Kunsang Lee 이건상

Backend Engineer

This portfolio expands on the work in my résumé: for each project, the problem, the decision and the result.

00 Contents

01 **Real-time 1:1 chat system**
May – Aug 2026
Distributed lock for simultaneous requests · message delivery across servers

02 **Internal AI agent: design and execution rules**
Jun – Nov 2026
Error-alert analysis · write conditions · read requests · code changes and review

03 **Test database cleanup**
Oct – Nov 2026
TRUNCATE only written tables · before/after measurement · a dropped hypothesis

04 **Bedrock call path and the transaction split**
Oct – Nov 2025
Transaction split · malformed-response handling · concurrent uploads

05 **ai-ready: rules moved from docs into tests**
May – Oct 2026
Architecture tests · one-command checks · per-module guides

Email gosmdochee@gmail.com
GitHub github.com/kunsanglee ↗
Web kunsanglee.github.io ↗

ai-ready github.com/kunsanglee/ai-ready ↗
Blog Part 1: Building the agent ↗ · Part 2: Redesigning it ↗

Real-time 1:1 chat system

Owned the 1:1 chat system from design through build | May – Aug 2026

Kotlin · Spring Boot · WebSocket · STOMP · Redis Pub/Sub

Two simultaneous chat requests end up in the same room

Context Cre8orClub is a platform where creators publish portfolios and invite each other to talk and collaborate. A 1:1 chat starts when one member requests a conversation and the other accepts.

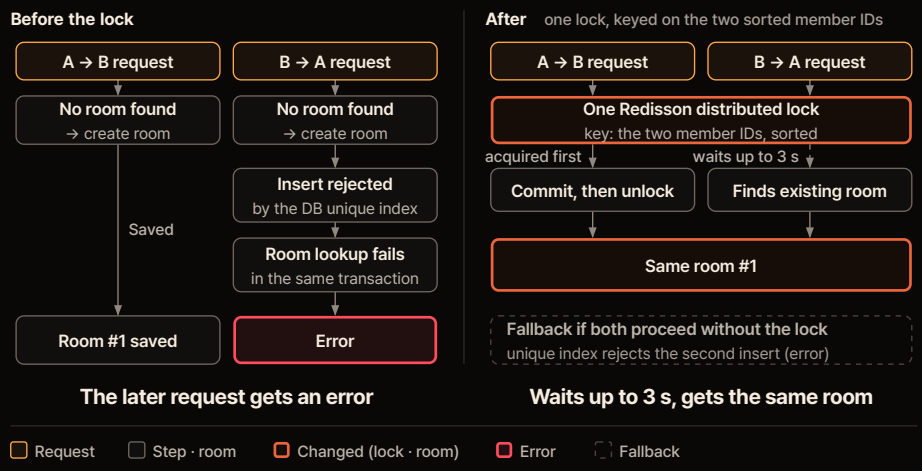
Problem When A requested a chat with B and B requested one with A at almost the same time, one of them got an error. Both requests saw that no room existed between the two yet, and each tried to create one. The database had a unique index that allows only one active chat request between the same two members, so the second request failed with a unique-constraint violation.

Cause The existing code caught this unique-violation exception and tried to find and return the room that had been created first. But a transaction in which an exception has been thrown is marked rollback-only, and PostgreSQL does not run further queries in a transaction that has hit an error. So the lookup of the existing room, made inside the same transaction, failed as well, and the request ended in an error.

Constraints Both people wanted the conversation, so neither should get an error; both should receive the same room. Production ran on several servers, so the two requests could be handled by different servers, and a lock inside one server could not prevent the collision.

Decision I made both requests acquire the same single lock before creating a room. The lock key is built from the two member IDs in sorted order, so that the A→B request and the B→A request produce the same key, and the lock is a distributed lock (Redisson) on the Redis instance that the servers share. The lock is released after the transaction that creates the room commits; if it were released before the commit, the waiting request would not see the unsaved room and would try to create another. I kept the unique index for cases where both requests proceed without the lock, such as the lock expiring after 30 seconds; in that case the second request gets an error.

Result When two people request each other at the same moment, the request that acquires the lock first creates the room, and the other waits up to 3 seconds and then receives that room with a normal response (200). A request that cannot acquire the lock within 3 seconds receives 409 Conflict.



With both requests acquiring the same lock, the second request waits and receives the same room

Options I did not choose, and why

Option	Why not
After the exception, look up the existing room in a new transaction (REQUIRES_NEW)	This finds the existing room in a transaction separate from the failed one. The code gets more complicated, and because the extra transaction appears only for requests that collide, the path a request took becomes harder to trace.
Drop the lookup and let the second request fail	This relies on the unique index alone, with no lock. The second request ends in an error every time, and how the app should behave on receiving that error had not been defined.
Acquire the lock, but return 409 immediately to a request that cannot acquire it	Both people want the same room, so waiting briefly and returning the room created first matches what they wanted better than giving one of them an error.

Messages reach a recipient connected to a different server

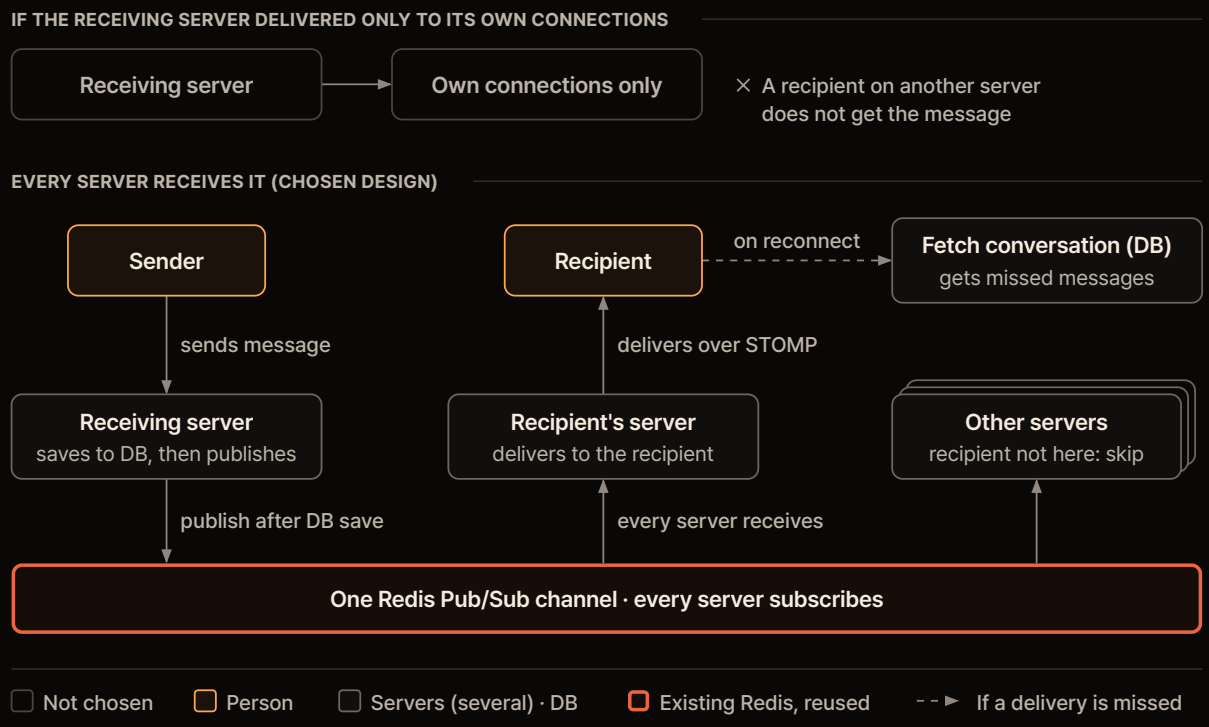
Context The app exchanges messages with the server over a WebSocket connection (using the STOMP protocol). The connection is made to one server, and each server can push messages only to the users connected to it.

Problem Production ran on several servers, so the sender and the recipient could be connected to different ones. If the server that received a message delivered it only to its own connections, a recipient connected to another server would not get it. The servers needed a way to tell each other about new messages.

Constraints All that was needed was to notify the servers running at that moment of a new message. Messages are stored in the database, so the channel between servers did not need to keep messages itself.

Decision The server that receives a message saves it to the database and then publishes it on a single Pub/Sub channel of the Redis instance that the product was already using. Every server subscribes to that channel and, if the recipient is connected to it, delivers the message to that person.

Result The recipient gets the message whichever server they are connected to. No new infrastructure was added for this.



After the save, one publish on a Redis channel reaches every server, and the server the recipient is connected to delivers the message

Why not Kafka or RabbitMQ

A message broker such as Kafka or RabbitMQ keeps messages in the broker, so a consumer that disconnects and reconnects can pick up what it missed. Chat messages are stored in the database and missed ones are fetched from there, so that capability was not needed. There was no reason to bring in and operate a new broker for delivery alone, so I chose the Redis already in use.

Only the message body is encrypted

In the event published to Redis, only the conversation text is encrypted; the information needed for delivery is left as it is. With everything encrypted, an operator could not watch the channel to investigate an incident.

Trade-off: a missed live delivery is not resent

Redis Pub/Sub does not keep published events; it delivers only to the servers subscribed at that moment. A user who misses a delivery may not see that message until they reconnect.

An event is published only after the message has been saved (committed) to the database, so on reconnect the app fetches the conversation after the last message it received. A missed real-time delivery can be recovered by fetching again, but a message that was never saved cannot. That is why the save comes first and real-time delivery is allowed to be missed.

Internal AI agent that works alongside the team in Slack

Planned, designed and operated; code written with an AI coding tool (Claude) | Jun – Nov 2026

TypeScript · Claude Agent SDK · PostgreSQL (pgvector)

Questions that used to wait for the one person who knew the answer are answered from the code and the data; error alerts are analyzed automatically

235

Team questions answered over 15 weeks (Jun 10 – Sep 22, 2026). Counted directly through the Slack API.

96s

Median time from the first question to the posted answer, across the same 235. Nine in ten were answered within 4.3 minutes.

99s

Median time to first answer for the 39 questions posted before 9:00 or after 19:00 on weekdays, or on weekends, in the same 15 weeks.

2.5h

Colleague time saved per week. A calculated figure that assumes 10 minutes of a colleague's time per question.

Problem Checking code in another part of the stack or looking up production data meant the person who knew that area had to stop what they were doing and answer. When that person was away, the person asking was blocked until the answer came, even for a question that takes a few minutes to answer.

Constraints More than half of the questions from colleagues came from non-engineering roles such as product planning, and the questions spanned the iOS, Android, web and backend repositories and the database. Tools that affect systems outside the agent, such as creating a ticket or changing code (called write tools below), must not run on the model's judgment alone, and the production DB had columns holding personal data.

Role I was responsible for the planning, design and operation of this agent. The code was written with an AI coding tool (Claude).

Result Colleagues no longer had to stop their own work to answer, and the person asking no longer had to wait for them to come back. Setting aside 9 messages that were commands or casual chat leaves 226 questions. At about 10 minutes of a colleague's time each (5 to answer, 5 to get back into their own work), that is 2,260 minutes: **about 38 hours** over 15 weeks, or **about 2.5 hours a week**.

When an error alert arrives, the agent analyzes the cause and posts it automatically

83%

Share of the 277 errors over 15 weeks that received a root-cause analysis (231 errors).

When a Sentry alert arrives in an error-alert channel, the agent starts a session for that issue and posts a root-cause analysis in the alert's thread. Over the same 15 weeks (Jun 10 – Sep 22, 2026), the median time from the first alert to the analysis was 113 seconds, and 228 of the 231 analyses were posted within 10 minutes.

When the same error keeps firing at short intervals, the model is not called again. Alerts with the same Sentry issue number and environment are treated as the same error, and an alert that arrives within 30 minutes of the last time that issue was handled is only logged.

Which repositories allow automatic fixes is set by an allow-list in the configuration. For an allowed repository, I designed the flow to continue through implementation, review and fixes, up to opening a fix PR and deploying it to the alpha (QA) server. Alerts from repositories outside the list get the analysis only.

First analysis by environment, same 15 weeks

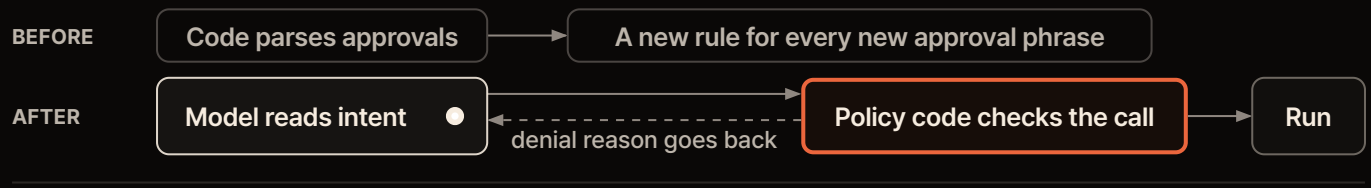
Environment	Errors that got an analysis	Median from first alert to analysis
Development	136 of 163	118 s
Alpha (QA)	71 of 84	105 s
Production	24 of 30	68 s

Counted directly through the Slack API on Sep 23, 2026. An alert is a top-level post by the Sentry app; an analysis is the agent's first reply in the thread of the same issue. For the 49 analyzed errors that arrived before 9:00 or after 19:00 on weekdays, or on weekends, the median was 106 seconds.

The model works out what a message means; code checks three conditions right before a write tool runs

September 2026 redesign: code no longer parses approval phrases

Area	Before the redesign	After the redesign (September 2026)
Approval	Creating a ticket or changing code ran only after a person approved, and code decided with regular expressions whether a person's message was an approval. Only the exact approval phrase the agent had shown was accepted.	The model works out what the message means. Code checks only three conditions (below) right before a write tool runs.
Problem and result	An expression that was not on the list, such as a casual shorthand for "go ahead", was not read as approval, so a rule had to be added every time a new expression appeared. As task types were added, the states (awaiting approval, in progress, done) and the transitions between them multiplied.	A new expression does not need a new rule. The condition check makes no external calls, so 91 tests cover it.



When policy code denies a call, it returns the reason to the model

Three conditions for running a write tool

The agent keeps one conversation with the model (a session) per Slack thread, and messages in the same thread continue in that session. In a thread where a person mentioned the agent, a write tool runs only when all three conditions hold.

① The agent posted in this thread first.

② A person replied after that.

③ There is no record of the same tool having run with the same input.

If a condition does not hold, policy code denies the call and returns the reason to the model. The model then posts what it intends to do in the thread and waits for a person's reply.

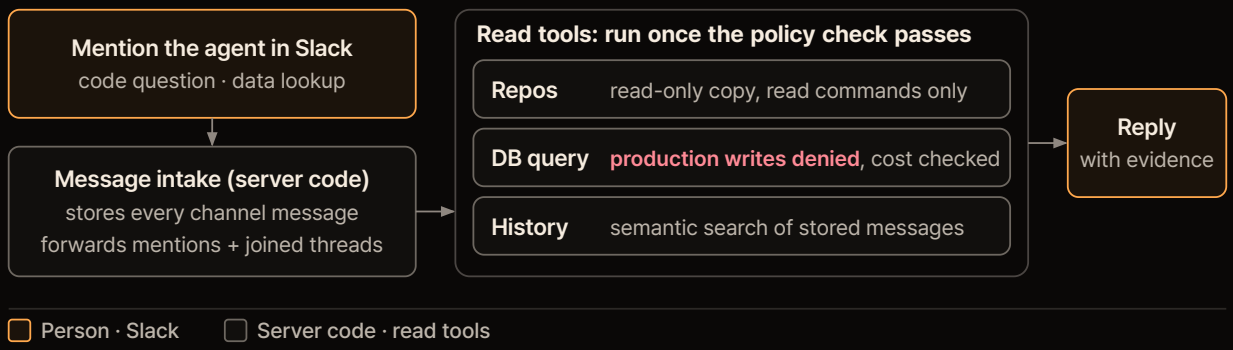
Condition ③ is there to stop a restart or a retry from creating the same ticket twice. An automatic fix started by an error alert does not wait for a person's reply. It runs only for repositories where automatic fixes are allowed, and in that case only ③ is checked.

How each kind of tool is decided

Tool	Decision
Read tools: looking up tickets, PRs and Sentry issues, searching past conversations, replying in Slack	Allowed.
Write tools: creating or editing a ticket, commenting on, editing or merging a PR, canceling a code task	The three conditions.
Code changes and deployment	In a session started by an alert, for a repository on the allow-list: only the execution record (③). Otherwise the three conditions.
Database	Development and alpha: reads allowed, writes under the three conditions. Production: writes denied, reads decided by estimated cost.

The table shows the main kinds. A tool with no entry in the policy's tool table is denied by default, and the decision function does not receive the text people wrote. In case the policy wrongly allows a call, the sandbox and the other limits on the last page of this chapter apply as well.

Questions are answered by reading the repositories and the database, with evidence, and the agent cannot write to the production DB



Read tools run once the policy check passes, and the answer is posted in the thread with its evidence. Only costly reads on the production DB wait for a person's reply.

Channel messages are stored; mentions and joined threads get answers

Every message posted in a channel the agent has joined is stored in the database, except DMs. The purpose is to answer questions such as why something was decided last month by searching past conversations, and storage is handled by server code with no model involved. Of these messages, only those that mention the agent and those in threads the agent has already joined are sent to the model for an answer; the rest are only stored.

Code questions are answered by exploring the repository directly

The agent searches and navigates files in a read-only copy of the repository, the way a person would. Code keeps changing, and a search over a prebuilt index would have needed the index rebuilt every time the code changed. Shell commands are limited to read-only commands inside the repository.

I also considered a commercial code-search product, whose per-seat pricing was too much for the size of the team, and querying the repository host's API directly, which was a poor fit for the dozens of searches and file reads that a single question takes.

Production DB access is limited by permissions and by the execution plan

I worked out with the infrastructure team the permissions for a dedicated agent DB account that cannot query personal-data columns, and the agent queries the production DB only through that account. The redesign added two things.

Policy code denies writes to the production DB. Before a read runs, its estimated cost is checked from the execution plan (EXPLAIN); at or below a configured limit, it runs right away. Above the limit, the agent first posts the estimated cost and row count in the thread and runs the query only after a person replies. A read whose cost could not be checked is denied, because there is no basis for treating it as cheap.

Past conversations are searched by meaning, with a local embedding model

Exact-text search could not connect "timeout" with "no response is coming back", so I chose embeddings and decided to add pgvector to the PostgreSQL already in use instead of running a separate vector database. Stored conversations are not added to every prompt up front, because most questions do not need them and they would only add tokens; the model calls a search tool when it judges that it needs them.

Sending the raw text of internal Slack conversations to a server outside the company to build embeddings was a security concern, so in October 2026 I decided to move embedding from an external API to a local model, EmbeddingGemma 2, running on the same Mac as the agent.

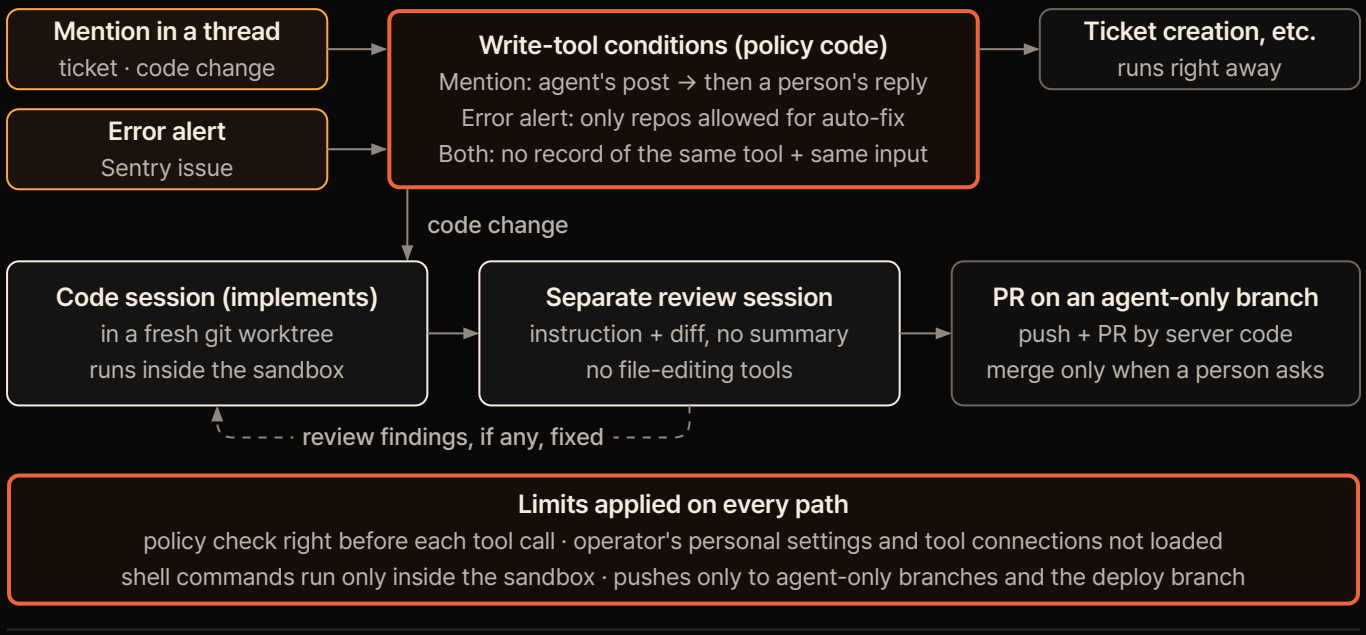
Structure rules as tests in the agent's own code

So that the structure holds when several AI agents change one repository, the nine integrations with external services each live in their own folder, and the structure rules were moved into 52 tests with no exemptions (as of Oct 5, 2026). Type checking and these tests run on every PR, so a change that breaks a rule fails before merge.

Long code tasks run from a queue

A code task used to run for 30 to 40 minutes inside a tool call, holding its thread and one of the concurrent-session slots for that time. After the change, tasks go into a PostgreSQL queue table and the tool call returns immediately. A separate runner executes them, and a lock is taken per repository only around the git commands that touch the shared copy.

Code changes are submitted only as a PR on an agent-only branch, after a separate review



Writes run only after policy code has checked the conditions; a code change goes through implementation, review and fixes, and is then opened as a PR

Implementation and review are separate sessions

An agent that evaluates its own code change tends to evaluate it favorably. So the review is done by a separate session, which is given the instruction and the diff, not the summary written by the implementing side. The review session's tool list has no file-editing tools (Edit, Write), and the implementing session fixes the problems the review finds.

Each code change happens in a fresh working directory

When a code change is requested, server code creates a new working directory (git worktree) and an agent-only branch in the repository. Each task's changes are made in its own directory, so one task's edits do not mix into another task or into the read-only copy used for answers.

Server code does the push and opens the PR

Pushing to the repository and opening the PR are done by server code after the session ends, not by the model. The shell the model uses cannot reach the repository host. Server code pushes to the agent-only branch; it pushes to the deploy branch only when an alert-triggered automatic fix applies the PR's commits there. Merging happens only when a person asks for it in the thread.

Shell commands run only inside a sandbox

What a model-issued shell command will do cannot be fully vetted in advance. Running a build or tests also executes the repository's build scripts and dependency code, and a code comment or an issue body that the model reads may contain text telling it to run a command. So shell commands run only inside a sandbox in which the operating system limits the process's file and network access. Files can be written only in the working directory and the build cache, and network access goes only to hosts on an allow-list.

The operator's personal settings are not loaded

The agent runs under my user account, as its operator. During the pilot I found that my personal instruction file for my AI coding tool, and information from external tools connected to my own account, had leaked into the agent's sessions. I had the sessions changed to load the project settings only.

A task that hits its limit is not thrown away

Each code task has a limit on the number of model calls and on cost. During the pilot, a task that hit the limit lost every file it had changed. After that, even at the limit, the changed files are kept as a commit, and the commit and PR titles are prefixed with an “incomplete” tag so that a person can take over.

Longer write-ups Part 1: Building the agent · Part 2: Redesigning it

Test database cleanup

Changed how cleanup selects tables; measured the result | Oct – Nov 2026 | Kotlin · PostgreSQL 16 · Testcontainers · JFR

Cleanup truncates only the tables a test wrote to, which took the full test run from 17 min 10 s to 9 min 5 s

Context Tests run against a PostgreSQL instance started in Docker (Testcontainers), and after each test the tables are emptied (TRUNCATE) for the next one.

Problem Running all 6,557 tests took more than 17 minutes. By re-aggregating profiler (JFR) recordings that already existed, I confirmed that most of the time tests spent waiting on the database went to this cleanup. It ran TRUNCATE on all 218 tables after every test.

Cause TRUNCATE does not delete rows one by one; it replaces the table's data file with a new empty file. So even for a table with no rows it creates a new file, updates the system catalog (the internal tables that describe tables) and deletes the old file. That cost is fixed per table, so cleanup time scaled with the number of tables, not the amount of data a test wrote.

Constraints Rolling back the test transaction could not replace cleanup. Tests for after-commit listeners commit the transaction themselves midway, through a shared test helper. Asynchronous (@Async) work and services that run in a new transaction (REQUIRES_NEW) also commit separately from the test transaction. Rows committed this way are not removed by a rollback; with cleanup taken out, 241 of the 4,662 tests at that time failed.

Decision In PostgreSQL, once a row has gone into a table, an 8 KB page stays in the table's file whether the transaction commits or rolls back. A file size of zero means that no row has entered the table since the last cleanup. So cleanup skips tables whose file size is zero and truncates only those above zero, and if there are none it sends no TRUNCATE at all. Because it decides from the database's own state rather than tracking in the application which tables were written, it does not miss writes that bypass Spring's transaction manager.

Check I added a test that checks that cleanup leaves no rows behind. It inserts one row committed in the test transaction and one row committed in a new transaction (a different DB connection), then confirms that no table has rows after cleanup. I also confirmed that this test fails when the cleanup code is deliberately changed to truncate nothing.

Result On the same machine, as the mean of two runs per condition, the full run went from 17 min 10 s (17 min 29 s and 16 min 50 s) to 9 min 5 s (9 min 9 s and 9 min 0 s), 47% shorter. After the change reached main, the longest stage in CI went from 41 min 23 s to 13 min 6 s (one run each).

What changed in a single cleanup

Measure	Before	After
Tables truncated per cleanup	All 218	Median 13, maximum 30
TRUNCATE time per cleanup	Median 163.0 ms, 95th percentile 179.7 ms	Median 17.0 ms, 95th percentile 22.7 ms

Measured over 486 cleanups in the tests of one package.

Options I did not choose, and why

Option	Why not
Clean up only after tests that committed	Writes that bypass Spring's transaction manager are not visible as commits and would be missed. In experiments it was also no faster than truncating only the written tables.
Empty tables with DELETE instead of TRUNCATE	One cleanup is about 15 ms faster (benchmark), but DELETE cannot see rows that another transaction has not yet committed and can leave them behind. TRUNCATE waits for the table lock and removes those rows too.

A hypothesis I dropped after measuring: fsync

fsync is the PostgreSQL setting that makes it wait until changes have actually been written to disk. A separate PostgreSQL container I started for experiments had it on, and turning it off more than halved cleanup time. I added the setting to the test container too, but when I measured again in the real tests, neither the total time nor the TRUNCATE time went down. It turned out that the Testcontainers setup the tests use already starts PostgreSQL with fsync off, so there was nothing left to turn off. My mistake was not starting the experiment environment with the same options as the real one; I removed the setting.

Bedrock call path and the transaction split

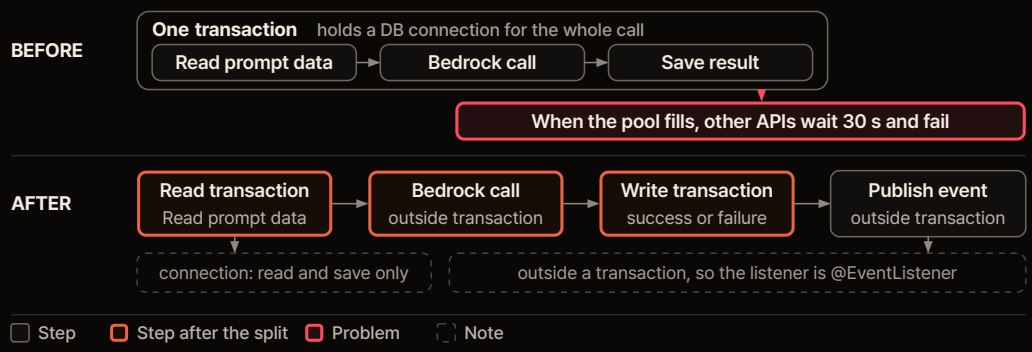
Designed and implemented independently | Oct – Nov 2025

Kotlin · Spring Boot · AWS Bedrock (Converse API) · Coroutines · S3

The server holds no DB connection while it waits for the model

Context The product's AI features call models on AWS Bedrock. A model call is a call to an external API, so the server cannot control how long the response takes or what form it arrives in, and a response takes from a few seconds to tens of seconds. The first row of the table below is a fix to the call structure of an AI feature that already existed; the other two rows come from an AI feature I built.

Area	Existing implementation	Resulting problem	What I changed
Transaction	An existing AI feature wrapped its whole asynchronous job in one transaction and, inside it, read the data for the prompt, called the model and saved the result, in that order.	It held a DB connection the whole time it waited for the model. When such requests overlapped and filled the connection pool, a profile API unrelated to this feature waited 30 seconds for a connection and failed.	I took the model call out of the transaction. Only the data read and the result save run in transactions, one each, and if the call fails, a failure status is saved.
Response format	The AI feature I built takes the model's response as JSON.	When the model's response could not be parsed as JSON, the raw error reached the user's screen unchanged.	On a JSON parse error only, the call is retried, up to 3 attempts in total including the first; if all fail, the API returns a defined error code.
Attachment upload	The first implementation of the same feature uploaded attachments to S3 one at a time.	Upload time grew with the number of files.	Only the upload step runs concurrently, with Kotlin coroutines (async/awaitAll). Making the whole server asynchronous would have meant rewriting blocking infrastructure such as JDBC and JPA, and other domains with it, so I kept Spring MVC.



With the model call outside the transaction, a DB connection is used only while reading and saving

What I missed when I made the split

After the split, the event announcing that generation had finished was published with no transaction in progress. But the listener that receives it was declared to run only after a transaction commits (@TransactionalEventListener AFTER_COMMIT). Such a listener does not run when there is no transaction in progress, so the completion event was dropped without any error.

The listener does not read the database again; it uses only the values carried in the event, so the after-commit condition was not needed. I changed it to a plain listener (@EventListener) and, at the same time, added logging of the time each step takes.

Result and what followed

No DB connection is held while the model is being called. I documented this pattern (an asynchronous job wrapped in a transaction that makes an external call inside it) in the repository's anti-pattern document, with the cause and the alternative, and split one other external call the same way.

Bedrock calls were gathered behind one shared interface, and three of the product's AI features call the model through it. The function that extracts JSON from a response is shared as well. Prompts, and the retry loop for JSON parse failures, are kept per feature.

ai-ready: rules moved from docs into tests

Planned, designed and maintained | May – Oct 2026 | Applied to the company's backend repository | Kotlin · ArchUnit · ktlint

Six recurring rules became checks that fail on a violation; context that checks cannot express went into per-module docs

Problem My teammates built features with AI coding agents. A model's output is not the same every time, so an agent could read a rule written in the docs and still break it. Rules in docs are not enforced, and some of them cause errors in production when they are broken.

Decision 1 If code that breaks a rule fails at compile time, in lint or in a test, it is caught the same way no matter how the model writes it. So I deleted from the docs the rules that the compiler and existing tests already catch, and moved six recurring rules that existed only in docs into architecture tests (ArchUnit). The repository guide also states the principle: if a check can prevent it, add the check and do not write it in the docs.

Decision 2 Context that cannot be seen from the code alone, such as why something was built the way it was or what must not be done in a given module, cannot be moved into checks, so it stayed in docs. Putting all of it in one document at the repository root would load instructions unrelated to the task into the model's input (context) every time. So each Gradle module has its own guide (AGENTS.md), holding the instructions needed for work in that module, and the root document keeps only short common rules and a table that points to the docs to read for each kind of task.

Result Breaking any of the six rules makes the architecture tests fail. These tests, ktlint, compilation, and the doc link and format checks are bundled into one command (scripts/verify.sh) to run before a PR.



A rule in the docs can be broken and the build still passes; a rule moved into a check fails when it is broken

The six rules moved into tests

Four of them: use a converter instead of @Enumerated on enum columns, do not nest DTOs inside controllers, put @Enable* only on configuration classes, and do not read Excel cells directly as strings. The other two forbid the Querydsl transform(groupBy) that throws at runtime on Hibernate 6, and multi-key EXISTS and UNLINK on a Redis cluster. Before moving them, I first removed the existing code that the new tests would flag (unused imports that had been left behind), so the tests started from zero violations.

What stayed in docs, and the published tool

I compared the rules in the docs with the code and configuration and sorted them into four groups: already enforced by a check, can be moved into a check, cannot be checked automatically, and out of date with the code. Rules that cannot be checked automatically stayed in docs, and docs that had drifted from the code were corrected to match it.

ai-ready is the tool that runs these two steps, audit and apply, on a repository's documented rules; it is public on GitHub.

GitHub kunsanglee/ai-ready ↗