**System Design**

1. Design a Bank Credit Systems
   1. - user will apply for a credit card
   2. - user will login and see a dashboard
   3. - customer will make a payment and do transactions
   4. - user will check their outstanding balance and pay the outstanding balance

**Consider creating a flowchart.**

Assuming my role as a Full stack Software Engineer, designing a banking system requires a deep understanding of the SRS (System Requirements Specification) which include both functional and non-functional requirements. Also, i will use AWS Services to host the system.

My approach would involve several key steps:

First let me ask some Clarifying Questions to understand the scope of the system or the functional requirements:

1. What are the business users? Are they Individual(B2C) or Cooperate(B2B)?
2. Is this system intended for local or global users? This impacts regional AWS deployments
3. What are the expected peak user loads? This is essential for planning and scaling strategies.
4. What are the Recovery Time Objective (RTO) in case of regional Disasters? This defines how resilience the system is.

**Based on these, I'd prioritize critical Non-Functional Requirements:**

1. Scalability: The system mut shandle fluctuating loads by utilizing AWS services such as AWS Autoscaling which will scale horizontally to adjust and maintain the performance of the application. This will be a good choice because Auto scaling can scale in and scale out depending on the workload.
2. Availability: The system must be available 24/7 to the users. This would involve multiple AZs deployments, redundant components.
3. Resilience & Durability: The system should be able to quickly recover from failures. This can be achieved by regular backup or snapshot. Eg Creating Snapshot of EBS Volume.
4. Security: I will setup Robust security measures such as AWS WAFs(Web Application Firewall), DDoS (Distributed Denial of Services) protection (**protect network and transport layer through CloudFront and Route 53**), secure **API gateways (Sit between clients and backend services)** and Creating Identity Management Service.
5. Low Latency: Optimized data access, caching mechanisms, and efficient API design utilizing Edge location in different region.
6. Monitoring: for Monitoring, i will use AWS Cloud Watch to monitor the matrices and Log of the system.

**Architecture Design - Microservices Approach**

1. I would design the system following a Microservices architecture, which is well-suited for the complexity, scalability and well suited for the banking system. This would typically involve several independent, loosely coupled services communicating via lightweight APIs.

**The application would be structured into 3 distinct layers:**

1. Presentation Layer (Client-Side):
   1. This layer would focus on dynamic and responsive user interfaces for web and mobile. I'd utilize modern front-end frameworks like Angular or React for web applications, following a component-based approach and implementing role-based access control (RBAC) on the client-side for UI elements, alongside server-side validation. For mobile, React Native could be considered for a unified codebase across iOS and Android.
2. Application Layer (Business Logic - Microservices):
   1. This layer would host various specialized microservices responsible for distinct banking functionalities (e.g., User Service, Account Service, Transaction Service, Payment Gateway Service, Loan Service, Notification Service). This is where the AWS Lambda Function Sits.

Development platforms like Node.js (with Express.js) or Spring Boot (Java) are excellent choices, allowing for rapid development and high performance. Given Capital One's tech stack, Node.js and Java are strong fits.

Each microservice would expose RESTful APIs for communication.

An API Gateway (**Sit between clients and backend services**) would sit in front of the microservices to handle authentication, authorization, routing, caching, which will provide a single-entry point for client applications.

1. Database Layer:

Each microservice will have its own database to ensure loose coupling and service independence. The choice between relational and non-relational databases will depend on the use case:

* Relational Databases (MySQL, Amazon RDS):
  + Best for transactional data requiring strong consistency.
  + Use Cases:
    - Account Service**: Relational Integrity**
    - Transaction Service: Historical Record
    - Payment records: Reconciliation : Matching records
* Non-Relational Database (Amazon DynamoDB, MongoDB)
  + Suitable for high-throughput, flexible-schema workloads.
  + Use Cases:
    - Audit logs
    - User Service: Flexible Schema incase info changes
    - Notification data

**Testing**:

Following development, we will perform testing to ensure system integrity and reliability.

* **Integration Testing:**  
  This phase will validate the interaction between various layers of the application and confirm proper communication with external dependencies and services.
* **Deployment:**

After successful testing, I will package the application into a Docker image and publish it to Docker Hub, preparing it for deployment.

For deployment, I will use **AWS Elastic Beanstalk**, which simplifies application deployment by automating tasks such as provisioning, load balancing, auto-scaling, and monitoring

If I choose to manage the infrastructure manually, I will leverage **AWS Virtual Private Cloud (VPC)** to create a secure and scalable environment.

**VPC Creation and IP Allocation:**

I would begin by creating a Non-Default VPC (Either AWS Console or AWS CLI) and Allocate IP Addresses following Standard CIDR(Classless Inter domain routing).

* Subnets will be distributed across multiple **Availability Zones (AZs)** for high availability. A common practice is to distribute the Ip addresses across public and private subnets.
* Public Subnets:
  + Connected to an **Internet Gateway (IGW)** to allow internet access.
  + You can host your webserver in Public Subnet:
* Private Subnets:
  + Isolated from direct internet access for security.
  + Hosts **application servers and databases**, where core logic and sensitive data reside.
  + **Network Access Translation: NAT Gateway** (in public subnet) allows outbound internet access for software updates or third-party API calls.

Question How AWS Lambda function work?

1. Search for AWS Lambda
2. Create a function
3. Select the Runtime

Question 2: How do you configure Route 53 for bank.com to direct to your AWS Container instances?

1. Create a Hosted Zone in Route 53 for bank.com (this will create NS records)
2. Update Domain Registrar (e.g., GoDaddy) by replacing the default name servers in the registrar's DNS settings with the NS records from Route 53.
3. Deploy your containerized app (ECS/EKS) behind an ALB (Application Load Balancer) ALB should forward traffic to these targets.
4. Create a Record Set (A/AAAA or CNAME) in Route 53 and set Alias Target to the ALB's DNS name. This maps bank.com to your Load Balancer, which routes to containers.

This routes user traffic from bank.com to your containerized application through the load balancer.

**Example Flow: bank.com → ECS App**

1. User types bank.com
2. DNS resolver queries Route 53
3. Route 53 replies with ALB's DNS
4. User Browser connects to ALB
5. ALB forwards request to ECS container
6. Response returned to the user

Question 3

**When designing, what comes to your mind, what are the deliverables as an Architect? What are you delivering to your development team?**

When designing a system, I think about **scalability, reliability, security, performance, and maintainability** from day one. My goal as an architect is to align the **technical solution with business goals**, while enabling the development team to build efficiently and confidently.

What Comes to Mind:

* SRS or Functional requirements and Non-Functional Requirements
* Tech stack selection
* Service boundaries and interactions
* Data flow and storage
* Security and compliance
* Disaster recovery and scaling

**Key Deliverables to the Development Team:**

1. **High-Level Architecture Diagram using software tools like D2 diagram which is very useful to create Flowchart.**
   * Components, services, interactions, and deployment layers
2. **System Design Document with user stories.**
   * Covers modules, data flow, API contracts, security considerations, and infrastructure choice
3. **API Specifications**
   * REST request/response formats, status codes
4. **Infrastructure Design**
   * VPC layout, networking, load balancers, storage, monitoring (e.g., via Terraform or CloudFormation)
5. **Non-Functional Requirements (NFRs)**
   * SLAs, RTO/RPO, scaling plans, latency targets, compliance
6. **DevOps Strategy**
   * CI/CD pipeline guidance, containerization standards, environment setup (dev, stage, prod)
7. **Security Model**
   * IAM roles, encryption, audit logging, data access controls

**Question 3.**

**How do you incorporate SLA (Service Layer Agreement: Also known as the application layer) requirements from networks like Visa/Mastercard (e.g., 500ms or 200ms)?**

To incorporate **SLA requirements** from networks like **Visa/Mastercard** (e.g., response times under **200–500ms**), I follow a performance-first design strategy across all layers:

**🔧1. Architectural Design**

* Use **low-latency, high-throughput services** (e.g., **AWS ALB + ECS/EKS**, **gRPC** for internal communication)
* Design for **asynchronous processing** where possible (e.g., message queues) to decouple time-sensitive paths

**2. Data Optimization**

* Use **in-memory caches** (e.g., **Redis** via Amazon ElastiCache) for frequent lookups (like card BIN data)
* Minimize DB roundtrips in hot paths; use read replicas and optimized indexes

**🌐 3. Network Efficiency**

* Host services in **low-latency regions**, close to payment network endpoints
* Leverage **VPC endpoints** for secure, fast communication with AWS services

**🔍 4. Monitoring & Alerting**

* Use **AWS CloudWatch** or **Datadog** to monitor latency at each tier
* Set alerts when thresholds approach SLA limits

**🧪 5. Testing & Validation**

* Perform **load testing** (e.g., with Gatling or k6) to validate response times under peak loads
* Profile and tune code paths and DB queries to meet strict latency budgets

Question 4

**How do you improve the performance of your database?**

1. **Strategic Indexing:** Create optimized indexes on frequently queried columns.
2. **Query Optimization:** Tune slow queries, avoid SELECT \*, use efficient joins.
3. **Caching:** Implement in-memory caches (ElastiCache/Redis) for frequently accessed data.
4. **Scaling:** Utilize **read replicas** (Aurora) for read scalability, and potentially **sharding/partitioning** for very large datasets.
5. **Efficient Connections:** Use connection pooling from the application.
6. **Resource Provisioning:** Ensure adequate CPU, memory, and IOPS.

**What kind of language are you going to use for writing your Lambda function?**

I'd primarily use **Node.js** for Lambda functions.

**Why:** Because it’s fast, lightweight, and starts quickly. It works well with AWS, especially for tasks that run in response to events like API calls or file uploads. It's also good at handling asynchronous operations. This makes it ideal for APIs and event-driven microservices. Python is great for data-heavy tasks, and Java for high-performance CPU-bound tasks, but Node.js offers a strong balance for most serverless use cases.

Short Questions:

Stateless: The server does not store or remember the client session information between requests. Each request is independent. Eg: REST APIs, AWS Lambda

Stateful: The server **remembers previous interactions or session**. State is maintained across requests. WebSockets

What are RESTFUL APIs: A REST API (Representational State Transfer Application Programming Interface) is a type of API that allow applications to communicate with each other over the internet. It's sends and receives data between a client and a server using standard HTTP methods such as GET, POST, PUT, DELETE.

Synchronous: Tasks happen **one after another**, waiting for each to complete eg: API request → waits for response

**Asynchronous**: Tasks run independently, without waiting. Eg: Message Queues, Promises

**Synchronous = Waits** — blocks execution until the task finishes.

**Asynchronous = Doesn’t wait** — keeps going and deals with the result later using promise or callback functions.

Common Status Codes by Category

|  |  |  |
| --- | --- | --- |
| Code | Message | Meaning |
| 200 | Ok | Request Success |
| 201 | Created | Resource created (POST) |
| 400 | Bad Request | Invalid |
| 404 | Not Found | Resource not found |
| 500 | Internal Server Error | Server Error. |

**Additional Details**

**AWS Services:**

1. S3 Bucket: Amazon simple storage service is an Object storage that is used to store and retrieved any amount of data at any given time. The end-user can access the content using the API endpoint or the object URL.

How to create S3 bucket.

* Search S3 > give bucket name > Set the Object Ownership > configure public access settings > create bucket.
* Click on the bucket > Under object > click on upload > select the image you want to upload > set the permissions and configure public access > click upload
* Use the object url as the API endpoint to access the object

**Note: !!!**

**How do you provide the Object URL more securely to the end-user?**

After creating the S3 Bucket, it is not ideal to provide the URL to the end-user if the bucket contain more sensitive information. Because the object url contains the bucket name, region where the bucket is created and the file name.

For sensitive information, you configure the public access by enabling “block public access” create a cloud front distribution and map the distribution to the S3 Bucket.

Any user that invokes the cloud front url, will be redirected the request to the S3 Bucket.

**How do you create a custom url?**

To create a custom url, you used Route 53.

1. VPC: Is it a networking Visualization that provides logical networking environments within enterprises.
   1. When you create an EC2 instance, AWS Allocate a default VPC with the CIDR that contain the IP address range.
   2. Non-Default VPC is the vpc that is created by the customer. The customer will have to create resources like subnet, Security groups and Allocate IP addresses.

Note:!!!

VPC is specific to region.

Region has availability zones.

Subnet is specific to availability zones.

Route Table: This is where you define the rules for your traffic flows.

Public Subnet and Private Subnet.

To create an Enterprise Application:

1. Create a non-default VPC > Base on the CIDR on the Non-Default VPC, calculated IP addresses and allocate them to the EC2 instances.
   1. Eg: If you derived 64 ip addresses from the CIDR, you allocate 32 ip addresses to public subnet and 32 to the private subnet
   2. Create a CIDR for the Public subnet and Private Subnet
   3. Note: !!
      1. Public Subnet: Host your web server on the public subnet
      2. Private Subnet: Host your application server

Questions:

**Why do you host your web server to the public subnet and App server to the private subnet?**

I will host the webserver on the public subnet because i want the end user to have access to it over the internet.

I will host the app server on the private subnet because it contains private information and wants more restriction.

Database

1. **How do you handle database design and optimization?**

One of the way i can handle database design is by optimizing DB queries

1. In**dexing** key fields for faster lookup.
   1. For MongoDB, to index the user\_email field, first define it in the schema, and then create an index on that field using the schema’s. index () method. This will help improve database query performance in real-world projects.

const userSchema = new mongoose.Schema({

name: String,

user\_email: {type: String, unique: true} // Define the field

});

// Create an index on the user\_email field

userSchema.index({ user\_email: 1 });

const User = mongoose.model("User", userSchema);

* 1. In SQL: I have used **Non-Clustered Indexes** to speed up queries. For example, if i frequently query a **user’s** table by email, I would create an index like this:

CREATE NONCLUSTERED INDEX idx\_users\_email → Creates a **non-clustered index** named idx\_users\_email.

ON Users (Email) → Specifies that the index is created **on the Email column** of the Users table.

This helps optimize **SELECT queries** when filtering users by email.