Deployment and Environment Setup Guide

Target Audience: DevOps Engineers, System Administrators
Purpose: Defining the repeatable, secure, and scalable process for moving code from "Laptop" to "Live Production".

---

1. Environment Strategy

We utilize a 3-tier environment strategy to ensure stability.

1.1 Development (Local)

- **Infrastructure:** Developer's laptop (Docker Compose).
- **Data:** Seeded sample data (`seed_db.py`).
- **Purpose:** Feature development and rapid iteration.
- **Config:** `.env` file (locally integrated).

1.2 Staging (UAT)

- **Infrastructure:** Cloud VM (e.g., AWS EC2 or DigitalOcean Droplet) mirroring Production.
- **Data:** Anonymized production clone or synthetic heavy load.
- **Purpose:**
  - **User Acceptance Testing (UAT):** Client clicks through new features.
  - **Integration Testing:** Verifying microservices talk to each other correctly.
- **Trigger:** Automated deploy on `git push origin develop` .

1.3 Production (Live)

- **Infrastructure:** Kubernetes Cluster (EKS/AKS) or Scaled Docker Swarm.
- **Data:** Live Customer Data (Strictly protected).
- **Purpose:** Serving real-world traffic. High availability (HA) is mandatory.
- **Trigger:** Manual approval + Tagged Release (`v1.2.0`).

---

2. Configuration Strategy

We adhere to the **12-Factor App** methodology: **Store config in the environment.**

- **Non-Secret Config:** (e.g., `LOG_LEVEL=INFO`, `FEATURE_NEW_UI=true`)
  - Managed via Environment Variables injected by the Orchestrator (K8s ConfigMap).
- **Secret Config:** (e.g., `DB_PASSWORD`, `STRIPE_KEY`)
  - **Local:** `.env` file (git-ignored).
  - **Production:** Secrets Manager (AWS Secrets Manager, HashiCorp Vault, or GitHub Secrets).
  - *Rule:* Secrets are **NEVER** committed to Git.

---

3. Containerization Strategy

- **Images:** Each microservice has a `Dockerfile`.
- **Base Image:** `python:3.11-slim` (Lightweight, fewer vulnerabilities).
- **Build Process:**
  1. Install system dependencies.
  2. Copy `requirements.txt`.
  3. `pip install` (Cache layer).
  4. Copy source code.
  5. Define entrypoint (`uvicorn`).

---

4. Deployment Flow (CI/CD Pipeline)

We recommend GitHub Actions or GitLab CI.

Step 1: Build & Test (On Pull Request)

1. Checkout code.
2. Run Unit Tests (`pytest`).
3. Lint Code (`flake8`, `black`).
4. **Gate:** If tests fail, Merge is blocked.

Step 2: Package (On Merge to Main)

1. Build Docker Images (`tms-auth-service:latest`, `tms-payment-service:latest`).
2. Scan for Vulnerabilities (Trivy/Snyk).
3. Push to Container Registry (ECR/Docker Hub).

Step 3: Deploy (CD)

1. SSH into Staging/Prod Server (or talk to K8s API).

2. Update Deployment manifest with new Image Tag.
3. **Rolling Update:**
    - Spin up new Container (v2).
    - Wait for Health Check (`/health` endpoint returns 200 OK).
    - Shift Traffic.
    - Kill old Container (v1).

---

5. Database Migrations

- **Rule:** Migrations run **before** the code deployment finishes.
- **Mechanism:** Docker `init container` or a Pre-Deploy Job runs `alembic upgrade head`.
- **Constraint:** Migrations must be non-destructive (backwards compatible) to support zero-downtime rolling updates.

---

6. Rollback Strategy

"Things go wrong. How do we undo?"

- **Fast Rollback:** Since configuration and images are versioned:
    - Command: `kubectl rollout undo deployment/auth-service`
    - Result: The orchestrator instantly reverts to the previous known good Image Tag.
- **Database Rollback:** Much harder.
    - Policy: Avoid rolling back DB migrations unless catastrophic. Instead, "Roll Forward" with a fix script.

---

7. Monitoring & Observability

"Is it working?"

- **Logs:** All containers output to `stdout/stderr`. Aggregated via Fluentd/ELK Stack or Loki.
- **Metrics:** Prometheus scrapes `/metrics` endpoints. Grafana dashboards visualize CPU, RAM, and Request Latency.
- **Alerting:** PagerDuty/Slack notification if `Error Rate > 1%`.