Oracle ACE Live

Empowering Business Analysts with SQL Access in Oracle Fusion AI Agent Studio



Design-time SQL for workflow agents, parametrized tools for supervisor agents. The LLM never writes the SQL.

Alex Rabinovich, Oracle ACE Pro
CEO, CID SOFTWARE SOLUTIONS LTD
24 November 2026 | Fusion Apps 26C

About the speaker and CID



Alex Rabinovich

Oracle ACE Pro

CEO, CID SOFTWARE SOLUTIONS LTD

What CID does

Oracle Fusion Apps and OCI services. 25+ years in Oracle technology. Teams work alongside clients' own staff.

Credentials

Alex holds multiple architecture certifications.

CID is recognized as a company with five service expertise areas: Data Management, Security, Cloud Native Application, Migration and Integration.

Shared with the community

Blog posts, code, training sessions, and the free CID Fusion Query Chrome extension.

Agenda

1. Who writes the SQL decides: powerful or dangerous
2. The approach, then the build: a REST endpoint in Fusion that runs SQL
3. Design-time SQL: the analyst defines the query in the function call
4. Never LLM-written SQL: parametrized tools instead
5. Field notes: P2T cloning, quarterly updates, guardrails
6. Additional technique: wrapping Fusion REST errors so they become results

Who builds what: a developer builds the utility once (section 2). Business analysts then write the SQL and define the tools (sections 3 and 4).

What you will take away

Build	A reusable SQL data tool for AI Agent Studio: Application Composer Groovy as a Business Object Function, registered in Agent Studio
Design-time SQL	The analyst defines the query in the function call: reviewed before any run, full SQL power. The natural fit for workflow agents
Never LLM-written SQL	The BIP proxy runs any SQL: an LLM-generated query can break the security model, return wrong results and hurt performance
Parametrized tools	Pre-defined query plus named, described, typed parameters the LLM fills: the natural fit for supervisor-agent chatbots
Choose the pattern	Which pattern fits each agent type, and exactly where the line sits between powerful and dangerous
Field notes	The placeholder-to-parameter pattern, the JSON-escaping workflow, P2T cloning and quarterly Fusion updates

Powerful or dangerous: it depends on who writes the SQL

Design-time SQL

The analyst writes the full query in the Business Object function call, in a workflow node or a tool definition.

Known and reviewed before any run.

Parametrized SQL

Same fixed query, with placeholders. The LLM fills only named, described, typed parameters.

Bounded: only values vary.

LLM-generated SQL

The LLM composes the query itself.

Breaks the security model, risks wrong results and performance problems. We never do this.

Agent type is not the deciding factor: workflow and supervisor agents can both use a design-time function call.

The approach: a REST endpoint in Fusion that runs any SQL

The constraint

AI Agent Studio calls Fusion REST APIs. It cannot call SOAP services.

The gap

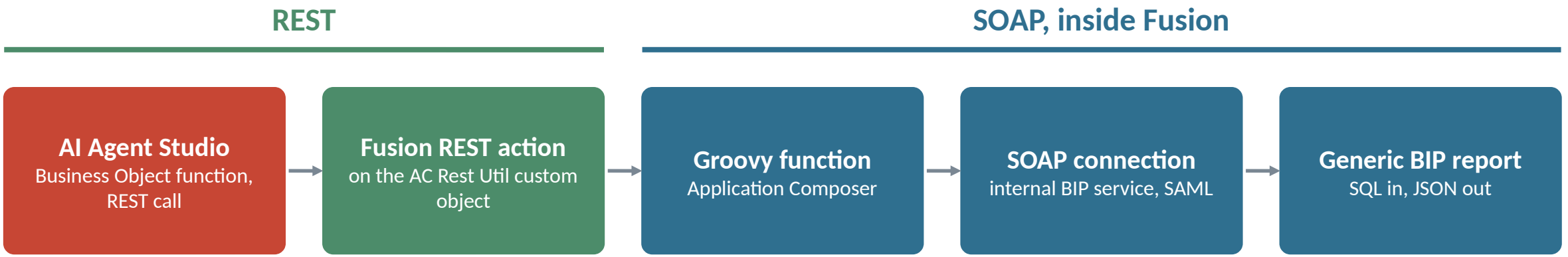
BI Publisher can run any SQL and return data, but only over SOAP. There is no REST endpoint for it.

The technique

Application Composer exposes a REST action. Its Groovy calls BIP over SOAP through a generic proxy report.

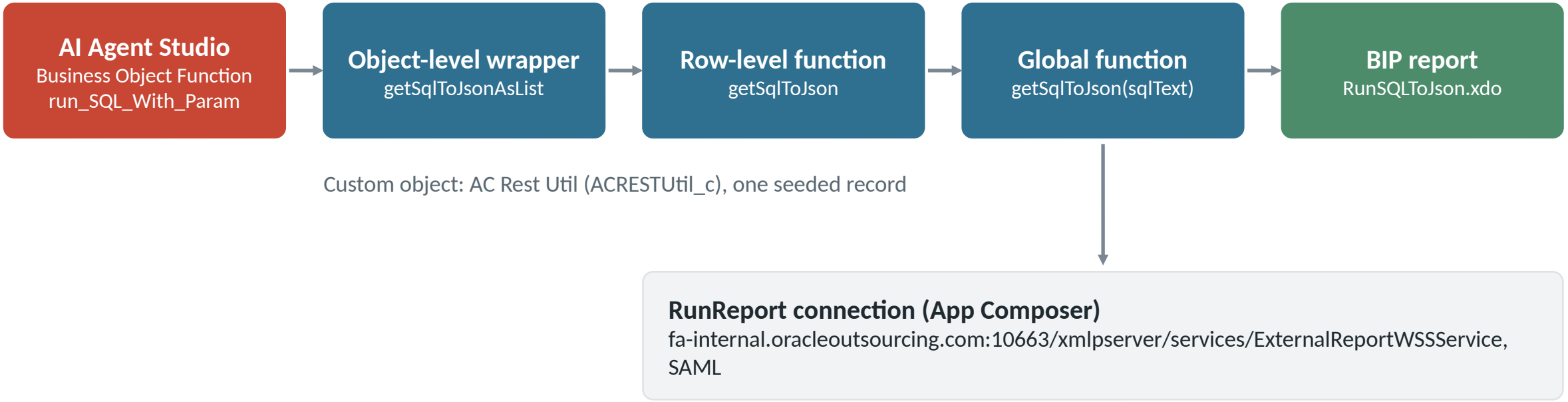
Result: a Fusion REST endpoint that receives any SQL and returns JSON. No OIC, no external middleware.

REST outside, SOAP inside, nothing leaves Fusion



- No OIC and no external middleware: nothing outside Fusion to host, secure or monitor
- The internal fa-internal URL with SAML: Production-to-Test clean, no reconfiguration after a clone
- The agent only ever sees a REST endpoint; SOAP and BIP stay hidden inside Fusion

The whole solution on one slide



AI Agent Studio sees one Business Object function: run_SQL_With_Param(sqlText). It returns JSON rows.

One Business Object function, two ways to use it

Business Object function
run_SQL_With_Param, receives sqlText as a parameter

Way 1: directly in the agent flow

SQL defined at design time
the analyst types it into sqlText in the flow node

Agent flow runs the fixed query
reviewed before any run

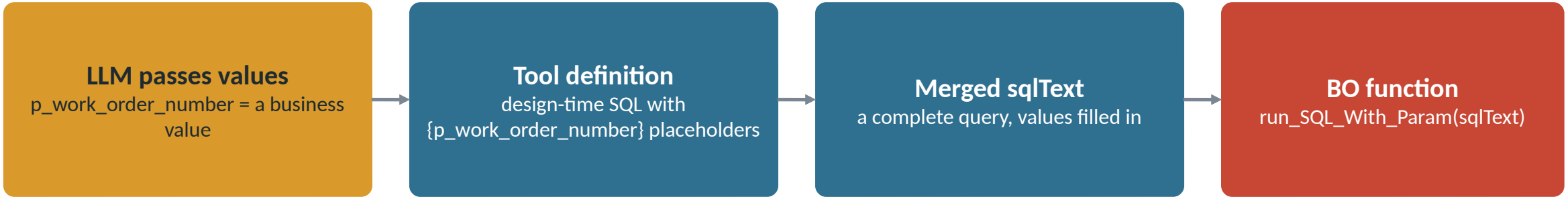
Way 2: inside a Tool

Tool exposes business parameters
subscription, work order, item… not sqlText

Tool is used later in the agent flow design
the LLM fills only the business parameters

Same function, same query engine. What changes is who supplies which value.

Way 2 in detail: from business parameters to SQL



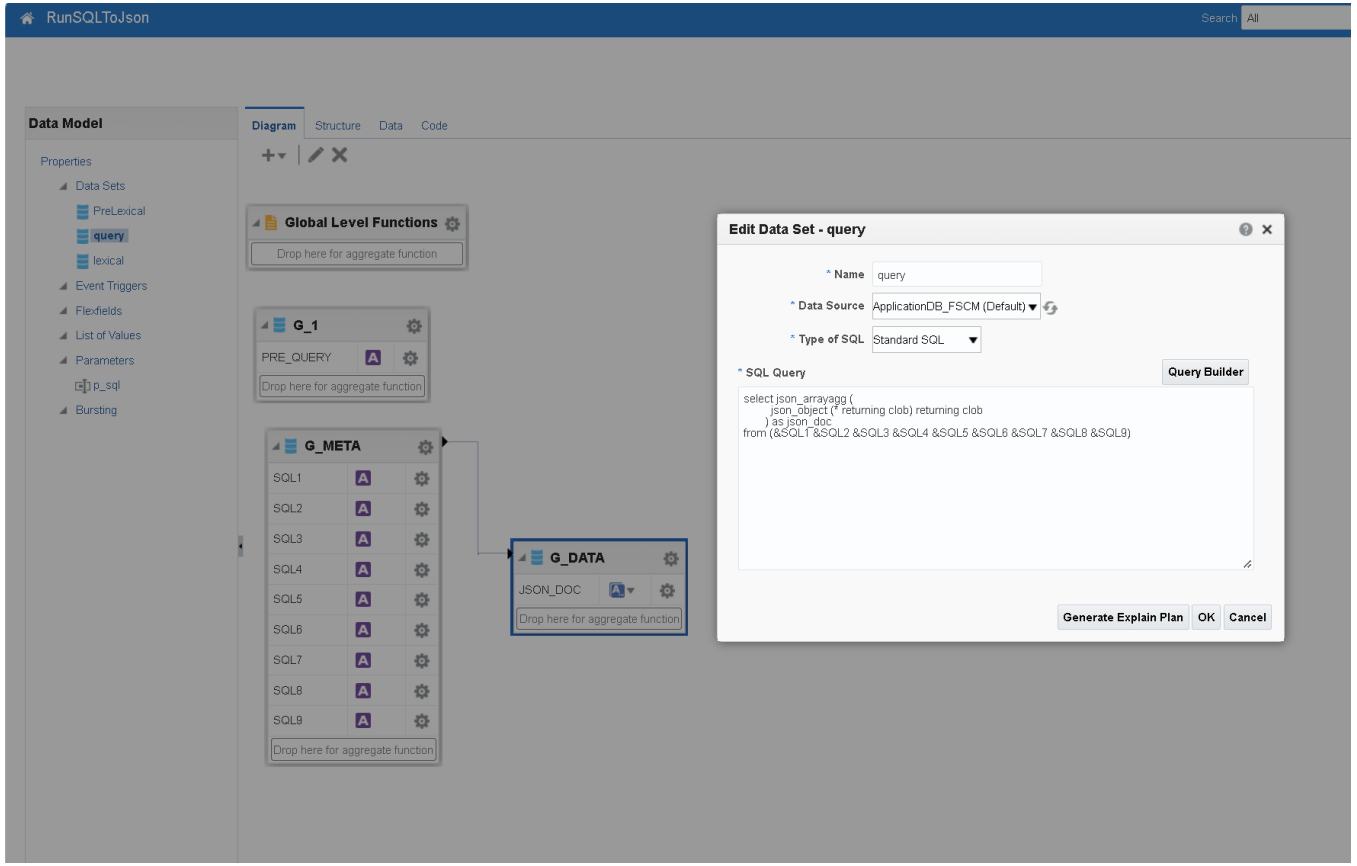
- The tool exposes business parameters with names, types and descriptions: work order, subscription, item
- The analyst defines the SQL once, with placeholders, at the tool definition level
- The LLM chooses the tool and supplies the values; the BO function receives a finished sqlText
- In a workflow node the analyst types sqlText directly; in a tool the same sqlText is built from parameters

Build the reusable SQL tool

BIP report > global Groovy function > row-level function > object-level wrapper > Business Object Function

Step 1: a generic BIP report that receives the SQL

- One parameter: p_sql, the SQL text
- Report: /Custom/Integrations/Utils/RunSQLToJson.xdo
- Data set query: lexical references &SQL1 to &SQL9 inside json_arrayagg(json_object(*))
- Output: one JSON_DOC element holding all rows
- Same technique as our free CID Fusion Query Chrome extension



Step 2: global Groovy function getSqlToJson(String sqlText)

Call the report

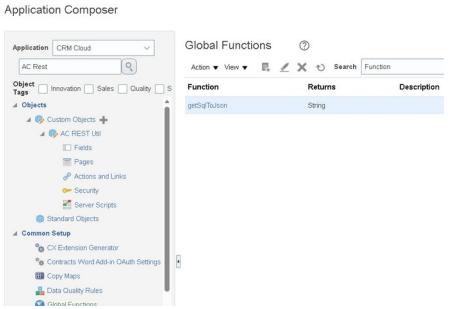
```
def reportRequest = [
  byPassCache: true,
  reportAbsolutePath:
    '/Custom/Integrations/Utils/RunSQLToJson.xdo',
  sizeOfDataChunkDownload: -1,
  attributeFormat: 'xml',
  parameterNameValues: [ item: [[
    name: 'p_sql',
    values: [ item: [[ item: sqlText ]] ]
  ]] ]
]
def report = adf.webServices.ACRunReport
              .runReport(reportRequest, '')
def repResult =
  decodeBase64(report.reportBytes.toString())
```

Extract and clean the JSON

```
def json = ''
if (contains(repResult, '</JSON_DOC>')) {
  json = substringAfter(repResult, '<JSON_DOC>')
  json = substringBefore(json, '</JSON_DOC>')
}
if (json == '') return '[]'

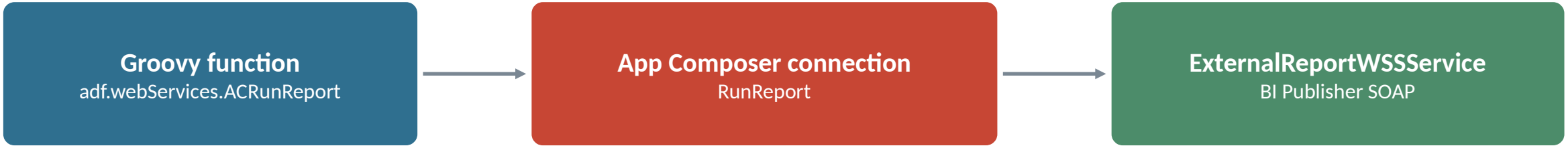
return json.replaceAll("&amp;", "&")
           .replaceAll("&lt;", "<")
           .replaceAll("&gt;", ">")
           .replaceAll("&quot;", "\"")
           .replaceAll("&apos;", "'")
```

Where it goes: Global Functions



The Step 2 code is added as the global function getSqlToJson, returning String.

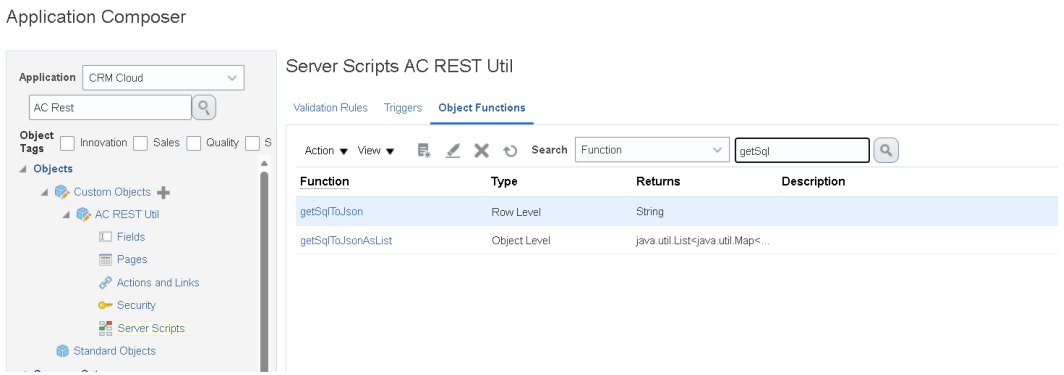
Step 3: the RunReport connection is P2T-safe




```
https://fa-internal.oracleoutsourcing.com:10663/xmlpserver/services/ExternalReportWSSService?wsdl
```

- Authentication: SAML, propagating the signed-in user identity
- Internal URL, so no pod hostname to fix after a Production-to-Test clone
- Our P2T-safe fa-internal pattern (first published November 2024), the same one we use for REST and SOAP elsewhere

Where it goes: AC REST Util, Object Functions



The call is made from the row-level function getSqlToJson on the AC REST Util custom object.

Step 4: why we need an object-level wrapper

The constraint

The function above must be called from a row-level Groovy function. An AI agent has no record in context, so it has no ID to pass.

Our fix

A thin object-level wrapper opens a view on the Rest Util custom object, takes its first record, and calls the row-level function.

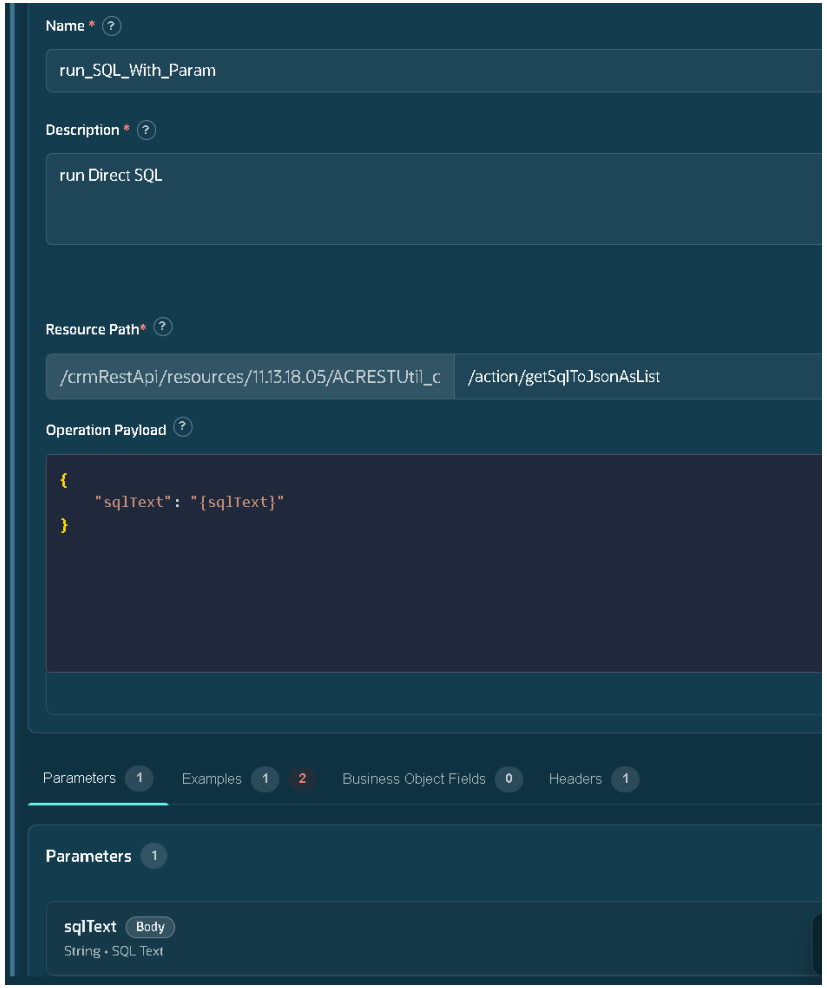


Step 5: getSqlToJsonAsList, the object-level wrapper

```
def vo = newView('ACRESTUtil_c')
def rec = vo.first()
def jsonRes = rec.getSqlToJson(sqlText)
def res = (java.util.List<java.util.Map<java.lang.String,java.lang.String>>)
            adf.util.parseJson(jsonRes)
return res
```

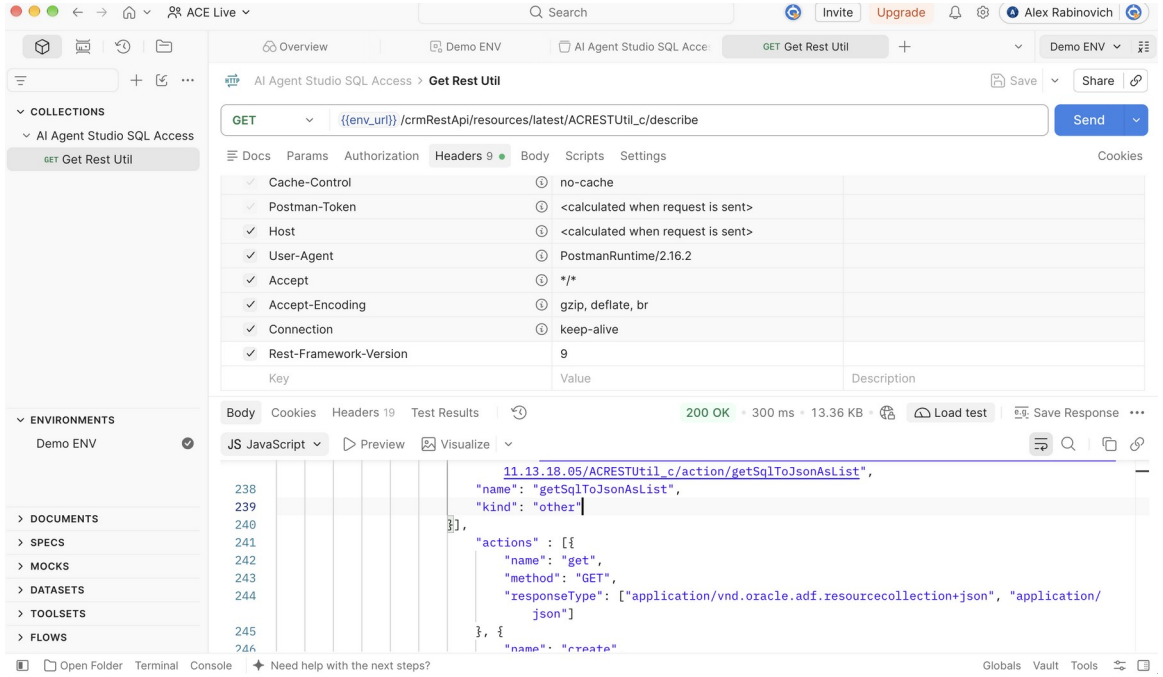
- Defined as an object-level function, input: String sqlText, visible to external systems
- getSqlToJson, defined at record level, calls the global function from Step 2
- Returns a list of maps: structured rows, not a raw JSON string

Step 6: expose it to AI Agent Studio as a Business Object Function

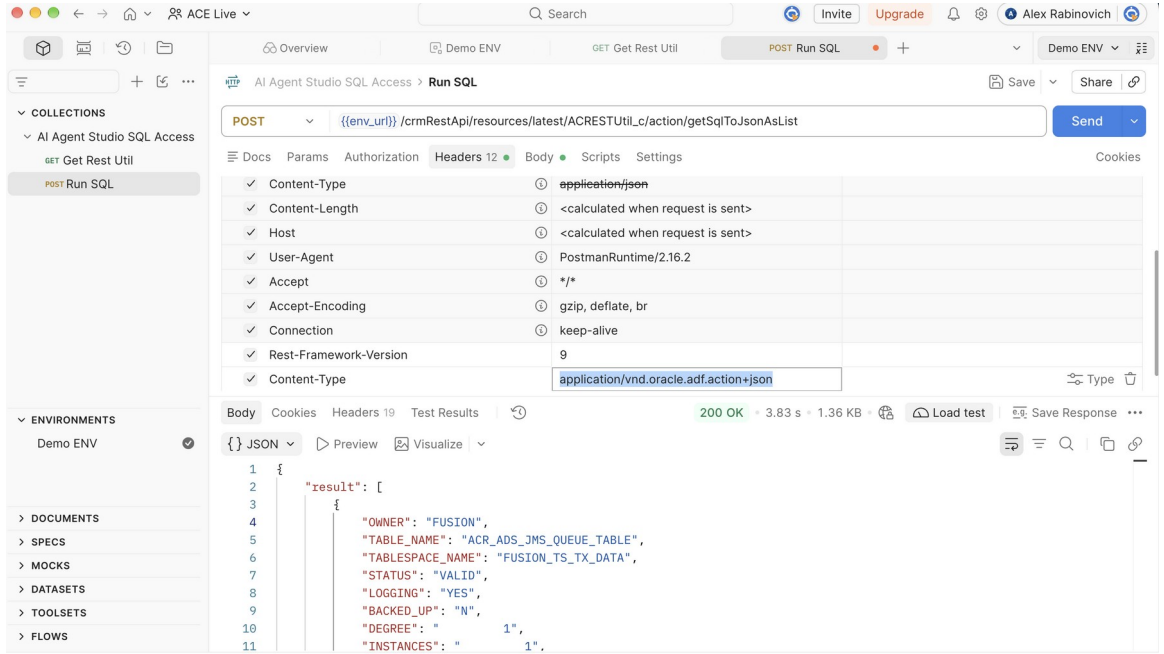


- Name: run_SQL_With_Param
- Resource path: the ACRESTUtil_c resource, action/getSqlToJsonAsList
- Operation payload: sqlText as the single body parameter
- The designer of the agent sets the query inside Agent Studio. It is not generated by the LLM: it is the business analyst's responsibility

Test the tool before the agent: Postman

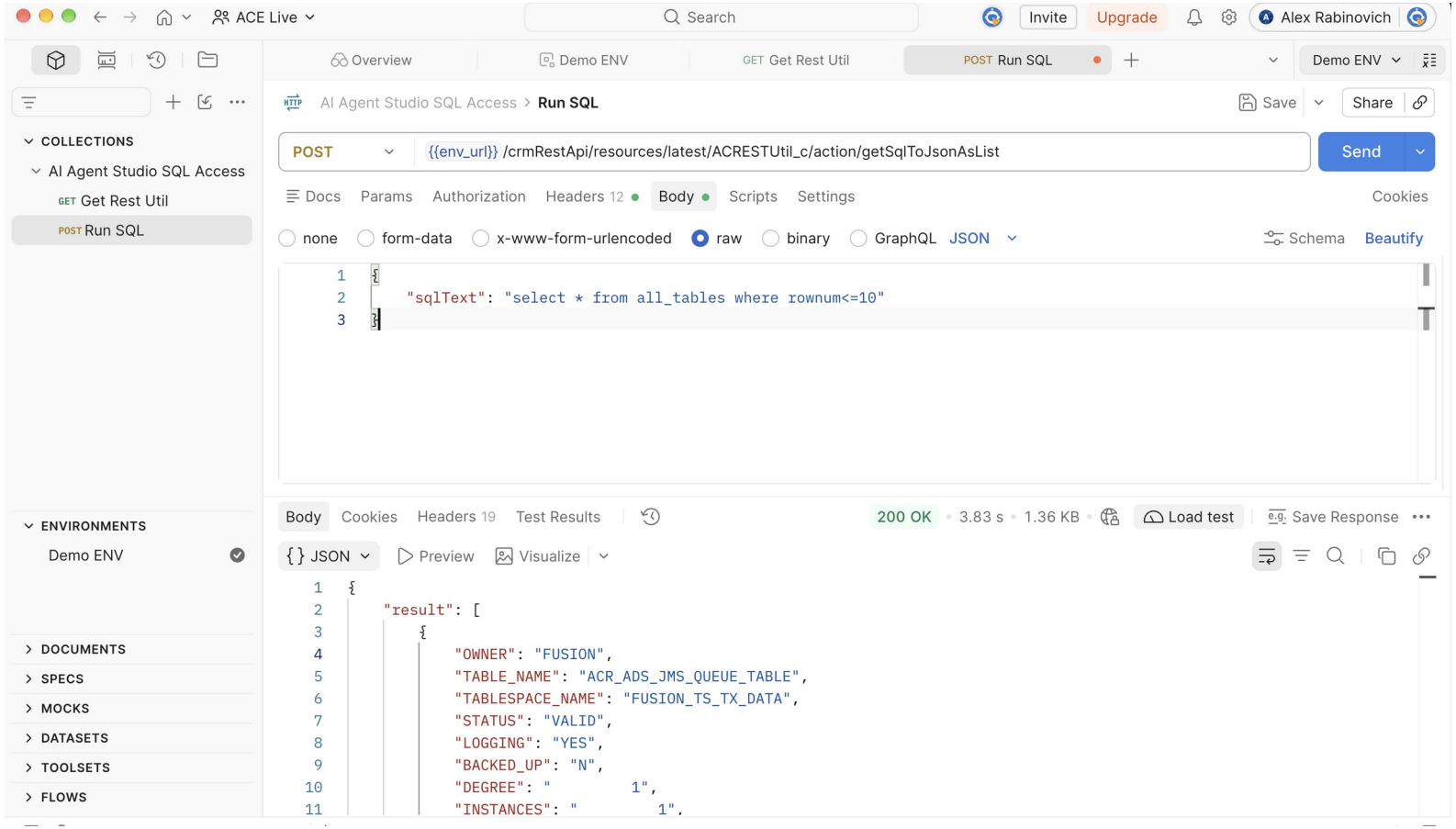


1. GET .../ACRESTUtil_c/describe: the action getSqlToJsonAsList is exposed



2. Headers: Content-Type application/vnd.oracle.adf.action+json, Rest-Framework-Version 9

Postman: run the SQL through the action

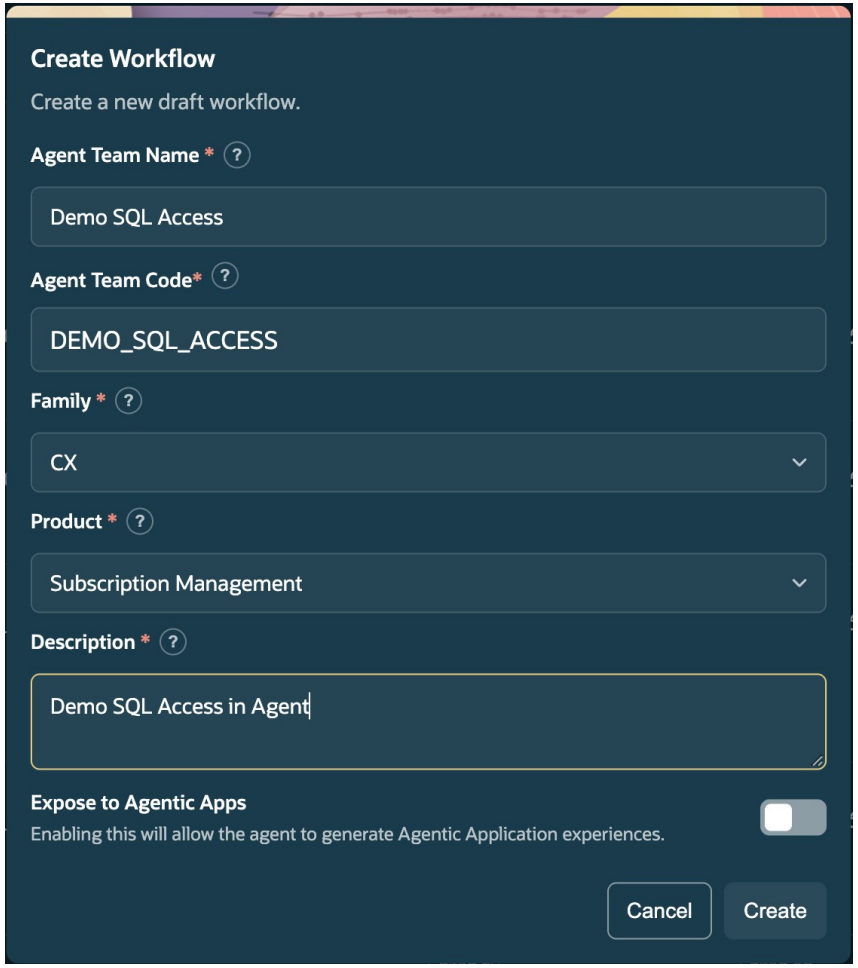


- POST …/ACRESTUtil_c/action/getSqlToJsonAsList
- Body: sqlText with the query
- 200 OK, a JSON result array
- The same call the agent makes

Design-time SQL: the analyst writes the query

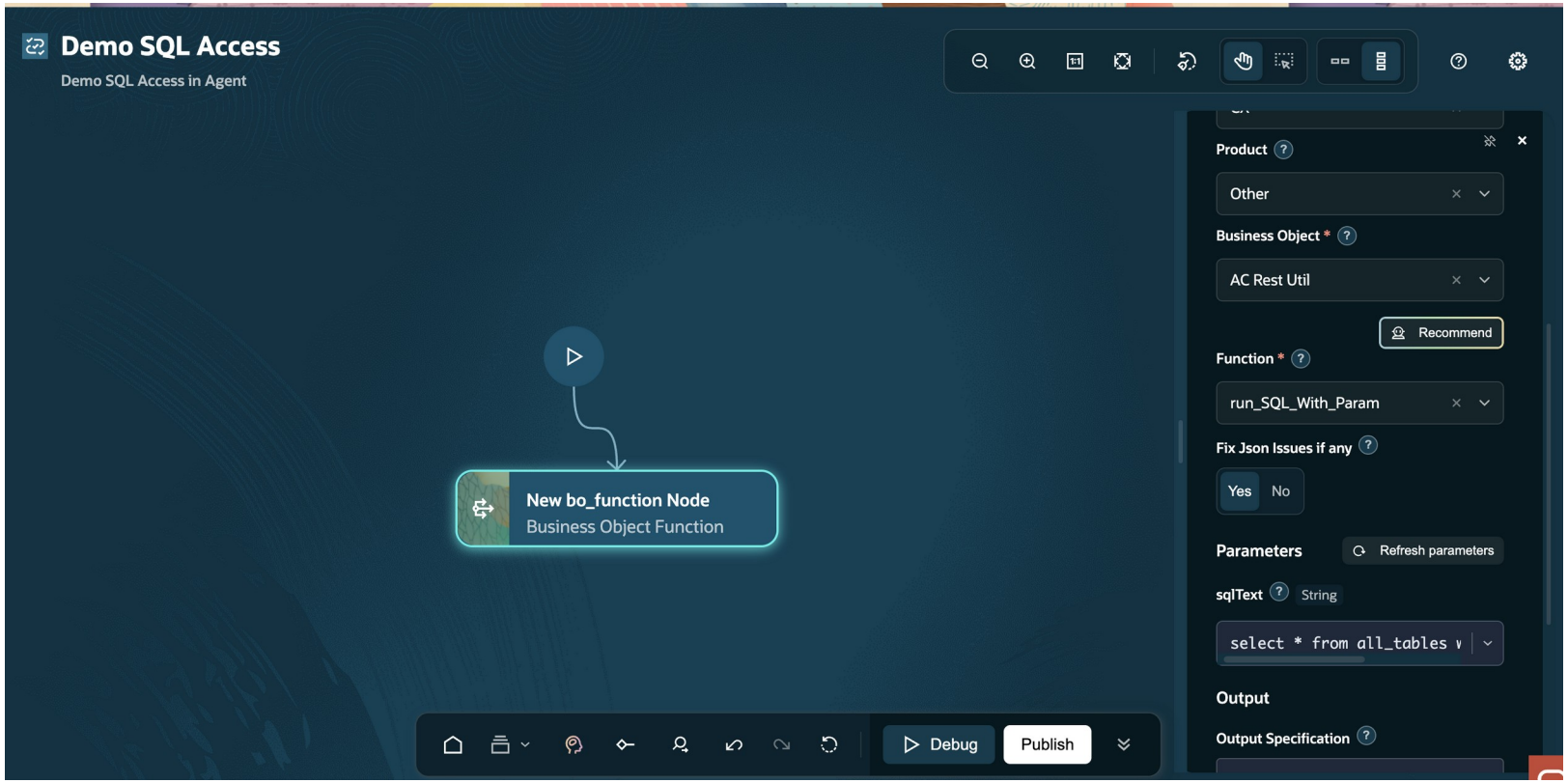
The SQL lives in the Business Object function call, defined and reviewed before any run. Example here: a workflow node; the same call works in a supervisor tool.

Design-time SQL, step 1: create the agent team



- Agent Team Name and Code: Demo SQL Access, DEMO_SQL_ACCESS
- Family CX, product Subscription Management
- Shown in a workflow agent; a supervisor tool uses the same function call
- Next: add a Business Object Function node

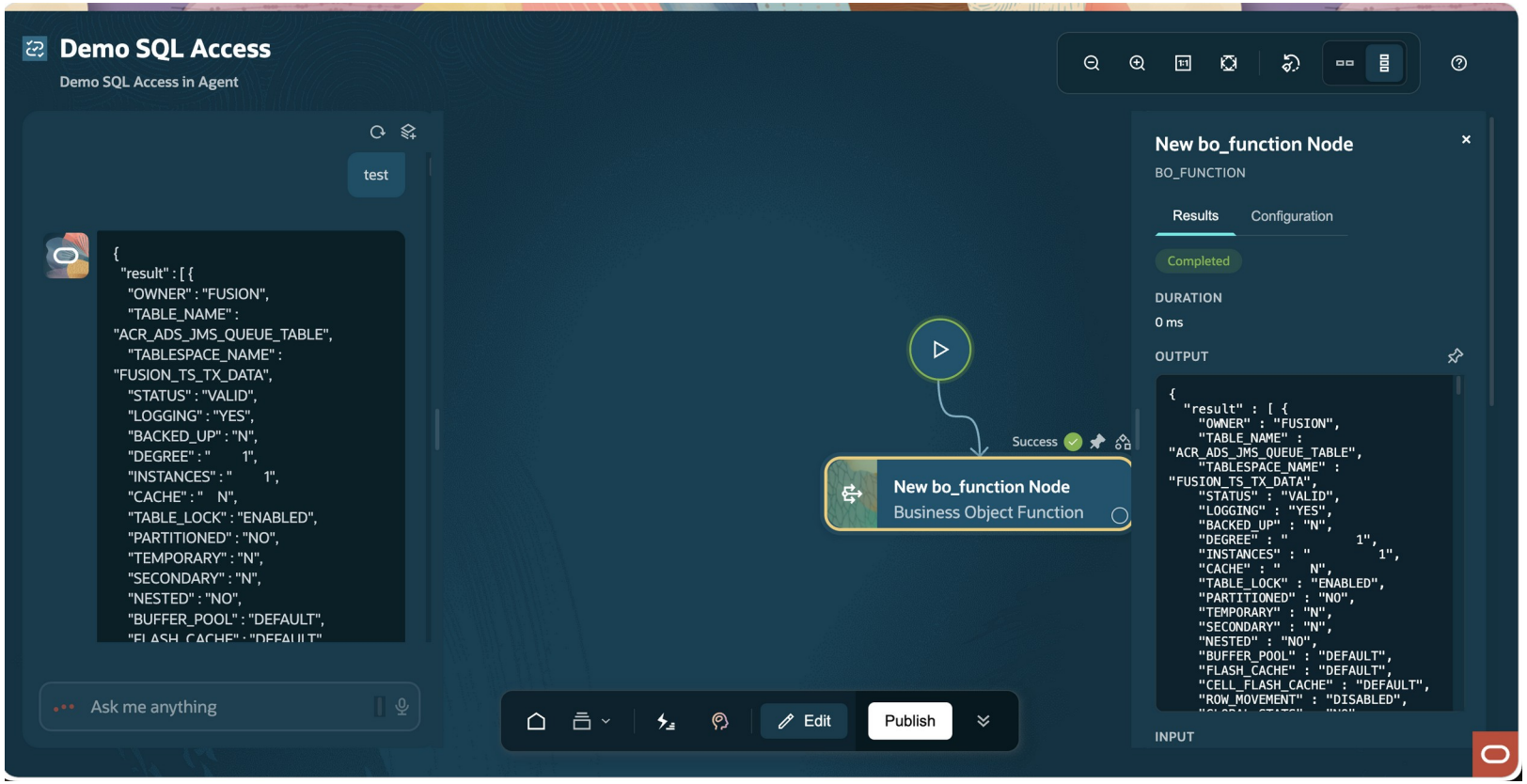
Design-time SQL, step 2: the analyst types the SQL into the call



- Business Object: AC Rest Util
- Function: run_SQL_With_Param
- Fix Json Issues if any: Yes
- sqlText is typed by the analyst in the designer, not by the LLM

The LLM never writes this query. The designer does, and reviews it before the agent runs.

Design-time SQL, step 3: run it, rows come back as JSON



- Node status: Success, Completed
- Output: a JSON result array
- The same rows appear in the chat reply
- A harmless query on system metadata

Full SQL power, defined at design time, with no open surface at runtime.

Never let the LLM write the SQL

Publish parametrized tools instead: the query is fixed at design time, the LLM fills only the parameters. Applies to workflow and supervisor agents.

Why we never let the LLM create SQL

Security model

The BIP proxy runs any SQL, with the function owner's access. The LLM could read data the user was never meant to see.

Wrong results

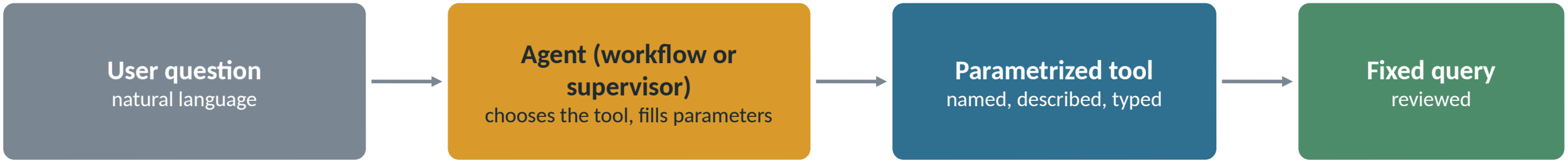
A plausible but wrong join or filter returns rows that look right. The agent presents them as the answer.

Performance

No forced filter or row limit: wide scans and unnecessary load on the database, for every conversation.

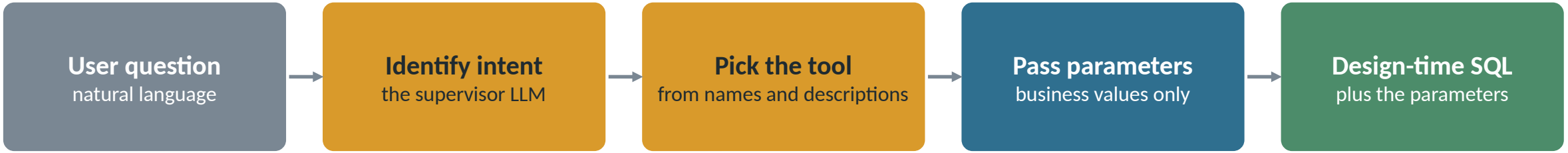
The proxy will run whatever SQL it is given. Someone has to decide what that is: a person, at design time.

The fix: a parametrized tool, not a SQL tool



- The query is pre-defined and reviewed by the analyst, not by the LLM
- The LLM supplies only named, described, typed parameters: never SQL. The natural fit for a supervisor-agent chatbot
- Every query that reaches the BIP proxy stays known, bounded and safe

How the supervisor chatbot picks a tool



Our responsibility

Give every tool a meaningful name and description. They are all the LLM has to choose the right tool for the user's intent.

Not exposed

The generic Business Object function with a free sqlText, as used inside the workflow. The supervisor never gets it.

Exposed

A specific tool that takes business parameters. The tool merges them into the SQL the analyst defined at design time.

Illustrative example

Walkthrough: one question, from intent to answer

1	**User asks**	Which tables start with HZ_ in the FUSION schema?
2	**Supervisor identifies the intent**	A lookup of database table metadata
3	**Supervisor picks the tool**	Table Lookup, chosen from its name and description: lists tables by owner and name prefix
4	**Supervisor passes business parameters**	`p_owner = FUSION` `p_table_prefix = HZ_`
5	**Tool merges them into the design-time SQL**	`select owner, table_name from all_tables where owner = '{p_owner}' and table_name like '{p_table_prefix}%' and rownum <= 50`
6	**Rows come back and the supervisor answers**	JSON rows from the Business Object function, summarized in plain language for the user

Where is the line between powerful and dangerous?

Design-time SQL

Full query typed by the analyst in the function call. Workflow node or supervisor tool.

Reviewed before any run

Parametrized SQL

Query fixed in the tool definition, placeholders filled by the LLM. Workflow or supervisor.

Bounded: only values vary

LLM-generated SQL

Generic tool with sqlText exposed to the LLM, in any agent type.

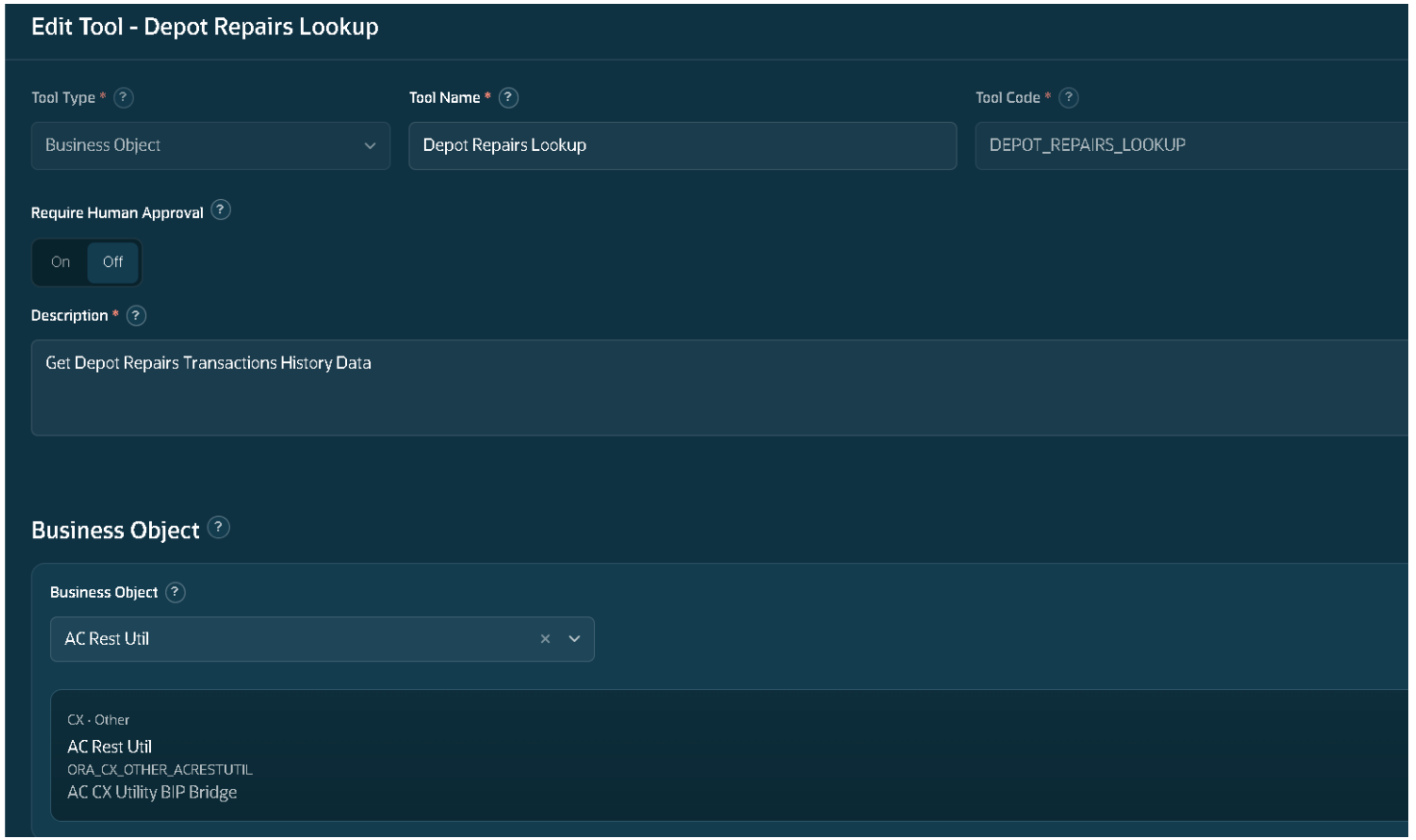
Unbounded: do not do this

Whoever writes the query that runs is where the risk sits.

Choosing how to expose the SQL tool

	Design-time SQL	Parametrized tool	LLM-generated SQL
Who writes the SQL	The analyst, in the function call	The analyst, in the tool definition	The LLM, at runtime
What the LLM supplies	Nothing about the query	Typed parameter values only	The whole query
Review point	Before the agent runs	When the tool is defined	None
Natural fit	Workflow agent: a fixed path you control	Supervisor agent: open conversation, many users	No agent type
Also possible	Supervisor tool with a fixed query	Workflow node with parameters	Never

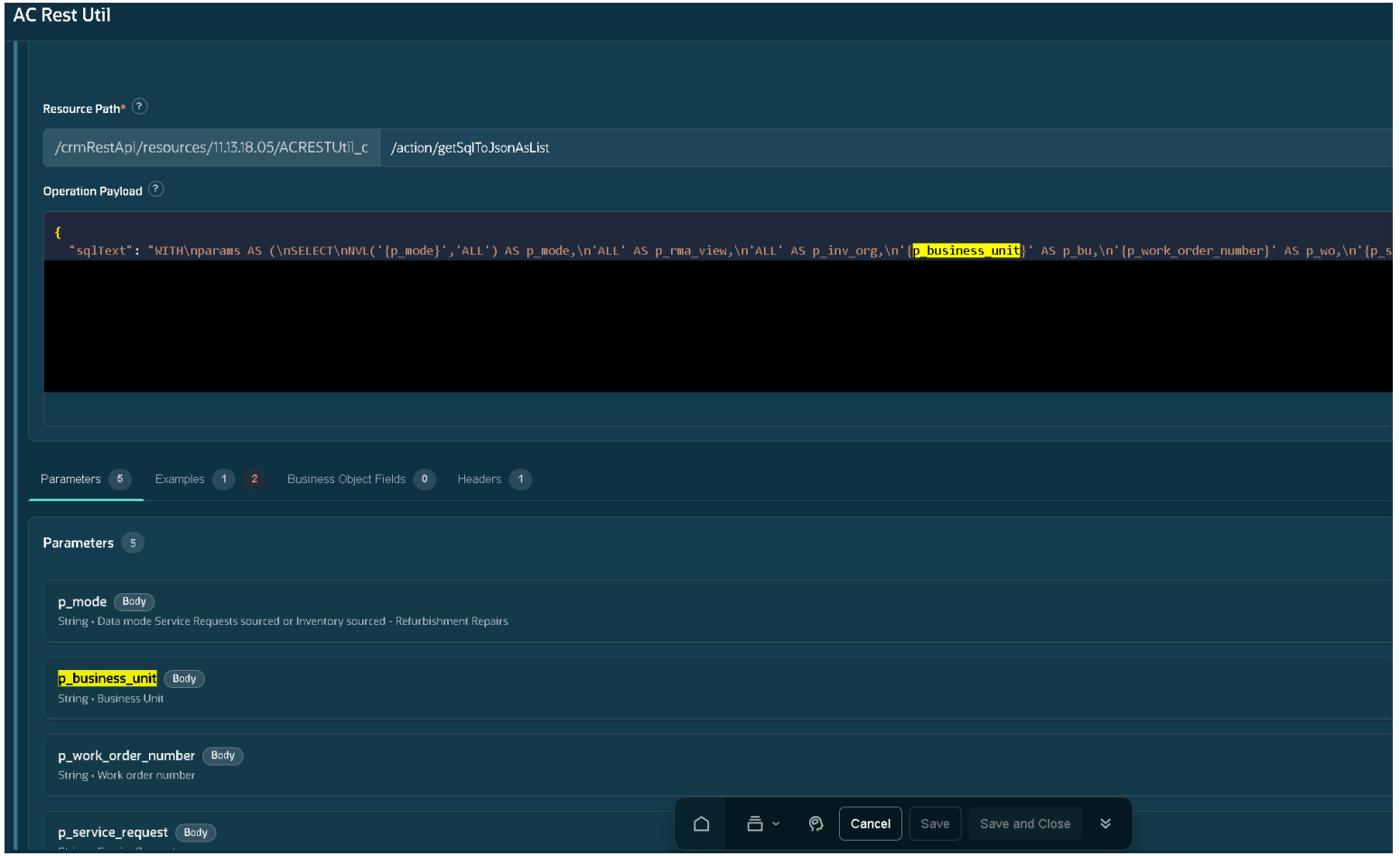
A real tool definition: Depot Repairs Lookup



- Tool Type: Business Object, the same AC Rest Util utility
- Name and code describe one business question, not SQL
- Description: what the supervisor reads to match the user's intent
- Require Human Approval: available per tool (Off here)
- Behind it: a fixed query and typed parameters

The supervisor picks the tool from the user's intent. It never sees the SQL.

Extra tool parameters are embedded into sqlText



- One sqlText holds the whole query
- Placeholders {p_mode}, {p_business_unit}, {p_work_order_number}… sit once, in a WITH params block
- Each placeholder is a parameter: name, type, description
- The LLM fills the values. It never fills sqlText

What the LLM sees: a free-text SQL box or typed parameters

Generic tool: run_SQL_With_Param

One parameter exposed:
sqlText [Body] String SQL Text

If the LLM fills it, the LLM decides what to write.
Nothing bounds the query.

Parametrized tool (Depot Repairs Lookup)

Parameters defined on the tool, each with name, type and description:
p_mode String Data mode
p_business_unit String Business Unit
p_work_order_number String Work order number

They are embedded into the sqlText parameter inside the tool definition. The LLM never fills sqlText.

The description is the only guidance the LLM gets, so write it for the business meaning of the value, not for the database.

Pasting a multi-line query: the JSON-escaping workflow

1

Write and test the query

Multi-line, readable, run with real values

2

Replace values with placeholders

{p_mode}, {p_business_unit}

3

Convert to a JSON-escaped string

Multi-line query becomes one escaped line

4

Paste into the operation payload

Refresh parameters, describe each one

Escape once, paste straight into the payload, and the tool is ready for the agent.

Parameter values are untrusted input

The risk

Values are merged into the SQL text. A value that carries quotes or SQL could change the shape of the query, whether it comes from the LLM or from a user typing in the chat.

Controls on the values

Narrow parameters: one business meaning each, typed and described.

Validate or reject unexpected characters in the wrapper before the merge.

Controls on the query

A read-only SELECT with a row limit.

A least-privilege function owner, since the SQL runs with that user's data access.

The LLM chooses values. It must never be able to change the shape of the query.

Guardrails: treat the tool as privileged

Privileged by design

The SQL runs with the function owner's data access. Restrict who can build and edit agents that use the SQL function, and limit access to the report folder to developer or analyst roles, especially in production.

Fix Json Issues is not a control

It helps with malformed output. It does not replace access controls or query review.

No LLM-fillable sqlText

In any agent type, never expose sqlText as a parameter the LLM fills.

Design-time or parametrized

Workflow node or supervisor tool: reviewed query, typed parameters, bounded results.

Field notes: P2T cloning and Fusion 26C

P2T cloning

The RunReport connection uses the internal fa-internal URL with SAML, so it needs no reconfiguration after a clone.

Quarterly update: Fusion 26C

BI Publisher now validates SQL in data models on save and upload, and rejects DML and DDL.

Any reference to DBMS_SQL is rejected. The upload only says Upload Failed; open the data model and re-apply each data set to see the real error.

We hit it in our own SQL data model and fixed it in CID Fusion Query v3.3.0.

After every quarterly update: search custom data models for DBMS_SQL and re-test the tool.

Takeaways

Build once

One reusable SQL tool: BIP report, global Groovy function, wrapper, Business Object Function.

Design-time SQL

The analyst writes the query in the function call, in a workflow node or a supervisor tool. Reviewed, full power.

Never LLM-written SQL

The BIP proxy runs any SQL: security model, wrong results, performance. Use parametrized tools.

Operate safely

P2T-safe connection, privileged-function mindset, re-test after quarterly updates (26C added a BIP SQL injection check).

Q & A

Real-world use cases, implementation ideas

Thank you

Alex Rabinovich

CID SOFTWARE SOLUTIONS LTD

Alex.Rabinovich@cidsolutions.com

linkedin.com/in/alex-rabinovich-cid

cidsolutions.com/blog



Try the free CID Fusion Query Chrome extension: the same BIP technique, for Fusion developers and analysts.

Additional technique: agent-safe Fusion REST

Business Objects expose any Fusion REST as a function. The catch: a non-200 response.

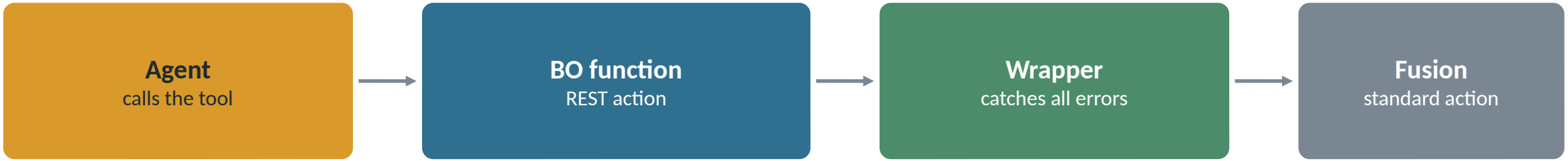
The problem: a non-success REST response breaks the agent

Business Object functions

Any Fusion REST endpoint can be exposed to an agent as a Business Object function, as long as it succeeds.

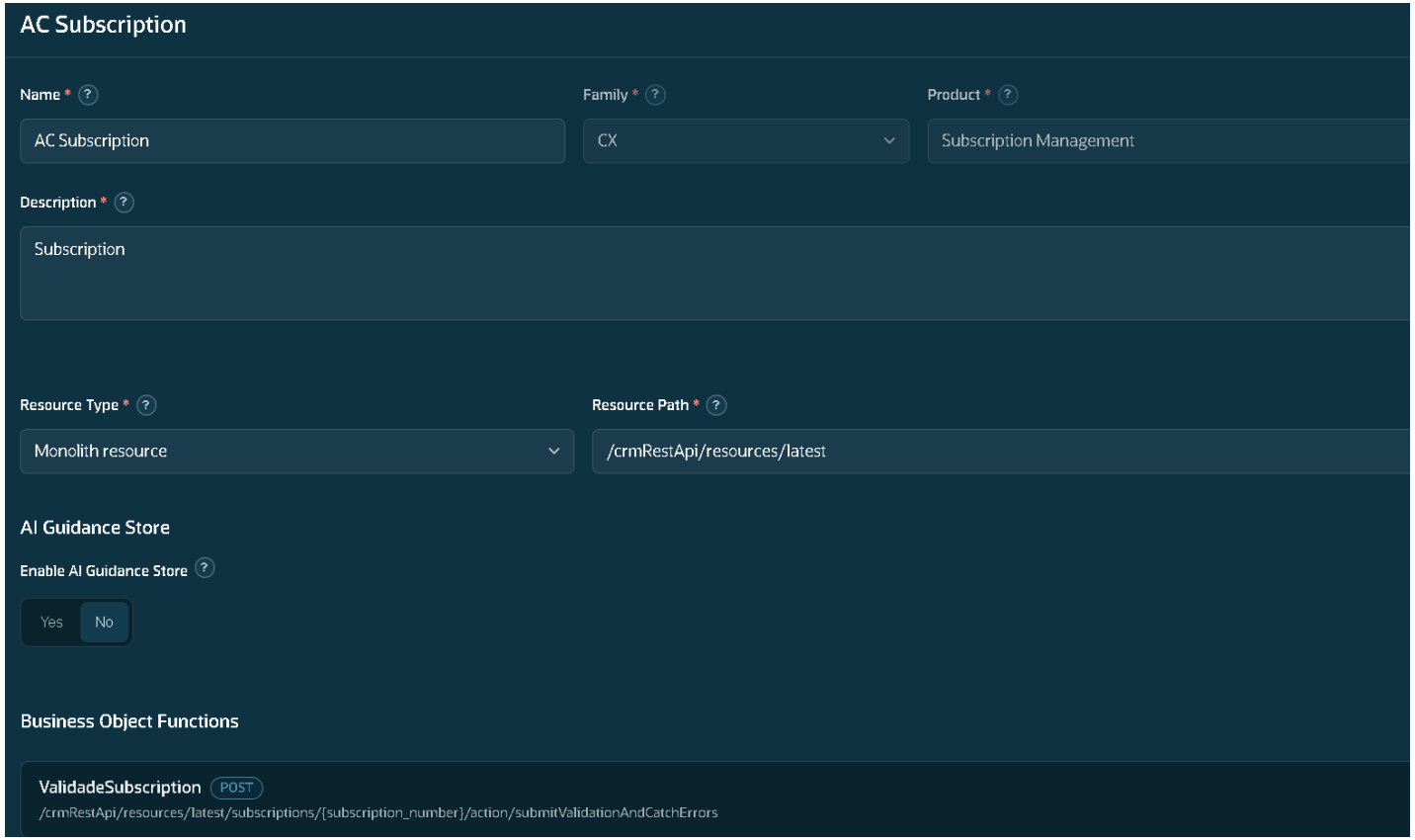
The catch

If the endpoint returns a non-200 status, the call errors out and the agent cannot easily use the result.



Same technique as the SQL tool: a Groovy wrapper that returns errors as a normal response. Our CID Fusion Query extension does the same with an Error row.

Real example: subscription validation



- Business Object: AC Subscription (Subscription Management)
- Function: POST on subscriptions/{subscription_number}, action submitValidationAndCatchErrors
- Validation errors come back as a normal result, not as a failed call
- The agent reads the errors and explains them

Same idea as getSqlToJsonAsList: wrap it in Groovy so the agent gets a result it can use.

The wrapper: submitValidationAndCatchErrors

```
def conn = adf.webServices.ACLocalCrmRest

def headers = [
  'Content-Type': 'application/vnd.oracle.adf.action+json',
  'REST-Framework-Version': 9
]

conn.requestHTTPHeaders = headers;

def num = this.getAttribute("SubscriptionNumber")

try {
  def response = conn.POST("subscriptions", num, "action", "submitValidation", "", "","","","","", [ 'ignoreWarnings': 'N' ])
  return response?.result
}
catch (Exception e) {
   return e.toString()
}
```

Edit Script

- Row-level function on the Subscription object, callable by external systems, returns a String
- Calls the standard submitValidation action through the local CRM REST connection, header REST-Framework-Version: 9
- try returns the result; catch returns the error text as a normal String, so the agent always gets an answer it can read