Interview Model Answers

Intern & Associate Software Engineer - Muve Mobility (Pvt) Ltd, Kandy
Candidate: Hashini Tharuka

This guide gives a model answer and an interviewer note for each of the 30 questions: 10 basic, 10 medium and 10 advanced.

How to use it. The model answers show what a strong response covers; they are not scripts. Accept different wording as long as the key ideas and trade-offs are present, and ask follow-up questions that reference the candidate's own projects.

Scoring suggestion. Rate each answer 0-3: 0 = no understanding, 1 = partial or memorised, 2 = correct with an example, 3 = correct with trade-offs and real experience. Basic and medium questions carry most of the weight for an intern or associate role; treat the advanced set as a stretch.

Known gaps to probe fairly. The CV does not mention Flutter, Kubernetes, Angular, Kafka or RabbitMQ. The job post lists some of these as an advantage, so judge the learning approach rather than expecting mastery. Spring Boot appears in the skills list but not in the experience section, so ask where it was used.

Level	Questions	Focus
Basic	B1 - B10	Fundamentals: OOP, REST, SQL/NoSQL, Git, JS, Docker, Agile, Spring Boot
Medium	M1 - M10	Spring layers, JWT, microservices, events, React, CI/CD, indexes, Flutter
Advanced	A1 - A10	Sagas, Kafka, Kubernetes, performance, security, RAG, system design, CAP

Basic Questions

Fundamentals every intern or associate should be comfortable with.

B1 Tell me about yourself and why you want to join Muve Mobility as a software engineer.

A strong answer is a 1-2 minute story in four parts:

- **Present:** Computer Engineering graduate (University of Ruhuna, 2026) focused on full-stack development, APIs and applied AI.
- **Experience:** six-month Software Engineering internship at SLT (AI-assisted bug categorisation, an ASP.NET + SQL Server maintenance system, TM Forum-compliant REST APIs).
- **Projects:** one or two highlights, such as Edumind or the hospital bed allocation platform, with the impact they had.
- **Why Muve:** interest in mobility and real-world systems, the Java/Spring Boot/microservices stack, learning from a production team, and willingness to work on-site in Kandy.

Interviewer note: Concise, tied to the role, and honest about gaps (for example, Flutter and Kubernetes). Avoid reciting the CV line by line.

B2 What are the four pillars of OOP? Give a real example of each from your own code.

- **Encapsulation:** bundle data with the methods that operate on it and hide internals. Example: private fields with validated setters so a balance can never go negative.
- **Inheritance:** a subclass reuses and extends a parent class. Example: *Car* and *Bike* extend *Vehicle*.
- **Polymorphism:** one interface, many behaviours. Example: *PaymentMethod.pay()* implemented differently by Card and Wallet; the caller does not care which.
- **Abstraction:** expose what something does, hide how. Example: a repository interface hides whether data comes from MySQL or MongoDB.

Interviewer note: The candidate should use examples from their own projects instead of textbook Animal/Dog examples.

B3 What is the difference between an abstract class and an interface in Java?

- An **abstract class** can hold state (instance fields), constructors, and both abstract and concrete methods. A class can extend only one.
- An **interface** defines a contract. A class can implement many. Since Java 8 it can have *default* and *static* methods; its fields are implicitly *public static final*.
- **Rule of thumb:** use an abstract class for a shared base with common state and behaviour (an "is-a" relationship); use an interface for a capability that unrelated classes can share (for example *Comparable*, *Serializable*).

Interviewer note: Mentions multiple inheritance of type, default methods, and a concrete use case.

B4 What is a REST API? Explain GET, POST, PUT, PATCH and DELETE with examples.

REST is an architectural style where resources are identified by URLs and manipulated through standard HTTP methods. It is stateless: each request carries everything the server needs.

- **GET /vehicles/12**: read a resource. Safe and idempotent.
- **POST /vehicles**: create a resource. Not idempotent. Returns 201 Created.
- **PUT /vehicles/12**: replace the whole resource. Idempotent.
- **PATCH /vehicles/12**: partially update (for example only the status).
- **DELETE /vehicles/12**: remove it. Idempotent. Returns 204 No Content.

Good answers also cover status codes: 200, 201, 204, 400, 401, 403, 404, 409 and 500.

Interviewer note: Understands idempotency, status codes, and statelessness. Candidate built TM Forum REST APIs, so expect depth.

B5 What is the difference between SQL and NoSQL databases? Why did you choose MongoDB for some projects and MySQL for others?

- **SQL (relational):** fixed schema, tables and joins, strong ACID transactions. Best for structured, relational, transactional data.
- **NoSQL (for example MongoDB):** flexible document schema, nested data, easy horizontal scaling. Best for evolving or semi-structured data.
- **Application:** MongoDB suited Edumind and the resume-matching platform (flexible documents plus Atlas Vector Search for embeddings). A relational database such as SQL Server or MySQL suits data with strict relationships and transactions, like the maintenance system with stored procedures and triggers.

Interviewer note: Choice is justified by data shape and consistency needs, not by "MongoDB is easier". It is not an either/or: many systems use both.

B6 What is Git? Explain branching, merging, and how you resolve a merge conflict.

Git is a distributed version control system. A **branch** is an isolated line of work (for example *feature/login*); a **merge** combines branches. A **conflict** happens when both branches changed the same lines.

To resolve it:

- Run *git status* to see conflicted files and open them.
- Choose or combine the correct code and delete the markers.
- Run *git add* on the file, then *git commit* (or *git merge --continue*).
- Build and run the tests, and talk to the teammate if the intent is unclear.

```
<<<<<<< HEAD
your version of the lines
=======
incoming version of the lines
>>>>>>> feature/login
```

Interviewer note: Mentions pull requests, rebase vs merge, and small frequent commits as ways to avoid conflicts.

B7 What is the difference between let, const, and var in JavaScript?

- **var:** function-scoped, hoisted (initialised as undefined), can be redeclared.
- **let:** block-scoped, can be reassigned, not usable before its declaration (temporal dead zone).
- **const:** block-scoped and cannot be reassigned. The contents of an object or array declared with const can still be mutated.

Modern practice: default to *const*, use *let* when reassignment is needed, avoid *var*.

Interviewer note: Knows about hoisting, the temporal dead zone, and the const-object nuance.

B8 What is Docker, and why do developers use it?

Docker packages an application with its dependencies into a **container** that runs the same way on any machine. A **Dockerfile** describes how to build an **image**; a container is a running instance of an image.

- **Why:** consistent environments ("works on my machine" disappears), fast startup, easy CI/CD, and isolation for microservices.
- **Versus VMs:** containers share the host kernel, so they are lighter and faster than full virtual machines.

- **Docker Compose** runs multi-container setups (app + database) locally.

> **Interviewer note:** Distinguishes image vs container and container vs VM. Candidate lists Docker, so ask what they actually containerised.

B9 What is Agile/Scrum? What was your role in a team using it?

Agile is an iterative approach that delivers working software in small increments and adapts to feedback. **Scrum** is a framework for it:

- **Roles:** Product Owner, Scrum Master, Development Team.
- **Events:** sprint (1-4 weeks), sprint planning, daily stand-up, sprint review, retrospective.
- **Artifacts:** product backlog, sprint backlog, increment.

The candidate should then describe their own experience (SLT internship, university projects, Jira boards) with specifics such as how tasks were estimated and how blockers were raised.

> **Interviewer note:** A real example of working in sprints, not only definitions.

B10 You list Spring Boot in your skills. What does it do for a Java developer, and what is a basic Spring Boot project structure?

Spring Boot is an opinionated layer on top of Spring that removes boilerplate: **starters** bundle dependencies, **auto-configuration** wires beans sensibly, an **embedded server** (Tomcat) lets you run the app as a plain JAR, and **Actuator** adds health and metrics endpoints.

A typical layout:

```
com.muve.app
  Application.java      (@SpringBootApplication)
  controller/           (REST endpoints)
  service/              (business logic)
  repository/           (Spring Data JPA)
  model/ or entity/     (domain classes)
  dto/                  (request/response objects)
  config/ exception/    (security, error handling)
resources/application.yml
```

> **Interviewer note:** NOTE: the CV shows Node/Express at SLT and Spring Boot only as a listed skill. Probe how deeply it was used (a coursework project?).

Medium Questions

Applied knowledge: frameworks, architecture and practical trade-offs.

M1 In Spring Boot, explain dependency injection, @RestController, @Service, and @Repository. How do they work together in a layered architecture?

Dependency injection: the Spring container (IoC) creates objects (beans) and supplies them to the classes that need them, instead of those classes calling *new*. Constructor injection is preferred because it makes dependencies explicit, supports immutability and eases unit testing.

- **@RestController:** handles HTTP requests and returns JSON (a @Controller combined with @ResponseBody).
- **@Service:** holds business logic and transaction boundaries (@Transactional).
- **@Repository:** data-access layer; also translates database exceptions into Spring exceptions.

Flow: **Controller -> Service -> Repository -> Database**, with DTOs at the edge so entities are not exposed directly.

```
@RestController
@RequestMapping("/api/vehicles")
class VehicleController {
  private final VehicleService service;
  VehicleController(VehicleService service) { this.service = service; }
}
```

Interviewer note: Constructor injection, separation of concerns, and DTO vs entity awareness.

M2 What is the difference between HashMap, ConcurrentHashMap, and TreeMap in Java? When would you use each?

- **HashMap:** average O(1) get/put, unordered, allows one null key, **not thread-safe**. Default choice in single-threaded code.
- **ConcurrentHashMap:** thread-safe with fine-grained locking/CAS, so reads and writes scale under concurrency. No null keys or values. Use for shared caches and counters.
- **TreeMap:** red-black tree, keys kept **sorted**, O(log n) operations, supports range queries (*subMap*, *floorKey*). Use when ordering matters.

Bonus: *Collections.synchronizedMap* locks the whole map and performs worse than ConcurrentHashMap under contention.

Interviewer note: Mentions thread safety, null handling and complexity.

M3 Explain how JWT authentication works. How did you implement role-based access (student/instructor/admin) in Edumind?

A JWT has three Base64URL parts: **header.payload.signature**. After login the server signs a token containing claims (user id, *role*, *exp*). The client sends it as *Authorization: Bearer ...*; the server verifies the signature and expiry on each request without storing a session.

Role-based access: put the role in the token, then a middleware/guard checks the role against the route (for example only instructors can create quizzes, only admins manage users). Validate input (for example with Zod) on top of that.

Security points: the payload is encoded, **not encrypted**, so never put secrets in it; use short-lived access tokens with refresh tokens; use HTTPS; prefer httpOnly cookies over localStorage to reduce XSS risk; always check authorisation on the server, not only in the UI.

Interviewer note: Candidate describes their own middleware and token storage choice, and mentions token expiry and revocation trade-offs.

M4 What is a microservices architecture? What are its advantages and drawbacks compared to a monolith?

Microservices split an application into small, independently deployable services, each owning one business capability and usually its own data.

- **Advantages:** independent deployment and scaling, fault isolation, technology flexibility, and team autonomy.
- **Drawbacks:** distributed-system complexity, network latency and failures, data consistency across services, harder testing and debugging, and heavier DevOps/observability needs.

A good rule: start with a well-structured monolith unless scale or team size justifies the split; draw boundaries along business domains (bounded contexts).

> **Interviewer note:** Balanced view. Does not present microservices as always better.

M5 What is an event-driven architecture? Explain producers, consumers, and message brokers, and give an example where it fits better than REST calls.

Services communicate by publishing **events** (facts that happened) to a **broker** such as Kafka or RabbitMQ. **Producers** publish; **consumers** subscribe and react asynchronously. Producers do not know who consumes the event.

Example: when a ride is completed, a *RideCompleted* event is published. Billing, notifications and analytics each consume it independently. With chained REST calls, a slow or failing notification service would block or break the whole flow.

Trade-offs: loose coupling and resilience versus eventual consistency, harder debugging and the need to handle duplicates (at-least-once delivery means consumers must be idempotent).

> **Interviewer note:** Mentions idempotency, ordering and eventual consistency. The CV shows no Kafka experience, so assess understanding of the concept.

M6 In React, explain useState, useEffect, and props vs state. How would you prevent unnecessary re-renders?

- **useState:** adds local state to a component; updating it triggers a re-render.
- **useEffect:** runs side effects (data fetching, subscriptions) after render. The dependency array controls when it re-runs, and the returned function cleans up.
- **Props vs state:** props are read-only inputs passed from a parent; state is data a component owns and can change.

Reducing re-renders: keep state as local as possible, use *React.memo*, *useMemo* and *useCallback* for expensive or stable values, use stable *key*s in lists, split large components, and lazy-load routes. Measure with the React Profiler before optimising.

> **Interviewer note:** Avoids premature optimisation and understands dependency arrays.

M7 Explain a CI/CD pipeline. What stages would you include for a Spring Boot application?

CI runs automatically on every push or pull request; **CD** delivers or deploys passing builds.

- **Stages:** checkout, build (Maven/Gradle), unit tests, static analysis (Checkstyle/SonarQube), dependency vulnerability scan, package the JAR.
- **Delivery:** build the Docker image, push it to a registry, deploy to staging, run integration and smoke tests, manual approval, deploy to production (rolling or blue-green), then monitor with an easy rollback.

Tools: GitHub Actions, Jenkins or GitLab CI.

> **Interviewer note:** Includes testing gates, image build, environments and rollback.

M8 At SLT you wrote stored procedures and triggers in SQL Server. What are they, when should you use them, and what are their drawbacks?

- **Stored procedure:** precompiled, parameterised SQL stored in the database. Benefits: fewer network round trips, central business rules, and permission control.
- **Trigger:** code that runs automatically on INSERT, UPDATE or DELETE. Good for audit logs and enforcing integrity rules.
- **Drawbacks:** logic is hidden from application code, harder to debug, version and unit-test; ties you to one database vendor; triggers add overhead and can cascade unexpectedly; scaling business logic inside the database is difficult.

Use triggers sparingly (for example auditing) and keep most business logic in the application layer.

> **Interviewer note:** Ties the answer to the maintenance system: what was automated (backups, notifications, reports) and why in the DB.

M9 What are database indexes? How do they speed up queries, and when can they hurt performance?

An index is a separate data structure (usually a B-tree) that lets the database locate rows without scanning the whole table. It speeds up *WHERE*, *JOIN* and *ORDER BY*.

- **Composite indexes** follow the leftmost-prefix rule.
- **They hurt when:** there are too many (every INSERT/UPDATE/DELETE must maintain them), columns have low cardinality (for example a boolean), tables are tiny, or queries wrap the column in a function so the index cannot be used.

Check with *EXPLAIN* instead of guessing.

> **Interviewer note:** Mentions write cost, composite order and query plans.

M10 Our role involves Flutter. Have you used it? If not, how would you approach learning it, and how does its widget/state model compare to React's component model?

Honest answer expected: the CV does not list Flutter. A strong candidate says so, then shows a plan.

Concepts: Flutter is Google's UI toolkit using Dart; one codebase for Android and iOS. Everything is a widget; *StatelessWidget* has no mutable state and *StatefulWidget* uses *setState*. Larger apps use Provider, Riverpod or Bloc.

Compared with React: widgets are like components, *build()* is like render, *setState* is like *useState*, and both are declarative. Flutter draws its own UI instead of using native views.

Plan: learn Dart basics, follow the official codelabs, then build a small app that consumes a REST API (for example a vehicle list) and add state management.

> **Interviewer note:** Honesty, transferable React knowledge, and a concrete, time-boxed learning plan.

Advanced Questions

Stretch questions on distributed systems, performance and design. Treat strong answers as a bonus.

A1 **You have an order service and a payment service. How do you maintain data consistency across them without a distributed transaction? Explain the Saga pattern.**

Two-phase commit is avoided in microservices: it blocks, reduces availability and couples services. A **Saga** is a sequence of **local transactions**, each publishing an event; if a step fails, **compensating actions** undo earlier steps.

Example: Order created (PENDING) -> payment service charges the card -> success: order becomes CONFIRMED. If payment fails: *PaymentFailed* event -> order set to CANCELLED. If a later step fails after payment: issue a refund.

- **Choreography:** services react to each other's events. Simple but harder to follow as it grows.
- **Orchestration:** a central coordinator drives the steps. Easier to track and reason about.

Supporting patterns: the **Outbox pattern** (write the event and the DB change in one local transaction, then publish) and **idempotency keys** so retries never charge twice.

> **Interviewer note:** Compensation, idempotency, and awareness that sagas give eventual (not immediate) consistency.

A2 **How would you design an event-driven system using Kafka or RabbitMQ for real-time vehicle tracking? Cover ordering, retries, idempotency, and dead-letter queues.**

- **Ingestion:** vehicles or a gateway publish location events to a Kafka topic, **keyed by vehicleId**, so all events for one vehicle land in the same partition and stay ordered.
- **Scaling:** multiple partitions with consumer groups; add consumers to scale.
- **Reliability:** producer *acks=all* with the idempotent producer enabled; consumers commit offsets only after successful processing (at-least-once).
- **Idempotency:** include an event id or sequence number; store the latest position with an upsert so duplicates or late events are harmless.
- **Retries and DLQ:** retry transient failures with exponential backoff; route poison messages to a dead-letter topic/queue for inspection.
- **Serving:** latest position in Redis, history in a time-series store, and a WebSocket/SSE service pushing updates to the live map. Monitor consumer lag.

RabbitMQ difference: queue-based with per-message acks and dead-letter exchanges; Kafka is a replayable log better suited to high-throughput streams.

> **Interviewer note:** Partition key choice, at-least-once plus idempotent consumers, DLQ, and lag monitoring.

A3 **Explain how Kubernetes works: pods, deployments, services, and ingress. How would you deploy and scale a Spring Boot microservice on it?**

- **Pod:** smallest deployable unit, one or more containers sharing network and storage.
- **Deployment:** declares the desired number of replicas, manages rolling updates and rollbacks via ReplicaSets.
- **Service:** stable virtual IP/DNS name that load-balances across pods (ClusterIP, NodePort, LoadBalancer).
- **Ingress:** HTTP/HTTPS routing from outside the cluster to services (needs an ingress controller).

Deploying Spring Boot: build the Docker image and push it to a registry; write Deployment and Service YAML; use ConfigMaps/Secrets for configuration; add liveness and readiness probes (Spring Actuator health endpoints); set CPU/memory requests and limits; expose via Ingress; use a **Horizontal Pod Autoscaler** to scale on CPU or custom metrics.

> **Interviewer note:** NOTE: Kubernetes is not on the CV and is only an advantage for this role. Credit sound fundamentals and a learning path.

A4 A Spring Boot service is slow under load. How do you diagnose it (profiling, thread dumps, DB queries, connection pools), and what fixes would you consider?

Measure first. Use Actuator/Micrometer with Prometheus and Grafana (or an APM) to find where time goes: CPU, memory/GC, threads, database, or downstream calls. Reproduce with a load test (JMeter or k6).

- **Database:** slow-query log and *EXPLAIN*; look for missing indexes, **N+1 queries** (fix with fetch joins or entity graphs), and unbounded result sets (add pagination).
- **Connection pool (HikariCP):** pool exhaustion shows up as threads waiting for connections; tune pool size and fix long-held transactions.
- **Threads:** take thread dumps (*jstack*) to find blocked or deadlocked threads.
- **Memory/GC:** heap dumps and GC logs for leaks and long pauses.
- **Fixes:** caching (Caffeine/Redis), async processing, timeouts and circuit breakers on external calls, batching. Re-run the load test to confirm the improvement.

Interviewer note: Evidence-driven approach (measure, hypothesise, fix, verify) rather than guessing.

A5 How would you secure communication between microservices and handle API gateway, rate limiting, and token propagation?

- **API gateway** (Spring Cloud Gateway, Kong): single entry point handling TLS termination, authentication, routing, CORS and rate limiting.
- **Authentication:** OAuth2/OIDC with an identity provider such as Keycloak; the gateway and/or each service validates the JWT signature, expiry and scopes.
- **Token propagation:** forward the user's token downstream, or use token exchange; for service-to-service calls use client-credentials tokens or **mTLS**.
- **Rate limiting:** token-bucket per client or IP, backed by Redis, returning 429 with a Retry-After header.
- **Zero-trust mindset:** do not trust the internal network; apply least-privilege scopes, validate inputs, keep secrets in a vault, and keep audit logs.

Interviewer note: Defence in depth, and knows internal traffic still needs authentication.

A6 In your TM Forum REST API work, what did CTK conformance testing involve? What design decisions did you make for versioning, error handling, and backward compatibility?

TM Forum Open APIs standardise telecom REST interfaces. The **CTK (Conformance Test Kit)** is an automated suite that checks an implementation against the specification: resource schemas, mandatory attributes, endpoints, status codes, filtering and pagination, and notification/hub behaviour.

Typical design decisions:

- **Versioning:** version in the path (for example */v4*), with additive, non-breaking changes inside a version.
- **Errors:** consistent error body with code and message, and correct status codes, as the spec defines.
- **Backward compatibility:** only add optional fields, never rename or remove; deprecate with notice.
- **Other:** field selection, offset/limit pagination, validation against the schema.

Interviewer note: This is the candidate's own work. Look for specifics (which TMF API, which CTK failures they fixed). Vague answers deserve follow-up.

A7 Walk me through the architecture of your RL-based hospital bed allocation platform. How would you productionize the Python model and serve it to a Java/Spring backend reliably?

Expected architecture: a Next.js dashboard, an API layer, and a Python RL agent (PyTorch) trained in a simulated hospital environment that recommends bed allocations.

Productionizing:

- Wrap inference in a **FastAPI** service with versioned model artifacts, loaded once at startup, with validated inputs.
- The Spring backend calls it through *WebClient* with timeouts, retries and a **circuit breaker** (Resilience4j), and a **fallback to rule-based allocation** if the model is down.
- Containerise and scale the model service separately; keep training offline and separate from online inference. Consider exporting to ONNX to run inside the JVM.
- Use a model registry (for example MLflow), shadow or A/B rollout, and monitor latency, data drift and outcome metrics.
- Because this is healthcare, keep a **human in the loop** who can override recommendations.

Interviewer note: Awareness of the simulation-to-reality gap, fallbacks, versioning, monitoring, and safety.

A8 Explain RAG and vector search as you used them in Edumind. How would you handle embedding drift, chunking strategy, hallucination, and latency in production?

RAG (Retrieval-Augmented Generation): embed documents into vectors, store them (for example MongoDB Atlas Vector Search), embed the user's question, retrieve the top-k similar chunks, and pass them to the LLM as context so the answer is grounded in the source material.

- **Chunking:** roughly 300-800 tokens with some overlap, split on semantic boundaries (headings, paragraphs), with metadata (course, lesson) for filtering.
- **Embedding drift:** store the embedding model version with each vector; changing models means re-embedding everything, so plan a re-index and a cut-over.
- **Hallucination:** instruct the model to answer only from the context and say "I don't know" otherwise; set a relevance threshold, add citations, optionally rerank, and evaluate faithfulness and recall@k.
- **Latency:** tune top-k, use ANN indexes, cache frequent queries, stream responses, and ingest documents asynchronously.

Interviewer note: Understands retrieval quality as the main driver of answer quality, and mentions evaluation.

A9 Design a ride/vehicle booking system for a mobility company. Cover the services, database choices, caching, scalability, and failure handling.

Requirements: search nearby vehicles, book, pay, track in real time, notify. Needs high availability, low latency, and no double booking.

- **Services:** user/auth, fleet/vehicle, location/tracking, booking, pricing, payment, notification, behind an API gateway.
- **Data:** relational DB (PostgreSQL/MySQL) for bookings and payments (ACID); **Redis GEO** for nearby-vehicle lookup and availability; a time-series or wide-column store for location history; Kafka for events.
- **Avoid double booking:** optimistic locking or a unique constraint, plus a short Redis hold with TTL during checkout.
- **Consistency:** a Saga across booking and payment, with idempotency keys on payment calls.
- **Scale:** stateless services behind load balancers, read replicas, sharding by region, and WebSocket/SSE for live tracking.
- **Resilience:** timeouts, retries with backoff, circuit breakers, graceful degradation (cached prices), and monitoring/alerting.

Interviewer note: Structured approach: requirements first, then components, then trade-offs. Look for clarifying questions.

A10 Explain the CAP theorem and eventual consistency. Which would you choose for a booking system versus a notification system, and why?

CAP: during a network partition a distributed system must choose between **Consistency** (every read sees the latest write) and **Availability** (every request gets a response). Partition tolerance cannot be given up in practice, so the real choice is CP vs AP.

Eventual consistency: replicas may briefly disagree but converge if no new updates arrive.

- **Booking system: favour consistency (CP).** Two users must never book the same vehicle, and money is involved. Accept rejecting or delaying a request during a partition.
- **Notification system: favour availability (AP).** A delayed or duplicated notification is acceptable; use retries and idempotent delivery.

Good answers add that the trade-off is not binary and can be tuned per operation (for example tunable consistency levels).

> **Interviewer note:** Can apply the theory to a concrete business case and justify the choice.