University of Phoenix®

Project Plan Draft

Fill out each of the sections below with information relevant to your project. Be sure to include the company name associated with your project.

Company Name:

Network Technology Recommendations

Network Technology Selection Criteria

Selection Criteria Name ***(short name to ID the criteria)***	**Selection Criteria Description** ***(Define the criteria for technology to associate with a point value.)***	**Selection Criteria Value (Weighting in Points)** ***(e.g., 3 – Excellent, 2 – Good, 1 – Acceptable)***
Scalability	Ability to handle 10,000+ concurrent connections and scale on demand	3
Security	Multi-layer security, encryption, isolation (private subnets, IAM)	3
Performance	Low-latency response for global users; minimized network hops	2
Manageability	Ease of configuration and maintenance (automation, infrastructure-as-code)	2
Cost Effectiveness	Overall TCO (infrastructure, maintenance)	2

Network Technology Recommendation

Recommended Network Technologies	**Description**	**Benefits**	**Aggregate Selection Criteria Score** ***(Score for this technology based on the selection criteria detailed above.)***
Multi-Tier Hub-	A layered VPC architecture	- **High Scalability**: Quickly add	3 + 3 + 3 = 9

and-Spoke on AWS VPC	with public subnets (load balancers) and private subnets (application, DB). Leverages AWS application load balancer, Security groups, and VPC peering	subnets or replicate them in multiple regions. - **Strong Security**: Private subnets, IAM, and security groups protect backend resources. - **Manageable**: Infrastructure-as-code, plus native AWS tools for monitoring and automation.	
Azure Virtual Network with Hub-and-Spoke Architecture	This design centralizes connectivity and routing, making it easier to manage traffic between multiple subnets and regions.	- **Scalability:** Easily expands by adding new spokes or regions as your demand grows. - **Security:** Utilizes built-in Network Security Groups (NSGs) and Azure Firewall to enforce security policies and isolate sensitive resources. - **Manageability:** Centralized management via the Azure Portal and automation through ARM templates simplifies configuration and ongoing maintenance.	3 + 3 + 2 = 8

Network Technology Vendor Selection Criteria

(*Third-party technology provider*)

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Trusted in Tech	Vendor must have extensive experience with enterprise-scale cloud networking	3
Security & Compliance	Must offer compliance with banking/financial industry standards (PCI DSS, SOC 2, etc.)	3
Global Presence	Data centers or PoPs worldwide for low-latency and redundancy	2
Integration	Seamless integration with AWS services (VPC, Route 53, CloudFront, WAF)	3

Network Technology Recommended Vendors

Vendor Name	Vendor Strengths	Vendor Weaknesses	Products/Services Provided to Project	Aggregate Selection Criteria Score
Amazon Web Services (AWS)	- AWS VPC integration - Strong security and compliance - Globally available	- Proprietary environment - Cost can rise at scale	- AWS VPC, Subnets, NAT Gateways - AWS Route 53, WAF, CloudFront - Security Groups, Network ACLs	3 + 3 + 3 = 9
Microsoft Azure	- Seamless integration with Azure Virtual Network - Enterprise-grade security and compliance certifications - global data centers with robust hybrid cloud support	- Licensing and management complexity - Costs may increase with scale	- Azure Virtual Network, Subnets, and VPN Gateway - Azure ExpressRoute, Azure Firewall, and Traffic Manager - Network Security Groups (NSGs) and Route Tables	3 + 3 + 2 = 8

Network Technology Deployment Challenges

Deployment Challenge ***(short name to ID the challenge)***	**Deployment Challenge Description** ***(What obstacles can potentially complicate or delay deployment of technology, and affect the project timeline?)***
Network Complexity	Correctly configuring subnets, routing tables, and security groups to ensure no unauthorized access but still allow internal traffic can be quite challenging
Global Latency	Ensuring minimal latency for users worldwide—may require multi-region strategy or CloudFront caching
Compliance Audits	Rigorous documentation and proof that the network meets banking/financial compliance standards (PCI DSS, SOC 2, etc.)
Improvement #1	Implement AWS Config and AWS Security Hub to automatically monitor compliance with

	financial regulations (PCI DSS, SOC 2). (This change is based on research highlighting the importance of automated compliance tracking in finance to reduce audit failure risks and support real-time visibility.)

Technology Adoption Methods

Method Name ***(short name to ID the method)***	**Method Description** ***(Summarize the process for adopting the technology.)***
Infrastructure-as-Code (IaC)	Use AWS CloudFormation or Terraform to define all subnets, security groups, routing, etc. in versioned templates, enabling repeatable, controlled deployments
Pilot / Test Environment	Deploy a smaller-scale pilot environment in a non-production account, validate latency, security configurations, then replicate for production

Cost/Benefit Considerations

Benefits	**Costs**	**Considerations**
Simplifies adding new regions or subnets	AWS service charges for NAT Gateways, data transfer, NAT traffic	Must factor in egress data transfer for global users
Centralized security controls	Price increases with scaling	Possibly more complex to configure at first
Automatic scaling and load balancing	Price increases with scaling	Plan for multi-AZ usage to ensure high availability

Database System Recommendation

Database System Selection Criteria

Selection Criteria Name	**Selection Criteria Description**	**Selection Criteria Value (Weighting in Points)**
ACID Compliance	Must support atomic, consistent, isolated, durable transactions	3
Scalability	Ability to scale read/write operations (horizontal read replicas, vertical scaling)	2

Security & Compliance	Encryption at rest, encryption in transit, auditing, user access controls, meets financial data standards	3
Performanc e	Low latency for queries and transactions (under 2 seconds for 95% of operations)	2

Database System Recommendation

Recommend ed Database System	Description	Benefits	Aggregate Selection Criteria Score
Amazon RDS (MySQL or PostgreSQL) (Relational)	Managed relational database service offering built-in security features, automated backups, read replicas, and multi-AZ failover. Supports ACID transactions critical for finance.	- Acid Compliance - Automatic Scaling - Automated Backups - Built-in Security	3
Azure SQL Database	A fully managed relational database service that supports ACID transactions, automated backups, and high availability. It’s designed to scale dynamically with your application's needs while ensuring robust security and compliance for financial data.	- Acid Compliance - Automatic Scaling - Automated Backups - Built-in Security	3

Database System Vendor Selection Criteria

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Reliability & Uptime	Vendor should offer 99.99% or higher SLA	3
Automated Management	Automated patching, backups, and failover	2
Scalability Options	Support for read replicas, or horizontal scale solutions	2
Security & Compliance Features	Must provide encryption, auditing, and adhere to financial compliance	3

Database System Recommended Vendors

Vendor Name	Vendor Strengths	Vendor Weaknesses	Products/Services Provided to Project	Aggregate Selection Criteria Score
Amazon Web Services (RDS	- Automated Backups - High availability - Encryption in transit/rest	- Proprietary hosting - Additional cost for read replicas	Amazon RDS for MySQL/PostgreSQ	3
Microsoft Azure SQL DB	managed service with automated backups and high availability - Built-in encryption, auditing, and compliance features - Seamless integration with other Azure services (e.g., monitoring, scaling, analytics)	- Proprietary platform that can lead to vendor lock-in - Potentially higher costs at scale, especially with advanced features - Newer AI features may be complex to use	Azure SQL Database (Managed relational database service supporting ACID transactions, geo-replication, and high performance)	2

Database System Deployment Challenges

Deployment Challenge	Deployment Challenge Description
Migration of Data	Importing data from legacy systems or other systems may come with problems
Compliance Auditing	Setting up logs, encryption keys, and audits to prove compliance
Ensuring ACID on High Load	Handling spikes in transactions without losing performance

Technology Adoption Methods

Method Name	Method Description
Proof of Concept (PoC)	Start with a smaller RDS instance in a dev environment to test queries, performance, and security before scaling to production
Database as Code	Manage DB configurations using AWS CloudFormation or Terraform, ensuring consistent dev/stage/prod environments

Cost/Benefit Considerations

Benefits	Costs	Considerations
Fully Managed (backups and patching)	Pay-as-you-go for instances, storage, I/O	Must plan for future capacity growth
Scalable read replicas	Additional cost for Multi-AZ support	Need to evaluate read/write patterns for cost optimization
Strong security & compliance	Additional cost for advanced features	Maintain patch schedules that align with uptime/service windows

Software Application Recommendations

Software Application Selection Criteria

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Modern Web Framework	Must support server-side rendering, fast builds, easy dev experience	3
Security and Auth	Ability to integrate multi-factor auth, secure sessions, and user role management	3
Performance Optimization	Minimizing page load times globally (CDN integration, code splitting, caching)	2
Developer Productivity	Clear documentation, strong community, robust tooling	2

Software Application Recommendation

Recommended Software Application	Description	Benefits	Aggregate Selection Criteria Score

Next.js (Frontend + Backend)	React-based framework for SSR, SSG, and API routes. Facilitates secure, rapid dev, dynamic pages, and easy scaling with serverless or container-based hosting	- **Modern Web Framework**: Developer-friendly, SSR/SSG for performance. - **Security**: Integrates easily with OAuth, multi-factor auth libraries. - **Performance**: Code splitting, caching, works well with CDN. - **Productivity**: Large community, official docs, plugin ecosystem.	3 + 3 + 3 + 9 = 12
Angular	A server-side rendering solution for Angular applications.	- **Modern Web Framework:** Provides a full-featured, opinionated framework with strong modular architecture and two-way data binding. - **Security:** Incorporates built-in security best practices and leverages TypeScript for robust, maintainable code, enhanced by Angular Universal for improved SEO and server-side performance. - **Performance:** Employs Ahead-of-Time (AOT) compilation, efficient change detection, and tree-shaking to optimize bundle sizes and loading times. - **Productivity:** Features an integrated CLI, extensive official documentation, and a rich ecosystem of libraries and tools that streamline development.	3 + 3 + 3 + 2 = 11
Nuxt.js	A framework built on Vue.js that offers server-side rendering, static site generation, and a streamlined development experience.	- **Modern Web Framework:** Seamlessly extends Vue.js with built-in support for SSR and static site generation, offering a balanced mix of flexibility and simplicity. - **Security:** Leverages Vue's reactive architecture alongside ecosystem tools for authentication and	3 + 2 + 2 + 3 = 10

		authorization to build secure applications. - **Performance:** Automatically handles code splitting, optimizes bundle sizes, and prefetches resources to ensure fast page loads and responsive interfaces. - **Productivity:** Boasts a robust module ecosystem, intuitive configuration, and strong community support that accelerates development and maintenance.	

Software Application Vendor Selection Criteria

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Active Community & Support	Must have extensive community support, frequent updates, and official documentation	3
Production Use Cases	Framework proven in large-scale production apps with strong performance	3
Integration with AWS	Must easily integrate with AWS serverless or container hosting	2

Software Application Recommended Vendors

Vendor Name	Vendor Strengths	Vendor Weaknesses	Products/Services Provided to Project	Aggregate Selection Criteria Score
Vercel (Creators of Next.js)	- Official maintainers of Next.js - Advanced edge network solutions - One-click deployments, strong dev tooling	- May incur higher costs at scale - Some advanced features proprietary	- Next.js core framework - Possibly serverless Edge Function	3 + 2 = 5
Open-	- Completely	- No official	- Next.js framework itself	2

Source Next.js (OpenNext)	free to use - Large, active GitHub community	“enterprise” support		

Software Application Deployment Challenges

Deployment Challenge	Deployment Challenge Description
SSR & Serverless Integration	Large Next.js bundles might exceed Lambda code package size—may require container-based or specialized serverless framework
Authentication & Session Handling	Must ensure secure session tokens or OAuth flows that meet banking security standards
Rapid Release Cycles	Frequent iteration demands robust CI/CD to avoid downtime or regression

Technology Adoption Methods

Method Name	Method Description
Agile Sprints	Break development into sprints (1-2 weeks). Each sprint includes Next.js feature dev, testing, stakeholder review
CI/CD Integration	Use AWS CodePipeline or GitHub Actions to automate building and deploying Next.js code on each commit

Cost/Benefit Considerations

Benefits	Costs	Considerations
Rapid dev cycle with Next.js	Development team wages	Plan for code optimization (tree shaking, minification)
Strong performance via SSR & caching	Learning curve for server components	Incorporate security best practices
Compatible with AWS	DevOps integration	Ensure ongoing alignment between Next.js releases and AWS infrastructure updates

Cloud Services Recommendations

Cloud Services Selection Criteria

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Scalability & Reliability	Must automatically handle large traffic spikes (10,000+ concurrent users) and deliver 99.99% uptime	3
Global Distribution	Ability to serve content at low latency worldwide (CDN, multi-region)	3
Security & Compliance	Built-in compliance with banking standards, encryption, and identity management	3
Integration with Next.j	Seamless or minimal-friction deployment for SSR/SSG or container-based Next.js apps	2

Cloud Services Recommendation

Recommended Software Application	Description	Benefits	Aggregate Selection Criteria Score
AWS ECS on Fargate for Next.js API	Container-based hosting; fully managed orchestration with no servers to manage. Deployed behind an ALB for global, secure access	- High Uptime: Multi-AZ, automated failover. - Security: IAM roles, private subnets, easy compliance docs. - **Integration**: Works well with AWS CodePipeline, RDS, VPC.	3 + 3 + 2 = 8
AWS Lambda (Serverless) (optional)	Alternative for serverless Next.js – auto-scale, pay-per-use. Good for microservices or certain SSR endpoints.	Event-driven scale with no server management. - Cost-effective at sporadic loads. - Security: Minimum OS patching, Per function IAM	3 + 2 + 2 = 7

Cloud Services Vendor Selection Criteria

Selection Criteria Name	Selection Criteria Description	Selection Criteria Value (Weighting in Points)
Security & Compliance	Must meet PCI DSS, SOC 2, provide advanced encryption,	3

	auditing, identity & access controls	
Integration & Ecosystem	Vendor solutions must integrate with Next.js, RDS, VPC, IAM, WAF	3
Scalability	Ability to handle concurrency bursts with minimal manual intervention	3
Cost Transparency	Clear pricing model for compute, data transfer, storage	2

Cloud Services Recommended Vendors

Vendor Name	Vendor Strengths	Vendor Weaknesses	Products/Services Provided to Project	Aggregate Selection Criteria Score
Amazon Web Services	Industry-leading compliance - Rich ecosystem (Lambda, ECS, RDS, S3, CloudFront) - Flexible cost models, pay-as-you-go	Complexity in cost management - Proprietary environment	ECS Fargate for containers - AWS Lambda (optionally) - RDS for database - CloudFront, WAF, Route 53 for global distribution & DNS	3
Microsoft Azure	- Comprehensive compliance portfolio with industry certifications - Rich, integrated ecosystem (AKS, Azure Functions, SQL Database, CDN, Front Door) - Flexible cost models and enterprise agreements	- Complexity in managing hybrid deployments - Potential vendor lock-in and cost management challenges	- Azure Kubernetes Service (AKS) for container orchestration (similar to ECS Fargate) - Azure Functions as a serverless alternative to AWS Lambda - Azure SQL Database for managed database needs - Azure CDN, Azure Front Door, Azure WAF, and Azure DNS for global distribution and security	2

Cloud Services Deployment Challenges

Deployment Challenge	Deployment Challenge Description
Multi-Region Coordination	Setting up identical infrastructure across regions for best global coverage; can be more complex in code and cost.

Vendor Lock-In	Relying heavily on AWS-specific features (e.g., ECS, Lambda) can make future migrations harder.
Monitoring & Cost Management	Ensuring CloudWatch alerts, usage dashboards, and budgeting are in place to avoid unexpected bills
Improvement #2	Add AWS budgets along with Cloudwatch dashboards to monitor resource usage and alert cost thresholds (This recommendation comes from research findings that cost overruns are a significant risk in cloud projects; real-time monitoring tools help mitigate this by offering visibility and control.)

Technology Adoption Methods

Method Name	Method Description
Phased Migration	Gradually deploy services (e.g., dev -> staging -> prod) to validate performance and cost at each step
Well-Architected Reviews	Use AWS Well-Architected Framework to review reliability, security, cost optimization, performance, and operational excellence
Improvement #3	Add structured onboarding with sandbox environments and training for developers to ensure all developers are working at the same level of technical knowledge (Research indicated that a steep learning curve could delay the project. Structured onboarding improves productivity and ensures smoother adoption of CI/CD, IaC, and SSR patterns.)

Cost/Benefit Considerations

Benefits	Costs	Considerations
Automatic scaling & patching	Potentially higher cost at scale	Evaluate usage patterns carefully to rightsize infrastructure

High reliability (SLA 99.99%)	Data transfer (egress) fees may increase with global usage	Use monitoring and budgeting tools to control spending
Global low latency with CDN	Additional cost for advanced security/compliance features (e.g., WAF)	Plan for multi-region and multi-AZ deployments to ensure redundancy and performance
Mature security & compliance features (PCI DSS, SOC 2, etc.)	Skilled personnel/training needed for specialized cloud configurations	Factor in the overhead of regular compliance audits and certifications

University of Phoenix®

Supporting Research Report

Fill out each section with information relevant to your project. Be sure to include the name and purpose of your project.

Supporting Research Report for: Kenneth Quiggins

Project name: Kentech Banking

Purpose of project:

To design and deploy a scalable, secure, and high performing financial web platform using cloud-native technologies to support modern banking needs such as secure web portals, real-time transactions, and maintain compliance with financial industry regulations.

- ## Executive Summary

The Kentech banking project aims to modernize the financial banking industry through a secure, responsive, and scalable web platform hosted in the cloud. The project adopts Amazon Web Services (AWS) for infrastructure, leverages Next.js for the frontend/backend framework, and utilizes Amazon RDS for secure, ACID-compliant relational data storage. The platform is designed to meet industry standards in security, performance, and global availability. The proposed solution replaces legacy systems with a modular, cloud-native architecture that simplifies maintenance and supports future growth.

- ## Industry Background

The banking industry has seen a dramatic shift toward digital experiences over the past decade, a trend accelerated by evolving consumer expectations and competitive pressures.

Financial institutions face increasing pressure to provide digital services with strong security and low latency. Traditional on-premises infrastructure is often too rigid and costly to scale, making cloud migration a necessary evolution. Regulatory compliance (e.g., PCI DSS, SOC 2) adds further complexity, as systems must be auditable, encrypted, and resilient. Cloud-native technologies now dominate the landscape, with providers like AWS enabling rapid deployment, security automation, and elasticity needed for modern financial operations.

Recent data from Statista showed there has been a continued decline in people attending U.S. bank branches in person. In the fourth quarter of 2024, 45 percent of U.S. bank account holders reported conducting activities in person at a branch, a decrease from 53 percent from the first half of 2019 (Newsweek, 2025).

- ## Technology Trends

The financial services industry is undergoing a technological shift that influences everything from consumer expectations to operational strategy.

Influenced by rapid digital transformations during the pandemic, banks of all sizes are migrating to cloud services. This shift helps them meet customer demands, fend off competition, enhance efficiency, and accelerate business growth (PwC, 2025).

Between 2011 and 2021, the global percentage of adults with bank accounts rose from 51% to 76%, driven by digital and cloud technologies. This transformation reduced service costs significantly, exemplified by Nubank in Brazil, which serves over 100 million customers at an average operating cost of less than $2/month (Forbes, 2025).

Larger banks, through national and global consumer brands, are well-positioned to dominate profitable sectors like consumer banking and wealth management. However, specialized local banks still play vital roles by fostering community relationships (Forbes, 2025).

- ## Project Approach

This project will follow a modern agile methodology, ensuring iterative development and early user feedback throughout the process.

The platform will be developed in sprints using agile methodology. AWS will host the infrastructure with a multi-tier VPC network, RDS for relational data, and Fargate or Lambda for application logic. The frontend/backend will be built in Next.js to support SSR and SSG, ensuring global performance and security. Deployment pipelines (CI/CD) will be integrated via GitHub Actions or AWS CodePipeline. IaC will define all resources, ensuring consistency across environments. Security will follow best practices with IAM roles, security groups, encryption, and routine audits.

- ## Alternative Approach

An alternative to this solution could involve using a Platform-as-a-Service (PaaS) like Heroku or Firebase for hosting, and a non-relational database like Firestore or MongoDB Atlas. While easier to manage initially, this approach presents limitations in ACID compliance, data structure flexibility, and vendor lock-in. Additionally, compliance with financial standards may be harder to prove without control over network and storage configurations, making it less favorable for banking applications.

.

- ## Impact Analysis

Implementing this cloud-native financial platform is expected to drive both technical and business-level benefits.

Positive Impacts:

- Enhanced user experience due to faster page loads and responsive design.
- Improved security and compliance through encryption and role-based access control.

- Reduced downtime via automated failover and multi-AZ deployments.
- Faster deployment cycles due to IaC and CI/CD.

- **Negative Impacts:**
  - Higher learning curve for teams unfamiliar with AWS and Next.js.
  - Potential for increased cost if resource usage is not optimized.

- **Risk Analysis**

High Risks:

- **Compliance Failure:** Incomplete audit trail or data exposure could violate regulations. Mitigation: Use AWS Config, CloudTrail, and KMS.
- **Cost Overrun:** Misconfigured services may lead to unexpected billing. Mitigation: Budgets, alerts, and usage reviews.
- **Performance Bottlenecks:** Improper configuration of scaling policies or DB reads. Mitigation: Load testing and performance tuning in test phases.

Medium Risks:

- **Developer Ramp-up:** Team may need training on AWS services and SSR/SSG patterns.
- **Vendor Lock-in:** Reliance on AWS-native services makes future migration more complex.

Low Risks:

- **Delays in Agile Sprints:** May occur if the feature scope is not well-defined. Mitigation: Tight backlog grooming and stakeholder reviews.

References:

Amazon EC2 Autoscaling. (2024): https://docs.aws.amazon.com/autoscaling/ec2/userguide/create-asg-launch-template.html

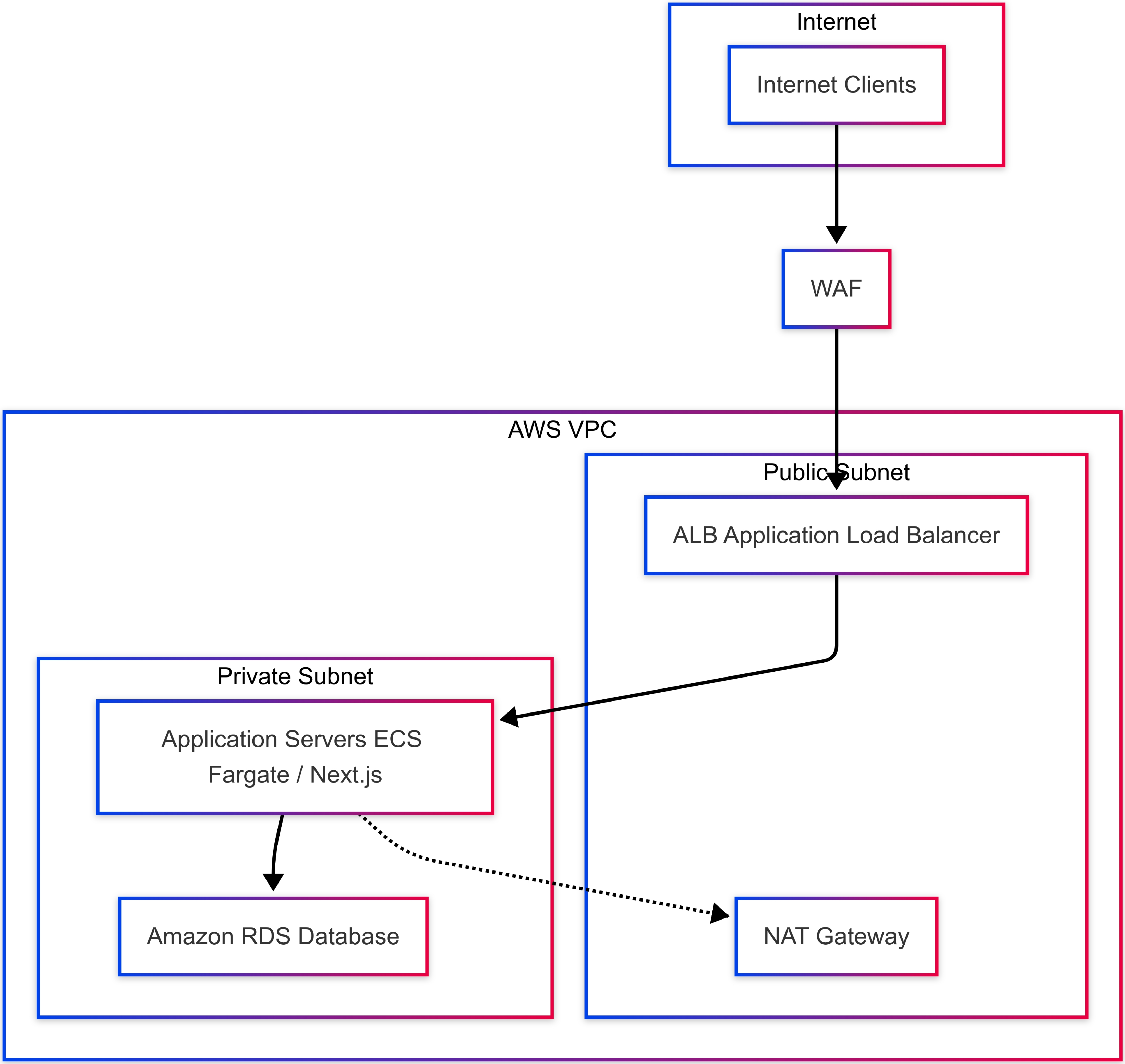
Chat-GPT. (n.d.): https://chatgpt.com

Cloud Maintenace 101. (2024): https://www.comptia.org/content/articles/cloud-maintenance-101-checking-the-pulse-of-your-cloud-technology

Continuous integration. (n.d.): The Free Encyclopedia: http://en.wikipedia.org/wiki/Continuous_integration

Simplify your way to growth through a cloud-powered bank. (2025): https://www.pwc.com/us/en/industries/financial-services/library/cloud-banking-trends.html

Top 10 Trends For Banking In 2025 – The Future Is Back. (2025): https://www.forbes.com/sites/michaelabbott/2025/01/13/top-10-trends-for-banking-in-2025--the-future-is-back/


Internet
Internet Clients
WAF
AWS VPC
Public Subnet
ALB Application Load Balancer
Private Subnet
Application Servers ECS Fargate / Next.js
Amazon RDS Database
NAT Gateway

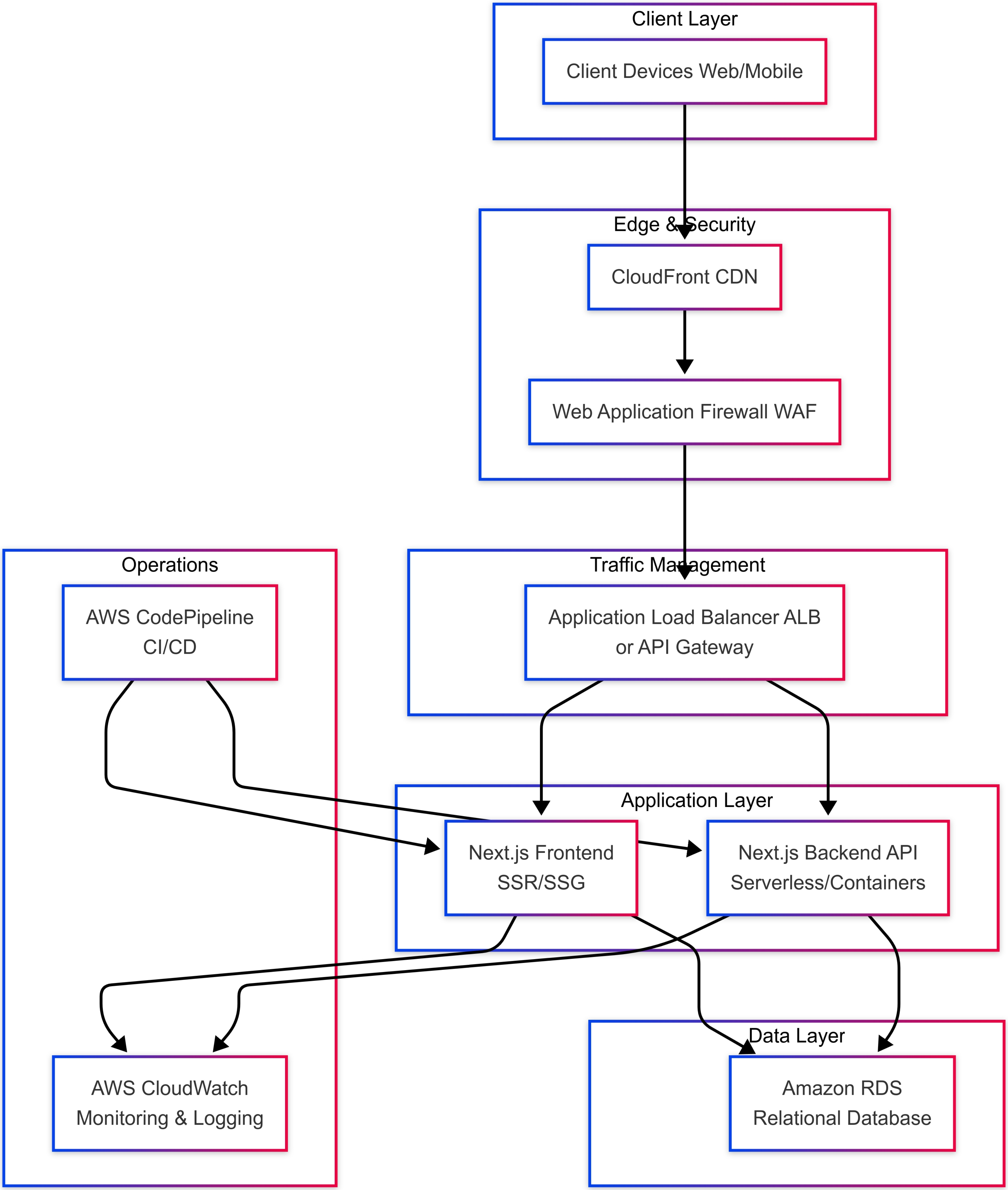

Client Layer
Client Devices Web/Mobile
Edge & Security
CloudFront CDN
Web Application Firewall WAF
Operations
AWS CodePipeline CI/CD
Traffic Management
Application Load Balancer ALB or API Gateway
Application Layer
Next.js Frontend SSR/SSG
Next.js Backend API Serverless/Containers
AWS CloudWatch Monitoring & Logging
Data Layer
Amazon RDS Relational Database

Table Definitions Document

1. USERS Table

- **Purpose**: Manages customer records and core user profiles (name, contact details, creation date).
- **Primary Key**: `user_id`
- **Foreign Keys**: None
- **Relationships**:
  - One-to-Many with **ACCOUNTS** (a user can own multiple accounts).
  - One-to-Many with **LOANS**, **INSURANCE_POLICIES**, **INVESTMENTS**, and **TAX_RECORDS** (a user can hold multiple financial products).
- **General Structure**:

  Contains user identification details (e.g., name, email) and timestamps. Acts as a central reference for linking all financial products back to the customer.

---

2. ACCOUNTS Table

- **Purpose**: Represents deposit accounts (checking, savings, money market, etc.) owned by users.
- **Primary Key**: `account_id`
- **Foreign Key**: `user_id` → **USERS**.
- **Relationships**:
  - One-to-Many with **CARDS** (an account can have multiple credit/debit cards).
  - One-to-Many with **TRANSACTIONS** (each transaction often references a deposit account).
- **General Structure**:

Tracks account type, balance, status, and the date opened. Balances are updated by relevant **TRANSACTIONS**.

3. CARDS Table

- **Purpose**: Stores credit or debit card data linked to a deposit or credit account.
- **Primary Key**: `card_id`
- **Foreign Key**: `account_id` → **ACCOUNTS**
- **Relationships**:
  - One-to-Many with **TRANSACTIONS** (if this card is used for purchases or payments).
- **General Structure**:

  Contains card number (typically encrypted), type (credit or debit), expiration, and card status.

4. LOANS Table

- **Purpose**: Captures information about loans (personal, auto, mortgage, etc.) extended to users.
- **Primary Key**: `loan_id`
- **Foreign Key**: `user_id` → **USERS**
- **Relationships**:
  - One-to-Many with **TRANSACTIONS** (loan payments or disbursements can appear in transactions).
- **General Structure**:

  Stores details such as the principal, remaining balance, interest rate, and status (current, delinquent, etc.).

5. INSURANCE_POLICIES Table

- **Purpose**: Records various insurance policies (life, auto, health, etc.) held by users.
- **Primary Key**: `policy_id`
- **Foreign Key**: `user_id` → **USERS**
- **Relationships**:
    - Typically independent but references **USERS**. May optionally relate to **TRANSACTIONS** if premium payments are tracked.
- **General Structure**:

  Includes policy type, coverage amount, monthly premium, dates (start/end), and policy status.

---

6. INVESTMENTS Table

- **Purpose**: Represents user-owned investment or portfolio accounts (brokerage, IRA, 401K, etc.).
- **Primary Key**: `investment_id`
- **Foreign Key**: `user_id` → **USERS**
- **Relationships**:
    - Typically independent but can be linked to **TRANSACTIONS** if buy/sell orders or distributions are recorded.
- **General Structure**:

  Holds the type of investment account, current portfolio value, and key dates (date opened, last updated).

---

7. TAX_RECORDS Table

- **Purpose**: Maintains tax-related information for each user, such as forms (W2, 1099) and amounts withheld.
- **Primary Key**: `tax_record_id`
- **Foreign Key**: `user_id` → **USERS**

- **Relationships**:
  - Typically independent but may combine data from **ACCOUNTS** or **LOANS** for interest reporting.
- **General Structure**:

  Includes the tax year, form type, relevant financial data (gross income, withheld), and filing date.

---

8. TRANSACTIONS Table

- **Purpose**: Logs monetary movements across deposit accounts, card purchases, or loan payments.
- **Primary Key**: `transaction_id`
- **Foreign Keys**:
  - `related_account_id` → **ACCOUNTS**
  - `loan_id` → **LOANS** (if this transaction is a loan payment)
  - `card_id` → **CARDS** (if this transaction is a card transaction)
- **Relationships**:
  - Many-to-One with **ACCOUNTS**, **LOANS**, and **CARDS** (each transaction ties to one or more of these).
- **General Structure**:
  Captures transaction type (deposit, withdrawal, payment, purchase), amount, date, description, and status.

---

9. AUDIT_LOGS Table (Optional)

- **Purpose**: Records database operations (INSERT, UPDATE, DELETE, SELECT) for compliance and auditing.
- **Primary Key**: `audit_id`
- **Foreign Key** (typical usage): `performed_by` → **USERS** (or system account)
- **Relationships**:

- Indirectly references all tables by logging `table_name` and the primary key of the affected row.

- **General Structure**:

  Documents each DML or read operation, storing the table name, operation type, date/time, and user/system performing it.

Overview of Relationships

- **USERS** is the parent table for most other entities (e.g., ACCOUNTS, LOANS, INSURANCE_POLICIES, INVESTMENTS, TAX_RECORDS).
- **ACCOUNTS** ties to **CARDS** and **TRANSACTIONS**.
- **LOANS** and **CARDS** optionally link into **TRANSACTIONS** for financial events.
- **AUDIT_LOGS** is a meta-table that can reference any row's changes for compliance.

Capstone Project

Data Dictionary

1. USERS Table

Column	Data Type	Constraints	Description
user_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the user.
first_name	VARCHAR(50)	NOT NULL	Customer’s first name.
last_name	VARCHAR(50)	NOT NULL	Customer’s last name.
email	VARCHAR(100)	UNIQUE, NOT NULL	User’s unique email address.
phone_number	VARCHAR(20)	NULLABLE	Contact phone number.
date_created	DATETIME	DEFAULT CURRENT_TIMESTAMP, NOT NULL	Timestamp of when user was created.
last_updated	DATETIME	ON UPDATE CURRENT_TIMESTAMP, NOT NULL	Timestamp of last update to the user’s record.

2. ACCOUNTS Table

Column	Data Type	Constraints	Description
account_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the account.
user_id	INT	FK → users.user_id, NOT NULL	ID of the user who owns this account.
account_type	VARCHAR(30)	CHECK (account_type IN ('CHECKING','SAVINGS', ...))	Type of account (e.g. CHECKING, SAVINGS, MONEY_MARKET, etc.).
balance	DECIMAL(12, 2)	DEFAULT 0, NOT NULL	Current monetary balance of the account.
status	VARCHAR(20)	CHECK (status IN ('ACTIVE','FROZEN','CLOSED', ...))	Current status of the account.
date_opened	DATETIME	DEFAULT CURRENT_TIMESTAMP, NOT NULL	Timestamp when the account was opened.
last_updated	DATETIME	ON UPDATE CURRENT_TIMESTAMP, NOT NULL	Timestamp of the last update to the account record.

CARDS Table

Column	Data Type	Constraints	Description
card_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the card.
account_id	INT	FK → accounts.account_id, NOT NULL	Links the card to its associated deposit or credit account.
card_number	VARCHAR(255)	Encrypted/Tokenized, NOT NULL	The card number (encrypted or tokenized).
card_type	VARCHAR(20)	CHECK (card_type IN ('CREDIT_CARD','DEBIT_CARD', ...))	Type of card (credit or debit).
expiration_date	VARCHAR(7)	NOT NULL	Expiration in MM-YYYY format (or separate month/year fields).

cvv_hash	VARCHAR(255)	Encrypted/Tokenized, NULLABLE	Encrypted or tokenized CVV (omit storing actual CVV if not permissible).
credit_limit	DECIMAL(12, 2)	NULLABLE	Max credit limit (only relevant for credit cards).
status	VARCHAR(20)	CHECK (status IN ('ACTIVE','BLOCKED','EXPIRED', ...))	Status of the card.

LOANS Table

Column	Data Type	Constraints	Description
loan_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the loan record.
user_id	INT	FK → users.user_id, NOT NULL	The user who took out the loan.
loan_type	VARCHAR(50)	CHECK (loan_type IN ('PERSONAL','AUTO','MORTGAGE','STUDENT', ...))	Type of loan.
principal_amount	DECIMAL(15, 2)	NOT NULL	Original loan principal.
remaining_balance	DECIMAL(15, 2)	NOT NULL	Current outstanding balance.
interest_rate	DECIMAL(5, 2)	NOT NULL	Annual interest rate (e.g., 5.75).
status	VARCHAR(20)	CHECK (status IN ('CURRENT','DELINQUENT','PAID_OFF', ...))	Loan status (active, delinquent, etc.).
start_date	DATE	NOT NULL	Date loan funds were disbursed.
end_date	DATE	NULLABLE	Projected or actual payoff date.
last_updated	DATETIME	ON UPDATE CURRENT_TIMESTAMP, NOT NULL	Timestamp of the last update to the loan record

5. INSURANCE_POLICIES Table

Column	Data Type	Constraints	Description
policy_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the insurance policy record.
user_id	INT	FK → users.user_id, NOT NULL	The policy holder.
policy_type	VARCHAR(30)	CHECK (policy_type IN ('LIFE','AUTO','HOME','HEALTH', ...))	Type of insurance policy.
coverage_amount	DECIMAL(15, 2)	NOT NULL	The coverage limit for the policy.
monthly_premium	DECIMAL(10, 2)	NOT NULL	Recurring premium amount due monthly.
start_date	DATE	NOT NULL	Policy effective start date.
end_date	DATE	NULLABLE	Policy expiration date, if applicable.
status	VARCHAR(20)	CHECK (status IN ('ACTIVE','CANCELED','EXPIRED', ...))	Status of the policy.
last_updated	DATETIME	ON UPDATE CURRENT_TIMESTAMP, NOT NULL	

6. INVESTMENTS Table

Column	Data Type	Constraints	Description
investment_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the investment record.
user_id	INT	FK → users.user_id, NOT NULL	Owner of the investment account.
investment_type	VARCHAR(30)	CHECK (investment_type IN ('BROKERAGE','IRA','401K','STOCK_OPTIONS'))	Type of investment or investment account.
portfolio_value	DECIMAL(15, 2)	NOT NULL	Current total market value of this investment or portfolio.
date_opened	DATETIME	NOT NULL	Timestamp when this investment account was opened.
status	VARCHAR(20)	CHECK (status IN ('ACTIVE','CLOSED','SUSPENDED', ...))	Current status of the investment account.
last_updated	DATETIME	ON UPDATE CURRENT_TIMESTAMP, NOT NULL	Timestamp of the last update to the investment record

7. TAX_RECORDS Table

Column	Data Type	Constraints	Description
tax_record_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the tax record.
user_id	INT	FK → users.user_id, NOT NULL	The user to whom this tax record belongs.
tax_year	INT	NOT NULL	The year for which these records apply (e.g., 2025).
tax_form_type	VARCHAR(20)	CHECK (tax_form_type IN ('W2','1099INT','1099DIV'))	The type of tax form relevant to the record.
gross_income	DECIMAL(15, 2)	NOT NULL	Gross income for that tax year.
tax_withheld	DECIMAL(15, 2)	NOT NULL	Amount withheld for that year.
filed_date	DATE	NULLABLE	When it was filed (NULL if not yet filed).
notes	VARCHAR(255)	NULLABLE	Additional remarks or references.

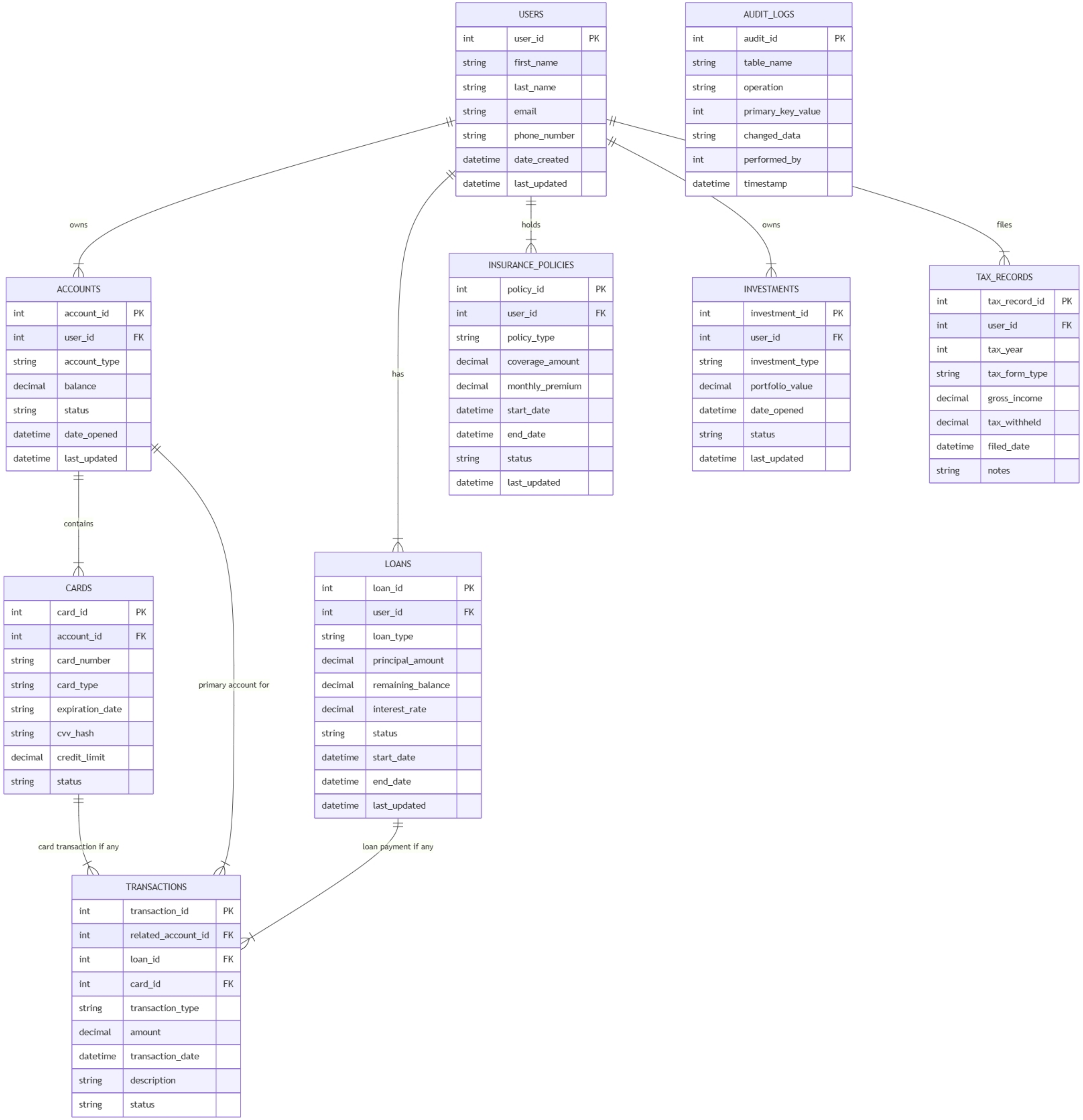
8. TRANSACTIONS Table

Column	Data Type	Constraints	Description
transaction_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for the transaction.
related_account_id	INT	FK → accounts.account_id, NULLABLE	The deposit account primarily affected by this transaction (if applicable).

loan_id	INT	FK → loans.loan_id, NULLABLE	If the transaction is a loan payment, references the associated loan.
card_id	INT	FK → cards.card_id, NULLABLE	If the transaction is a card purchase or payment, references the associated card.
transaction_type	VARCHAR(30)	CHECK (transaction_type IN ('DEPOSIT','WITHDRAWAL','PAYMENT','PURCHASE','TRANSFER',...))	Type of transaction.
amount	DECIMAL(15, 2)	NOT NULL	Monetary amount for the transaction.
transaction_date	DATETIME	DEFAULT CURRENT_TIMESTAMP, NOT NULL	The exact time the transaction occurred.
description	VARCHAR(255)	NULLABLE	Free-text description (e.g. “Mobile Deposit”, “Online Bill Pay”, etc.).
status	VARCHAR(20)	CHECK (status IN ('PENDING','COMPLETED','FAILED'))	

9. AUDIT_LOGS Table

Column	Data Type	Constraints	Description
audit_id	INT	PK (AUTO_INCREMENT), NOT NULL	Unique primary key for each audit record.
table_name	VARCHAR(50)	NOT NULL	Name of the table that was accessed or modified.
operation	VARCHAR(10)	CHECK (operation IN ('INSERT','UPDATE','DELETE','SELECT'))	Type of operation performed.
primary_key_value	INT	NULLABLE	The PK of the affected row, if applicable.
changed_data	TEXT	NULLABLE	Details on what changed (often stored in JSON).
performed_by	INT	FK → users.user_id or a system account ID, NOT NULL	Who performed the operation (or system ID).
timestamp	DATETIME	DEFAULT CURRENT_TIMESTAMP, NOT NULL	Date/time the operation occurred.



USERS		
int	user_id	PK
string	first_name	
string	last_name	
string	email	
string	phone_number	
datetime	date_created	
datetime	last_updated	

AUDIT_LOGS		
int	audit_id	PK
string	table_name	
string	operation	
int	primary_key_value	
string	changed_data	
int	performed_by	
datetime	timestamp	

ACCOUNTS		
int	account_id	PK
int	user_id	FK
string	account_type	
decimal	balance	
string	status	
datetime	date_opened	
datetime	last_updated	

INSURANCE_POLICIES		
int	policy_id	PK
int	user_id	FK
string	policy_type	
decimal	coverage_amount	
decimal	monthly_premium	
datetime	start_date	
datetime	end_date	
string	status	
datetime	last_updated	

INVESTMENTS		
int	investment_id	PK
int	user_id	FK
string	investment_type	
decimal	portfolio_value	
datetime	date_opened	
string	status	
datetime	last_updated	

TAX_RECORDS		
int	tax_record_id	PK
int	user_id	FK
int	tax_year	
string	tax_form_type	
decimal	gross_income	
decimal	tax_withheld	
datetime	filed_date	
string	notes	

CARDS		
int	card_id	PK
int	account_id	FK
string	card_number	
string	card_type	
string	expiration_date	
string	cvv_hash	
decimal	credit_limit	
string	status	

LOANS		
int	loan_id	PK
int	user_id	FK
string	loan_type	
decimal	principal_amount	
decimal	remaining_balance	
decimal	interest_rate	
string	status	
datetime	start_date	
datetime	end_date	
datetime	last_updated	

TRANSACTIONS		
int	transaction_id	PK
int	related_account_id	FK
int	loan_id	FK
int	card_id	FK
string	transaction_type	
decimal	amount	
datetime	transaction_date	
string	description	
string	status	

Cybersecurity Plan for Kentech Banking Project

Overview

This cybersecurity plan provides a structured and platform-agnostic approach to securing the Kentech Banking application, database infrastructure, and network environment. It focuses on data protection, access control, network security, compliance adherence, incident response, and ongoing monitoring applicable to any deployment platform.

Data Protection

- **Encryption:**
  - Data at Rest: Employ robust encryption standards such as AES-256.
  - Data in Transit: Utilize TLS/SSL encryption universally for data transmissions.
- **Database Security:**
  - Enforce strict data validation and constraints as defined in the data dictionary (e.g., account types, card statuses, loan statuses).
  - Implement robust database access control mechanisms using role-based access and granular permissions.

2. Access Control

- **Identity and Access Management (IAM):**
  - Adopt a comprehensive IAM strategy based on the principle of least privilege.
  - Regularly audit roles and permissions to maintain optimal security.
- **Multi-Factor Authentication (MFA):**

    - Require MFA for administrative and sensitive user operations.
- **Role-Based Access Control (RBAC):**
    - Integrate RBAC throughout the application for efficient management of user roles and permissions.

3. Network Security

- **Network Architecture:**
    - Design and implement a segmented network architecture with clearly defined public and private zones.
    - Utilize firewall rules and network ACLs to strictly control inbound and outbound traffic based on service-specific requirements.
- **Web Application Firewall (WAF):**
    - Deploy platform-agnostic WAF solutions to protect against common web vulnerabilities (OWASP top 10).
- **Load Balancing and Traffic Management:**
    - Implement secure load balancing and traffic management tools to distribute incoming traffic effectively and securely.

4. Compliance and Auditing

- **Standards Adherence:**
    - Maintain adherence to industry standards such as PCI DSS, SOC 2, and relevant regulatory requirements through continuous auditing and monitoring.
- **Automated Compliance Tools:**
    - Utilize automated compliance and security tools to continuously monitor, detect, and remediate compliance violations and misconfigurations.
    - Integrate continuous security scanning tools (e.g., OWASP ZAP, AWS Inspector) into the CI/CD pipeline to detect vulnerabilities during development.
- **Audit Logging:**

  - Maintain comprehensive audit logs to document critical actions and events, accessible for timely review and incident investigation.

5. Incident Response

- **Incident Response Plan:**
  - Develop and document a thorough incident response strategy detailing roles, responsibilities, communication channels, and escalation procedures.
  - Regularly conduct incident response drills and tabletop exercises.
  - Conduct quarterly disaster recovery drills to validate backup restoration and failover processes, documenting RTO (e.g., 4 hours) and RPO (e.g., 15 minutes).
- **Monitoring and Alerting:**
  - Implement continuous monitoring and real-time alerting systems to detect anomalies and suspicious activities proactively.

6. Continuous Monitoring and Improvement

- **Vulnerability Assessments and Penetration Testing:**
  - Schedule regular and systematic vulnerability assessments and penetration tests.
  - Quickly address and remediate identified vulnerabilities.
- **Security Training:**
  - Regularly provide security awareness training to maintain staff vigilance and knowledge of current threats and best practices.
- **Regular Reviews:**
  - Conduct regular security reviews, incorporating feedback, evolving threats, and incident outcomes into continuous improvement initiatives.

7. Risk Management

- **Risk Identification and Mitigation:**

    - Routinely review and update the project's risk register, identifying and addressing high-impact risks like compliance breaches, cost overruns, and performance issues.
- **Platform Independence:**
    - Document and implement flexible architecture principles and clearly defined migration strategies to mitigate platform dependency risks.

University of Phoenix®

Sample Software Test Plan

Project name: Kentech Banking Application

Purpose of project: To design and deploy a scalable, secure, and high-performing financial web platform that supports modern banking services, ensures secure transactions, and complies with financial industry regulations.

Features To Be Tested/Not To Be Tested

- User registration and authentication (including multi-factor authentication)
- Account creation and management (Checking, Savings, Loans)
- Transaction processing (Deposits, Withdrawals, Payments, Transfers)
- Card management (Activation, Block, Unblock)
- Loan management (Application, Approval, Payments)
- Investment account management
- Insurance policy handling
- Compliance and auditing mechanisms
- Data encryption in transit and at rest
- Performance under load
- Responsive UI and accessibility
- Security mechanisms (e.g., RBAC, audit logging)
- Third-party integration validation: Test connectivity, data exchange, and error handling for critical third-party services (e.g., payment gateways, KYC APIs).
- Disaster recovery and backup restoration (e.g., failover to secondary regions, data restoration from backups)

Features Not To Be Tested:

None. All critical third-party integrations (e.g., payment gateways, external APIs) will be tested in collaboration with vendors to ensure compatibility and reliability. Dedicated integration tests will validate end-to-end functionality**.**

Reason: This change ensures third-party integrations are included in the testing scope, addressing the risk of untested dependencies.

Testing Pass/Fail Criteria

- Functional Tests: Pass if features operate according to specifications without critical errors
- Security Tests: Pass if no vulnerabilities are detected
- Performance Tests: Pass if the system meets defined performance benchmarks under load
- Compliance Tests: Pass if the system meets regulatory compliance requirements

Testing Approach

Testing will include automated and manual methods covering unit, integration, system, and acceptance tests. The testing strategy employs agile practices with iterative testing cycles, continuous integration and delivery, and regular regression testing.

Testing Cases

1. User registration validation
2. User login/logout (including MFA)
3. Checking account creation
4. Funds transfer between accounts
5. Credit card application and activation
6. Loan application submission and approval
7. Investment portfolio view and update
8. Insurance policy creation and premium calculation
9. Data encryption verification
10. Load test for 50,000 concurrent users, including auto-scaling validation for ECS Fargate
11. Security breach simulation
12. Role-based access validation
13. Audit log integrity verification
14. Transaction processing under heavy load
15. UI responsiveness and accessibility test: Validate WCAG 2.1 compliance using automated tools (Axe, WAVE) and user testing with diverse groups (e.g., users with visual/motor disabilities).
16. Disaster recovery validation: Test backup restoration, failover to secondary AWS region, and recovery time objectives (RTO/RPO).

Testing Materials (Hardware/Software Requirements)

- Hardware: Test servers, workstations, mobile devices, sandbox environments
- Software: Testing suites (e.g., Selenium, JMeter, Postman), database clients, performance testing tools
- Tools: Jira for issue tracking, GitHub for code management, Jenkins for CI/CD, security testing tools (nmap, wireshark, nessus)

Testing Schedule

Testing Activity	Duration	Resource	Comments
Requirement Analysis	3 days	QA Lead	

Testing Activity	Duration	Resource	Comments
Environment Setup	4 days	Infrastructure team	Setup testing infrastructure
Initial Unit Testing	15 days	Development Team	
Comprehensive Integration Testing	20 days	QA Engineers	
Functional and System Testing	12 days	QA team	
Security and Compliance Testing	14 days	Security Analysts	Includes penetration testing, compliance audits (PCI DSS, SOC 2), and continuous vulnerability scanning with tools like OWASP ZAP
Performance StressTesting	10 days	QA Engineers	Test for 50,000 concurrent users, validate auto-scaling, and benchmark response times against industry standards
Accessibility Testing	5 days	End Users / Product Owners	
Final Regression Testing	6 days	End Users / Product Owners	
Disaster Recovery Testing	7 days	QA Engineers, Infrastructure Team	Test backup restoration and regional failover using AWS Backup and multi-region setups
Accessibility Testing	7 days	QA Engineers End Users	Use automated tools (e.g., Axe, WAVE) and validate against WCAG 2.1 standards, include diverse user testing

Risks and Contingencies Matrix

Risk	Probability	Risk Type	Owner	Contingencies/Mitigation Approach
Resource Availability	30%	Project Resources	QA Manager	Testing schedule will be Flexible allocation of testers and timely resource forecasting

Risk	Probability	Risk Type	Owner	Contingencies/Mitigation Approach
Technical Issues with Test Environments	20%	Infrastructure	Infrastructure Lead	Provision backup environments and allocate additional setup time
Insufficient Skills in Testing Team	15%	Project Resources	QA Manager	Conduct targeted training sessions, allocate tasks based on skills

University of Phoenix®

Project Implementation Plan Example

.

Project Name: Kentech Banking Project

Implementation Phase: Final Deployment and Go-Live

Project Plan

Task	Activity Name	Resource	Schedule Start Date	Schedule Finish Date	Schedule Comments
Prepare for Implementation	*Identify the implementation team*	Project Sponsor, IT Director	*4/20/2025*	*4/24/2025*	The team includes security analysts, QA engineers, and developers
Prepare for Implementation	Order cloud services and software subscriptions	*IT Manager*	04/21/2025	*05/05/2025*	AWS or Azure services, and third-party security subscriptions
Prepare for Implementation	Set up Identity and Access Management (IAM)	Security team	05/01/2025	05/08/2025	MFA and RBAC configured
Prepare for Implementation	Create security infrastructure	*System Admin*	*05/04/2025*	05/15/2025	Firewalls, WAF, network ACLs
Prepare test environment	Configure test servers and environments	QA and Infrastructure Teams	05/06/2025	05/12/2025	Install testing software (Selenium, JMeter)
Prepare Test Environment	Data migration to test environment	Data Specialists	05/10/2025	05/17/2025	Load data for comprehensive testing
Prepare Production Environment	Configure production servers	Infrastructure Team	05/18/2025	05/24/2025	Integration with cloud infrastructure
Prepare Production Environment	Finalize security settings	Security Analysts	05/22/2025	05/28/2025	Verify encryption and secure communications
Data Conversions	Perform final data conversions	Data Specialists	05/25/2025	06/08/2025	Data verification and integrity checks

Task	Activity Name	Resource	Schedule Start Date	Schedule Finish Date	Schedule Comments
Documentation	Complete implementation documentation	Technical Writers	05/28/2025	06/12/2025	Include user manuals, security protocols, compliance documents
Training	Provide administrator training	Trainers	06/10/2025	06/14/2025	System and security administration
Training	Provide support training	Trainers	06/12/2025	06/17/2025	Customer support and troubleshooting
Training	Provide end-user training	Trainers	06/15/2025	06/20/2025	User guides, FAQs, online sessions
Testing	Conduct final comprehensive testing	QA Team	06/18/2025	06/26/2025	Functional, security, performance, and compliance tests
Go-Live	Production Go-live	Project Sponsor, IT Director	07/01/2025	07/01/2025	Official launch and monitoring of the production environment

This implementation plan outlines the critical phases, activities, resources, and schedules for successfully deploying the Kentech Banking Project. The project is structured to ensure a smooth transition from the preparation stage through testing and finally to the production go-live phase.

Initially, the implementation team, including the project sponsor, IT director, security analysts, QA engineers, and developers, will be identified and assigned roles between April 20 and April 24, 2025. Concurrently, cloud services from AWS and Azure and necessary software subscriptions will be ordered from April 21 to May 5, 2025, managed by the IT Manager.

Security preparation begins early with the setup of Identity and Access Management (IAM), including Multi-Factor Authentication (MFA) and Role-Based Access Control (RBAC), from May 1 to May 8, 2025, overseen by the security team. Further security infrastructure setups, such as firewalls, Web Application Firewalls (WAF), and network ACLs, are planned from May 4 to May 15, 2025.

The test environment configuration, including the installation of testing tools like Selenium and JMeter, is scheduled between May 6 and May 12, 2025, by the QA and Infrastructure teams. Data migration to the test environment will follow immediately, conducted by Data Specialists from May 10 to May 17, 2025.

Preparation of the production environment, including server configurations and integration with cloud services, is scheduled for May 18 to May 24, 2025. Final security settings verification, including encryption and secure communications, will be completed between May 22 and May 28, 2025.

Final data conversions, including comprehensive verification and integrity checks, are scheduled from May 25 to June 8, 2025. Documentation required for the implementation, including user manuals, security protocols, and compliance documents, will be finalized between May 28 and June 12, 2025.

Training activities are structured into three segments. Administrator training will occur from June 10 to June 14, 2025, focusing on system and security administration. Support staff training, covering customer support and troubleshooting, is scheduled from June 12 to June 17, 2025. End-user training sessions, including the distribution of user guides, FAQs, and online sessions, will run from June 15 to June 20, 2025.

The final comprehensive testing phase, including functional, security, performance, and compliance assessments, is set from June 18 to June 26, 2025, conducted by the QA Team.

The project will officially go live on July 1, 2025, marking the deployment of the Kentech Banking application into the production environment. This phase will involve real-time monitoring to ensure smooth operation and quick resolution of any immediate post-launch issues.

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.

References

Cloud Testing – Software Testing. (2025): https://www.geeksforgeeks.org/software-testing-cloud-testing/

Jenkins User Documentation. (n.d.): jenkins.io

Selenium Testing. (2024): https://www.browserstack.com/selenium

Test Cases for Banking Applications (With Sample Test Cases). (2025): https://testsigma.com/blog/test-cases-for-banking-application/

What is Unit Testing. (2025): https://aws.amazon.com/what-is/unit-testing/

Chat-GPT. (n.d.): https://chatgpt.com