

Impact & root cause analysis

Catch downstream breakage before deployment.
Trace root cause issues in minutes.



Table of Contents

Why impact analysis comes up short	03
What parser-based lineage changes	03
Root cause in minutes	04
The proof to ask for	04
The metric that moves	04
Where this leaves the practice	05

A column gets renamed as part of a routine change, and three days later a dashboard breaks in production.

Then the incident response starts: who changed what, when, and what did it touch. The investigation runs backward through logs, Slack threads, and whoever is on call that week.

If someone happens to remember which pipeline reads from that column, the dependency gets traced by hand. Otherwise the team assembles the picture one piece at a time, out of systems that share no metadata with each other.

So the analysis that belonged before the change gets done after the break.

Why impact analysis comes up short

Log-based lineage captures what executed during the observation window. What sits defined in the code, what runs next week, and what fires quarterly and last ran 70 days ago all stay outside it.

That gap makes impact analysis probabilistic. The report says 14 objects depend on this column. The real number is 19, and the five you're missing are the consumers that haven't run recently, so the first sign of them is the production incident.

Manual verification covers the gap only for what the team already knows to check. Tribal knowledge has a scope limit: a new engineer has no reason to know what the legacy SSIS package touches, and the quarterly report that joins to that table is remembered by one person on the team.

Root cause after the fact has the same structural cause. Tracing a wrong number back means jumping between the lineage tool, the catalog, the source repository, and the ETL logs, each holding part of the answer. Assembling the whole picture takes hours, usually from whoever has the most context, under pressure, while stakeholders wait.

What parser-based lineage changes

MetaKarta's lineage engine parses the code artifacts that define the estate: SQL, stored procedures, dbt models, SSIS packages, Informatica PowerCenter mappings, and the rest of a 400+ connector inventory that reaches from legacy to modern. Lineage by design, not by observation.

Every object that depends on a column appears in the lineage graph, because the dependency is written into the code the engine reads. A consumer that hasn't executed in six months appears. A quarterly job shows its dependencies whether or not it fired during any observation window.

Impact analysis then runs before a schema change ships, and it returns every affected downstream consumer: tables, transformations, BI reports, semantic models, AI queries. Know the blast radius before the first ticket lands.

Root cause in minutes

When something breaks anyway, the same engine traces the broken number back to its origin.

The path from a wrong dashboard number to the upstream column that caused it runs through parsed lineage, with every transformation in the chain visible. Each step traces to the code artifact that defines it, down to the line of SQL, the ETL mapping, and the calculation inside the BI expression. The investigation that took a morning runs as a query against the lineage graph.

The proof to ask for

Pick a column in a system the team considers well documented. Then, in one session:

- ✓ Run parser-based impact analysis against that column and read the full consumer list.
- ✓ Compare it against what the current log-based tool returns for the same column.
- ✓ Find the consumers in the gap and check when each last executed.
- ✓ Take a number off a BI report and trace it back to the line of code that produced it.

Step 3 decides the evaluation. A small gap means the current tooling is doing its job, and a large one is the number to bring to the review.

The metric that moves



Coverage delta.

The count of downstream consumers a parser-based analysis returns for a given column, against the count the log-based tool returns for the same column. That difference is the set of dependencies currently shipping unreviewed.



Mean time to root cause.

Hours per investigation today, measured against the same investigation run as a lineage query. This is the number that shows up in on-call load and in incident review.

Where this leaves the practice

Impact analysis moves from a post-incident activity to a standard pre-change step, because it's fast enough and complete enough to be worth running every time.

Schema change tickets carry an impact report. The team reads the downstream risk before deployment, and a change showing 19 affected objects where the team expected three gets reviewed on different terms than one that shows what everyone assumed.

The effects compound over the following quarters. Incidents caused by silent schema changes decline as the reviews catch them, root cause investigations shrink from days to minutes, and the on-call engineer spends the shift on engineering work instead of reconstructing dependency maps by hand. Request a demo to see a parser-based impact analysis surface the consumers a log-based tool misses, and a broken number traced back to the line of code behind it.

[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