Talk to Your Data Quality
TestGen MCP and AI Agents

A DataKitchen Webinar · 2026

Before We Start

Christopher Bergh
CEO, Head Chef
DataKitchen

Welcome!

Slides, recording, and transcription will be shared this week.

Put your questions in the chat window; we will answer questions at the end of the session.

We have targeted 45 minutes for our presentation, with questions at the end.

Agenda

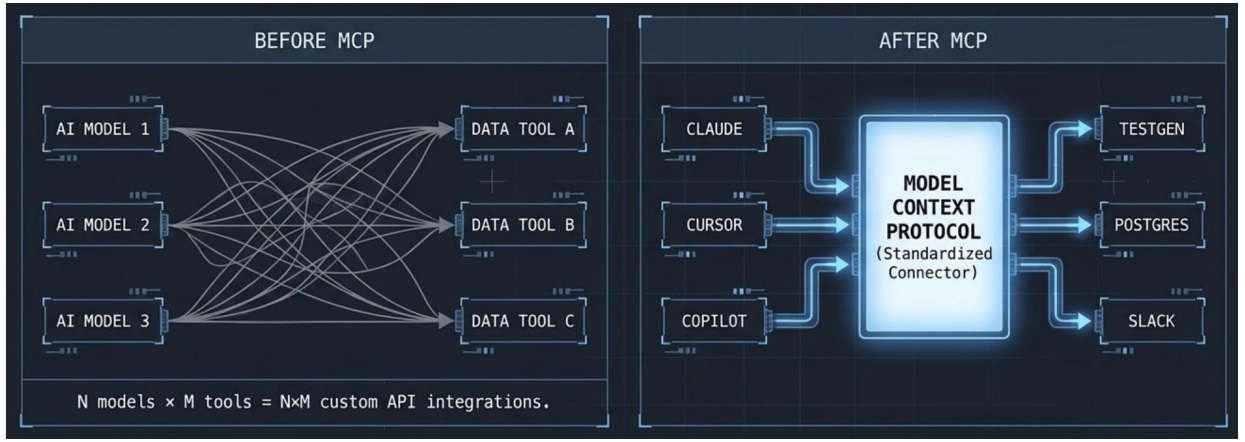
1. **What is MCP and why does it matter?**
2. Intro To Testgen MCP Server
3. Testgen UI vs. MCP
4. Testgen MCP Server As An AI Dialog-Based UI
5. Testgen MCP As The Foundation For AI Agents
6. Conclusion

DataKitchen

MCP is USB-C for AI

One standardized connector instead of a custom integration for every model and tool

- MCP stands for **Model Context Protocol**
  - Open source standard for connecting AI applications to external systems
- Data sources, tools, and workflows, all through one protocol
- Before MCP: N models × M tools = N×M custom integrations
- After MCP: build once, connect everywhere



DataKitchen

The three pieces of MCP

The Host is the app you use. It talks to a Server through a Client that lives inside it.

- Five capabilities: **Resources, Prompts, Tools, Sampling, Roots**
  - In practice, Tools do the work: query a database, run a job, send a message

DataKitchen

Why the MCP ecosystem won

One protocol, and work done once benefits everyone

- Everyone building with LLMs hits the same wall: getting a model to use tools reliably
- Every major data quality and observability vendor shipped an MCP server within six months
    - No Open Source – enterprise pricing.
- The agentic story is settled. What still differs is where the server runs.
    - TestGen's MCP server runs behind your firewall, Apache 2.0, on the laptop you use for dbt
    - Your warehouse credentials stay on your network.

DataKitchen

Agenda

1. What is MCP and why does it matter?
2. **Intro To Testgen MCP Server**
3. Testgen UI vs. MCP
4. Testgen MCP Server As An AI Dialog-Based UI
5. Testgen MCP As The Foundation For Ai Agents
6. Conclusion

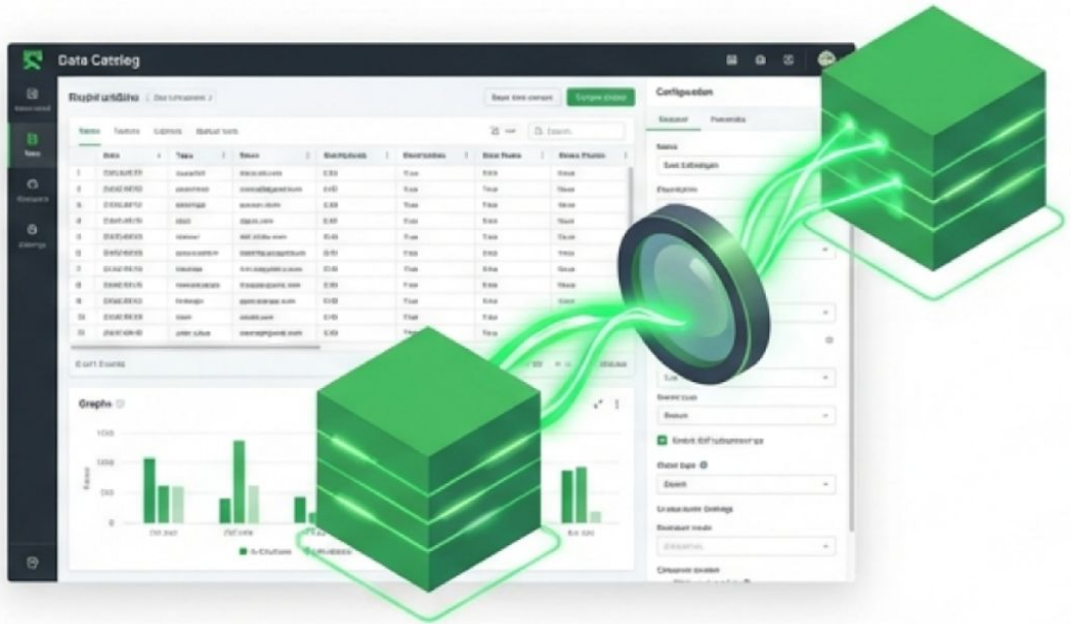
DataKitchen

DataKitchen DataOps Data Quality TestGen:

Open Source, Full Featured, Data Quality Tool.

IT DOES SIX TASKS:

1. Data Profiling
2. Dataset Screening And Hygiene Review
3. AI Generation of Data Quality Validation Tests, Custom Tests
4. Data Testing
5. Anomaly Monitoring
6. Data Quality Scoring



KEY FEATURES

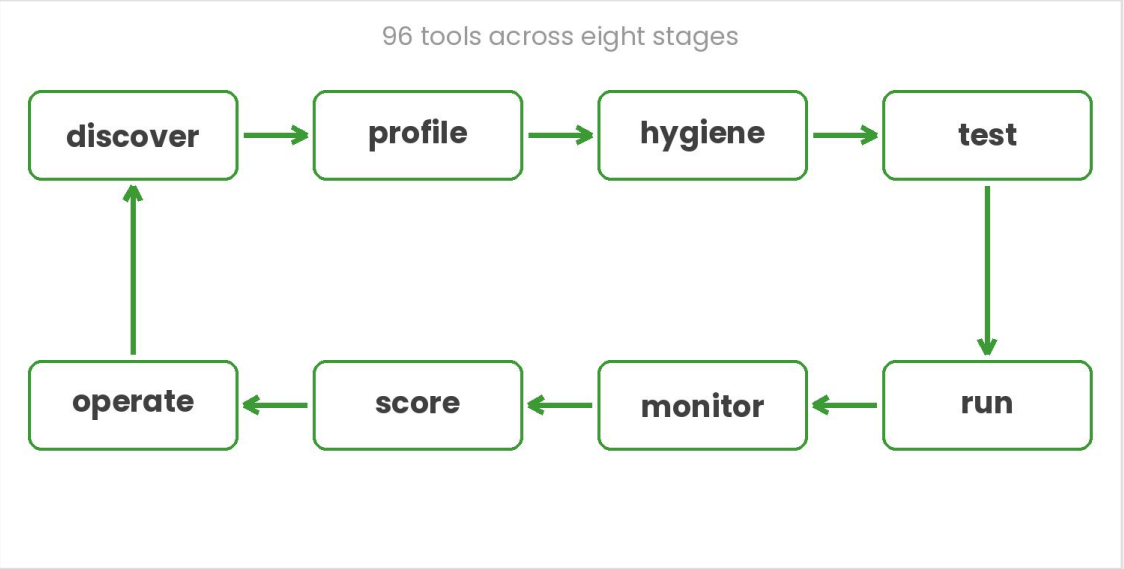
- ✓ In Database Execution
- ✓ Full Featured (UI, AI, Rules) One User
- ✓ Enterprise Version starts at $100 per user per month

DataKitchen

TestGen MCP: the complete loop

TestGen 5.70.2 completes the server. 96 tools covering the whole data quality loop.

- Eight stages: **discover, profile, hygiene, test, run, monitor, score, operate**
- Callable from Claude, Claude Code, Cursor, Copilot, or the agent you're building
- You no longer need the TestGen UI open to run TestGen. You need a chat window.
- Open source under Apache 2.0, same as TestGen itself



DataKitchen

TestGen MCP: Secure

- Every tool that writes to your TestGen instance carries an orange dot on the cheat sheet. Everything else is read-only.
- The server authenticates as you and runs with your TestGen role
  - An analyst's assistant can read scores. It cannot delete scorecards.
- Dispositions, test notes, and run history land in TestGen the same way as UI actions.
- The agent leaves footprints your governance team can follow.
- https://datakitchen.io/blog/testgen-mcp-cheat-sheet/

Five-minute setup

No SKU, no per-seat agent tax

1. Point your MCP client at `https://<your-testgen-host>/mcp`
2. Sign in with browser OAuth or a personal access token
3. Ask: **"List the projects I have access to in TestGen."**
4. If your projects come back, you're live

Setup guide

docs.datakitchen.io/testgen/mcp/setup covers personal access tokens and hosted platforms like Databricks Genie

DataKitchen

Agenda

1. What is MCP and why does it matter?
2. Intro To Testgen MCP Server
3. **Testgen UI vs. MCP**
4. Testgen MCP Server As An AI Dialog-Based UI
5. Testgen MCP As The Foundation For Ai Agents
6. Conclusion

DataKitchen

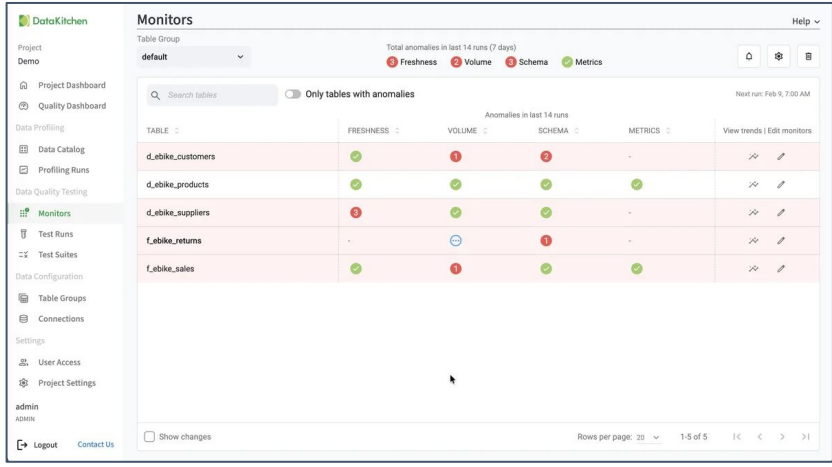
The TestGen UI isn't going anywhere

MCP is a second front door, not a replacement

DataKitchen

The TestGen UI isn't going anywhere

MCP is a second front door, not a replacement



Use the UI

- Interactive and easy: Quality Dashboard, Score Explorer, Data Catalog, test results
- Dense and quick once you know it
- If you live in TestGen every day, the UI is still the fastest way to work

Where the UI stops

- No AI chat interface, on purpose. TestGen has no built-in chatbot.
- MCP fills the gap for three moments:
  - You'd rather ask in English from Slack
  - You hand a teammate a way in without training them
  - You let an agent run on a cron at 3 am

Same data, same security model, different surface.

DataKitchen

Agenda

1. What is MCP and why does it matter?
2. Intro To Testgen MCP Server
3. Testgen UI vs. MCP
4. **Testgen MCP Server As An AI Dialog-Based UI**
5. Testgen MCP As The Foundation For AI Agents
6. Conclusion

DataKitchen

AI dialogue: one window, not five screens

You ask in plain English. The tool answers with the actual data and the actual history.

The questions you actually ask

- Why did this fail? Is it new?
- Show me the rows
- What changed since last Tuesday?
- What test is missing from this table?

What you can do

- **Diagnose** Root Causes
- **Update** source data quality, make changes to data
- **Improve** your test suites by adding new tests
- **Do** almost everything the UI can do

The investigation that used to span five screens fits in one window.

DataKitchen

A real conversation: blind to trusted

One question, thirty seconds, a morning of work

- **"What data do I have and how healthy is it?"** Full inventory, a quality score of 78.9 on a table group we'd been ignoring
- **"What's dragging the score?"** 30 hygiene issues nobody had tested for
  - Every column loaded as varchar(65535). Prices 96% numeric, dates 97% dates, and the remaining 3% strings like ERROR and UNKNOWN hiding in numeric columns.
- **"Show me the rows behind that."** The actual offending records from the source database, not a chart about them
- **Look at the product table in our warehouse. Find the latest test failures, investigate root causes, recommend fixes, and graph the failure counts over the last 30 days. Then suggest any new tests we should add and add them to the suite.**
- **"Recommend tests and run them."** Twelve failures. Dismiss one, confirm the rest, every disposition lands with an audit trail.

DataKitchen

Six slash commands for triage

Triage is where data quality programs die. The assistant doesn't get tired of failure 40 at 8 am.

- **`/health_check`** What's on fire, in a summary, not a dashboard
- **`/investigate_failures`** Failures grouped by test type and column, with the actual rows
- **`/hygiene_triage`** Confirm, dismiss, or mute the 32 hygiene categories
- **`/compare_runs`** What changed since yesterday
- **`/table_health`** Is this specific table healthy
- **`/profiling_overview`** Onboard a new dataset

DataKitchen

Dialogue finds the tests you're missing

Profile + existing tests + 49 possible test types = coverage gaps fall out

- The assistant holds three things in context at once:
  - What your data looks like (the profile)
  - What tests already exist (the suite)
  - What tests are possible (the catalog of 49 test types)
- Example: email column is 99.7% valid format in the profile, but no Email Format test exists in the suite. Recommend one, with the column, the threshold, and the reason.
- You confirm the ones you want. The assistant adds them to the suite.

Column	Profile says	Recommend
email	99.7% valid format, no test	Email Format
customer_id	100% unique, not null, no test	Uniqueness
country_code	47 values, should be 2-letter ISO	List of Values
signup_date	tight 3-year range, no test	Date Range

DataKitchen

Demo

Update Poor Data Quality

DataKitchen

> look at all the hygene issues and data test failures and list the first three I should investigate. ignore the priority (Likely, definite) and type (error warning) and make your suggestion.

● Let me pull both hygiene issues and test failures in parallel.

Called testgen **3** times

● Here are the three I'd tackle first, based on severity, scale, and downstream impact:

---

1. Duplicate customer_id in d_ebike_customers (25 duplicates)

This is a primary key integrity failure. Every join, every sales analysis, every customer metric built on this table is potentially double-counting or mis-attributing records. It poisons everything downstream. Fix the root cause here before trusting any customer-level reporting.

---

2. credit_card in d_ebike_customers -- 169 duplicates AND 169 pattern mismatches

Two separate tests failing on the same column at the same scale (169 records) is a strong signal of something systemic -- corrupted load, field misuse, or a schema change that left the column storing the wrong thing. The overlap between both failures at identical counts makes this look like a wholesale column problem, not scattered bad rows.

---

3. supply_reliability in d_ebike_suppliers is completely empty (28 of 28 records)

Every row in the supplier table has an empty string in this column, and manufacturing_certifications is also 100% null. A table named after suppliers, where the reliability and certification fields carry no data at all, means any supplier risk analysis is running blind. The question is whether this data was never loaded or was lost -- either way, it is a gap that silently undermines operational decisions.

The PII flags and the sales amount minimums are real but lower urgency -- the three above affect data integrity at scale or create structural blind spots.

› what is the sql to fix the customer_id issue in d_ebike_customers? It is a postgres DB.

● Let me look at the actual duplicate data before writing a fix.

  Called testgen 2 times

● Now I have everything I need. Here is the SQL, in the order you should run it.

  Step 1 -- See the duplicates before touching anything:

```
SELECT customer_id, COUNT(*) AS occurrences
FROM demo.d_ebike_customers
GROUP BY customer_id
HAVING COUNT(*) > 1
ORDER BY occurrences DESC;
```

  Step 2 -- Preview which rows will be deleted (keep the most recently contacted record per customer_id):

```
WITH ranked AS (
  SELECT ctid,
         customer_id,
         last_contact,
         ROW_NUMBER() OVER (
           PARTITION BY customer_id
           ORDER BY last_contact DESC, ctid ASC
         ) AS rn
  FROM demo.d_ebike_customers
)
SELECT * FROM ranked WHERE rn > 1;
```

Step 3 -- Delete the duplicates (wrap in a transaction so you can roll back):

```
BEGIN;

WITH ranked AS (
  SELECT ctid,
         ROW_NUMBER() OVER (
           PARTITION BY customer_id
           ORDER BY last_contact DESC, ctid ASC
         ) AS rn
  FROM demo.d_ebike_customers
)
DELETE FROM demo.d_ebike_customers
WHERE ctid IN (SELECT ctid FROM ranked WHERE rn > 1);

-- Verify: should return 0
SELECT COUNT(*) FROM demo.d_ebike_customers
GROUP BY customer_id
HAVING COUNT(*) > 1;

-- If clean, commit. Otherwise ROLLBACK.
COMMIT;
```

Step 4 -- Enforce uniqueness going forward:

```
ALTER TABLE demo.d_ebike_customers
  ADD CONSTRAINT d_ebike_customers_customer_id_pk PRIMARY KEY (customer_id);
```

A few caveats before you run this:

- The tiebreaker here is last_contact DESC -- keeping the most recently active record. If your business rule is different (e.g., keep earliest first_contact), swap the ORDER BY.
- Check whether f_ebike_sales has a foreign key on customer_id. If duplicate rows in the customer table are referenced by sales rows, you may need to repoint those sales records before deleting.
- Run Step 3 in a transaction and verify the count check returns 0 before committing.

* Baked for 44s

> what is the SQL to fix the hygiene issue Similar Values Match When Standardized on Table d_ebike_products
> Column
> product_type

- Let me pull the actual values in that column before writing anything.

  Called testgen **2** times

- The picture is clear. Eight distinct values collapse to three when standardized. "Bicycle" and "E-scooter" are clean. The problem is five variants of "E-bike" that should all be the same value.

Raw value	Count
E-bike	9
e-bike	1
EBike	1
Ebike	1
EBIKE	1
E--Bike	1

Step 1 -- Preview what will change:

```
SELECT product_type, COUNT(*) AS records
FROM demo.d_ebike_products
WHERE product_type IN ('e-bike', 'EBike', 'Ebike', 'EBIKE', 'E--Bike')
GROUP BY product_type
ORDER BY product_type;
```

Step 2 -- Fix it (normalizing all variants to "E-bike", the dominant form):

```
BEGIN;

UPDATE demo.d_ebike_products
SET product_type = 'E-bike'
WHERE product_type IN ('e-bike', 'EBike', 'Ebike', 'EBIKE', 'E--Bike');

-- Verify: should show exactly 3 distinct values
SELECT product_type, COUNT(*) AS records
FROM demo.d_ebike_products
GROUP BY product_type
ORDER BY product_type;

COMMIT;
```

After the fix you should see:

product_type	records
Bicycle	15
E-bike	14
E-scooter	12

One thing to confirm before running: make sure "E-bike" is the canonical value your business uses. If a style guide or downstream system expects a different casing (e.g., "eBike" or "E-Bike"), substitute that into the SET clause instead.

> look at the info in testgen, what is a custom test that you would add?

● Let me explore the data and existing test coverage to find a meaningful gap.

  Called testgen 3 times

● Now I have a full picture. 214 tests across four tables, but one gap stands out: there is no cross-table business rule test anywhere in the suite. Every test validates a single column in isolation.

  The most valuable custom test to add:

  Discount amount exceeds the product's maximum allowed discount

  f_ebike_sales has a discount_amount column, and d_ebike_products has a max_discount column that exists precisely to cap it. No standard TestGen test can join across tables to enforce that rule. Given the existing failures on adjusted_total_amount and the Aggregate Balance failure on SUM(total_amount), an over-applied discount is a plausible cause worth catching explicitly.

```
SELECT
    s.sale_id,
    s.product_id,
    p.product_name,
    s.discount_amount,
    p.max_discount,
    s.total_amount,
    s.adjusted_total_amount
FROM demo.f_ebike_sales s
JOIN demo.d_ebike_products p ON s.product_id = p.product_id
WHERE s.discount_amount > p.max_discount
```

  This is a Custom SQL Test -- it returns one row per violation. Zero rows means the test passes.

  Why this one over the alternatives:

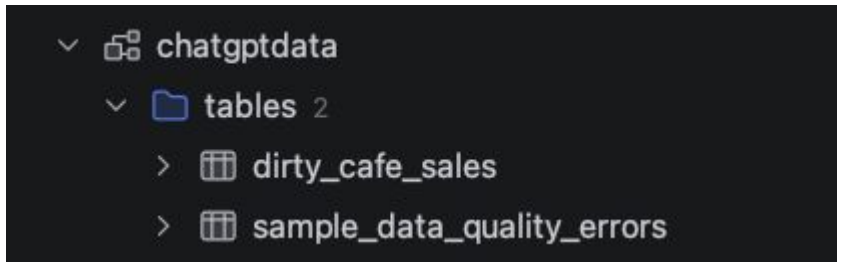
  - f_ebike_returns has no tests at all, but I'd need to see its schema before writing meaningful rules.
  - A total_amount = sale_price * quantity_sold - discount_amount math check is also good, but the existing Aggregate Balance test already approximates that.
  - The discount cap is the only test that would catch a real pipeline error -- a discount rate applied at load time that exceeds what the product allows -- and it is impossible to express with any standard test type.

  Want me to add it to the suite?

Demo

Recommend and Add New Tests

DataKitchen



DataKitchen

Create a Database Connection named: chatgptdata-connection …

Then
Create a table group named: chatgptdata-tablegroup
With schema: chatgptdata
From the Database Connection named chatgptdata-connection

Then
Profile the data in the chatgptdata-tablegroup

Then
Summarize the results

Then
Create a test suite: chatgptdata-testsuite
From the table group named: chatgptdata-tablegroup

Then
Run the tests in chatgptdata-testsuite

Then
Summarize the results

Then
Review the test results and the hygiene issue; give me a list of recommended tests to add to the test suite

Then
Create the test and add a note on the test itself you added saying 'created from claude'

Recommended tests to add

These fill the gaps between what the hygiene issues found and what the auto-generated suite currently checks. Grouped by table.

dirty_cafe_sales

1. Numeric format check on `Price Per Unit`, `Total Spent`, and `quantity`. All three are stored as text but roughly 96 percent numeric. Add a pattern or regex test so the non-numeric junk (the "ERROR" and "UNKNOWN" strings) gets caught and counted.
2. Date format check on `Transaction Date`. Stored as text, 96.9 percent valid dates, at least three competing patterns. A format test flags the remaining bad strings.
3. Dummy value check on `Payment Method`, `location`, `item`, `quantity`, `Price Per Unit`, `Total Spent`, and `Transaction Date`. A "value should not be in list" test against your known placeholder tokens closes the completeness blind spot described above.
4. Positive value range on `quantity` and `Price Per Unit`. Both should be greater than zero once cast.
5. Cross column reconciliation: `Total Spent` should equal `quantity` times `Price Per Unit`. A custom test here catches arithmetic and data entry errors that no single column test will find.
6. Value in set on `Payment Method` and `location`. Constrain each to its real list of allowed categories.

sample_data_quality_errors

7. Range check on `age`. Constrain to a plausible band such as 0 to 120. Only unique and required are checked today.
8. Non-negative range on `Annual Income`.
9. Past date check on `Date of Birth`. Reject future dates and implausibly old ones.
10. Phone format check on `Phone Number`. A pattern test standardizes what is currently only length and missing checks.
11. Value in set on `gender`. Constrain to your allowed codes to catch the 202 non-standard blanks profiling found.

Both tables, operational

12. Row count and freshness tests once these become live feeds rather than static samples. Volume drift and stale loads are the failures that column tests never surface.

Multi-Select

TABLE	COLUMN / FOCUS	TEST TYPE	ACTIVE	LOCKED	URGENCY	FLAGGED	NOTES	BASED ON PROFILING	LAST MANUAL UPDATE	OBSERVABILITY
dirty_cafe_sales	Price Per Unit	Custom Test	✓		Fail		1		2026-07-14 09:28:39	Inherited (No)
dirty_cafe_sales	Price Per Unit	Custom Condition	✓		Fail				2026-07-14 09:19:48	Inherited (No)
dirty_cafe_sales	Price Per Unit	Value Match	✓		Fail			2026-07-14 08:58:24		Inherited (No)
dirty_cafe_sales	Price Per Unit	Percent Missing	✓		Warning			2026-07-14 08:58:24		Inherited (No)
dirty_cafe_sales	quantity	Custom Test	✓		Fail		1		2026-07-14 09:28:50	Inherited (No)
dirty_cafe_sales	quantity	Custom Condition	✓		Fail				2026-07-14 09:20:25	Inherited (No)
dirty_cafe_sales	quantity	Custom Condition	✓		Fail				2026-07-14 09:20:52	Inherited (No)
dirty_cafe_sales	quantity	Value Match	✓		Fail			2026-07-14 08:58:24		Inherited (No)
dirty_cafe_sales	quantity	Percent Missing	✓		Warning			2026-07-14 08:58:24		Inherited (No)
dirty_cafe_sales	Total Spent	Alpha Truncation	✓		Fail			2026-07-14 08:58:24		Inherited (No)
dirty_cafe_sales	Total Spent	Custom Test	✓		Fail		1		2026-07-14 09:29:26	Inherited (No)
dirty_cafe_sales	Total Spent	Custom Test	✓		Fail		1		2026-07-14 09:28:45	Inherited (No)

Rows per page: 500

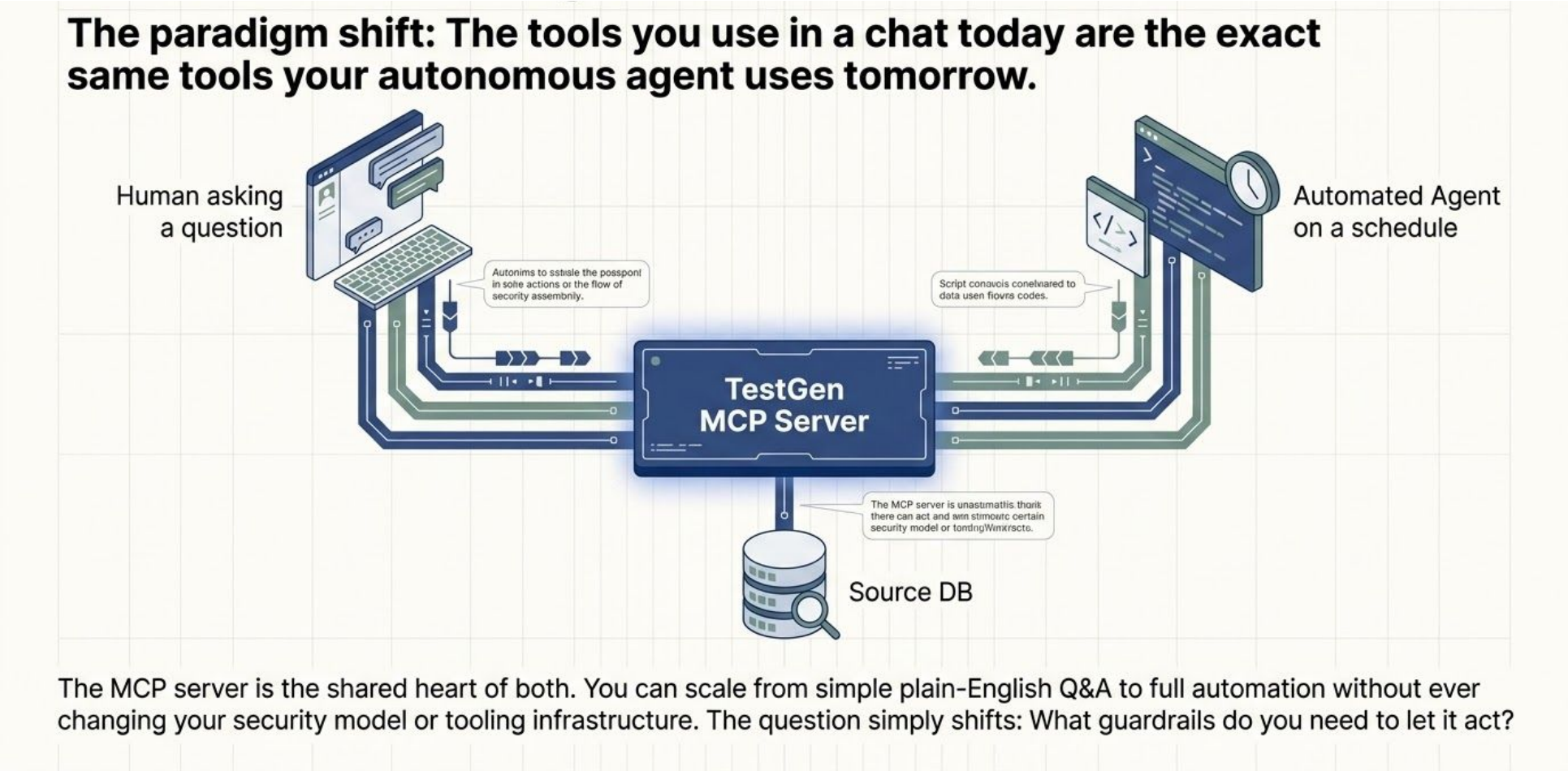
Agenda

1. What is MCP and why does it matter?
2. Intro To Testgen MCP Server
3. Testgen UI vs. MCP
4. Testgen MCP Server As An AI Dialog-Based UI
5. **Testgen MCP As The Foundation For AI Agents**
6. Conclusion

DataKitchen

From chat to delegation

An agent is a program that acts on your behalf

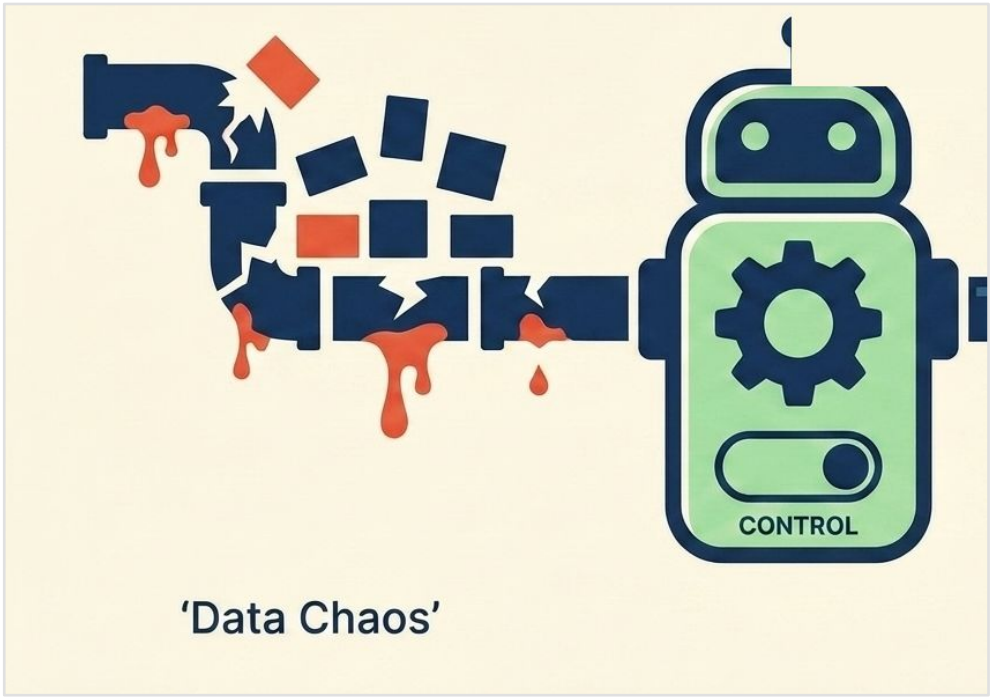


DataKitchen

Agents that fix data quality, with control

A design pattern: the fix is easy, the control is the hard part

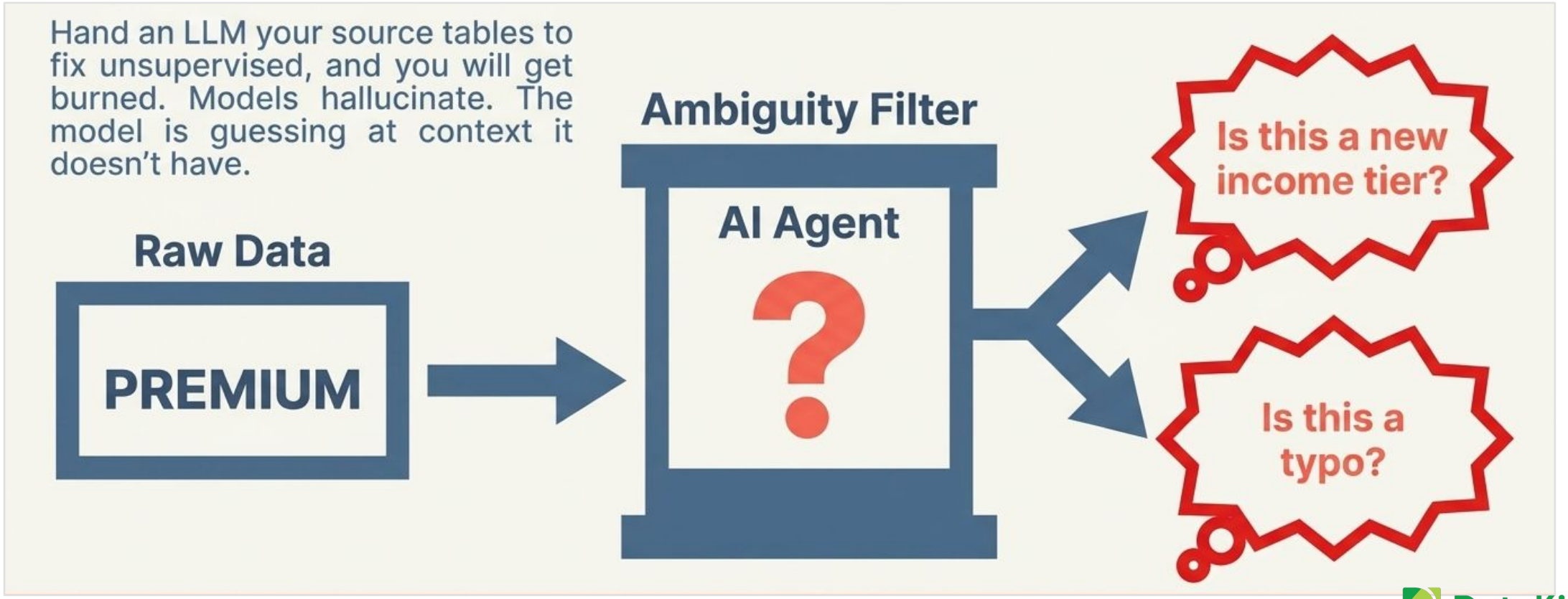
- You don't want an agent rewriting your source tables on its own at 2 am
- You want it to find the problem, propose the exact SQL, and wait for a human to say "yes, fix that one"
- Two small Python programs: one finds problems and proposes fixes, one applies only what you approved



DataKitchen

Why "just automate it" is a trap

On source data, a wrong fix is expensive. You don't notice until the dashboard is off and the bad write already ran.



DataKitchen

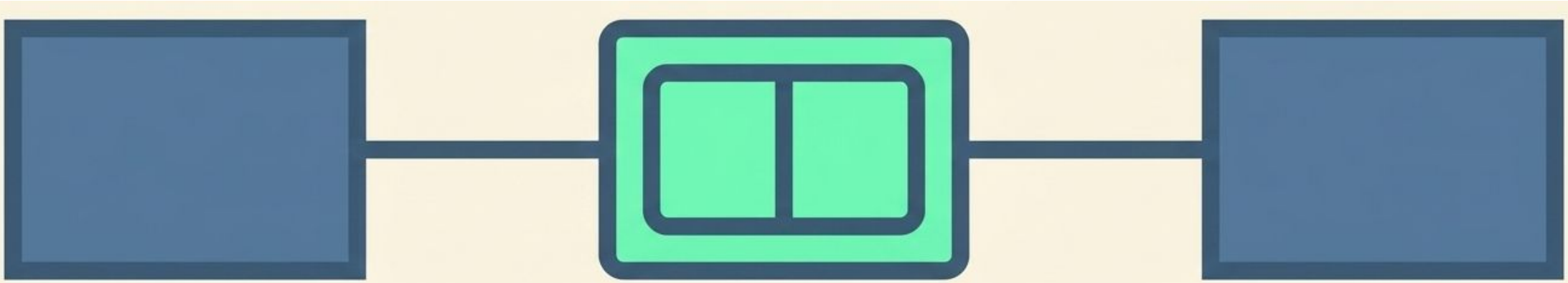
The problem is the judgment, not the SQL

You save time by handing off the boring work and keeping the one decision that matters

DataKitchen

The new paradigm: the human-gated agent

The agent proposes. You approve. The agent applies.



Step 1: The Discovery Engine

The agent reads the source schema, finds hygiene issues, and writes the exact SQL to fix them.

Step 2: The Human Gate

The agent stops and files a Jira ticket. It waits for the human to review the plain-language findings.

Step 3: The Execution Engine

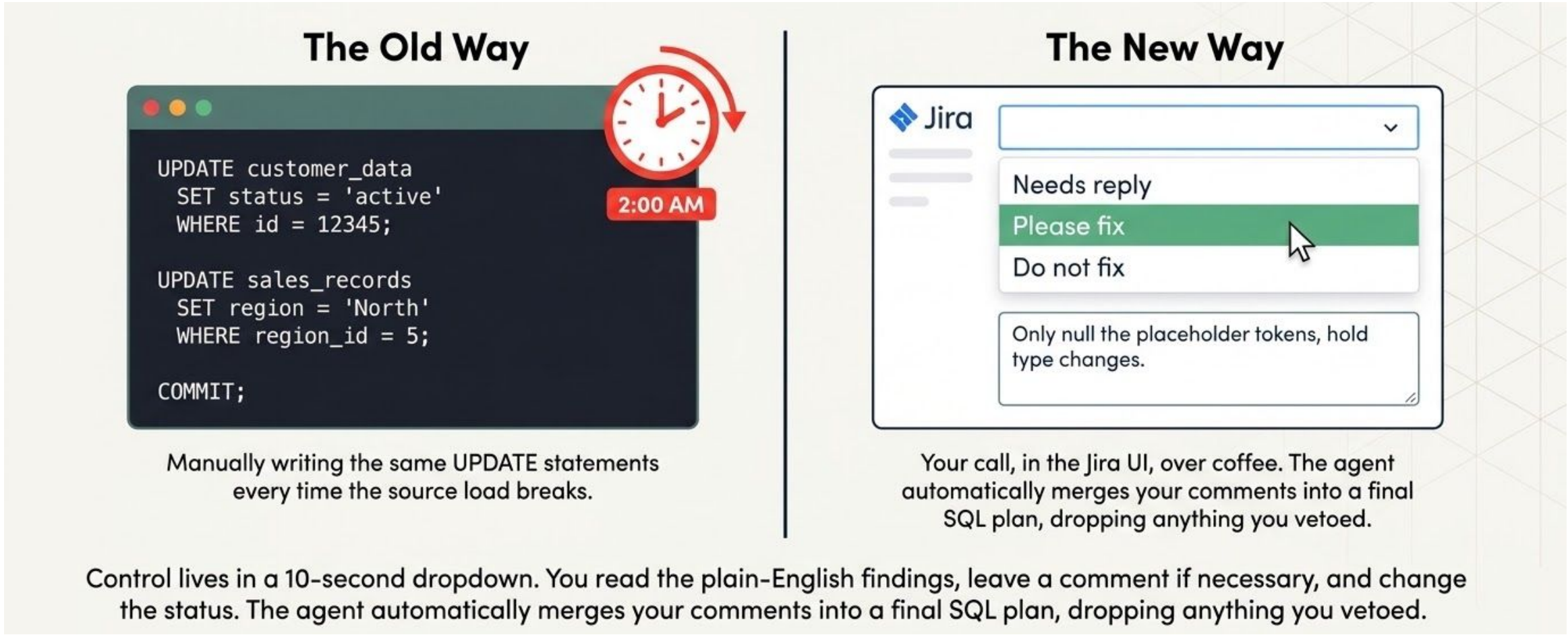
Once approved, the agent safely runs the SQL and logs exactly what it did.

The discovery engine is Claude calling the TestGen MCP server. No glue code, no scraping a UI.

DataKitchen

Coffee-break data quality

The 2 am UPDATE becomes a dropdown: Please Fix, Needs Reply, or Do Not Fix



DataKitchen

The data quality resolution

Speed of automation, and you keep the veto

	Speed & Scalability	Safety & Accuracy	Human Effort
Manual Coding	Slow / Low	High	Exhausting
Unsupervised AI	Lightning Fast	Unpredictable / High Risk	Zero
Human-Gated Agent	**Fast**	**100% Controlled**	**Minimal / 10 Seconds**

DataKitchen

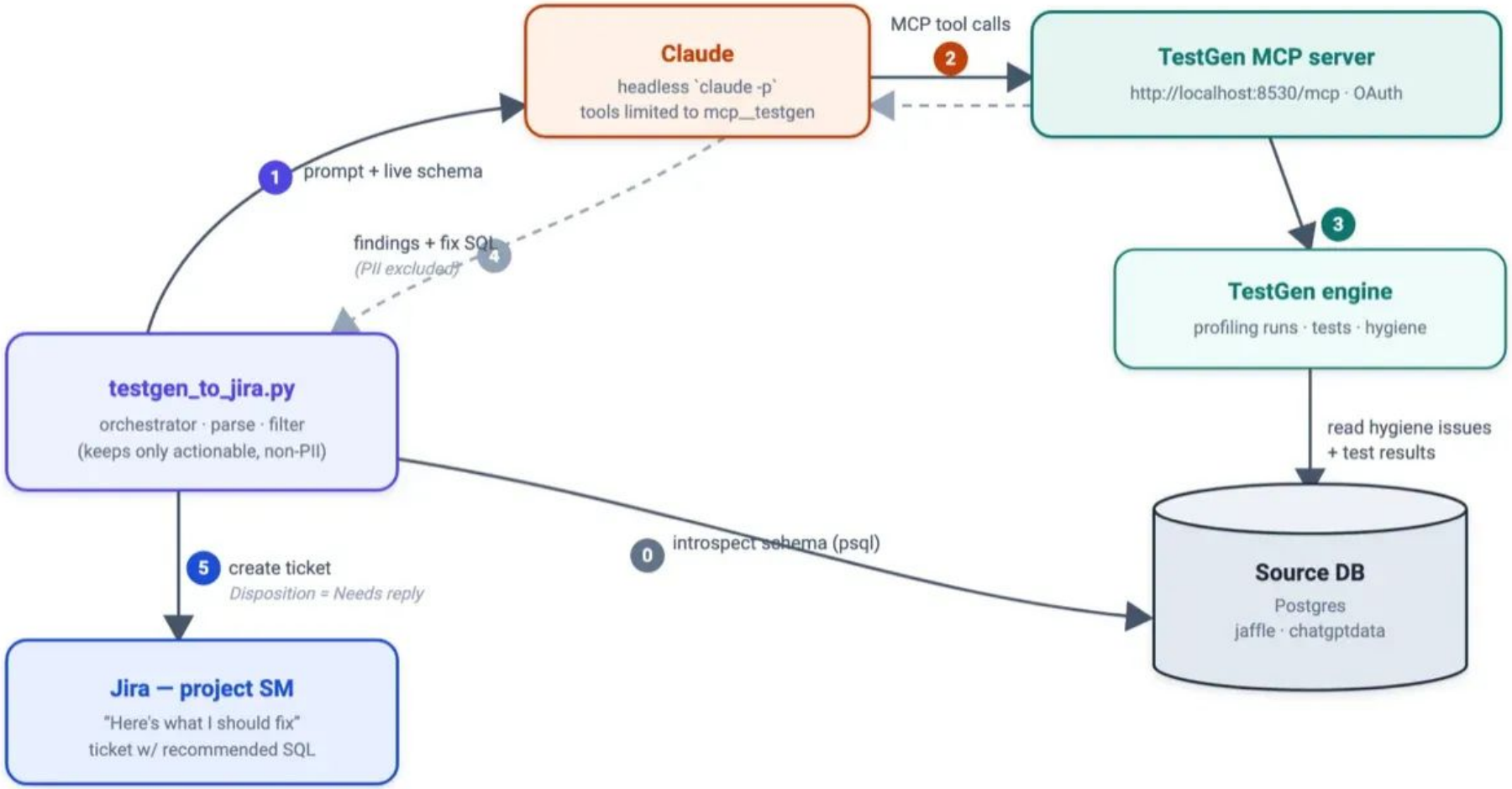
Example

The human-gated agent loop

- The agent files a Jira ticket with proposed SQL
- Flip the disposition flag to Please Fix
- The fix applies on a copy table, and the ticket closes itself

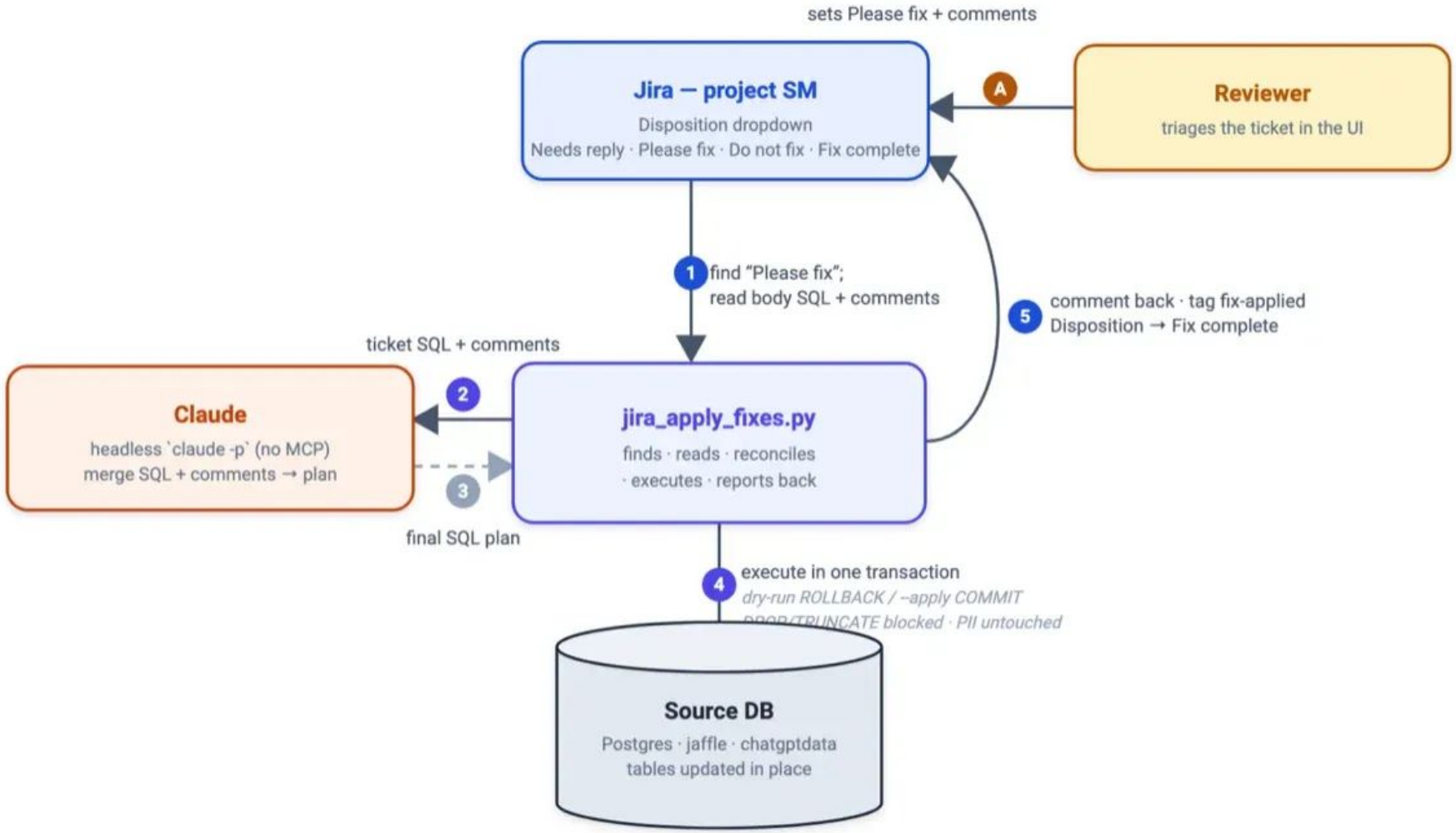
DataKitchen

Demo - Find the list of data issues



DataKitchen

Demo – fix the list of data issues



DataKitchen

Automation on your terms

Your sources will keep sending bad data. Now you fix it once, on your terms, and the agent handles the rest.

- Benefits
  - You get the speed of automation, and you keep the veto
  - Bad data stops being a recurring chore and becomes a ticket you approve over coffee
  - The MCP server gives the agent eyes, hands, and memory. What you build on top is the policy.

If you want to recreate this in your own environment, we wrote a PRD you can load straight into Claude Code.

- It names the two programs, the contract between them, the safety gates, and the config — enough for Claude to rebuild the pattern against your stack.
- Download the PRD and load it into Claude Code to build the human-gated data quality fix agent in your own environment.
- blog: https://datakitchen.io/blog/agents-fix-data-quality-automatically-with-control/

DataKitchen

Agenda

1. What is MCP and why does it matter?
2. Intro To Testgen MCP Server
3. Testgen UI vs. MCP
4. Testgen MCP Server As An AI Dialog-Based UI
5. Testgen MCP As The Foundation For AI Agents
6. **Conclusion**

DataKitchen

Conclusion

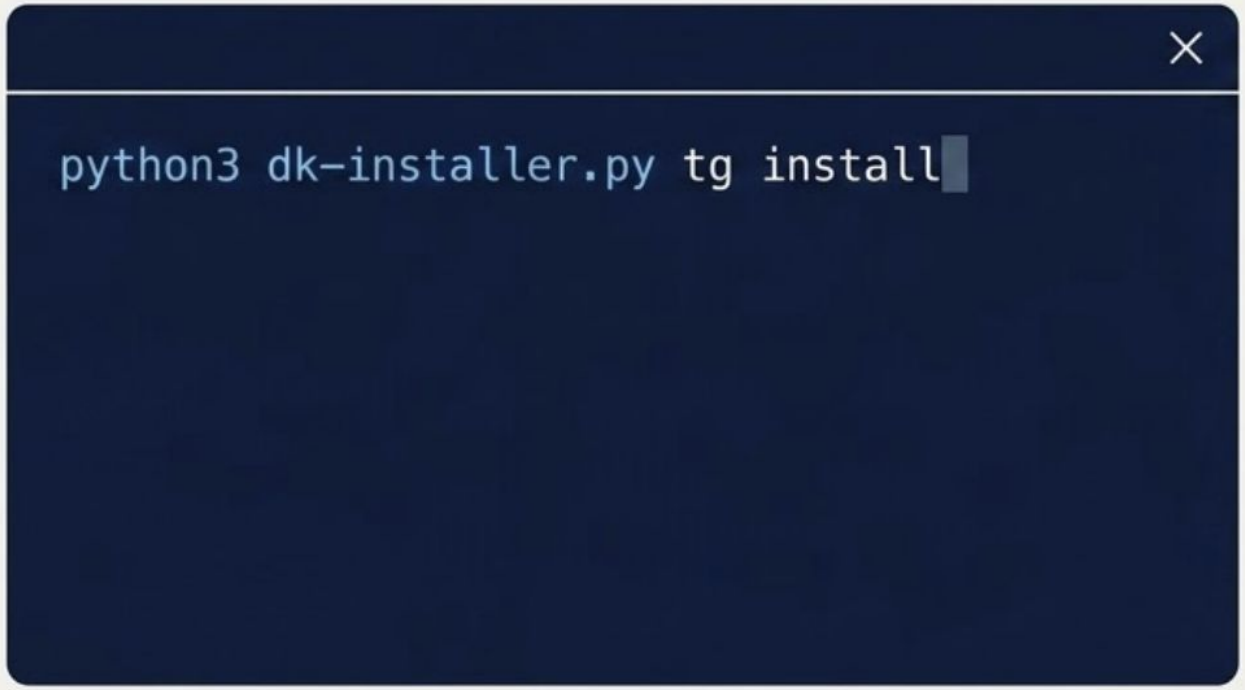
1. **What fast, direct answers?** **Use the TestGen UI**
2. **Want to:** **Use TestGen MCP in an LLM**
   a. Diagnose Root Causes?
   b. Update source data quality, make changes to data?
   c. Improve your test suites by adding new tests?
3. **Want to Automatically, repeatedly:** **Create An Agent (with control)**
   a. Fix data,
   b. add tests,
   c. automate data quality?

DataKitchen

DataKitchen

Build it tonight

Install, connect, and ask what's wrong with your data. It will tell you.
That's the uncomfortable part. Twenty minutes, no sales call.



Open Source

Available under Apache 2.0 at
github.com/DataKitchen/dataops-testgen

TestGen MCP Cheat Sheet

The 96-tool 1-pager.

Human-Gated Agent PRD

Load our Product Requirements Document straight into Claude Code to build the pattern in your own environment.

DataKitchen