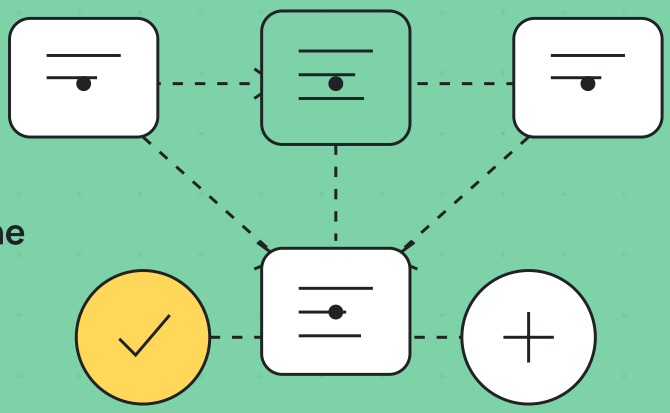
H/ HelpMeSwitch

FREE SAMPLE

INTERVIEW PLAYBOOK / BACKEND

Backend Interview Playbook

Prepare for the second question - the one that tests what you owned, what failed and why you chose this design over the alternative.



THE PREMISE

You rarely lose the round on the first answer. You lose it when the interviewer asks *why*, *what failed*, and *what did you personally do?*

108	422+	108	12	12
QUESTIONS	FOLLOW-UPS	FRAMEWORKS	MODULES	DRILLS

FREE SAMPLE · 8 OF 108 QUESTIONS

What's inside this sample

One flagship question from eight different rounds — the complete weak-answer teardown, follow-ups and answer framework for each, exactly as they appear in the full playbook.

01 A client calls your API, times out, and retries. Walk me through what your server does. — p. 3
API design and integration · Q2.3

02 You're getting deadlocks in production. How do you diagnose and fix them? — p. 4
Databases and transactions · Q3.4

03 A message can't be processed - it keeps failing. What happens to it? — p. 5
Kafka, queues and asynchronous processing · Q4.4

04 A popular cached key expires and thousands of requests hit the database at once. What is this and how do you prevent it? — p. 6
Redis and caching · Q5.3

05 Two services disagree about the state of the world. How does that happen and how do you fix it? — p. 7
Distributed systems and microservices · Q6.4

06 A downstream dependency you call is slow or failing. Walk me through how your service behaves. — p. 8
Reliability and resilience · Q7.1

07 Requests are hanging and latency is spiking. You have no idea why. Where do you start? — p. 9
Production incidents and debugging · Q8.2

08 How do you make sure one authenticated user can't access another user's data? — p. 10
Security and data protection · Q10.7

Q2.3 A client calls your API, times out, and retries. Walk me through what your server does.

WHAT IT REALLY TESTS Whether idempotency and duplicate-handling are designed in, or the system silently double-processes.

! WEAK ANSWER

> "The retry would just come in as a new request and get processed. Usually that's fine because most of our operations were reads."

WHY IT FAILS

- Assumes retries are harmless without checking write semantics.
- No idempotency mechanism for the writes that aren't safe.
- Doesn't consider the timeout might have fired after the server committed.
- No dedupe, no idempotency key.

↳ FOLLOW-UP LADDER

1. FAILURE — **The first request actually succeeded but the response was lost - what does the retry do?** The hard case that breaks naive designs.
2. TECHNICAL — **How do you make a POST idempotent?** Concrete mechanism.
3. TRADEOFF — **Where do you store the idempotency key and for how long?** Practical implementation detail.
4. MEASUREMENT — **What status code does the replay return?** Contract precision.

→ ANSWER FRAMEWORK

1. Classify the operation → 2. Safe-to-retry check → 3. Idempotency key on unsafe writes → 4. Storage and TTL of keys →
5. What the replay returns → 6. Distinguish 'in progress' from 'completed' → 7. Guard against concurrent duplicates.

× REJECTION SIGNALS

- Assumes all retries are safe.
- No mechanism for the lost-response case.
- Cannot describe idempotency key storage.
- Confuses idempotency with retries themselves.

✓ MODEL ANSWER SKELETON

- First I classify: GETs and PUTs are naturally idempotent, so a retry is fine. For a POST that creates a charge, a naive retry double-charges, so I require an `Idempotency-Key`.
- On the first request I insert the key into a table inside the same transaction that does the work; if the client retries and the response was lost, the retry hits a unique-constraint violation on the key, and I return the stored original response with the original 201 rather than re-executing.
- To handle the case where the first request is still in flight, the key row has a status - `in_progress` returns a 409 or makes the retry wait, `completed` replays the result.
- I keep keys for 24 hours, which comfortably covers client retry windows, and expire them to bound the table.
- That way 'timed out and retried' always converges to exactly one charge.

Q3.4

You're getting deadlocks in production. How do you diagnose and fix them?

WHAT IT REALLY TESTS Whether you understand deadlock mechanics (lock ordering) and the standard fixes, or just increase timeouts.

! WEAK ANSWER

> "We'd see deadlock errors in the logs. We increased the timeout and added retries so the transaction would just try again, which mostly solved it."

WHY IT FAILS

- Retrying masks the deadlock instead of removing the cause.
- No mention of lock-ordering, the actual root cause.
- Never inspected the deadlock graph to see which locks collided.
- "Mostly solved" - the deadlocks are still happening.

↳ FOLLOW-UP LADDER

1. TECHNICAL
   What's the root cause of a two-transaction deadlock?
   Lock-ordering understanding.
2. MEASUREMENT
   How did you see which two statements deadlocked?
   Uses the deadlock log/graph.
3. TRADEOFF
   How does consistent lock ordering prevent it?
   The real fix.
4. FAILURE
   Why can retries alone be dangerous here?
   Understands masking vs fixing.

→ ANSWER FRAMEWORK

1. Reproduce/observe the deadlock → 2. Read the deadlock graph → 3. Identify the two lock-acquisition orders →
4. Impose a consistent order → 5. Shorten transactions / reduce lock scope → 6. Add bounded retry as a safety net →
7. Verify frequency dropped.

× REJECTION SIGNALS

- Only solution is retry + timeout.
- Never looked at what actually deadlocked.
- Doesn't know lock ordering causes it.
- Thinks a bigger timeout is a fix.

✓ MODEL ANSWER SKELETON

- → Deadlocks in production almost always mean two transactions acquire the same locks in opposite orders - transaction A locks row 1 then waits on row 2, B locks row 2 then waits on row 1. First I read the database's deadlock log, which in Postgres tells you the two statements and the rows involved.
- → In one case we had a transfer that locked the two accounts in the order they appeared in the request, so A→B and B→A transfers deadlocked.
- → The fix wasn't a bigger timeout - it was to always lock accounts in a deterministic order, e.g. ascending by account id, so both transactions grab them in the same sequence and one simply waits instead of deadlocking.
- → I also shortened the transactions to reduce the window.
- → I kept a bounded retry with backoff as a safety net for the rare residual case, but retries were the seatbelt, not the fix - the lock ordering was.

Q4.4

A message can't be processed - it keeps failing. What happens to it?

WHAT IT REALLY TESTS Poison-message handling, dead-letter queues, and not blocking the whole partition.

! WEAK ANSWER

> "We'd retry the message a few times, and if it still failed we'd log an error and move on to the next one."

WHY IT FAILS

- "Move on" can mean silently dropping data.
- Retrying in place on an ordered partition blocks everything behind it.
- No dead-letter queue for later inspection.
- No distinction between transient and permanent failures.

4 FOLLOW-UP LADDER

1. FAILURE
   Retrying in place on an ordered partition - what does it block?
   Head-of-line blocking awareness.
2. TRADEOFF
   Transient vs permanent failure - how do you treat them differently?
   Retry classification.
3. TECHNICAL
   What goes into a dead-letter queue and who looks at it?
   DLQ operational design.
4. REFLECTION
   How do you replay a fixed message from the DLQ?
   Recovery loop.

→ ANSWER FRAMEWORK

1 Classify transient vs permanent → 2 Bounded retries with backoff for transient → 3 Route permanent failures to a DLQ →
4 Commit offset so the partition isn't blocked → 5 Alert on DLQ growth → 6 Inspect and fix → 7 Replay from DLQ.

× REJECTION SIGNALS

- Drops failed messages silently.
- Retries forever in place, blocking the partition.
- No DLQ.
- No alert on the DLQ filling up.

✓ MODEL ANSWER SKELETON

- → I separate transient from permanent failures. Transient - a downstream timeout - gets bounded retries with backoff, because it'll likely succeed shortly.
- → But retrying a permanently bad message in place is dangerous on an ordered partition: the whole partition is stuck behind it, so everything after that order's events stops flowing - head-of-line blocking.
- → So after N failed attempts I route the message to a dead-letter topic with the original payload, the error, and a trace id, then commit the offset so the live partition keeps moving.
- → I alert when the DLQ grows, because a filling DLQ is a real signal something systematic broke, not a shrug.
- → An engineer inspects the DLQ, fixes the bug or the data, and I have a small tool to replay messages from the DLQ back into the main topic once they're fixed, so we don't lose data - we quarantine it.

Q5.3

A popular cached key expires and thousands of requests hit the database at once. What is this and how do you prevent it?

WHAT IT REALLY TESTS Cache stampede / thundering herd - a classic that separates real operators from theorists.

! WEAK ANSWER

> "We'd just increase the TTL so the key doesn't expire as often, which reduces how often we hit the database."

WHY IT FAILS

- Longer TTL delays but doesn't prevent the stampede.
- Doesn't name the problem or the standard fixes.
- No locking, no early recompute, no stale-while-revalidate.
- Ignores that a cold cache or eviction triggers it too.

4 FOLLOW-UP LADDER

1. TECHNICAL
   How does a mutex/lock stop the stampede?
   Single-flight recompute.
2. TRADEOFF
   What's probabilistic early expiration (XFetch)?
   Advanced prevention.
3. MEASUREMENT
   How does stale-while-revalidate change the user experience?
   Serving stale to protect the DB.
4. FAILURE
   How would you protect against the same thing on a cold start?
   Cold-cache stampede.

→ ANSWER FRAMEWORK

1. Name it (stampede/thundering herd) → 2. Single-flight lock so one request recomputes → 3. Others wait or serve stale →
4. Probabilistic early recompute → 5. Stale-while-revalidate → 6. Warm the cache on deploy →
7. Jitter TTLs to avoid synchronised expiry.

× REJECTION SIGNALS

- Only fix is a longer TTL.
- Can't name the phenomenon.
- No locking or single-flight concept.
- No cold-start consideration.

✓ MODEL ANSWER SKELETON

- That's a cache stampede, or thundering herd - one hot key expires and every concurrent request misses simultaneously, so a query that normally runs once a minute suddenly runs thousands of times and can take the database down. A longer TTL just postpones it.
- The core fix is single-flight: when the key is missing, requests take a short-lived Redis lock (SET NX), and only the lock winner recomputes and repopulates while the others briefly wait or serve the last stale value.
- I also like stale-while-revalidate - serve the expired value immediately and refresh asynchronously - so users never block on a recompute.
- For very hot keys, probabilistic early expiration (the XFetch approach) has requests randomly refresh slightly before the TTL, spreading the recompute so it never all lands at once.
- And I jitter TTLs so a batch of keys populated together doesn't all expire in the same instant, and warm critical keys on deploy so a cold start doesn't stampede.

Q6.4 Two services disagree about the state of the world. How does that happen and how do you fix it?

WHAT IT REALLY TESTS Understanding of eventual consistency, reconciliation, and that distributed state drifts.

! WEAK ANSWER

> "That shouldn't happen if the services are communicating properly. We'd just make sure they always sync their data correctly."

WHY IT FAILS

- Denies that drift is normal in distributed systems.
- "Sync correctly" is not a mechanism.
- No reconciliation or source-of-truth concept.
- Doesn't consider dropped events or partial failures as causes.

4 FOLLOW-UP LADDER

1. TECHNICAL
   An event is dropped or processed twice - how do the two states diverge?
   Understands drift causes.
2. OWNERSHIP
   Which service is the source of truth, and how does the other reconcile?
   Reconciliation design.
3. MEASUREMENT
   How would you detect the divergence before a user does?
   Drift detection.
4. FAILURE
   How do you repair state without double-applying?
   Idempotent reconciliation.

→ ANSWER FRAMEWORK

1 Accept drift is normal → 2 Causes (lost/duplicate events, partial failures) → 3 Define single source of truth →
4 Periodic reconciliation job → 5 Idempotent repair → 6 Detect drift with metrics/audits → 7 Alert before users notice.

× REJECTION SIGNALS

- Claims drift shouldn't happen.
- No source of truth.
- No reconciliation mechanism.
- Repair double-applies changes.

✓ MODEL ANSWER SKELETON

- → In any event-driven system state drift is normal, not a bug - an event gets dropped, processed twice, or a consumer is down during a change, and two services' views diverge. So I design for it rather than deny it.
- → First I name a single source of truth for each piece of data - say the orders service owns order status - and everyone else holds a derived copy they reconcile against it.
- → I run a periodic reconciliation job that compares the derived state to the source and repairs mismatches, and crucially that repair is idempotent and set-based, so re-running it or applying it to an already-correct record does nothing.
- → To catch drift early I emit a metric for reconciliation mismatches and alert if it climbs, so we find divergence from monitoring rather than from a customer complaint.
- → The mental model I keep is: assume messages can be lost or duplicated, keep one authority, and continuously reconcile toward it.

Q7.1

A downstream dependency you call is slow or failing. Walk me through how your service behaves.

WHAT IT REALLY TESTS Whether the candidate has timeouts, circuit breakers, and fallbacks - or lets a slow dependency take them down.

! WEAK ANSWER

> "We'd retry the call a few times, and if it kept failing we'd return an error to the user. We also had monitoring to alert us."

WHY IT FAILS

- Retries without backoff can amplify the downstream's overload.
- No timeout mentioned, so a slow dependency ties up threads.
- No circuit breaker to stop hammering a dead service.
- No fallback or graceful degradation.

↳ FOLLOW-UP LADDER

1. FAILURE
   No timeout on that call - what happens to your thread pool?
   Resource-exhaustion cascade.
2. TECHNICAL
   How does a circuit breaker help versus just retrying?
   Fail-fast under sustained failure.
3. TRADEOFF
   Retrying a struggling service - how do you avoid making it worse?
   Backoff + jitter + budget.
4. REFLECTION
   What can you serve when the dependency is down?
   Fallback / degradation.

→ ANSWER FRAMEWORK

1. Timeout on every call
2. Bounded retries with backoff + jitter
3. Circuit breaker to fail fast under sustained failure
4. Fallback / cached / degraded response
5. Bulkhead to isolate the dependency
6. Alert
7. Recover when breaker half-opens.

× REJECTION SIGNALS

- No timeout on external calls.
- Retries without backoff.
- No circuit breaker.
- No fallback; user just gets an error.

✓ MODEL ANSWER SKELETON

- The first line of defence is a timeout on every outbound call, because without one a slow dependency holds my request threads open until my own pool exhausts and I take down my whole service over one struggling downstream. On failure I retry, but bounded and with exponential backoff plus jitter, because fixed immediate retries from every instance at once just pile onto an already-overloaded service and turn a blip into an outage.
- On top of that I run a circuit breaker: after a threshold of failures it opens and I fail fast for a cooldown instead of sending doomed calls, then half-opens to test recovery before closing.
- While the breaker is open I serve a fallback - a cached value, a default, or a degraded response - so the user gets something useful rather than a spinner.
- I also bulkhead that dependency onto its own thread pool so its failure can't starve unrelated requests.
- The principle is contain the blast radius: one sick dependency should degrade one feature, not the service.

Q8.2 Requests are hanging and latency is spiking. You have no idea why. Where do you start?

WHAT IT REALLY TESTS A systematic debugging method under uncertainty - the actual skill being scored.

! WEAK ANSWER

> "I'd check the logs to see if there were any errors, and probably restart the service to see if that fixed it."

WHY IT FAILS

- Restart-first can destroy the evidence and only masks the problem.
- No hypothesis-driven method.
- Logs alone without metrics/traces is a needle-in-haystack.
- No mention of recent changes as the prime suspect.

↳ FOLLOW-UP LADDER

1. TECHNICAL
   What changed recently - deploy, config, traffic?
   Change-correlation instinct.
2. MEASUREMENT
   Which signal tells you if it's CPU, memory, DB, or downstream?
   Resource isolation via metrics.
3. FAILURE
   Why might restarting be the wrong first move?
   Preserving evidence.
4. TRADEOFF
   How do you narrow it to one component in a chain?
   Tracing to localise.

→ ANSWER FRAMEWORK

1. Check what changed recently
2. Look at the golden signals (latency, traffic, errors, saturation)
3. Isolate the resource (CPU/mem/DB/IO/downstream)
4. Use traces to localise the slow hop
5. Form a hypothesis
6. Test safely
7. Mitigate
8. Preserve evidence for the postmortem.

× REJECTION SIGNALS

- Restarts before investigating.
- No hypothesis, just pokes around.
- Ignores recent changes.
- Logs-only, no metrics or traces.

✓ MODEL ANSWER SKELETON

→ My first question is always 'what changed' - a deploy, a config flip, a traffic surge, a downstream degradation - because most incidents correlate with a recent change, and that's the fastest path to mitigation. In parallel I look at the golden signals: latency, traffic, error rate, and saturation.

→ Saturation is what tells me whether it's CPU-bound, memory/GC, a full connection pool, disk IO, or a slow downstream, so I'm isolating the resource rather than guessing.

→ Then I pull a trace of a slow request to see which hop actually eats the time - often it's one dependency or one query, not the whole service. From that I form a specific hypothesis and test it safely, for example checking the DB's active-connection count if I suspect pool exhaustion.

→ I deliberately don't restart first, because a restart can clear the symptom and destroy the evidence - the heap, the connection state, the thread dump - that I need to actually fix it.

→ I mitigate, but I capture diagnostics before I do.

Q10.7 How do you make sure one authenticated user can't access another user's data?

WHAT IT REALLY TESTS Object-level / row-level authorization - BOLA - the most common real API vulnerability, from a design angle.

! WEAK ANSWER

> "Since they're authenticated with their own token, they can only access their own account. The token identifies them."

WHY IT FAILS

- Confuses authentication with per-resource authorization.
- Assumes the token restricts data access - it only proves identity.
- No ownership check on the requested resource.
- Exactly the BOLA vulnerability the question probes.

4 FOLLOW-UP LADDER

1. TECHNICAL
   They change the id in the URL to someone else's - what's your server-side check?
   Explicit ownership verification.
2. FAILURE
   Why isn't 'they have a valid token' enough?
   Authn ≠ authz.
3. TRADEOFF
   How do you enforce this consistently across hundreds of endpoints?
   Centralised/systematic enforcement.
4. MEASUREMENT
   How does row-level security in the DB help as a backstop?
   Defence in depth.

→ ANSWER FRAMEWORK

1. Never trust client-supplied ids for access
2. Verify resource ownership against the authenticated principal on every access
3. Centralise the check (middleware/policy)
4. Don't expose or trust sequential ids alone
5. Row-level security as a backstop
6. Audit access
7. Test with cross-account attempts.

× REJECTION SIGNALS

- Thinks a valid token restricts data access.
- No ownership check on resources.
- Relies on obscure ids instead of authorization.
- No systematic enforcement.

✓ MODEL ANSWER SKELETON

→ The dangerous assumption is that a valid token limits what data you can reach - it doesn't, it only says who you are. If `/orders/1234` returns whatever order 1234 is, then any authenticated user just changes the number and reads someone else's order - that's broken object-level authorization, the most common API vulnerability in the wild.

→ So on every resource access I verify ownership server-side: does this order's `user_id` match the authenticated principal, or does their role grant access - and I never trust a client-supplied id as authorization.

→ To keep it consistent across hundreds of endpoints I centralise the check into a policy layer or middleware rather than hand-rolling it per handler, because the one endpoint someone forgets is the breach.

→ As a backstop I use database row-level security so even a buggy query can't return another tenant's rows, and I audit access to sensitive records.

→ And I explicitly test it - automated tests that authenticate as user A and try to read user B's resources and assert a 403 - because this is too important to leave to code review.

YOU JUST READ 8. THERE ARE 108.

The full round is waiting.

This sample gave you one question from each of eight rounds. The full playbook carries every question, every follow-up, and the answer framework for all 108 — the ones most likely to appear in your next interview.

01

See the full contents

Every module, every question and the exact follow-ups an interviewer asks after your first answer.

02

Practise the second question

Anyone can rehearse the opener. Strong candidates separate on what failed, what they owned, and what they would change now.

03

Walk in ready

Downloadable PDF, yours to keep. Read it the night before and stop being surprised by the follow-up.

> **A polished answer can sound borrowed. A specific answer sounds true.**

GET THE FULL PLAYBOOK helpmeswitch.in/playbooks/backend-interview-playbook