CVS Staff Engineer Interview Questions

Code-Included

# Medium Difficulty Questions (Code-Enhanced)

## Technical – API Orchestration (with Code)

**Question:**

* Write a Node.js Express route handler that orchestrates calls to two backend services: /user-profile and /medications. The /medications endpoint is slower (2s latency). How would you parallelize these calls and handle errors?

**Sample Expected Approach:**

app.get('/aggregate', async (req, res) => {

 try {

   const [profile, meds] = await Promise.all([

     axios.get('<https://api.cvs.com/user-profile>'),

     axios.get('<https://api.cvs.com/medications>')

   ]);

   res.json({ profile: profile.data, medications: meds.data });

 } catch (err) {

   res.status(500).send('Aggregation failed');

 }

});

**Bad Answer:**

* Look for sequential fetches each endpoint;
* no error handling; ignores latency issues.

**Follow-up:**

* How would you handle partial failures (e.g., one API fails, the other succeeds)?

## Procedural – Technical Debt

**Question:**

* Describe technical debt and how do you identify and track technical debt in your codebase? What tools or metrics would you use in a Node.js microservices architecture?

**Good Answers:**

* Use **SonarQube**, **ESLint**, and custom scripts to flag anti-patterns.
* Track debt in Jira, tie it to sprints, and prioritize via impact.
* Measure code complexity, test coverage, and cyclomatic complexity.

**Bad Answer:**

* Says “we’ll fix it when we can” or “just rely on code reviews.”

**Follow-up:**

* Can you describe an example of a technical debt story you’d create in a backlog?

## Interpersonal – Code Review Scenarios

**Question:**

* A junior developer submits a PR introducing a global variable for shared state in a Node.js service. What do you do?

**Good Answers:**

* Review PR, explain risks (e.g., race conditions, scalability).
* Suggest alternatives (e.g., dependency injection, request-scoped context).
* Provide code examples in comments.

**Bad Answer:**

* Anything that sounds like “just accept it” or “just ignore it” without mentoring.

**Follow-up:**

* How would you characterize the difference between module-scoped state and request-scoped state in Node.js?

# Difficult Questions (Code-Enhanced)

## Scalability – AWS Lambda and DynamoDB (with Code)

**Question:**

* Write or describe a Lambda function in Node.js that handles a prescription refill event and stores it in DynamoDB with conditional writes to prevent duplicates.

**Sample Expected Code:**

app.get('/aggregate', async (req, res) => {

 try {

   const [profile, meds] = await Promise.all([

     axios.get('<https://api.cvs.com/user-profile>'),

     axios.get('<https://api.cvs.com/medications>')

   ]);

   res.json({ profile: profile.data, medications: meds.data });

 } catch (err) {

   res.status(500).send('Aggregation failed');

 }

});

**Bad Answer:**

* No conditional writes; risk of duplicates; no error handling.

**Follow-up:**

* How would you scale this Lambda if refill events spike 1000x?

## Auth Mechanisms (Code-Oriented)

**Question:**

* Write or describe a function in Node.js to validate a JWT token for the SuperApp API using the jsonwebtoken library.

**Sample Expected Code:**

const jwt = require('jsonwebtoken');

function validateToken(token) {

 try {

   return jwt.verify(token, process.env.JWT\_SECRET);

 } catch (err) {

   throw new Error('Invalid token');

 }

}

**Bad Answer:**

* Forgets to validate token expiration; ignores exceptions.

**Follow-up:**

* Why not just use API keys instead of JWTs here?

## Latency – Code Optimization

**Question:**

* Given the following code, what’s wrong with it, and how would you optimize it for latency?

app.get('/user-data', async (req, res) => {

 const profile = await axios.get('/profile');

 const history = await axios.get('/history');

 const preferences = await axios.get('/preferences');

 res.json({ profile, history, preferences });

});

**Good Answer:**

* Parallelize API calls using Promise.all() to reduce overall latency.

**Bad Answer:**

* Leaves sequential calls, no batching, no error handling.

**Follow-up:**

* How would you further optimize this if one API is much slower?

## Multi-Client Logic Handling

**Question:**

* How would you design API responses to support both mobile and web clients with different data needs without duplicating logic?

**Good Answers:**

* Use query parameters or headers to control response shape.
* Implement GraphQL or Experience APIs.
* Modularize response logic.

**Bad Answer:**

* Duplicate endpoints per client; bloats backend.

**Follow-up:**

* Can you write or describe an example of a controller method that adapts response shape based on client type?

## Inter-Org Collaboration

**Question:**

* CVS uses GraphQL, Aetna uses REST APIs. How do you integrate their systems for the SuperApp backend?

**Good Answers:**

* Use a **GraphQL federated gateway** that wraps REST endpoints.
* Build **API mediators** with caching and schema mapping.
* Define a canonical data model.

**Bad Answer:**

* Anything that shows a lack of understanding of the issue or that the engineer has not dealt with this problem in the past

**Follow-up:**

* Would you normalize data at the API layer or the data layer? Why?

# Past Experience – Code-Centric

**Question:**

* Describe a backend system you built with complex business logic and multiple clients. What were the hardest parts to code, and how did you test your solution?

**Follow-up:**

* What would you refactor today if you revisited that system?

# Really Hard Design Question – API + Real-Time Requirement

**Question:**

* You’ve built a Node.js API for prescription refills using REST. Now you must add **real-time status updates to mobile clients** when refill status changes. Describe your updated architecture and show code for emitting events.

**Expected Answer:**

* Add DynamoDB Streams or SNS topics for events.
* Use WebSockets (API Gateway) or push notifications via SNS Mobile Push.
* Example code for SNS publish:

const AWS = require('aws-sdk');

const sns = new AWS.SNS();

async function notifyStatusChange(userId, status) {

 const params = {

   Message: JSON.stringify({ userId, status }),

   TopicArn: process.env.SNS\_TOPIC\_ARN,

 };

 await sns.publish(params).promise();

}

**Bad Answer:**

* Polls the API repeatedly; no event-driven design.

**Follow-up:**

* How would you guarantee delivery and handle retries?