# Building a dbt Knowledge Graph Chatbot Proof of Concept

## Executive Summary: The Value Proposition of a dbt Knowledge Graph Chatbot

Modern data teams rely heavily on dbt (data build tool) to manage complex data transformation pipelines. While dbt’s core functionality automates the creation of a Directed Acyclic Graph (DAG) that visualizes model dependencies, understanding the intricate relationships within this graph remains a significant challenge for data professionals.1 The existing

dbt docs tool provides a static visualization that is not conducive to answering complex, interactive queries. For instance, a data analyst cannot easily answer a question like, "Which reports or dashboards will be impacted if a specific source table is updated?" or "Trace the full lineage of the customer\_lifetime\_value metric back to its original sources" without manual investigation. This manual effort creates a bottleneck for critical tasks such as data governance, root-cause analysis during data quality incidents, and compliance auditing.3

This report details a blueprint for a conversational chatbot that addresses these limitations. The proposed solution is a Proof of Concept (PoC) that converts dbt’s metadata from the manifest.json artifact into a dynamic, queryable knowledge graph. This graph is stored in a local graph database. The system is enhanced by a locally-run Large Language Model (LLM) powered by Ollama, which translates natural language questions into formal graph queries (Cypher).7 The query results are then translated back into a human-readable response. This architecture not only makes complex data lineage information accessible to a wider audience within an organization but also provides a robust foundation for building automated data governance and discovery tools. The project demonstrates how leveraging the inherent graph structure of dbt artifacts can be a powerful first step toward a more transparent and manageable data environment.

## Foundational Concepts and Architectural Overview

### The dbt Manifest as the Foundation for a Knowledge Graph

The conceptual foundation of this project lies in the structure of dbt itself. The dbt manifest.json artifact serves as a complete "metadata blueprint" for a dbt project.9 Generated by any dbt command that parses the project, this file contains a comprehensive representation of all resources, including models, sources, tests, and exposures.10 More importantly, the file includes a

parent\_map and a child\_map, which explicitly define the upstream and downstream dependencies that form the project’s DAG.10 The use of

ref() functions within dbt models is the mechanism through which these relationships, the edges of the graph, are created.1

This inherent graph structure within manifest.json makes it a natural candidate for a graph database. The file is not simply a collection of metadata; it is a serialized representation of a directed graph. The nodes of this graph are the dbt resources (e.g., models, sources, tests), and the edges are the dependencies between them. By externalizing this structure into a dedicated graph database, we move from a static file to a dynamic, queryable data store. This approach capitalizes on the fact that dbt is already a graph-centric tool, operationalizing its core structure for a new purpose: answering complex queries through a conversational interface.

### The Architectural Necessity of Graph Databases for Lineage

Graph databases are an ideal choice for modeling data lineage because their architecture is fundamentally aligned with the nature of the data itself.3 In a graph database, data is stored as a network of interconnected entities (nodes) and their relationships (edges). This contrasts sharply with traditional relational databases, where relationships are inferred by joining data across multiple tables. For a data lineage problem, which involves tracing a path through a chain of dependencies, this difference is profound.

Consider a multi-hop query to trace the full lineage of a model. In a relational database, this would require a complex series of recursive JOIN operations, which can be computationally intensive and difficult to write. A graph database, on the other hand, can execute this query efficiently as a single, simple traversal operation, such as MATCH (a)-->(b) using a language like Cypher.3 The schema of a graph database is also highly flexible, which is a major advantage given that data pipelines and their dependencies are constantly evolving. This flexibility allows for the easy addition of new node types (e.g., reports, dashboards) and relationship types (e.g.,

OWNS, HAS\_TAG) without requiring a complete data migration. Choosing a graph database is therefore a strategic architectural decision that directly matches the structure of dbt’s DAG, enabling highly efficient and intuitive querying for data lineage.

### Beyond Basic RAG: The Strategic Role of GraphRAG

The chatbot’s conversational capability is enabled by a technique known as Retrieval-Augmented Generation (RAG). RAG enhances an LLM's responses by providing it with external, factual data to prevent it from generating inaccurate or misleading information, a common issue known as hallucination.11 However, a basic RAG system, typically based on vector similarity search, retrieves unstructured documents or text snippets that are semantically similar to a user's query.13 This approach works well for answering factual questions but falls short when dealing with structured, relational data like a data lineage graph.

This is where GraphRAG becomes essential. GraphRAG is a specialized form of RAG that uses a knowledge graph to provide the LLM with a highly structured, connected context.11 Instead of retrieving a document, the system retrieves a subgraph of relevant entities and relationships. This structured context allows the LLM to perform complex, multi-hop reasoning, which is necessary to answer questions like, "What models are upstream of

fct\_sales?" or "Which columns in the customer\_details table are derived from PII-tagged sources?" The LLM's task shifts from merely summarizing text to reasoning over a symbolic representation of the data world. This approach, exemplified by tools like NeoConverse, represents a more advanced application of LLM technology, moving beyond simple information retrieval to genuine structured reasoning.14

## Setting Up the Local Development Environment

The PoC is designed to run entirely on a local machine, which is a deliberate choice to ensure reproducibility, minimize costs, and maintain data privacy. A local-first architecture eliminates the need for cloud services or API keys, as all data processing and LLM inference occur on the local machine. This section details the necessary components and the Python-centric approach that will be used to integrate them.

The core components of the environment are:

* **Python**: Version 3.8 or higher.
* **Graph Database**: Neo4j Desktop, recommended for its ease of use and integrated management of databases and plugins. Other options include running Neo4j Community Server or a Docker container.16
* **LLM Runtime**: Ollama, an open-source tool that simplifies the process of running large language models locally.
* **Python Libraries**: A set of key Python packages that serve as the connective tissue for the entire system.

This architecture leverages Python's extensive ecosystem to serve as the primary orchestration layer. Dedicated client libraries for Neo4j and Ollama enable programmatic control over the database and LLM, while a specialized parser library simplifies the handling of dbt's metadata artifacts. This unified approach, built around a single programming language, streamlines the development process and allows the developer to focus on the core logic of the chatbot rather than the complexities of integrating disparate technologies.

### Local Development Environment Checklist

The following table outlines the required software and libraries for setting up the local PoC environment.

| Component | Recommended Version | Installation Method | Purpose in PoC |
| --- | --- | --- | --- |
| **Python** | 3.8+ | (<https://python.org/>) or use a package manager. | The primary programming language and orchestration layer. |
| **Graph Database** | Neo4j Desktop 5.x | (<https://neo4j.com/download-center/>) the application. | Stores the dbt lineage as a knowledge graph. Manages plugins like APOC. |
| **LLM Runtime** | Ollama v0.1.x | (<https://ollama.com/>) or use a package manager. | Runs open-source large language models locally for natural language processing. |
| **Python Library** | neo4j | pip install neo4j | The official Neo4j Python driver for executing Cypher queries against the database.18 |
| **Python Library** | dbt-artifacts-parser | pip install dbt-artifacts-parser | Parses manifest.json and other dbt artifacts into Python objects.20 |
| **Python Library** | ollama | pip install ollama | The official Python client for interacting with the local Ollama LLM runtime.7 |
| **Python Library** | streamlit | pip install streamlit | A rapid prototyping framework for building the chatbot's web-based user interface.22 |

The performance of the local LLM will be limited by the available hardware, a trade-off that is acceptable for a PoC. A more powerful LLM hosted in the cloud would offer better performance and accuracy but at the cost of complexity and ongoing expense.11

## Building the Data Ingestion Pipeline: From manifest.json to Knowledge Graph

The first major task in this project is to populate the Neo4j database with the dbt project’s metadata. The process is a classic Extract-Transform-Load (ETL) pipeline, orchestrated entirely by a Python script.

### The Ingestion Challenge as a Structured Data Problem

The naive approach to ingesting the manifest.json data might be to feed the raw JSON file to an LLM and ask it to parse the contents. However, this is highly inefficient and risks data integrity issues, as LLMs can misinterpret complex, structured data and produce inconsistent outputs. The correct approach is to treat the manifest.json as a structured data source and use a purpose-built parsing library. The dbt-artifacts-parser is a Python library specifically designed for this task. It handles the nuances of dbt's artifact versions and provides a stable, predictable way to load the JSON data into a Python object model.20 This ensures that the ingestion process is robust, accurate, and reproducible. The data ingestion stage is a traditional data engineering problem; the LLM's power is best reserved for the subsequent, more complex task of natural language understanding.

The manifest.json provides all the necessary information to model the dbt project as a graph, including nodes (models, tests, etc.), sources, metrics, and their parent\_map and child\_map relationships.10 The file also contains detailed information about each resource, such as

unique\_id, name, description, tags, and raw\_code.10 The following table provides a blueprint for mapping these dbt artifacts into a Neo4j Labeled Property Graph (LPG) model.

### dbt manifest.json to Graph Schema Mapping

| dbt Artifact Key (.json field) | Neo4j Node Label(s) | Neo4j Properties | Relationships (Edges) from depends\_on or child\_map |
| --- | --- | --- | --- |
| nodes (sub-keys: model, seed, analysis) | :Model, :Seed, :Analysis | unique\_id, name, description, tags, raw\_code | --> to nodes or sources |
| nodes (sub-key: test) | :Test | unique\_id, name, description | --> to nodes or sources |
| sources | :Source | unique\_id, name, database, schema | --> from nodes |
| exposures | :Exposure | unique\_id, name, description, type | --> from nodes |
| metrics | :Metric | unique\_id, name, description | --> from nodes |

### Idempotent Data Loading for Robust Pipelines

The ingestion script must be designed for repeated execution. In a real-world scenario, the manifest.json is regenerated after every dbt run, and the graph database must be updated to reflect the latest state of the project. If the script were to use the CREATE command in Cypher, it would create duplicate nodes and relationships with every run, leading to a cluttered and inaccurate graph.

The correct and professional approach is to use the MERGE clause in Cypher.24

MERGE is an idempotent command that either matches an existing pattern in the graph or creates it if it does not exist. This is the perfect solution for a data ingestion pipeline that needs to be run repeatedly. By using MERGE on a node's unique\_id property, the script can guarantee that each dbt resource is represented by a single, unique node in the graph. The script can also use ON CREATE clauses to set initial properties, like a created\_at timestamp, and ON MATCH to update other properties, such as a last\_updated\_at timestamp. This design choice ensures that the data pipeline is not only functional for a one-off PoC but is also robust enough for a production environment where the underlying dbt project is constantly evolving.

## Creating the Natural Language Querying Engine

The core of the chatbot's intelligence is its ability to understand a natural language question and translate it into a query that the graph database can execute. This process, often referred to as Text2Cypher, is a key challenge for any natural language interface for graph databases.

### The Text2Cypher Challenge and the Role of In-Context Learning

An LLM can be prompted to generate Cypher, but its performance is highly dependent on the quality of the provided context. Without a clear understanding of the graph's schema (node labels, relationship types, and properties), the LLM is likely to produce inaccurate or syntactically incorrect queries. A crucial strategy is to provide the LLM with a "semantic layer" that grounds its reasoning.26 This context should include:

* A simplified representation of the graph's schema, such as the node labels and their properties.
* Definitions and descriptions for key dbt resources, derived from the manifest.json itself.
* A few-shot learning examples, which are pairs of natural language questions and their corresponding, correct Cypher queries.15

This method does not just ask the LLM to generate a query in a vacuum; it instructs the LLM on how to navigate the specific logical model of the dbt project. This in-context learning is far more effective than relying on the LLM's pre-trained knowledge alone.

### The Agentic Approach to Query Execution

A sophisticated approach to the Text2Cypher task is to implement the LLM as an agent with access to specific tools. In this architecture, the LLM is not just a passive query generator but an active component that decides when and how to interact with its environment. The primary "tool" for this agent is a Python function that can execute a Cypher query against the Neo4j database.

The workflow is as follows:

1. A user asks a question in natural language via the chatbot interface.
2. The system sends this question, along with a carefully crafted system prompt, to the local Ollama LLM. The system prompt instructs the LLM that it has access to a tool named execute\_cypher\_query.
3. The LLM's task is to analyze the user's question and determine if it can be answered by calling this tool. If it can, the LLM generates a Cypher query as the argument for the tool call.
4. The system intercepts the LLM's tool call, executes the generated Cypher query against the Neo4j database using the Python driver, and retrieves the results.
5. The query results, which may be complex and technical, are then returned to the LLM.
6. The LLM's final task is to synthesize the query results into a concise, human-readable answer that addresses the user's original question.

This agentic pattern, supported by the Ollama Python client's tool-calling capabilities, provides a robust and traceable way to bridge the natural language front-end with the structured graph database backend.7

## Building the Chatbot Interface and Putting It All Together

The user interface for the chatbot is a critical component for demonstrating the PoC. While a full-stack web application would provide a polished user experience, it would require significant development time and expertise. For a PoC, the primary objective is to quickly and effectively demonstrate the core value proposition: natural language querying of dbt lineage.

### Rapid Prototyping with Streamlit

Streamlit is an ideal choice for building the chatbot interface for this PoC. It is a Python-native framework that allows developers to create interactive web applications with minimal code.22 This single-script approach eliminates the complexities of setting up separate frontend and backend services (e.g., using Flask and a JavaScript framework).30 The interface can be built with basic Streamlit components: a title, a text input field for the user’s question, and a display area to show the chatbot’s responses.23 Streamlit’s architecture, which reruns the entire script upon user interaction, is simple and effective for this use case. This allows the developer to focus on the core logic of the LLM-database integration rather than the boilerplate of web development, thereby accelerating the prototyping process.

The final system architecture connects these components in a cohesive workflow:

1. **Data Source**: A local dbt project with a manifest.json artifact.
2. **Ingestion Script**: A Python script that uses dbt-artifacts-parser and the Neo4j Python driver to load the dbt metadata into the Neo4j database.
3. **Chatbot Application**: A Streamlit application that provides a user interface.
4. **LLM Backend**: A local Ollama instance running a suitable LLM (e.g., llama3.1).
5. **LLM-Database Bridge**: The core Python logic within the Streamlit app that uses the Ollama client to prompt the LLM, capture its tool calls, execute Cypher queries on Neo4j, and return formatted results.

This integrated system creates a fully functional, end-to-end PoC that effectively demonstrates the power of a dbt knowledge graph chatbot.

## Insights, Limitations, and Future Roadmap

The PoC outlined in this report is a powerful demonstration of a new capability for data teams. It validates the core architectural principles of using a graph database and a local LLM to enable natural language querying of dbt lineage. However, a responsible expert report must also acknowledge the limitations of this PoC and provide a clear roadmap for a production-ready solution.

The current implementation has three primary limitations:

1. **Limited Lineage Granularity**: The PoC relies exclusively on manifest.json, which only provides table-level lineage information.6 The chatbot cannot answer questions about column-level dependencies (e.g., "Where does the  
   total\_revenue column come from?") because this information is not explicitly captured in the artifact. Achieving column-level lineage requires more advanced methods, such as parsing the compiled SQL queries with a dedicated SQL parser library or analyzing database query logs.6 Furthermore, dbt Python models pose a challenge to lineage parsing, as their transformations cannot be inferred from static SQL analysis.6
2. **Scoped Data Sources**: The PoC is limited to a single dbt project. Real-world enterprise data environments are far more complex, encompassing multiple dbt projects, external data sources, dashboards, and other artifacts.35 A production-ready solution must be able to ingest and stitch together metadata from these diverse sources to provide a truly comprehensive view of the data ecosystem.
3. **Performance and Scalability Constraints**: The local-first architecture, while excellent for a PoC, is inherently limited by local hardware. A large dbt project could result in a graph too big for a single-node database, and a local LLM will be slower and less powerful than a cloud-based equivalent.37

The architectural pattern demonstrated in this PoC—ingesting metadata and representing relationships in a graph—is the same foundation used by mature, open-source data catalogs like OpenMetadata, Amundsen, and Apache Atlas.39 These platforms offer a comprehensive suite of features, including a Unified Metadata Graph, support for multiple data connectors, and built-in governance capabilities for managing tags, owners, and descriptions.44

### PoC vs. Production System Feature Comparison

| Feature | PoC Implementation (Local-first) | Production-Ready System (Enterprise-scale) |
| --- | --- | --- |
| **Data Scope** | Single dbt project from one manifest.json. | Multiple dbt projects, dashboards, BI tools, and external data sources. |
| **Lineage Granularity** | Table-level lineage only, derived from depends\_on. | Column-level lineage via advanced SQL parsing or query log analysis. |
| **Performance** | Dependent on local machine's CPU/GPU and I/O speed. | High-performance, scalable cloud-based graph database and LLM services. |
| **Scalability** | Limited by local hardware and the size of a single manifest.json. | Horizontal scaling to handle thousands of data assets and complex relationships. |
| **User Interface** | Simple, single-script Streamlit application. | Integrated, multi-page web application with rich visualizations and search. |
| **Cost** | Free and open source. | Managed cloud service costs or on-premise infrastructure/licensing fees. |
| **Data Governance** | Basic retrieval of tags and descriptions from manifest.json. | Comprehensive governance with automated policy enforcement and compliance auditing. |

## Conclusions and Recommendations

The dbt knowledge graph chatbot PoC is a successful demonstration of a powerful, modern data architecture. It establishes a direct and effective way to use LLMs and graph databases to solve the perennial problem of data lineage discovery. The core finding is that dbt's manifest.json is a perfect foundation for a knowledge graph, and a GraphRAG approach is uniquely suited for performing multi-hop reasoning over this structure. The project's local-first design provides a tangible, reproducible blueprint for any data professional to explore this technology.

For organizations looking to move beyond this PoC, the clear recommendation is to adopt or build upon a dedicated, open-source metadata platform like OpenMetadata. These platforms are built on the same architectural principles demonstrated here but with the scale, features, and robustness required for enterprise environments. The next steps in the roadmap should involve:

1. **Enhanced Lineage Ingestion**: Upgrading the ingestion pipeline to parse compiled dbt SQL to capture column-level lineage and function-level transformations.
2. **Cross-Source Integration**: Expanding the ingestion process to include metadata from other data sources and BI tools to create a truly end-to-end lineage graph.
3. **Production Deployment**: Migrating the architecture to a managed cloud service or on-premise, containerized environment to ensure scalability, reliability, and security.

By following this roadmap, an organization can transform the functional PoC into a strategic asset that improves data literacy, accelerates data-driven decision-making, and establishes a new standard for data governance.

#### Works cited

1. How we made dbt runs 30% faster, accessed August 20, 2025, <https://www.getdbt.com/blog/how-we-made-dbt-runs-30-faster>
2. DBT Integration: The Essential Guide for Data Engineers – Mage AI Blog, accessed August 20, 2025, <https://www.mage.ai/blog/dbt-integration-the-essential-guide-for-data-engineers>
3. How to track and visualize data lineage - Linkurious, accessed August 20, 2025, <https://linkurious.com/blog/how-to-track-and-visualize-data-lineage/>
4. What Is Data Lineage? Tracking Data Through Enterprise Systems - Neo4j, accessed August 20, 2025, <https://neo4j.com/blog/graph-database/what-is-data-lineage/>
5. Data Lineage: Using Knowledge Graphs for Deeper Insights into Your Data Pipelines, accessed August 20, 2025, <https://neo4j.com/blog/knowledge-graph/data-lineage-using-knowledge-graphs-deeper-insights-data-pipelines/>
6. Column-level lineage | dbt Developer Hub - dbt Docs, accessed August 20, 2025, <https://docs.getdbt.com/docs/explore/column-level-lineage>
7. Using Ollama with Python: Step-by-Step Guide - Cohorte Projects, accessed August 20, 2025, <https://www.cohorte.co/blog/using-ollama-with-python-step-by-step-guide>
8. Ollama API Usage Examples - GPU Mart, accessed August 20, 2025, <https://www.gpu-mart.com/blog/ollama-api-usage-examples>
9. Unpacking dbt Artifacts: Powering Transparency in Modern Data, accessed August 20, 2025, <https://hevodata.com/data-transformation/dbt-artifacts/>
10. Manifest JSON file | dbt Developer Hub, accessed August 20, 2025, <https://docs.getdbt.com/reference/artifacts/manifest-json>
11. How to Improve Multi-Hop Reasoning With Knowledge Graphs and LLMs - Neo4j, accessed August 20, 2025, <https://neo4j.com/blog/genai/knowledge-graph-llm-multi-hop-reasoning/>
12. LLM Knowledge Graph - ArangoDB, accessed August 20, 2025, <https://arangodb.com/llm-knowledge-graph/>
13. Azure OpenAI in Azure AI Foundry Models embeddings - Azure OpenAI - embeddings and cosine similarity | Microsoft Learn, accessed August 20, 2025, <https://learn.microsoft.com/en-us/azure/ai-foundry/openai/concepts/understand-embeddings>
14. Query Your Graph with Natural Language - WedaCon IAM Blog, accessed August 20, 2025, <https://blog.wedacon.net/blog/query-your-graph-with-natural-language/>
15. NeoConverse - Graph Database Search with Natural Language - Neo4j Labs, accessed August 20, 2025, <https://neo4j.com/labs/genai-ecosystem/neoconverse/>
16. Getting Started with Neo4j: Installation and Setup Guide - KDnuggets, accessed August 20, 2025, <https://www.kdnuggets.com/getting-started-with-neo4j-installation-and-setup-guide>
17. Local Setup - Introduction to Graph Algorithms in Neo4j 4.x, accessed August 20, 2025, <https://neo4j.com/graphacademy/training-iga-40/05a-iga-local-setup/>
18. Build applications with Neo4j and Python - Neo4j Python Driver Manual, accessed August 20, 2025, <https://neo4j.com/docs/python-manual/current/>
19. Connection - Neo4j Python Driver Manual, accessed August 20, 2025, <https://neo4j.com/docs/python-manual/current/connect/>
20. A dbt artifacts parser in python - GitHub, accessed August 20, 2025, <https://github.com/yu-iskw/dbt-artifacts-parser>
21. dbt-artifacts-parser - PyPI, accessed August 20, 2025, <https://pypi.org/project/dbt-artifacts-parser/>
22. Using forms - Streamlit Docs, accessed August 20, 2025, <https://docs.streamlit.io/develop/concepts/architecture/forms>
23. Input widgets - Streamlit Docs, accessed August 20, 2025, <https://docs.streamlit.io/develop/api-reference/widgets>
24. Understanding how MERGE works - Knowledge Base - Neo4j, accessed August 20, 2025, <https://neo4j.com/developer/kb/understanding-how-merge-works/>
25. MERGE - Cypher Manual - Neo4j, accessed August 20, 2025, <https://neo4j.com/docs/cypher-manual/current/clauses/merge/>
26. Semantic models | dbt Developer Hub - dbt Docs, accessed August 20, 2025, <https://docs.getdbt.com/docs/build/semantic-models>
27. The importance of the semantic knowledge graph - metaphacts Blog, accessed August 20, 2025, <https://blog.metaphacts.com/importance-of-semantic-knowledge-graph>
28. Natural Language Querying of GraphDB in LangChain - Ontotext, accessed August 20, 2025, <https://www.ontotext.com/blog/natural-language-querying-of-graphdb-in-langchain/>
29. Ollama - ️ LangChain, accessed August 20, 2025, <https://python.langchain.com/docs/integrations/providers/ollama/>
30. Creating Web Applications with Flask | PyCharm Documentation - JetBrains, accessed August 20, 2025, <https://www.jetbrains.com/help/pycharm/creating-web-application-with-flask.html>
31. Simple Chatbot application using Python, GoogleAPIKey - GeeksforGeeks, accessed August 20, 2025, <https://www.geeksforgeeks.org/python/simple-chatbot-application-using-python-googleapikey/>
32. Column-Level Lineage (Open-Source Tools Explained) - Galaxy, accessed August 20, 2025, <https://www.getgalaxy.io/learn/glossary/column-level-lineage-open-source-tools-explained>
33. Column level lineage · dbt-labs dbt-core · Discussion #4458 - GitHub, accessed August 20, 2025, <https://github.com/dbt-labs/dbt-core/discussions/4458>
34. Column-Level Lineage - OpenLineage, accessed August 20, 2025, <https://openlineage.io/docs/integrations/spark/spark_column_lineage>
35. Multi-project collaboration · dbt-labs dbt-core · Discussion #6725 - GitHub, accessed August 20, 2025, <https://github.com/dbt-labs/dbt-core/discussions/6725>
36. Explore multiple projects | dbt Developer Hub - dbt Docs, accessed August 20, 2025, <https://docs.getdbt.com/docs/explore/explore-multiple-projects>
37. dbt python sdk · dbt-labs dbt-core · Discussion #7173 - GitHub, accessed August 20, 2025, <https://github.com/dbt-labs/dbt-core/discussions/7173>
38. Graph Embedding For Machine Learning in Python - YouTube, accessed August 20, 2025, <https://www.youtube.com/watch?v=uszt88Z-0Fc&pp=ygUJI25vZGUydmVj>
39. Ingest Lineage from dbt | Official Documentation, accessed August 20, 2025, <https://docs.open-metadata.org/latest/connectors/ingestion/workflows/dbt/ingest-dbt-lineage>
40. Overview - Amundsen, accessed August 20, 2025, <https://www.amundsen.io/amundsen/>
41. OpenMetadata: Join the #1 Open Source Data Community, accessed August 20, 2025, <https://open-metadata.org/>
42. Lineage Ingestion | OpenMetadata Data Lineage Setup Guide, accessed August 20, 2025, <https://docs.open-metadata.org/latest/connectors/ingestion/lineage>
43. dbt Workflow | OpenMetadata Data Build Tool Integration - Collate Documentation, accessed August 20, 2025, <https://docs.getcollate.io/connectors/ingestion/workflows/dbt>
44. Ingest Owner from dbt | Official Documentation, accessed August 20, 2025, <https://docs.open-metadata.org/latest/connectors/ingestion/workflows/dbt/ingest-dbt-owner>