METAKARTA

Whitepaper

Which lineage holds up outside the demo?

Eight practical tests for determining whether a data lineage platform holds up beyond the demo. Evaluate how each approach performs against the complexity, scale, and change found in a real enterprise data estate.

Table of Contents

What data lineage has to get right	03
The four approaches	04
Comparing the four	06
How to run the evaluation	06
Test 1: Coverage against your actual inventory	06
Test 2: Refresh cost at your estate’s scale	07
Test 3: Parsing depth, down to the line	07
Test 4: Cross-system identity	08
Test 5: One record, every consumer	08
Test 6: Visualization under load	09
Test 7: Point-in-time recall and change	09
Test 8: Impact before deployment	10
Mapping the diagnostic to the tests	10
Scoring your evaluation	11
Where MetaKarta lands on each test	12
What to do next	16

Every vendor in the metadata market will show you a lineage diagram, and most of them look convincing in a 30-minute demo.

What you're actually buying is whether that diagram still holds when it spans nine systems, two clouds, a COBOL copybook, and a stored procedure that last ran in March.

The eight tests in this guide separate the vendors who can hold that diagram together from the ones who can't. Each targets a place where lineage quietly fails in production: coverage gaps, refresh cost, parsing depth, cross-system identity, a single shared record, rendering under load, point-in-time recall, and pre-deployment impact.

Score every vendor on the same eight, then take the top two into a proof of concept on your own data before you sign.

If you've already run the six-question diagnostic in The Four Jobs Enterprise Data Lineage Now Has to Do, bring your answers. Every question in it maps to a test here, and the questions you failed are the tests to weight.

What data lineage has to get right

Data lineage represents your data assets, how they relate, how they're transformed, and where they're consumed. It's also the foundation for what describes those assets: glossary terms, process descriptions, policies, and ownership.

The hard part is accuracy across three dimensions at once.



Across tools.

Every system that stores, moves, or transforms data.



Across environments.

On-premises and cloud, legacy and modern, in the same view.



Across time.

What the estate looked like on any date in question, today's state included.

Miss any one of the three and you have a diagram where you needed a control.

There's now a fourth pressure. Agents and text-to-SQL interfaces like Snowflake Cortex Analyst and Databricks Genie read your metadata to decide what a column means before they answer a question. When someone asks how an AI system arrived at a number, lineage is the record that answers, and it can only answer as far as the metadata underneath it is complete.

The four approaches

Four approaches are on the market. Three describe how lineage gets derived. The fourth describes how the tooling gets assembled, which cuts across the other three and is the one buyers evaluate least.

Inference from patterns

The pitch is that you point AI at your estate and it works out the lineage on its own. It's an appealing idea, and it may get there eventually.

Today it works by applying probabilities to the discovery of lineage relationships themselves. The accuracy range is wide and the results vary between runs on the same inputs, while an auditor asking where a number came from needs a deterministic answer with the code attached.

Analysis of log files

This approach has more substance and has been around for years. It reads query logs and execution logs, then reconstructs how data moved.

The limit is what logs are for. They exist to help operations teams maintain applications and systems, so they carry detail about internal operations and very little about how data gets manipulated. Lineage built from them comes back incomplete for non-relational sources and script-based integration, and noisy with backups, restores, one-time migrations, and other activity that has nothing to do with ongoing data flow.

Some vendors extend the approach by requiring every read and write to be tagged in code so it shows up in the log. Maintaining that across an enterprise's full DI, ETL, and BI footprint is a standing tax on every engineering team you have.

SQL log-based lineage has been pushed hard for the past five to seven years. Enterprises that bought it generally found the covered portion useful and spent the following year asking what to do about everything the logs never saw.

Code parsing

This approach reverse-engineers the estate by reading the actual code and design artifacts that move data: SQL scripts, stored procedures, ETL mappings, BI semantic models, notebooks. Lineage gets computed from what was designed, before anything executes.

The coverage question moves to the parsers themselves. A parser exists for a given technology or it doesn't, so gaps are visible in the connector list at evaluation time. Parsing also reads assets that haven't run recently, which is where log analysis goes quiet.

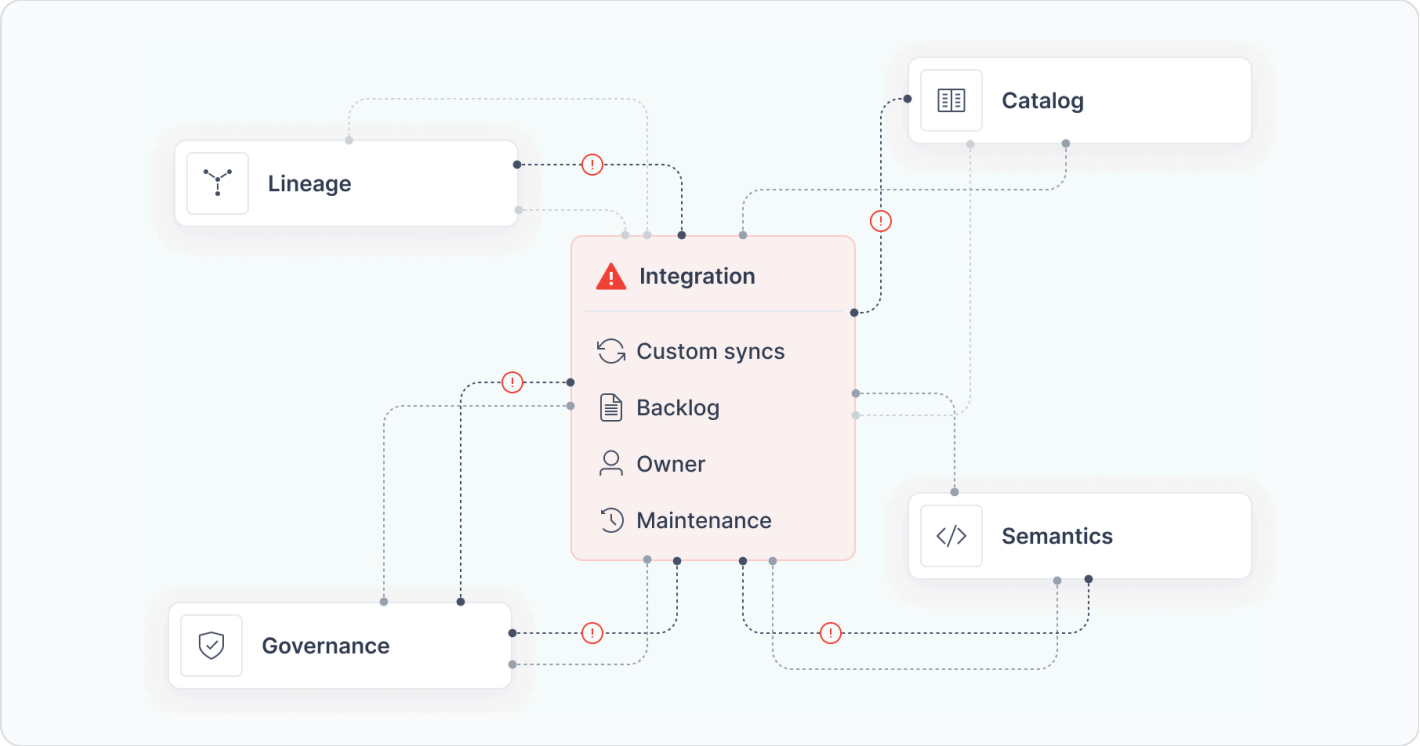
The cost sits with the vendor. Building transformation language parsers and data flow emulators for every ETL platform, database dialect, and BI tool takes years of system-specific engineering, so the market is thin at this depth. Verify the parser list against the technologies actually in your estate, since a vendor claiming parsing across the board is claiming a large engineering surface.

The assembled solution

This one is a buying behavior, so it has no line item on anyone's price list and no vendor presenting it. It's also the most common architecture in production.

You buy a lineage tool, a catalog, a governance workflow tool, and a standalone semantic product. Each one maintains its own metadata store, connects to your sources separately on its own refresh schedule, and holds its own model of what a column is.

The lineage then has to be reconciled against the catalog, the catalog against the governance policies, and the policies against the semantic definitions. That reconciliation work never ends and grows with every source you add, and no amount of API maturity in any one tool retires it.



Comparing the four

Approach	Derived from	Deterministic	Covers dormant assets	Coverage limit
Inference from patterns	Probabilistic matching	No	Partially	Confidence thresholds
Log file analysis	Execution and query logs	Yes, for observed activity	No	What executed and was captured
Code parsing	Source code and design artifacts	Yes	Yes	Parser availability per technology
Assembled solution	Whichever method each tool uses	Varies by tool	Varies by tool	Reconciliation between stores

Check the fourth row first. An assembled solution inherits the limits of every method underneath it and adds reconciliation on top, and Test 5 below is the one that exposes it.

How to run the evaluation

The eight tests split across two settings. Shortlist tests run in a vendor demo or as a paper exercise, so you can put three or four vendors through them in a couple of weeks. POC tests need your own data, so they run only with the two vendors that make the shortlist.

Each test below says where it belongs. A vendor who asks to move a POC test into the demo, on their reference dataset, is asking you to score their data.

Test 1: Coverage against your actual inventory

Where: Shortlist, as a paper exercise before the first demo.

Ask for: a line-by-line match of the vendor's connector list against your own tool inventory, including the systems you'd rather forget.

Why it matters: a lineage graph with three unsupported systems in the middle has three holes in it. Coverage gaps break the chain at the gap, and everything downstream becomes inference.

Connecting to everything is a different problem from opening a JDBC connection and reading system tables. It takes API-level connections and full extraction of the metadata and code specific to each tool.

What a weak answer sounds like:

"We support that through our generic JDBC connector," or "That one's on the roadmap."

Test 2: Refresh cost at your estate's scale

Where: Shortlist demo.

Ask for: a demonstration of the second harvest, then the cost of running it daily.

Why it matters: the first collection is a project with a budget. The 200th is an operating cost, and it's where lineage programs quietly die, because estates with hundreds of millions of assets can't afford a full re-extraction on every cycle.

What a weak answer sounds like:

"We re-scan everything nightly," with no answer on compute cost or run time at your volume.

Test 3: Parsing depth, down to the line

Where: Preview it in the shortlist demo with one of your files; prove it in the POC.

Ask for: a transformation you consider hard. A stored procedure with nested CASE logic, a PySpark notebook, an ETL mapping with 15 transformations. Then ask to see the specific line of code behind a specific column.

Why it matters: column-level lineage that stops at "this table feeds that table" can't explain how a value was produced. You need the expression, the join condition, and the filter.

What a weak answer sounds like:

a column-to-column arrow with no expression behind it, or "Our customers usually open the source file for that part."

Test 4: Cross-system identity

Where: POC.

Ask for: the same logical entity as it exists in three different systems, stitched into one view. Vendor tables, customer tables, and general ledger accounts are good candidates because they're everywhere.

Why it matters: the general ledger feeds tax reporting and rebate accrual. Project teams create overlapping copies out of necessity or because they didn't know the original existed, and a full view of how data gets used requires reconciling identity across systems that describe the same thing differently.

The stitching problems are specific and hard, which is why they get skipped:

1. The same server reached through different hostname aliases.
2. Different SQL syntax against the same database, with default schema names present in one place and absent in another.
3. One application requesting columns by name while another uses SELECT *.
4. Ambiguous syntax, as in SELECT A FROM T1, T2. Which table does column A belong to?
5. SQL that addresses columns by ordinal position.
6. Stored procedures and functions sharing a name across different parameter signatures.

What a weak answer sounds like:

three separate nodes with a manual "same as" link someone drew in the demo environment.

Test 5: One record, every consumer

Where: POC.

Ask for: one column's lineage, its catalog entry, the policy that governs it, and the metadata an AI agent receives when it asks about that column. Then ask how many metadata stores those four answers came from.

Why it matters: this is the test for the assembled solution. If the four answers come from four stores, someone reconciles them on a schedule, and the agent may be reading a definition the catalog has already changed.

It also covers the fastest-growing consumer of lineage. An agent can't ask a follow-up question, so whatever it receives has to carry the definition, the policy, and the derivation together.

What a weak answer sounds like:

"Those sync every night," or an agent integration that returns the column description with no lineage attached.

Test 6: Visualization under load

Where: Shortlist demo on the vendor's largest reference estate, then again in the POC on yours.

Ask for: the diagram for the largest table available, at full depth, with a stopwatch running. Then click through it.

Why it matters: off-the-shelf graph libraries make a slick demo on 200 nodes. Rendering an estate with hundreds of millions of assets responsively takes real engineering, and the difference shows up on your data.

What a weak answer sounds like:

"Let me filter that down first," before the diagram has rendered even once.

Test 7: Point-in-time recall and change

Where: POC.

Ask for: the state of a specific model as of a date six months ago, compared side by side with today's.

Why it matters: lineage is a system of record for regulatory compliance. Proving you understood your data assets means proving it as of the date the regulator names, which requires versioning across the full scope of a model and a visible diff between versions.

What a weak answer sounds like:

"We can export snapshots going forward," which answers nothing about March.

Test 8: Impact before deployment

Where: POC.

Ask for: a proposed schema change from your backlog, and the list of downstream consumers it would affect, produced before the change ships. Include consumers that haven't run this quarter.

Why it matters: regulatory control, incident resolution, and change management all resolve to one question: what does this touch? Impact analysis that runs after production breaks is a post-mortem tool, and it slows every release because the safe move becomes manual verification.

What a weak answer sounds like:

a consumer list built from the last 30 days of query logs.

Mapping the diagnostic to the tests

If you ran the diagnostic in The Four Jobs, your answers carry straight across. The other two tests are new at this stage.

Test	Four Jobs diagnostic question
Test 1: Coverage	Does the trace cross every system in the path, including the oldest one?
Test 2: Refresh cost	When was this lineage last refreshed, and what does a refresh cost?
Test 3: Parsing depth	Can you get from this number to the specific line of code that produced it?
Test 4: Cross-system identity	Does the same entity across three systems resolve to one view?
Test 5: One record	No diagnostic question. It tests Job 4 in that paper, supplying meaning to AI systems, plus the assembled-solution problem

Test	Four Jobs diagnostic question
Test 6: Visualization under load	No diagnostic question. Rendering speed only shows up once a vendor loads a large estate, so it starts here
Test 7: Point in time	What did this definition mean six months ago?
Test 8: Impact	Which consumers would a proposed change to this column break?

Scoring your evaluation

Score each vendor 0, 1, or 2 on every test.

A 2 means passed on your data or your inventory. A 1 means shown only on the vendor's reference data, or passed with manual steps in between. A 0 means not shown.

Test	Pass condition	Where	Vendor A	Vendor B	Vendor C
1. Coverage	Every system in your inventory has a named connector	Shortlist			
2. Refresh cost	Incremental harvest demonstrated, daily cost known	Shortlist			
3. Parsing depth	The specific line of code behind a specific column	Shortlist preview, POC proof			
4. Cross-system identity	One entity, three systems, one stitched view	POC			
5. One record	Lineage, catalog, policy, and agent metadata from one store	POC			
6. Visualization	The largest diagram, timed, clickable	Shortlist and POC			
7. Point in time	A model's state on a date you name, diffed against today	POC			
8. Impact	Pre-deployment consumer list, dormant consumers included	POC			

The total matters less than where the zeros fall. A zero on Test 1 is a hole in the chain no other score makes up for, and a zero on Test 5 means you're buying the reconciliation work along with the tool.

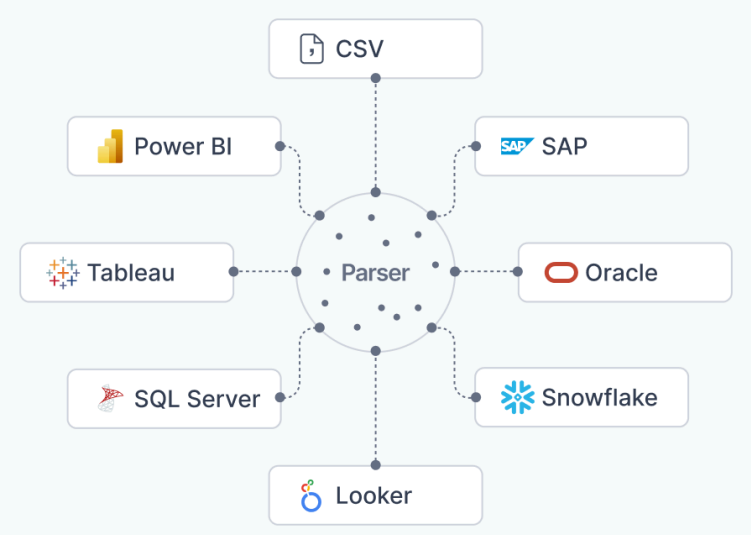
Take the two highest scorers into a POC and rerun Tests 3 through 8 on your estate. A vendor who passes seven and waves at the eighth has told you where the gap is.

Where MetaKarta lands on each test

We built this guide around the tests our own platform has to pass, so here are our answers, test by test. Hold every vendor, including us, to running them on your data.

Coverage.

400+ metadata connectors, from legacy COBOL systems through modern cloud platforms, spanning storage, data integration and ETL, business intelligence, standards, and applications. They're the same connectors that run as the OEM engine inside Microsoft Purview, Informatica from Salesforce, IBM, Oracle, and Qlik Talend. Check them against your inventory in the connector explorer.



Refresh cost.

The connectors track what they've already collected and compare it against current state, so only changed assets get imported into the repository. Change management sits with the connector.

Parsing depth.

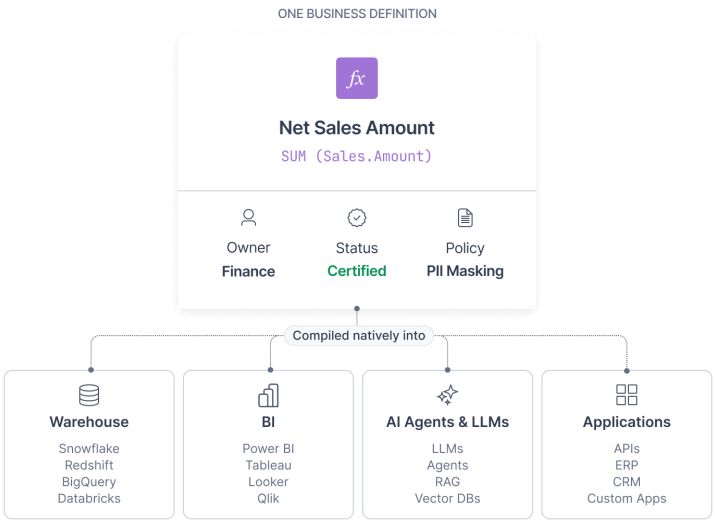
Transformation logic gets parsed down to individual columns, including the expressions and control conditions that determine how data is transformed. SQL parsing covers Microsoft SQL Server Transact-SQL, Oracle PL/SQL, Teradata BTEQ and Fastload, PostgreSQL, Snowflake, and others, with Python and Scala support for Apache Spark and Databricks, and transformation logic is harvested from DI and ETL platforms including Informatica PowerCenter, IBM DataStage, Qlik Talend, and dbt. At the column level, lineage exposes the process, its context, contributing data operations, transformation expressions, and control operations such as joins, lookups, and filters.

Cross-system identity.

Pattern matching and inference run across metadata and sample data, with probability weighting and confidence scoring used to resolve which assets refer to the same thing. This runs on top of lineage the parsers have already resolved deterministically, so inference decides identity across systems while the lineage itself stays derived from code.

One record.

Data Lineage, Data Catalog, Data Governance, and Semantic Hub all read from and write to one shared metadata repository, so the lineage, the catalog entry, and the policy for a column are the same record. MetaKarta MCP serves that record to agents, with governed lineage and catalog metadata through the Metadata Management Tools and compiled context through the Semantic Hub Tools.

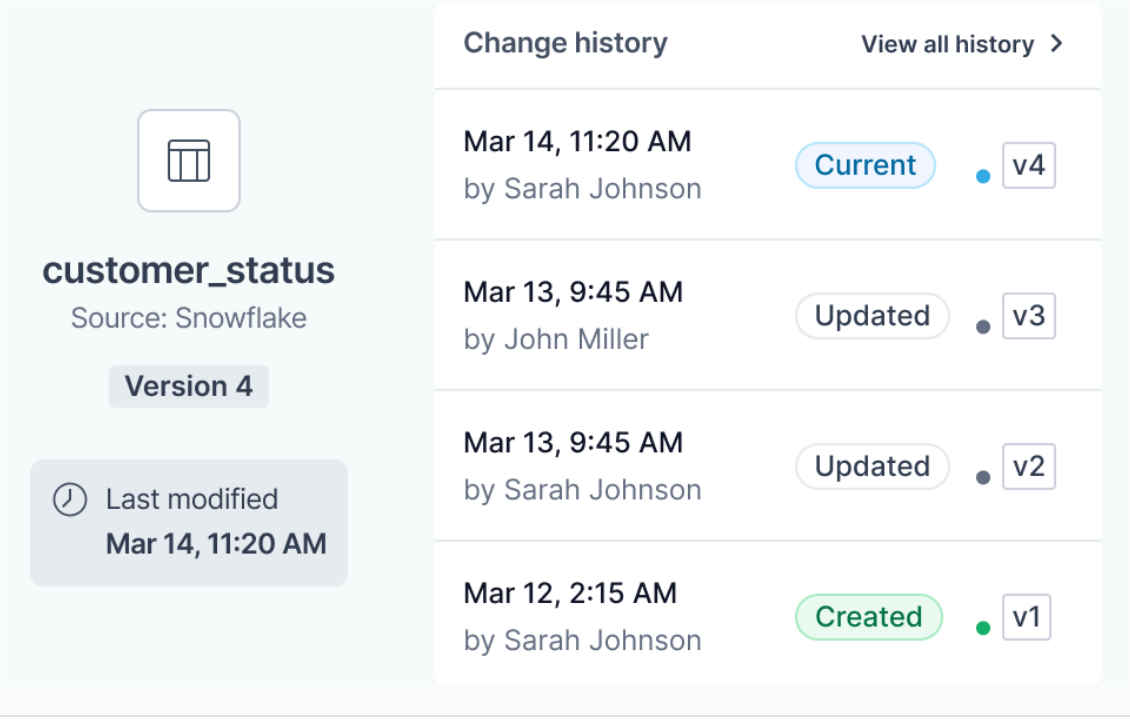


Visualization.

Lineage graphs are retrieved from the repository and rendered through an interactive graphics engine, with backend filtering to reduce the data sent for display. Users can limit scope and progressively expose the columns and relationships they need, and view lineage as an interactive diagram, classic diagram, or tree, with filters for temporary, internal, and external objects and for conditional labels such as PII.

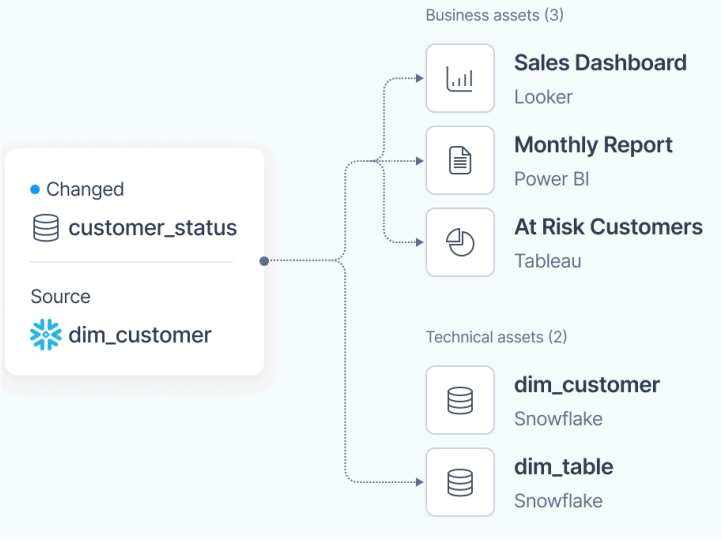
Point in time.

Version and configuration management runs across any scope of lineage model components, called multi-models, and Compare Metadata shows additions, changes, and removals between versions side by side. Multi-model versions also make incremental harvesting practical, since only changed databases, schemas, packages, reports, or files need re-collection.



Impact.

Impact analysis runs from parser-derived dependencies, so a consumer that hasn't executed in six months appears in the list alongside one that ran last night. Lineage by design, not by observation.



What to do next

Score the shortlist on the eight tests, then take the top two into a POC. What Should a Lineage POC Prove on Your Own Estate? covers the six workflows to run there, what to baseline before the first harvest, and the numbers to bring back for the business case.

Get in touch to run these tests against your own inventory.

Request Demo

About Meta Integration Technology

Meta Integration Technology, Inc. (MITI) is a pioneer in metadata integration, semantic modeling, and data lineage with decades of experience serving global enterprises and technology providers. MITI's technology is embedded in products from leading data and analytics vendors and powers MetaKarta, its enterprise platform for metadata-driven data management. MetaKarta delivers data lineage, data cataloging, data governance, and semantic enforcement through a single integrated system, connecting the full enterprise data estate across 400+ native connectors, from legacy to modern and on-premises to the cloud.

650 Castro Street, Suite 120-431, Mountain View, California 94041-2055 USA
Phone: 650.429.8853
info@metakarta.com
www.metakarta.com