Engineering Playbook

Orcta Engineering

June 28, 2025

Contents

1 Introduction

1.1 Purpose of the Playbook

This playbook serves as a comprehensive guide to the engineering practices, culture, and principles that define Orcta Engineering. It ensures consistency, accelerates onboarding, and acts as a reference for decision-making. It is designed to evolve with the team. Updates are tracked in a version-controlled Git repository, with a changelog maintained in Notion.

1.2 Who Should Read It

All engineers at Orcta, from interns to tech leads, including freelancers and cross-functional collaborators, should read and understand this document. Product managers and stakeholders may also find sections useful for understanding how engineering operates.

1.3 Connection to Engineering Philosophy

This document encapsulates our belief that engineering is a craft rooted in clarity, consistency, and continual learning. It aligns with our mission to build reliable, scalable, and meaningful software that uplifts communities.

2 Engineering Culture and Principles

2.1 Code as Craft

We treat software development as a discipline that requires care, intention, and respect. Clean code, thoughtful abstractions, and maintainability are valued over fast hacks or clever one-liners.

2.2 Consistency Over Cleverness

Readable and consistent code outlasts clever code. If two implementations are equivalent, we choose the one that's simpler, more familiar to the team, and easier to maintain.

2.3 Humility and Collaboration

We assume positive intent and embrace diverse perspectives. We engage in respectful debate, ask questions, and recognize that code review is about improving code, not judging people.

2.4 Ownership and Accountability

Every engineer owns their code, from local development to production monitoring. If you ship it, you own it. We embrace blameless postmortems and take responsibility collectively.

3 Code Standards

3.1 Language-Specific Guidelines

We maintain specific style guides for JavaScript/TypeScript, Python, and Go. Each guide includes best practices, patterns to avoid, and idioms to follow.

3.2 Linting and Formatting

Every repo must include automated linting (e.g., `ESLint`, `Flake8`) and consistent formatting tools (e.g., `Prettier`, `Black`). CI checks enforce code hygiene.

3.3 Naming Conventions

Names should be descriptive and consistent. Use `camelCase` for variables, `PascalCase` for components/classes, and `snake_case` for filenames (as appropriate).

3.4 Folder/Project Structure

Project directories follow well-defined conventions to reduce onboarding time. Common layouts include separation of concerns (e.g., `components/`, `services/`, `utils/`).

4 Version Control and Branching Strategy

4.1 Git Conventions

All code must be version-controlled using Git. Commits should be atomic and descriptive. Use imperative mood (e.g., `Add login form`, not `Added login form`).

4.2 Branching Model

We follow a simplified Git Flow: `main` for production, `dev` for integration, `feature/{name}` for features, and `hotfix/{name}` for urgent patches.

4.3 Pull Request Etiquette and Merge Policy

Pull Requests (PRs) must pass CI, include descriptions, and be reviewed by at least one peer. Do not self-merge unless authorized. Squash commits before merging to `main`.

5 Code Review Process

5.1 Purpose and Value

Code reviews are vital for maintaining quality, sharing knowledge, and mentoring. They're not gatekeeping, but collaboration opportunities.

5.2 Pull Request Quality Checklist

Before requesting review:

- Run tests and linters
- Write clear PR titles and descriptions
- Link to related issues
- Include screenshots or test evidence if needed

5.3 Review Responsibilities and Timing

Reviewers must respond within 48 hours. Authors should address feedback promptly. If discussions stall, escalate to a tech lead or hold a short sync.

6 CI/CD and Deployment

6.1 Tools and Pipelines

We use `GitHub Actions` for CI, `Docker` for builds, and `Railway` or `Vercel` for deployments. Pipelines must be deterministic and reproducible. See Section 7 for testing integration in CI.

6.2 Deployment Environments

Our stack includes `dev`, `staging`, and `production` environments. Every change goes through `dev` and `staging` before `production`.

6.3 Release Permissions and Monitoring

Only team leads or assigned reviewers can deploy to production. Every release must be tracked with monitoring tools (e.g., `Sentry`, `Grafana`).

6.4 CI/CD Pipeline Diagram

Figure 1 illustrates our CI/CD pipeline, showing the flow from code commit through testing, building, and deployment across environments. It highlights the use of `GitHub Actions` for automation and `Docker` for consistent builds.

Figure 1: CI/CD Pipeline for Orcta Engineering

7 Testing Strategy

7.1 Types of Tests

We define and use unit tests, integration tests, and end-to-end tests. Testing tools include `Jest`, `Playwright`, and `Pytest`.

7.2 When and How to Test

All business logic must be unit-tested. Features with user impact should have end-to-end tests. New code should not reduce test coverage.

7.3 Coverage Expectations

Our target is 80% coverage for core services. Low-coverage areas must be justified. Coverage is checked via CI before merging.

8 Incident Response and Monitoring

8.1 Logging and Observability

All services must log structured events. Logs should be queryable and stored in a central system (e.g., `Logtail`, `Datadog`).

8.2 Alerting and Escalation

Critical services must have alerts set. On-call rotations handle escalations. Incidents are logged and documented.

8.3 Postmortem Process

We conduct blameless postmortems within 72 hours of any critical incident. Postmortems must include:

- What happened
- Why it happened
- How we responded
- Preventative actions

9 Documentation Culture

9.1 Writing and Maintaining Docs

Documentation is not optional. Every new service or feature must have relevant documentation: how to use, how to run, and known issues.

9.2 Tools Used

We use `Notion` for planning and `GitHub Wikis` or `Markdown` for in-repo docs. Docs should be versioned and reviewed like code.

9.3 Discoverability

Documentation must be linked from `READMEs` and repos. A central doc index lives in `Notion` and is updated monthly.

10 Communication and Meetings

10.1 Standups and Retros

Each team holds daily async standups and bi-weekly retrospectives. Meeting notes are documented and accessible.

10.2 Async vs Sync

Async communication (via `Slack`, `Notion`, `GitHub`) is preferred. Synchronous meetings are reserved for alignment, blockers, or design discussions.

10.3 Escalating Blockers

If a blocker lasts over 24 hours, raise it in your team channel. Escalate to project leads if it persists.

11 Tooling and Local Development

11.1 Dev Setup and Tooling

Each repo includes a `README.md` with setup instructions. Use `Docker` or prebuilt scripts for consistent environments.

11.2 Shared CLIs and Scripts

Teams maintain shared tools (e.g., `orcta-cli`) to scaffold components, run tests, or deploy.

11.3 Secrets Management

Never commit secrets. Use environment variables and secrets managers (e.g., `Doppler`, `Vault`).

12 Onboarding and Growth

12.1 Onboarding Phases

New hires follow a 3-phase onboarding: Week 1 (Setup), Weeks 2-3 (Shadow and Contribute), Month 2+ (Own and Improve).

12.2 Mentorship

Each new engineer is paired with a mentor for the first 90 days. Mentorship includes code reviews, 1:1s, and growth check-ins.

12.3 Growth Paths

Engineers have access to a documented growth framework outlining expectations for Junior, Mid-level, Senior, and Lead roles.