pganalyze

Postgres Performance Optimization with AI Tools + MCP: Why (not)?

Lukas Fittl
lukas@pganalyze.com

Agenda for today

The Evolution of the "DBA" role

The Problem with Postgres MCP Servers

The need for a better abstraction

Introducing the pganalyze MCP Server

Demo: Investigating Production Database Issues with Claude Code

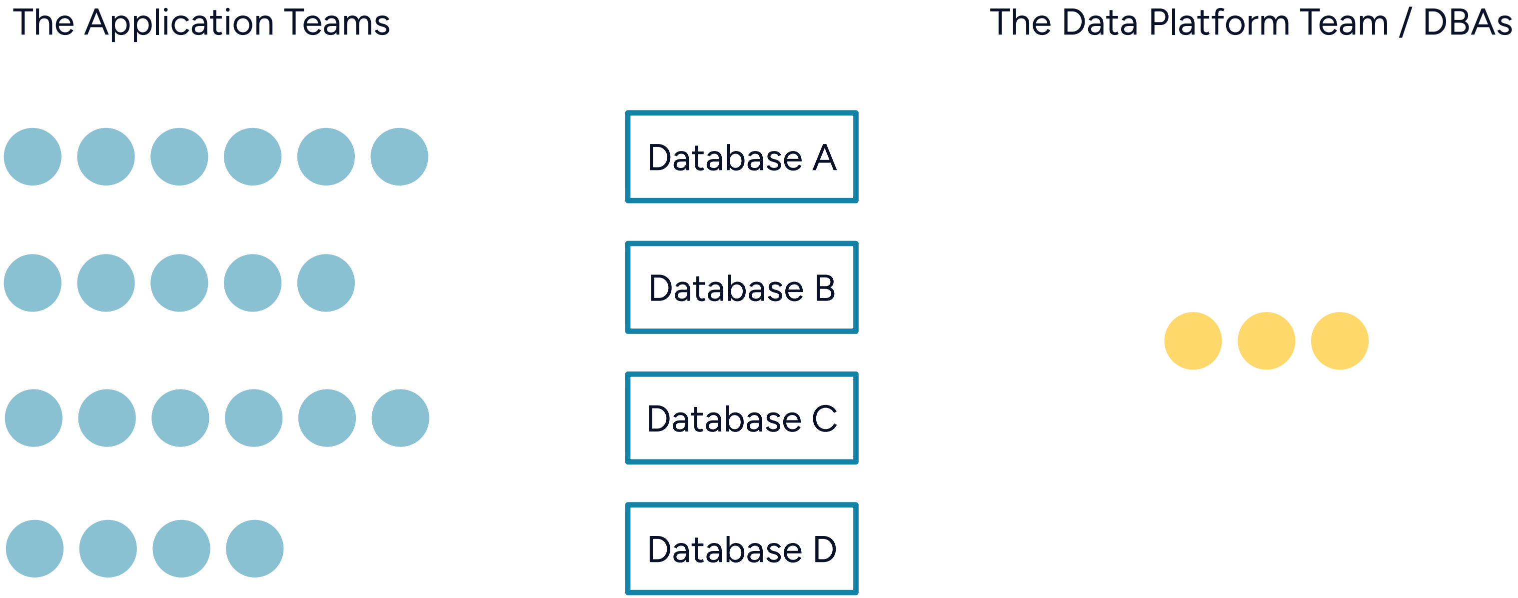
Demo: Combining pganalyze MCP Server with APM traces

What about Agents?

Our vision for pganalyze

The Evolution of the "DBA" role

A Typical Organization Using Postgres



The Application Teams

The Data Platform Team / DBAs

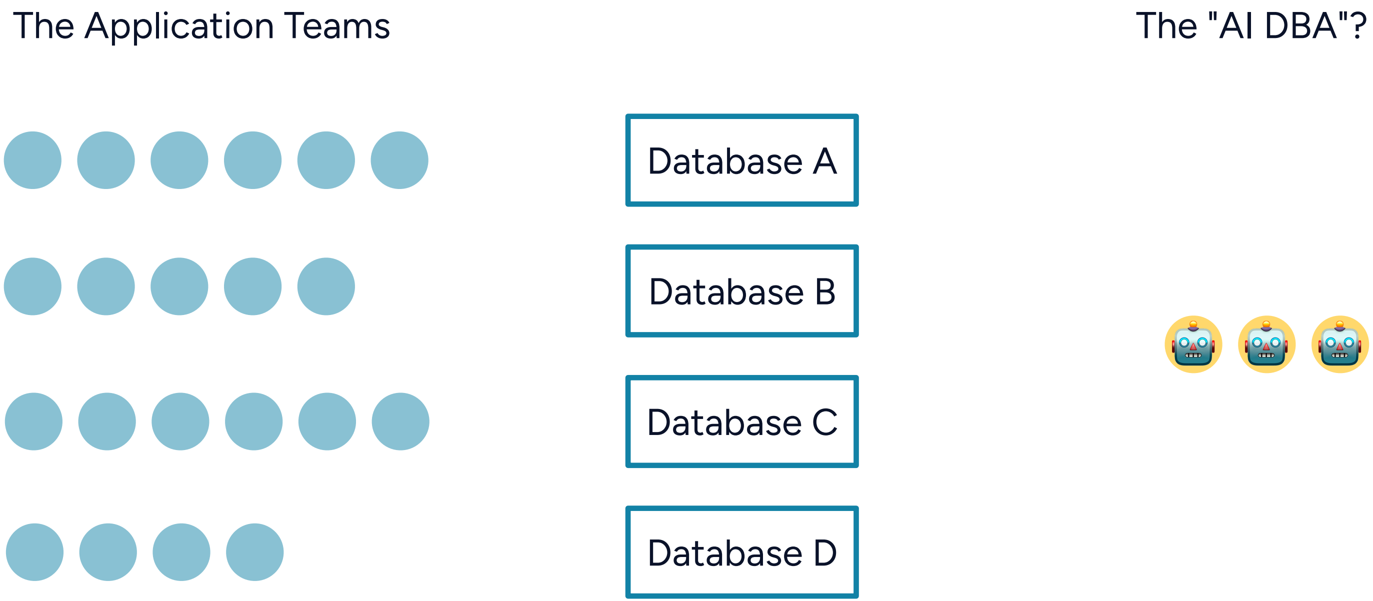
Database A

Database B

Database C

Database D

Is "AI DBA" the solution?



The Application Teams

The "AI DBA"?

Database A

Database B

Database C

Database D

Why don't we have OpenClaw be our AI DBA?

because of the quality (but that's getting better with better models)

because of the risk / threat model
- unintended actions
- data leakage

because organizational knowledge does matter, for production applications

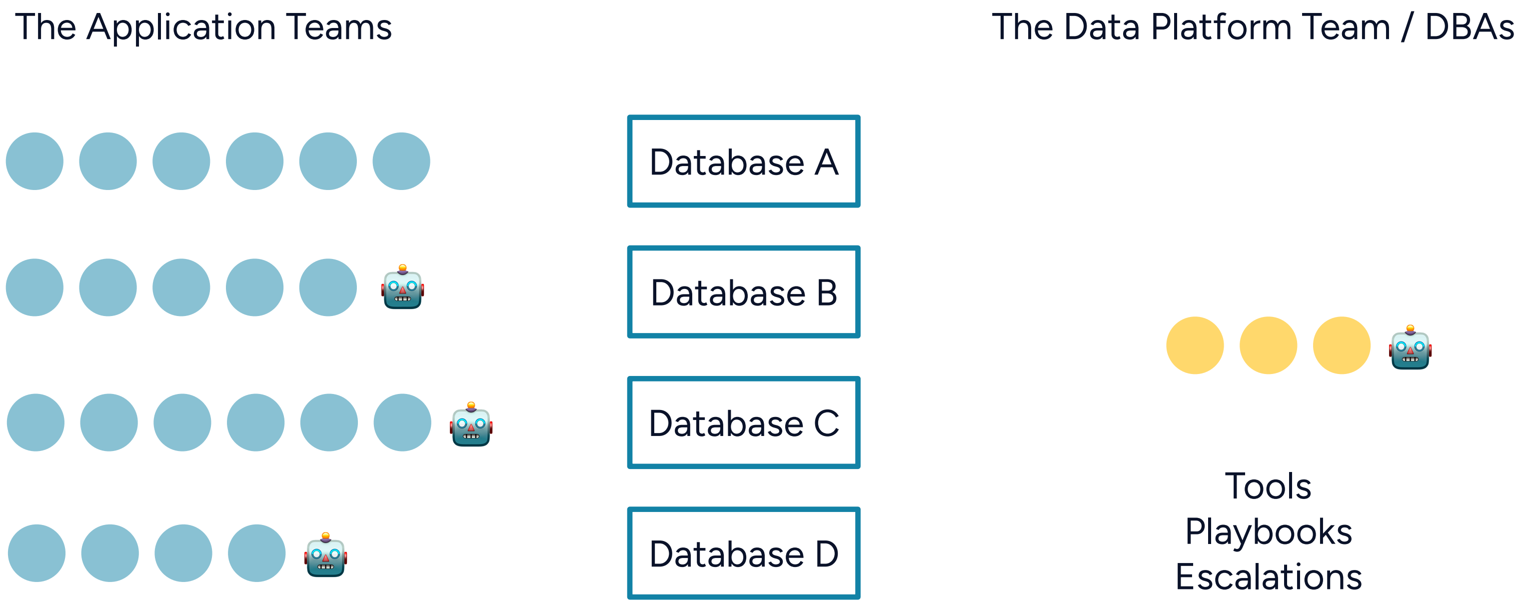
What if we enable engineers, instead of replacing DBAs?

AI makes it easy to make reasonably looking but flawed database changes. If we do nothing, our production-facing teams will be overwhelmed fighting a firehose of changes not thought through.

This is similar to the problem with "AI SRE":

"The vision behind AI SREs is that incidents and outages are one-off exceptions to paper over and move on from, rather than a structural and emergent consequence of what you do (and how you do it) and from which you should learn."
[Fred Herbert - The Picture They Paint Of You]()

What if we instead did this:



The Application Teams

The Data Platform Team / DBAs

Database A

Database B

Database C

Database D

Tools
Playbooks
Escalations

But by weeks 7 or 8, one team hit a wall. They could no longer make even simple changes without breaking something unexpected. When I met with them, the team initially blamed technical debt: messy code, poor architecture, hurried implementations. But as we dug deeper, the real problem emerged: n**o one on the team could explain why certain design decisions had been made** or how different parts of the system were supposed to work together.

The code might have been messy, but the bigger issue was that **the theory of the system, their shared understanding, had fragmented or disappeared entirely.** They had accumulated cognitive debt faster than technical debt, and it paralyzed them.

— **Margaret-Anne Storey on Cognitive Debt** (Feb 2026)

Learning Postgres still matters.

But if we get it right, AI tools can help with increasing understanding.

"People *observing* learn via osmosis, even if they're not actively involved in taking actions." https://hazelweakly.me/blog/stop-building-ai-tools-backwards/

"Computers can be understood" https://blog.nelhage.com/post/computers-can-be-understood/

Best practices can be encoded with skills

Timescale's:
https://github.com/timescale/pg-aiguide

Neon's:
https://github.com/neondatabase/ai-rules

Parade DB:
https://github.com/paradedb/agent-skills

(but models like Claude Opus 4.6 are actually pretty good at Postgres, all by themselves!)

AI is good at tedious tasks

What are the tedious tasks for a DBA?

Structured analysis of a problem

Collecting different EXPLAIN plans

Trying different optimization strategies

Teaching best practices to the teams

Researching the application code

The Problem with Postgres MCP Servers

The "simple" Postgres MCP servers

DBHub
https://github.com/bytebase/dbhub
(database agnostic)

AWS Labs postgres-mcp-server
https://awslabs.github.io/mcp/servers/postgres-mcp-server

Google AlloyDB MCP Server
https://docs.cloud.google.com/alloydb/docs/reference/mcp

Anthropic's Postgres MCP server
https://github.com/modelcontextprotocol/servers-archived/tree/main/src/postgres
(unmaintained)

The "do it all" Postgres MCP servers

pgEdge Postgres MCP

https://github.com/pgEdge/pgedge-postgres-mcp

CrystalDBA Postgres MCP

https://github.com/crystaldba/postgres-mcp

Comparing Postgres MCP Tools

	"Run a query"	"Run EXPLAIN"	"Get Schema"	"Get docs"
pgEdge Postgres MCP	query_database	execute_explain	get_schema_info	search_knowledge..
DBHub	execute_sql	-	search_objects	-
Anthropic's Original Postgres MCP	query	-	Table Schemas resource	-
CrystalDBA Postgres MCP	execute_sql	explain_query	list_objects	-
AWS Labs pg-mcp-server	run_query	-	get_table_schema	

Comparing Postgres MCP Tools - "Read Only"

	"Run a query"	Safety Measure	Protects against writes?
pgEdge Postgres MCP	query_database	READ ONLY TRANSACTION	🛑 No! (multi-statement strings)
DBHub	execute_sql	Query keyword check	🛑 No! (wCTEs, functions)
Anthropic's Original Postgres MCP	query	READ ONLY TRANSACTION	🛑 No! (multi-statement strings)
CrystalDBA Postgres MCP	execute_sql	READ ONLY TRANSACTION + multi-statement check	Maybe?
AWS Labs pg-mcp-server	run_query	Query keyword check	🛑 No! (EXECUTE in a DO block, functions)

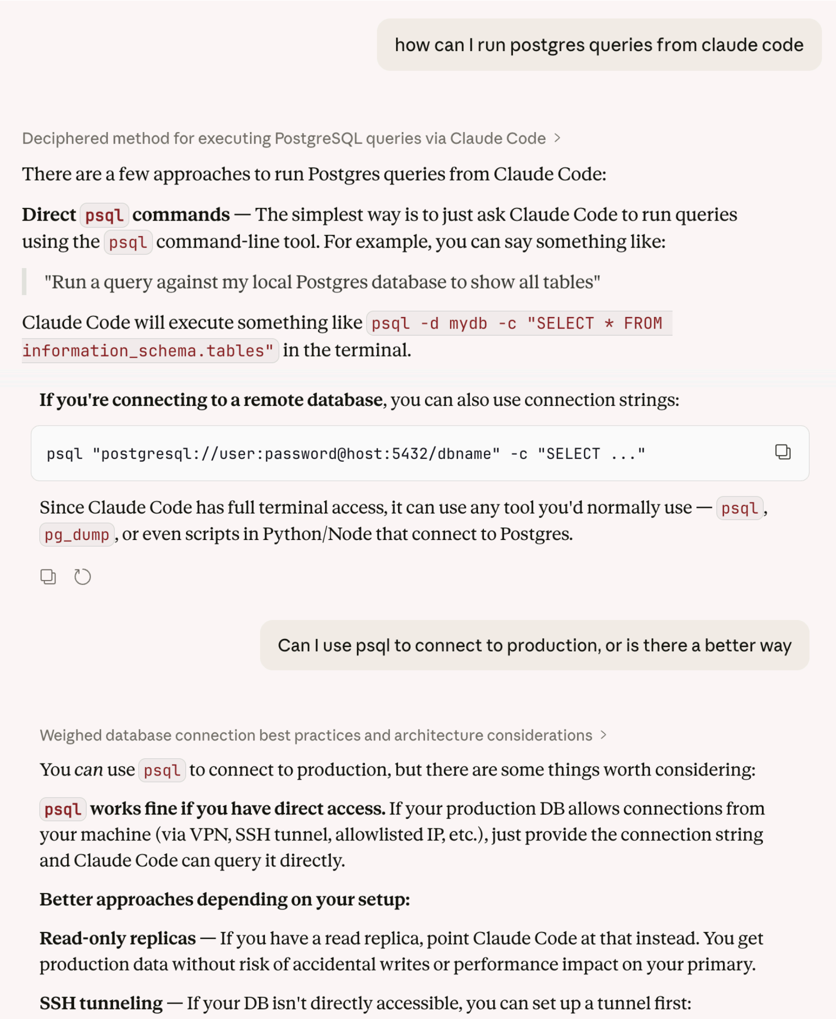
Comparing Postgres MCP Tools - Maintenance

	Last Update	Maintained?	Notes
pgEdge Postgres MCP	Feb 2026	✅ Yes	
DBHub	Feb 2026	✅ Yes	
Anthropic's Original Postgres MCP	Apr 2025	🛑 No	Archived May 2025
CrystalDBA Postgres MCP	Jan 2026	Maybe?	CrystalDBA acquired by Temporal Sept 2025
AWS Labs pg-mcp-server	Feb 2026	✅ Yes	

Always restrict access on the database itself, not just via the MCP server

Why not just use psql?

Let's ask Claude Opus 4.6



how can I run postgres queries from claude code

Deciphered method for executing PostgreSQL queries via Claude Code >

There are a few approaches to run Postgres queries from Claude Code:

Direct `psql` commands — The simplest way is to just ask Claude Code to run queries using the `psql` command-line tool. For example, you can say something like:

> "Run a query against my local Postgres database to show all tables"

Claude Code will execute something like `psql -d mydb -c "SELECT * FROM information_schema.tables"` in the terminal.

If you're connecting to a remote database, you can also use connection strings:

```
psql "postgresql://user:password@host:5432/dbname" -c "SELECT ..."
```

Since Claude Code has full terminal access, it can use any tool you'd normally use — `psql`, `pg_dump`, or even scripts in Python/Node that connect to Postgres.

Can I use psql to connect to production, or is there a better way

Weighed database connection best practices and architecture considerations >

You *can* use `psql` to connect to production, but there are some things worth considering:

`psql` works fine if you have direct access. If your production DB allows connections from your machine (via VPN, SSH tunnel, allowlisted IP, etc.), just provide the connection string and Claude Code can query it directly.

Better approaches depending on your setup:

Read-only replicas — If you have a read replica, point Claude Code at that instead. You get production data without risk of accidental writes or performance impact on your primary.

SSH tunneling — If your DB isn't directly accessible, you can set up a tunnel first:

Context can leak!

Never pass credentials directly through an LLM

(use a CLI tool as a wrapper)

All of this is nice and fun.

Until we want to access production.

It's very likely your InfoSec team **forbids using any Postgres MCP servers to connect to production servers.**

That presents a challenge for real-world database debugging with AI tools.

Different layers of data access

destructive access (write/DDL)

unrestricted read access

monitoring + explain access

monitoring + explain filtered access

metadata only access

Different layers of data access

destructive access (write/DDL)

unrestricted read access

monitoring + explain access

monitoring + explain filtered access

metadata only access

The need for a better **abstraction**

Tools required for Performance Optimization

Query performance statistics (pg_stat_statements, wait events)

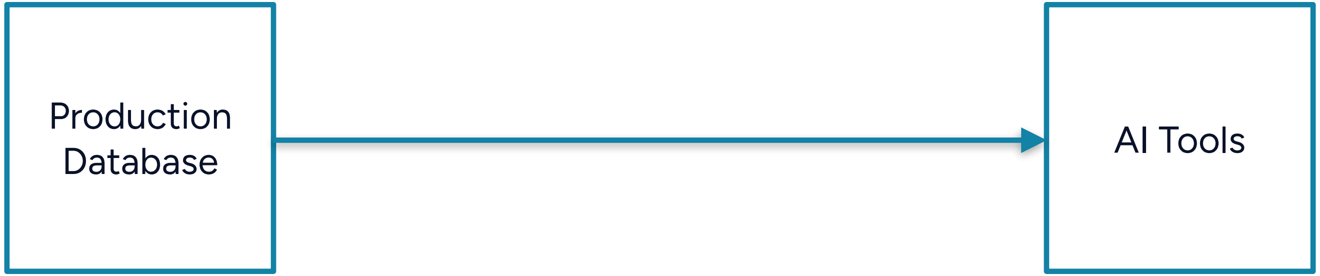
Slow query plans (auto_explain from logs)

EXPLAIN ANALYZE

Production schema information (sizes, statistics)

System metrics

Tools required for Performance Optimization


Production Database

AI Tools

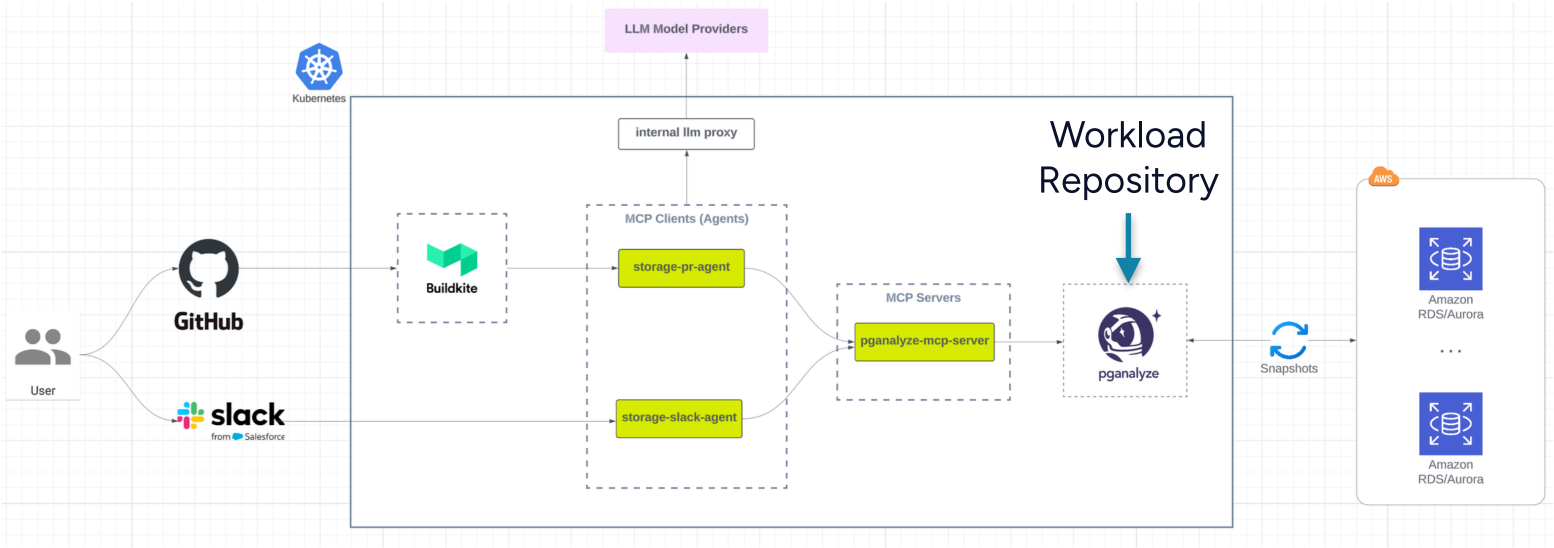
Tools required for Performance Optimization


Production Database

Workload Repository

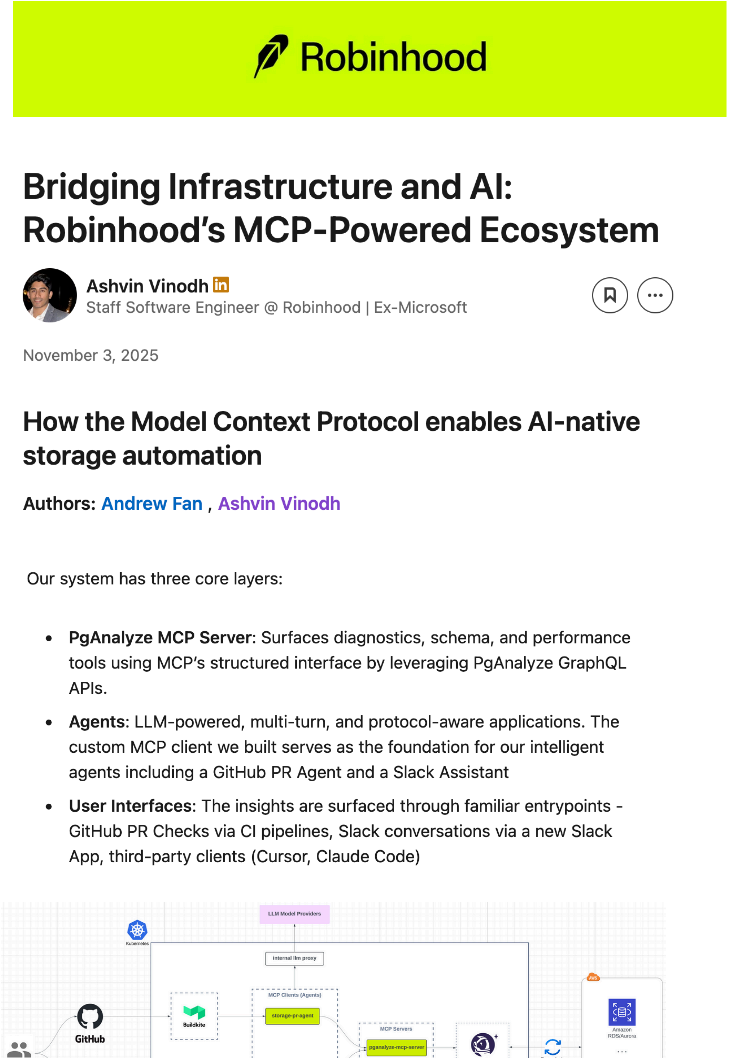
AI Tools

An example architecture from Robinhood



We didn't start out writing an MCP Server for pganalyze, a customer (Robinhood) did.

Link



Robinhood

Bridging Infrastructure and AI: Robinhood's MCP-Powered Ecosystem

Ashvin Vinodh
Staff Software Engineer @ Robinhood | Ex-Microsoft

November 3, 2025

How the Model Context Protocol enables AI-native storage automation

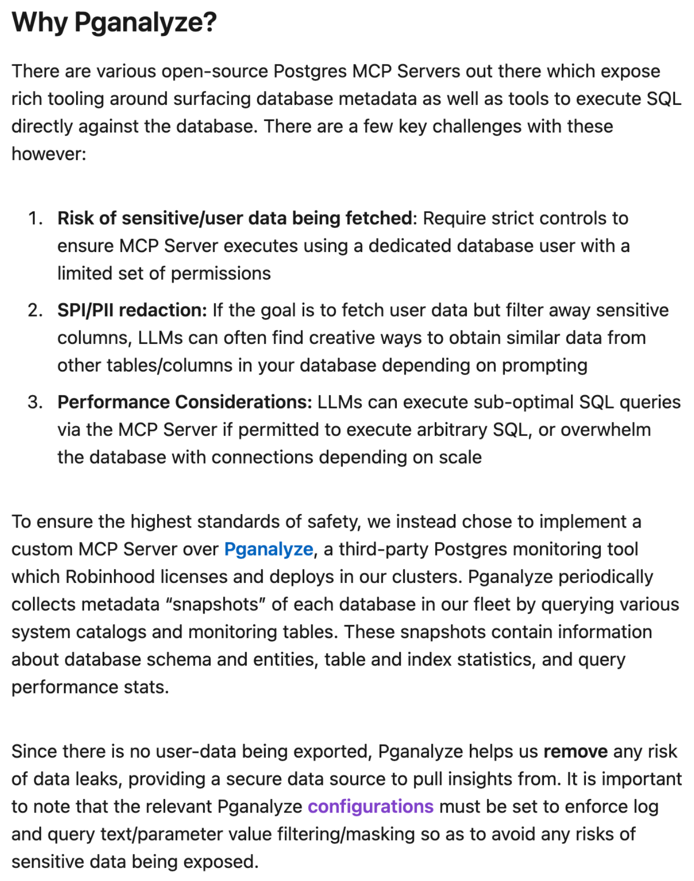
Authors: Andrew Fan , Ashvin Vinodh

Our system has three core layers:

- **PgAnalyze MCP Server**: Surfaces diagnostics, schema, and performance tools using MCP's structured interface by leveraging PgAnalyze GraphQL APIs.
- **Agents**: LLM-powered, multi-turn, and protocol-aware applications. The custom MCP client we built serves as the foundation for our intelligent agents including a GitHub PR Agent and a Slack Assistant
- **User Interfaces**: The insights are surfaced through familiar entrypoints - GitHub PR Checks via CI pipelines, Slack conversations via a new Slack App, third-party clients (Cursor, Claude Code)

We didn't start out writing an MCP Server for pganalyze, a customer (Robinhood) did.

Link



Why Pganalyze?

There are various open-source Postgres MCP Servers out there which expose rich tooling around surfacing database metadata as well as tools to execute SQL directly against the database. There are a few key challenges with these however:

1. **Risk of sensitive/user data being fetched**: Require strict controls to ensure MCP Server executes using a dedicated database user with a limited set of permissions
2. **SPI/PII redaction:** If the goal is to fetch user data but filter away sensitive columns, LLMs can often find creative ways to obtain similar data from other tables/columns in your database depending on prompting
3. **Performance Considerations:** LLMs can execute sub-optimal SQL queries via the MCP Server if permitted to execute arbitrary SQL, or overwhelm the database with connections depending on scale

To ensure the highest standards of safety, we instead chose to implement a custom MCP Server over Pganalyze, a third-party Postgres monitoring tool which Robinhood licenses and deploys in our clusters. Pganalyze periodically collects metadata “snapshots” of each database in our fleet by querying various system catalogs and monitoring tables. These snapshots contain information about database schema and entities, table and index statistics, and query performance stats.

Since there is no user-data being exported, Pganalyze helps us **remove** any risk of data leaks, providing a secure data source to pull insights from. It is important to note that the relevant Pganalyze configurations must be set to enforce log and query text/parameter value filtering/masking so as to avoid any risks of sensitive data being exposed.

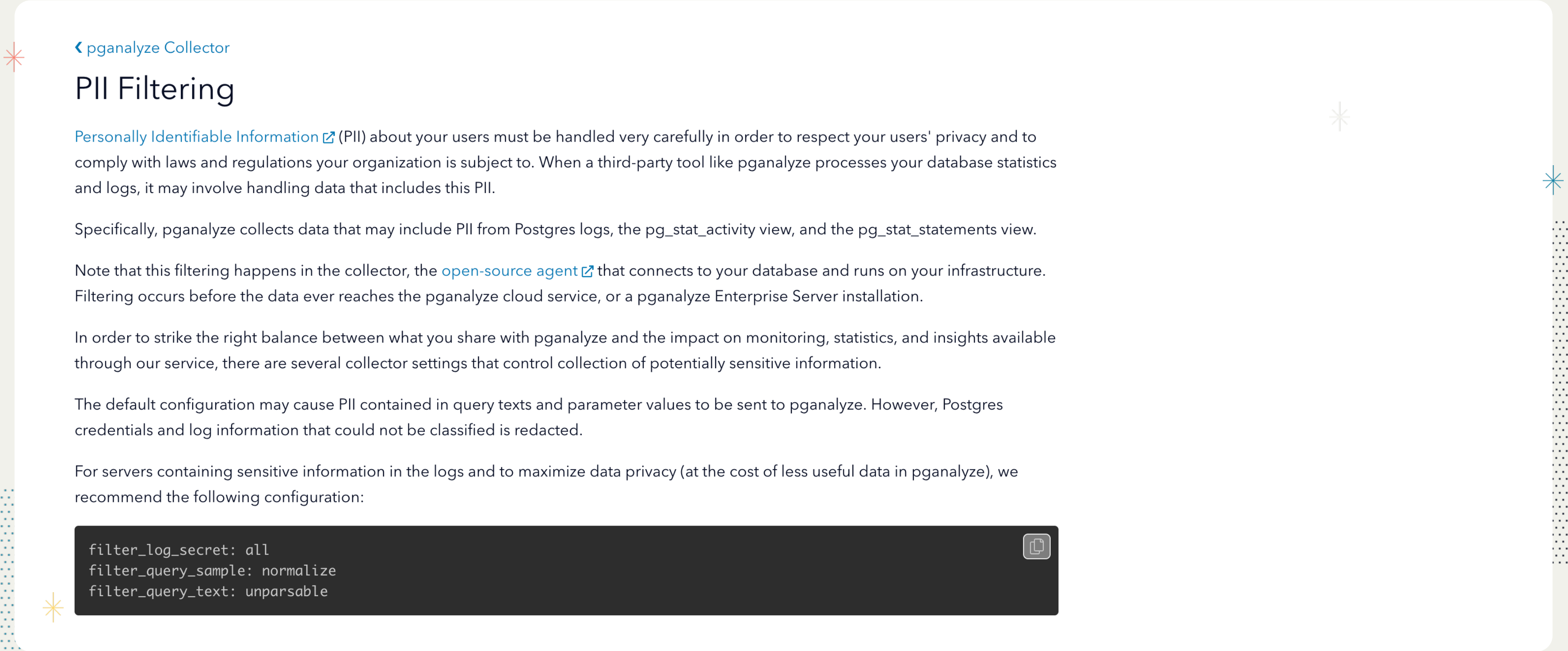
Introducing the
pganalyze MCP Server

The pganalyze MCP Server

The pganalyze MCP Server

Key tools:
- get_query_stats
- get_table_stats
- get_backends (wait events)
- get_query_explains
- get_query_explain_from_trace

What if we filtered out PII before it reaches the LLM?



‹ pganalyze Collector

PII Filtering

Personally Identifiable Information (PII) about your users must be handled very carefully in order to respect your users' privacy and to comply with laws and regulations your organization is subject to. When a third-party tool like pganalyze processes your database statistics and logs, it may involve handling data that includes this PII.

Specifically, pganalyze collects data that may include PII from Postgres logs, the pg_stat_activity view, and the pg_stat_statements view.

Note that this filtering happens in the collector, the open-source agent that connects to your database and runs on your infrastructure. Filtering occurs before the data ever reaches the pganalyze cloud service, or a pganalyze Enterprise Server installation.

In order to strike the right balance between what you share with pganalyze and the impact on monitoring, statistics, and insights available through our service, there are several collector settings that control collection of potentially sensitive information.

The default configuration may cause PII contained in query texts and parameter values to be sent to pganalyze. However, Postgres credentials and log information that could not be classified is redacted.

For servers containing sensitive information in the logs and to maximize data privacy (at the cost of less useful data in pganalyze), we recommend the following configuration:

```
filter_log_secret: all
filter_query_sample: normalize
filter_query_text: unparsable
```

Principles

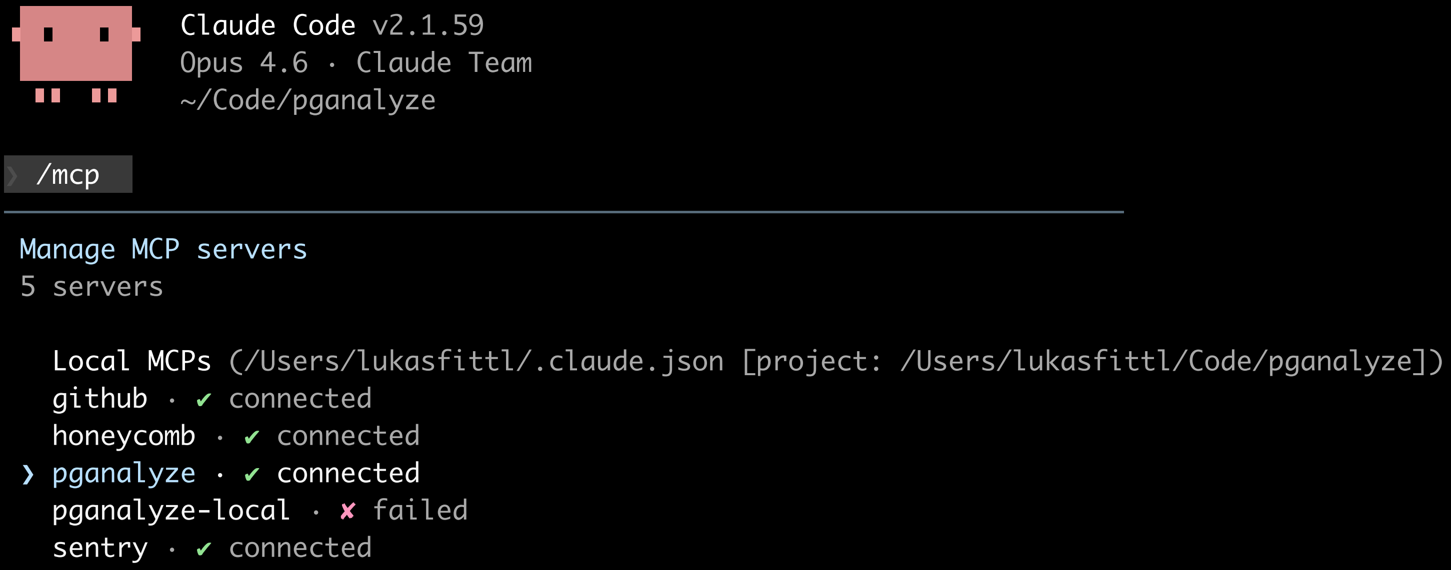
1. Control: Call LLMs in the user's environment where feasible, you can always turn the API off.

2. **Authorization**: the default flow should be to authorize a subset of what a **user** is allowed to see.

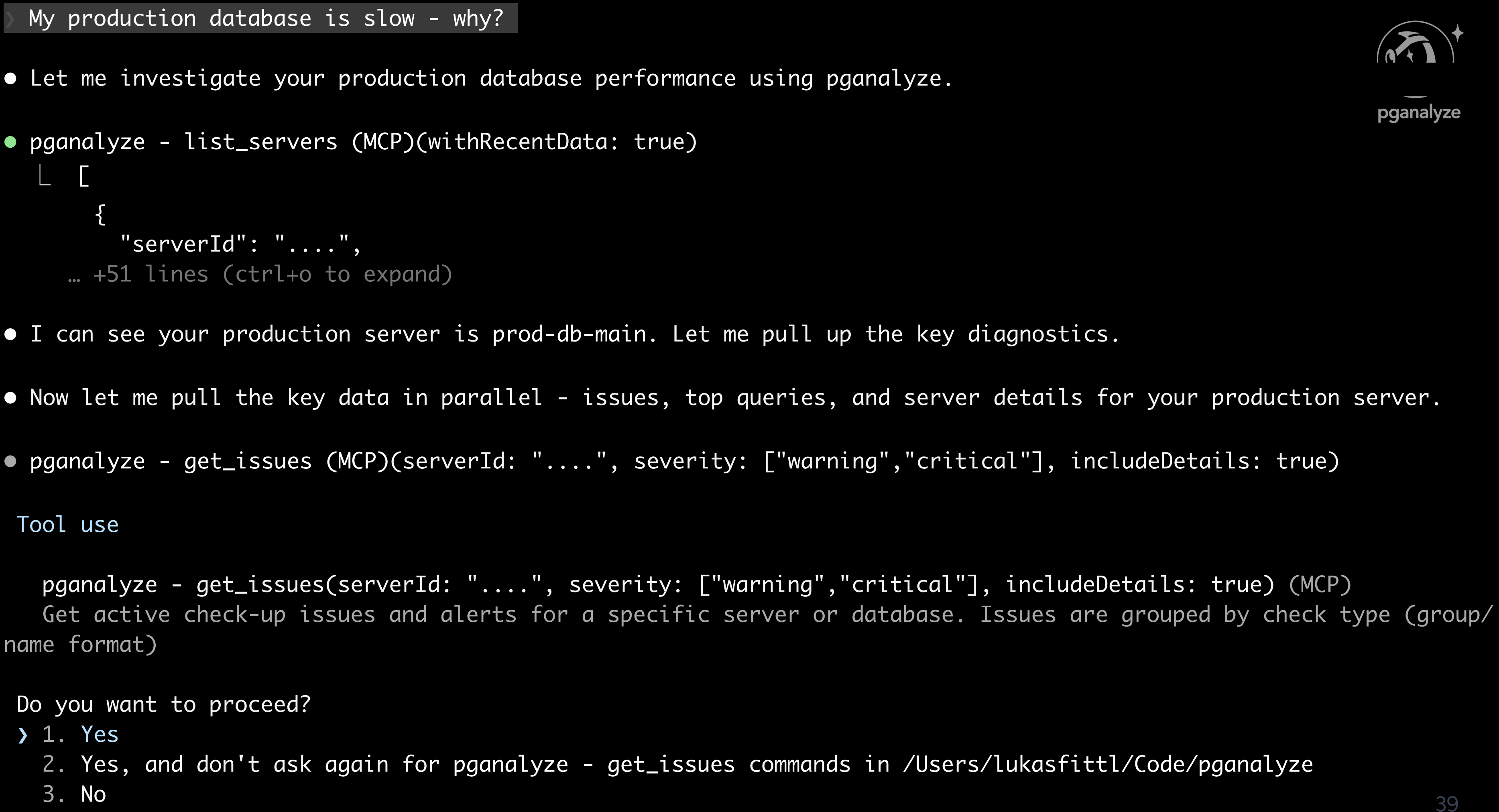
3. **Filtering**. Any end customer PII should be filtered out before it passes to the LLM.

[Demo] Investigating **Production Database Issues with Claude Code**

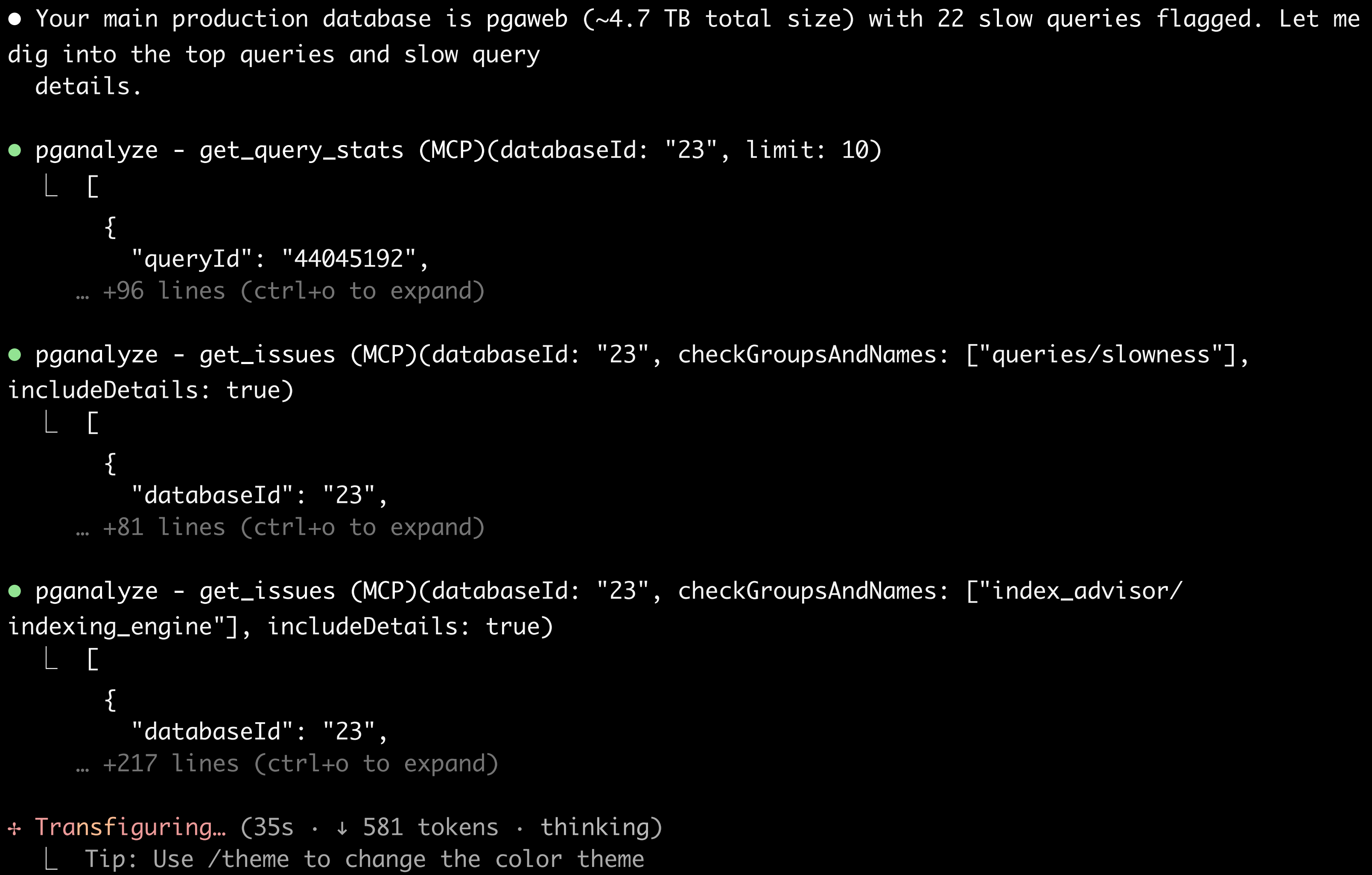
Let's go!



```
      Claude Code v2.1.59
      Opus 4.6 · Claude Team
      ~/Code/pganalyze

❯ /mcp
────────────────────────────────────────────────────────────

 Manage MCP servers
 5 servers

   Local MCPs (/Users/lukasfittl/.claude.json [project: /Users/lukasfittl/Code/pganalyze])
   github · ✔ connected
   honeycomb · ✔ connected
 ❯ pganalyze · ✔ connected
   pganalyze-local · ✘ failed
   sentry · ✔ connected
```



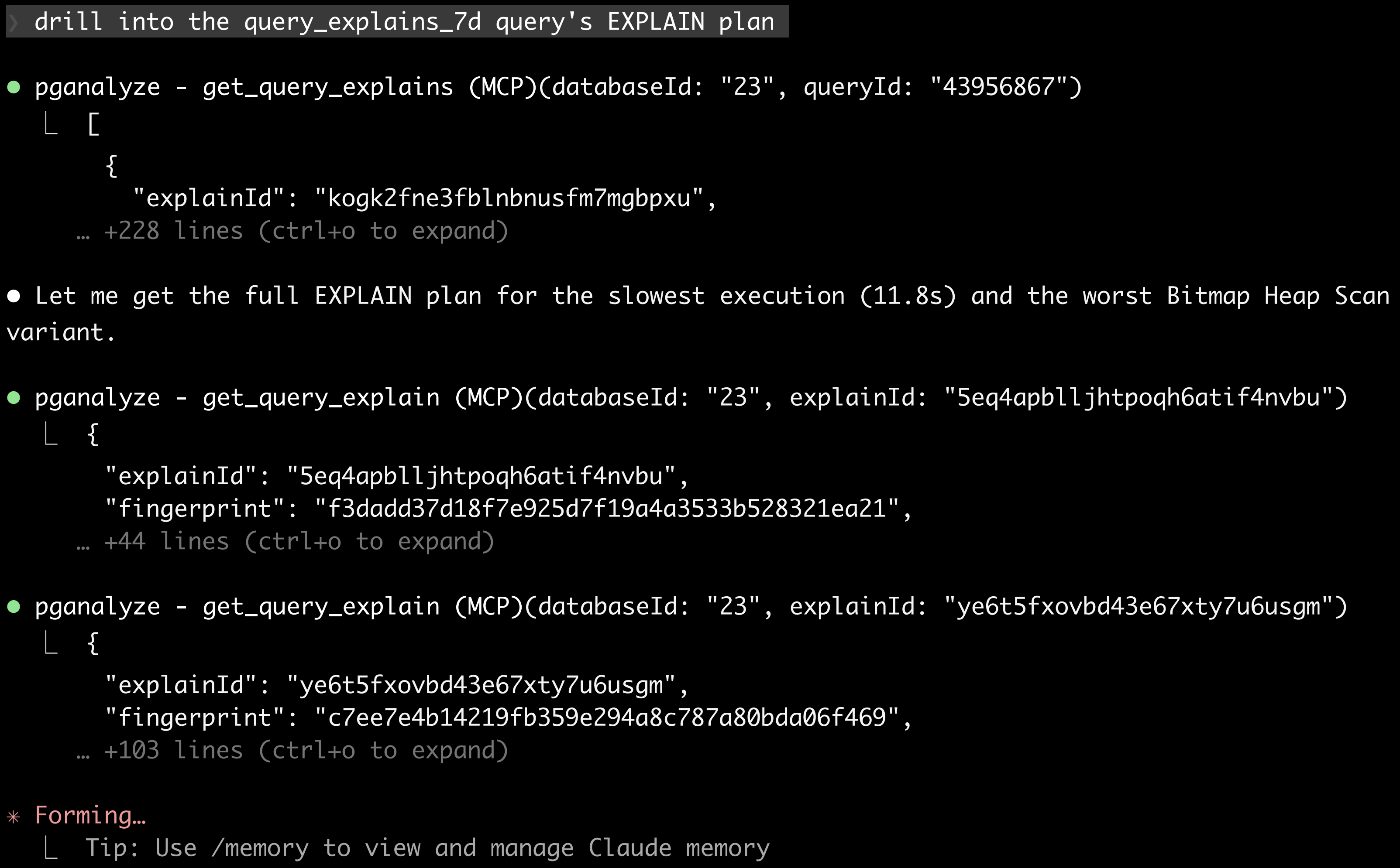
```
> My production database is slow - why?

● Let me investigate your production database performance using pganalyze.

● pganalyze - list_servers (MCP)(withRecentData: true)
  └ [
      {
        "serverId": "....",
     … +51 lines (ctrl+o to expand)

● I can see your production server is prod-db-main. Let me pull up the key diagnostics.

● Now let me pull the key data in parallel - issues, top queries, and server details for your production server.

● pganalyze - get_issues (MCP)(serverId: "....", severity: ["warning","critical"], includeDetails: true)

 Tool use

   pganalyze - get_issues(serverId: "....", severity: ["warning","critical"], includeDetails: true) (MCP)
   Get active check-up issues and alerts for a specific server or database. Issues are grouped by check type (group/
name format)

 Do you want to proceed?
 ❯ 1. Yes
   2. Yes, and don't ask again for pganalyze - get_issues commands in /Users/lukasfittl/Code/pganalyze
   3. No
```



```
● Your main production database is pgaweb (~4.7 TB total size) with 22 slow queries flagged. Let me
dig into the top queries and slow query
  details.

● pganalyze - get_query_stats (MCP)(databaseId: "23", limit: 10)
  ⎿  [
       {
         "queryId": "44045192",
     … +96 lines (ctrl+o to expand)

● pganalyze - get_issues (MCP)(databaseId: "23", checkGroupsAndNames: ["queries/slowness"],
includeDetails: true)
  ⎿  [
       {
         "databaseId": "23",
     … +81 lines (ctrl+o to expand)

● pganalyze - get_issues (MCP)(databaseId: "23", checkGroupsAndNames: ["index_advisor/
indexing_engine"], includeDetails: true)
  ⎿  [
       {
         "databaseId": "23",
     … +217 lines (ctrl+o to expand)

✢ Transfiguring… (35s · ↓ 581 tokens · thinking)
  ⎿  Tip: Use /theme to change the color theme
```

```
  Biggest Concerns

  1. query_explains_7d ranked scan (Query #43956867) - The single worst query. Averages 2.7 seconds with
a 46.7% buffer hit ratio, meaning over
  half of reads go to disk. It scans query_explains_7d filtering on jsonb_array_length(node_insights)
with a window function. This is the
  automated EXPLAIN check-up query.

  2. Old snapshot cleanup (Query #44015223) - DELETE cascade from query_occurrences averaging 1,065 ms
with spikes up to 616 seconds. The delete_old_snapshots service is doing heavy write work.

  3. Queries lookup with unnest (Query #43990962) - 571 ms avg, 93.7% buffer hit ratio, and a worst case
of 577 seconds. This suggests the queries table working set doesn't fit well in memory.

  Missing Indexes (14 tables flagged)

  The Index Advisor found missing indexes on several tables, with the highest potential improvements on:

  - database_stats_35d - 239x estimated improvement
  - users - 215x improvement (e.g., stripe_customer_id lookup on organizations)
  - schema_table_scan_query_associations - 145x improvement
  - log_lines_packed_7d - 143x improvement
  - vacuum_run_stats_35d - 92x improvement
  - query_plan_stats_packed_35d - 89x improvement
  - log_stats_7d - 87x improvement
```

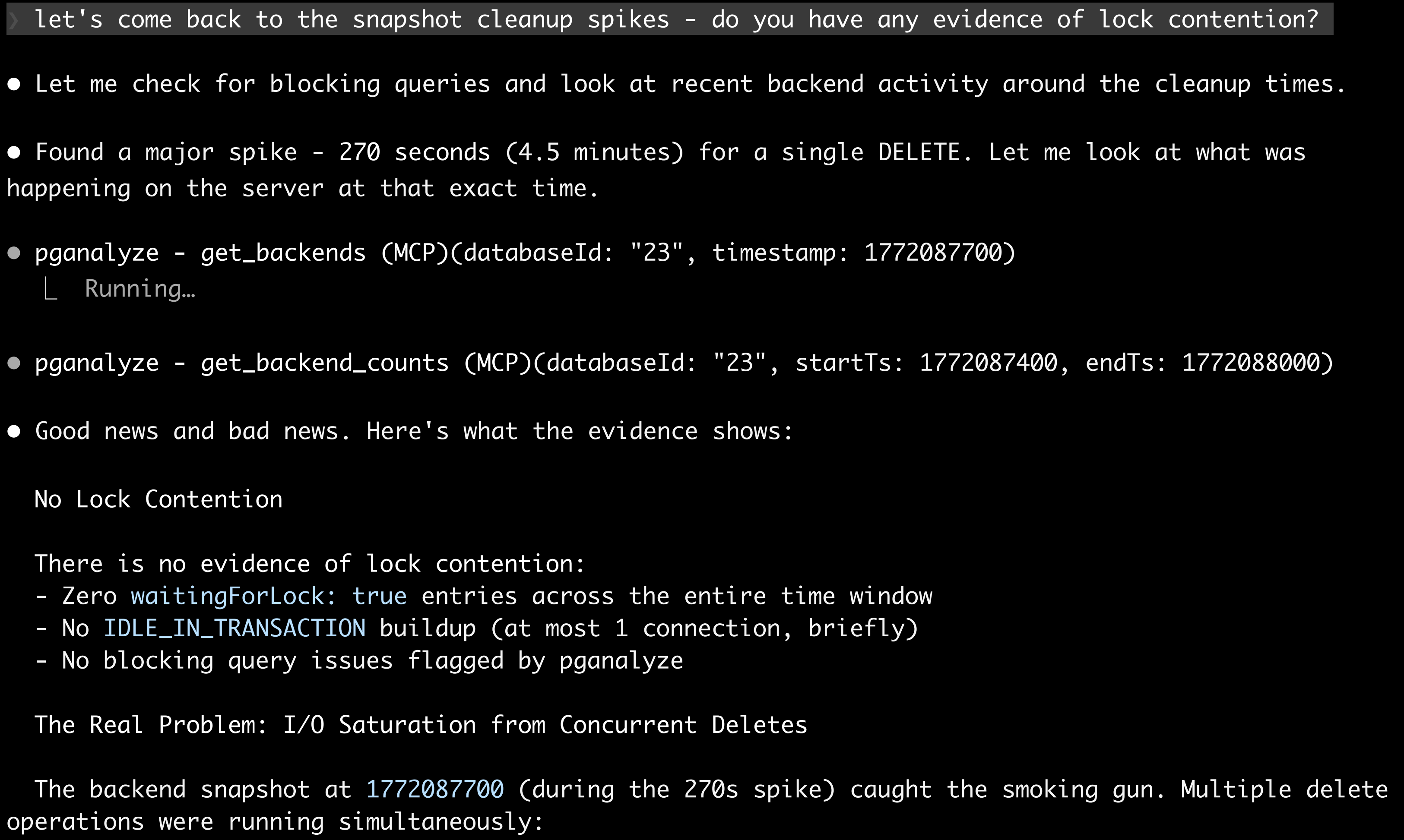
```
  Recommendations

  1. Address the query_explains_7d scan - The 46.7% buffer hit ratio suggests this table's hot data
exceeds available shared buffers. Consider
  adding an index that supports the server_id + seen_at filter, or pre-computing
jsonb_array_length(node_insights) > 0.
  2. Review the missing index suggestions in pganalyze's Index Advisor - especially for
database_stats_35d, log_lines_packed_7d, and
  schema_table_scan_query_associations which have the highest estimated improvements.
  3. Investigate the snapshot cleanup spikes - The DELETE cascade (max 616s) could be causing lock
contention. Consider batching the deletes more
  aggressively.

  Would you like me to drill into any specific query's EXPLAIN plan or look at the missing index
recommendations in more detail?

✻ Churned for 58s
```



```
› drill into the query_explains_7d query's EXPLAIN plan

● pganalyze - get_query_explains (MCP)(databaseId: "23", queryId: "43956867")
  ⎿  [
       {
         "explainId": "kogk2fne3fblnbnusfm7mgbpxu",
     … +228 lines (ctrl+o to expand)

● Let me get the full EXPLAIN plan for the slowest execution (11.8s) and the worst Bitmap Heap Scan
variant.

● pganalyze - get_query_explain (MCP)(databaseId: "23", explainId: "5eq4apblljhtpoqh6atif4nvbu")
  ⎿  {
       "explainId": "5eq4apblljhtpoqh6atif4nvbu",
       "fingerprint": "f3dadd37d18f7e925d7f19a4a3533b528321ea21",
     … +44 lines (ctrl+o to expand)

● pganalyze - get_query_explain (MCP)(databaseId: "23", explainId: "ye6t5fxovbd43e67xty7u6usgm")
  ⎿  {
       "explainId": "ye6t5fxovbd43e67xty7u6usgm",
       "fingerprint": "c7ee7e4b14219fb359e294a8c787a80bda06f469",
     … +103 lines (ctrl+o to expand)

✳ Forming…
  ⎿  Tip: Use /memory to view and manage Claude memory
```

```
● Here's the full analysis of the two plan variants:

  Query

  WITH ranked AS (
    SELECT qe.*,
           row_number() OVER (PARTITION BY qe.query_fingerprint, qe.database_id
                              ORDER BY qe.runtime_ms DESC) AS rn
      FROM query_explains_7d qe
     WHERE qe.server_id = $1
       AND qe.seen_at BETWEEN $2 AND $3
       AND jsonb_array_length(qe.node_insights) > $4
  )
  SELECT * FROM ranked WHERE rn <= $5

  Plan Variant 1: Index Scan (11.9s, the common plan)

  WindowAgg (actual: 11,121ms..11,126ms, rows=450)
    Buffers: hit=19,013 read=14,027  I/O: 10,984ms
    └─ Sort (quicksort, 14MB memory)
       └─ Append (actual: 1ms..11,096ms, rows=13,365)
          ├─ Index Scan query_explains_7d_20260219 (964ms, 1,039 rows, 7,199 filtered)
          ├─ Index Scan query_explains_7d_20260220 (1,051ms, 1,061 rows, 8,447 filtered)
          ├─ Index Scan query_explains_7d_20260221 (970ms, 906 rows, 5,057 filtered)
          ├─ Index Scan query_explains_7d_20260222 (928ms, 980 rows, 4,206 filtered)
          ├─ Index Scan query_explains_7d_20260223 (2,593ms, 1,867 rows, 7,253 filtered)
          ├─ Index Scan query_explains_7d_20260224 (2,713ms, 3,419 rows, 9,422 filtered)
          ├─ Index Scan query_explains_7d_20260225 (1,797ms, 3,575 rows, 7,590 filtered)
```

```
  Root Cause

  The problem is clear - almost all execution time is spent on disk I/O (10,984ms out of 11,873ms =
92.5% I/O wait):

  1. The index only narrows by (server_id, seen_at), which returns ~50K-65K rows across all 8 daily
partitions
  2. The jsonb_array_length(node_insights) > 0 filter is applied as a post-fetch filter, requiring
PostgreSQL to read all the TOAST data for the
  node_insights JSONB column from disk for every matching row
  3. Only ~20% of rows pass the filter (13,365 kept out of ~64K fetched) - so ~80% of I/O is wasted
  4. Buffer hit ratio is terrible because the JSONB TOAST pages are large and rarely cached

  Recommendations

  1. Add a stored/generated column like has_node_insights boolean GENERATED ALWAYS AS
(jsonb_array_length(node_insights) > 0) STORED and include
  it in the index. This avoids detoasting the JSONB just to check its length.
  2. Alternative: partial index on (server_id, seen_at) with WHERE jsonb_array_length(node_insights) >
0. This would eliminate the 80% of rows
  that are filtered out, at the cost of maintaining the index on writes.
  3. Alternative: expression index including jsonb_array_length(node_insights) as an indexed column,
e.g. CREATE INDEX ON query_explains_7d
  (server_id, seen_at, jsonb_array_length(node_insights)) WHERE jsonb_array_length(node_insights) > 0.

  Option 2 (partial index) is likely the best tradeoff - it would cut I/O by ~80% with minimal write
overhead, since query_explains_7d is
  partitioned daily and old partitions stop receiving writes.
```

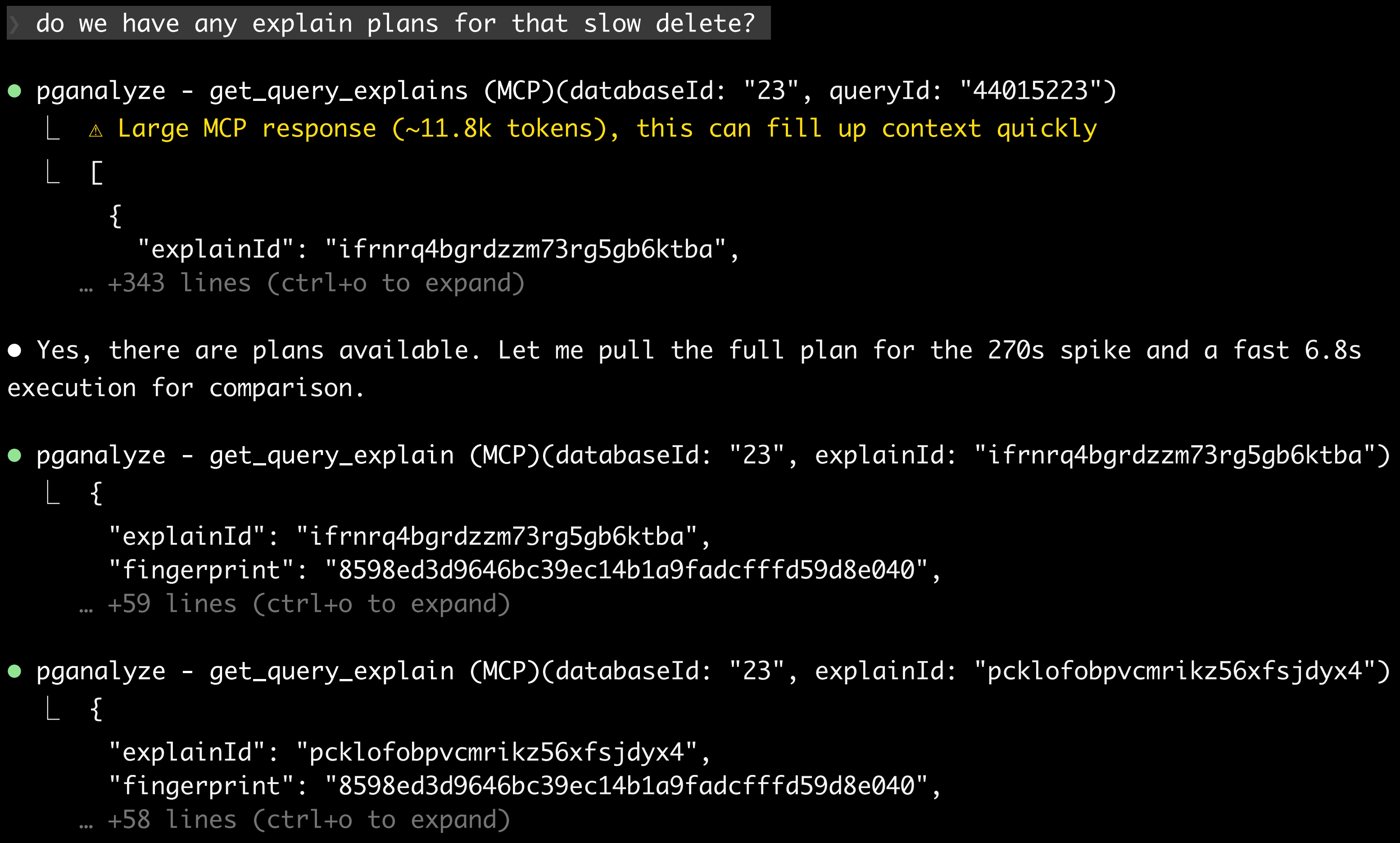


```
❯ let's come back to the snapshot cleanup spikes - do you have any evidence of lock contention?

● Let me check for blocking queries and look at recent backend activity around the cleanup times.

● Found a major spike - 270 seconds (4.5 minutes) for a single DELETE. Let me look at what was
happening on the server at that exact time.

● pganalyze - get_backends (MCP)(databaseId: "23", timestamp: 1772087700)
  ⎿  Running…

● pganalyze - get_backend_counts (MCP)(databaseId: "23", startTs: 1772087400, endTs: 1772088000)

● Good news and bad news. Here's what the evidence shows:

  No Lock Contention

  There is no evidence of lock contention:
  - Zero waitingForLock: true entries across the entire time window
  - No IDLE_IN_TRANSACTION buildup (at most 1 connection, briefly)
  - No blocking query issues flagged by pganalyze

  The Real Problem: I/O Saturation from Concurrent Deletes

  The backend snapshot at 1772087700 (during the 270s spike) caught the smoking gun. Multiple delete
operations were running simultaneously:
```

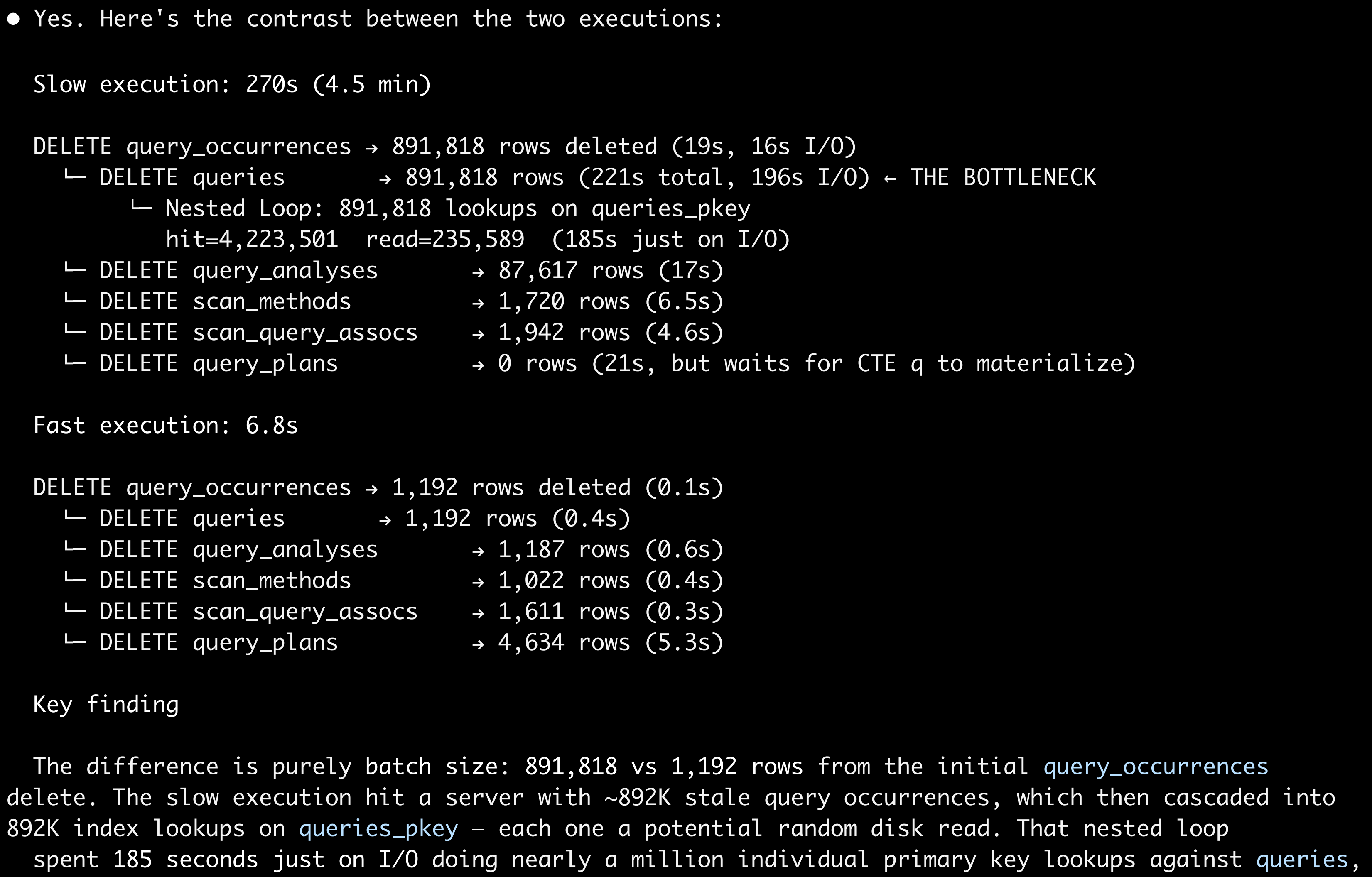
```
┌───────┬───────────┬──────────────┬──────────────────────────────────────────────────┐
│  PID  │    Age    │  Wait Event  │                      Query                       │
├───────┼───────────┼──────────────┼──────────────────────────────────────────────────┤
│ 8151  │ 3 minutes │ DataFileRead │ WITH q AS (DELETE FROM query_occurrences ...) ... │
├───────┼───────────┼──────────────┼──────────────────────────────────────────────────┤
│ 9480  │ 2 minutes │ (running)    │ WITH q AS (DELETE FROM query_occurrences ...) ... │
├───────┼───────────┼──────────────┼──────────────────────────────────────────────────┤
│ 9478  │ <5s       │ (running)    │ DELETE FROM query_stats WHERE ...                │
├───────┼───────────┼──────────────┼──────────────────────────────────────────────────┤
│ 10397 │ <5s       │ DataFileRead │ DELETE FROM vacuum_runs WHERE ...                │
├───────┼───────────┼──────────────┼──────────────────────────────────────────────────┤
│ 10398 │ <5s       │ DataFileRead │ DELETE FROM vacuum_runs WHERE ...                │
└───────┴───────────┴──────────────┴──────────────────────────────────────────────────┘
```

Two instances of the same cascading delete (PIDs 8151 and 9480) were running concurrently for 2-3 minutes, plus additional deletes against `query_stats` and `vacuum_runs`. All of them waiting on `DataFileRead` - they're hammering the disk together.

The 270-second runtime isn't caused by lock waits, it's caused by I/O starvation - multiple large delete jobs competing for disk bandwidth simultaneously. The cascading CTE delete touches 6 tables (`query_occurrences`, `queries`, `query_analyses`, `schema_table_scan_methods`, `schema_table_scan_query_associations`, `query_plans`), generating massive random I/O, and running two of them in parallel doubles the pain.



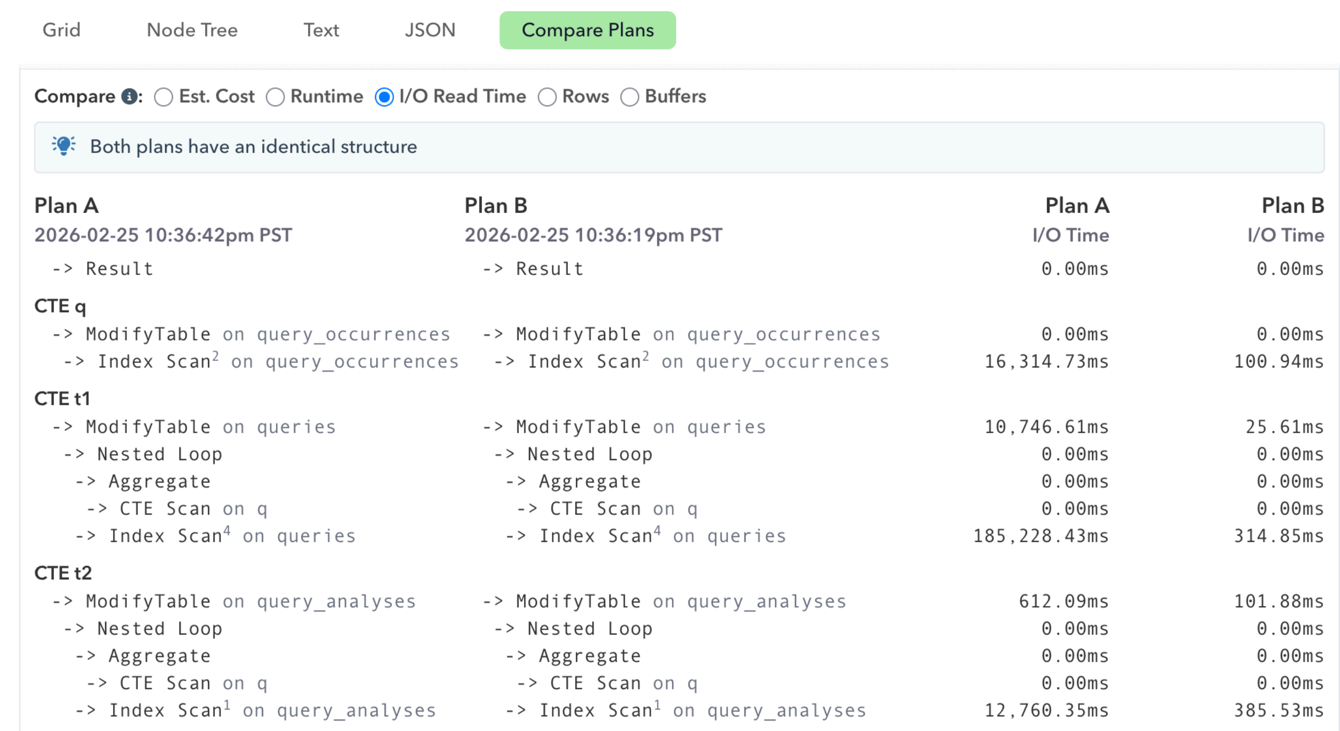
```
> do we have any explain plans for that slow delete?

● pganalyze - get_query_explains (MCP)(databaseId: "23", queryId: "44015223")
  ⎿  ⚠ Large MCP response (~11.8k tokens), this can fill up context quickly
  ⎿  [
       {
         "explainId": "ifrnrq4bgrdzzm73rg5gb6ktba",
    … +343 lines (ctrl+o to expand)

● Yes, there are plans available. Let me pull the full plan for the 270s spike and a fast 6.8s
execution for comparison.

● pganalyze - get_query_explain (MCP)(databaseId: "23", explainId: "ifrnrq4bgrdzzm73rg5gb6ktba")
  ⎿  {
       "explainId": "ifrnrq4bgrdzzm73rg5gb6ktba",
       "fingerprint": "8598ed3d9646bc39ec14b1a9fadcfffd59d8e040",
    … +59 lines (ctrl+o to expand)

● pganalyze - get_query_explain (MCP)(databaseId: "23", explainId: "pcklofobpvcmrikz56xfsjdyx4")
  ⎿  {
       "explainId": "pcklofobpvcmrikz56xfsjdyx4",
       "fingerprint": "8598ed3d9646bc39ec14b1a9fadcfffd59d8e040",
    … +58 lines (ctrl+o to expand)
```



```
● Yes. Here's the contrast between the two executions:

  Slow execution: 270s (4.5 min)

  DELETE query_occurrences → 891,818 rows deleted (19s, 16s I/O)
    └─ DELETE queries       → 891,818 rows (221s total, 196s I/O) ← THE BOTTLENECK
         └─ Nested Loop: 891,818 lookups on queries_pkey
            hit=4,223,501  read=235,589  (185s just on I/O)
    └─ DELETE query_analyses       → 87,617 rows (17s)
    └─ DELETE scan_methods         → 1,720 rows (6.5s)
    └─ DELETE scan_query_assocs    → 1,942 rows (4.6s)
    └─ DELETE query_plans          → 0 rows (21s, but waits for CTE q to materialize)

  Fast execution: 6.8s

  DELETE query_occurrences → 1,192 rows deleted (0.1s)
    └─ DELETE queries       → 1,192 rows (0.4s)
    └─ DELETE query_analyses       → 1,187 rows (0.6s)
    └─ DELETE scan_methods         → 1,022 rows (0.4s)
    └─ DELETE scan_query_assocs    → 1,611 rows (0.3s)
    └─ DELETE query_plans          → 4,634 rows (5.3s)

  Key finding

  The difference is purely batch size: 891,818 vs 1,192 rows from the initial query_occurrences
delete. The slow execution hit a server with ~892K stale query occurrences, which then cascaded into
892K index lookups on queries_pkey — each one a potential random disk read. That nested loop
  spent 185 seconds just on I/O doing nearly a million individual primary key lookups against queries,
```

Links

- Slow (270s): https://staging.pganalyze.com/databases/23/queries/44015223/explains/ifrnrq4bgrdzzm73rg5gb6ktba
- Fast (6.8s): https://staging.pganalyze.com/databases/23/queries/44015223/explains/pcklofobpvcmrikz56xfsjdyx4



Compare: ○ Est. Cost ○ Runtime ● I/O Read Time ○ Rows ○ Buffers

Both plans have an identical structure

Plan A 2026-02-25 10:36:42pm PST	Plan B 2026-02-25 10:36:19pm PST	Plan A I/O Time	Plan B I/O Time
-> Result	-> Result	0.00ms	0.00ms
CTE q			
-> ModifyTable on query_occurrences	-> ModifyTable on query_occurrences	0.00ms	0.00ms
-> Index Scan^2 on query_occurrences	-> Index Scan^2 on query_occurrences	16,314.73ms	100.94ms
CTE t1			
-> ModifyTable on queries	-> ModifyTable on queries	10,746.61ms	25.61ms
-> Nested Loop	-> Nested Loop	0.00ms	0.00ms
-> Aggregate	-> Aggregate	0.00ms	0.00ms
-> CTE Scan on q	-> CTE Scan on q	0.00ms	0.00ms
-> Index Scan^4 on queries	-> Index Scan^4 on queries	185,228.43ms	314.85ms
CTE t2			
-> ModifyTable on query_analyses	-> ModifyTable on query_analyses	612.09ms	101.88ms
-> Nested Loop	-> Nested Loop	0.00ms	0.00ms
-> Aggregate	-> Aggregate	0.00ms	0.00ms
-> CTE Scan on q	-> CTE Scan on q	0.00ms	0.00ms
-> Index Scan^1 on query_analyses	-> Index Scan^1 on query_analyses	12,760.35ms	385.53ms

pganalyze Plan Comparison let's you verify the conclusion

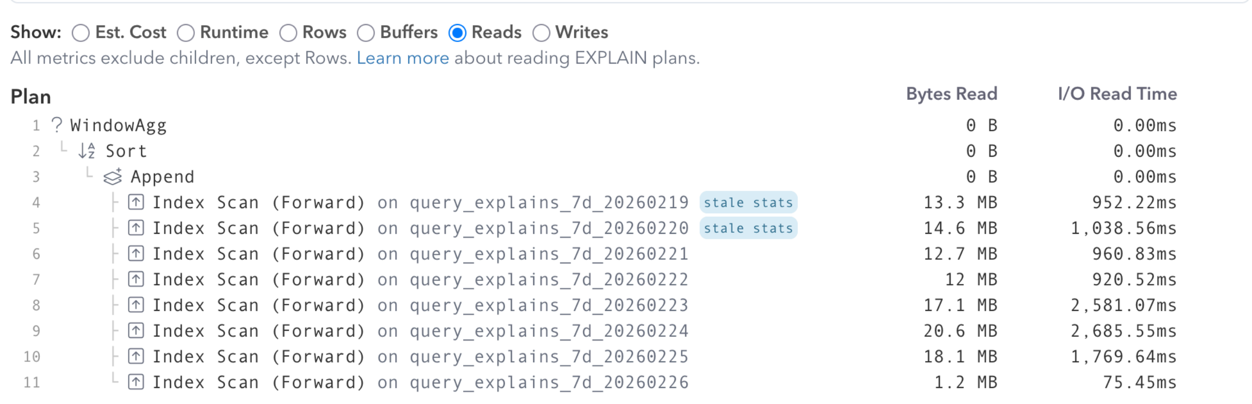
❯ do you have a link for that explain plan we talked about earlier?

- The EXPLAIN plan for the slowest execution (11.8s) of the query_explains_7d ranked scan is:

  https://staging.pganalyze.com/databases/23/queries/43956867/explains/5eq4apblljhtpoqh6atif4nvbu

  And the Bitmap Heap Scan variant (also 11.8s, but with 3x more disk reads):

  https://staging.pganalyze.com/databases/23/queries/43956867/explains/ye6t5fxovbd43e67xty7u6usgm

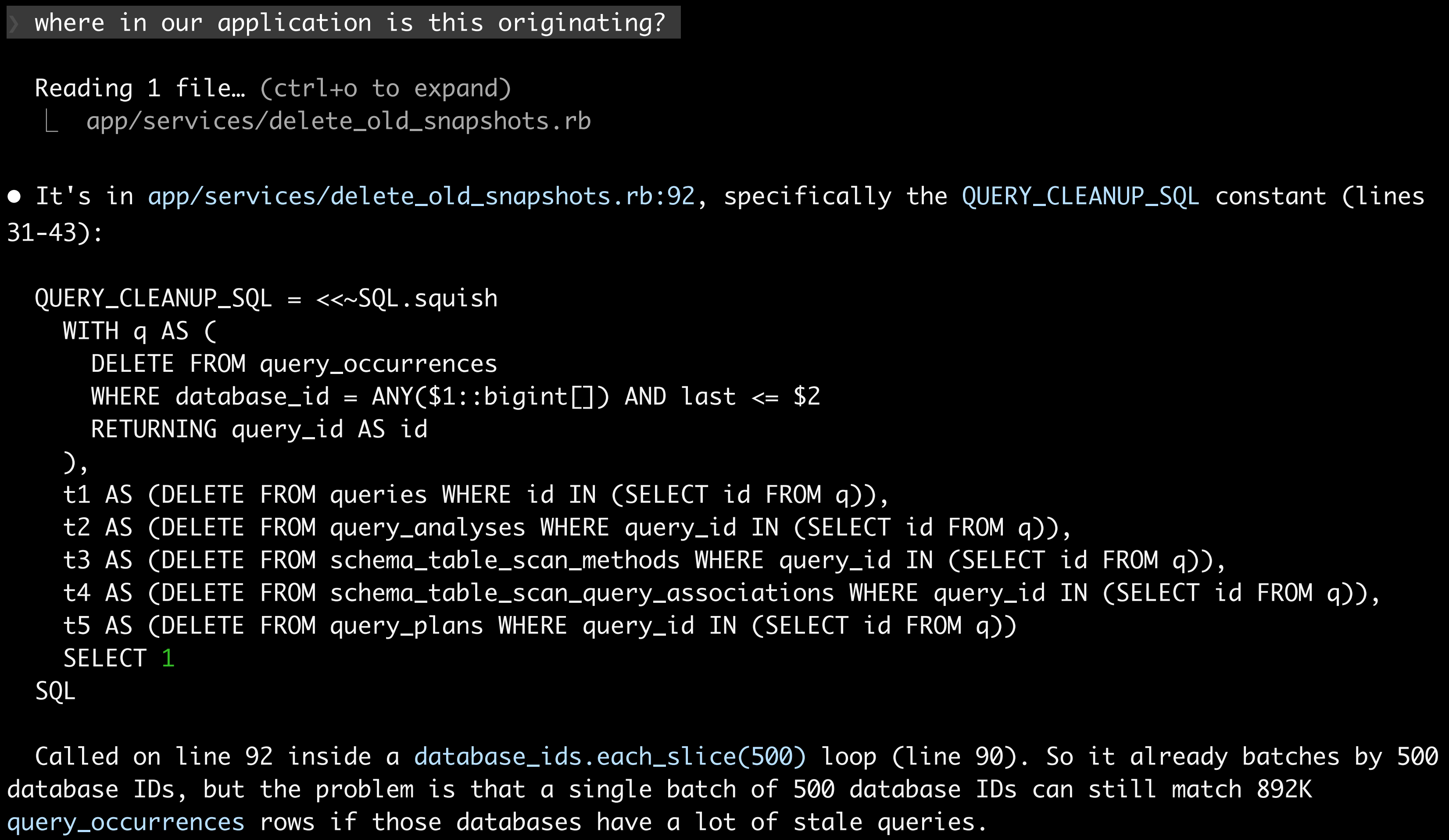


Show: ○ Est. Cost ○ Runtime ○ Rows ○ Buffers ● Reads ○ Writes

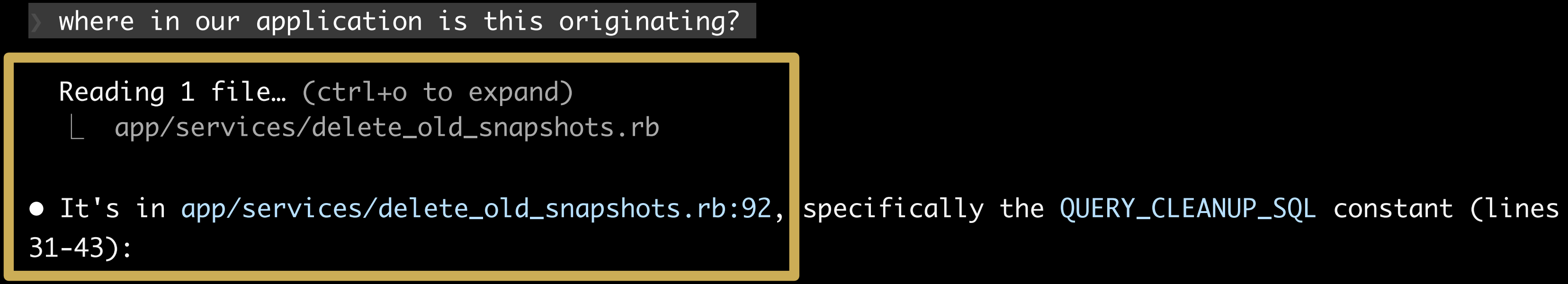
All metrics exclude children, except Rows. Learn more about reading EXPLAIN plans.

	Plan	Bytes Read	I/O Read Time
1	WindowAgg	0 B	0.00ms
2	Sort	0 B	0.00ms
3	Append	0 B	0.00ms
4	Index Scan (Forward) on query_explains_7d_20260219 stale stats	13.3 MB	952.22ms
5	Index Scan (Forward) on query_explains_7d_20260220 stale stats	14.6 MB	1,038.56ms
6	Index Scan (Forward) on query_explains_7d_20260221	12.7 MB	960.83ms
7	Index Scan (Forward) on query_explains_7d_20260222	12 MB	920.52ms
8	Index Scan (Forward) on query_explains_7d_20260223	17.1 MB	2,581.07ms
9	Index Scan (Forward) on query_explains_7d_20260224	20.6 MB	2,685.55ms
10	Index Scan (Forward) on query_explains_7d_20260225	18.1 MB	1,769.64ms
11	Index Scan (Forward) on query_explains_7d_20260226	1.2 MB	75.45ms

Easy backlinks to share with the team as evidence



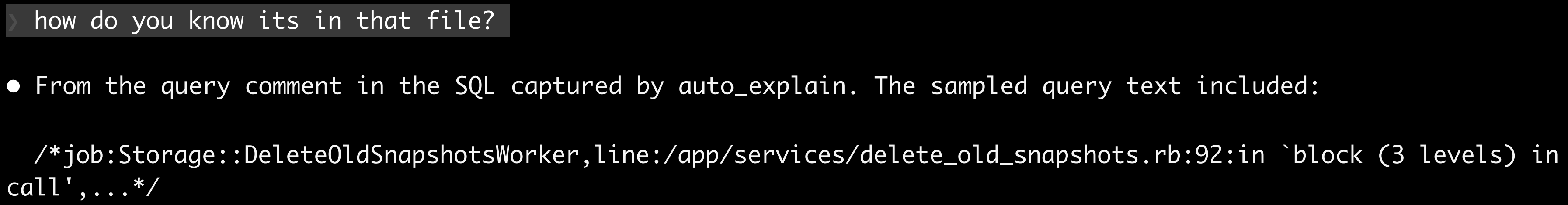
```
> where in our application is this originating?

  Reading 1 file… (ctrl+o to expand)
   ⎿  app/services/delete_old_snapshots.rb

● It's in app/services/delete_old_snapshots.rb:92, specifically the QUERY_CLEANUP_SQL constant (lines
31-43):

  QUERY_CLEANUP_SQL = <<~SQL.squish
    WITH q AS (
      DELETE FROM query_occurrences
      WHERE database_id = ANY($1::bigint[]) AND last <= $2
      RETURNING query_id AS id
    ),
    t1 AS (DELETE FROM queries WHERE id IN (SELECT id FROM q)),
    t2 AS (DELETE FROM query_analyses WHERE query_id IN (SELECT id FROM q)),
    t3 AS (DELETE FROM schema_table_scan_methods WHERE query_id IN (SELECT id FROM q)),
    t4 AS (DELETE FROM schema_table_scan_query_associations WHERE query_id IN (SELECT id FROM q)),
    t5 AS (DELETE FROM query_plans WHERE query_id IN (SELECT id FROM q))
    SELECT 1
  SQL

  Called on line 92 inside a database_ids.each_slice(500) loop (line 90). So it already batches by 500
database IDs, but the problem is that a single batch of 500 database IDs can still match 892K
query_occurrences rows if those databases have a lot of stale queries.
```



```
› where in our application is this originating?

  Reading 1 file… (ctrl+o to expand)
   ⎿  app/services/delete_old_snapshots.rb

● It's in app/services/delete_old_snapshots.rb:92, specifically the QUERY_CLEANUP_SQL constant (lines
31-43):
```

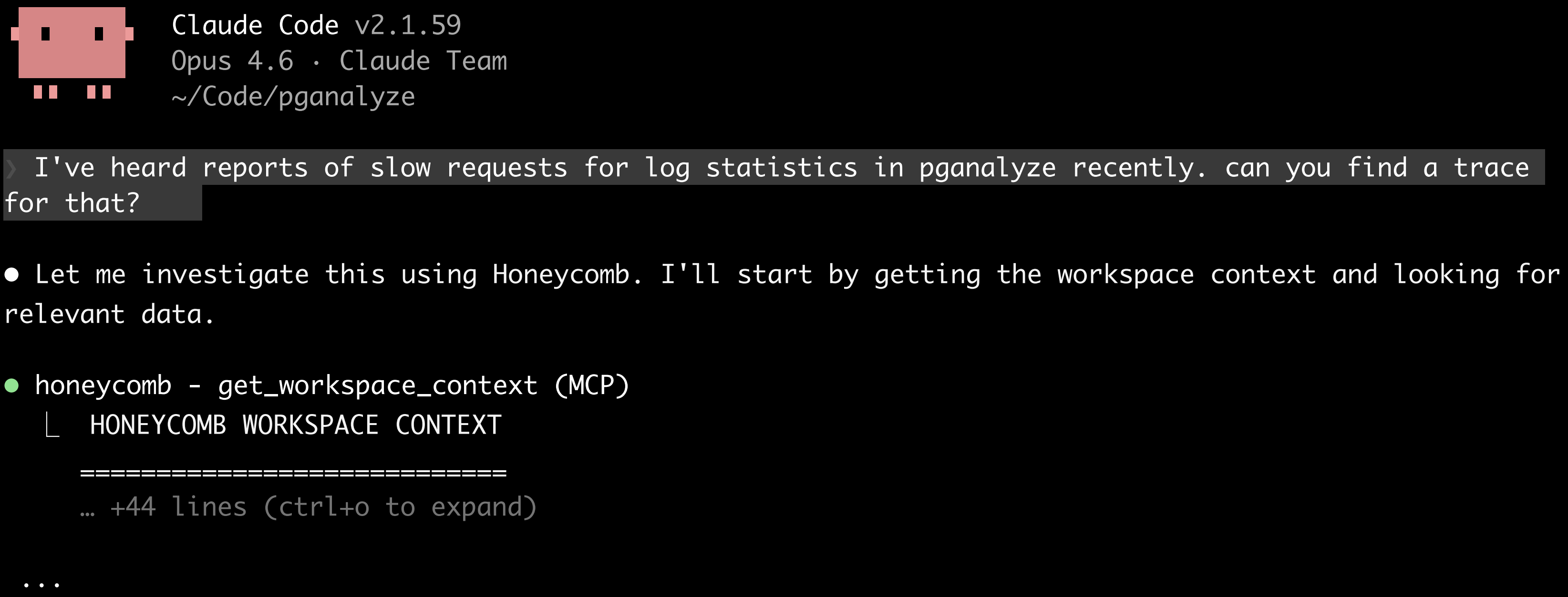
How did Claude know?

Source code location in query tags!
(easy to do in certain frameworks, like **Rails 7.2+**)



```
› how do you know its in that file?

● From the query comment in the SQL captured by auto_explain. The sampled query text included:

  /*job:Storage::DeleteOldSnapshotsWorker,line:/app/services/delete_old_snapshots.rb:92:in `block (3 levels) in
call',...*/
```

[Demo] **Combining** pganalyze MCP Server **with APM traces**



```
Claude Code v2.1.59
Opus 4.6 · Claude Team
~/Code/pganalyze

❯ I've heard reports of slow requests for log statistics in pganalyze recently. can you find a trace
for that?

● Let me investigate this using Honeycomb. I'll start by getting the workspace context and looking for
relevant data.

● honeycomb - get_workspace_context (MCP)
  ⎿  HONEYCOMB WORKSPACE CONTEXT

    ============================
    … +44 lines (ctrl+o to expand)

 ...
```

This demo uses Honeycomb as the APM provider,
but this works with any OpenTelemetry-compatible provider
https://pganalyze.com/docs/opentelemetry

```
● I can see the app is primarily GraphQL-based. Log statistics would be a GraphQL operation. Let me
search for GraphQL operations related to logs.

● honeycomb - run_query (MCP)(environment_slug: "production", dataset_slug: "pganalyze-app",
include_samples: true, query_spec:
                             {"calculations":[{"op":"COUNT"},{"op":"P95","column":"duration_ms"},
{"op":"MAX","column":"duration_ms"}],"breakdowns"
                             :["name"],"filters":
[{"column":"name","op":"contains","value":"Log"}],"orders":[{"op":"MAX","column":"duration_ms","o
                             rder":"descending"}],"time_range":"7d","limit":30})
   ⎿  Error: result (345,300 characters) exceeds maximum allowed tokens. Output has been saved to ...
     Format: JSON array with schema: [{type: string, text: string}]
     … +9 lines (ctrl+o to expand)

● Large result set. Let me extract the key data.

● Bash(…)

● Let me run a cleaner query without samples to get the aggregate data first.

...

● Found it. LogStats is clearly problematic. Now let me grab a specific slow trace for it.
```

```
● Found plenty of slow traces. Let me get the trace details for the slowest one to see what's causing
the latency.

● honeycomb - get_trace (MCP)(environment_slug: "production", trace_id:
"e1225213e10904a869788ac8fb255a6a", time_range: "7d", view_mode: "auto")
  ⎿  | span_id | parent_id | start_unix_ns | duration_ns | kind | status_code | error | http_status |
service | route | name | annotation_type | link_span_id | link_trace_id | depth | child_count |
descendant_count | event_count | is_collapsed |
     | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
     … +40 lines (ctrl+o to expand)

● Here's what I found:

  LogStats slow request summary

  The LogStats GraphQL operation is indeed slow. Over the last 7 days:

  ┌────────────────┬──────────┐
  │     Metric     │  Value   │
  ├────────────────┼──────────┤
  │ Total requests │ 980      │
  ├────────────────┼──────────┤
  │ P95 latency    │ 4,343ms  │
  ├────────────────┼──────────┤
  │ Max latency    │ 25,202ms │
  └────────────────┴──────────┘
```

Example slow trace

```
  I pulled the trace for the worst offender — a 25.1s request that timed out with a 503:

  Trace: https://ui.honeycomb.io/pganalyze/environments/production/trace?
trace_id=e1225213e10904a869788ac8fb255a6a&trace_start_ts=1771490178&trace
  _end_ts=1772094978

  The bottleneck is clear from the trace:

  1. Root span — Api::GraphqlController#graphql (LogStats) — 25,111ms, ended in ERROR (503)
  2. Dataload.select_rows — 25,007ms (99.6% of total time), also errored
  3. Inside that, a single pgaweb DB query — 25,004ms — which was cancelled due to PG::QueryCanceled:
ERROR: canceling statement due to statement timeout

  So essentially a single database query inside the Dataload.select_rows span is consuming nearly the
entire request time, hitting the 25-second statement timeout. The other spans (auth, server lookup,
etc.) are all under 22ms.

  There are also several other recent slow traces in the 5-18 second range if you want to compare. Want
me to dig into what SQL query is being run, or look at the relevant resolver code?
```

```
❯ can we look at the explain plan for the 18 second one?

● honeycomb - get_trace (MCP)(environment_slug: "", trace_id: "", time_range: "7d", view_mode: "full")
...

● I can see the trace has EXPLAIN Plan spans from the Postgres dataset. Let me use the pganalyze MCP
tool to get the explain plan from this trace.

● I need the db.postgresql.plan attribute from the EXPLAIN Plan spans. Let me check the Postgres
dataset for those span details.

● honeycomb - run_query (MCP)(environment_slug: "production", dataset_slug: "postgres--pganalyze-",
...

● I can see the 17,093ms EXPLAIN Plan span. Let me resolve it to the actual explain plan in pganalyze.

● pganalyze - get_query_explain_from_trace (MCP)(serverId: "qlosi5joffcpfppzmz6aufoiga", datname:
"pgaweb", fingerprint: "1012895bd5a59b5b", timestamp: 1772037133)

────────────────────────────────────────────────────────────────────────────────────────
 Tool use

   Resolve a "db.postgresql.plan" trace span attribute (structured as
   "/servers/{serverId}/databases/{datname}/queries/{fingerprint}/samples/{timestamp}?role={role}")
from another MCP server to the EXPLAIN plan in pganalyze that has the specific slow execution details.
```

- Here's the explain plan for the 18-second LogStats query:

  EXPLAIN Plan (17,093ms runtime)

  The query scans 8 weekly-partitioned log_stats_7d tables (Feb 18-25) via an Append node, filtering by database_id and a time range on occurred_at_1min.

  Root cause: I/O bound on cold data

  The total time is almost entirely I/O wait: 16,676ms out of 17,093ms (97.6%) is spent on disk reads.

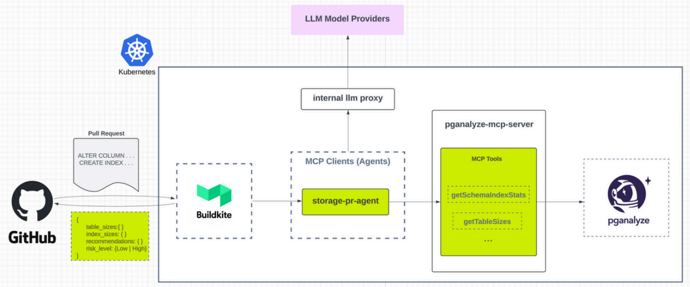
```
┌───────────────────────┬───────┬───────────────┬────────────────────┬───────────────────┐
│       Partition       │ Rows  │ I/O Read Time │ Shared Blocks Read │ Shared Blocks Hit │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260218 │ 2,628 │ 3,198ms       │ 6,946              │ 0                 │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260219 │ 7,307 │ 2,846ms       │ 6,808              │ 0                 │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260220 │ 7,305 │ 2,867ms       │ 6,832              │ 0                 │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260221 │ 6,882 │ 2,678ms       │ 6,378              │ 0                 │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260222 │ 7,118 │ 2,544ms       │ 6,541              │ 0                 │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260223 │ 6,865 │ 2,538ms       │ 6,426              │ 34 hit            │
├───────────────────────┼───────┼───────────────┼────────────────────┼───────────────────┤
│ log_stats_7d_20260224 │ 8,256 │ 4ms           │ 9                  │ 7,475 hit         │
```

What about Agents?

Robinhood's Storage PR Agent

Storage PR Agent

The Storage PR Agent is a multi-turn agent that performs automated reviews of schema migration Github Pull Requests. This surfaces an HTTP API which is called via our Buildkite CI pipeline, and serves as the first line of defense against dangerous schema migration changes.



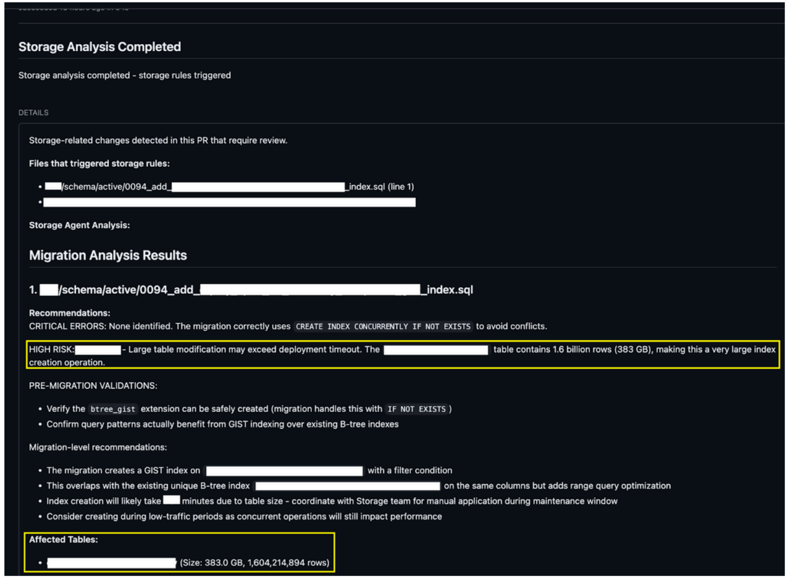
Example Schema Migration:

Consider the following migration where we add a new index on a large table:

```
10 [redacted]schema/active/0094_add_[redacted]_index.sql
@@ -0,0 +1,10 @@
+ -- +migrate Up notransaction
+ CREATE EXTENSION IF NOT EXISTS btree_gist;
+
+ CREATE INDEX CONCURRENTLY IF NOT EXISTS [redacted]
+ ON [redacted]
+ USING gist ([redacted])
+ WHERE [redacted];
```

The analysis posted back to the GitHub PR explicitly calls out the size of the table, and the risk of timing out application deployments if this index creation is run as part of it.

Robinhood's Storage PR Agent



Storage Analysis Completed

Storage analysis completed - storage rules triggered

DETAILS

Storage-related changes detected in this PR that require review.

Files that triggered storage rules:

- [redacted]/schema/active/0094_add_[redacted]_index.sql (line 1)
- [redacted]

Storage Agent Analysis:

Migration Analysis Results

1. [redacted]/schema/active/0094_add_[redacted]_index.sql

Recommendations:
CRITICAL ERRORS: None identified. The migration correctly uses `CREATE INDEX CONCURRENTLY IF NOT EXISTS` to avoid conflicts.

HIGH RISK:[redacted] - Large table modification may exceed deployment timeout. The [redacted] table contains 1.6 billion rows (383 GB), making this a very large index creation operation.

PRE-MIGRATION VALIDATIONS:

- Verify the `btree_gist` extension can be safely created (migration handles this with `IF NOT EXISTS`)
- Confirm query patterns actually benefit from GIST indexing over existing B-tree indexes

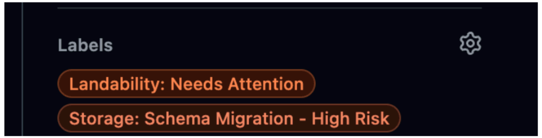
Migration-level recommendations:

- The migration creates a GIST index on [redacted] with a filter condition
- This overlaps with the existing unique B-tree index [redacted] on the same columns but adds range query optimization
- Index creation will likely take [redacted] minutes due to table size - coordinate with Storage team for manual application during maintenance window
- Consider creating during low-traffic periods as concurrent operations will still impact performance

Affected Tables:

- [redacted] (Size: 383.0 GB, 1,604,214,894 rows)

Moreover, the PR is explicitly labelled as “**High Risk**” for clarity.

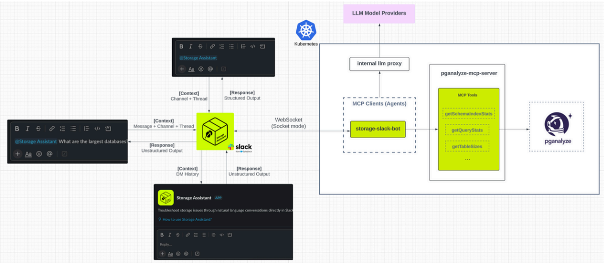


The above example showcases a schema migration defined as SQL, but we also support a similar workflow for Django (python-based) schema migration definitions.

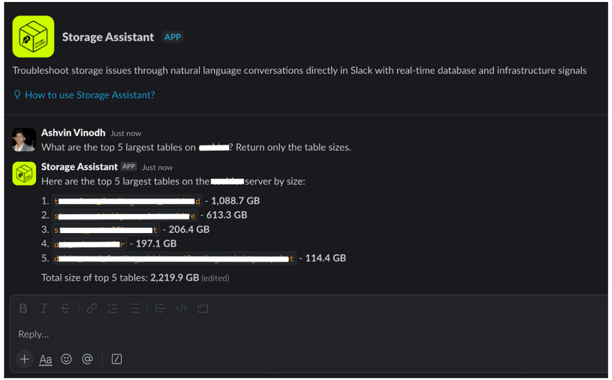
Robinhood's Slack Storage Assistant

Slack Storage Assistant

The Slack assistant offers engineers a conversational interface for querying over database metadata and insights, without needing access to specialized dashboards or knowledge of MCP tooling.



We wanted to make this Assistant available via various surfaces, including DMs, threads and channels. For this purpose, we built a custom MCP Client which integrates with the Slack API to aggregate the relevant message context needed, such as thread, channel and DM history.



Storage Assistant APP

Troubleshoot storage issues through natural language conversations directly in Slack with real-time database and infrastructure signals

How to use Storage Assistant?

Ashvin Vinodh Just now

What are the top 5 largest tables on [redacted]? Return only the table sizes.

Storage Assistant APP Just now

Here are the top 5 largest tables on the [redacted] server by size:

1. [redacted] - 1,088.7 GB
2. [redacted] - 613.3 GB
3. [redacted] - 206.4 GB
4. [redacted] - 197.1 GB
5. [redacted] - 114.4 GB

Total size of top 5 tables: **2,219.9 GB** (edited)

Robinhood Lessons Learned

"Impact
Since adopting this ecosystem, we've seen dramatic improvements across our workflows:

Developer Velocity: Reduced operational burden on the Storage team by enabling seamless database performance analysis and automating complex queries such as cross-shard schema mismatch detection

Schema review coverage: Expanded from a handful of critical services to automated coverage across all (hundreds) of services

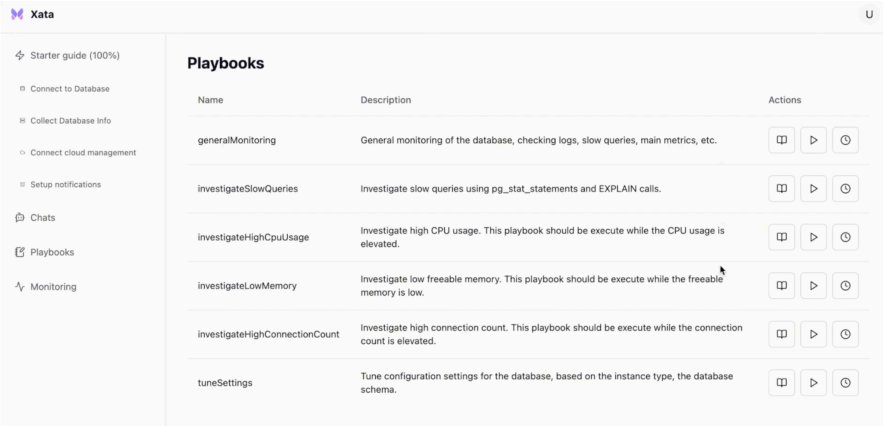
PR review latency: Decreased from 10+ minutes to <2 minutes for most cases"

The Xata Agent

https://github.com/xataio/agent

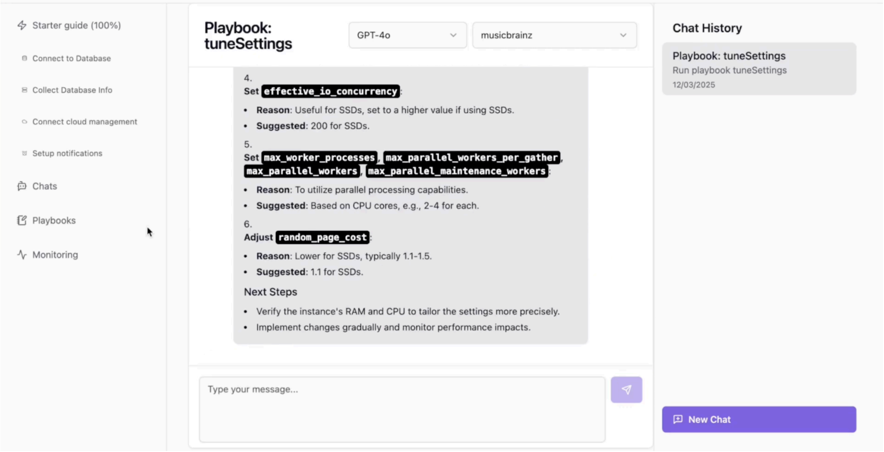
Implements investigations using LLM-based playbooks.

Open-source agent, somewhat active, but haven't seen it used much in production.



Playbooks

Name	Description	Actions
generalMonitoring	General monitoring of the database, checking logs, slow queries, main metrics, etc.	
investigateSlowQueries	Investigate slow queries using pg_stat_statements and EXPLAIN calls.	
investigateHighCpuUsage	Investigate high CPU usage. This playbook should be execute while the CPU usage is elevated.	
investigateLowMemory	Investigate low freeable memory. This playbook should be execute while the freeable memory is low.	
investigateHighConnectionCount	Investigate high connection count. This playbook should be execute while the connection count is elevated.	
tuneSettings	Tune configuration settings for the database, based on the instance type, the database schema.	



Playbook: tuneSettings

4. Set `effective_io_concurrency`:
   - **Reason**: Useful for SSDs, set to a higher value if using SSDs.
   - **Suggested**: 200 for SSDs.
5. Set `max_worker_processes`, `max_parallel_workers_per_gather`, `max_parallel_workers`, `max_parallel_maintenance_workers`:
   - **Reason**: To utilize parallel processing capabilities.
   - **Suggested**: Based on CPU cores, e.g., 2-4 for each.
6. Adjust `random_page_cost`:
   - **Reason**: Lower for SSDs, typically 1.1-1.5.
   - **Suggested**: 1.1 for SSDs.

Next Steps

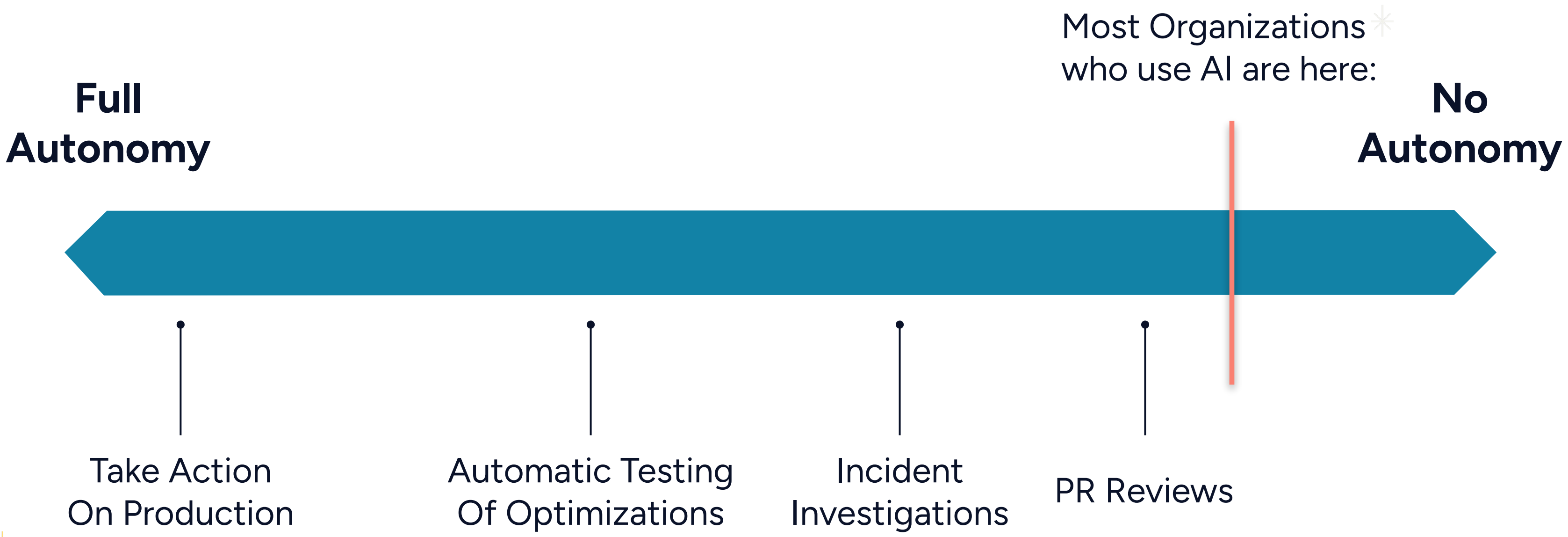
- Verify the instance's RAM and CPU to tailor the settings more precisely.
- Implement changes gradually and monitor performance impacts.

Chat History

Playbook: tuneSettings
Run playbook tuneSettings
12/03/2025

Can we do better?

The Agent Autonomy Problem



Most Organizations who use AI are here:

Full Autonomy

No Autonomy

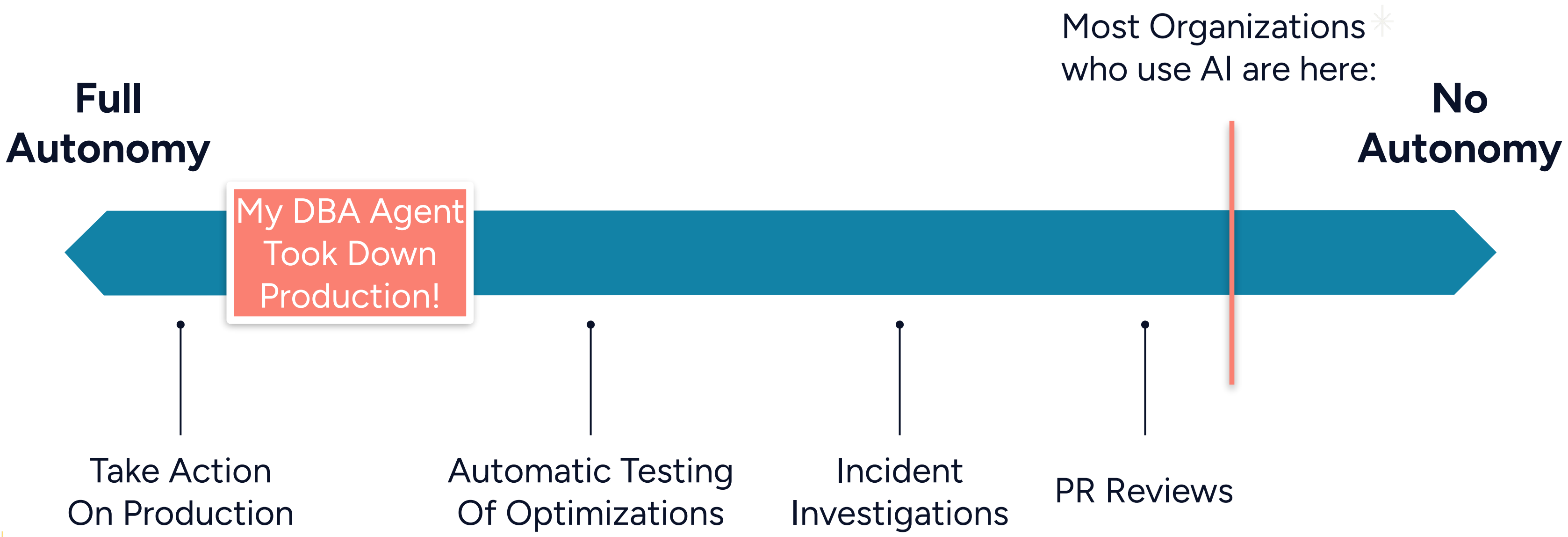
Take Action On Production

Automatic Testing Of Optimizations

Incident Investigations

PR Reviews

The Agent Autonomy Problem



Full Autonomy

No Autonomy

Most Organizations who use AI are here:

My DBA Agent Took Down Production!

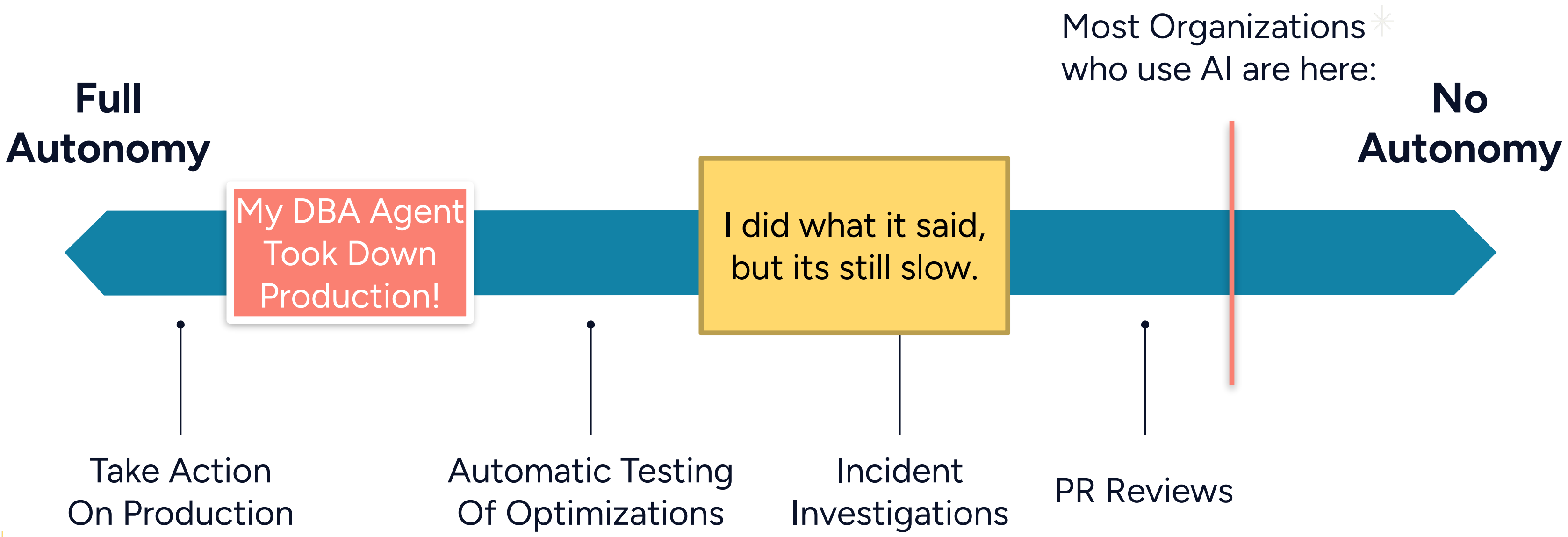
Take Action On Production

Automatic Testing Of Optimizations

Incident Investigations

PR Reviews

The Agent Autonomy Problem



"Harness Engineering"

"Any time you see AI do a bad thing, try to provide tooling that it could have called out to to prevent or course-corrected that bad thing."

[Mitchell Hashimoto's new way of writing code]

High quality tools are needed to make AI-using engineers successful with Postgres production problems.

Agent backpressure

Automated feedback mechanisms — type systems, test suites, linters, and pre-commit hooks — that allow AI agents to detect and correct their own mistakes without human intervention.
latentpatterns.com

What if:

- We let an agent do EXPLAIN ANALYZE with restrictions (timeouts, recurrence, read-only)

- We let an agent test things on a database copy/clone (e.g. to verify indexing outcomes)

- We tracked what was suggested, acted on, & provide that as memory back to the agent

Agents dealing with production data **should be designed and set up by platform and engineering teams**.

Vendors selling **"one size fits all" agents** is good for stock prices, but that is likely too rigid for most organizations, and unnecessarily locks into one vendor's ecosystem.

Our vision for pganalyze

pganalyze

pganalyze

Postgres-specific monitoring, analysis and recommendation logic.

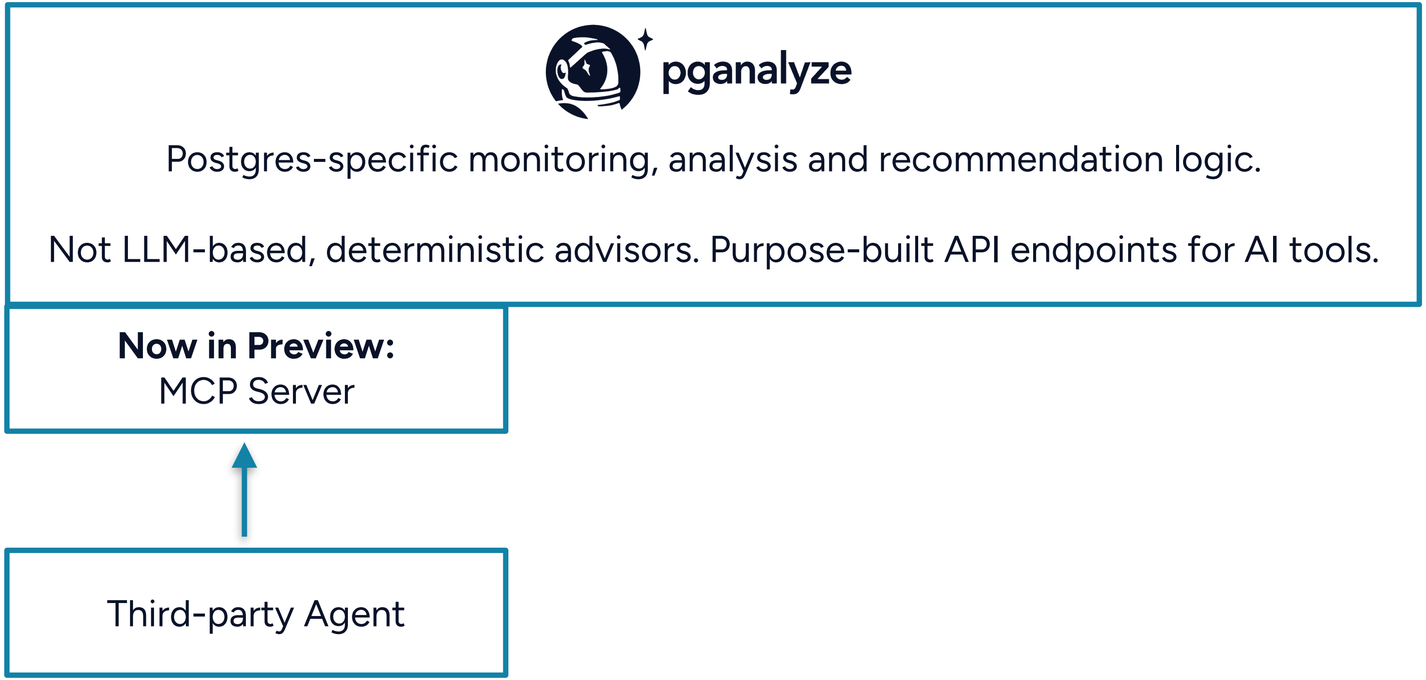
Not LLM-based, deterministic advisors. Purpose-built API endpoints for AI tools.

pganalyze

Postgres-specific monitoring, analysis and recommendation logic.

Not LLM-based, deterministic advisors. Purpose-built API endpoints for AI tools.

Now in Preview:
MCP Server



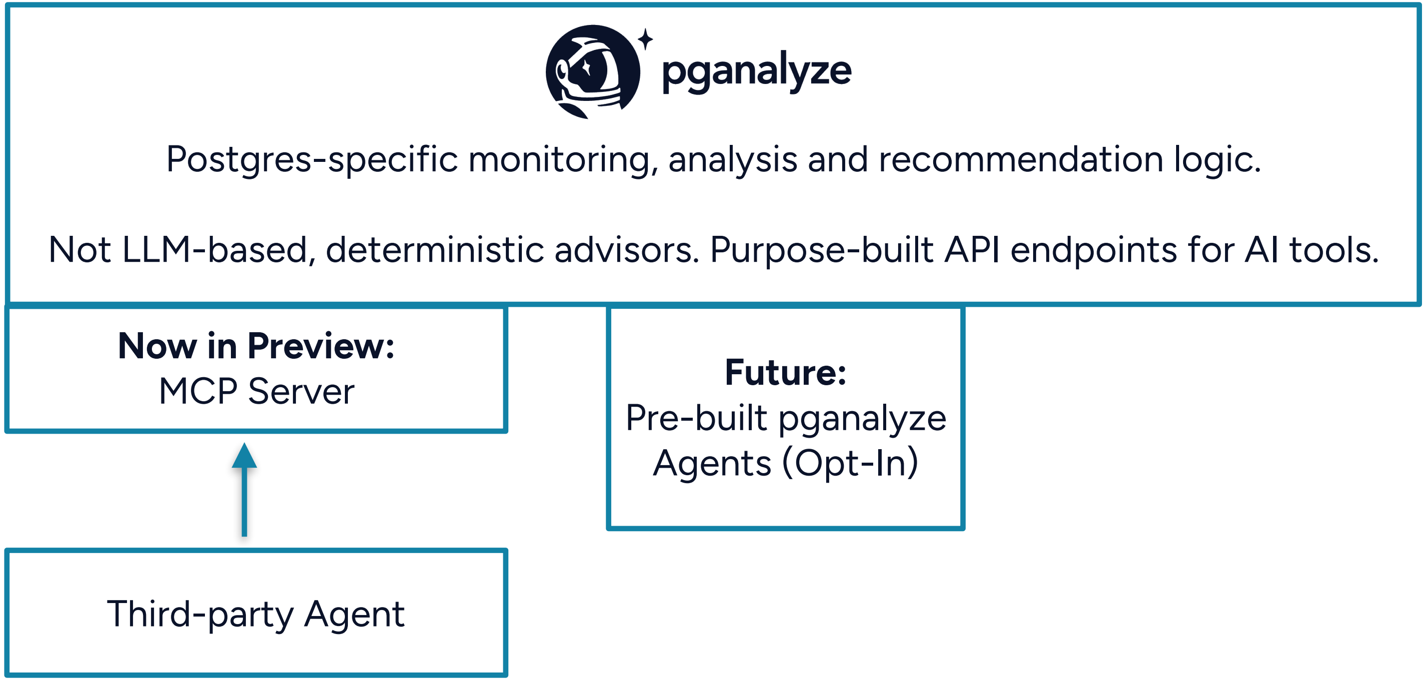
pganalyze

Postgres-specific monitoring, analysis and recommendation logic.

Not LLM-based, deterministic advisors. Purpose-built API endpoints for AI tools.

Now in Preview:
MCP Server

Third-party Agent



pganalyze

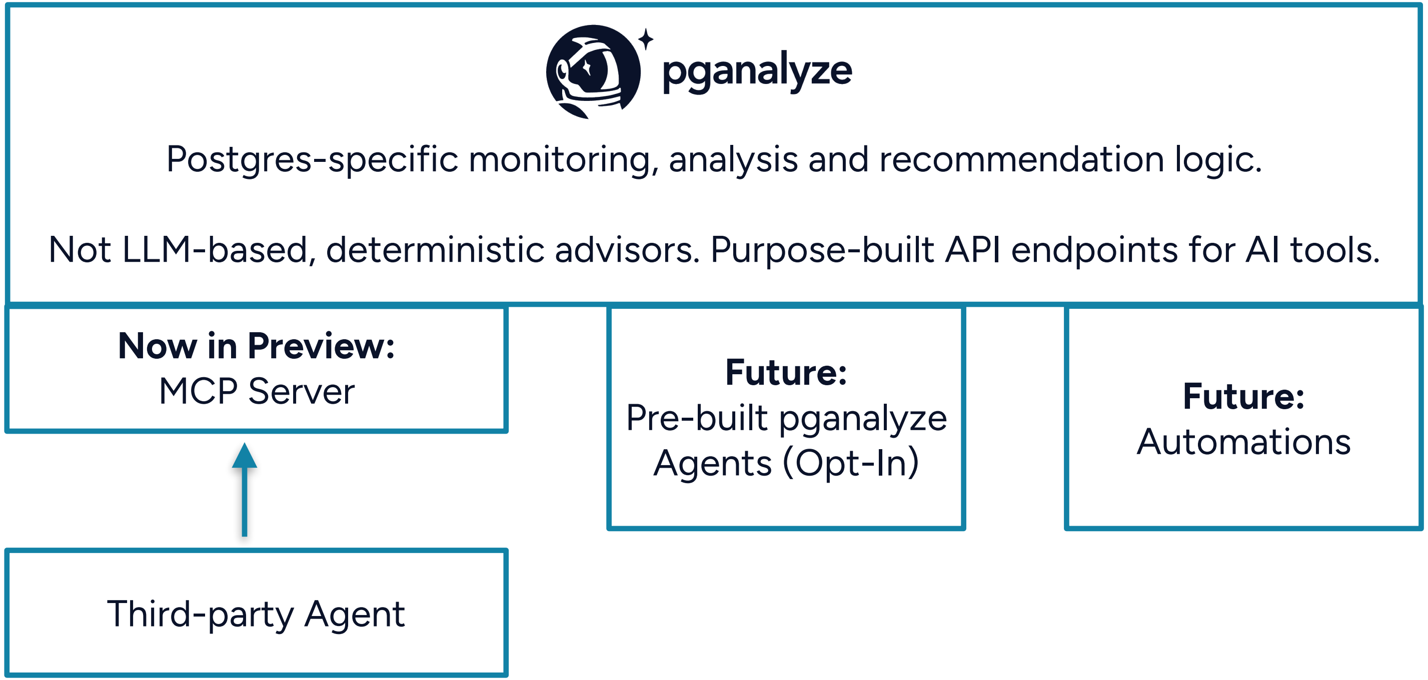
Postgres-specific monitoring, analysis and recommendation logic.

Not LLM-based, deterministic advisors. Purpose-built API endpoints for AI tools.

Now in Preview:
MCP Server

Future:
Pre-built pganalyze
Agents (Opt-In)

Third-party Agent



pganalyze

Postgres-specific monitoring, analysis and recommendation logic.

Not LLM-based, deterministic advisors. Purpose-built API endpoints for AI tools.

Now in Preview:
MCP Server

Future:
Pre-built pganalyze Agents (Opt-In)

Future:
Automations

Third-party Agent

It's not all about AI

We're investing into the key challenges that make Postgres hard at scale

Plan Statistics (Regression Detection)

Plan Management (Prevent Bad Plans)

Better Postgres-specific tools, with more customization options

Large-scale fleet management

The best purpose-built visualizations and analysis workflows

pganalyze MCP Server is now early access

Fill out the post-webinar survey to learn details!

pganalyze

Thank you!

Learn more at pganalyze.com

Contact us: lukas@pganalyze.com

Additional Q&A

pganalyze

Where do you see Postgres observability going, in the community? (Roberto M.)

Observability is a big topic (even if you narrow it to Postgres), lets look at three things in core Postgres. All of these could benefit both AI tool users, and those that prefer to work directly without AI in the picture:

1) **Better query monitoring**: pg_stat_statements is the solution today, and it got a lot better in Postgres 18 (IN list handling, etc) -- but it still has issues for certain workloads. There are efforts in the community to manage the query text file better (e.g. moving it to shared memory stats), and we intend to propose improvements for the Postgres 20 cycle to incorporate percentiles into pg_stat_statements, which is one of the gaps today (since its just avg/min/max).

2) **Plan monitoring and management**: From our side, we've been pushing for more statistics on which query plans get used (pg_stat_plans), for capturing plan regressions. Our goal is to get this more widely adopted by cloud providers this year. Some of the work currently in discussion for Postgres 19 (pg_plan_advice) could then be used to influence/fix bad query plans, rooted in that data.

3) **Improved I/O monitoring**: With PG 16 we got pg_stat_io, but we're still missing some ways to drill down better, especially with asynchronous I/O. There are no patches proposed yet, but I'd expect that we'll see more activity here in the Postgres 20 cycle.

Additional Q&A

Is it possible to define my own custom queries/data collection queries to supply to the "workload repository", i.e. pganalyze, that the mcp server could then have access to as a tool call? (Nick B.)

Not yet! That said, we're considering allowing the definition of predefined diagnostic queries (that can feed into the internal pganalyze database), that could then be accessed by the MCP server.

More of a suggestion, but would love to have automation integrated with alerting that highlights newly introduced slow queries and link that to code changes. Seems like LLMs would be particularly good at that kind of code analysis linking to the info pganalyze has. (Benjamin F.)

Agreed - in our own testing we've found the code analysis aspect to be very useful, especially when queries are annotated with source code locations. Right now we're focused on enabling workflows that utilize existing tooling already in use (Claude Code, Copilot, etc), where you could pull in the MCP server information for slow queries. Later we are also considering offering a built-in experience in pganalyze (opt-in) which could provide that code analysis out of the box together with alerts.

Additional Q&A

Any observations you could share on the cost implications of these solution? (Arunkumar M.)

During early access the preview is free of charge. We haven't finalized pricing for the MCP server when going out of preview, but broadly speaking, intend to offer the core functionality for all current Scale plan customers (i.e. not on legacy plans) without an upcharge. Note that any LLM token use would go through your existing tooling and subscriptions.

We are however likely to rate-limit, and charge extra for agentic use cases that perform significant amount of tool calls to the pganalyze MCP server.

If I'm building some chat-analytic app should I use the more dynamic MCP Server (if yes which one's the best?) or should I just create a tools for every analytic table I have myself? (Galih P.)

Assuming you're referencing which Postgres MCP server to use during development, it shouldn't matter too much when you're working on your own machine. For production use I would not have your LLM call any Postgres MCP server directly, but rather rely on purpose-built APIs and tool calls.

Additional Q&A

Datadog MCP server does offer Database Monitoring tool as well. What makes pganalyze MCP server better than their DBM tooling? (Mehmet S.)

Datadog's MCP server appears to offer two tools today for getting samples from DBM, but appears to be currently missing abilities to get other information (e.g. statistics, wait events, etc).

More broadly speaking, pganalyze offers a more comprehensive solution for Postgres monitoring, and can perform advanced analysis, including index recommendations across multiple queries, VACUUM advisor recommendations, and per-table statistics and analysis.

We also include log monitoring in our product (charged extra by Datadog), and utilize log information across the product to enhance details like query plans, and other functionality.

If you're interested to discuss in more details, please reach out and we'd be happy to walk through a comparison.

Additional Q&A

How could you make pganalyze work with pgbadger? could be a good idea? (Hugo A.)

pganalyze already includes the built-in Log Insights feature, which functions similar to pgbadger. Different from pgbadger (which analyzes a provided log file) it enables continuous log monitoring, so you can see Postgres log events as they come in, including integrations with major cloud provider APIs.

We get missing index alerts without sufficient information available in the pganalyze API to to create automated PRs (for human review of course), it sounds like this might be solved by the MCP product? (Benjamin F.)

We've actually added a dedicated API to make getting index recommendations easier for the MCP server, which is also available through the GraphQL API (new "summary" field in getSchemaTableIndexSelection), so you could use that if you prefer, or go through the MCP server - your choice. Feel free to reach out for details!