

Knowledge Graph Tools for Rust Codebases

Introduction

Building a **knowledge graph** from a Rust codebase can greatly enhance code exploration, visualization, and LLM-driven Q&A (GraphRAG). This involves parsing the code, extracting semantic relationships (like function calls, type definitions, module dependencies), storing them as a graph, and providing ways to query or visualize these relationships. Below we survey tools and toolchains – from end-to-end solutions to composable components – that support static analysis of Rust (with optional runtime data) and can integrate with graph databases and LLMs. We organize the tools by their role in the pipeline and highlight their capabilities and strengths, followed by a summary table.

End-to-End Code Knowledge Graph Solutions

GitLab Knowledge Graph (GKG) – *Full pipeline solution*. GKG is an open-source tool by GitLab that natively parses multiple languages (including Rust) and builds a “Universal Call Graph” and other code relationships ¹. It uses the high-performance **Kuzu** graph database for storage and querying ², enabling complex queries like cross-references, dependency tracing, and architecture analysis. GKG provides a CLI and a web UI for exploring the graph (e.g. navigate references, visualize architecture) ³. It also supports the **Model Context Protocol (MCP)** for AI integration ⁴, meaning LLM agents (e.g. in an IDE or via a chat UI) can query the code graph to retrieve context. GKG is a comprehensive, developer-oriented platform (currently in public beta) that makes it easy to index a project (`gkg index`) and then query via UI or API for use cases like code navigation, impact analysis, and codebase-aware assistant prompts ⁵. *Strengths*: multi-language support, purpose-built graph engine, out-of-the-box UI, and AI readiness.

CodeGraph (Jakedismo’s GraphRAG) – *Full pipeline solution*. CodeGraph is a Rust-based tool that turns your codebase into a **searchable knowledge graph** with LLM integration ⁶. It parses code in multiple languages using Tree-sitter (optimized for Rust/Python) ⁷ to extract an AST and relations (function calls, imports, etc.), and indexes these in **SurrealDB** (a multi-model DB) as a property graph. It also generates semantic **embeddings** for code chunks and stores them alongside the graph, enabling hybrid *structural + semantic* search ⁸. Queries are served through an MCP-compatible server, so AI tools (Claude, LM Studio, etc.) can ask natural language questions and receive graph-derived answers ⁶. CodeGraph emphasizes performance – Rust for speed, SurrealDB’s HNSW index for vectors (2–5ms query latency) ⁸ – and developer workflow integration via a CLI. It automatically captures dependency graphs and relationships, and supports incremental indexing on code changes ⁹. *Strengths*: high performance, unified graph+vector store, semantic code search, and direct LLM interfacing for GraphRAG.

CodeGraph Analyzer (Neo4j) – *Full pipeline solution (extensible)*. An open-source static analysis engine by Chris Royse, it parses code (currently many languages; Rust support planned) and exports a rich graph into a **Neo4j** database ¹⁰ ¹¹. It builds a “digital twin” of the codebase, identifying nodes for files, functions, classes, variables, etc., and mapping edges for relationships like *CALLS*, *IMPORTS*, *EXTENDS*, *USES* etc. ¹¹. This yields a property graph in Neo4j that developers can query with Cypher or visualize using Neo4j’s tools. CodeGraph Analyzer supports multi-language projects (cross-language links) and integrates with MCP for AI assistants ¹² ¹³. *Strengths*: multi-language, comprehensive

relationship mapping, leverages Neo4j's ecosystem (scalability, visualization), suitable if you prefer a customizable toolchain with a standard graph DB backend.

Sourcegraph (LSIF/SCIP) – *Code indexing & navigation platform*. Sourcegraph's code intelligence platform (while not a traditional graph database) indexes repositories and provides code exploration features akin to a knowledge graph of definitions and references. For Rust, Sourcegraph uses **rust-analyzer** to produce an LSIF/SCIP index of symbols and relationships (via the `rust-analyzer scip` command) ¹⁴. The index is stored and can be queried using Sourcegraph's GraphQL API or UI – allowing “go to definition,” “find references,” dependency graph views, etc. In effect, it creates a graph of code symbols across your codebase. This can be leveraged in retrieval-augmented generation by querying Sourcegraph for relevant code contexts. Sourcegraph has also introduced semantic search with code embeddings. *Strengths*: mature cross-referencer, multi-repo scale, integrates with editors and has an API. (For self-hosted or open usage, LSIF/SCIP data can also be ingested by other systems like Glean, below.)

Composable Toolchain Components

Parsing & Semantic Extraction: Tools in this stage build the AST or semantic model of the Rust code, forming the nodes and edges for the graph. - *Rust Analyzer (Language Server)* – Rust's official language server can output a structured index of the code (SCIP/LSIF format) containing symbols, definitions, references, and relationships in the code. This provides a ready map of function declarations, trait impls, module imports, call references, etc., which can be imported into graph databases or query engines ¹⁵ ¹⁴. It's a reliable static analyzer, aware of Rust's complex name resolution and traits. - *Tree-sitter* – A parser generator and library for building concrete syntax trees. Tree-sitter has a Rust grammar and can parse Rust files quickly to produce an AST. Many custom pipelines (e.g. CodeGraph) use Tree-sitter to iterate over syntax nodes and extract relationships. Tree-sitter is incremental and fast ⁷, and can be embedded in Rust or Python, making it a great choice for building a language-agnostic code graph pipeline. - *Rustdoc JSON* – The Rust compiler's documentation tool (`rustdoc`) can emit JSON with a crate's documentation and item metadata. This JSON includes the **schema of all items (structs, enums, traits, functions, modules, impls, etc.) and links between them**. For example, it lists trait implementations, module contents, and cross-crate dependencies. Tools like Autognosi's MCP server leverage this by parsing rustdoc JSON and building a graph: “*extracting semantic relationships between modules, traits, implementations, and functions. Each entity becomes a graph node... Relationship edges capture dependencies, implementations, and containment hierarchies.*” ¹⁶ ¹⁷. This approach focuses on documented API relationships – useful for framework or library graphs – and can be stored in a lightweight DB (Autognosi uses NetworkX in-memory and SQLite for persistence ¹⁸). - *Syn/AST crates* – Rust's `syn` crate can parse Rust source into a syntax tree (proc-macro style AST). While `syn` gives the syntax, one would still need to resolve semantic info (e.g., name resolution) manually or via `rustc`. In practice, `rust-analyzer`'s index or `rustdoc` JSON are higher-level sources of semantic graph info, whereas `syn` or the compiler's HIR require more work to connect nodes across modules.

Graph Construction & Storage: Once code facts are extracted, these tools help build a graph structure and provide query capabilities. - *Code Property Graph (Joern)* – The **Code Property Graph (CPG)** is a language-agnostic graph model that unifies AST, control-flow, and data-flow graphs for code. **Joern** is an open-source platform that converts source code into a CPG and provides a query DSL for code exploration ¹⁹. In a CPG, nodes represent program entities (methods, variables, etc.) and labeled edges represent relations like *CONTAINS*, *CALLS*, *RETURNS* ²⁰. Joern historically could use Neo4j or other graph DB backends ²¹, but now uses its own optimized store (OverflowDB). While Joern has frontends for many languages, Rust is not officially supported yet. However, one can generate a CPG for Rust by

compiling to LLVM bitcode and using an LLVM2CPG tool (as done in research like RustPräzi ²²). *Strengths*: rich representation (ideal for advanced static analysis or security queries), cross-language querying with a single DSL, and integration with graph databases or custom query engines. It's more low-level and might require custom effort for Rust, but it's a powerful concept for code graphs. - *Kythe* – Google's **Kythe** is a pluggable code indexing framework that extracts cross-reference facts from code and stores them in a graph schema (often backed by a document store or graph query layer). Kythe defines a language-agnostic schema for nodes (called "vnames") like file, package, class, function, and edges like *defines/defines-ref/calls*. It supports C++, Java, Go, etc., and can be extended to new languages by writing an indexer (often by instrumenting the compiler or using an existing index). Rust support in Kythe is not out-of-the-box (early experiments were later removed), but one approach is to use Rust's compiler or rust-analyzer to emit Kythe entries ²³. In essence, Kythe provides the blueprint to build a code knowledge graph and query it (with tools to export to GraphQL or other query APIs). *Strengths*: open schema, proven at Google-scale for cross-language code search, but would require a Rust indexer or adapting a format like LSIF to Kythe's schema. - *Glean (Meta)* – **Glean** is a fact database and query system for code, open-sourced by Meta. It collects facts about code structure in a variety of languages and allows querying them with a Datalog-like query language ²⁴. Glean has native indexers for C++, Java, etc., and it supports ingesting LSIF/SCIP indexes for others. In particular, **Rust is supported via rust-analyzer's LSIF output** ²⁵ – you run `rust-analyzer scip` and import the resulting index into Glean. Glean stores code facts in a specialized database and can derive new facts (e.g. transitive references) via logic rules. It's used to power large-scale code search and "find all references" at Meta. *Strengths*: very scalable, uniform query interface across languages, and can integrate multiple languages' data. However, it's more of a back-end engine; you'd need to set up its query UI or integrate it into an IDE/LLM pipeline yourself.

- *Graph Databases & Query Engines*: Instead of a specialized code graph system, one can use general **graph databases**. For example, **Neo4j** (with Cypher query) is popular for custom code graphs – e.g., one can parse Rust code and insert nodes for functions or modules and relationships for calls or imports, then query with Cypher for paths. Neo4j has the advantage of powerful queries and visualization tools (Neo4j Browser and Bloom). Other graph stores like **SurrealDB** (used by CodeGraph) combine graph and vector capabilities, which is useful for GraphRAG (store code relationships and embeddings together) ⁸ ²⁶. SurrealDB natively supports graph traversals and very fast vector similarity search in one engine ²⁷. There's also **Kuzu** (used in GKG) – an embedded, **GPU-accelerated graph DB** optimized for fast graph analytics on a single machine ²⁸. If a semantic web/RDF approach is preferred, frameworks like RDF triplestores or GraphQL-based query layers can be used, but those are less common for code modeling. The choice of storage may depend on the scale and query complexity: for instance, Kuzu and Neo4j excel at complicated path queries, while SurrealDB or TigerGraph can handle large graphs with high query throughput.

Visualization & Exploration: Visualizing a code knowledge graph helps developers understand architecture and trace relationships. - *Built-in UIs*: Some end-to-end tools come with UIs – **GitLab GKG's web interface** lets you interactively explore the code graph (e.g., view dependency diagrams or navigate call graphs in the browser). **Sourcegraph's web UI** similarly allows clicking through definitions and references across repositories. These are tailored for developer exploration. - *Graph Visualization Tools*: If using a general graph DB like Neo4j, you can use its visualization or plugins to create diagrams of the code relationships. Neo4j Bloom, for example, can produce visual graphs of nodes/edges matching a query (e.g., a subgraph of a module and its function calls). For quick visuals, one can also export portions of the graph to **Graphviz DOT** format. Tools like `scip-callgraph` demonstrate this by generating a DOT file and SVG image of a Rust project's call graph ¹⁵ ²⁹. This is useful for documentation or high-level understanding (e.g., architecture charts). There are also graph libraries (like `networkx` in Python or D3.js in web apps) if you want a custom visualization of the code graph. - *IDE Integration*: Bringing the graph into the IDE can boost productivity. GKG mentions integration into IDEs/editors ³ – likely via its CLI or MCP, one could highlight a function and query the graph for its

references or dependencies. Sourcegraph provides IDE plugins for inline code navigation. In principle, an MCP-based server (as in CodeGraph or GKG) can serve as a backend for editor extensions that show on-demand graphs (e.g., “show me a call graph of this function” in VSCode).

LLM Integration (GraphRAG): A key use-case is to let LLMs utilize the graph for smarter code assistance. - *Model Context Protocol (MCP):* This emerging standard (introduced by Anthropic) defines a unified way for AI agents to query external tools. Both CodeGraph and GKG implement MCP servers ⁶ ⁴, exposing the code knowledge graph through standardized endpoints that an AI assistant can call (either via HTTP or stdio). For example, an LLM can ask “Find all references to function X” and the MCP server responds with graph query results, which the LLM can use in its answer. MCP is essentially a **“universal API”** that negates custom integration for each model ³⁰, making tools plug-and-play for any MCP-enabled client. This is a powerful way to do GraphRAG: the graph storage handles retrieval and the LLM focuses on reasoning with that structured context. - *Graph Query APIs:* Without MCP, one can still integrate graphs by using a query API. Many graph DBs have HTTP REST or GraphQL endpoints. For instance, a Rust knowledge graph in Neo4j can be queried via a REST call for Cypher. One could incorporate this in an LLM chain (e.g., using a LangChain GraphQA tool to let the LLM formulate Cypher queries to fetch code relationships). Sourcegraph’s GraphQL API is another way – an LLM could be given access to search code or lookup references via Sourcegraph’s API. The key is that the graph can answer structured queries (like “which functions call X?”) more reliably than text search. - *Embedding + Graph Hybrid:* Some solutions embed code to allow semantic search when structural search falls short. CodeGraph’s approach is to retrieve both by graph traversal and by vector similarity ⁸. For example, to answer “How is data validated before database insert?”, an agent might find relevant functions via semantic search on code comments, then use graph links to trace how those functions connect to DB insert calls. The combination of embedding-based RAG and graph-based RAG (GraphRAG) can significantly improve the relevance of context provided to the LLM ³¹ ³².

Summary of Tools and Capabilities

The following table summarizes each tool or framework, its key capabilities, strengths, and how it fits into a Rust code→graph pipeline:

Tool / Framework	Capabilities & Pipeline Role	Strengths and Notes
GitLab Knowledge Graph (GKG) ¹ ⁴	End-to-end solution: Parses Rust (and other languages) via <code>gitlab-code-parser</code> ; builds a call graph and dependency graph stored in Kuzu (graph DB); provides fast queries (Cypher-like) and a web UI + CLI for exploration; offers MCP API for LLM integration.	Multi-language support; purpose-built graph engine (Kuzu) for complex queries ²⁸ ; integrated UI for devs; AI-ready (MCP) for GraphRAG use ⁴ . In public beta (open-source).

Tool / Framework	Capabilities & Pipeline Role	Strengths and Notes
CodeGraph (Jakedismo) 6 8	End-to-end solution: 100% Rust implementation. Uses Tree-sitter to parse code (Rust, Python, etc.) into a knowledge graph; stores graph in SurrealDB (nodes/edges as well as vector embeddings); supports semantic search and structured queries. Runs an MCP server so LLMs/agents can query project knowledge.	High performance (Rust + SurrealDB HNSW index) 8 ; hybrid search (structural + semantic) across the codebase; automatic dependency and reference graph extraction; CLI and programmatic access. Designed for GraphRAG scenarios (project-aware AI assistants).
CodeGraph Analyzer (Neo4j) 10 11	End-to-end (extensible): Static analyzer that parses many languages (Rust support planned) into a property graph in Neo4j . Captures rich relations (calls, imports, extends, etc.) and allows Cypher queries and visualization via Neo4j tools. Integrates with MCP for AI.	Handles cross-language projects (unified graph); leverages Neo4j's mature ecosystem (scalability, security, visualization); "two-pass" analysis resolves cross-file links for completeness. Good for custom pipeline builders who want a ready parser-to-graph solution with a standard DB.
Rust Analyzer + LSIF/SCIP 15 25	Parsing/Indexing component: Rust Analyzer can generate an LSIF/SCIP index containing the Rust code's semantic graph (definitions, references, types). This index can be loaded into code intelligence platforms or converted to graph form. Often used with Sourcegraph or Glean.	Official Rust tool – accurate and up-to-date on language features. Yields a comprehensive map of symbol relationships. Strength: Easy way to get structured data without writing a parser. The LSIF data can be queried via tools like Sourcegraph's GraphQL or imported into Glean 25 for further analysis.
Tree-sitter (Rust grammar) 7	Parsing component: General-purpose parser library for code. Produces concrete syntax trees for Rust quickly, enabling custom extraction of nodes and edges. Often the first step in custom pipelines (you traverse the AST to add graph nodes for functions, etc.).	Robust and incremental parsing, with broad language support – ideal for building multi-language graphs. Strength: Can parse incomplete code (useful for live analysis) and is embeddable in many environments. Requires custom logic to interpret the AST into semantic relations (e.g., resolve identifiers).

Tool / Framework	Capabilities & Pipeline Role	Strengths and Notes
Joern / Code Property Graph ¹⁹ ²¹	Graph construction & query: Converts source code into a Code Property Graph (AST + control/data flow). Offers a Scala-based DSL (CPGQL) to query patterns (e.g., find all call paths to a given function). Can export or interface with graph databases (older versions used Neo4j). Rust integration via LLVM bitcode (no direct frontend).	Strengths: Rich static analysis representation (captures more than just call graphs – also data flows, CFG, etc.); cross-language querying with one query language. Useful for advanced analysis or vulnerability research ¹⁹ . However, direct Rust support is DIY (e.g., compile to LLVM IR and import) and the tool is heavier to set up.
Kythe	Graph framework: Defines an open schema for code graphs and indexers to populate it. Emits a graph of “facts” (nodes for symbols with edges like <i>ref/call/defines</i>) which can be stored and queried (often via GraphQL or a custom query tool). No built-in Rust indexer, but Rust code can be indexed by adapting compiler outputs to Kythe’s schema.	Strengths: Language-agnostic and proven on large codebases (Google). It’s highly extensible – if a Rust indexer is implemented, you get cross-repo and cross-language linking for free. Typically requires integration with build systems (e.g., Bazel) to extract data. Good for those looking to adopt a standard schema and possibly integrate multiple languages’ graphs.
Glean (Meta) ²⁵	Graph storage & query: A fact database that stores code relations and allows Datalog-like queries. Ships with indexers for some languages and supports Rust via LSIF import ²⁵ . Glean can answer questions like “what functions call X” or “where is type Y defined” by querying the stored facts. Often used headless (via APIs or a query console), but one can build tools on top.	Scales to massive codebases (designed for Meta’s monorepos); flexible query logic (you can compose queries to traverse relationships arbitrarily). Strength: Unifying data from different languages or analysis passes. The downside is the setup complexity and lack of a friendly UI by default – it’s a powerful back-end for those willing to script queries or integrate into other tools.
CodeQL (GitHub) ³³	Static analysis DB: Not a graph DB per se, but CodeQL builds a relational database of code and allows writing queries in a declarative QL to explore it. Now supports Rust (in preview) ³³ . Developers can query for data flows, inheritance, call trees, etc., using the provided libraries. It’s mainly used for security scanning, but the database it creates can serve as a knowledge base of the code.	Strengths: Deep semantic analysis (e.g., it understands taint flows, type hierarchies); comes with a huge library of standard queries. Can find complex patterns across a codebase. It requires code to be compiled to create the DB, and queries need to be written in CodeQL’s language. Best suited if your use case involves <i>searching for specific patterns or paths</i> (and you might trade off interactive visualization for powerful analysis).

Tool / Framework	Capabilities & Pipeline Role	Strengths and Notes
Visualization Tools (GraphViz, Neo4j Bloom, etc.)	Visualization: These are not code-specific but worth noting. GraphViz/DOT can render graphs of functions or modules (some tools auto-export to DOT for call graphs ²⁹). Neo4j Bloom/Browser let you explore the graph with custom queries and visual node-link diagrams. GKG's built-in UI and similar dashboards also fall here.	Visual tools help in architecture understanding and developer onboarding by turning the graph data into intuitive diagrams. While not stand-alone solutions, they complement the above tools: e.g., using Bloom on a Neo4j code graph, or embedding a GraphViz diagram in docs. Choose based on your stack – many graph databases come with some visualization capability out of the box.
LLM Integration Mechanisms (MCP, GraphQL APIs)	Integration: This row highlights how tools connect to LLMs. MCP (Model Context Protocol) is supported by CodeGraph and GKG, enabling direct natural language queries on the graph ⁴ ⁶ . Other tools might be used via GraphQL/REST API (e.g., Sourcegraph's API or a custom service that runs Cypher queries on Neo4j). In all cases, the integration provides a retrieval layer for GraphRAG.	Strengths: MCP is a standardized, real-time channel – it can turn complex graph queries into conversational answers ³⁴ . GraphQL/REST approaches require more manual prompt engineering (the LLM needs to formulate the right query), but are viable with tools like LangChain agents. Ensure whichever tool you pick, it has an accessible interface for the LLM (be it an API, SDK, or MCP compliance) to realize the GraphRAG use case.

Sources: The information above is drawn from official documentation and developer resources for each tool, including GitLab's Knowledge Graph docs ¹ ⁴, the CodeGraph repository and analysis articles ⁶ ⁸, Joern's documentation ¹⁹, Glean's documentation ²⁵, and others as cited. Each tool addresses a slice of the pipeline – from parsing to graph databases to AI integration – and they can often be combined. For example, one might use rust-analyzer/SCIP to generate a base index, import it into Neo4j or Glean for querying, and then use an MCP server to let an LLM access those queries. Prioritizing an end-to-end solution (like GKG or CodeGraph) can jump-start project-aware code assistance, while a custom toolchain allows tailoring to specific needs (e.g., incorporating runtime data or specialized analysis). The landscape is evolving quickly, but the tools listed here represent the state-of-the-art for **creating a knowledge graph from Rust code** and leveraging it for developer insights and intelligent assistants. ⁶ ⁴

1 2 3 4 5 28 **GitLab Knowledge Graph | GitLab Knowledge Graph**

<https://gitlab-org.gitlab.io/rust/knowledge-graph/>

6 8 9 26 27 **GitHub - Jakedismo/codegraph-rust: 100% Rust GraphRAG implementation with a MCP server for indexing large codebases with blazingly fast speed and querying the graph with natural language**

<https://github.com/Jakedismo/codegraph-rust>

7 30 31 32 **CodeGraph by Jakedismo: The High-Performance AI Tool Redefining Code Analysis?**

<https://skywork.ai/skypage/en/codegraph-ai-tool-code-analysis/1980487274945171456>

10 11 12 13 **GitHub - ChrisRoyse/CodeGraph**

<https://github.com/ChrisRoyse/CodeGraph>

14 15 29 **GitHub - Beneficial-AI-Foundation/scip-callgraph**

<https://github.com/Beneficial-AI-Foundation/scip-callgraph>

16 17 18 34 **Knowledge Graph MCP Server: Semantic Research for Rust Documentation | by Carlo C. | Medium**

<https://autognosi.medium.com/knowledge-graph-mcp-server-semantic-research-for-rust-documentation-f70995757c6a>

19 20 21 **Code Property Graph | Joern Documentation**

<https://docs.joern.io/code-property-graph/>

22 **Announcing RustPräzi: a tool to build an entire call graph of crates.io - announcements - The Rust Programming Language Forum**

<https://users.rust-lang.org/t/announcing-rustprazi-a-tool-to-build-an-entire-call-graph-of-crates-io/22696>

23 **Kythe is a pluggable, (mostly) language-agnostic ... - GitHub**

<https://github.com/kythe/kythe>

24 **Indexing code at scale with Glean - Engineering at Meta - Facebook**

<https://engineering.fb.com/2024/12/19/developer-tools/glean-open-source-code-indexing/>

25 **GitHub - facebookincubator/Glean: System for collecting, deriving and working with facts about source code.**

<https://github.com/facebookincubator/Glean>

33 **Supported languages and frameworks - CodeQL - GitHub**

<https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/>