SALT

Platform Engineering Playbook

Step-by-step playbook for modernizing legacy systems and building scalable cloud infrastructure. Includes SRE best practices and DevOps automation strategies.

Salt Technologies, Inc.
Software Consulting & Development Partner
Austin, TX, USA | Pune, MH, India

1. What is platform engineering (and why it matters now)?

Most engineering teams are under pressure to:

- Ship features faster.
- Keep systems stable and secure.
- Adopt cloud-native, containers, Kubernetes, data platforms, and now AI…
- Without burning out developers.

Platform engineering is about building the internal “product” that your engineers use to deliver software:

- Standardized, paved paths for building, testing, and deploying services.
- Self-service environments and infrastructure, instead of ticket queues.
- Observability, security, and SRE practices baked in by default.

The goal: make the right thing the easy thing for every product team.

This playbook walks through how to move from fragile legacy setups to a modern, scalable platform—step by step.

2. Principles of a good platform

Before you touch tools, align on principles.

A good platform is:

- Product-like:
  - It has users (your engineers).
  - It has a roadmap, feedback loops, and clear ownership.
- Opinionated but flexible:
  - Provides strong defaults and paved paths.
  - Still allows escape hatches for edge cases.
- Self-service and automated:
  - Engineers don’t wait days for infra or deployments.
  - Most “ops work” is done via APIs, CLIs, or portals.
- Secure and observable by design:
  - Identity, access, logging, and monitoring are baked in.
  - Every new service gets these features automatically.

Keep these principles visible when making decisions about tools, architectures, and processes.

3. Step 1 – Assess your current landscape

Modernization starts with a clear map of where you are.

3.1 Inventory systems and services

Create a simple catalog:

- Applications and services (monoliths, microservices, cron jobs).
- Tech stack: languages, frameworks, runtime environments.
- Hosting model: on-prem, VMs, containers, cloud services.
- Dependencies: databases, queues, 3rd-party APIs, batch jobs.

3.2 Map pain points

Talk to devs, ops, and business stakeholders:

- Where do outages and incidents usually start?
- Which deployments are scary or frequently rolled back?
- Where does work get stuck—environments, approvals, manual steps?
- What's the usual cause of "we can't deliver this on time"?

Capture these in a **simple list**: problem, impact, owners, frequency.

3.3 Identify constraints

You also need to know your boundaries:

- Regulatory requirements (data residency, PII, logs retention).
- Existing contracts (cloud providers, tools, managed services).
- Skillsets of current team (Kubernetes experience, IaC, SRE, etc.).

This baseline informs your modernization strategy and priorities.

4. Step 2 – Define your target platform & architecture

You're not trying to modernize everything at once. You're defining **what "good" looks like** for the next 2–3 years.

4.1 Choose your "platform spine"

Decide core choices (not every low-level detail):

- **Cloud provider(s):** AWS, Azure, GCP, hybrid, or on-prem + cloud.
- **Compute model:** Containers on Kubernetes, managed PaaS, serverless, or mix.
- **Infra as Code:** Terraform, Pulumi, CloudFormation, etc.
- **CI/CD core:** GitHub Actions, GitLab CI, Azure DevOps, Jenkins, etc.

The goal is to have a **consistent backbone** most teams can ride on.

4.2 High-level target architecture

Define:

- How services will be structured (modular monolith vs microservices vs “macroservices”).
- How they talk: APIs, gRPC, event-driven messaging, etc.
- Shared components: API gateways, identity provider, messaging, observability stack, secrets management.

Create **simple diagrams**, not 50-page architecture docs.

5. Step 3 – Establish platform foundations

This is where platform engineering really starts: standard rails for teams.

5.1 Environments and Infra as Code

- Standardize environments: `dev`, `test`, `staging`, `prod` (and maybe `sandbox`).
- Define them in **code**:
  - Networks, subnets, security groups.
  - Databases and storage.
  - Kubernetes clusters or PaaS resources.

Every change to infrastructure goes through version control and review.

5.2 CI/CD pipelines

Create **template pipelines** that teams can reuse:

- Triggered on PRs and main branch merges.
- Steps typically include:
  - Build & unit tests.
  - Security scans / linting.
  - Packaging (container images, artifacts).
  - Deploy to test/staging.
  - Automated smoke tests.
  - Controlled promotion to production (with approvals if needed).

Developers should be able to create a new service and get a working pipeline with minimal configuration.

5.3 Observability stack

Standardize:

- **Logging:** centralized logs with structured fields (correlation IDs, tenant, user, request ID).
- **Metrics:** system & application metrics (latency, error rate, throughput, saturation).
- **Tracing:** distributed tracing across services.
- **Dashboards & alerts:** pre-defined templates for new services.

“No service goes to production without standard observability baked in.”

6. Step 4 – Apply SRE best practices

Platform engineering and SRE go hand in hand. SRE gives you the **operational playbook**.

6.1 SLOs and error budgets

For critical services:

- Define Service Level Objectives (SLOs):
  - e.g., 99.9% availability, p95 latency under 300 ms, etc.
- Derive **error budgets** (how much unreliability is acceptable).
- Use them to balance reliability vs feature velocity.

6.2 Incident management

Create a simple incident process:

- Severity levels (SEV1, SEV2, etc.) and examples.
- On-call schedules and escalation paths.
- Templates for:
  - Incident declaration.
  - Status updates.
  - Post-incident review (blameless, focused on learning).

Platform teams should provide shared tools: paging, incident channels, dashboards.

6.3 Capacity & performance

- Basic capacity planning for critical services (traffic patterns, seasonal peaks).
- Load testing and performance baselines for key endpoints.
- Autoscaling rules (for containers/serverless) based on metrics, not guesswork.

7. Step 5 – DevOps automation and self-service

Now you shift from “platform exists” to “platform is actually convenient.”

7.1 Self-service workflows

Make common actions one click or one command:

- Provision a new service (repo + pipeline + basic infra + observability).
- Request or refresh a database/schema.
- Rotate secrets and credentials.
- Create a new environment (ephemeral or long-lived).

This can be via:

- A portal / internal developer platform UI.
- CLI tools / templates.

- ChatOps (commands in Slack/Teams).

7.2 Golden paths and templates

Provide opinionated templates for:

- REST/GraphQL services in your primary languages.
- Event-driven consumers and producers.
- Batch/ETL jobs.
- Frontend apps (e.g. Next.js with your design system).

Each template includes:

- Basic folder structure.
- CI/CD config.
- Observability hooks.
- Security baseline (linting, dependencies checks, auth hooks).

7.3 Guardrails, not gates

Replace slow, manual approvals with **automated checks**:

- Static analysis, SAST, dependency scanning.
- Policy-as-code (e.g., for infrastructure policies).
- Deployment rules: canary releases, feature flags, rollbacks.

Platform engineers define **guardrails** that keep the system safe, while dev teams move fast within them.

8. Step 6 – Security, compliance, and governance

Your platform should make secure and compliant behavior the default.

8.1 Identity and access

- Centralized identity provider (SSO, SAML/OIDC).
- Role-based access control for:
  - Cloud consoles.
  - CI/CD systems.
  - Production environments.
- Just-enough and just-in-time access where possible.

8.2 Secrets and sensitive data

- Use a dedicated secrets manager; never store secrets in code or config repos.
- Rotate secrets regularly (and automatically, where possible).
- Classify data (public, internal, confidential, restricted) and handle accordingly.

8.3 Compliance automation

Where relevant (e.g., SOC 2, ISO, GDPR):

- Automate evidence collection from cloud, CI/CD, and platform tools.

- Standardize logging and retention policies.
- Embed security checks in pipelines (so they're not a separate, manual gate).

9. Step 7 – Rolling out the platform & driving adoption

A platform that no one uses is just an expensive science project.

9.1 Treat the platform as a product

- Define your **users**: backend devs, frontend devs, data engineers, SREs, etc.
- Create a feedback loop:
  - Office hours.
  - Slack channel for questions.
  - Regular surveys or NPS-style score for platform satisfaction.

9.2 Start with one or two pilot teams

- Pick teams that are motivated and representative.
- Onboard them to:
  - New CI/CD pipelines.
  - Observability stack.
  - Self-service workflows.

- Iterate based on their feedback before rolling out to everyone.

9.3 Documentation & enablement

- Lightweight docs:
  - “How to create a new service.”
  - “How to deploy.”
  - “How to debug issues using logs, metrics, and traces.”
- Short videos or internal workshops to demonstrate workflows.
- Internal champions in each product team to help peers adopt the platform.

10. A 90–day platform engineering roadmap

You don’t need a 2-year big-bang. Start with a focused, 90-day push.

Days 0–30: Discover & design

- Map systems, pain points, and constraints.
- Choose your cloud/platform spine (at least for initial scope).
- Define 1–2 **pilot teams** and their goals.
- Design your first version of:
  - Standard CI/CD.

    - Observability.
    - Basic IaC structure.

Days 30–60: Build core platform slices

- Implement:
    - Core CI/CD templates.
    - Logging/metrics/tracing stack.
    - Baseline IaC for dev/stage/prod.
- Onboard pilot teams to:
    - New pipelines.
    - Environment provisioning.
    - Observability tools.

Days 60–90: Prove value & harden

- Run real deployments and feature work through the new platform.
- Capture metrics:
    - Deployment frequency.
    - Mean time to recovery (MTTR).
    - Lead time for changes.
- Refine:
    - Guardrails and automation.
    - Documentation and templates.

- Decide on next steps: more teams, more services, additional capabilities (self-service infra, feature flags, etc.).

By day 90, your goal is to have **one meaningful part of your system running on the new platform** and at least one product team saying, “This makes our life easier.”

11. How Salt Technologies can help

Platform engineering is where architecture, DevOps, and SRE all intersect—and where many teams get stuck trying to do everything at once.

At **Salt Technologies**, we help organizations:

- Assess legacy systems and deployment pipelines to identify high-impact modernization opportunities.
- Design and implement cloud-native platforms with **IaC, CI/CD, observability, and security** baked in.
- Introduce SRE practices—SLOs, error budgets, incident response—and integrate them into your platform.
- Build self-service developer experiences so your internal teams can create, deploy, and operate services with confidence.

We typically work in focused, iterative phases:

1. **Scope & Shape:** understand your systems, constraints, and goals.

2. **Plan & Pilot:** design a pragmatic target platform and validate it with one or two teams.
3. **Architect & Automate:** codify environments, pipelines, and guardrails.
4. **Release & Run:** move real workloads onto the platform and stabilize operations.
5. **Keep Improving:** refine, extend to more teams, and optimize for speed, reliability, and cost.

If you're looking to modernize legacy systems, move more confidently to the cloud, or give your teams a better way to ship and run software, Salt can act as your **platform engineering partner**—from strategy and architecture to hands-on implementation.

https://www.salttechno.com/ | sales@salttechno.com