

Generating Executable Agentic AI Systems from Knowledge Graph Specifications

Ravie Arjun Nadhief¹, Naura Ranantya Kinanti¹, Jihan Dwi Athanaya¹,
Geraldine Angelie Tari Damauly Hutagalung¹, Jocelin Marella Ramadhaniska¹,
Muhammad Alfian Amrizal¹, Kabul Kurniawan^{1,2*}, and Andreas Ekelhart^{3,4}

¹ Department of Computer Science and Electronics, Universitas Gadjah Mada,
Indonesia

² Center for Cryptography and Cybersecurity Research, Universitas Gadjah Mada,
Indonesia

³ SBA Research, Vienna, Austria

⁴ University of Vienna, Vienna, Austria

Abstract. Modern LLM-based multi-agent frameworks embed agent logic and coordination directly in code, making designs hard to analyze, reuse, or port. We present a semantic-driven pipeline that deterministically compiles knowledge-graph specifications, expressed in the Agentic AI Ontology (AgentO), into executable multi-agent projects for four structurally distinct frameworks: AutoGen, CrewAI, LangGraph, and Mastra AI. The pipeline extracts agents, tasks, goals, and tools via SPARQL, builds a framework-agnostic intermediate model, and emits framework-native Python or TypeScript through per-framework adapters and templates. Evaluated on 50 workflow scenarios under same-framework and 4×4 cross-framework conditions (200 generation runs), every generated project compiles. Same-framework generation reaches 100% executability, 95.6% concept coverage, and 94.4% workflow-graph similarity. Two behavioural case studies under real execution show equivalent runtime semantics for a sequential workflow, but not for a hierarchical one. Portability is bounded by the target framework’s execution model rather than the ontology, with configuration-level semantics (process type and orchestration metadata) identified as the main remaining gap toward fully automated translation.

Keywords: Agentic AI · Knowledge Graphs · Multi-Agent Systems · Code Generation · Cross-Framework Interoperability · Model-Driven Engineering.

1 Introduction

LLM-driven multi-agent (or agentic AI) frameworks have recently emerged to orchestrate complex tasks with collaborating AI agents [1]. Popular frameworks include CrewAI⁵, AutoGen [2], LangGraph [3], and Mastra AI⁶, which assign

* Corresponding author: kabul.kurniawan@ugm.ac.id

⁵ <https://crewai.com/>

⁶ <https://mastra.ai/>

agents roles, goals, and tools to plan and execute multi-step workflows. However, these frameworks encode agent workflows directly in code or configuration files, without a formal model, which makes high-level designs difficult to analyze, validate, or port. Details such as tool configurations and delegation rules must be hard-coded rather than derived from a shared specification.

The cost of this coupling grows once more than one framework is in play, since the four share no common execution model. CrewAI organizes work as role-bearing agents advancing through a largely linear process, AutoGen as a conversation whose next speaker is chosen at runtime, LangGraph as an explicit state machine over nodes and edges, and Mastra AI as a statically typed workflow. Migrating a workflow between them is therefore a manual rewrite rather than a port, and translating directly between N frameworks would require $N(N - 1)$ pairwise translators, each obliged to recover design intent from imperative code. A shared declarative description reduces this to one specification and N mappings. Knowledge graphs and ontologies provide exactly such a description, capturing entities and relations in an explicit, machine-readable graph [4]—but only if the graph retains enough of the workflow to regenerate running code. We therefore ask whether a single knowledge graph can be translated into multiple frameworks while preserving equivalent functionality.

To date, work integrating knowledge graphs with multi-agent AI has treated the graph as a runtime knowledge source rather than as a system specification [5]. The Agentic AI Ontology (AgentO) [6] is the exception in that it models the structural components of agentic frameworks, but it stops at description and does not generate anything from them. Building on AgentO, we address the automated generation of executable multi-agent systems from knowledge graph specifications, and thereby operationalize the interoperability that AgentO represents but does not itself realize.

Our approach starts with a knowledge graph in RDF, expressed in Turtle, and turns it into code through a four-stage pipeline. SPARQL queries extract agents, tasks, tools, and workflow structure, a framework-agnostic intermediate representation normalizes them, a per-framework adapter maps that representation onto the target’s execution model, and templates render a complete project in Python for AutoGen and CrewAI or in TypeScript for LangGraph and Mastra AI.⁷ Because rendering is template-driven, the same graph always emits the same project.

We evaluate the pipeline on 50 agent workflow scenarios from the four framework ecosystems, compiling every graph into every target, and inspect the runtime behaviour of a sequential and a hierarchical workflow under real execution.

The main contributions of this paper are as follows. (i) **Transformation method**: a four-stage architecture for transforming ontology-based knowledge graph specifications into executable multi-agent systems. (ii) **Prototype implementation**: a working system targeting four frameworks that span four

⁷ Our LangGraph target is the framework’s official TypeScript implementation, `@langchain/langgraph`, matching the TypeScript reference workflows in our corpus.

orchestration paradigms and two programming languages. (iii) **Use case application**: an end-to-end demonstration tracing a single knowledge graph to a running project, and showing that the same graph compiles unmodified into all four targets. (iv) **Execution-validated evaluation**: an evaluation under same-framework and cross-framework conditions using four quantitative metrics plus a composite compatibility score, together with two behavioural case studies under real execution.

2 Related Work

Sapkota et al. distinguish individual AI agents from agentic AI systems characterized by coordinated autonomy and dynamic task decomposition [1], and AutoGen [2] and LangGraph [3] formalize the conversation-centric and graph-based orchestration models respectively. What these models lack is a description of the system that exists outside any one framework, and knowledge graphs are a natural candidate for one [4].

One line of research studies how collaborative LLM systems can be organized for dependable task execution, spanning accountability and role clarity through RACI-based allocation [7], surveys of coordination strategies [8], and task-level role separation for question answering [11]. Reviews of LLM-based multi-agent systems for software engineering confirm that specialization and cross-agent verification improve robustness, yet they also observe that most existing systems remain manually designed [9]. Complementing these studies, Shin and Utz propose WorkflowML as a platform-independent workflow modelling language grounded in ontology integration, suggesting that formal descriptions of tasks, roles, and workflow steps can help bridge heterogeneous execution environments [10]. In model-driven engineering terms, our knowledge graph is the platform-independent model and code generation the platform-specific mapping.

A second direction integrates structured knowledge representations directly into agentic execution. Closest to our setting, AGENTiGraph combines specialized agents with knowledge graph manipulation to support domain-specific interaction, graph updates, and explainable reasoning [5]. Beyond it, knowledge-graph-guided coordination has been applied in domains from team formation to public fund allocation and cybersecurity retrieval [12,13,14,15,16,17]. AgentVerse makes a related point from the other side, showing that collaborative behaviour does not emerge on its own but depends on carefully designed coordination mechanisms [18]. In all of these studies the graph is something the agents consult while running. None of them uses it to define the agents, tasks, tools, and workflow in the first place.

The transformation of ontology-instantiated workflows into runnable software therefore remains largely unexplored. This work addresses that gap by generating framework-native projects from the graph for four targets and by validating the result through compilation and execution rather than structural inspection alone.

3 Proposed Approach

Agents, tasks, tools, and workflow structures are declared as RDF triples using AgentO [6] rather than implemented as procedural orchestration logic, which keeps the specification independent of any framework. A fixed pipeline with no LLM calls then compiles that graph into a runnable project, as Fig. 1 illustrates. Its structure is identical across all four targets.

AgentO in Brief AgentO [6] describes an agentic system at four levels. At the actor level, a `:Team` groups `:LLMAgent` and `:HumanAgent` individuals through `:hasAgentMember`. Each agent has a role, a goal, a language model, and a set of tools it may use. At the work level, a `:Task` is performed by an agent and carries a `:Prompt` that holds the instruction, the input data, and the context in natural language. Tasks are connected through the resources they require and produce. At the orchestration level, a `:WorkflowPattern` is composed of `:WorkflowStep` individuals chained by `:nextStep` and `:stepOrder`. A pattern starts at a `:StartStep`, ends at an `:EndStep`, and may nest other patterns through `:hasSubPattern`. At the configuration level, settings for the team, the agents, and the tools are all expressed as `:Config` individuals that hold a plain key and value string. The ontology therefore has explicit constructs for who acts, what is done, and in which order, but none for conditional branching, termination predicates, or dynamic speaker selection. Such behaviour can only be recorded as a configuration string.

Knowledge Graph Extraction A pre-parsing pass first repairs syntax irregularities in the input Turtle file. The pipeline then executes a SPARQL query suite, covering 15 AgentO classes and 48 properties, to retrieve all ontology-defined entities and attributes, including agents, language models, tools, tasks, workflow steps, team structures, and execution parameters. The suite consists of 51 queries, one group per entity type. Within each query, only the type assertion is mandatory and every attribute is wrapped in an `OPTIONAL` clause. An agent without a declared goal or a task without a prompt is therefore still extracted rather than dropped. Additionally, this stage analyzes the network of `:nextStep` properties across `:WorkflowStep` instances to infer the underlying graph topology (linear, branching, or cyclic).

Intermediate Representation Construction The extracted entities are compiled into a normalized, strongly typed *Intermediate Representation* (IR). `:LLMAgent` nodes become agent models with identifier, role, LLM configuration, and tool capabilities, and `:Task` nodes become task models with description, assigned agent, inputs, and expected output. Because each IR component aligns with AgentO classes rather than framework-specific ones, all source graphs arrive at the same canonical abstraction. This stage also connects task outputs to the tasks that consume them, which establishes data flow and execution order, and fills in defaults for optional parameters omitted from the graph.

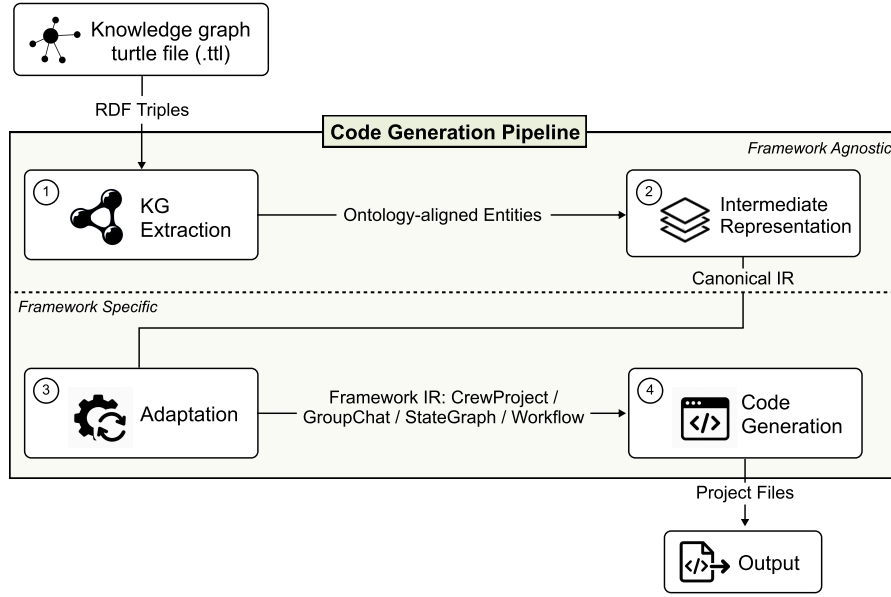


Fig. 1. The four-stage compilation pipeline. Stages 1–2 are shared by all targets; only Stages 3–4 are framework-specific, so a single knowledge graph compiles into any of the four frameworks.

Framework-Specific Adaptation A dedicated adapter for each target maps the canonical IR to the target’s native execution paradigm, and each one customizes orchestration logic to fit its framework. The CrewAI adapter orders task dependencies via topological sorting to fit CrewAI’s linear `Process.sequential` model. The LangGraph adapter translates graph nodes and edges directly into a state machine, using conditional routing when the graph carries a supervisor topology. The AutoGen adapter structures agent teams as conversational group chats. The Mastra AI adapter emits statically typed workflows with explicit schema definitions. The quantitative consequences of differing target expressivity are analyzed in Section 5.

Code Generation In the final phase, the framework-adapted data structures are synthesized into a complete, runnable project directory. Source code is rendered from Jinja2 templates, and where a framework expects YAML configuration it is written via PyYAML. Both avoid raw string formatting so that values taken from the graph are escaped correctly. Rather than emitting a single script, the generator lays the project out following each framework’s conventions. Agent specifications configure roles, models, and capabilities. Task modules encode execution steps. Orchestration files implement the overall control flow.

4 Prototypical Implementation and Use Case Application

4.1 Prototypical Implementation

Our prototype implementation of the pipeline described in Section 3 is available at <https://github.com/raviearjun/agentic-generator-paper>. It uses the AgentO ontology [6] as its semantic schema. Stages 1–2 form a shared front end, consisting of rdflib-based parsing with defensive normalization, the SPARQL query suite, and the Pydantic canonical IR. Stages 3–4 consist of one adapter–generator pair per target framework. Each generator emits the target’s idiomatic project layout, which Section 4.2 details for CrewAI. Language model credentials are read from `.env` files rather than embedded in the generated sources. Tool implementations are generated as stubs, since the graph carries a tool’s name and description but not its code.

Across the 50 knowledge graphs in the evaluation dataset, the pipeline (6,524 lines of extraction, adaptation, and generation code) deterministically produces 196 projects over the 200 source–target combinations. All but one graph are processed end to end without manual intervention.⁸

4.2 Use Case Application: Game Builder Crew

To illustrate the pipeline end to end, we use the *Game Builder Crew* scenario, in which three agents collaborate sequentially to produce a Python game from a user-provided description. A *Senior Engineer Agent* writes the initial implementation, a *QA Engineer Agent* reviews it, and a *Chief QA Engineer Agent* validates the result against the intended functionality. Fig. 2 shows the transformation, and we walk through its four numbered steps.

Knowledge graph input. The workflow is specified in `game-builder-crew.ttl`. The central `:GameBuilderCrew` individual is a `:Team` linked to its three agents, its sequential workflow pattern, its goal, and its system configuration. All of these are RDF resources, and the specification contains no procedural logic.

Step (1) — Knowledge graph extraction. The Turtle file is loaded into an RDF graph and queried with the SPARQL suite, recovering the team, its sequential process, the configured language model, and each agent’s identifier, role, and goal.

Step (2) — Intermediate representation construction. The extracted entities are normalized and validated into typed Pydantic instances and aggregated into a framework-agnostic `AgenticProject`, which fixes the sequential process, the agent and task lists, and the workflow ordering (`code_task` → `review_task` → `evaluate_task`).

Step (3) — Framework adaptation. The CrewAI adapter topologically sorts the three tasks, which here trivially yields the linear chain, maps the workflow onto `Process.sequential`, and produces a `CrewProject`.

⁸ The exception is a Mastra AI graph whose Turtle file is truncated mid-literal and cannot be parsed for any target.

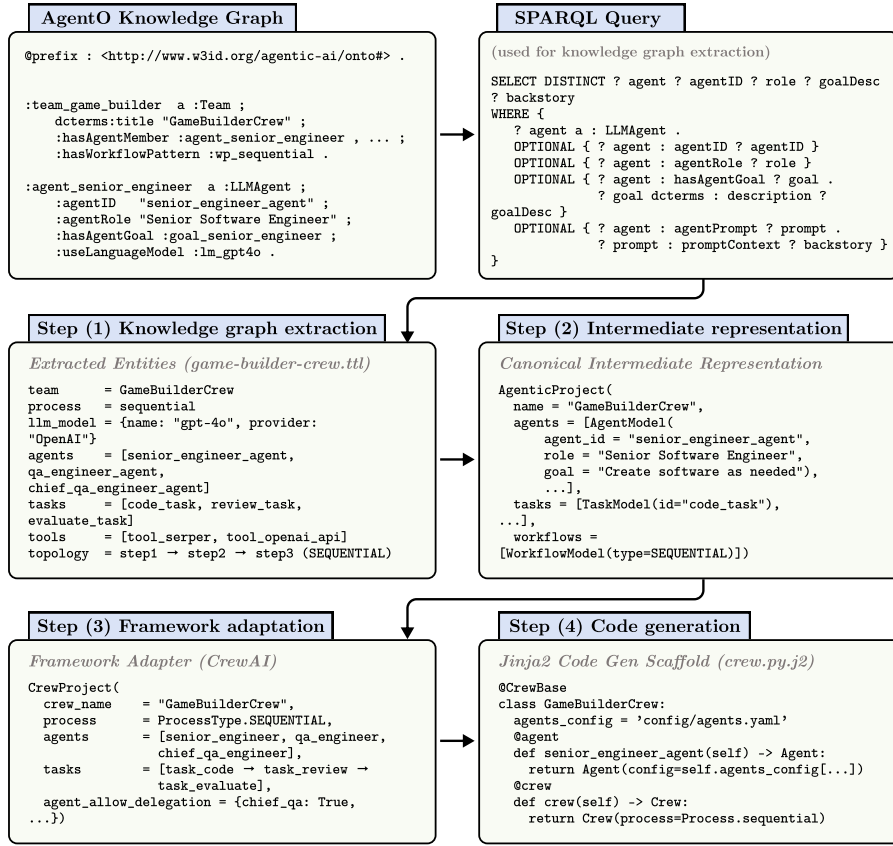


Fig. 2. Four-stage transformation of the Game Builder Crew example. The numbered panels show the entities and sequential topology extracted from game-builder-crew.ttl (1), their canonical AgenticProject intermediate representation (2), the CrewAI-adapted CrewProject structure (3), and the generated crew.py scaffold (4).

Step (4) — Code generation. The generator renders the `CrewProject` into the runnable project shown at the bottom right of the figure, consisting of `agents.yaml`, `tasks.yaml`, `crew.py`, and the project scaffold. At runtime the project instantiates the agents, loads the configuration, and executes the tasks in order.

Because extraction and the IR are framework-agnostic (Stages 1–2), the same `game-builder-crew` graph compiles without modification into the other three targets. The fidelity of such substitutions is evaluated in Section 5.

5 Evaluation and Discussion

The evaluation uses agentic workflow scenarios encoded in Turtle, derived from the AgentO benchmark [6], which models workflows from publicly available exam-

ple projects in four framework repositories. The dataset comprises 50 knowledge graphs, namely 6 for AutoGen, 16 for CrewAI⁹, 8 for LangGraph, and 20 for Mastra AI. The graphs were constructed with LLM assistance and manually verified, and we treat them as given input specifications. Initial runs surfaced two systematic irregularities, namely malformed Turtle syntax and agent persona attributes declared inconsistently across graphs. We absorbed these in the pipeline through a pre-parsing repair pass and canonical persona mapping, which Section 5.3 describes. Both were added before the evaluation runs reported below, on the same corpus, so all figures in this section include their effect.

5.1 Setup, Questions, and Metrics

Each generated project is evaluated against its source specification under two setups: a *same-framework* baseline, compiling each framework’s native graphs back into its own code, and a *cross-framework* setup, compiling every graph into all three alternative target frameworks (4×4 matrix, yielding $50 \times 4 = 200$ generation and evaluation runs). The evaluation investigates five core questions:

- **EQ1:** Can ontology-specified knowledge graphs be automatically compiled into executable projects across all four target frameworks?
- **EQ2:** How effectively do synthesized projects preserve the entities and relations declared in the source graph?
- **EQ3:** Are the generated codebases fully runnable without requiring manual code fixes?
- **EQ4:** Do generated systems maintain equivalent execution semantics at runtime?
- **EQ5:** Can a single knowledge graph specification be compiled into multiple distinct framework paradigms while preserving its structure?

Evaluation relies on four quantitative metrics at two levels. The first level measures compilation and initialization readiness. The *Syntactic Correctness Rate* (SCR) is the percentage of generated codebases that compile without syntax errors, using `py_compile` for Python and `tsc --noEmit` for TypeScript. The *Executability Rate* (ER) is deliberately an *initialization* metric. Each entry-point script is run under dummy credentials and passes if it either completes or fails gracefully at authentication. This isolates generation defects such as import or name errors from live API access, but it does not establish that the project performs its task end to end, which Section 5.2 examines for two workflows.

The second level measures semantic fidelity to the source graph. The *Concept Coverage Rate* (CCR) is the proportion of essential ontology concepts (agents, tasks, tools, steps, and key control-flow relations) recovered from the generated code via token and alias matching (Section 6). *Workflow Graph Isomorphism* (WGI) quantifies topology preservation as the maximum of a labeled graph edit distance score and an unlabeled graph signature score, so that renamed nodes do not mask a preserved structure. For cross-framework comparisons we

⁹ CrewAI examples repository: <https://github.com/crewaiinc/crewai-examples>

Table 1. Same-framework KG-to-code evaluation results. *KGs* is the number of source knowledge graphs and *Err.* the number that failed to parse. All rates are computed over the graphs that generated successfully.

Framework	KGs	Err.	SCR	CCR	ER	WGI
CrewAI	16	0	100.0%	95.0%	100.0%	91.3%
AutoGen	6	0	100.0%	93.5%	100.0%	100.0%
LangGraph	8	0	100.0%	95.4%	100.0%	94.9%
Mastra AI	20	1	100.0%	98.4%	100.0%	91.5%
Avg			100.0%	95.6%	100.0%	94.4%

combine the four into a single *Cross-Framework Compatibility Score*, CFCS = $0.2\text{SCR} + 0.2\text{ER} + 0.3\text{CCR} + 0.3\text{WGI}$.

5.2 Experiment Results

Same-Framework Results (EQ1–EQ3) As shown in Table 1, SCR and ER were both 100.0% for all four frameworks. Every generated project compiled, and every entry point either ran to completion or reached the expected authentication call, with no manual repair required (EQ1, EQ3). CCR ranged from 93.5% to 98.4% and WGI from 91.3% to 100.0% (EQ2). Disaggregating CCR by ontology construct, pooled over all projects, shows that entity-level constructs are recovered almost perfectly (agent and workflow_step 100.0%, tool 99.1%, task 97.8%) and inter-entity relations only slightly lower (agent interaction 98.4%, task-agent 95.0%). The clear outlier is the *workflow-level* category at 60.9%. It holds process-type and orchestration metadata that has no canonical field in several target APIs.

Cross-Framework (EQ2, EQ3, EQ5) Table 2 reports the cross-framework condition. Across the twelve cross-framework pairs, CCR averaged 92.8% (lowest 81.1%, CrewAI→AutoGen), ER 92.0% (lowest 68.4%, Mastra AI→CrewAI), WGI 95.4% (lowest 79.2%, Mastra AI→CrewAI), and CFCS 94.9% against a 97.0% same-framework baseline.

The pattern is directional. Reading down the target columns, AutoGen and Mastra AI absorbed every source with ER 100.0%, whereas LangGraph (ER 90.3%) and especially CrewAI (ER 85.9%, WGI 87.5%) did not. The three weakest ER cells in the study, Mastra AI→CrewAI (68.4%), Mastra AI→LangGraph (73.7%), and LangGraph→CrewAI (75.0%), all have a graph-based or typed-workflow framework as source. Reading along the rows confirms this. AutoGen and CrewAI, the two sources with the simplest execution models, port best (CFCS 97.4% and 96.3%), and LangGraph and Mastra AI port worst (95.0% and 92.8%). Coverage and executability did not move together. The pair with the lowest CCR, CrewAI→AutoGen at 81.1%, still recorded ER 100.0% (EQ2, EQ3, EQ5).

Table 2. Cross-framework results for all source–target pairs, in %. Rows are the source framework and columns the target. Diagonal entries are same-framework baselines. The final column gives the Cross-Framework Compatibility Score averaged over the four targets. SCR was 100.0% for every pair and is omitted.

Source / Target	CrewAI			AutoGen			LangGraph			Mastra AI			CFCS
	CCR	ER	WGI	CCR	ER	WGI	CCR	ER	WGI	CCR	ER	WGI	
CrewAI	95.0	100.0	91.3	81.1	100.0	100.0	92.9	87.5	100.0	99.3	100.0	100.0	96.3
AutoGen	88.1	100.0	92.6	93.5	100.0	100.0	91.2	100.0	100.0	100.0	100.0	100.0	97.4
LangGraph	92.9	75.0	87.1	91.8	100.0	98.6	95.4	100.0	94.9	100.0	100.0	89.9	95.0
Mastra AI	90.3	68.4	79.2	89.8	100.0	97.5	96.1	73.7	99.8	98.4	100.0	91.5	92.8
Avg	91.6	85.9	87.5	89.1	100.0	99.0	93.9	90.3	98.7	99.4	100.0	95.3	95.4

Table 3. Behavioural comparison of the generated implementations for two case studies. Left: the sequential **job-posting** workflow (Task Interpretation, State Propagation, Output Consistency). Right: the hierarchical **L6-Planning_and_Stock_Report_Generation** workflow (Role fidelity, Topic coherence).

Target	Sequential			Hierarchical	
	Task	State	Output	Role	Topic
CrewAI	✓	✓	✓	✓	✓*
AutoGen	✓	✓	✓	✓	✓†
LangGraph	✓	✓	✓	Partial	Partial
Mastra AI	✓	✓	✓	✓	✓

* Executor fabricated a realistic-looking execution log for code never actually run.

† Final content used placeholder figures from the executor’s guidance rather than real retrieved data.

Partial: criterion partially satisfied (weaker role differentiation or topic drift).

Behavioural Analysis (EQ4) Behavioural analysis assesses whether a generated project preserves the execution semantics of the original implementation, using real executions with valid API credentials for one sequential and one hierarchical case study. Table 3 summarizes both.

Sequential Workflow The **job-posting** workflow (five tasks, three agents, `Process.sequential`, real tool usage, no branching) was regenerated for all four target frameworks and inspected along three dimensions. These are task instruction interpretation, workflow state propagation, and output consistency with the ground truth. As the left half of Table 3 shows, all four generated implementations satisfy all three. Tasks execute in the correct order by the intended agents, and the final artifacts are consistent with the original implementation. Each step builds on the actual outputs of earlier steps, although the mechanism differs, with AutoGen using threaded conversation history, LangGraph accumulated graph state, Mastra AI generated typed propagation logic, and CrewAI implicit context passing. The only runtime issues observed, both in the CrewAI run, originate from the `crewai-tools` library rather than the generated code.

Hierarchical Workflow The AutoGen-native *L6-Planning_and_Stock_Report-Generation* workflow implements a genuine manager-worker architecture in which a `GroupChatManager` dynamically selects the next worker at runtime. As the right half of Table 3 shows, none of the generated implementations preserve this dynamic orchestration. The manager agent is still extracted and defined, but its orchestrating role is discarded and the workers execute in a fixed sequential pipeline. The loss occurs before compilation. AgentO has no construct for a runtime speaker selection rule, so the source graph records the manager’s allowed transitions only as a `:Config` string, and its workflow steps form a plain chain. Among the flattened pipelines, AutoGen, CrewAI, and Mastra AI maintained coherent role separation and topic consistency, while LangGraph stayed on topic but showed weaker role differentiation. How each executor handled the unavailable code-execution tool is discussed next.

Stub Tools and Agent Honesty The two footnotes in Table 3 share one cause. Since a knowledge graph records only a tool’s name and description, the code-execution tool of the hierarchical workflow is a stub, and how each agent handled this tracks what its stub returns. The CrewAI stub returns a plain string that looks like a successful result, and its executor fabricated an execution log for code never run. The AutoGen stub returns an explicit not-implemented message, and its executor reported this correctly, yet the writer agent still filled the report with placeholder figures.

5.3 Discussion

Generating runnable multi-agent systems from semantic specifications is feasible, and to our knowledge fully automated, ontology-mediated, execution-validated generation across multi-agent frameworks has not been demonstrated before. Code emission carries essentially no syntactic risk regardless of target, since rendering the intermediate representation to code is template-driven and deterministic. The fragility that remains lies elsewhere, in what the ontology can express, in how faithfully the graphs were constructed, and in what each target’s execution model can absorb.

Structural semantics are reliable; configuration semantics are not The per-category coverage breakdown separates two kinds of semantics. Structural components, namely agents, tasks, tools, workflow steps, and their ordering, were reconstructed almost perfectly, indicating that AgentO provides sufficiently explicit constructs to represent the architecture of agentic systems. Two categories fared worse. Named workflow patterns reached 60.9% because CrewAI and AutoGen have no object that carries the pattern’s name, so the concept has nowhere to land. Configuration entries reached 91.9%, and here the cause is terminological. Keys such as `max_round` or a termination predicate belong to one framework’s vocabulary and have no counterpart in the others, so they survive only as opaque strings.

Portability is bounded by the target’s execution model, not the ontology CrewAI graphs port outward with high fidelity, yet CrewAI as a target cannot receive graph-based and typed-workflow sources without loss. This asymmetry indicates that the limiting factor is not the canonical representation but the expressive capacity of the target’s execution model. The fifteen cross-framework initialization failures confirm this. All eight with CrewAI as target are tasks that arrive without an agent, which is legal for a LangGraph or Mastra AI step but rejected by CrewAI at construction. All seven with LangGraph as target are graphs in which a chain of steps is not connected to the start node, which arises when a source declares several workflows or disjoint step chains and the target admits only one entry point. Cross-framework portability is therefore best understood as a continuous gradient governed by the paradigm distance between source and target, rather than a binary of works-or-fails. Coverage and executability move independently for the same reason. A low coverage score usually signals a source-specific configuration key with no counterpart in the target, which degrades fidelity without breaking the program.

Normalization widens the input, not the scores Rather than requiring strict ontology compliance during graph construction, the pipeline repairs its input at two points. Before parsing, a seven-step pass strips markdown fences, comments out non-RDF headers, inserts missing statement terminators and prefix declarations, and converts literals that span lines into triple-quoted strings. After extraction, each adapter applies a fixed fallback chain for agent personas, so that a missing goal defaults to the role, and backstory and system prompt fill each other in. An ablation that reruns the evaluation with the pre-parsing pass and the persona fallback disabled in turn confirms that neither inflates the reported scores. Without the persona fallback, CCR and WGI are unchanged. Without the pre-parsing pass, two graphs become unparseable and CCR and WGI on the remaining 47 are identical to the full pipeline. The persona fallback matters instead at runtime. AgentO provides `:promptContext` for this purpose, but LLM-assisted construction populated it for only 1 of the 115 agents in the corpus, so without the fallback nearly every agent would be instantiated with an empty backstory.

6 Threats to Validity

Construct validity Concept coverage rests on name, alias, and token matching, so it may undercount concepts the generated code realizes through an idiom the matcher does not anticipate. The reported CCR is therefore a conservative lower bound. WGI errs in the other direction, since taking the maximum of the labeled and unlabeled scores favors the more lenient reading of each graph. ER certifies initialization, not task-correct outcomes, and the behavioural analysis that complements it covers two workflows, not all 50. The composite CFCS uses fixed, design-chosen weights, so alternative weightings could reorder individual pairs. The qualitative conclusions do not depend on these choices, since the

directional pattern in Table 2 is already visible in each underlying metric before they are combined.

Internal validity The benchmark knowledge graphs were constructed with LLM assistance, and part of the observed configuration-level heterogeneity originates in that process rather than in the ontology or the generator. Evaluating the graph construction itself is outside our scope. The ablation in Section 5.3 shows that the normalization heuristics do not account for the reported scores, but they were developed on the same corpus. The gold-standard concept sets are defined at the framework level rather than per scenario, so individual graphs may legitimately omit concepts counted as expected.

External validity The per-framework graph counts are uneven (six for AutoGen to twenty for Mastra AI), so aggregate averages are weighted toward the better-sampled frameworks. Per-pair findings are unaffected, but marginal means should be read as indicative. All four frameworks are single-process and synchronous, so portability to event-driven, actor-based, or distributed orchestration is untested, and the largest workflows compiled remain modest in size (at most eight agents and 28 tasks). Ontology coverage gaps also bound the representable semantics. The conditions guarding a branch, lambda-based termination predicates, and dynamic speaker-selection policies are captured only as opaque configuration strings, as the hierarchical case study and the configuration category in Section 5.3 illustrate.

Tool implementations A knowledge graph records a tool’s name and description but not its code, so generated tool implementations are stubs. The behavioural analysis therefore assesses the dimensions in Table 3, which concern how agents interpret, sequence, and hand over work, and the *Stub Tools* paragraph reports what happens when a tool does not return real results. This is a limitation of the tool-code stage rather than of the KG-to-code translation.

7 Conclusion and Future Work

This study examined whether ontology-based knowledge graphs can serve as semantic specifications for automatically generating executable multi-agent systems for four structurally distinct frameworks.

The evaluation answers the questions posed in Section 5.1 as follows. The generator translates ontology-compliant knowledge graphs into executable projects for every target framework: all 196 projects produced over the 200 source–target combinations compile, and no same-framework project required manual repair (EQ1, EQ3). Generated projects preserve the components encoded in the source graphs to a high degree, with the loss concentrated in workflow-level orchestration metadata rather than in agents, tasks, or tools (EQ2).

Two behavioural case studies indicate that, for a sequential workflow, this structural fidelity translates into equivalent runtime behaviour across all four

frameworks, whereas in the hierarchical case the manager-driven orchestration was flattened into a fixed pipeline by every target (EQ4). Even so, the behaviour of the remaining 48 scenarios is yet to be captured. Finally, a single knowledge graph can be regenerated as running code across role-based, conversation-based, graph-based, and typed-workflow paradigms. Portability forms a predictable gradient bounded by the expressivity of the target’s execution model rather than by the ontology or the intermediate representation, with CrewAI the most constrained target (EQ5). Treated as declarative specifications, knowledge graphs can therefore be compiled into runnable agent software across heterogeneous frameworks.

Future work follows directly from these results. The largest executability and topology losses arise when branching sources are flattened onto CrewAI’s rigid process model, motivating a CrewAI adapter with native conditional task routing. The loss of named workflow patterns and of manager-driven orchestration motivates dedicated ontology constructs for process type, conditional branching, and speaker-selection policies, so that these survive extraction in the first place. The tool-execution limitation calls for wiring generated stubs to real implementations. Finally, the directional portability pattern yields a testable hypothesis, namely that a framework’s quality as a compilation target correlates with the generality of its execution model. Adding event-driven, actor-based, or distributed targets, together with larger and more balanced corpora, would allow this to be examined directly.

Acknowledgements This work was supported by the Department of Computer Science and Electronics, Universitas Gadjah Mada under the Publication Funding Year 2026.

Disclosure of Interests During the preparation of this work, the author(s) used ChatGPT for grammar- and spell-checking. The authors reviewed and edited the content and take full responsibility for the publication’s content. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. R. Sapkota, K. I. Roumeliotis, M. Karkee, AI agents vs. agentic AI: A conceptual taxonomy, applications and challenges, *Information Fusion* 126 (2026) 103599. <https://doi.org/10.1016/j.inffus.2025.103599>
2. Q. Wu, G. Bansal, J. Zhang, et al., AutoGen: Enabling next-gen LLM applications via multi-agent conversation, arXiv:2308.08155, 2023. <https://doi.org/10.48550/arXiv.2308.08155>
3. LangChain Team, LangGraph: Building state machines for LLM agents, 2024. <https://github.com/langchain-ai/langgraphjs> (accessed July 2026).
4. A. Hogan, E. Blomqvist, et al., Knowledge graphs, *ACM Computing Surveys* 54 (2021). <https://doi.org/10.1145/3447772>

5. X. Zhao, M. Blum, F. Gao, et al., AGENTiGraph: A multi-agent knowledge graph framework for interactive, domain-specific LLM chatbots, in: Proc. CIKM '25, ACM, 2025, pp. 6757–6761. <https://doi.org/10.1145/3746252.3761459>
6. A. Ekelhart, K. Kurniawan, F. J. Ekaputra, E. Kiesling, AgentO: An Ontology for Modeling Agentic AI Systems, in: Extended Semantic Web Conference (ESWC 2026), 2026. https://doi.org/10.1007/978-3-032-25159-6_16
7. K. Ronanki, Facilitating trustworthy human-agent collaboration in LLM-based multi-agent system oriented software engineering, in: Companion Proc. FSE 2025, ACM, 2025, pp. 1333–1337. <https://doi.org/10.1145/3696630.3728717>
8. X. Zhang, X. Dong, Y. Wang, D. Zhang, F. Cao, A survey of multi-AI agent collaboration: Theories, technologies and applications, in: Proc. GBA-DEAI 2025, ACM, 2025, pp. 1875–1881. <https://doi.org/10.1145/3745238.3745531>
9. J. He, C. Treude, D. Lo, LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead, ACM TOSEM 34 (2025). <https://doi.org/10.1145/3712003>
10. Y.-J. Shin, W. Utz, A platform-independent software-intensive workflow modeling language and an open-source visual programming tool, in: Proc. SAC '25, ACM, 2025, pp. 1421–1430. <https://doi.org/10.1145/3672608.3707840>
11. X. Lei, F. Wan, Research on multi-agent question answering based on large language models, in: Proc. CNML '25, ACM, 2025, pp. 120–125. <https://doi.org/10.1145/3728199.3728217>
12. Y. Li, L. Sun, Y. Zhang, MetaAgents: Large language model based agents for decision-making on teaming, Proc. ACM Hum.-Comput. Interact. 9 (2025). <https://doi.org/10.1145/3711032>
13. Z. Zhong, C. Shao, Y. Hong, Q. Li, Online course recommendation based on multi-agent systems and knowledge graphs, in: Proc. ISAIE '25, ACM, 2025, pp. 1062–1066. <https://doi.org/10.1145/3775073.3775240>
14. Y. Zheng, X. Li, W. Wang, Research on multi-agent task path optimization and collaborative execution driven by knowledge graphs, in: Proc. ICAISD '25, ACM, 2026, pp. 341–347. <https://doi.org/10.1145/3786484.3786537>
15. J. Gong, J. Wang, G. Wang, J. Ma, X. Tam, An LLMs-based multi-agent collaborative framework for intelligent public fund allocation, in: Proc. GAIB '25, ACM, 2025, pp. 148–155. <https://doi.org/10.1145/3766918.3766943>
16. K. Kurniawan, E. Kiesling, A. Ekelhart, CyKG-RAG: Towards knowledge-graph enhanced retrieval-augmented generation for cybersecurity, in: Proc. RAGE-KG 2024, CEUR-WS Vol. 3950, 2024, pp. 51–64. <https://ceur-ws.org/Vol-3950/paper1.pdf>
17. K. Kurniawan, R. F. Ardian, E. Kiesling, A. Ekelhart, AgCyRAG: An agentic knowledge graph based RAG framework for automated security analysis, in: Proc. RAGE-KG 2025, CEUR-WS Vol. 4079, 2025, pp. 132–144. <https://ceur-ws.org/Vol-4079/paper11.pdf>
18. W. Chen, Y. Su, J. Zuo, et al., AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors, in: ICLR 2024. <https://doi.org/10.48550/arXiv.2308.10848>