositories.

- Allowing production jobs from unprotected branches.

- Not storing the plan as an artifact.

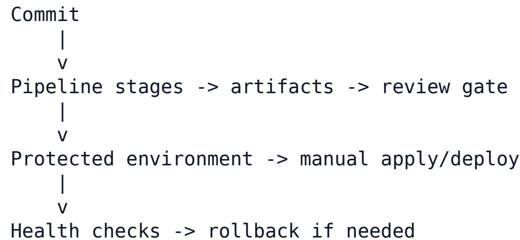
- Using runners with excessive privileges for normal jobs.

2.6 - GitLab CI Templates

Part 2 - English operational chapter

GitLab CI Templates is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Factorize CI/CD jobs to avoid duplication and standardize team workflows. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Separate fmt, validate, plan, review and apply stages.

- Keep production apply manual and protected with approvals.

- Publish the plan as an artifact and review it before approval.

- Use templates and extends to reduce duplication across repositories.

Tools commonly involved

Tool / area	How it helps
GitLab CI	Supports gitlab ci templates by adding automation, evidence, traceability or control.
GitLab Runners	Supports gitlab ci templates by adding automation, evidence, traceability or control.
Protected variables	Supports gitlab ci templates by adding automation, evidence, traceability or control.
Artifacts	Supports gitlab ci templates by adding automation, evidence, traceability or control.
Review Apps	Supports gitlab ci templates by adding automation, evidence, traceability or control.

2.6 - GitLab CI Templates: Production Use

Production example

A GitLab pipeline fails in the plan stage. Instead of retrying blindly, the engineer checks runner tags, protected variables, the working directory and the backend credentials. The root cause is a production variable not available from an unprotected branch.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
gitlab-runner verify
printenv | sort | sed 's/=.*$/=<redacted>/'
ls -la $CI_PROJECT_DIR
cat .gitlab-ci.yml
terraform plan -out=tfplan
```

Anti-patterns

- Copy-pasting pipeline jobs across repositories.

- Allowing production jobs from unprotected branches.

- Not storing the plan as an artifact.

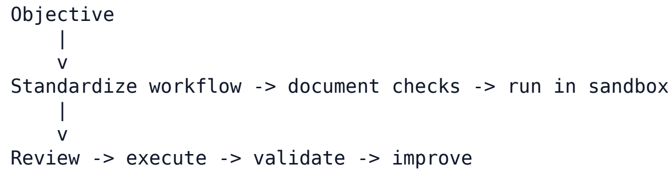
- Using runners with excessive privileges for normal jobs.

2.7 - Approvals and Protections

Part 2 - English operational chapter

Approvals and Protections is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Protected branches, protected environments, approvals and separation of duties. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Objective
    |
    v
Standardize workflow -> document checks -> run in sandbox
    |
    v
Review -> execute -> validate -> improve
```

What good looks like

- Use least privilege and short-lived credentials wherever possible.

- Keep production variables protected, masked and environment-scoped.

- Prefer OIDC or assume-role flows over long-lived static keys.

- Log privileged actions and define break-glass access explicitly.

Tools commonly involved

Tool / area	How it helps
Vault	Supports approvals and protections by adding automation, evidence, traceability or control.
Cloud IAM	Supports approvals and protections by adding automation, evidence, traceability or control.
OIDC	Supports approvals and protections by adding automation, evidence, traceability or control.
Secrets Manager	Supports approvals and protections by adding automation, evidence, traceability or control.
Audit logs	Supports approvals and protections by adding automation, evidence, traceability or control.

2.7 - Approvals and Protections: Production Use

Production example

A CI token is suspected to be exposed in logs. The team treats it as compromised, revokes it, rotates credentials, checks audit logs and adds a guard preventing secret echo in future jobs.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
aws sts get-caller-identity
aws iam simulate-principal-policy ...
trivy image myapp:tag
cosign verify image
printenv | grep -E 'TOKEN|KEY|SECRET' # never print values
```

Anti-patterns

- Long-lived cloud keys in CI variables.

- Wildcard IAM policies without justification.

- Secrets printed in logs or stored in artifacts.

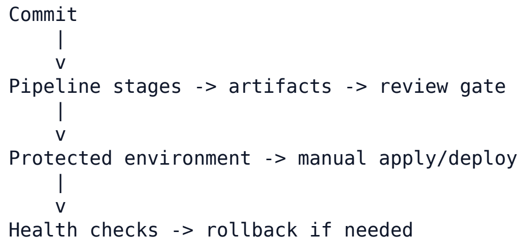
- Break-glass access with no audit trail.

2.8 - Pipeline Debugging

Part 3 - English operational chapter

Pipeline Debugging is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Diagnose a GitLab pipeline that is blocked, failing or behaving unexpectedly. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Separate fmt, validate, plan, review and apply stages.

- Keep production apply manual and protected with approvals.

- Publish the plan as an artifact and review it before approval.

- Use templates and extends to reduce duplication across repositories.

Tools commonly involved

Tool / area	How it helps
GitLab CI	Supports pipeline debugging by adding automation, evidence, traceability or control.
GitLab Runners	Supports pipeline debugging by adding automation, evidence, traceability or control.
Protected variables	Supports pipeline debugging by adding automation, evidence, traceability or control.
Artifacts	Supports pipeline debugging by adding automation, evidence, traceability or control.
Review Apps	Supports pipeline debugging by adding automation, evidence, traceability or control.

2.8 - Pipeline Debugging: Production Use

Production example

A GitLab pipeline fails in the plan stage. Instead of retrying blindly, the engineer checks runner tags, protected variables, the working directory and the backend credentials. The root cause is a production variable not available from an unprotected branch.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
gitlab-runner verify
printenv | sort | sed 's/=.*$/=<redacted>/'
ls -la $CI_PROJECT_DIR
cat .gitlab-ci.yml
terraform plan -out=tfplan
```

Anti-patterns

- Copy-pasting pipeline jobs across repositories.

- Allowing production jobs from unprotected branches.

- Not storing the plan as an artifact.

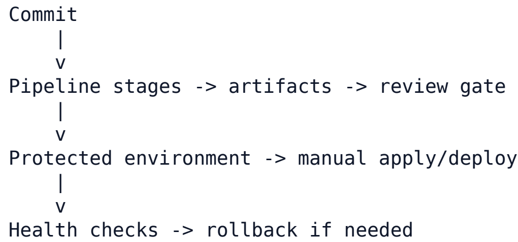
- Using runners with excessive privileges for normal jobs.

3.5 - Deployment Strategies

Part 3 - English operational chapter

Deployment Strategies is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Rolling, blue/green, canary, maintenance page and application rollback. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Make deployment units small, observable and reversible.

- Define pre-checks and post-checks before the production window.

- Keep rollback steps visible in the merge request or change ticket.

- Use canary or blue/green for risky user-facing changes.

Tools commonly involved

Tool / area	How it helps
GitLab Environments	Supports deployment strategies by adding automation, evidence, traceability or control.
Helm	Supports deployment strategies by adding automation, evidence, traceability or control.
Nginx/LB	Supports deployment strategies by adding automation, evidence, traceability or control.
Feature flags	Supports deployment strategies by adding automation, evidence, traceability or control.
Rollback plan	Supports deployment strategies by adding automation, evidence, traceability or control.

3.5 - Deployment Strategies: Production Use

Production example

A team uses deployment strategies to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

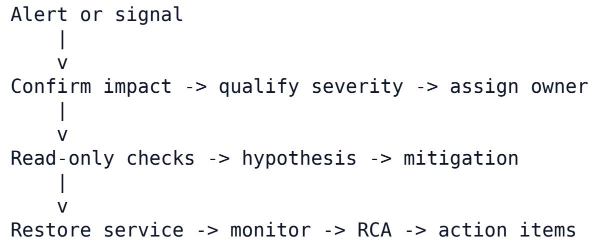
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.6 - Capacity and Performance

Part 3 - English operational chapter

Capacity and Performance is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. CPU, RAM, disk, connections, queues, saturation and alert thresholds. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
node exporter	Supports capacity and performance by adding automation, evidence, traceability or control.
APM	Supports capacity and performance by adding automation, evidence, traceability or control.
queue metrics	Supports capacity and performance by adding automation, evidence, traceability or control.
load test	Supports capacity and performance by adding automation, evidence, traceability or control.
capacity plan	Supports capacity and performance by adding automation, evidence, traceability or control.

3.6 - Capacity and Performance: Production Use

Production example

A team uses capacity and performance to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

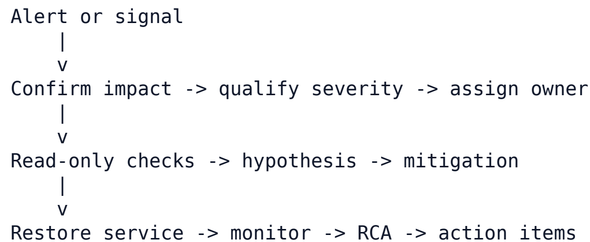
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.7 - Database Incidents

Part 3 - English operational chapter

Database Incidents is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Connections, locks, slowness, full disk, backups and restoration. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Alert or signal
    |
    v
Confirm impact -> qualify severity -> assign owner
    |
    v
Read-only checks -> hypothesis -> mitigation
    |
    v
Restore service -> monitor -> RCA -> action items
```

What good looks like

- Check connections, locks, slow queries and disk before restarting anything.

- Verify that backups are recent and restorable, not only present.

- For migrations, separate schema change, backfill and cleanup where possible.

- Have a clear escalation path to DBA/application owners.

Tools commonly involved

Tool / area	How it helps
psql/mysql	Supports database incidents by adding automation, evidence, traceability or control.
pg_stat_activity	Supports database incidents by adding automation, evidence, traceability or control.
slow query logs	Supports database incidents by adding automation, evidence, traceability or control.
backups	Supports database incidents by adding automation, evidence, traceability or control.
restore drills	Supports database incidents by adding automation, evidence, traceability or control.

3.7 - Database Incidents: Production Use

Production example

Checkout is slow because DB connections are exhausted. The team confirms active sessions and locks, mitigates by reducing application concurrency, then fixes pool sizing and adds a connection saturation alert.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
SHOW PROCESSLIST;
SHOW ENGINE INNODB STATUS;
SELECT * FROM pg_stat_activity;
SELECT now(), count(*) FROM pg_locks;
df -h
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

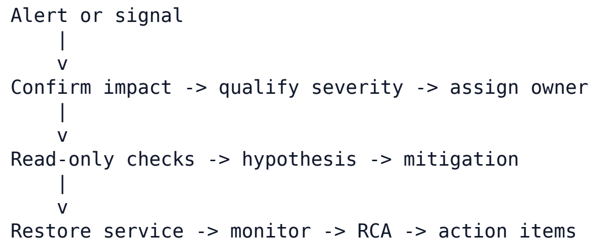
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

3.8 - Network Incidents

Part 3 - English operational chapter

Network Incidents is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. DNS, TLS, Nginx, firewall, load balancer, routes and ports. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Check DNS, TLS, routing, firewall and load balancer separately.

- Use read-only commands first: dig, curl, openssl, ss and logs.

- Keep certificate renewal and Nginx reload procedures tested.

- Document allowed ports and source ranges for each environment.

Tools commonly involved

Tool / area	How it helps
dig/nslookup	Supports network incidents by adding automation, evidence, traceability or control.
curl/openssl	Supports network incidents by adding automation, evidence, traceability or control.
Nginx	Supports network incidents by adding automation, evidence, traceability or control.
Firewall	Supports network incidents by adding automation, evidence, traceability or control.
Load balancer	Supports network incidents by adding automation, evidence, traceability or control.

3.8 - Network Incidents: Production Use

Production example

Users report TLS errors after a certificate renewal. The engineer verifies DNS, certificate chain, SNI, Nginx config and load balancer listeners before reloading the correct certificate.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
dig example.com
curl -Iv https://example.com
openssl s_client -connect example.com:443 -servername example.com
ss -tulpn
nginx -t
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

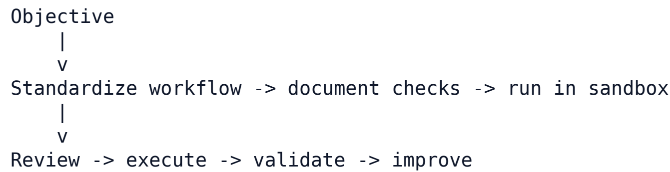
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

4.4 - IAM Deep Dive

Part 3 - English operational chapter

IAM Deep Dive is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Roles, policies, assume-role, separation of privileges and break-glass access. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Objective
    |
    v
Standardize workflow -> document checks -> run in sandbox
    |
    v
Review -> execute -> validate -> improve
```

What good looks like

- Use least privilege and short-lived credentials wherever possible.

- Keep production variables protected, masked and environment-scoped.

- Prefer OIDC or assume-role flows over long-lived static keys.

- Log privileged actions and define break-glass access explicitly.

Tools commonly involved

Tool / area	How it helps
Vault	Supports iam deep dive by adding automation, evidence, traceability or control.
Cloud IAM	Supports iam deep dive by adding automation, evidence, traceability or control.
OIDC	Supports iam deep dive by adding automation, evidence, traceability or control.
Secrets Manager	Supports iam deep dive by adding automation, evidence, traceability or control.
Audit logs	Supports iam deep dive by adding automation, evidence, traceability or control.

4.4 - IAM Deep Dive: Production Use

Production example

A CI token is suspected to be exposed in logs. The team treats it as compromised, revokes it, rotates credentials, checks audit logs and adds a guard preventing secret echo in future jobs.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
aws sts get-caller-identity
aws iam simulate-principal-policy ...
trivy image myapp:tag
cosign verify image
printenv | grep -E 'TOKEN|KEY|SECRET' # never print values
```

Anti-patterns

- Long-lived cloud keys in CI variables.

- Wildcard IAM policies without justification.

- Secrets printed in logs or stored in artifacts.

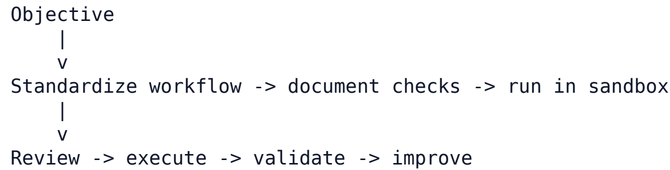
- Break-glass access with no audit trail.

4.5 - CI/CD Supply Chain

Part 3 - English operational chapter

CI/CD Supply Chain is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Docker images, dependencies, runners, artifacts and provenance. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
SBOM	Supports ci/cd supply chain by adding automation, evidence, traceability or control.
Trivy/Grype	Supports ci/cd supply chain by adding automation, evidence, traceability or control.
cosign	Supports ci/cd supply chain by adding automation, evidence, traceability or control.
SLSA	Supports ci/cd supply chain by adding automation, evidence, traceability or control.
immutable images	Supports ci/cd supply chain by adding automation, evidence, traceability or control.

4.5 - CI/CD Supply Chain: Production Use

Production example

A team uses ci/cd supply chain to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

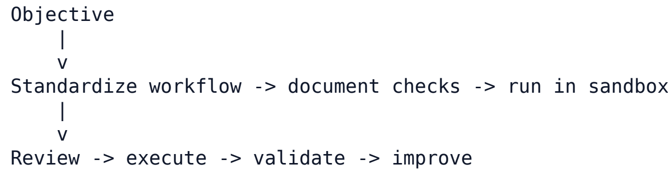
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

4.6 - Kubernetes for DevOps

Part 3 - English operational chapter

Kubernetes for DevOps is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Useful Kubernetes concepts even when the position is mainly Terraform/GitLab. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
kubectl	Supports kubernetes for devops by adding automation, evidence, traceability or control.
Helm	Supports kubernetes for devops by adding automation, evidence, traceability or control.
Ingress	Supports kubernetes for devops by adding automation, evidence, traceability or control.
HPA	Supports kubernetes for devops by adding automation, evidence, traceability or control.
Events	Supports kubernetes for devops by adding automation, evidence, traceability or control.

4.6 - Kubernetes for DevOps: Production Use

Production example

A team uses kubernetes for devops to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
kubectl get pods -A
kubectl describe pod <pod>
kubectl logs <pod> --tail=200
kubectl rollout status deploy/<name>
helm history <release>
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

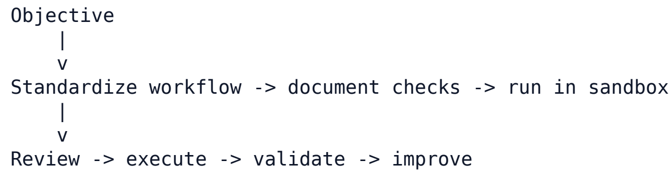
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

4.7 - Linux Hardening

Part 3 - English operational chapter

Linux Hardening is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. systemd, users, SSH, firewall, logs, permissions and updates. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Objective
    |
    v
Standardize workflow -> document checks -> run in sandbox
    |
    v
Review -> execute -> validate -> improve
```

What good looks like

- Harden SSH, disable unnecessary services and patch regularly.

- Use systemd and journalctl for service visibility.

- Keep firewall policy explicit and versioned when possible.

- Audit sudo, users, keys and file permissions.

Tools commonly involved

Tool / area	How it helps
systemd	Supports linux hardening by adding automation, evidence, traceability or control.
journalctl	Supports linux hardening by adding automation, evidence, traceability or control.
ss/lsof	Supports linux hardening by adding automation, evidence, traceability or control.
nftables/ufw	Supports linux hardening by adding automation, evidence, traceability or control.
auditd	Supports linux hardening by adding automation, evidence, traceability or control.

4.7 - Linux Hardening: Production Use

Production example

A team uses linux hardening to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
systemctl status app --no-pager
journalctl -u app -n 100 --no-pager
ss -tulpn
df -h
free -m
sudo ufw status verbose
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

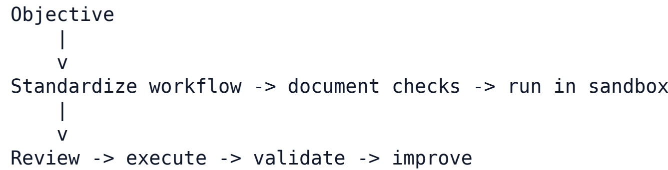
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

5.5 - Training New DevOps Engineers

Part 3 - English operational chapter

Training New DevOps Engineers is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Progression, exercises, deliberate mistakes, reviews and team rituals. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



```
Objective
    |
    v
Standardize workflow -> document checks -> run in sandbox
    |
    v
Review -> execute -> validate -> improve
```

What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
labs	Supports training new devops engineers by adding automation, evidence, traceability or control.
sandbox	Supports training new devops engineers by adding automation, evidence, traceability or control.
game days	Supports training new devops engineers by adding automation, evidence, traceability or control.
review checklist	Supports training new devops engineers by adding automation, evidence, traceability or control.
pairing	Supports training new devops engineers by adding automation, evidence, traceability or control.

5.5 - Training New DevOps Engineers: Production Use

Production example

A team uses training new devops engineers to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

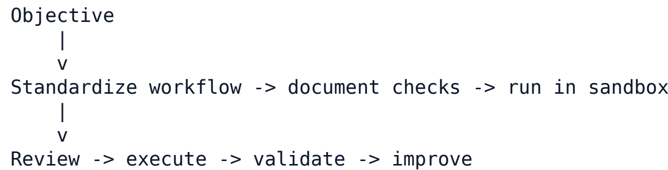
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

5.6 - Advanced Glossary

Part 3 - English operational chapter

Advanced Glossary is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Terms that new DevOps engineers encounter every day. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
team wiki	Supports advanced glossary by adding automation, evidence, traceability or control.
runbooks	Supports advanced glossary by adding automation, evidence, traceability or control.
onboarding	Supports advanced glossary by adding automation, evidence, traceability or control.
architecture maps	Supports advanced glossary by adding automation, evidence, traceability or control.
incident examples	Supports advanced glossary by adding automation, evidence, traceability or control.

5.6 - Advanced Glossary: Production Use

Production example

A team uses advanced glossary to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

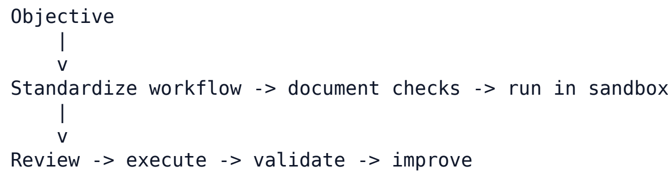
- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

6.1 - Practical Annexes

Part 3 - English operational chapter

Practical Annexes is a core capability for reliable DevOps work. In production, the objective is not only to make a change work once, but to make it understandable, repeatable, auditable and recoverable. Training plans, risk matrices, go/no-go procedures and operational templates. The practical goal is to reduce uncertainty: engineers should know what will change, how to validate it, how to roll it back and how to communicate the result.

Mental model



What good looks like

- Define the operational objective before choosing tools.

- Keep the workflow visible, reviewed and repeatable.

- Prefer small reversible changes over large opaque changes.

- Document decisions so new team members can learn from them.

Tools commonly involved

Tool / area	How it helps
risk matrix	Supports practical annexes by adding automation, evidence, traceability or control.
go/no-go	Supports practical annexes by adding automation, evidence, traceability or control.
RCA template	Supports practical annexes by adding automation, evidence, traceability or control.
change record	Supports practical annexes by adding automation, evidence, traceability or control.
checklists	Supports practical annexes by adding automation, evidence, traceability or control.

6.1 - Practical Annexes: Production Use

Production example

A team uses practical annexes to convert an informal practice into a clear operating model. The change is documented, reviewed, tested in a lower environment and then applied with visible verification steps.

Operational checklist

Check	Question to answer	Evidence
Context	Which service, environment and owner are involved?	Ticket, MR, service name, environment.
Risk	What is the blast radius and failure mode?	Risk note, affected users, dependency map.
Validation	How do we know it worked?	Health checks, metrics, logs, business check.
Rollback	How do we return to a stable state?	Rollback command, previous version, restore plan.
Communication	Who needs an update and when?	Incident channel, status note, next update time.

Useful commands or snippets

```
date -Is
curl -I https://service/health
tail -n 100 /var/log/nginx/error.log
journalctl -u app -n 120 --no-pager
ss -tulpn
```

Anti-patterns

- Changing production manually and documenting it later.

- Approving a plan without reading the destroy/update summary.

- Troubleshooting by repeatedly retrying without forming a hypothesis.

- Keeping the rollback procedure in someone's memory instead of the MR/runbook.

Appendix A - Go / No-Go Template

```
CHANGE:
- Service:
- Environment:
- Owner:
- Reviewer:
- Planned window:
- Expected impact:

PRE-CHECKS:
- Pipeline green:
- Terraform plan reviewed:
- No unexpected destroy:
- Secrets protected:
- Backup verified when data is involved:
- Monitoring dashboard ready:
- Rollback procedure ready:

DECISION:
- GO / NO-GO:
- Approver:
- Time:
- Conditions or follow-up:
```

Decision rules

- No rollback, no go for production changes with user impact.

- Unexpected destroy must block approval until explained.

- Missing monitoring means delayed deployment unless the risk is explicitly accepted.

- Data-impacting changes require verified backup and restore confidence.

Appendix B - Incident Template

```
INCIDENT:
- Title:
- Severity:
- Start time:
- Detection time:
- Incident commander:
- Services affected:
- Customer impact:

TIMELINE:
- HH:MM alert fired
- HH:MM triage started
- HH:MM mitigation started
- HH:MM service restored

FACTS:
- Confirmed observations only.

HYPOTHESES:
- Suspected causes and tests.

MITIGATION:
- Actions performed to reduce impact.

ROOT CAUSE:
- Technical cause:
- Process cause:
- Detection gap:

ACTION ITEMS:
- Action / owner / due date / priority
```

Appendix C - Risk Matrix

Level	Impact	Example	Expected control
1	No user impact	Tag update, non-prod cleanup	Standard review
2	Limited impact	Staging resource change	Review and plan
3	Production but rollback simple	Autoscaling or Nginx tweak	Go/no-go recommended
4	Critical production area	DB, IAM, public network	Senior approval required
5	Data loss/outage/security risk	Restore, major migration	Dedicated window and tested rollback

Scoring approach

```
risk score = impact x probability x blast radius x rollback difficulty
```

Appendix D - DevOps Onboarding Path

Period	Focus	Expected deliverable
Days 1-30	Architecture, GitLab pipeline reading, Terraform plan reading	Explain a real pipeline and summarize a plan.
Days 31-60	Sandbox changes, runbooks, observability and small MRs	Create a small MR and update a runbook.
Days 61-90	Staging autonomy and supervised production change	Lead a go/no-go checklist under supervision.
After 90	Incident simulation, RCA, module contribution	Run a game day or write a post-mortem.

End of document

A mature DevOps culture is built by repeating the same simple habits: small changes, clear reviews, visible plans, protected credentials, reliable observability, tested runbooks and honest post-incident learning.