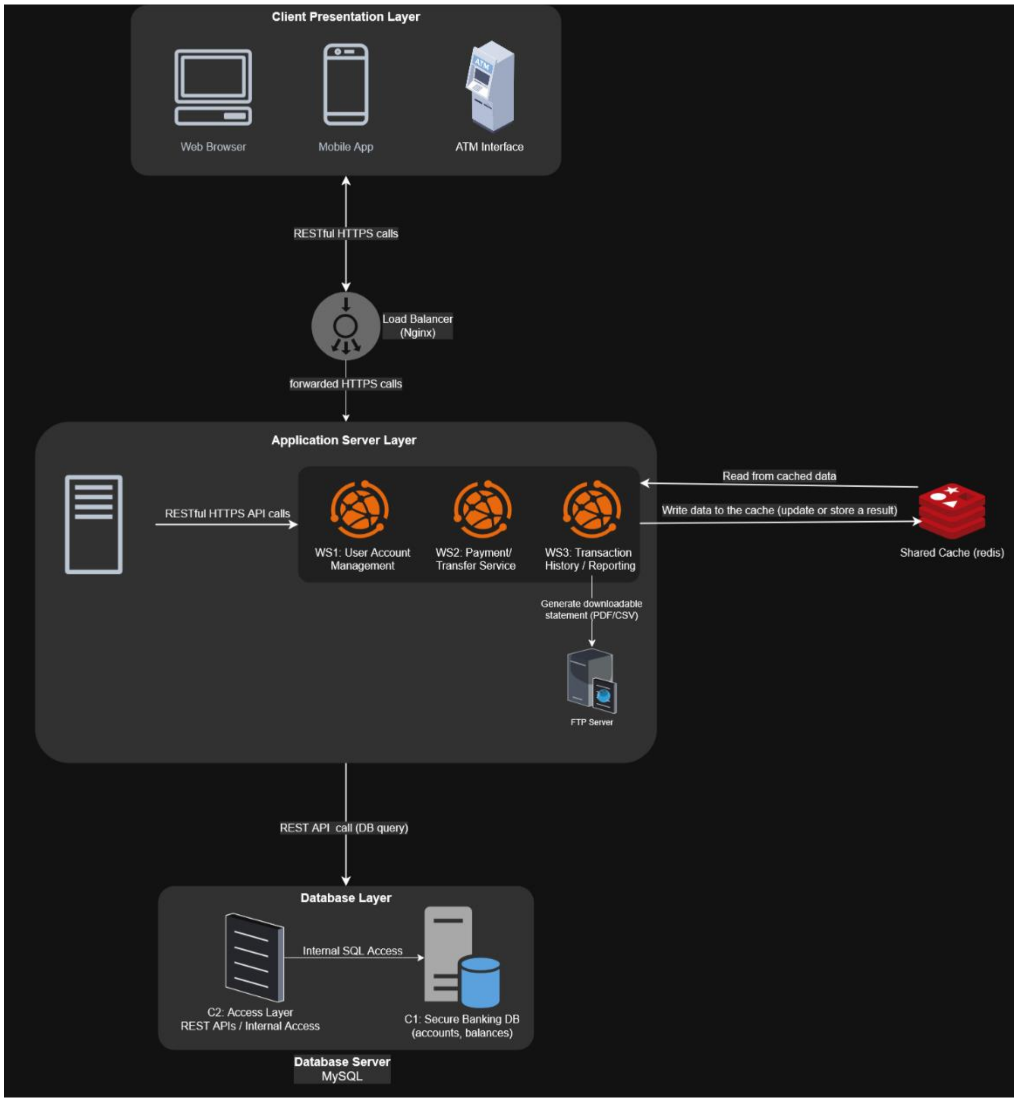
Assignment #1

Online Banking System / Digital Wallet

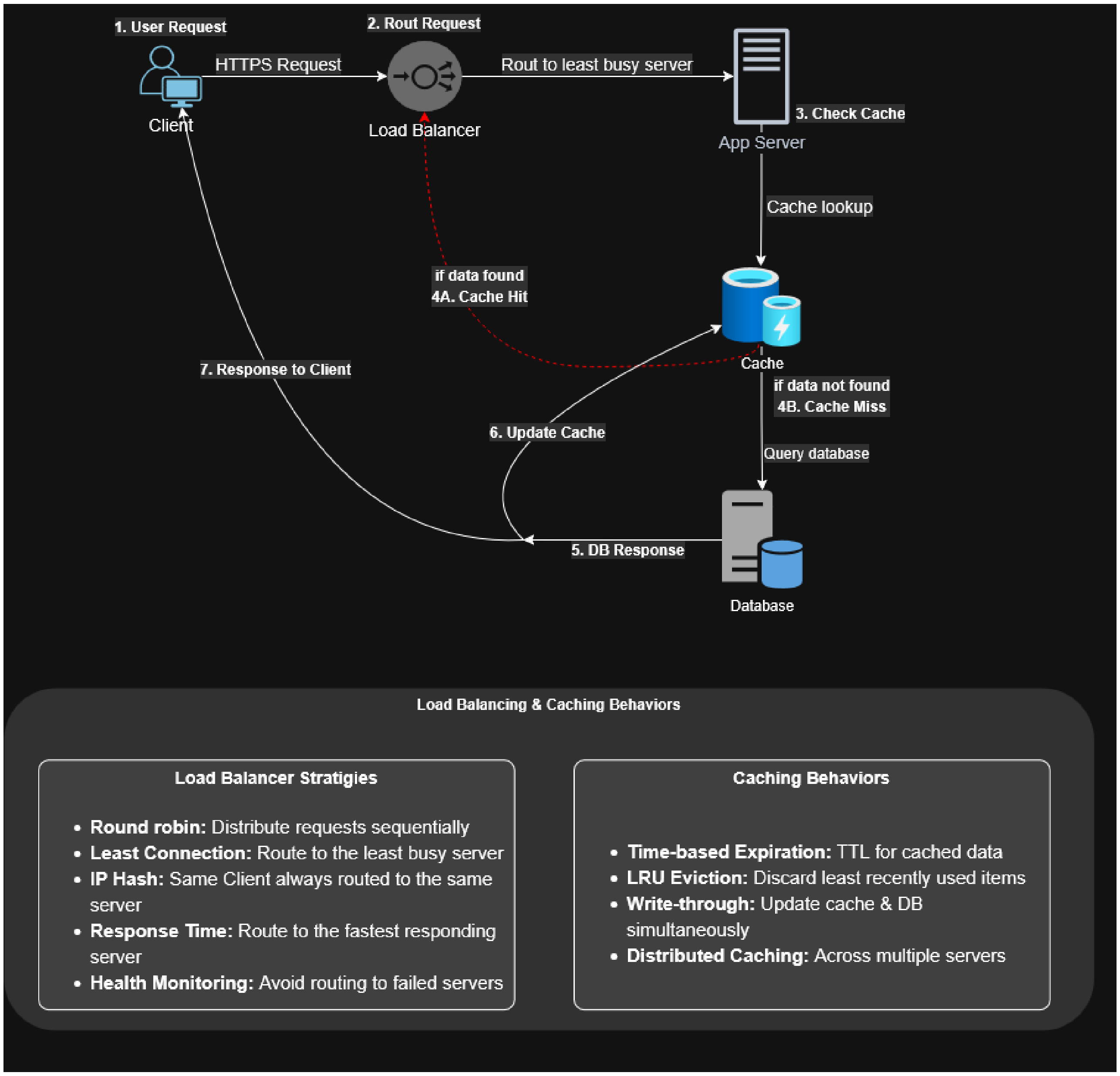
Q1: Software Architecture Design Tasks. You have to achieve the following 4 tasks with considering the applicable architecture and optimal QA

1. Client-Server Based Architecture Design:



Client Presentation Layer
Web Browser
Mobile App
ATM Interface
RESTful HTTPS calls
Load Balancer (Nginx)
forwarded HTTPS calls
Application Server Layer
RESTful HTTPS API calls
WS1: User Account Management
WS2: Payment/ Transfer Service
WS3: Transaction History / Reporting
Read from cached data
Write data to the cache (update or store a result)
Shared Cache (redis)
Generate downloadable statement (PDF/CSV)
FTP Server
REST API call (DB query)
Database Layer
Internal SQL Access
C2: Access Layer REST APIs / Internal Access
C1: Secure Banking DB (accounts, balances)
Database Server MySQL

2. Workflow Architecture: From Request to Reply



Load Balancer Stratigies

- **Round robin:** Distribute requests sequentially
- **Least Connection:** Route to the least busy server
- **IP Hash:** Same Client always routed to the same server
- **Response Time:** Route to the fastest responding server
- **Health Monitoring:** Avoid routing to failed servers

Caching Behaviors

- **Time-based Expiration:** TTL for cached data
- **LRU Eviction:** Discard least recently used items
- **Write-through:** Update cache & DB simultaneously
- **Distributed Caching:** Across multiple servers

3. Module Description

Client Presentation Layer

This is the user side of the system, people use it from phones, browsers, or ATMs. It shows their balance, lets them log in, do transactions, and download reports.

Web and Mobile Interfaces

Applications provide users with access to banking services through web browsers and mobile devices

I suggest using PHP for the web version, and Kotlin/Swift for mobile apps.

ATM Interface

Interface for ATM machines connecting to the banking system

I suggest using Java (cuz most of banking systems are implemented with java)

Load Balancer

It splits incoming traffic between app servers, so the system doesn’t crash when many people log in

I suggest using NGINX for this because it's lightweight and powerful

Caching Service

It used to store important data in memory so that i don’t keep asking the DB every second. It's mainly for stuff like user dashboards and reports that don’t change every second

I suggest using Redis for it

Application Server Layer

WS1: User Account Management

This handles login, signup, profile updates, password reset, and stuff like that

I suggest using Node.js with Express for this since it's fast for API-based stuff

WS2: Payment/Transfer Service

It’s used for processing transfers, payments, and maybe bills

Because it’s sensitive, Go is perfect for the best performance

WS3: Transaction History/Reporting

It gives users access to all their old transactions and generates reports

I suggest using Django with reporting libraries

ATM Services

It Handles the backend for ATMs like balance checking, deposits, and withdrawals

I suggest using Java because of its reliability and security libraries

FTP Service

It Generates the PDF reports and lets users download them

I suggest using SFTP server tools or AWS Transfer Family to make that secure

Database Layer

C1: Secure Banking Database

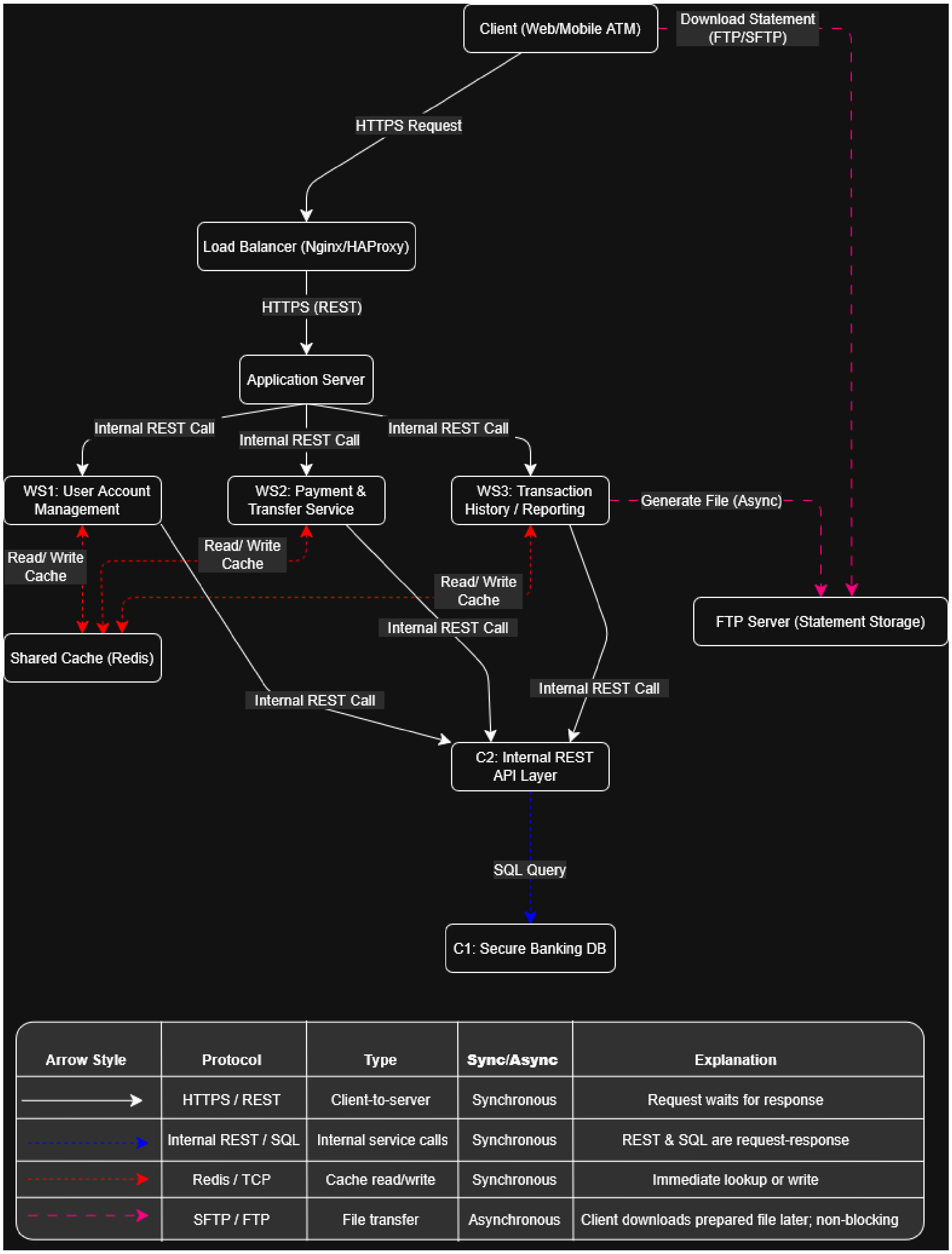
This is the main database for all core data: balances, transactions, users, etc.

I suggest using PostgreSQL with encryption, replication, and high availability

C2: REST APIs for Internal Access

This acts like a middleman between services and the database, so the DB is not exposed directly
I suggest using RESTful API frameworks like Django REST Framework

4. Communication Patterns (C&C)



Arrow Style	Protocol	Type	**Sync/Async**	Explanation
⟶	HTTPS / REST	Client-to-server	Synchronous	Request waits for response
⇢	Internal REST / SQL	Internal service calls	Synchronous	REST & SQL are request-response
⇢	Redis / TCP	Cache read/write	Synchronous	Immediate lookup or write
⇢	SFTP / FTP	File transfer	Asynchronous	Client downloads prepared file later; non-blocking

Q2: Component & Connector View – Detailed Explanation

1. Component Descriptions

Client Interface

This is what the user interacts with "mobile app, web app, or ATM screen". It sends HTTPS requests, shows account info, and handles things like PINs, transfers, and balance checks

- **Responsibilities**:
    1. Shows the UI for all banking actions.
    2. Handles input from users
    3. Displays balances and transaction history
    4. Reads cards and verifies PINs
    5. Supports cash withdraws and deposits (ATM)
    6. Lets users transfer money between accounts
- **Interfaces**:
    - **Provides**: Forms, dashboards, charts, and notifications
    - **Requires**: HTTPS connections to load balancer, SFTP to download reports

Load Balancer

It handles all incoming client traffic and spreads it out to the app servers so nothing crashes. It also checks if a server is down and avoids it

- **Responsibilities**:
    1. Splits traffic (round-robin, least connections, etc.) "shown in the workflow diagram figure"

2. Health checking for application servers
3. Terminates SSL
4. Keeps session sticky when needed
5. Filters basic DDoS attacks

- **Interfaces**:
    - **Provides**: Entry point for all clients
    - **Requires**: Connection to multiple application servers

User Account Management Service (WS1)

This service does things like login, signup, change password, and manage profile.

- **Responsibilities**:
    1. Login and authentication
    2. Create and update user accounts
    3. Reset passwords and 2FA
    4. User preference and settings management
- **Interfaces**:
    - **Provides**: REST APIs for account operations
    - **Requires**: Database connections and cache

Payment/Transfer Service (WS2)

Used when someone sends money, pays bills, or schedules transfers. It Needs to be very secure.

- **Responsibilities**:
    1. Handle account-to-account transfers
    2. Bill payments

3. Integrate with external payment systems and networks
4. Use fraud checks and validation

- **Interfaces**:
  - **Provides**: REST APIs for payments
  - **Requires**: DB API, cache, and external payment gateways

Transaction History/Reporting Service (WS3)

Used for reading old transactions and generating downloadable reports.

- **Responsibilities**:
  1. Show past transactions
  2. Filter/search history
  3. Generate monthly/annual reports
  4. Export files (PDF, CSV)
- **Interfaces**:
  - **Provides**: Transaction history API, reporting endpoints
  - **Requires**: DB API, cache, FTP service

Caching Service (Redis)

Used to store commonly used data like dashboards and reduce pressure on the DB.

- **Responsibilities**:
  1. Cache user info and dashboards
  2. Store temporary results
  3. Reduce DB queries
  4. Hold sessions if needed
- **Interfaces**:

    - **Provides**: High-speed key-value storage API
    - **Requires**: Memory and maybe sync with other caches

FTP Service

When someone asks for a report, WS3 sends it here, and the user can download it.

- **Responsibilities**:
    1. Store generated reports
    2. Let users download them safely
    3. Handle temp file cleanup
- **Interfaces**:
    - **Provides**: SFTP protocol endpoint, file management API
    - **Required**: File system access, security services

C1: Banking DB

This is the main database holding all balances, user data, and transactions.

- **Responsibilities**:
    1. Store all account data
    2. Make sure it's consistent and secure
    3. Handle transactions correctly
    4. Run backups and support failover
- **Interfaces**:
    - **Provides**: SQL query interface
    - **Requires**: Storage infrastructure and replication

C2: Internal DB API Layer

Middle layer between services and DB. It protects the DB and makes sure calls are validated.

- **Responsibilities**:
    1. Validate and manage DB access
    2. Hide direct DB structure
    3. Handle permissions
    4. Improve query performance
- **Interfaces**:
    - **Provides**: Internal RESTful API endpoints to access the data
    - **Requires**: DB access and auth service

2. Connector Descriptions

HTTPS/TLS Connector

- **Type**: Synchronous "request-response"
- **Protocol**: HTTPS (TLS 1.3)
- Used for client requests like login or checking balances "Fully encrypted"

Internal API Connector

- **Type**: Synchronous "service-to-service"
- **Protocol**: REST over HTTP or HTTPS
- Between internal services like WS1 → DB API. Might use mutual TLS or tokens for auth

Database Connector

- **Type**: Synchronous "query-response"

- **Protocol**: SQL
- Used to run queries from C2 to C1

Cache Connector

- **Type**: Synchronous “key-value operations”
- **Protocol**: Redis protocol (TSP)
- Used by services to check if the data is cached or store something new

SFTP Connector

- **Type**: Asynchronous “file transfer”
- **Protocol**: SFTP (SSH)
- Used for statement downloads. Doesn’t need a response directly “this is what async means”

3. Synchronous vs. Asynchronous Communication

Synchronous (real-time, user waits for reply):

- Client to Load Balancer → App Server (via HTTPS)
- App to WS1/WS2/WS3
- WS to Cache (check for value)
- WS to DB API (REST call)
- DB API to DB (SQL query)

Asynchronous (happens in background):

- WS3 generating reports and saving them to FTP
- Client downloading report via SFTP
- Scheduled jobs like recurring transfers or nightly backups
- Notification systems (e.g., email, SMS)

Hybrid:

Some operations start sync and finish async. For example, a transaction might show “processing” instantly (sync), but the actual settlement happens later (async), and the user gets a notification after.

Q3: Mapping Quality Attributes to Architecture

Quality Attribute	Architectural Feature Supporting It
Performance	I added Redis cache to help speed things up and reduce pressure on the database. Also async file stuff like generating reports happens in the background so it doesn't slow down users. Using DB pooling helps a lot too.
Scalability	The load balancer spreads the traffic across servers. Since each service like WS1 or WS2 is separated, I can scale each one on its own depending on what's needed.
Availability	I added things like health checks and backup servers so if something goes wrong, the system still works. The database also has replication to avoid downtime.
Security	Everything goes through HTTPS and SFTP for secure file transfer. Data is encrypted whether it's stored or being sent. I also used JWT tokens and added user roles to control access.
Modifiability	Because the system uses microservices, it's easy to update or fix just one part without breaking the rest. REST APIs make the parts loosely connected.

Quality Attribute	**Architectural Feature Supporting It**
Maintainability	The design is clean and each part has its own job. It's easy to track problems with logs and monitoring. i also followed layers so it’s easier to update stuff later.
Cost Efficiency	Caching helps lower database load so i save resources. Async tasks don't block the system and i only scale when needed. If i use containers “like docker”, it’ll be even cheaper to run and manage.