FREE SAMPLE

PRACTICAL

INTERVIEW RUNTIME

ANDROID INTERVIEW PREPARATION

Kotlin, Coroutines, Compose, and System Design
for Senior and Staff Engineers.

MIKE SALARI

FREE SAMPLE

The Timeout That Already Succeeded

One offline-sync question, followed until the plausible answer breaks

MIKE SALARI

From Interview Runtime 1.2: Android Interview Preparation

A free sample of *Interview Runtime 1.2: Android Interview Preparation*, 351 pages. One interview question, carried from the answer most candidates give to the one that survives the follow-up. This sample may be shared freely in its original, unmodified form.
INTERVIEWRUNTIME.COM

THE PROBLEM

The question sounds like plumbing

A field technician finishes an inspection in a basement with no signal. The app has to make sure the inspection reaches the server. The product team wants no lost inspections and no duplicate records.

> *How would you design the submission path so that a poor connection never loses the user's work?*

This is Task 3 in the book's diagnostic (chapter 02), and it is a standard senior Android question. Here is the answer a strong candidate gives, and the one you may well give yourself.

Save the inspection to Room first, so it survives a process kill. Put it in an outbox table. Enqueue a WorkManager job with a connected-network constraint. The worker reads the pending row, calls the API, and deletes the row on success. On a transient failure it returns `Result.retry()` *with exponential backoff, and after a bounded number of attempts it marks the row as needing attention so the user can see it.*

Nothing in that answer is wrong. Durable local write, scheduler that survives the process, backoff, a retry budget, a visible failure state. An interviewer writing notes would tick every box. The companion app in the book contains exactly this worker: eight attempts, a 30 second base delay that doubles, a six hour cap.

The question is what the answer says about one specific moment.

THE FOLLOW-UP

What did the timeout tell you, and what did it leave out?

The interviewer stops you at the worker and asks one thing.

> *The request leaves the device. The server receives it and commits the inspection. The response is lost on the way back, and your client sees a read timeout. What does your worker do next, and how does the server know what to do with the second request?*

Reciting a library name does not help here. WorkManager retries. OkHttp retries. Retrofit has a clean error type. None of them can answer the question, because the question is about what the client *knows*.

Chapter 08 puts it in one sentence: a timeout does not tell the client whether the service committed the operation. The device knows it attempted delivery. The service may already hold the inspection. Your worker maps the timeout to `RetryableFailure`, returns `Result.retry()`, and sends the same payload again. If the server has no way to recognize it, the technician now has two inspections, and your retry budget helped create the second one.

> **WHAT THE INTERVIEWER IS SCORING**
>
> Whether you separate what the device knows from what the server may know. A timeout is an unresolved outcome, not a failure.

Most candidates reach for "make it idempotent" at this point. That is correct and still not an answer. The interviewer's next questions are the real test:

- Who generates the idempotency key, and when?
- What does the server store, and in the same transaction as what?
- What happens if the same key arrives with different content?
- What if the app crashes after the server accepts but before the client records that it did?

Each of those has a plausible wrong answer. The first is where most of them live.

THE CORRECTION

The key is part of the data, not part of the request

The mechanism has three parts, and the chapter treats each as a separate decision.

The identity is created once, with the intent. The operation ID is written in the same local transaction as the inspection. Every attempt, after every timeout, every process restart and every backoff, sends that stored ID as the `Idempotency-Key`. In the companion, `DrainOutboxWorker` calls `service.submit(pending.id, accountId, pending.payload)`: the key is the row's ID, read from durable state, never minted at send time. The book also says where it must not be minted. An OkHttp interceptor that generates a key creates a new one per call, which defeats deduplication. Interceptors may carry the header, never invent it.

The difference is easy to see in a small model. A sketch of the same idea, compiled and run on plain Kotlin as a toy model (it is not the companion's code).

```
import java.io.IOException
import java.util.UUID

class Server {
    val effects = mutableMapOf<String, String>()
    var dropNext = true
    fun submit(key: String, body: String) {
        effects.getOrPut(key) { body } // commit
        if (dropNext) {
            dropNext = false
            throw IOException()        // reply lost
        }
    }
}

fun deliver(s: Server, key: () -> String) {
    repeat(3) {
        try {
            s.submit(key(), "revision 4")
            return
        } catch (_: IOException) { }
    }
}

fun main() {
    val fresh = Server()
    val stored = Server()
    deliver(fresh) { UUID.randomUUID().toString() }
    deliver(stored) { "op-7f3a" }
    println(fresh.effects.size)   // 2
    println(stored.effects.size)  // 1
}
```

Same retry loop, same lost response. A fresh key per attempt gives two effects. A stored key gives one.

The server stores the key and the effect in one transaction. If it records the key in one step and the business record in another, a crash between them reopens the hole. The companion's test service writes the operation identity and the inspection effect together, then closes the connection before sending any response.

The same key with different content is an error, not a replay. The book's contract tests include this case: reusing a key with different content in the same account is rejected, and the original effect is unchanged.

> **WHAT A SENIOR CAN TAKE FROM THIS**
>
> Create the operation identity once, store it with the data, and send that stored value on every attempt. The server must record the key and the effect together.

THE TEST

The companion tests this against a real HTTP exchange. The scenario is the one from the follow-up: the first request submits revision 4, the service commits, the connection closes. The technician then edits locally, so the draft becomes revision 5. The database is closed and reopened, the worker drains again, and the same operation is sent. The server ends with two requests and one effect, containing revision 4. Revision 5 stays local as newer work.

That last detail is worth saying in an interview. A retry resends the original intent, not the latest state of the screen.

The book is equally exact about the limit of this evidence. It establishes the tested protocol. It does not establish authentication, key expiry, replication or production-scale contention. In the recorded run of the companion's 31 instrumented tests, none failed.

THE CHANGED CONSTRAINT

Now the server cannot help

The interviewer changes one condition.

> *The backend team says they cannot add idempotency keys this quarter. Same requirement: no lost inspections, no duplicates. What do you ship?*

A candidate who memorized "send an idempotency key" has nothing left. The key is still sent and nothing reads it. The book's section on this case is blunt: a client-only queue cannot prove exactly-once business behavior.

The companion shows what that looks like. Its service fixture has a switch that disables deduplication while leaving the Android worker completely unchanged. With that switch on, the replay test and the competing-request test fail: each finds two effects where one was required. Two of the three service tests fail, and the client code is identical. The worker was never the thing preventing duplicates.

WHAT IS LEFT

So the method changes, and the book lists what remains available:

1. **Reconcile before resending.** If the server exposes a business identifier or a lookup, query for the inspection before retrying. A timeout then becomes a question the client can answer.
2. **Make the uncertainty a product state.** If neither exists, an ambiguous attempt becomes a visible status that asks for review, not a silent retry. The technician's work stays on the device and nothing is claimed that cannot be proved.
3. **Say plainly what you cannot guarantee.** The chapter's wording is to preserve the customer's work and provide a recovery path without falsely claiming duplicate prevention.

> #### THE ANSWER COVERS
>
> Without server support, the client cannot prove exactly-once. Reconcile first, show unresolved attempts as a visible state, and do not claim prevention you cannot show.

This is the moment that separates a memorized design from a method. The first answer was a pattern. The second is a decision about which guarantee you can own, given who controls which half of the contract. The diagnostic's own scoring note names the failure: retry timing presented as duplicate prevention.

WHERE YOU STAND

Five minutes with your own record

Pick the most recent offline or retry path you built or reviewed. Answer from what the code does, not from what you would say in an interview.

1. **Where is the operation ID created?** Before the first send, in the same transaction as the data, or at request time? If an interceptor, a repository call or a worker creates it, a retry may not carry the same one.
2. **What does the timeout branch do?** Find the line that handles a read timeout. Does it say "failed", or does it preserve an unresolved state?
3. **What did the server do with the second request?** If you cannot name the behavior, or the answer is "I assume they handle it", you have found the gap.
4. **What survives a crash between acceptance and your local write?** The book treats it as a normal recovery case, not an exotic one. Trace it.
5. **How many retries can your stack send in total?** Count the layers: your worker, your own interceptors, and OkHttp's `retryOnConnectionFailure`, which is on by default. If only one carries the key, the others are duplicates waiting to happen.

Then say your answer out loud, in under a minute, without the word "idempotent" until you have said what the client knows and what the server may know. Score it the way chapter 02 does: a 3 handles the stated problem and explains a failure path, a 4 adapts when the server cannot help, a 5 states the limits of its own evidence.

GET THE FULL BOOK

You have just done what the book is for.

You took one question, found where a plausible answer breaks, corrected it, and watched a changed constraint test whether the correction was a method or a memory. That is one loop.

The full edition runs that loop across the whole surface a senior interview covers, and scores the result so you can tell preparation from practice. A diagnostic scored on six dimensions, 79 questions and 74 timeboxed drills, and twelve simulations, each with its prompt, the private interviewer material, a critical gate and a correction route.

The companion. Loop Ready: a local-only Android app and a Kotlin/JVM project carrying the verification code, with drivers that run the experiments on an emulator.

Get the full book

A reader's review of this book

> ★★★★★
>
> As mobile developer, I can honestly say this is the best Android book I've ever had!!! The content is excellent, the design is beautiful, and it's been incredibly useful.
>
> **Ivan G., review on Gumroad**

Interview Runtime 1.2: Android Interview Preparation, 351 pages, PDF and EPUB. Mike Salari, staff mobile engineer, 500+ technical interviews from the hiring side. Copyright 2026. Share this sample freely. Not affiliated with, authorized by or endorsed by Google LLC.