

Software-Analyse mit Knowledge-Graphen und LLM

Andres Koch (akoch@objeng.ch), Remo Koch (rkoch@objeng.ch)

Object Engineering GmbH, Birmensdorferstrasse 32, CH-8142 Uitikon-Waldegg, <https://objeng.ch>

Abstract

Software-Reengineering beginnt typischerweise mit der Analyse bestehender Code-Basen, da in der Praxis meist auf einem Brownfield- und nicht auf einem Greenfield-Ansatz operiert wird. Angesichts der allgegenwärtigen KI-Diskussion liegt es nahe, sämtliche vorhandenen Artefakte einem geeigneten LLM zu übergeben und daraus Analyse-Ergebnisse abzuleiten. Da diese Code-Basen oft > 1 Mio. LOC umfassen, übersteigen sie die Kapazitätsgrenzen eines LLMs. Das Einlesen in einen Knowledge-Graphen ist hingegen ein deterministischer, wiederholbarer Vorgang. Die gezielte Kombination beider Ansätze liefert überzeugende Analyse-Resultate und ermöglicht Abfragen in natürlicher Sprache.

Einführung

In [1] wurde bereits demonstriert, wie sich Code-Basen in einen Graphen überführen und analysieren lassen. Knowledge-Graphen eignen sich besonders für deterministische Metadaten-Modelle – insbesondere wenn Daten für spätere Versionen erneut eingelesen werden sollen. LLMs hingegen stossen bei grossen Code-Beständen schnell an Speicher- und Kostengrenzen.

Die Kombination beider Ansätze erschliesst eine neue Dimension analytischer Leistungsfähigkeit: der Knowledge-Graph liefert die stabile, deterministische Wissensgrundlage, das LLM übernimmt Suche, Analyse und Report-Generierung.

Vorgehen

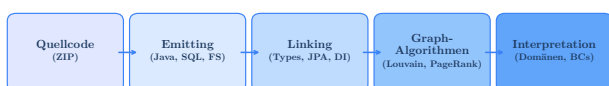


Abbildung 1: Analyse-Pipeline: Vom Quellcode über Emitting und Linking zum Property Graph

Der typische Prozess für das Analysieren einer Software sieht folgendermassen aus:

1. Einlesen der Programm-Artefakte und Überführen in den Graphen durch spezialisierte Import-Emitters.
2. Systematisches Durchgehen der gespeicherten Artefakte und Verbindungen untereinander erstellen.
3. Graphen-Algorithmen in Anwendung bringen, welche den Graphen weiter ergänzen → Knowledge Graph.

4. Interpretation des Knowledge-Graphen durch den Analysten mit Hilfe eines geeigneten LLM.

Knowledge-Graphen aufbereiten

Bestehende Code-Artefakte – dazu zählen nicht nur Quellcode, sondern auch sämtliche Metadaten wie Konfigurationsdateien oder Datenbankmetadaten – können mit geeigneten Parsern und Scannern eingelesen und nach einem definierten Schema in einer Graphen-Datenbank abgelegt werden [1]. Umfang und Detaillierungsgrad spielen dabei in der Regel keine einschränkende Rolle, sondern werden aufgrund des Anwendungsfalles bestimmt, genau so wie das Schema der Graphen-Datenbank sich flexibel auf den jeweiligen Anwendungsfall abstimmen lässt.

Nach dem Einlesen und Speichern der Software-Artefakte in der Graphen-Datenbank, laufen weitere deterministische Algorithmen ab, welche die Verbindung der Knoten mit anderen abhängigen Knoten herstellen.

In einem weiteren Schritt (4) können Graphen-Algorithmen auf den Knowledge-Graphen angewendet werden, welche diesen weiter mit Informationen ergänzen. Je nach Anwendungsfall werden Algorithmen wie Louvain/Leiden Community, Triangle Count, Page Rank u.a. [2] eingesetzt.

Optional lassen sich Graph-RAGs (vektorierte Artefakt-Abbildungen) für schnelle Ähnlichkeitssuchen erzeugen; für ein LLM ist Wissen im Graphen bereits gut bearbeitbar.

LLM-Zugriff

Der Zugriff eines LLMs auf den Knowledge-Graphen kann über die MCP-Schnittstelle erfolgen (Model Context Protocol), welche spezifisch für LLM definiert wurde. Die vorbereiteten Datenstrukturen ermöglichen effiziente Abfragen, ohne dass das LLM aufwändige Konvertierungen vornehmen muss. Das Modell findet den relevanten Kontext bereits aufbereitet vor und kann seine Erkenntnisse zudem direkt im Knowledge-Graphen persistieren. Die Neo4J-Graphdatenbank bietet eine MCP-Schnittstelle, die aus KI verwendet werden kann.

Als Architektur wählten wir für unsere Analyse-Pipeline den *Agentic AI*-Ansatz (Abb. 2): Wir stellen dem LLM (Claude Code) unsere Toolbox als aufrufbares Tool-CLI inklusive dem Zugriff auf den Knowledge-Graphen zur Verfügung. Das LLM kann damit das Schema des Graphen abfragen und zusätzliche Erkenntnisse – z.B. Attribute für bestimmte Knoten und Kanten – direkt persistieren.

Auf diese Weise lässt sich der direkte Code-Zugriff für das LLM vermeiden, was wünschbar ist, wenn

vom Kunden Einschränkungen auferlegt werden. Der Zugang erfolgt dann ausschliesslich über die strukturierten Metadaten.

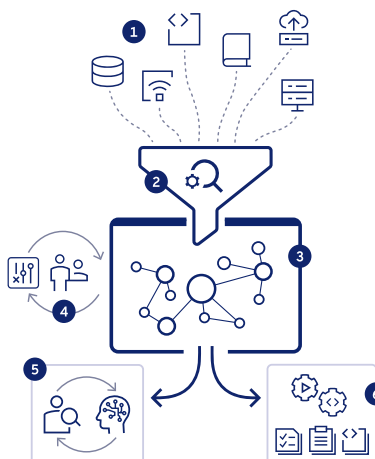


Abbildung 2: Tooling-Framework: 1. Import, 2. Parsing/Scanning, 3. Knowledge Graph, 4. Enhancement, 5. LLM Access, 6. Report-, Artefakt-Generierung

Graph in natürlicher Sprache befragen

Ein wesentlicher Vorteil dieses Ansatzes ist, dass Anwender ihre Fragen in ihrer eigenen Sprache und Denklogik formulieren können. Das LLM übersetzt diese automatisch in die Abfragesprache des Knowledge-Graphen und führt bei Bedarf kombinierte Abfragen durch oder instrumentalisiert unterschiedliche Werkzeuge des Knowledge-Graphen-Systems.

Dies ist von grossem Vorteil, da das LLM die Cypher-Queries für den Graphen selber erstellt, was sonst nicht jedermanns Sache ist. Die Überprüfung hat ergeben, dass diese Queries qualitativ sehr gut und expertenwürdig sind.

So lassen sich beispielsweise Fragen zu Modulgrenzen oder komplexen Abhängigkeitskonstellationen stellen, die bei einer direkten, manuellen Graph-Abfrage unverhältnismässig aufwändig wären [3]. Neben der interaktiven Verwendung des LLM durch Analysten oder Architekten sind natürlich auch Analyse-Berichte gefragt.

Praxis-Erfahrung

Das geschilderte Verfahren haben wir vorerst für Presale-Aktivitäten von Partner und zu Beginn von Projekten eingesetzt, um entweder fundierte Aufwandschätzungen, Risiko-Abklärungen und Entscheidungsgrundlagen für die Migrations-Architektur zu finden.

Interessant sind vor allem Wissenslücken oder falsche Erwartungen, welche gefühlsmässig bei manueller Analyse gemacht werden können. Hier drei Beispiele, welche dank LLM und dem Knowledge-Graphen gefunden wurden:

- Bei der Analyse einer industriell eingesetzten Delphi-Applikation, wurde vom LLM aufgezeigt, dass der Kunde auch nicht produktiv eingesetzte Applikation-Artefakte abgegeben hat, welche nicht relevant waren.
- Bei einer weiteren, ebenfalls industriell eingesetzten Applikation für optimierten Blechschnitt, bestand der Optimierungs-Algorithmus aus mehreren Tausend LOCs in komplex aussehendem Code (mit etlichen GOTOs). Während die Entwickler von der anscheinenden Komplexität beeindruckt waren, konnte das LLM aufzeigen, dass Code zwar einen Backtracking-Algorithmus „vorgab“ aber dieser nicht verwendet wurde und die angebliche Komplexität einfach aufzulösen war.
- In einer dritten grösseren Applikation (Java mit rund 1000 Klassen rund 300 KLOCs und Datenbank mit knapp 100 Tabellen) konnten Fragen des Kunden gezielt beantwortet und Modularisierungsmöglichkeiten, Domänen-Aufteilung und Risiko-Aufzeichnung sowie Architektur und Modul-Strukturen aufgezeigt werden.

Fazit

- Erste Praxiserfahrungen mit der Kombination von LLMs und Knowledge-Graphen haben die Leistungsfähigkeit dieses Ansatzes bei Abfrage und Auswertung eindrücklich belegt. Insbesondere, dass dieser Kombinations-Ansatz Lücken und Eigenschaften von Software-Artefakten aufzeigt, welche bei der manuellen Verwendung des Knowledge-Graphen möglicherweise untergegangen wären.
- Die Zuverlässigkeit der LLM-Antworten steigt spürbar mit der Qualität des zugrunde liegenden Knowledge-Graphen und liefert bemerkenswert präzise Ergebnisse.
- Die automatische Report-Generierung entlastet Analysten erheblich; LLM-generierte Reports neigen jedoch zu übermässigem Umfang und erfordern gezielte Instruktionen oder manuelle Nachbearbeitung.
- Mit wachsender Praxiserfahrung kann das Verfahren weiter – z.B. mit GraphRAGs – ausgebaut werden.

Referenzen

- [1] A. Koch und R. Koch, „Metadaten basiertes, teil-automatisiertes Software-Reengineering“, in *WS-RE 2018*, 2018.
- [2] M. Needham und A. E. Hodler, *Graph Algorithms: Practical Examples in Apache Spark & Neo4J*. O'Reilly, 2019.
- [3] A. Koch und R. Koch, „Vom Monolithen zur Service-Architektur mit Hilfe von Graphen“, in *WSRE 2020*, 2020.