Part 1: Crisis Assessment & Immediate Stabilization

Root Cause Analysis (Top 5 causes of 70% deployment failures)

1. Inconsistent Environment Configuration & Hardcoded Values: Using hardcoded environment variables within a code base will lead to runtime failure in environments. The development DB endpoint can be different from what the UAT and Production uses. There could be missing secrets also
2. Manual and Inconsistent Builds: Manual builds on different machines lead to unique Docker images. These small, inconsistent differences mean the same application can work on one developer's computer but fail unpredictably in other environments. This issue is why automated, consistent build processes are crucial.
3. Lack of Automated Testing and Quality Gate checks: Untested and broken code that should have been tested and detected in the development via the pipeline and UAT stages by the QA engineers gets to the production environment. If quality gate such as SonarQube had scanned the code to detect:
   - Security Hotspot
   - Quality of codes written
   - Coverage test written by developers

   Bugs deployed to production could have been prevented
4. Database migration done manually: There are usually issues like flyway version mismatch whenever DB migrations are ran separately. It could cause application failures or data loss
5. Health Checks for dependencies: Readiness/liveness probes should be set up in Kubernetes. For example, the below shows service dependencies
   payment-service → account-service (balance validation)
   payment-service → audit-service (compliance logging)
   When the payment service starts up and the dependent service (account or audit) is not available, the result is a chain reaction where a single failure cascades and takes down the entire application.

   Prioritization Timeline & Reasoning
   Priority #1: Health Checks for Dependencies
   This is the fastest way to prevent cascading failures and reduce the 70% deployment failure rate. Without proper readiness/liveness probes, Kubernetes sends traffic to broken pods, causing application failure. By adding probe endpoints this can be checkmate.

   Priority #2: Inconsistent Environment Configuration & Hardcoded Values
   This causes runtime failures. This can be fixed by externalizing all hardcoded values to Kubernetes ConfigMaps/Secrets or using a vault like HashiCorp or 1Password to centralized variables, secrets the application needs to run.

   Priority #3: Manual and Inconsistent Builds
   Implementing automated, consistent Docker builds ensures the same artifact tested in dev is what runs in production. This eliminates "it worked on my machine" failures and creates reproducible deployments.

Priority #4: Lack of Automated Testing and Quality Gates
We can add quality gates. By adding unit tests in the pipeline, and integration tests and basic security scanning like secret scans, Veracode, trivy, snyk. This prevents known bugs from reaching production while we build toward more comprehensive quality checks.

Priority #5: Database Migration Manual Process
Database changes carry the highest risk (potential data loss), so we address this last when we have more stability. Implementing automated, version-controlled migrations requires careful planning and rollback strategies.

30-Day Emergency Plan

Objective: Reach 90%+ deployment success rate in 1 month.

- **Week 1–2**
    - Standardize builds (CI pipelines, Maven + Docker buildx).
    - Store configs in Git + parameterize via Helm/Kustomize + secrets in Vault/SSM.
    - Add tests & service health probes.
- **Week 2–3**
    - Automate DB migrations with Flyway + rollback strategy.
    - Introduce Blue/Green or Canary deploys in Kubernetes.
    - Implement centralized logging (EFK/ELK) + Prometheus alerts for pod crash.
- **Week 4**
    - Train dev team on pipeline use, IaC, config hygiene.
    - Enforce security scanning (Snyk/Trivy for images, OWASP ZAP in pipeline).

Risk Assessment

Compliance Risks: Manual deployments violate audit trail requirements.
Solution: All changes must be committed using Git via a remote repository like GitHub or Bitbucket, this will provide an immutable audit log.

Security risks: Secrets hardcoded in configs, images unscanned.

Stabilization Risks: Migration automation may cause outages if rollback not tested
Mitigation:
- Apply changes to a single non-production environment first, validate, then proceed. You can have the development, UAT and DR.
- Keep manual approval gates for Prod during stabilization, change request with approval from the necessary parties before deployment to prod.
- Carry failover exercise on the Disaster Recovery environment from time to time