University of Phoenix®

Post-Mortem

Use the table to list the things that went well during the completion of this 5-week project and the things that didn't go well.

Date: 4/17/2025

Project Manager (your name): Kenneth Quiggins

Project Name: Kentech Banking

5 Things that Went Well During the Project	5 Things that Could Have Been Done Better
Robust AWS-based architecture design : The use of AWS services like CloudFront, ALB, ECS Fargate, and RDS ensured scalability and security.	Incomplete third-party integration testing : Third-party integrations were excluded from testing scope, risking untested dependencies.
Comprehensive database schema : The detailed data dictionary and table relationships (e.g., USERS, ACCOUNTS, TRANSACTIONS) supported a strong foundation for financial operations.	Limited performance testing scope : Load testing was planned for only 10,000 concurrent users, potentially underestimating real-world peak loads.
Strong cybersecurity plan : Encryption (AES-256, TLS/SSL), WAF, and RBAC ensured robust security.	Short security testing duration : Only 7 days were allocated for security and compliance testing, risking insufficient vulnerability detection.
Thorough testing strategy : The testing plan covered unit, integration, system, and acceptance tests with clear pass/fail criteria.	Lack of explicit disaster recovery testing : The plan did not include specific tests for backup and recovery processes.
Effective CI/CD pipeline : AWS CodePipeline and Jenkins facilitated continuous integration and delivery, streamlining development.	Insufficient accessibility testing resources : Accessibility testing relied on end users/product owners, potentially lacking expertise.

Below, discuss all 10 things from the list above in detail. As a project manager, or participant, what processes might you put in place to ensure the same things go well on future projects you're involved in? What processes can you put in place to improve the not-so-great things on future projects? Be specific.

Things that Went Well

1. Robust AWS-based architecture design
   - Why it went well: The project plan outlined a scalable and secure architecture using AWS services like CloudFront for CDN, Application Load Balancer (ALB) for traffic management, ECS Fargate for containerized Next.js applications, and Amazon RDS for relational data storage. The separation of public and private subnets with a NAT Gateway enhanced security, while CloudWatch provided monitoring and logging. This design

aligned with modern cloud best practices, ensuring performance and reliability for a financial application.

- Why highlighted: This is a strength because it leverages AWS's managed services to handle scalability and security, critical for a banking platform. The architecture supports high availability and fault tolerance, as evidenced by the use of ALB and CloudFront.
- Process to ensure success in future projects:
  - Standardized architecture templates: Create reusable cloud architecture templates (e.g., VPC configurations, subnet designs) to streamline future project setups.
  - Architecture review checkpoints: Implement mandatory peer reviews and cloud architect consultations during the design phase to validate scalability and security.
  - Training on cloud services: Provide team training on AWS services to ensure all developers understand the tools used, reducing misconfigurations.

2. Comprehensive database schema
   - Why it went well: The data dictionary and table definitions provided a detailed schema for tables like USERS, ACCOUNTS, CARDS, LOANS, and TRANSACTIONS, with clear primary/foreign key relationships, constraints (e.g., CHECK for account_type), and encryption for sensitive data (e.g., card_number). The USERS table acted as a central hub, enabling one-to-many relationships with financial products, which supported complex banking operations.
   - Why highlighted: A well-designed database is critical for financial applications, and this schema's clarity and robustness minimizes errors in data handling and ensures compliance with regulatory needs (e.g., audit logging).
   - Process to ensure success in future projects:
     - Database design workshops: Conduct early workshops with DBAs and developers to align on schema requirements before coding begins.
     - Automated schema validation: Use tools like Flyway or Liquibase to enforce schema consistency and version control across environments.
     - Documentation standards: Mandate detailed data dictionaries for all projects, including purpose, relationships, and constraints, to ensure clarity.
3. Strong cybersecurity plan
   - Why it went well: The cybersecurity plan outlined robust measures, including AES-256 encryption for data at rest, TLS/SSL for data in transit, WAF for web protection, and RBAC for access control. It also included compliance with PCI DSS and SOC 2, regular vulnerability assessments, and audit logging via the AUDIT_LOGS table (Page 11). These measures protected sensitive financial data and ensured regulatory adherence.
   - Why highlighted: Security is paramount in banking, and this plan addressed key risks (e.g., OWASP Top 10 vulnerabilities) while providing a framework for continuous monitoring and improvement.
   - Process to ensure success in future projects:
     - Security-first design: Integrate security requirements into the initial project scope, using frameworks like NIST or OWASP.
     - Automated security scans: Implement tools like Snyk or AWS Inspector for continuous vulnerability scanning during development.

        - Regular security drills: Schedule quarterly tabletop exercises to test incident response plans and maintain team preparedness.

4. Thorough testing strategy
    - Why it went well: The testing plan covered unit, integration, system, and acceptance tests, with specific cases for user registration, transaction processing, and security breach simulation. Clear pass/fail criteria (e.g., no vulnerabilities detected, meeting performance benchmarks) ensured quality. Tools like Selenium, JMeter, and Jenkins supported automated and manual testing, aligning with agile practices.
    - Why highlighted: Comprehensive testing reduced the risk of defects in critical banking features, ensuring reliability and user trust. The use of industry-standard tools and clear criteria made the process repeatable.
    - Process to ensure success in future projects:
        - Test-driven development (TDD): Encourage TDD practices to write tests alongside code, catching issues early.
        - Centralized test repository: Maintain a shared repository for test cases and scripts, accessible via GitHub, to reuse across projects.
        - Regular test reviews: Schedule QA team reviews to update test cases based on new features or regulatory changes.
5. Effective CI/CD pipeline
    - Why it went well: The use of AWS CodePipeline for CI/CD and Jenkins for automation enabled continuous integration and delivery, streamlining deployments. This reduced manual errors and ensured rapid iteration, critical for an agile project. The pipeline integrated with GitHub for code management and Jira for issue tracking, creating a cohesive workflow.
    - Why highlighted: A reliable CI/CD pipeline is essential for modern software projects, and this setup supported frequent, stable releases, aligning with the project's agile methodology.
    - Process to ensure success in future projects:
        - Pipeline templates: Develop standardized CI/CD pipeline configurations for common tools (e.g., Jenkins, CodePipeline) to reduce setup time.
        - Automated testing in CI/CD: Mandate that all pipeline stages include automated unit and integration tests to catch issues before deployment.
        - Pipeline monitoring: Use tools like AWS CloudWatch or Jenkins plugins to monitor pipeline performance and alert on failures.

Things that Could Have Been Done Better

1. Incomplete third-party integration testing
    - Why it didn't go well: The testing plan explicitly excluded third-party integrations from the scope, noting they would be tested by vendors. This omission risked untested dependencies (e.g., payment gateways, external APIs), which could cause failures in production, especially for a banking application reliant on external services.
    - Why highlighted: Third-party integrations are critical for banking platforms, and untested interfaces could lead to transaction errors or security vulnerabilities, undermining user trust.
    - Process to improve future projects:

        - Vendor testing coordination: Establish SLAs with third-party vendors requiring detailed test reports and joint integration testing sessions.
        - Mock API testing: Use tools like WireMock or Postman to simulate third-party APIs during testing, ensuring compatibility.
        - Integration test phase: Add a dedicated integration testing phase for third-party services, with specific test cases in the testing plan.

2. Limited performance testing scope
    - Why didn't it go well: The performance test case was limited to 10,000 concurrent users, which may not reflect peak loads for a banking application, especially during events like tax season or major sales. This narrow scope risked undetected performance bottlenecks under real-world conditions.
    - Why highlighted: Performance is critical for user experience in banking, and underestimating load could lead to slow transactions or outages, damaging reputation.
    - Process to improve future projects:
        - Load forecasting: Conduct user load analysis during requirements gathering to estimate realistic peak concurrent users (e.g., 50,000+).
        - Scalability testing: Include tests for auto-scaling (e.g., ECS Fargate scaling) and stress testing beyond expected loads using tools like JMeter.
        - Performance benchmarking: Define and test against industry-standard benchmarks (e.g., response times under varying loads) to ensure robustness.
3. Short security testing duration
    - Why it didn't go well: Only 7 days were allocated for security and compliance testing, despite the critical need for thorough vulnerability detection in a banking application. This short timeframe risked missing subtle vulnerabilities or compliance gaps, especially given the complexity of PCI DSS and SOC 2 requirements.
    - Why highlighted: Security is non-negotiable in financial systems, and insufficient testing could lead to breaches, fines, or loss of customer trust.
    - Process to improve future projects:
        - Extended security testing: Allocate at least 14 days for security testing, including penetration testing and compliance audits.
        - Continuous security scans: Integrate tools like OWASP ZAP or Nessus into the CI/CD pipeline for ongoing vulnerability detection.
        - External security audits: Engage third-party security firms for independent audits to complement internal testing.
4. Lack of explicit disaster recovery testing
    - Why it didn't go well: The testing plan did not include specific tests for disaster recovery or backup restoration, despite the cybersecurity plan's emphasis on continuous monitoring and incident response. This omission risked unpreparedness for data loss or system outages, critical for a banking platform.
    - Why highlighted: Downtime or data loss in banking can lead to significant financial and reputational damage, making disaster recovery testing essential.
    - Process to improve future projects:

        - Disaster recovery test cases: Add specific test cases for backup restoration, failover to secondary regions, and system recovery, using tools like AWS Backup.
        - Regular DR drills: Schedule quarterly disaster recovery drills to test and refine recovery processes.
        - Documented DR plan: Include a detailed disaster recovery section in the project plan, specifying RTO (Recovery Time Objective) and RPO (Recovery Point Objective).

5. Insufficient accessibility testing resources
    - Why it didn't go well: Accessibility testing was assigned to end users and product owners over 5 days, who may lack expertise in accessibility standards (e.g., WCAG 2.1). This approach risked missing critical accessibility issues, reducing inclusivity for users with disabilities.
    - Why highlighted: Accessibility is a legal and ethical requirement for financial applications, and inadequate testing could exclude users and invite compliance violations.
    - Process to improve in future projects:
        - Dedicated accessibility team: Assign trained QA engineers with WCAG expertise to conduct accessibility testing.
        - Automated accessibility tools: Use tools like Axe or WAVE to scan for accessibility issues during development and testing.
        - User testing with diverse groups: Include users with disabilities in beta testing to validate accessibility features.