Tutorials

Using AI Configs to review database changes

Published August 20th, 2025

by Kevin Kruger

Overview

At LaunchDarkly, we're constantly pushing the boundaries of what it means to move fast without breaking things. We ship frequently, serve **quadrillions of events per day**, and operate with **zero tolerance for downtime**. To keep pace, we need systems that help us ship confidently—even when change is happening at a rapid, "vibe coding" pace.

But what happens when that rapid change reaches your database?

From an SRE's perspective, the database is sacred. It's the source of truth—and one of the riskiest areas to touch without deep context. Even if your code is reviewed in a pull request:

- Does the reviewer understand the query access patterns?
- Could this schema change hurt index performance?
- Is the change touching critical production tables?
- Will the new model scale with usage?

With LaunchDarkly AI Configs, we finally have a way to automate this kind of insight — *reviewing database changes before they become production issues*.

Prerequisites

A LaunchDarkly account with AI Configs enabled. Sign up for free here.

- Access to a database system. These code samples are written to be compatible with a database using the PostgreSQL wire protocol. However, you could adapt them to suit other flavors of databases.
- Basic familiarity with SQL and database schema management
- A development environment where you can test database changes

What are AI Configs?

LaunchDarkly AI Configs allow you to customize, test, and roll out new large language models (LLMs) within your generative AI applications.

🧠 What the AI Reviewer Checks

Our system uses LaunchDarkly's internal AI Configs to analyze your schema and query changes directly from a CI build. It checks for:

- Are the **queries** optimized for access patterns?
- Are the right indexes in place?
- Are we modifying high-risk tables?
- Will the schema scale and evolve over time?

This isn't just a linter. It's an AI-powered reviewer trained on your environment.

If you want to skip right to reading the code, a complete example can be found here.

🛠️ Step 1: Collect the Right Data

The AI needs a complete snapshot of your system to make a meaningful review:

- Full set of SQL queries (sql-queries.json)
- Query diff (queries-diff.json)

📦 sql-proxy container

To evaluate database changes, we need to observe real SQL queries your application runs during CI.

We do this by inserting a lightweight PostgreSQL proxy between your app and the database. It logs and deduplicates queries, then exposes them via an API for analysis.

Here's the setup in Docker Compose / GitHub Action Service Container:

```
services:
  postgres:
    image: postgres
    env:
      POSTGRES_PASSWORD: postgres
    ports:
      - 5432:5432

  proxy:
    image: ghcr.io/${{ github.repository_owner }}/sql-proxy:latest
    env:
      LISTEN_PORT: 5433
      BACKEND_HOST: postgres
      BACKEND_PORT: 5432
      API_PORT: 8080
      DB_CONNECTION_STRING: host=postgres port=5432 user=postgres passw
    ports:
      - 5433:5433
      - 8080:8080
```

Every query is deduplicated and exposed via: `GET http://localhost:8080/queries`

Connecting to the database via 5433 will now pass the queries through the proxy.

table definitions, columns, indexes, and relationships.

This can be triggered early in the CI pipeline to run in parallel with your other steps:

```
- name: Migrate Database
  run: |
    cd schema_test
    go run .

- name: Start Schema Dump
  run: |
    curl -X POST http://localhost:8080/schema_dump
```

⏱️ **Note:** For large schemas, this can take a minute or two. Triggering it early(but after the migrations) avoids blocking downstream jobs.

📊 Step 2: Compare Against Main

After your tests or migrations run through the proxy, it now holds:

- The **full set of SQL queries** the app executed
- The **current state of the database schema**

With this data captured, you can run a GitHub Action that:

- Calls the proxy's API to fetch the captured data
- Downloads artifacts from the main branch (last known good state)
  - Note: it needs to run on the main branch at least once to have generated a proper artifact for comparison
- Compares current vs. main to generate:
  - `schema-diff.json`

Here's what that looks like in CI:

```yaml
- name: Get SQL Data
  uses: droptableifexists/recon@main
  id: get-sql-data
  with:
    SQL_PROXY_API_ADDRESS: localhost:8080
    GITHUB_REPOSITORY: ${{ github.repository }}
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

🤖 Step 3: Run the AI Review

After you have the four key files, you pass them into the AI Config system.

Save input to a file:

```yaml
- name: Save SQL data to file
  run: |
    cat << EOF > analysis_input.json
    {
      "sql_queries": ${{  toJSON(steps.get-sql-data.outputs.sql-queries
      "queries_diff": ${{  toJSON(steps.get-sql-data.outputs.queries-di
      "schema": ${{  toJSON(steps.get-sql-data.outputs.schema) }},
      "schema_diff": ${{  toJSON(steps.get-sql-data.outputs.schema-diff
    }
    EOF
```

Run the DB analysis tool:

```yaml
- name: Run Database Analysis
  uses: launchdarkly-labs/lddbai@main
  with:
    message: "Analyzing database changes..."
    input_file: analysis_input.json
    openai_api_key: ${{ secrets.OPENAI_API_KEY }}
    launchdarkly_sdk_key: ${{ secrets.LAUNCHDARKLY_SDK_KEY }}
```

Under the hood, here's what the code looks like:

```
def get_ai_config(payload: dict) -> tuple[AIConfig, LDAIConfigTracker]:
    aiclient = Deps().get_launchdarkly_ai()
    context = Context.builder('cockroachdb').kind('database').name('cock
    fallback_value = AIConfig(
        enabled=True,
        model=ModelConfig(name="gpt-4o-mini", parameters={"temperature":
        messages=[LDMessage(role="system", content="")],
        provider=ProviderConfig(name="my-default-provider"),
    )
    return aiclient.config('evaluate-database-changes', context, fallbac
        'schema': payload.get('schema', []),
        'schema_diff': payload.get('schema_diff', []),
        'sql_queries': payload.get('sql_queries', []),
        'queries_diff': payload.get('queries_diff', [])
    })
```

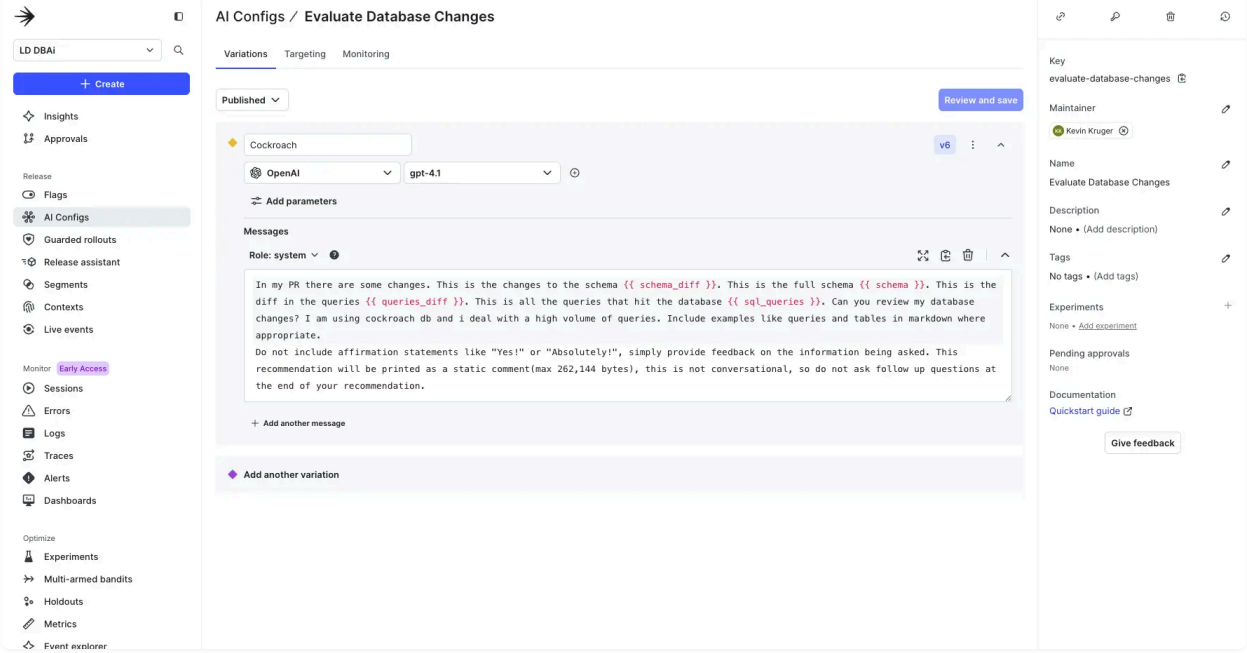
And to get the AI's recommendation:

```
def get_ai_recommendation(payload: dict) -> str:
    config, tracker = get_ai_config(payload)
    messages = [] if config.messages is None else config.messages
    response = tracker.track_openai_metrics(
        lambda: client.chat.completions.create(
            model=config.model.name,
            messages=[message.to_dict() for message in messages],
        )
    )
    Deps().get_launchdarkly().flush()
    return response.choices[0].message.content
```

🧠 Step 4: Add Context That Only You Know

- Database engine and version 🧰
- Critical tables to tread carefully around 🚨
- Average query volume and server specs 📈
- Team-specific data modeling principles 📐
- History of past incidents or patterns 📜

This context transforms the AI from a generic reviewer into a tailored risk advisor for your system.



Screenshot showing the LaunchDarkly UI for creating an AI Config to manage database changes.

🚀 Why This Matters

With AI Configs, you can:

- Catch performance and scaling issues before they hit production
- Share SRE intuition across your whole engineering team

You're no longer at the mercy of "who reviewed the PR." Every change gets a consistent, context-aware review.

🏁 Final Thoughts

Database changes don't have to be scary anymore.

By plugging into LaunchDarkly AI Configs, you can automate reviews, enforce data modeling best practices, and de-risk your deploys—without slowing anyone down. To get started, sign up for a free trial today or email us at `aiproduct@launchdarkly.com` if you have questions.

So yeah, go ahead. Vibe out. Ship confidently. And let the AI handle the hard stuff.

Previous How to implement WebSockets and kill switches in a Py... Next