AutoKGen: Autonomous Knowledge Graph Construction through Agentic AI-driven Code Generation and Ontology-based Validation

Giorgos Anagnostou[1,*], Konstantinos Kotis[1]

[1]Intelligent Systems Lab, Dept. of Cultural Technology and Communication, University of the Aegean, 81100, Mytilene, Greece

Abstract

Building a knowledge graph (KG) from scratch traditionally requires a multi-disciplinary team for data engineering, ontology interpretation, ETL development, and graph database administration, a combination that makes KG construction resource-intensive in both cost and time. This paper investigates whether an LLM agent, given only raw data files and a target ontology, can autonomously perform every KG construction-based engineering task without human-in-the-loop (HITL) supervision during execution. We present AutoKGen, an open-source system in which natural-language prompts serve as the source of truth, while the LLM writes, executes, and self-corrects generated TypeScript pipeline code, that is discarded after use. A four-step compliance model validates output at the mapping, generation, post-import, and SHACL stages, halting or correcting the pipeline whenever it deviates from the reference ontology, and a purpose-built Neo4j MCP server exposes graph database operations as structured tool calls. We evaluate AutoKGen across three heterogeneous domains: MoMA (157K records, EDM), DBLP (12.3M records, FaBiO with 261 classes), and GeoNames (790K records, GeoSPARQL), achieving ontology compliance scores of 95%, 82%, and 70%, and SHACL conformance above 98% over the generated nodes, without changes to the prompts. A cross-provider comparison shows Claude Code completes all three domains end-to-end, while OpenAI Codex succeeds only on the smallest dataset. A complementary WebUI exposes the resulting graph through a natural-language chat interface, allowing non-technical users to query the constructed KG conversationally. These results demonstrate that, paired with formal guardrails and structured tool access, LLM agents can autonomously construct validated KGs across diverse domains, effectively removing the HITL requirement from KG construction.

Keywords

Knowledge Graph generation, Large Language Models, Autonomous Code Generation, Ontology Compliance, SHACL Validation, Model Context Protocol, Agentic AI

CEUR-WS.org/Vol-4335/paper2.pdf

1. Introduction

Knowledge graphs (KGs) facilitate the use of structured, interconnected information across domains from biomedicine to cultural heritage, geospatial intelligence, and digital libraries [3]. By encoding entities as nodes and relationships as typed edges, constrained by formal ontologies, KGs enable semantic search, automated reasoning, link prediction, and explainable question answering. However, constructing a KG from raw data remains a significant task, as experts must understand the source data, design or adopt an ontology, write extraction and transformation logic, load the resulting triples or property graph, and validate against ontologies, each step requiring specialized expertise.

Existing automation efforts address certain parts of this pipeline. Recent research indicates that injecting LLM-based capabilities into declarative mapping pipelines [1] can automate rule generation, without proceeding to the mapping stage, leaving cleaning, importing and validating outside of scope. Additionally, recent work that applies LLMs to KG sub-tasks, such as entity typing and ontology alignment [2, 3] primarily identified these practices as components inside a human-orchestrated workflow, rather than as autonomous agents. Agentic LLM frameworks [4], where a model can plan multi-step

Proceedings of the 5th International Workshop on Natural Language Processing for Knowledge Graph Construction (NLP4KGC 2026), co-located with the 25th International Semantic Web Conference (ISWC 2026), Bari, Italy, October 26, 2026

*Corresponding author.

giorgos.anagnostou@aegean.gr (G. Anagnostou); kotis@aegean.gr (K. Kotis)

https://github.com/developedbygeo (G. Anagnostou)

0009-0006-1989-793X (G. Anagnostou); 0000-0001-7838-9691 (K. Kotis)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

tasks, generate and execute code, and invoke external tools, allow for further automation, with the primary goal being the delegation of the entire pipeline to an LLM agent. This requires structured database access, for which the Model Context Protocol (MCP) [5, 6] provides an intuitive interface, and on-demand generation of domain-adapted pipeline code, which modern LLMs paired with a robust execution environment can potentially satisfy.

We present AutoKGen, a system that allows an LLM agent to construct and consume ontology-compliant KG from structured data. The system consists of three subsystems: 1) an Neo4j MCP server that exposes graph operations as MCP tools, 2) an autonomous eight-step pipeline that constructs KGs, driven entirely by natural language prompts through LLM-generated TypeScript code [7] and 3) a complementary chat interface that consumes the resulting graph through natural language.

The aim of our work is to investigate whether an LLM agent, given only raw data files and a reference ontology, can autonomously perform every KG construction task, without human-in-the-loop (HITL) supervision. The contributions and outcomes of this work demonstrate that, paired with formal guardrails and structured tool access, LLM agents can autonomously construct KGs across diverse domains, effectively removing the HITL requirement from KG construction, using prompts, as the source of truth.

To achieve this goal and obtain the targeted output of our research, we evaluate AutoKGen across three heterogeneous domains, with a cultural dataset from MoMA [8] (157K records, EDM), a dataset from the Digital Bibliography & Library Project (DBLP) [9] (12.3M records, FaBiO with 261 classes), and a GIS dataset, GeoNames [10] (790K records, GeoSPARQL), producing compliant graphs across all three without modifying the source-of-truth initial prompts.

The structure of the paper is as follows: Section 2 reviews related work across KG construction, LLMs for KG sub-tasks, agents and tool usage, the MCP and its uses, LLM-based code generation and ontology validation. Section 3 describes our methodology, system architecture, the core pipeline, and the chatbot and MCP server. Section 4 examines the experiments and the respective results, while Section 5 refer to the limitations, potential future work and conclusion respectively.

2. Related work

AutoKGen draws on five lines of prior work. Knowledge graph construction pipelines define the methodology we automate. LLMs and KG tasks (2.1) demonstrate competence on discrete sub-tasks. Agents, tools and the MCP (2.2) provide the runtime that lets an LLM act autonomously on a database, LLM-based code generation (2.3) produces data-specific transformations on demand, and natural-language querying with ontology validation (2.4) dictates how the resulting graph is consumed and verified. AutoKGen integrates them into a single agent-driven pipeline.

Declarative mapping languages such as R2RML [11] and RML [12] remain the dominant approaches for transforming structured data into knowledge graphs, as Hofer et al. [1] use LLMs to zero-shot auto-configure RML mapping over a reference ontology, Arenas-Guerrero et al. [13] introduce RML Views to streamline tabular-to-RDF transformation, and FlexRML [14] provides a memory-efficient materializer for large-scale graphs. However, these are isolated, as cleaning, post-import validation, and pipeline orchestration remain outside their scope. A 2025 survey by Bian [15] documents how LLM rule-based extractors are giving way to language-driven, generative frameworks. Two paradigms have emerged, with top-down approaches that first generate competency questions and a schema, and bottom-up approaches that create schema directly from data. Microsoft's GraphRAG [16, 17] popularized LLM-driven graph construction at scale but operates on unstructured text and does not enforce ontology compliance. Neo4j's LLM Knowledge Graph Builder [18] shares both limitations. AutoKGen falls within the top-down paradigm, with the schema preceding extraction and constraining it. Unlike top-down methods that synthesize the schema from competency questions, AutoKGen takes a pre-existing ontology as input and treats it as the contract that every downstream stage, from profiling to SHACL validation which is respected, with each stage being automated via prompt-driven code generation rather than direct LLM extraction.

2.1. LLMs and KGs

The dynamic between LLMs and KGs, along with the opportunities the latest updates pose, were surveyed by Pan et al. [19] to examine their intersection and to distinguish KG-enhanced LLMs, LLM-augmented KGs, and synergized frameworks. Within LLM-augmented construction, NeOn-GPT [20] provides a prompt-driven workflow integrating ontology reuse and adaptive refinement, while OntoGenix [21] generates ontologies from tabular data. The LLMs4OM framework [3] and the LLMs4OL 2025 challenge [2] benchmark ontology matching and ontology learning across heterogeneous schemas. These efforts target individual sub-tasks rather than end-to-end construction.

2.2. Agents, tools and the MCP

Recent research by Qu et al. [22] indicates that agents can successfully organize tools into task planning, tool selection, execution and generation to efficiently approach complex problems, while Plaat et al. [4] surveyed how reasoning models, tool integration and agent-based modeling can further enhance their capabilities. The intersection of this research aligns with the development of the MCP as a means of tool discovery and invocation through a JSON-RPC [23] client-server architecture, based on the Language Server Protocol [24]. The adoption of MCP has been rapid, with OpenAI and several other service providers adopting it [25, 26, 27, 28]. Research has also analyzed the security and maintainability of MCP servers [5], benchmarking LLM performance when invoking MCP tools [29, 30] and utilizing the protocol across different domains for diverse tasks.

2.3. LLM-based Code Generation

Code generation has been surveyed by Jiang et al. [31] regarding code completion and autonomous coding agents that generate, debug, test and deploy code. Semantically closer to AutoKGen is the Prompt2DAG [32] which transforms natural language into executable code for Apache Airflow DAGs, as it shares the prompt-driven specification.

2.4. Ontology Validation and Natural Language Querying

Natural language to Cypher, as an LLM-backed practice has been recently presented by Yang et al. [33], highlighting the importance of high-quality training data and schema-aware prompting, while CypherBench [34] examined how the LLM context window, could hinder Cypher generation, with their top-performing model achieving execution accuracy of 61.58%. One of the core challenges is schema alignment, as LLMs can hallucinate types or relationships. In terms of ontology compliance, Shapes Constraint Language (SHACL) [35] is considered the standard, and recent research by Cortés et al. [36] provide thorough validation, while requiring pre-defined shapes.

3. Methodology

The methodology focuses on AutoKGen's core module, the pipeline which is autonomous from Step 1 to Step 8, as it transforms heterogeneous structured data into ontology-compliant KGs without human intervention, beyond providing input files, the reference ontology and the SHACL validation files (optional). The methodology is based on three core components, a) an LLM agent that plans and executes the pipeline, b) a custom Neo4j MCP server that exposes graph database operations as structured tool calls, which is also shared by both the construction and consumption phases and c) a four-stage compliance model that constrains every output to the reference ontology. This section describes the system architecture, the eight-step KG construction pipeline, the MCP-based database interface, the compliance and validation strategy, and the WebUI that consumes the KG.

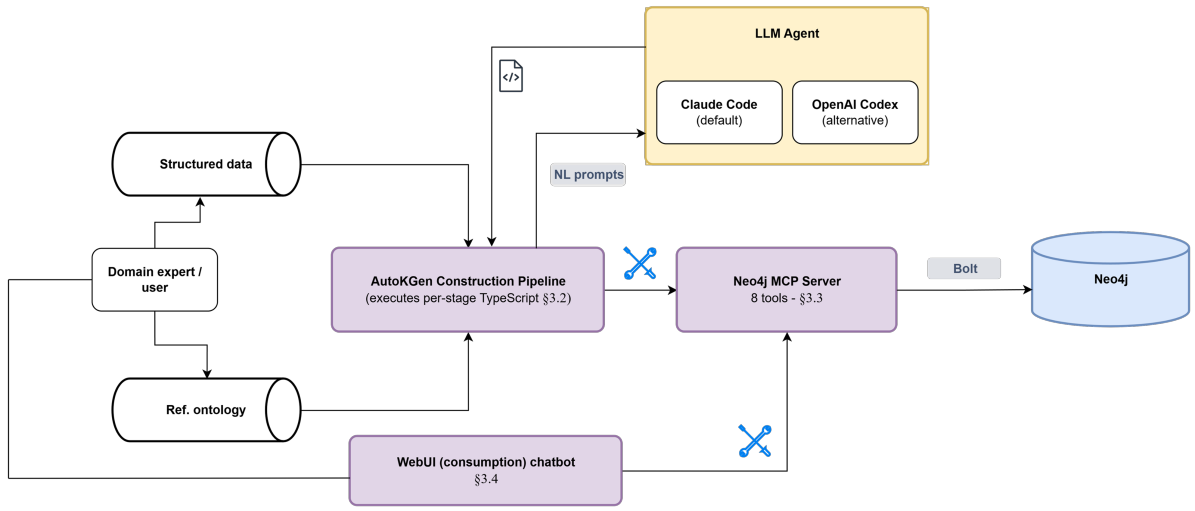


Figure 1: AutoKGen Full System Architecture

3.1. System Architecture: Introducing the Pipeline

AutoKGen's pipeline[1] follows a prompt-as-the-source-of-truth design. Each step is effectively a bash script that loads the runtime environment and dispatches a natural language specification to the agent. The agent reads the specification, along with prior outputs and generates TypeScript code [7] that is tailored to the current domain and data shape. The code is then executed against the filesystem and the Neo4j MCP server.

The benefit of this separation is threefold. First, domain adaptation is automatic, as the same prompts produce different code for each dataset, since the agent observes the ontology structure and data profile at generation time and adjusts the implementation strategy. Second, what the pipeline does is decoupled from how it does it, as the prompt encodes the specification and the agent determines the implementation. Third, the pipeline is agent-driven, as the agent does not invoke pre-written tools but rather, it authors the tools itself in each stage, treating the ontology and the dataset as the contract to satisfy.

Figure 1 depicts the system architecture, with the core modules in purple, while Figure 2 examines what happens inside of AutoKGen's KG construction pipeline, with the steps featuring a CK being orange. In both cases, the design is provider-agnostic, and the same prompt corpus can support different LLMs.

The pipeline consists of eight steps that are sequentially executed while also supporting standalone invocation. Three also validate compliance at the mapping, graph generation, and post-import validation checkpoints (CK). A fourth, optional checkpoint follows post-validation as part of the compliance model described in 3.4. Figure-2 shows the sequence and checkpoint placement.

Step 1. Merging: The agent inspects the input directory, classifies each file by extension and content and selects a merge strategy, union for files sharing a common schema, join for files sharing a foreign key and side-by-side for complementary aspects or single-file conversion.

Step 2. Profiling: A profile is generated per column, inferring datatypes, counting missing values, identifying cardinality, sampling values and basic numeric statistics. In case supplementary files are present, the profiler cross-references columns against the lookup tables and annotates them for enrichment. Any output is stored in a JSON file.

Step 3. Ontology parsing: The agent discovers ontology files in the directory, in popular formats (`.owl`, `.xml`, `.ttl`, `.rdf`, `.n3`, `.jsonld`), parses them with the n3 and rdflib [37, 38] packages and generates a normalized JSON representation. Within it, we find class hierarchies, object properties with domain, range, characteristics, data properties with domain and XSD range, and finally, namespace

[1] https://github.com/developedbygeo/AutoKGen/blob/main/README.md

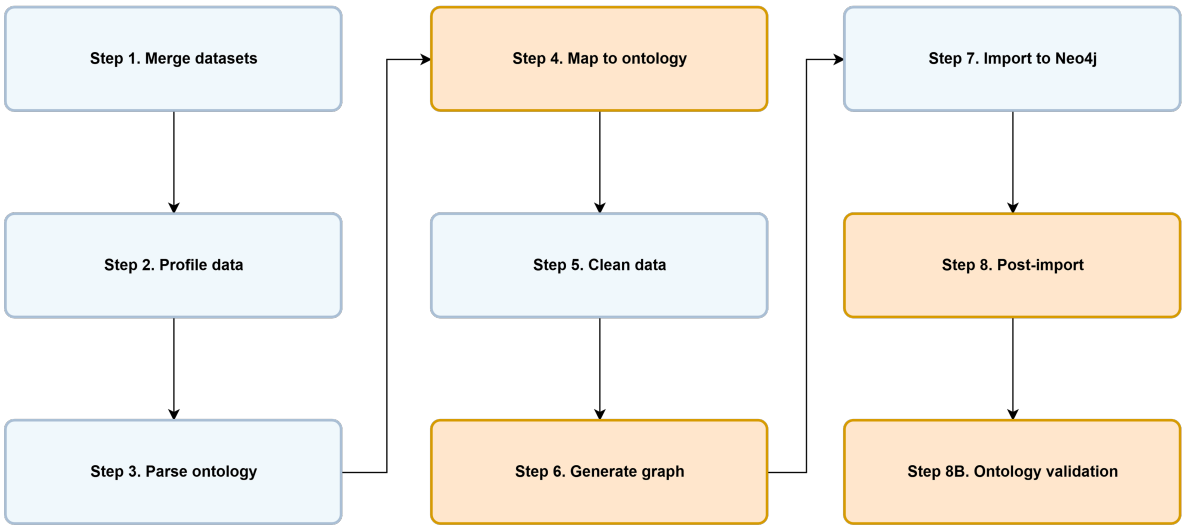


Figure 2: AutoKGen's eight-step KG construction pipeline with four compliance checkpoints (CK)

declarations. As a lossy projection, any declarations remain as labels and property keys, while axioms are dropped and, domain and range are checked.

Step 4. Mapping (CK1): The agent attempts to map each dataset column to an ontology class or property, using cardinality from the profiling to guide the entity and attribute classification. Each successful mapping results in the agent recording the source column, the reference ontology term, the role and the confidence score. Columns that cannot be mapped are flagged. Finally, a compliance score is computed and if the score falls below the threshold, the pipeline exits.

Step 5. Cleaning: Ontology-aware cleaning, that revolves around preparing the data. The first step is casting based on the XSD ranges in each ontology data property (e.g., xsd:date columns are normalized to ISO-8601). Following this, duplicate rows are identified and removed, while if supplementary lookup tables are present, coded values are validated against them.

Step 6. Graph generation (CK2): The cleaned dataset is transformed into a KG in two formats, JSON (`graph-data.json`) for MCP-based import and Cypher (`graph-import.cypher`) as a self-contained Neo4j script, along with an optional RDF/Turtle serialization (`graph-data.ttl`). The generated URIs follow a deterministic pattern (http://data.example.org/<class>/<id>), therefore runs are reproducible. In addition, the code generator applies in-action validation, as every node label, property key and relationship type is checked against the `ontology-structure.json`, and the domain and range constraints are verified. Any identified violations are logged to `graph-validation-errors.json`.

Step 7. Import: The graph is loaded into Neo4j through the MCP server (3.2). Prior to importing, the agent issues CREATE UNIQUE CONSTRAINT statements on each node-label/ID pair, as a per-class indexing strategy. Therefore, the relationship-creation phase can use indexed MATCHED lookups rather than full scans. Nodes are created in batches of 5,000 and relationships in batches of 2,500, with each requiring a MATCH plus CREATE. For graphs above 100,000 nodes, the prompt instructs the agent to split the nodes into 4 chunks and import them in parallel for efficiency.

Step 8. Post-import (CK3): A quick-pass post-import operation that checks the live Neo4j graph, to identify potential violations introduced by the import process (e.g. partial loads, indexing race conditions etc.). It additionally produces actionable Cypher queries.

Step 8B. Ontology validation (CK4): The core post-import audit that validates the Neo4j graph. The checks performed include label-set membership (using `db.labels()` against ontology classes), domain/range compliance, datatype consistency per property, presence of required properties and namespace consistency. Violations are categorized by severity and the agent generates a Cypher fix query. A subsequent compliance checkpoint, featuring SHACL, executes human pre-authored files against the imported graph via the Neosemantics [36] (n10s) plugin.

3.2. Neo4j MCP Server

The agent interacts with the Neo4j instance exclusively through a custom MCP server. The server runs as a stdio process, exposing eight tools that cover the full lifecycle of the graph. The core tools are a) `execute_cypher`, which runs a Cypher query with optional parameters, primarily used for ad-hoc reads and schema operations, along with b) `create_nodes` and c) `create_relationships`. Additionally, the d) `run_bulk_import` tool ingests a graph data file directly, allowing the agent to defer the work of streaming millions of triples, while the e) `get_schema` and f) `get_statistics` tools expose labels, relationship types, constraint listings and node/relationship counts, used primarily in CK3. The g) `validate_graph` tool performs an ontology-aware structural check and h) `clear_database` purges all data.

3.3. WebUI Consumption Interface

A complementary WebUI exposes the constructed KG through a natural-language chat interface[2], allowing non-technical users to query it conversationally. Leveraging React [37] and Vite [38] for the frontend, it communicates over Vercel's AI SDK [39], streaming through an Express [40] backend.

3.4. Four-step compliance checks

Validation is enforced proactively in-depth, with the goal being for potential violations to be caught as early as possible and rectified. The four validation steps operate at distinct points in the pipeline and against specific artifacts.

CK1 (Mapping) operates over `mapping-strategy.json`, to prevent any drift before the data is generated and is the most efficient detection point, as the dataset profile and the parsed ontology are the only inputs at this point.

CK2 (Generation) operates per triplet during KG construction to prevent invalid nodes or relationships from being written to the database. The check is in-process and is structural, examining labels, property keys, relationship types and domain/range constraints against the ontology.

CK3 (Post-import) operates on the live KG and identifies violations introduced by the import process (e.g. partial loads, indexing race conditions, type coercions etc.). If changes are required, it produces Cypher.

CK4 (SHACL) performs the strongest semantic check, based on expert pre-authored shapes that are domain-specific, to validate cardinality, value ranges, regex constraints and the pipeline loads and executes them without modifications. This is an optional step that where human input is required, in the format of SHACL shapes.

4. Experiments

We evaluate AutoKGen across three different domains/datasets: MoMA, DBLP, and GeoNames. We measure a) compliance at four checkpoints, b) scale and throughput, and c) cross-provider robustness with Codex acting on an identical prompt corpus.

4.1. Experimental setup

The evaluation is designed to stress AutoKGen across three datasets, in terms of scale, filetype and domain. We deliberately chose domains that are very diverse, MoMA (CSV, with a narrow EDM ontology of 14 classes and 157,000 rows), DBLP (XML, elaborated FaBiO ontology of 261 classes and 12.3 million rows) GeoNames (headerless TXT plus three supplementary lookup tables, along with a narrow Geo-SPARQL ontology of 6 classes and 790,000 rows).

[2] https://github.com/developedbygeo/AutoKGen/blob/main/README.md

Table 1
Domain identity

Domain	Format	Rows	Size	Ontology	Classes	Suppl.
MoMA	2 CSV (join)	157,457	72 MB	EDM v5.2.4	14	–
DBLP	1 XML	12.36 M	4.7 GB	FaBiO v2.2	261	–
GeoNames	7 TXT (union)	790,613	105 MB	GeoSPARQL	6	3 lookups

The same prompt corpus, MCP server and compliance model apply to every run. Our hypothesis is that AutoKGen's pipeline module should complete all three with no domain-specific edits.

The experiments revolve around two core providers and their frontier models (Claude Code and OpenAI Codex). By utilizing their CLI, and setting the model preference to 'Automatic', usage logs indicate that the largest, frontier models were selected (Opus 4.6 and GPT-5.4). Table-1 summarizes the three domain specifications, while subsequent tables examine the results of each step of the pipeline. Table-2 examines ontology compliance scores at CK1-CK3 (3.4, Table-3 for SHACL conformance at CK4 (4.3), and Table-4 for scale and execution profile. Table-5 reports the per-file code quality review of what each agent generated in each step.

The MoMA dataset consists of 2 CSV files (`artists.csv` - 15,627 rows and artworks.csv - 157,457 rows), which Step 1 joined on the ConstituentID foreign key, creating a 157,457-row, 38-column merged table. DBLP consists of a single 4.7 GB XML file with a 13 KB DTD, which Step 1 converted into a 12,358,922-row, 29-column CSV via SAX [41], processing the file record-by-record. GeoNames, on the other hand, consists of seven headerless, tab-delimited TXT country files (DE, ES, FR, GB, IT, NL, PL), amounting to 790,613 rows across 20 positional columns, which Step 1 stacked under the union strategy. Notably, GeoNames is the only domain dataset that features supplementary files (`countryInfo.txt` with country metadata and ISO codes, `admin1CodesASCII.txt` with administrative names, and `featureCodes_en.txt` with feature-code taxonomies).

4.2. Ontology compliance (CK1-CK3)

Table-2 reports per-checkpoint scores, with the agent-generated mapping score following a base deduction rule with the halting threshold being CK < 70, CK2 < 60 and CK3 < 70. This is depicted with the following formula where U being the number of dataset columns that the agent could not map to any ontology class or property and C the number of custom or invented terms the agent attempted to introduce that were not present in the `ontology-structure.json`.

$$\text{score}(\text{CK1}) = 100 - 10 \cdot |U| - 20 \cdot |C| \tag{1}$$

Table 2
Per-checkpoint compliance scores per domain and provider. Nodes represented with (N) and Relationships with (R)

Domain	Agent	CK1 (mapping)	CK2 (generation)	CK3 (post-import)
MoMA	Claude Code	95	95–100% N, 100% R	100
MoMA	Codex	0 (bypassed)	100 (invalid)	86 (invalid)
DBLP	Claude Code	82	82-75.12% N, 100% R	90
DBLP	Codex	10 (bypassed)	–	–
GeoNames	Claude Code	70	100% N, 100% R	98
GeoNames	Codex	– (crashed)	Failed at Step 6	–

Anthropic - Claude Code: All three domains were successfully processed with zero invented terms, with the agent flagging unmapped columns. Precisely, MoMA featured 6 unmapped, with all being duplicates or 100% null, DBLP featured 6 metadata fields without FaBiO equivalents and GeoNames had

5 admin-code columns that were lacking GeoSPARQL counterparts. At CK2, in-flight validation rejected no relationships, with the only regression being DBLP's 4.10M nodes, that were typed as `foaf:Person`, an external vocabulary referenced by FaBiO but not defined within it. At CK3, MoMA scored 100%, with no violations and two informational notes, DBLP scored 90%, reproducing the foaf:Person finding, for which CK3 generated a Cypher fix to re-label or remove affected nodes and GeoNames scored 98%, with four informational notes for abstract GeoSPARQL classes (SpatialObject, `FeatureCollection`, `SpatialObjectCollection`, `GeometryCollection`). Across all three Claude runs, the dominant residual non-compliance pattern is external vocabulary dependencies.

OpenAI – Codex: During all experimental runs, Codex consistently failed to successfully execute. On MoMA, Codex's mapping code recorded a 0, with only 43 out of 38 columns mapped, with its internal generator incorrectly inferring a score of 100 and proceeded. The resulting graph passed CK3, solely because the audit ran against the nodes that the generator emitted. On DBLP, the FaBiO ontology's lack of agent / person classes, drove CK1 to 10 (9 of 29 columns unmapped), and Codex's generator, failed to enforce the 70-halting threshold. Step 6 then exited cleanly, without providing output before CK2 could be evaluated. On GeoNames, Codex's pipeline crashed at Step 6 before producing a graph, so neither CK1 nor CK3 had a meaningful contribution. The common pattern is that the four-checkpoint model only constrains the agent's actions, as the generated code was routed around a checkpoint, with the model explicitly recording the bypass.

4.3. SHACL Validation (CK4)

CK4 ran against each domain's pre-authored SHACL shapes against the imported graph. The conformance formula where V being the identified count of violations and N the count of nodes that were successfully validated, can be represented as:

$$\text{conformance} = 1 - |V|/|N| \tag{2}$$

Table 3
SHACL conformance per domain and agent, using N10s against pre-authored shape files

Domain	Agent	Shapes	Nodes	Violations	Warnings	Conformance
MoMA	Claude Code	6	369,356	59	2,731	99.24%
MoMA	Codex	6	261,832	247,728	14,095	5.4%
DBLP	Claude Code	14	16,480,647	1	0	99.99%
DBLP	Codex	–	–	–	–	–
GeoNames	Claude Code	2	1,581,327	107	100	99.99%
GeoNames	Codex	–	–	–	–	–

All three runs for Claude Code surpassed the 98% mark. MoMA's 59 violations were identified as 56 `ProvidedCHO` nodes missing `dc:title` and 3 Agent nodes missing `skos:prefLabel`, which were classified as upstream data quality issues. DBLP's single violation was a one-off cardinality breach across 16.M nodes, while GeoNames's 107 violations revolve around Feature nodes missing an explicit `hasGeometry` edge, despite the corresponding Geometry node existing. This is attributed to the generator's side join, and was a candidate for an automated CK3 fix. Additionally, shapes target specifically the created labels, marking CK4 precision-only, without recall.

4.4. Scale, performance and memory

Table-4 summarizes the scale and execution profile of the experimental runs. It is particularly notable that Codex completed only the MoMA cell, as DBLP and GeoNames failed at Step 6 due to V8 string length limits (after attempting `JSON.stringify` and `fs.readFileSync` respectively). Claude Code timings represent the successful run in each step, while Codex's MoMA timings are summed, with the respective retries noted.

Table 4
Scale and end-to-end (E2E) execution profile per domain and agent. CC = Claude Code, Cdx = Codex

Metric	MoMA CC	MoMA Cdx	DBLP CC	DBLP Cdx	GeoN. CC	GeoN. Cdx
Input rows	157,457	157,457	12.36 M	12.36 M	790,613	790,613
Output nodes	369,356	261,823	16.48 M	–	1.58 M	–
Output relationships	960,593	181,762	44.81 M	–	2.37 M	–
Step 6 (generation) time	4m 40s	12m 9s (2 retries)	18m 17s	–	4m 44s	–
Step 7 (import) time	6m 15s	25m 44s (7 retries)	100m 46s	–	46m 27s	–
E2E	∼31 min	∼110m	∼160 min	–	∼73 min	–
Peak step 6 (heap)	<500 MB	unbounded	∼1.6 GB	V8 str. limit	<800 MB	V8 str. limit
Relationship match	100%	Partial	91.3%	–	100%	–
Orphaned nodes	0	80,671 (30.8%)	4,070 (0.02%)	–	0	–

The agent adapted Step 6 to scale without modifying the prompt, leveraging in-memory generation for MoMA and GeoNames, and utilizing NDJSON streaming [42] for DBLP. This practice kept peak heap to approximately 1.6 GB on the 16.5M-node, 44.8M-relationship graph. Step 7 throughput is bounded by relationship MATCH+CREATE we measured an estimate of: a) Node throughput: ∼9.1–9.7K/s (with CREATE on indexed labels), b) Relationship throughput: ∼38–137K/s (with the outlier being GeoNames, due to a simpler MATCH+CREATE lookup pattern)

$$\text{throughput}_x = |R_x|/t, \quad x \in \{\text{nodes}, \text{rels}\} \tag{3}$$

where $|R_x|$ is the number of records of type x (nodes or relationships), t is the import time in seconds (excluding the LLM code-generation latency), and batch sizes are 5,000 and 2,500 for nodes and relationships respectively. The rates are computed over the MCP recorded duration.

4.5. Cross-provider comparison: Claude Code (Opus 4.6) vs ChatGPT Codex (GPT-5.4)

The process was executed again to identify the reasons behind Codex's failures, and additionally, we attempted prompt fine tuning, with explicit streaming directives, Newline Delimited JSON (NDJSON) [42] examples, memory-constraint hints and chunk-processing scaffolding, none of which resolved the issues. Additionally, upon log examination, it was identified that, apart from the runtime and performance gaps between the two agents, Codex also failed to perform a valid self-assessment, bypassing the halting prompt and disregarding the explicit prompt acting as a guardrail.

4.6. Review of generated code

AutoKGen generates all executable code at runtime and therefore, the quality of the generated code itself is an experimental variable. We conducted a structured review of the TypeScript artifacts produced for the MoMA domain, which is the only domain where both providers produced successfully executable code and scored each file on seven dimensions: correctness (25%), space-time complexity (20%), structure and maintainability (15%), type safety (15%), error handling (10%), security (10%) and documentation (5%). Table-5 reports per-step scores.

In both agents, the aggregate scores are close code quality-wise, with Codex leveraging parameterized Cypher, parallel worker threads and progress tracking for importing. Certain key weaknesses consistently appeared during the code review, namely, monolithic architecture (both agents routinely created files exceeding 500 lines of code), backpressure (neither added backpressure to write streams, with Codex reading entire files synchronously), hardcoded credentials (both embedded credentials in the generated code instead of reading from the environment), domain coupling (in the mapping step, both hardcoded MoMA column names and EDM URIs, even though the prompt was ontology-agnostic. This indicates that the underlying models collapsed abstraction into domain-specific scaffolding).

Table 5
Code quality scores for Claude Code and Codex on the MoMA dataset. Each row corresponds to one of the pipeline steps

Step	Claude (score)	Codex (score)
Step 1 – Merging	6.65	6.95
Step 2 – Profiling	7.80	7.40
Step 3 – Ontology parsing	6.50	6.50
Step 4 – Mapping (CK1)	5.85	5.80
Step 5 – Cleaning	7.20	7.10
Step 6 – Graph generation (CK2)	6.40	6.20
Step 7 – Import	7.00	7.95
Step 8 – Ontology validation (CK3)	6.70	5.95
Step 8b – SHACL validation (CK4)	N/A[a]	7.95
Aggregate (weighted)	6.76 (over 8 files)	6.87 (over 9 files)

[a]Claude ran SHACL directly via the MCP `execute_cypher` tool.

5. Conclusion

AutoKGen, throughout its pipeline, shows that an LLM agent, given only structured data and a reference ontology, can autonomously plan, generate and execute the entire KG construction pipeline without domain-specific logic. Two architectural commitments enable this: code generation with prompt-as-the-source-of-truth, which leaves the implementation solely to the agent and tool-enabled database access through a purpose-built MCP server. Across MoMA, DBLP and GeoNames, the four-checkpoint model held the pipe-line to grade C or better at every stage, resulting in SHACL conformance above 99%. Meanwhile, the cross-provider experiments confirmed prompt portability, while exposing the code quality differences across two major frontier models, Claude Opus 4.6 and ChatGPT-5.4.

AutoKGen's pipeline is a starting point, and the experiments have several constraints that frame future work. Provider coverage is narrow, as only two frontier LLMs were tested. A wider survey across the latest models, including Gemini and open-weights models is needed, as the results suggest that today's frontier models are barely sufficient. In addition, the generated code quality was mediocre, with both providers sharing systemic issues. Compliance enforcement needs to be hardened, as it was proven that it can be bypassed. Another limitation is the user-authored SHACL shapes, which are not synthesized, and prompt-driven shape generation grounded in the ontology is the natural next step. Scope is limited to three domains, which although diverse, are insufficient to claim ontology-agnosticism. Structural validity is measured based on the created nodes and not coverage, therefore unable to compute recall. Security also needs to be taken into account, as LLM-generated code is executed.

Supplemental Material Statement. The source code is available in this repository and source data has been omitted due to size restrictions (MoMA, DBLP and GeoNames). They are, however, readily available.

Declaration on Generative AI

During the preparation of this work, the authors used Generative AI in order to:

1. Perform grammar and spelling checks
2. Receive assistance with formatting

After using this tool, the authors reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] M. Hofer, J. Frey, E. Rahm, Towards self-configuring Knowledge Graph Construction Pipelines using LLMs – A Case Study with RML, in: Proceedings of the Fifth International Workshop on Knowledge Graph Construction (KGCW 2024) co-located with ESWC 2024, volume 3718 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024.

[2] H. B. Giglou, J. D'Souza, N. Mihindukulasooriya, S. Auer, LLMs4OL 2025 Overview: The 2nd Large Language Models for Ontology Learning Challenge, Open Conference Proceedings 6 (2025). doi:10.52825/ocp.v6i.2913, accessed: 2026-05-03.

[3] H. B. Giglou, J. D'Souza, F. Engel, S. Auer, LLMs4OM: Matching Ontologies with Large Language Models, 2024. doi:10.48550/arXiv.2404.10317. arXiv:2404.10317, accessed: 2026-05-03.

[4] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, K. J. Batenburg, Agentic Large Language Models, a survey, Journal of Artificial Intelligence Research 84 (2025). doi:10.1613/jair.1.18675. arXiv:2503.23037, accessed: 2026-05-03.

[5] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, A. E. Hassan, Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers, 2026. doi:10.48550/arXiv.2506.13538. arXiv:2506.13538, accessed: 2026-05-03.

[6] Introducing the Model Context Protocol, https://www.anthropic.com/news/model-context-protocol, 2026. Accessed: 2026-05-03.

[7] Microsoft/TypeScript, Microsoft, 2025. Accessed: 2025-01-20.

[8] MuseumofModernArt/collection, https://github.com/MuseumofModernArt/Collection, 2026. Accessed: 2026-05-03.

[9] Dblp: Computer science bibliography, https://dblp.org/, 2024. Accessed: 2026-05-03.

[10] GeoNames, https://www.geonames.org/, 2026. Accessed: 2026-05-03.

[11] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, https://www.w3.org/TR/r2rml/, 2012. Accessed: 2024-04-19.

[12] RDF Mapping Language (RML), https://rml.io/specs/rml/, 2024. Accessed: 2026-05-03.

[13] J. Arenas-Guerrero, A. Alobaid, M. Navas-Loro, M. S. Pérez, O. Corcho, Boosting Knowledge Graph Generation from Tabular Data with RML Views, in: C. Pesquita, E. Jimenez-Ruiz, J. McCusker, D. Faria, M. Dragoni, A. Dimou, R. Troncy, S. Hertling (Eds.), The Semantic Web, volume 13870, Springer Nature Switzerland, Cham, 2023, pp. 484–501. doi:10.1007/978-3-031-33455-9_29, accessed: 2026-05-03.

[14] M. Freund, S. Schmid, R. Dorsch, A. Harth, FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer, in: A. Meroño Peñuela, A. Dimou, R. Troncy, O. Hartig, M. Acosta, M. Alam, H. Paulheim, P. Lisena (Eds.), The Semantic Web, volume 14665, Springer Nature Switzerland, Cham, 2024, pp. 40–56. doi:10.1007/978-3-031-60635-9_3, accessed: 2026-05-03.

[15] H. Bian, LLM-empowered knowledge graph construction: A survey, 2025. doi:10.48550/arXiv.2510.20345. arXiv:2510.20345, accessed: 2026-05-03.

[16] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, J. Larson, From Local to Global: A Graph RAG Approach to Query-Focused Summarization, 2025. doi:10.48550/arXiv.2404.16130. arXiv:2404.16130, accessed: 2026-05-03.

[17] Welcome - GraphRAG, https://microsoft.github.io/graphrag/, 2026. Accessed: 2026-05-03.

[18] Neo4j-labs/llm-graph-builder: Neo4j graph construction from unstructured data using LLMs, https://github.com/neo4j-labs/llm-graph-builder, 2026. Accessed: 2026-05-03.

[19] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, X. Wu, Unifying Large Language Models and Knowledge Graphs: A Roadmap, IEEE Transactions on Knowledge and Data Engineering 36 (2024) 3580–3599. doi:10.1109/TKDE.2024.3352100. arXiv:2306.08302, accessed: 2026-04-17.

[20] N. Fathallah, A. Das, S. D. Giorgis, A. Poltronieri, P. Haase, L. Kovriguina, NeOn-GPT: A Large Language Model-Powered Pipeline for Ontology Learning, in: A. Meroño Peñuela, O. Corcho, P. Groth, E. Simperl, V. Tamma, A. G. Nuzzolese, M. Poveda-Villalón, M. Sabou, V. Presutti, I. Celino, A. Revenko, J. Raad, B. Sartini, P. Lisena (Eds.), The Semantic Web: ESWC 2024 Satellite Events, volume 15344, Springer Nature Switzerland, Cham, 2025, pp. 36–50.

doi:10.1007/978-3-031-78952-6_4, accessed: 2026-05-03.
[21] M. Val-Calvo, M. Egaña Aranguren, J. Mulero-Hernández, G. Almagro-Hernández, P. Deshmukh, J. A. Bernabé-Díaz, P. Espinoza-Arias, J. L. Sánchez-Fernández, J. Mueller, J. T. Fernández-Breis, OntoGenix: Leveraging Large Language Models for enhanced ontology engineering from datasets, Information Processing & Management 62 (2025) 104042. doi:10.1016/j.ipm.2024.104042, accessed: 2026-05-03.
[22] C. Qu, S. Dai, X. Wei, H. Cai, S. Wang, D. Yin, J. Xu, J.-R. Wen, Tool Learning with Large Language Models: A Survey, Frontiers of Computer Science 19 (2025) 198343. doi:10.1007/s11704-024-40678-2. arXiv:2405.17935, accessed: 2026-05-03.
[23] JSON-RPC, https://www.jsonrpc.org/, 2025. Accessed: 2026-05-06.
[24] Model Context Protocol - Overview, https://modelcontextprotocol.io/specification/2025-11-25, 2026. Accessed: 2026-05-03.
[25] Model Context Protocol – Codex | OpenAI Developers, https://developers.openai.com/codex/mcp, 2026. Accessed: 2026-05-03.
[26] Scaling MCP adoption: Our reference architecture for simpler, safer and cheaper enterprise deployments of MCP, https://blog.cloudflare.com/enterprise-mcp/, 2026. Accessed: 2026-05-03.
[27] What is Model Context Protocol (MCP)? A guide, https://cloud.google.com/discover/what-is-model-context-protocol, 2026. Accessed: 2026-05-03.
[28] What is the AWS MCP Server (Preview)? - AWS MCP Server (Preview), https://docs.aws.amazon.com/aws-mcp/latest/userguide/what-is-mcp-server.html, 2026. Accessed: 2026-05-03.
[29] S. Fan, X. Ding, L. Zhang, L. Mo, MCPToolBench++: A Large Scale AI Agent Model Context Protocol MCP Tool Use Benchmark, 2025. doi:10.48550/arXiv.2508.07575. arXiv:2508.07575, accessed: 2026-05-03.
[30] Z. Wang, Q. Chang, H. Patel, S. Biju, C.-E. Wu, Q. Liu, A. Ding, A. Rezazadeh, A. Shah, Y. Bao, E. Siow, MCP-Bench: Benchmarking Tool-Using LLM Agents with Complex Real-World Tasks via MCP Servers, 2025. doi:10.48550/arXiv.2508.20453. arXiv:2508.20453, accessed: 2026-05-03.
[31] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A Survey on Large Language Models for Code Generation, ACM Transactions on Software Engineering and Methodology 35 (2026) 1–72. doi:10.1145/3747588. arXiv:2406.00515, accessed: 2026-05-03.
[32] A. Alidu, M. Ciavotta, F. DePaoli, Prompt2DAG: A Modular Methodology for LLM-Based Data Enrichment Pipeline Generation, 2025. doi:10.48550/arXiv.2509.13487. arXiv:2509.13487, accessed: 2026-05-03.
[33] C. Yang, C. Li, X. Hu, H. Yu, J. Lu, Enhancing knowledge graph interactions: A comprehensive Text-to-Cypher pipeline with large language models, Information Processing & Management 63 (2026) 104280. doi:10.1016/j.ipm.2025.104280, accessed: 2026-05-03.
[34] Y. Feng, S. Papicchio, S. Rahman, CypherBench: Towards Precise Retrieval over Full-scale Modern Knowledge Graphs in the LLM Era, 2025. doi:10.48550/arXiv.2412.18702. arXiv:2412.18702, accessed: 2026-05-03.
[35] Shapes Constraint Language (SHACL), https://www.w3.org/TR/shacl/, 2017. Accessed: 2026-05-03.
[36] C. Cortés, L. Ehrlinger, L. Etcheverry, F. Naumann, Is SHACL Suitable for Data Quality Assessment?, 2025. doi:10.48550/ARXIV.2507.22305, accessed: 2026-05-03.
[37] React Reference Overview – React, https://react.dev/reference/react, 2026. Accessed: 2026-05-04.
[38] Vite, https://vite.dev, 2026. Accessed: 2026-05-04.
[39] AI SDK by Vercel, https://ai-sdk.dev/docs/introduction, 2026. Accessed: 2026-05-04.
[40] Express - Node.js web application framework, https://expressjs.com/, 2025. Accessed: 2026-05-06.
[41] Xml.sax — Support for SAX2 parsers, https://docs.python.org/3/library/xml.sax.html, 2025. Accessed: 2026-05-04.
[42] NDJSON.com - JSON Lines Format Guide | NDJSON.com, https://ndjson.com/, 2026. Accessed: 2026-05-06.