EFTEDRA

ARCHITECTURE WHITEPAPER

Self-Service Reporting Without Data Exposure

How to let business users configure their own reports with a model assistant that never sees a single row of your data — and cannot reach one even if it tries.

EFTEDRA — Smart Workflow Solutions
Version 1.0 · 2026
eftedra.com

Contents

1. The reporting backlog

2. Why the obvious approach fails

3. Principle one: schema-only context

4. Principle two: configuration as output

5. Principle three: a deliberately restricted surface

6. Validation and execution

7. What the business user actually experiences

8. Threat model

9. Implementation checklist

1. The reporting backlog

> Every organisation running a workflow platform at scale accumulates the same queue. A business-unit manager wants to see something the standard screens do not show — how instances are progressing, which steps are ageing, how volumes break down by a dimension nobody anticipated. It is a small request. It becomes a ticket, a developer, a test cycle and a release, and it arrives six weeks later, by which point the question has moved on.

The requests are individually trivial and collectively enormous. They are also unavoidable: nobody can enumerate in advance every view of the data an organisation will need, and the ones that matter most are usually the ones discovered in response to something going wrong.

The obvious answer is self-service. Let the manager configure the report. The reason this has historically failed is not usability — it is that giving a business user a query builder against production data is a data-governance problem wearing a friendly interface. A model assistant makes the usability problem tractable for the first time. It does nothing on its own about the governance problem, and if deployed naively it makes it considerably worse.

2. Why the obvious approach fails

The naive implementation is to describe the database to a model, let the user ask in plain language, and execute whatever query comes back. It demonstrates beautifully. Through 2026 the industry documented, in detail, why it should not go to production.

The query that is valid and still wrong

Generated SQL can be syntactically perfect, execute without error, and return data the requester was never entitled to see. Nothing in the generation step knows about entitlement, so nothing in it enforces entitlement.

The confused deputy

The most serious structural flaw. The agent connects with its own credentials, which must be broad enough to serve every user. It therefore acts with more authority than the person asking, and the user's own permissions are silently bypassed — not through a bug, but through the design. A junior analyst asks a question and receives an answer computed with privileges they do not hold.

Destructive and unbounded statements

A model can be manipulated, or can simply err, into emitting a destructive statement or an unbounded dump of a table. Prompt injection reaches this directly: content inside the data itself can carry instructions.

Data in the context window

Approaches that improve accuracy by including sample rows put production data into a third-party API call, into logs, and into whatever retention applies. This is frequently the single fact that ends the procurement conversation, and it is usually introduced for convenience rather than necessity.

> **The signal worth noting**
>
> Anthropic publishes explicit guidance on securing text-to-SQL, treating privilege escalation, unsafe statement generation and injection as the default risks to design against. When a model vendor ships framework-level guidance on a pattern, the pattern is common enough — and dangerous enough — to warrant an architecture rather than a prompt.

3. Principle one: schema-only context

The first rule is absolute and it is what everything else rests on: **the model receives a description of the data structures. It never receives data.**

Concretely, the context contains table and view names, column names and types, relationships, and human-readable semantics — what a column means, its unit, whether it is a code from a controlled vocabulary. It does not contain rows. Not real rows, not sampled rows, not anonymised rows.

This is a stronger guarantee than anonymisation, and it is stronger for a reason worth stating plainly: anonymisation is a probabilistic control that can fail through re-identification, and it requires ongoing judgement about what is identifying. Sending no rows at all is a categorical control. It cannot fail quietly, and it does not require anyone to be right about what was sensitive.

The practical objection is that accuracy suffers without examples. In our experience this is much less true than expected, because the task is not open-ended analysis — it is selecting dimensions, measures and filters from a described, bounded surface. Where value semantics genuinely matter, the answer is to describe the vocabulary in the schema (*this column contains one of: OPEN, PENDING, CLOSED*) rather than to show rows. That is metadata, and it is authored deliberately by someone who knows what may be disclosed.

4. Principle two: configuration as output

The second rule: **the model emits configuration, not code.**

Its output is a JSON document conforming to a published schema — which view to draw from, which columns as dimensions, which as measures, which filters, how to sort, how to visualise. That configuration is what drives the application. Where a query is required it is scoped, read-only, and generated against the same restricted surface.

This collapses the risk surface for reasons that compound:

- **Schema-validated.** Output either conforms or is rejected. There is no partially-valid configuration.
- **Bounded by construction.** Configuration cannot express arbitrary logic because the schema does not admit any. Everything expressible was designed in advance by an engineer; the model chooses among sanctioned options.
- **Inspectable by non-engineers.** A manager can read a configuration and recognise whether it describes the report they asked for. They cannot do that with SQL, which means with SQL they approve without understanding.
- **Diffable and reversible.** Configurations version, compare and roll back like any other structured artefact.

The generalisation is worth carrying beyond reporting: *whenever you are tempted to have a model produce code, ask whether the thing you actually need is configuration.* It usually is, and the difference in what can go wrong is very large.

5. Principle three: a deliberately restricted surface

The third rule: **the model is only ever told about pre-approved, sanitised views.**

Not base tables. Purpose-built views that encode, at the database layer, the filters and masking the organisation requires. Columns that must not leave the system are not in the view, which means they are not in the described schema, which means they cannot be selected by a query written against a surface that does not know they exist.

Three properties follow:

Property	Why it matters
Absence, not refusal	The restricted column is not something the model declines to select. It is something the model has never heard of. There is no prompt to be talked out of.
Enforced at the data layer	The view exists in the database and is owned by whoever owns the data, not by the application. Application bugs cannot widen it.
Reviewable by the data owner	"Which views may the assistant see?" is a question with a finite, auditable answer that a data steward can approve without reading any application code.

Execution then uses a read-only principal whose grants are limited to precisely those views. This is the belt to the schema's braces: even a query that somehow named a forbidden object would fail at the database, because the grant does not exist. The safety property does not depend on the model behaving well.

6. Validation and execution

Everything returned is treated as untrusted input, because it is.

Configuration validation

Validate against the JSON Schema. Then check semantics the schema cannot express: that referenced columns exist in the named view, that aggregations are legal for their column types, that filters reference described columns, that the requested visualisation suits the shape of the result.

Query validation

Any generated statement is parsed — parsed, not pattern-matched — before execution, and rejected unless it is read-only, references only whitelisted views, carries an enforced row limit, and contains no unbounded joins or nested statements outside the permitted set. String matching for dangerous keywords is not sufficient and should not be relied upon.

Execution limits

Statement timeout, maximum returned rows, and a resource-limited connection pool. These protect availability rather than confidentiality, and they matter because a self-service feature is by definition used by people who cannot estimate the cost of what they are asking for.

Human publication

Nothing becomes a saved report without a person seeing it and choosing to keep it. The approval is recorded with the configuration. This is not a formality — it is the control that makes the whole feature defensible, because every artefact in the system was knowingly accepted by a named human.

7. What the business user actually experiences

None of the above is visible to them, which is the point.

They describe what they want to see — *show me open cases by team and how long they have been waiting, for this quarter*. A draft report appears. They adjust it in plain language or by direct manipulation. When it is right, they save it, and it becomes part of their dashboard.

What they never encounter is a query language, a schema browser, a ticket, or a six-week wait. What the organisation never encounters is a business user with ad-hoc access to production data, or a model with a database connection.

> **The honest limitation**
>
> This architecture does not let a business user ask arbitrary questions of arbitrary data. It lets them compose reports from a surface someone deliberately exposed. That boundary is a feature — but it means the work of defining and maintaining good views does not disappear. It moves to the data owner, where it belongs, and it is done once per surface rather than once per request.

8. Threat model

Four scenarios worth walking, because they are the ones that get asked.

Scenario	Outcome
A user tries to prompt the assistant into revealing salary data	The salary column is not in any described view. The model cannot name what it has not been told about; if it hallucinated the name, validation rejects the reference and the grant would refuse it anyway.
Prompt injection arrives inside the data	The model never reads data, so the injection surface does not exist. This is a side effect of the schema-only rule and one of its strongest arguments.
The model emits a destructive statement	Rejected at parse time as non-read-only; the read-only grant would refuse it regardless. Two independent controls, either sufficient.
The model is compromised or behaves unexpectedly	Its output is a schema-validated configuration document. The worst achievable outcome is a badly composed report over already-approved views — visible to the user before they save it.

The pattern across all four: no control depends on the model's cooperation. Each is a property of what the model can reach, and each is enforced somewhere the model does not run.

9. Implementation checklist

- Define the reporting views explicitly. Have the data owner approve the list.
- Author schema descriptions as deliberate metadata, reviewed for disclosure — column semantics and controlled vocabularies included.
- Confirm, by inspecting an actual outbound request, that no row data is present.
- Publish a JSON Schema for report configuration and validate every output against it.
- Create a read-only database principal granted only on the approved views.
- Parse and validate generated statements; never rely on keyword matching.
- Enforce statement timeouts and row limits.
- Require explicit human publication and record who approved what.
- Log every configuration produced, accepted and rejected — this is both the audit trail and your evaluation set.
- Define behaviour when the model is unavailable: existing saved reports must keep working, since only *authoring* depends on it.

That last point is easy to overlook and matters commercially. Reports are configuration; running them is ordinary application code. If the model API is down, every saved report still runs. Only the ability to compose a new one pauses — which is a materially different conversation to have with a risk committee.

About EFTEDRA. We build workflow automation on IBM Business Automation Workflow and Claude, including self-service configuration surfaces of the kind described here. This paper describes the architecture we build to. Components described are in active development; working demonstrations are available on request.