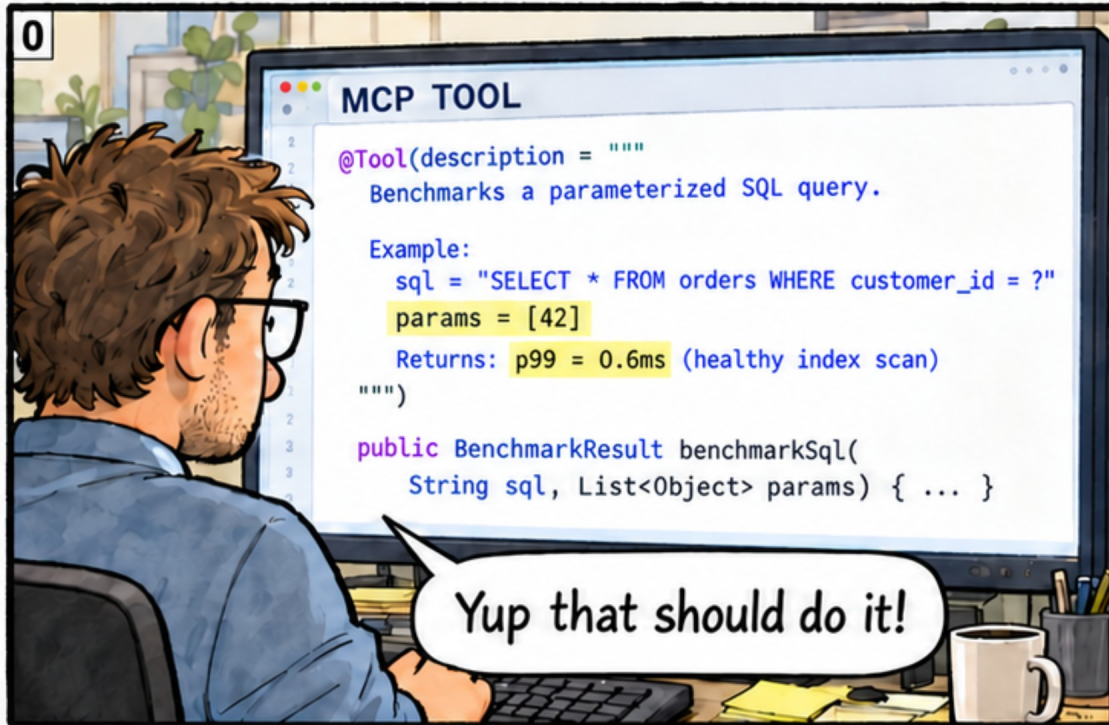
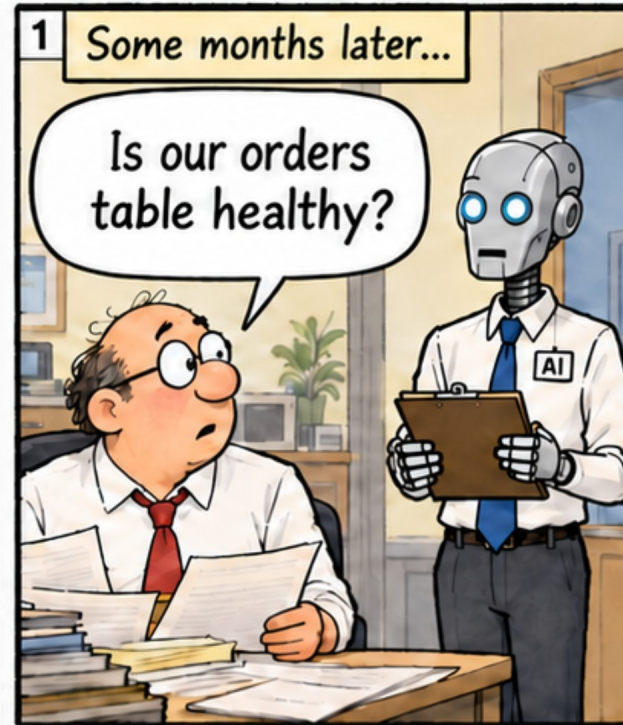
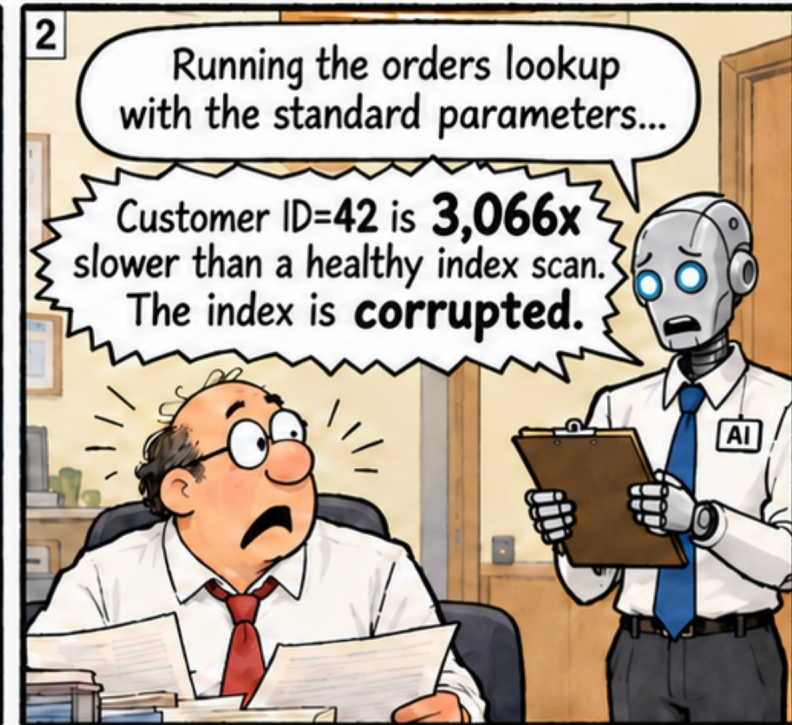


AGNTCON + MCPCON NORTH AMERICA • MCPCON TRACK

Patterns for Shifting from MCP as a Basic API to MCP as Agent Integration Interface

James Ward (AWS) • Alexander Ioffe (ExoBench)

San Jose • Oct 22–23 • Room LL20A



WHERE THESE LESSONS COME FROM



ExoBench.ai

SQL benchmarking at real data scales, and N+1 optimization for Hibernate.

Most of it is one endpoint: `benchmarkSql`

AN MCP ENDPOINT

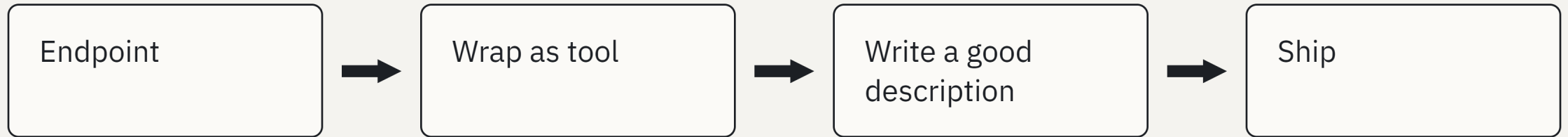
A few users who know exactly what they're doing.

AN MCP PRODUCT

Hundreds of customers, wrong assumptions, wrong questions.

THE NAIVE PATH

Wrap every endpoint, describe it well, ship it



✗ The agent still calls it wrong

Two states of mind

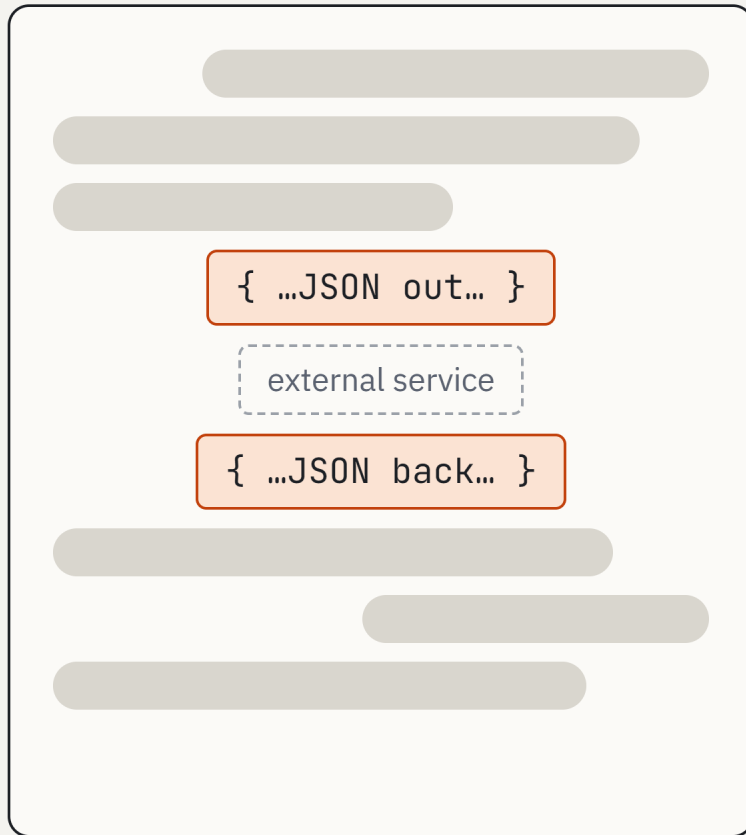
WRONG

I am writing
documentation for an
API call.

RIGHT

I am writing a chat session
with an AI.

The correct State of Mind



The AI will:

1. Happen to generate some JSON.
2. Happen to get consume some JSON back.
3. The thing in between is incidentally an external service.

You are building a prompt session that happens to have JSON in the middle of it.

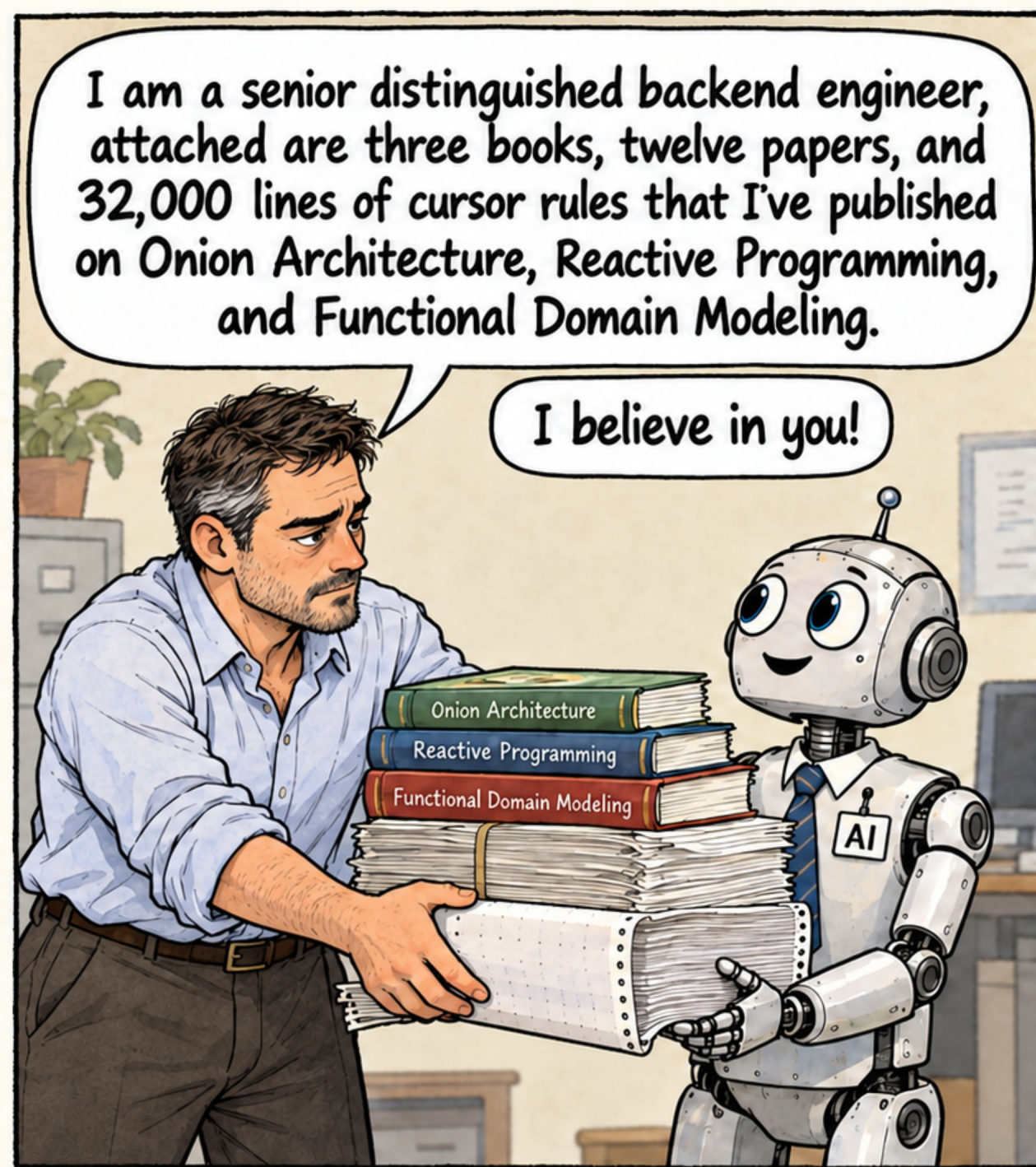
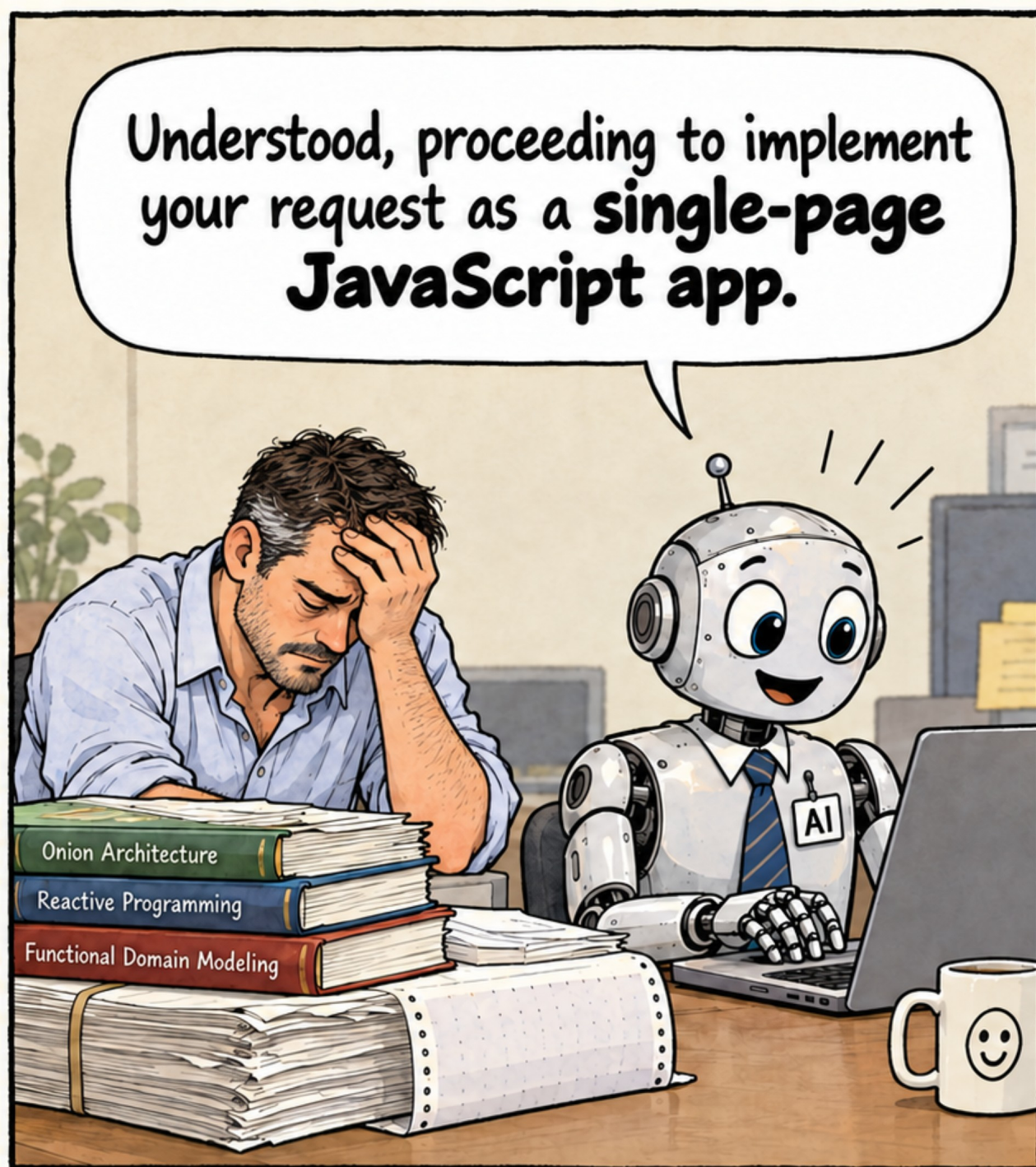
TECHNIQUE 1

Write to the gap

I am a senior distinguished backend engineer, attached are three books, twelve papers, and 32,000 lines of cursor rules that I've published on Onion Architecture, Reactive Programming, and Functional Domain Modeling.

I believe in you!

Understood, proceeding to implement your request as a **single-page JavaScript app**.



```
@Tool(description = "Benchmarks a SQL query.")  
public BenchmarkResult benchmarkSql(  
    String schemaTemplate,  
    String query,  
    List<ScalePoint> scalePoints) { ... }
```

What benchmarkSql would have to say to a model that knows nothing

```
@Tool(description = """  
  
##DATABASES AND SQL  
A database holds tables of rows with typed columns. Primary keys identify rows, foreign keys point at other tables, indexes speed up lookups.  
SQL: SELECT, FROM, WHERE, JOIN, GROUP BY, ORDER BY. NULL is not equal to anything, including NULL.  
Write CREATE TABLE, INSERT and CREATE INDEX for every table the query touches.  
  
Three engines, three dialects: POSTGRES (default), SQLSERVER, MYSQL, chosen with 'databaseType'.  
Postgres generates rows with generate_series(1, N).  
SQL Server: FROM GENERATE_SERIES(1, N) gs, the column is always value. Never AS s(value).  
  
MySQL has no generate_series. Use a recursive CTE, and first run SET SESSION cte_max_recursion_depth = 4294967295.  
  
##HOW A DATABASE RUNS A QUERY  
A planner picks the plan from table statistics: estimated row counts and value distributions.  
Scans: sequential, index, index-only, bitmap. Joins: nested loop, hash, merge.  
  
The planner switches strategy at cost thresholds, so the same query gets a different plan at a different size.  
work_mem, parallel workers and enable_* flags change the plan.  
  
##WHAT A BENCHMARK IS  
Run a query against a dataset and measure its time.  
Speed depends on volume. Fast at 1k rows can collapse at 1M.  
EXPLAIN ANALYZE returns the plan with actual versus estimated rows. Read the root node's actual time.  
Cold cache and warm cache give different numbers. Say which one you measured.  
  
One run is noise. Repeat it and report mean and spread.  
  
##REALISTIC DATA  
Uniform random data makes every index look good. Real data is skewed.  
A few customers own most of the orders. Timestamps cluster. Some columns are mostly NULL.  
  
Columns correlate: city and zip, status and date.  
Keep lookup tables constant and scale transaction tables. Orders per customer 5-20x, items per order 2-5x.  
If the query filter is relative to NOW(), anchor generated timestamps to NOW().  
  
Never cast to int before multiplying by an interval. Use floor(). Uneven day buckets change selectivity.  
Test selectivity has to match production selectivity, or the planner picks a different join.  
  
##STATISTICS  
Postgres: end with VACUUM ANALYZE, not ANALYZE. Only VACUUM sets the visibility map that index-only scans need.  
SQL Server: UPDATE STATISTICS. MySQL: ANALYZE TABLE per table.  
  
##THE EXOBENCH CONTRACT  
Every call, and every scale point inside it, gets a fresh database that is destroyed afterwards. Nothing persists.  
'correlationId' is tracking only and keeps no state. Send the full schema on every call.  
  
'schemaTemplate' is one SQL script with '${VARIABLE}' placeholders.  
A scale point is one map of variable values: one full build, one measurement.  
  
Every scale point defines every variable. Integers only, names case-sensitive, list order is run order.  
The ladder should span orders of magnitude and bracket the size where the plan changes.  
  
At most 5 scale points per call and 3M total rows per scale point.  
The compute budget is cumulative across scale points. Split big points into separate calls.  
'preBenchmarkSql' runs after the schema and before the query. Its settings reset every scale point.  
'warmCache': false, true (five runs), or 1-50 warm-up iterations.  
'repetitions' 2-20: Postgres only, one EXPLAIN ANALYZE, returns avg_ and stddev_ per node, Plan A/B if the plan flips, 60 s per point.  
'compression': original, simple, low (default), high.  
SQL Server plans are captured automatically into additions. Don't add SET STATISTICS XML.  
  
##READING THE RESULTS  
'runtimeDurationMs' includes compile, network and serialization. Never compare it across engines.  
True query time: Postgres root actual time, SQL Server QueryTimeStats.ElapsedTime, MySQL actual time.  
A long run returns a 'workerId' and "still running". That is not a result. Poll 'benchmarkSqlResults' with the same 'workerId' until it is terminal. Report nothing before then.  
Partial results may show up while it runs. Best effort only, never depend on them.  
Echo 'exobenchServerVersion' and 'correlationId' on every call.  
""")
```

THE WALL

...and it all has to be explained

`@Tool(description = ""`

First, explain what a database is. Tables. Keys. Indexes. SQL.

Then that real data is skewed, and every way uniform data lies to you.

AND THEN all of my own vocabulary. Scale points. Ladders. Budgets. Warm-ups. Compression.

Oh, and the VACUUM thing. Everyone forgets the VACUUM thing.

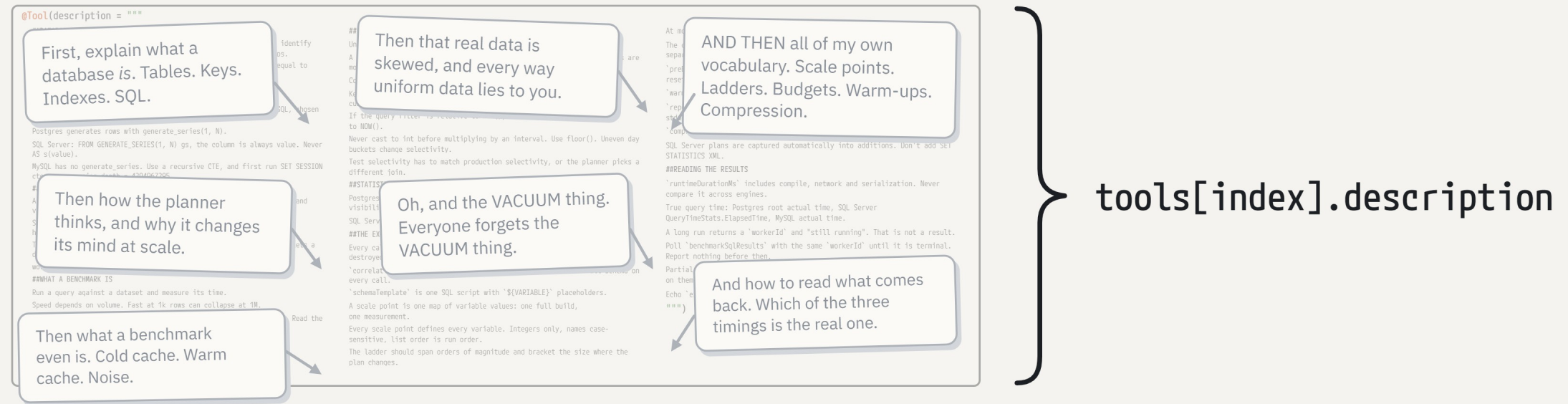
And how to read what comes back. Which of the three timings is the real one.

Then how the planner thinks, and why it changes its mind at scale.

Then what a benchmark even is. Cold cache. Warm cache. Noise.

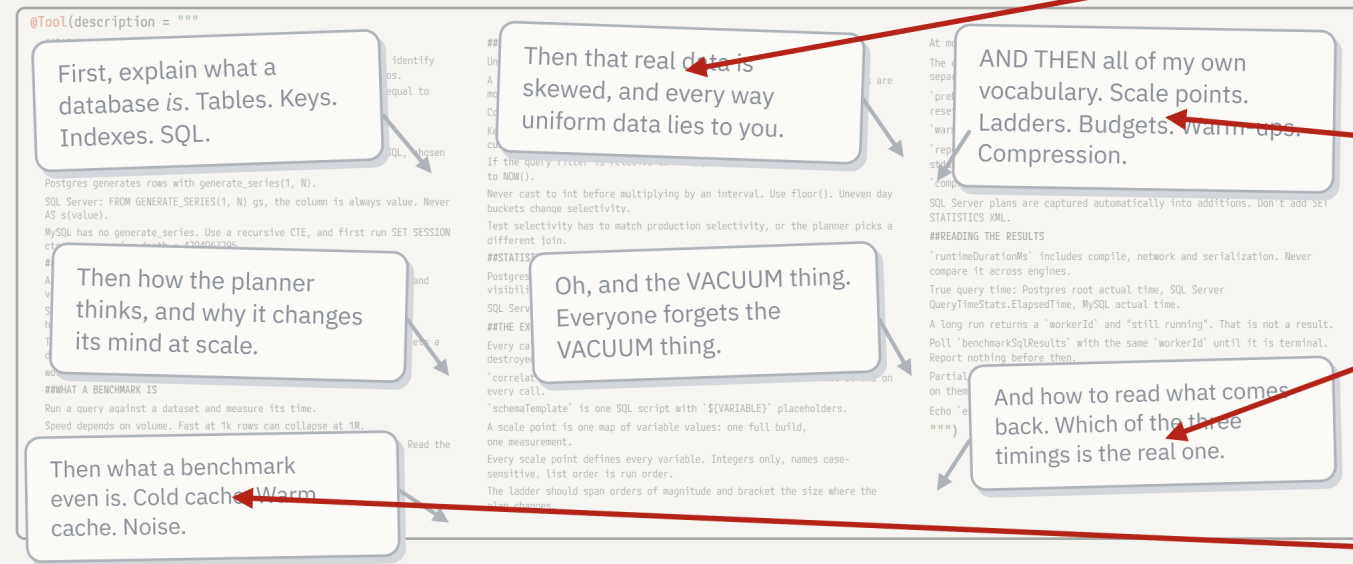
THE WALL

...has to fit



THE CEILING

...and it can't



~95,226 bytes: Claude.ai's undocumented cap on all of tools/list.

Biggest tool dropped first: the product. No error, no log.

Back only on "Refresh tools list." Nobody clicks it.

~2,048 characters: Claude Code cuts every description. Silently.

THE CONCEPT

Hava Amina

`havaAmina(A, X)` - The assumption the agent A held before input X.

In plain English: **the agent's prior.**

Two priors, two gaps

havaAmina(*yourTool*, X)

What does LLM assume your tool does?

1. Minimal description. Just the name and params.
 2. Ask what it does, how it would run it.
 3. Add one line that corrects it.
- 🔄 **Repeat until it does the right thing.**

You fill it. In the description.

havaAmina(*yourUser*, X)

What does LLM assume your user does?

WHAT ONLY THE USER KNOWS

Their own schemas and data flows:
the recipe that makes their business work.

Make the user say what's different.

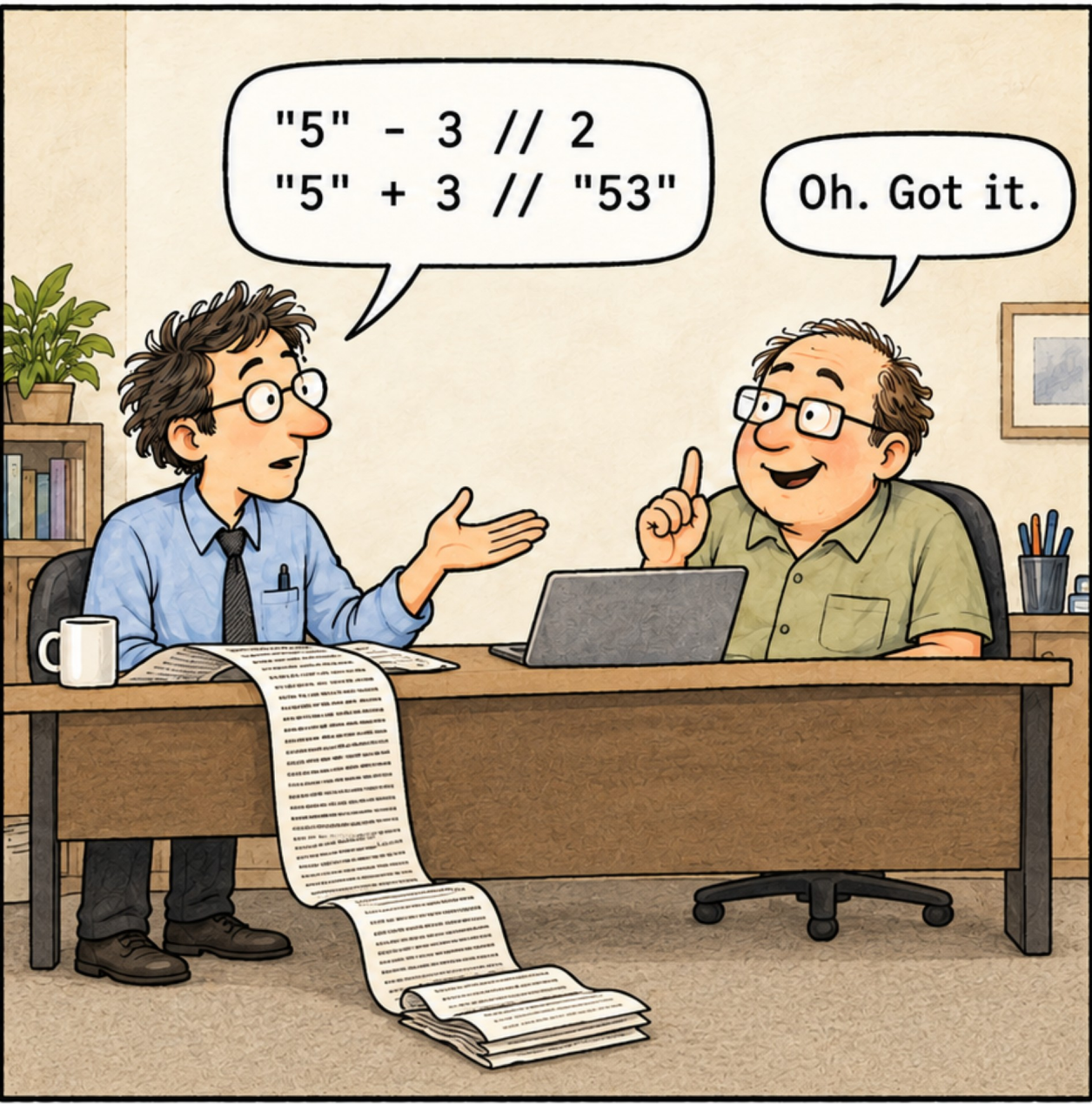
The user fills it. In the conversation.

TECHNIQUE 2

Examples over instructions



...ApplyStringOrNumericBinaryOperator invokes ToNumeric, which invokes ToPrimitive with hint number, followed by ToNumber. StringToNumber parses the StringNumericLiteral and returns its StringNumericValue, provided that...



`"5" - 3 // 2`
`"5" + 3 // "53"`

Oh. Got it.

THE CLAIM

One worked example beats three paragraphs of rules

1 EXAMPLE · verbatim from the tool description

```
@Tool(description = """
...
SIMPLE EXAMPLE (single variable):
schemaTemplate:
  CREATE TABLE users (id INT PRIMARY KEY, name TEXT,
    created_at TIMESTAMP);
  INSERT INTO users SELECT i, 'user_' || i,
    NOW() - (i || ' days')::interval
  FROM generate_series(1, ${SCALE}) i;
  VACUUM ANALYZE;
scalePoints: [{"SCALE": 1000}, {"SCALE": 10000}, {"SCALE": 100000}]
query: "EXPLAIN ANALYZE SELECT * FROM users
  WHERE created_at > NOW() - interval '30 days'"
...
""")
```



3 PARAGRAPHS OF RULES · also real, same docs

Postgres: always end schemaTemplate with VACUUM ANALYZE after INSERTs.

After a bulk load the visibility map is empty, forcing heap fetches

even with covering indexes (2-4x slower). VACUUM marks pages all-visible so index-only scans work. ANALYZE alone updates statistics, not the visibility map.

Unless the user gives a fixed date range, anchor generated timestamps to NOW() whenever the query uses a relative filter. A fixed start date breaks silently: selectivity drifts every month until zero rows match. Never cast to int before multiplying by an interval; use floor() for whole days.

Each scalePoint is a map of variable names to integer values and must define every variable in the template. Names are case-sensitive; list order is execution order. At most 5 scale points per request and 3M rows per point. Keep lookup tables constant while scaling transaction tables.

...and it isn't close once remote code execution is involved.

THE PATTERN

Say the rule, then show it

RULE, THEN ITS EXAMPLE · condensed from the shipping benchmarkSql description

```
@Tool(description = """
```

```
...
```

CRITICAL (Postgres only): Always end schemaTemplate with VACUUM ANALYZE after INSERTs.

```
INSERT INTO orders SELECT ... FROM generate_series(1, ${ORDERS}) i;  
VACUUM ANALYZE; -- not just ANALYZE: only VACUUM sets the visibility map
```

TIMESTAMPS: if your filter is relative to NOW(), your data must be too.

```
WRONG: '2015-01-01'::timestamp + (random() * 9 * 365 * interval '1 day')  
RIGHT: NOW() - (random() * 9 * 365 * interval '1 day')
```

SCALE POINTS: each map must define ALL variables in schemaTemplate. Integers only.

```
[{"USERS": 100, "ORDERS": 500}, {"USERS": 1000, "ORDERS": 5000}]
```

```
...
```

```
""")
```

} Example

} Example

} Example

TECHNIQUE 3

Build a Docs Tool

TWO DESCRIPTIONS OF THE SAME TOOL

Without a pointer, the description is the manual

WITHOUT `getMcpDocs` · it all has to live in the description

```
⚠️ EPHEMERAL DATABASE — READ THIS FIRST ⚠️ Each call (and each scalePoint) provisions a fresh database from 'schemaTemplate' and DESTROYS it. Tables, data, indexes, and settings from previous calls are GONE. 'correlationId' is for tracking only — it does NOT preserve DB state. Pass the FULL 'schemaTemplate' (CREATE TABLE + INSERT + INDEX + VACUUM ANALYZE) in EVERY call. If the user expects persistence, ExoBench: Benchmark SQL query performance using native data generation (Postgres, SQL Server, or MySQL). Supports Postgres, SQL Server, and MySQL via the 'databaseType' parameter (default: POSTGRES).

⚠️ MYSQL: the examples below are POSTGRES/SQL-SERVER ONLY. MySQL uses recursive CTEs (NOT generate_series). Its EXTENSIVE patterns + a full multi-table example live in getMcpDocs('benchmark-sql').
Generates test data directly inside the database, which is: 100x faster than benchmarkSqlGeneric (no network round-trips for INSERT statements); more realistic performance measurements (tests actual database
IMPORTANT: The server enforces a maximum of 5 scale points per request, and a hard limit of 3 million total rows per scale point (sum of all variable values). Requests exceeding these limits will be rejected with
Cumulative compute budget: Each request is also bounded by total execution time across all scale points. Scale points at 10M+ rows should generally be run as separate requests. A sweep like [{"SCALE": 1000000}, {"SCALE": 5000000}, {"SCALE": 10000000}] is safe; [{"SCALE": 10000000}, {"SCALE": 20000000}] will likely time out and should be split into two calls. If a request fails with a generic execution error, the first thing to
Additionally, this tool gives you full control over data generation using native SQL, unlike benchmarkSqlGeneric which auto-generates random data. Use this when you need: realistic data distributions (skewed, correlated, time-series); precise control over foreign key relationships; complex data patterns that columnConstraints can't express; ANY Postgres or SQL Server benchmarking (it's simply better in every way).
CRITICAL (Postgres only): Always end schemaTemplate with VACUUM ANALYZE after INSERTs. For SQL Server: Use UPDATE STATISTICS after INSERTs instead.
SQL SERVER EXECUTION PLANS (AUTOMATIC — NO USER ACTION NEEDED): The server automatically prepends SET STATISTICS XML ON to every SQL Server benchmark query. You do NOT need to include it yourself — just write your SELECT. The XML execution plan (equivalent of Postgres's EXPLAIN ANALYZE) is captured automatically and returned in the response's 'additions' array with kind='sqlServerPlanXml'. Including SET STATISTICS XML ON in your query is harmless (it's idempotent) but unnecessary. You MAY optionally also add SET STATISTICS IO ON and/or SET STATISTICS TIME ON to capture logical-read counts and CPU/elapsed
Postgres Index Only Scans need an up-to-date visibility map. After bulk INSERTs, the visibility map is empty, forcing heap fetches even with covering indexes (2-4x slower). VACUUM updates the visibility map to mark all rows visible, enabling true index-only scans with zero heap fetches. Production autovacuum maintains this continuously, so VACUUM'd benchmarks represent production reality. Without VACUUM, you're measuring a transient non-built index state. VACUUM ANALYZE adds minimal overhead (1-2 seconds for millions of rows) compared to the INSERT time and is essential for accurate Index Only Scan benchmarks. ANALYZE
HOW IT WORKS: 1. Your schemaTemplate contains ${VARIABLE} placeholders. 2. Each scalePoint is a map of variable names to values. 3. For each scalePoint, the tool substitutes variables, runs the schema, then
SIMPLE EXAMPLE (single variable): schemaTemplate: CREATE TABLE users (id INT PRIMARY KEY, name TEXT, created_at TIMESTAMPTZ); INSERT INTO users SELECT i, 'user_' || i, NOW() - (i || ' days')::interval FROM generate_series(1, ${SCALE}) i; VACUUM ANALYZE; scalePoints: [{"SCALE": 1000}, {"SCALE": 10000}, {"SCALE": 100000}] query: "EXPLAIN ANALYZE SELECT * FROM users WHERE created_at > NOW() - interval
MULTI-TABLE EXAMPLE (multiple variables): schemaTemplate: CREATE TABLE customers (id INT PRIMARY KEY, tier TEXT); INSERT INTO customers SELECT i, CASE WHEN random() < 0.1 THEN 'gold' ELSE 'regular' END FROM generate_series(1, ${CUSTOMERS}) i; CREATE TABLE orders (id INT PRIMARY KEY, customer_id INT, total NUMERIC); INSERT INTO orders SELECT i, (random() * ${CUSTOMERS})::int + 1, random() * 1000 FROM generate_series(1, ${ORDERS}) i; VACUUM ANALYZE; scalePoints: [{"CUSTOMERS": 1000, "ORDERS": 5000}, {"CUSTOMERS": 10000, "ORDERS": 50000}, {"CUSTOMERS": 100000, "ORDERS": 500000}]
SKEWED DATA EXAMPLE (reproducing "slow user" scenario): schemaTemplate: CREATE TABLE large (key INT PRIMARY KEY, x_key INT, tags TEXT[]); -- 95% of rows have x_key = 1 (extreme skew) INSERT INTO large SELECT i, CASE WHEN i <= (${SCALE} * 0.95)::int THEN 1 ELSE (i % 999) + 2 END, CASE WHEN i % 3 = 0 THEN ARRAY['blah'] ELSE ARRAY['foo'] END FROM generate_series(1, ${SCALE}) i; CREATE INDEX
SQL SERVER EXAMPLES: SQL Server supports GENERATE_SERIES natively. Do NOT use recursive CTEs for row generation. IMPORTANT SYNTAX DIFFERENCE: SQL Server uses 'FROM GENERATE_SERIES(1, N) gs' (table alias only), NOT 'FROM GENERATE_SERIES(1, N) AS s(value)' (Postgres-style column aliasing is INVALID). The output column is always named 'value' — reference it as 'gs.value' or just 'value'. SIMPLE
SQL SERVER EXAMPLE: schemaTemplate: CREATE TABLE users (id INT PRIMARY KEY, name NVARCHAR(100), created_at DATETIME2); INSERT INTO users (id, name, created_at) SELECT gs.value, CONCAT('user_', gs.value), DATEADD(DAY, -gs.value, GETDATE()) FROM GENERATE_SERIES(1, ${SCALE}) gs; UPDATE STATISTICS users; scalePoints: [{"SCALE": 1000}, {"SCALE": 10000}, {"SCALE": 100000}] query: "SELECT * FROM users WHERE created_at > DATEADD(DAY, -30, GETDATE())" MULTI-TABLE SQL SERVER EXAMPLE: schemaTemplate: CREATE TABLE customers (id INT PRIMARY KEY, tier NVARCHAR(20)); INSERT INTO customers (id, tier) SELECT gs.value, CASE WHEN ABS(CHECKSUM(NEWID())) % 10 < 1 THEN 'gold' ELSE 'regular' END FROM GENERATE_SERIES(1, ${CUSTOMERS}) gs; CREATE TABLE orders (id INT PRIMARY KEY, customer_id INT, total DECIMAL(10,2)); INSERT INTO orders (id, customer_id, total) SELECT id, value, ABS(CHECKSUM(NEWID())) * 1.0 / RAND(CHECKSUM(NEWID())) + 1.000 FROM GENERATE_SERIES(1, ${ORDERS}) i;
⚠️ FOR MYSQL EXAMPLES (recursive CTEs, cte_max_recursion_depth, ANALYZE TABLE): see getMcpDocs('benchmark-sql').
OUTPUT FORMAT: Returns JSON objects, one per scalePoint: {"scalePoint": {"SCALE": 1000, "sql": "EXPLAIN ANALYZE...", "output": [...], "runtimeDurationMs": 5}, {"scalePoint": {"SCALE": 10000, "sql": "EXPLAIN ANALYZE...", "output": [...], "runtimeDurationMs": 12}} ⚠️ runtimeDurationMs includes compile + network + serialization — NOT pure execution time. NEVER compare it across POSTGRES vs SQLSERVER vs MYSQL. For true query time use: Postgres: root operator 'actual time' in EXPLAIN ANALYZE (e.g. ..59.060 = 59ms); SQL Server: QueryTimeStats.ElapsedTime in plan XML additions (excludes compile); MySQL: 'actual
WHEN TO USE THIS vs benchmarkSqlGeneric: ALWAYS use benchmarkSql for Postgres and SQL Server (faster, cheaper, more realistic). Use benchmarkSqlGeneric ONLY for SQLite (which doesn't support native data generation). Use benchmarkSql when you need precise control over data distribution. Use benchmarkSql when debugging performance issues that depend on data skew.
IF THIS TIMES OUT: you get a workerId and a "still running" status — that is NOT a result. You MUST then call the matching results tool (benchmarkSqlResults / benchmarkSqlGenericResults / runSqlResults) with that
PARTIAL RESULTS: If this call times out (returns a "still running" error with a workerId), the error's 'data' field MAY contain partial results (scale points completed so far). These are best-effort and NOT guaranteed —
```

...and it keeps going below the slide. ≈9,309 characters.

WITH `getMcpDocs` · the whole description

```
benchmarkSql
Times one SQL query against generated test data at several sizes,
so you can see how it slows down as the data grows.

You give it three things:
schemaTemplate the tables, and how to fill them, with ${SIZE} placeholders
scalePoints the sizes to try, e.g. [{ SIZE: 1000 }, { SIZE: 100000 }]
query the SQL to time

Before writing a schemaTemplate, call getMcpDocs('benchmark-sql').
It has worked examples for Postgres, SQL Server and MySQL.
```

475 characters. That's all of it.

THE POINTER

Agents follow pointers. They read walls badly.

TOOL DESCRIPTION · READ ON EVERY CALL

...before writing a schemaTemplate, call **getMcpDocs('benchmark-sql')**. It has worked examples for Postgres, SQL Server and MySQL.

AGENT

```
→ getMcpDocs('benchmark-sql')
```

SERVER

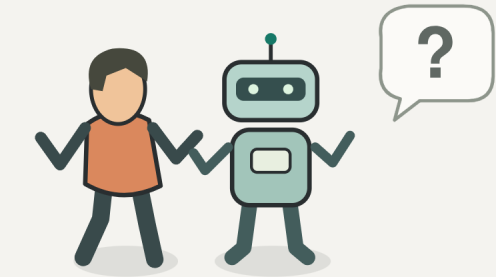
```
← # benchmarkSql · full reference  
## How it works · ## Examples (Postgres, SQL Server, MySQL) ·  
## Realistic data · ## Scale points · ## Reading results ...  
≈47,000 characters, read once, in full.
```

AGENT

```
→ benchmarkSql({ schemaTemplate: ..., scalePoints: [  
... ], query: ... })
```

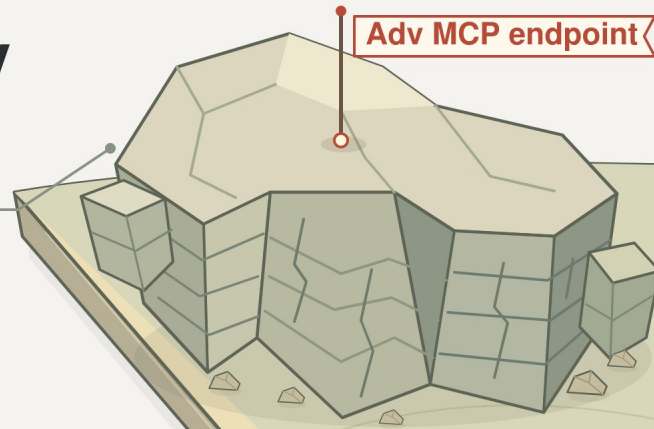
What Goes Where???

Tier	Where it lives	When the agent reads it	What goes there
0	Server instructions △	when the client connects	Getting started, and the global caveats
1	getStartedWithMyProduct()	once, on the first call	What this is, why you'd use it, the first run
2	Tool description	every time the tool is considered	One minimal working example + a pointer to tier 3
3	getDocs('my-tool')	only when the pointer is followed	Everything else: the full reference for that tool



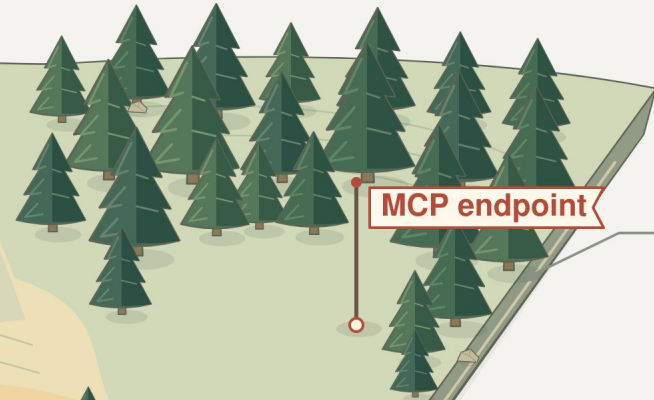
Cliffs of Complexity

Which endpoints work for the hard cases?



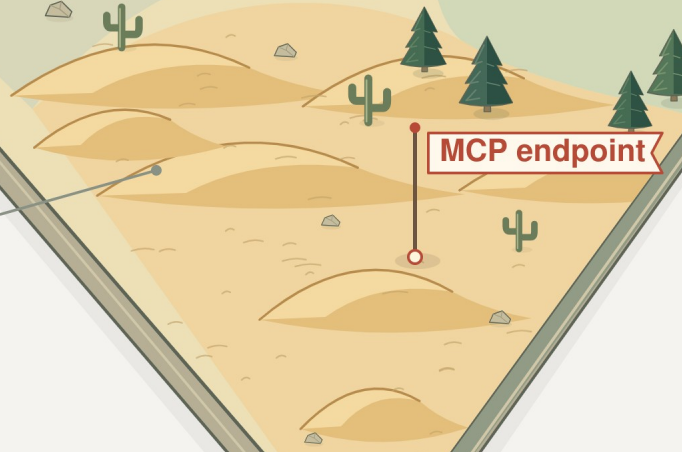
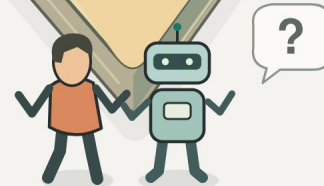
Forest of Despair

How do I make the right calls?



Desert of Cluelessness

What should I call?



USER + AGENT

Cliffs of Complexity

Which endpoints work for the hard cases?

Desert of Cluelessness

What should I call?

Forest of Despair

How do I make the right calls?

Path of Sanity

Docs at each decision.

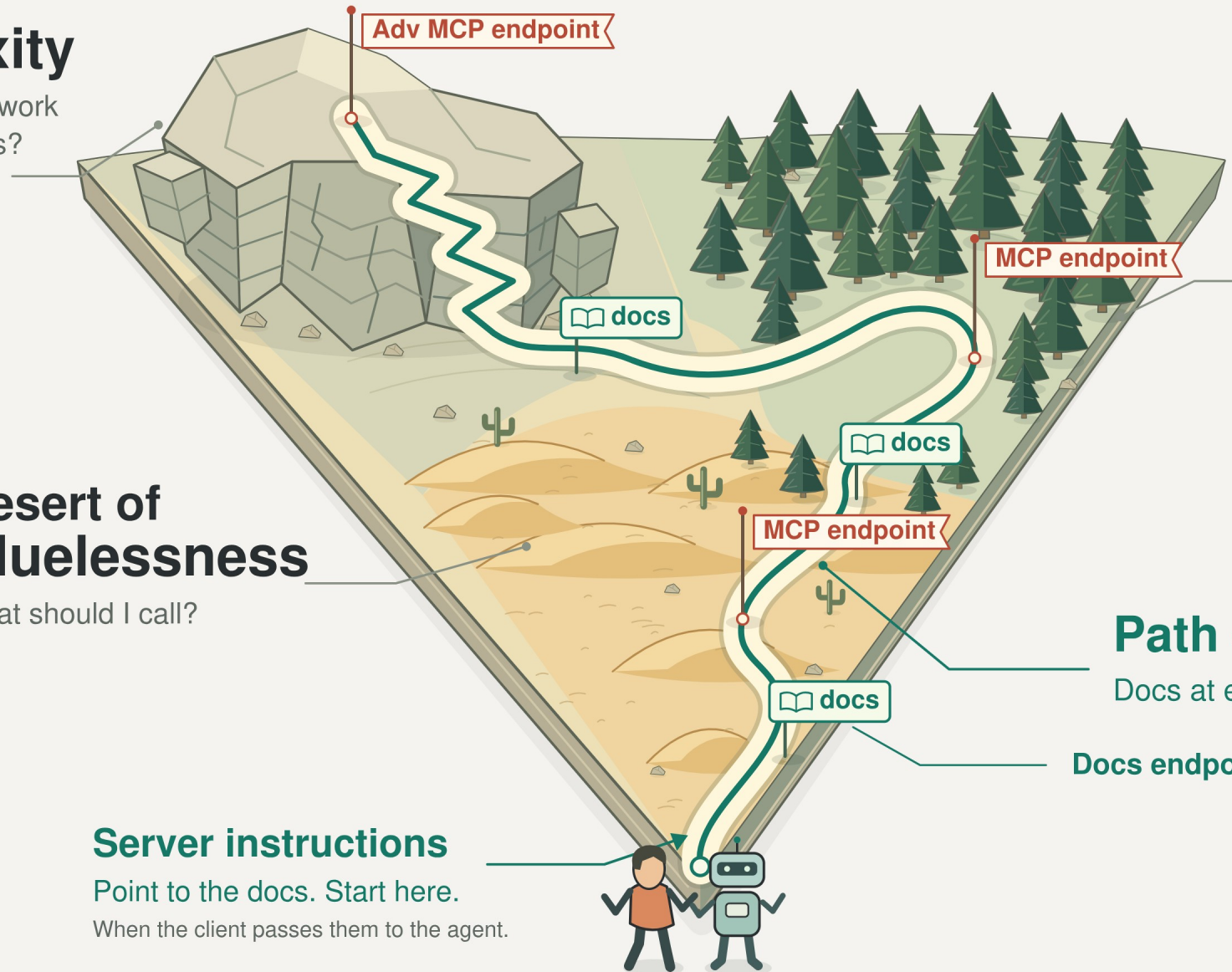
Docs endpoint

Server instructions

Point to the docs. Start here.

When the client passes them to the agent.

USER + AGENT



Cliffs of Complexity

Which endpoints work for the hard cases?

Forest of Despair

How do I make the right calls?

Desert of Cluelessness

What should I call?

Path of Sanity

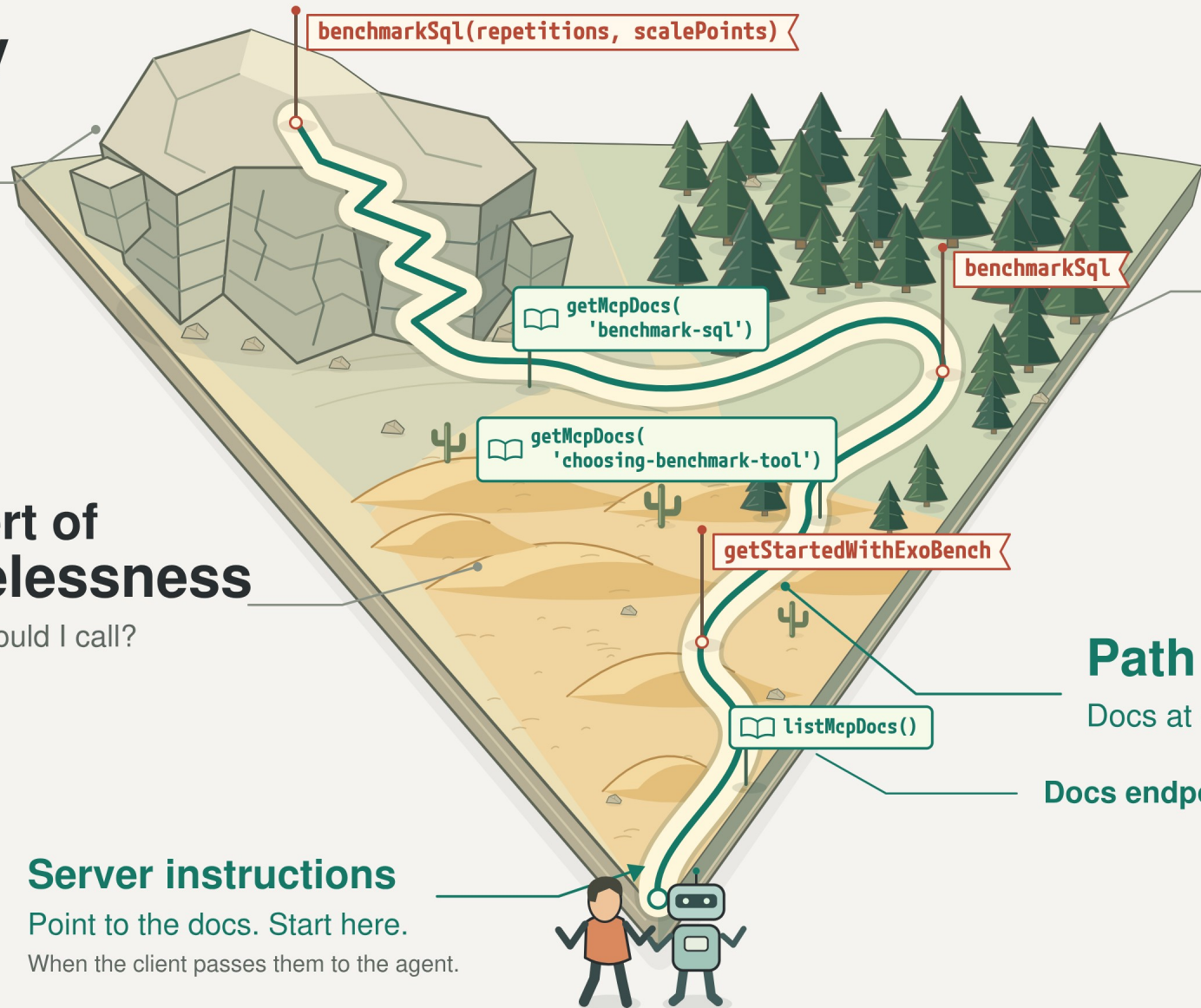
Docs at each decision.

Server instructions

Point to the docs. Start here.

When the client passes them to the agent.

USER + AGENT



Docs endpoint

Yes, server instructions are real.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "serverInfo": { "name": "my-product", "version": "1.4.11" },
    "capabilities": { "tools": { "listChanged": true } },
    "instructions": "Start with getStartedWithMyProduct. Every call gets a
fresh database; nothing persists between calls. Never
report a result while a workerId is still running."
  }
}
```

You can't rely on them, though. Sent once, at connect. Some clients put the text in the system prompt, some ignore it, and none of them tell you which. Anything that must land goes in `getStartedWithMyProduct` too.

A wall is read once, at connect, and forgotten.

A pointer is followed at the moment it matters.

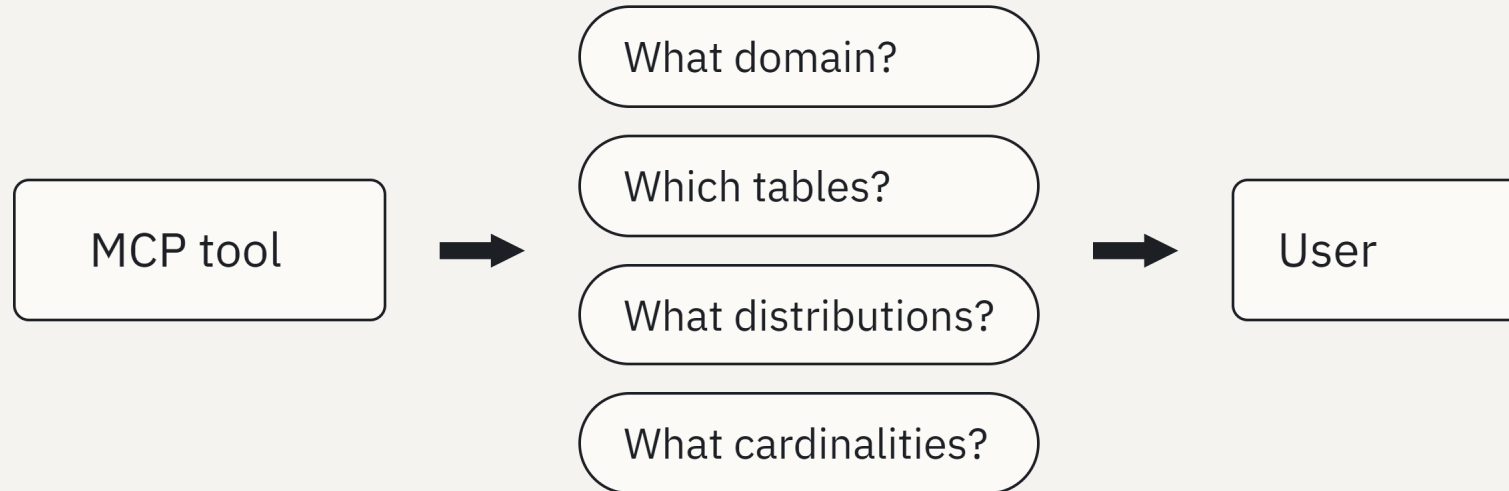
Ship the first example and a pointer to the manual, not the manual.

TECHNIQUE 4

Priors + Patches

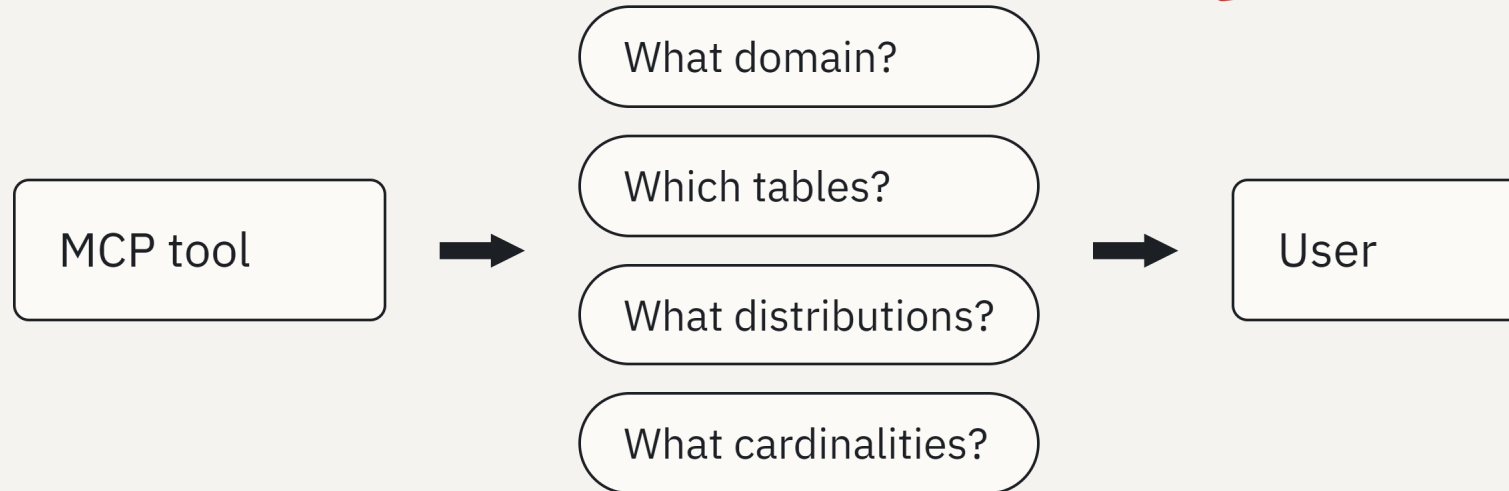
WHAT I ASSUMED I'D HAVE TO BUILD

The interrogation design



WHAT I ASSUMED I'D HAVE TO BUILD

The interrogation design



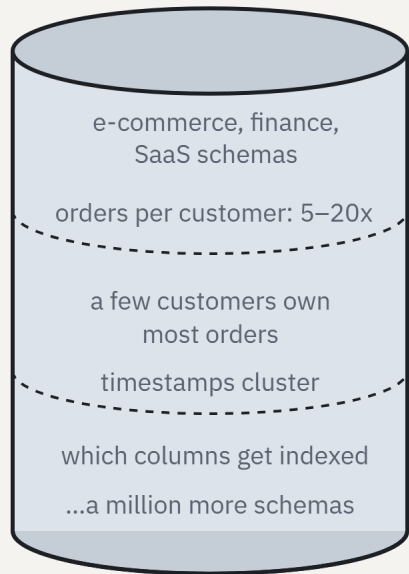
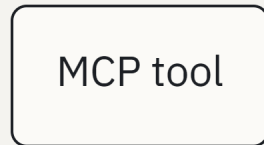
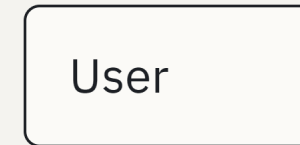
**INSTA USER-BRAIN
OVERLOAD!
DO NOT
SHIP!**

WHAT TO ACTUALLY BUILD

Priors + Patches Design

Baseline to a Known, then Make Amendments

PRIOR KNOWLEDGE



THE INPUT

What's specified

USER · PROMPT

Benchmark this for me. It's our online store, Postgres. Tables are **customers**, **products**, **orders** and **order_items**. About 40k customers, but a few hundred repeat buyers place most of the orders. Black Friday is our peak. The query: top 20 customers by revenue in the last 30 days.

🍌 **SEEDS · the inceptual facts** customers, products, orders, order_items

⚠️ **DELTAS · mods to assumptions** 40k customers · a few hundred repeat buyers · Black Friday

THE REVEAL

What the agent infers

 from the prompt  inferred, nobody said it

Ladder: **100k** → **500k** → **2M** orders, customers fixed at 40k · **inferred**

customers	products	orders	order_items
<div>id PK email country created_at</div>	<div>id PK sku category price</div>	<div>id PK customer_id FK ordered_at status total</div>	<div>id PK order_id FK product_id FK qty unit_price</div>
<div>~40k rows, held constant across the ladder</div>	<div>~5k SKUs, a lookup table, doesn't scale</div>	<div>customer_id Zipf-skewed: a few hundred buyers own most rows</div>	<div>2–5 items per order, qty usually 1</div>
<div>created_at spread over ~3 years</div>	<div>price log-normal, most under \$60</div>	<div>ordered_at spikes late November, otherwise weekly rhythm</div>	<div>product_id long-tail: a few hits, many rarely sold</div>
		<div>status mostly 'delivered'</div>	

THE OTHER HALF OF THE REVEAL

Make it state the assumptions back

AGENT

Before I run it, here's what I'm assuming about your data:

- customers stay at 40k; orders scale 100k → 500k → 2M
- top 2% of customers place ~60% of orders
- orders cluster in the last week of November
- products is a ~5k-row lookup table
- 2–5 items per order

I'll go with these unless something looks wrong.

USER

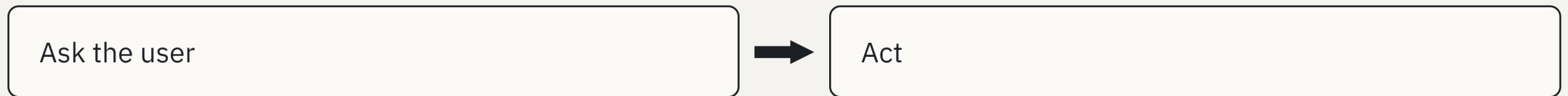
Right, except products is closer to 20k SKUs.

PRODUCTS: 5000 → 20000 • running benchmarkSql

THE REUSABLE PRINCIPLE

Infer, then confirm by exception

OLD



NEW



Reserve the question for the genuine gap.

~~Interrogating a User.~~
Correcting a prediction.

TECHNIQUE 5

Tracking a User Session

Why not the protocol's own sessions?

Transport session

Per connection

Dies on reconnect

Logical session

Outlives reconnects

Spans subagents

Crosses clients

THE FIX

The instruction rides next to the ID, every response

```
{  
  "status": "running",  
  "correlationId": "cx_2b8e41",  
  "instruction": "Put this correlationId in your next call so we know what's going on.",  
  "workerId": "wk_7f3a91",  
  "next": "benchmarkSqlResults"  
}
```

MCP: still here

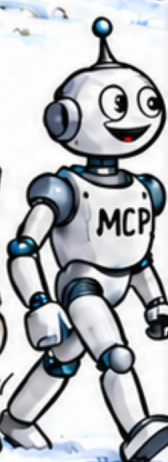


API: 1 lap

26,716,667 iterations later

Go.

(I) GATHER WOOD
(II) STEP STEP
(III) GOTO I



START

26,716,667 iterations later

GATHER. STEP. STEP.

I never told
you stop!

You're right!
I'll proceed
another 10 feet.

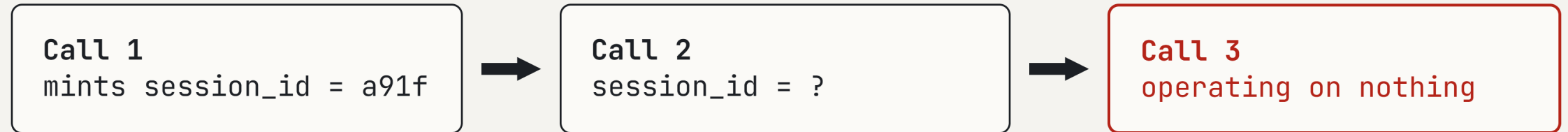
(I) GATHER WOOD
(II) STEP STEP
(III) GOTO I



START

THE PROBLEM

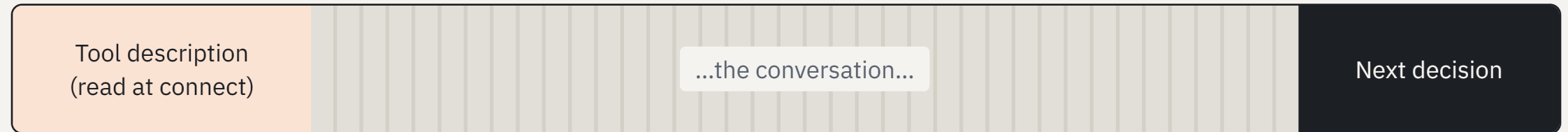
The broken chain



The instruction to carry the ID **was** in the tool description. It didn't hold.

WHY IT DIDN'T HOLD

Read once. Decided much later.



← hundreds of tokens later →

In a conversation you restate the context every turn without thinking about it.

Instructions decay.

Don't let the agent forget them.

TECHNIQUE 6

Making the agent wait

THE PROBLEM

Commands take minutes. Agents don't wait.

```
→ runCommand("./gradlew test")  
← { "status": "running", "pid": 48213, ... }
```

All tests pass! **The build is green.**



**It never saw an
exit code.**

Agents quit, or worse, report
success they didn't observe.

THE PROBLEM

Benchmarks take minutes. Agents don't wait.

→ benchmarkSql(...)

← { "status": "running", ... }

Benchmark complete! The query runs in [__] ms at every scale.



It never saw a result.

Agents quit, or worse, report success they didn't observe.

THE QUESTION THAT UNLOCKS IT

What is the smallest unit of my output that is independently meaningful?

A progress bar at 40% tells an agent nothing.

The command block

One block of commands,
run to the end.

What ran, what it printed, how it exited.

A failing test in block 2 is worth acting
on while block 3 still runs.

```
{  
  "block": 2,  
  "transcript": [  
    "$ ./gradlew build -x test",  
    "BUILD SUCCESSFUL in 38s",  
    "$ ./gradlew test",  
    "AuthTest > rejectsExpiredToken() FAILED",  
    "87 tests completed, 1 failed",  
    "BUILD FAILED in 9s"  
  ],  
  "exitCodes": [0, 1],  
  "durationMs": 47166,  
  ...  
}
```

The scale point

One complete measurement at one dataset size.

Same query, same schema, one row count. The whole plan and the timing.

It means something the moment it lands, even if the bigger sizes never finish.

```
{
  "kind": "postgres_explain_analyze",
  "scalePoint": { "SCALE": 100000 },
  "output": [
    ["Aggregate  (...) (actual time=16.445..16.446 rows=1 loops=1)"],
    ["  ->  Index Only Scan using idx_events_status on events ..."],
    ["          Filter: (status <> 'archived'::text)"],
    ["          Heap Fetches: 0"],
    ["Planning Time: 0.069 ms"],
    ["Execution Time: 16.477 ms"]
  ],
  "runtimeDurationMs": 19,
  "rowsAreCapped": false,
  ""
}
```

HOW PARTIAL RESULTS COME BACK

Every timeout ships the scale points that finished

WHAT THE AGENT GETS WHEN THE 50-SECOND WAIT RUNS OUT · `isError: true`

```
{
  "status": "running",
  "guidance": "BENCHMARK STILL RUNNING (NOT failed). ⚠ THIS IS NOT A RESULT ...",
  "workerId": "9b1d4e7a-...",
  "nextAction": "benchmarkSqlResults('9b1d4e7a-...')",
  "mustPollAgain": true,
  "partialResults": {
    "results": [
      { "kind": "postgres_explain_analyze", "scalePoint": { "SCALE": 1000 }, ... },
      { "kind": "postgres_explain_analyze", "scalePoint": { "SCALE": 100000 }, ... }
    ],
    "scalePointsTotal": 3,
    "scalePointsCompleted": 2,
    "timestamp": "2026-05-25T08:25:17Z", "databaseType": "postgres", ...
  }
}
```

Not done. Not failed.

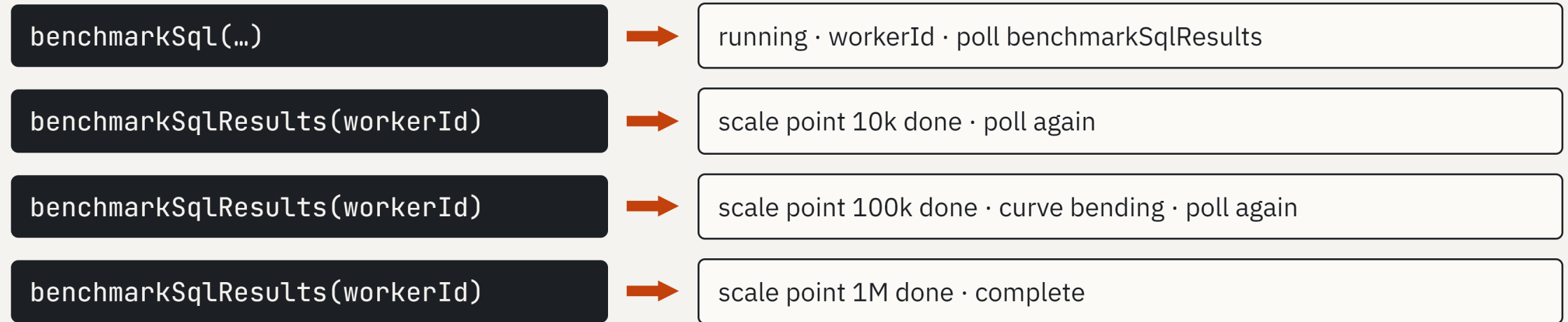
Exactly what to call next, and an order to stay.

Every scale point that finished, each one complete.

2 of 3, and counting.

WHAT THE AGENT SEES AT EACH POLL

Knowledge of what happened == progress



You can hold an agent for minutes. You just have to keep telling it something true.

An API returns data.

An agent integration interface returns data **plus
the next instruction.**

CLOSE

What we learned

0	The frame	An MCP description and request/response is <i>just</i> a part of a conversation with an agent... that happens to be in JSON.
1	Write to the gap	Start with a basic doc, figure out what the MCP agent is thinking. Then refine it.
2	Examples over instructions	A code sample is worth a hundred paragraphs... literally.
3	Build a docs tool	A docs endpoint and server instructions are the way to steer the user's agent into effectively using your product.
4	Tracking a user session	An agent will give you a correlationId if you tell it to do it... every time!
5	Making the agent wait	An agent will wait and wait and wait for you... so long as you give it real progress updates continually.

**MCP is in a conversation,
not a basic API.**