

A Modular Multi-Agent LLM Architecture for Text-to-Diagram Generation and User-Guided Refinement

Hartwig Grabowski* 

*Corresponding author: hartwig.grabowski@hs-offenburg.de

Article info

Dataset link: <https://zenodo.org/records/17741490>

Keywords:

diagram generation
multi-agent LLM pipeline
engineering informatics
automated modeling
computational design

Submitted: 24 Feb. 2026

Accepted: 6 May 2026

Available online: 13 May 2026

Abstract

Context: Recent advances in LLM-based diagram generation increasingly rely on coordinated agent systems rather than single-model prompts.

Objective: This work highlights how modular multi-agent architectures improve reliability, semantic grounding, and iterative refinement in text-to-diagram workflows.

Method: We analyze a pipeline composed of specialized agents for interpretation, synthesis, validation, and correction, each contributing a bounded and inspectable transformation.

Results: The agent system provides deterministic validation, structured reasoning, and controlled refinement loops that outperform monolithic LLM generation.

Conclusions: Multi-agent LLM pipelines represent a robust foundation for precise, verifiable diagram generation and serve as a reproducible alternative to single-pass text-to-diagram models.

1. Introduction

Automated diagram generation has evolved from manual drawing to intelligent, text-driven visual synthesis. This progression reflects increasing abstraction from geometric representation to semantic reasoning. We categorize the evolution into four stages:

- Drawing → Diagram
- Template/Pattern → Diagram
- Formal Specification → Diagram
- Text → Diagram

These stages illustrate how the field transitioned from handcrafted representations to symbolic modeling and, more recently, to reasoning-centric LLM pipelines capable of self-verification and stylistic refinement.

1.1. From drawing to diagram

Early design began with hand drawings and sketches – the most fundamental means of visual reasoning. Brito et al. [1] highlight the role of manual drawing in the first 150 years of

engineering, showing how sketching supported problem-solving and concept communication. Taraszkievicz [2] contrasts freehand and digital design tools, underscoring that hand drawing still cultivates spatial thinking and design creativity. Willis et al. [3] bridge sketching and CAD by showing how engineering sketches encode geometric and semantic cues for later automation. Agrawal et al. [4] review hand-drawn diagram recognition, documenting early attempts to computationally interpret sketches. Finally, Donnici [5] integrates hand sketching, digital modeling, and generative AI, revealing how analog creativity and computational representation can co-exist. Autodesk [6] and Trimble [7] represent the commercial lineage of this digital-design evolution.

1.2. From template/pattern to diagram

The next step introduced reusable visual grammars and templates. Rule-based CAD macros and parametric patterns produced diagrams from parameter sets. Later, Generative Adversarial Networks (GANs) [8–11] automated the creation of structured layouts, learning spatial design grammars from data. In parallel, prompt-engineering research by White et al. [12] formulated prompt patterns (e.g., Critic, Refine, Plan) – textual analogs of design templates. These works represent the movement from handcrafted geometry to pattern-based diagram synthesis.

1.3. From formal specification to diagram

Formal and software-engineering research established methods to automatically generate diagrams from structured specifications. Buhr et al. [13] introduced Software CAD, converting software design specifications into statecharts and communication diagrams. Object-oriented modeling (Rumbaugh et al. [14]; Booch [15]; UML spec [16]) and graph-transformation systems (Schürr et al. [17]; France & Rumpe [18]) formalized the transformation from abstract models to diagrams. Domain-Specific Languages (DSLs) [19] and story diagrams [20] extended these principles into executable, visual specification languages.

1.4. From text to diagram

1.4.1. Pre-LLM foundations

Early approaches used syntactic NLP pipelines to generate diagrams from restricted text domains. Ghosh et al. [21] presented an ER-Generator that transformed English text into Entity–Relationship diagrams using handcrafted linguistic rules. Although functional, such methods were brittle and domain-limited, revealing the need for semantic reasoning.

1.4.2. LLM reasoning and prompting

Recent advances introduced structured prompting and multi-step reasoning. Rane et al. [22], Giray [23], Wei et al. [24], and Yao et al. [25] explored techniques such as Chain-of-Thought and Tree-of-Thought prompting that enable LLMs to produce semantically coherent structured outputs. Building upon these foundations, Zhang et al. [26] proposed an LLM agent that uses ReAct prompting and retrieval-augmented generation (RAG) to convert natural-language descriptions into AutoCAD-compatible structural drawings. Their system

is conceptually similar to ours – transforming text into diagrammatic representations – but lacks two elements central to our framework: (1) autonomous feedback loops for verification, and (2) stylistic cognition layers for applying contextual flavors and visual refinement.

2. Our architecture: Multi-agent feedback system for diagram generation

2.1. Motivation

While prior approaches to automated diagram generation typically operate as single-pass systems – producing diagrams directly from prompts or specifications – they lack mechanisms for self-correction, contextual adaptation, and aesthetic refinement. In contrast, our framework (Figure 1) adopts a multi-agent architecture inspired by distributed cognition and reflective reasoning. Each agent operates semi-autonomously within a shared state space, contributing a specialized capability (selection, generation, verification, refinement) to collectively ensure accuracy, coherence, and stylistic quality. This architecture embodies the principle of “reason–generate–validate–refine”, forming a closed feedback loop that mimics expert-level design iteration.

2.2. Agent 3 – Meta-selector

Agent 3 functions as the meta-reasoning component responsible for determining the most appropriate diagram type for a given source input. It begins by retrieving a catalog of available diagram types, each defined by its structural semantics, representational intent,

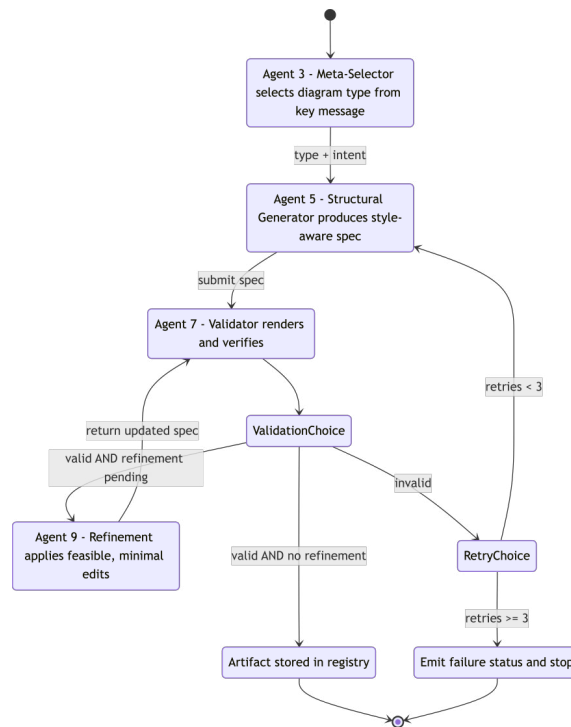


Figure 1. Agentic workflow

and domain of applicability (e.g., UML, ER, flowchart, or sequence diagrams). This catalog serves as the agent’s design space and reference ontology. Subsequently, Agent 3 reads the source material – which may consist of program code, structured data, or descriptive text – from which a diagram is to be derived. It then invokes a large language model (LLM) to perform high-level semantic abstraction, transforming the raw source into a key message that captures its core functional or conceptual meaning. Using this key message, Agent 3 performs a mapping operation between the extracted semantic intent and the entries in its diagram catalog, again by requesting the LLM. The outcome of this reasoning process is the selection of the most contextually appropriate diagram type, which is then passed as structured metadata to downstream agents for generation and validation.

2.3. Agent 5 – Structural generator

Agent 5 acts as the structural synthesis component of the architecture, translating the conceptual representation provided by Agent 3 into a formally defined, executable diagram specification. It not only generates the structural content of the diagram but also integrates contextual style directives, referred to as flavors, which guide the visual and aesthetic realization of the final output.

2.3.1. Structural synthesis

Upon receiving the diagram-type metadata and semantic intent from Agent 3, Agent 5 loads the corresponding syntax–semantics specification for that diagram category. This specification defines the grammar, permissible constructs, and visual conventions required to produce a valid diagram instance (e.g., entity relationships in ER diagrams, message lifelines in sequence diagrams, or cardinalities in UML class models). Using this formal schema, Agent 5 constructs a structured generation prompt and invokes a diagram-generation model (typically an LLM fine-tuned for code synthesis). The model produces a well-formed diagram specification – such as PlantUML, Mermaid, or Graphviz code – encoded according to the chosen visual language.

2.3.2. Integration of flavors

In addition to structural information, Agent 5 receives flavor directives that guide the diagram’s visual appearance and stylistic composition. Flavors represent high-level customization instructions that can be expressed either as free-form textual guidelines (e.g., “use rounded corners for nodes,” “render in black and white,” “omit datatypes from attributes”), or example diagrams that implicitly define the desired appearance through color schemes, font choices, and layout properties. Two categories of flavors are distinguished:

- **Global flavors** – define uniform design conventions across all generated diagrams. They establish the overall aesthetic identity of the system, enforcing rules such as monospaced fonts, consistent color palettes, uniform spacing, or minimalistic edge styles.
- **Local flavors** – apply to individual diagram instances and specify fine-grained refinements or stylistic deviations. These may include color highlighting for selected node types, UML layout constraints, or simply providing an example diagram whose color, font, and style parameters are extracted and reused for the current generation task.

During synthesis, Agent 5 merges the semantic structure (from Agent 3) with the flavor constraints to produce a style-aware structural output. This ensures that every generated

diagram is not only syntactically correct and semantically coherent but also visually aligned with the desired aesthetic conventions. The resulting diagram specification is serialized and transmitted to Agent 7 for validation and possible regeneration.

2.4. Agent 7 – Deterministic validator and artifact constructor

Agent 7 functions as the deterministic validation and artifact-construction layer within the multi-agent feedback loop. Unlike the reasoning-driven agents that precede it, Agent 7 performs purely mechanical verification by executing the appropriate rendering engine for the selected diagram framework. Its primary objective is to ensure that the structural specification produced by Agent 5 is syntactically valid and formally renderable within its designated visualization environment. Upon receiving the diagram specification and diagram-type metadata, Agent 7 invokes the corresponding rendering engine – for example Mermaid.js for Mermaid diagrams, PlantUML.jar (with Graphviz) for UML or architecture diagrams, or specialized back-ends for domain-specific cases. The rendering engine performs a deterministic compilation of the diagram code and produces diagnostic feedback in the form of logs or console messages. If the rendering process fails, Agent 7 encapsulates the extracted diagnostics into a structured feedback message and redirects it to Agent 5, requesting regeneration or localized correction. This establishes a closed validation loop that enables the system to autonomously refine the generated code until a formally valid output is produced. To maintain operational determinism and prevent infinite recursion, Agent 7 enforces a retry limit of three iterations. If no valid rendering is achieved after three validation cycles, the agent terminates the process and emits a failure status report to the system controller. When rendering succeeds, Agent 7 stores the resulting validated artifact – typically an SVG, PNG, or JSON-encoded graph representation – into the system’s artifact registry. Thus, Agent 7 represents the mechanical boundary between generative reasoning and visual materialization, ensuring that only formally consistent outputs enter the refinement stage.

2.5. Agent 9 – Refinement

Agent 9 implements the optional refine-and-retry phase of the workflow. Whereas Agent 7 focuses on validating a given diagram against the chosen engine (Mermaid, PlantUML, or Markdown) and producing the corresponding artifacts, Agent 9 is invoked when refinement instructions are present and a previous validation run suggests that a targeted revision might improve the result. In the control graph, execution flows from Agent 7 to Agent 9 whenever the shared state marks a refinement as pending; after Agent 9 has finished, the updated diagram is passed back to Agent 7 for a final validation pass. Beyond cosmetic edits, Agent 9 is designed for domain-specific, user-driven semantic adjustments (e.g., “remove risk node from mindmap”) that tailor an already valid diagram to its intended purpose.

Internally, Agent 9 orchestrates two LLM-based sub-agents: a feasibility checker and the actual refinement agent. The feasibility step inspects the user’s refinement instructions together with the diagram format, the logical diagram type, an excerpt of the current diagram code, and any available specification text. Its task is to decide whether the requested changes can be realized without violating the underlying diagram language or the documented constraints. If the instructions are deemed infeasible (for example because they conflict with the grammar or with mandatory attributes), Agent 9 aborts the refinement

phase, records a concise explanation and recommendation in the state, and hands control back to the workflow without modifying the diagram.

If the instructions are feasible, Agent 9 invokes the refinement agent, which applies the requested changes to the diagram code in a series of bounded attempts. Each attempt is guided by three layers of context: the explicit instructions, diagram-specific flavor guidance, and general flavor guidance, as well as any validator feedback from earlier runs. The refinement agent aims for minimal, surgical edits that satisfy the instructions while preserving syntactic validity; after each candidate update it re-runs the appropriate validator (Mermaid CLI, PlantUML, or the Markdown pipeline). On success, Agent 9 stores the refined diagram and a short refinement summary in the shared state, marks the refinement as completed, and returns control to Agent 7, which then re-validates the improved diagram and produces the final artifacts.

3. Graphical user interface

To complement the command-line workflow, a lightweight, Tkinter-based graphical front-end provides an interactive interface to the same multi-agent pipeline. The main window mirrors the agent architecture: the first row presents free-form source text on the left and a read-only preview of the generated diagram code on the right; the second row exposes general and diagram-specific flavor guidance; and the third row collects refinement instructions that are forwarded into the refinement phase. Below these inputs, controls allow users to request a diagram-type suggestion (invoking the selection capability) or to run the full workflow including validation and optional refinement.

The interface does not re-implement agent logic. Instead, it delegates execution to the existing command-line entry point by launching background tasks with the appropriate arguments. At run time, the GUI prepares the current source, flavor guidance, and (if present) refinement instructions as temporary artifacts, then invokes the pipeline with the corresponding parameters for source, specification location, flavor configuration, engine, language, and refinement mode. While the pipeline executes, the GUI monitors process output and shared logs, updates a live log pane, and displays an “agent chain” status indicator showing which components are currently active. When execution completes, the interface locates the generated diagram artifact (e.g., Mermaid, PlantUML, or Markdown), displays its raw code in the preview pane, and offers quick actions such as opening the rendered SVG, re-running validation, or triggering a focused refinement pass.

Key advantages:

- **Single source of truth** – All workflow logic (graph wiring, agents, validation, refinement, configuration) resides in the core pipeline; the GUI simply invokes that entry point, avoiding divergent implementations.
- **Consistent behavior** – Graphical and command-line runs exercise the same pipeline with the same parameters and defaults. Any change to the workflow automatically applies to both interfaces.
- **Simpler maintenance** – Fixes and features are implemented once in the shared pipeline. The GUI code focuses on layout, event handling, and process orchestration, not diagram logic.
- **Isolation of concerns** – The UI layer remains thin and does not depend on internal graph state or agent APIs; it prepares inputs, invokes the pipeline, and renders outputs and logs.

- **Easier testing and debugging** – The pipeline can be tested headlessly via the CLI (e.g., in CI) while the GUI acts as a thin shell. Logs and behavior remain consistent regardless of how the workflow is triggered.

Overall, the results demonstrate that the **multi-agent LLM pipeline is robust across a heterogeneous set of LLM back ends**, and that deterministic validation via Agent 7 effectively homogenizes quality: once wrapped in the same generate–validate–refine loop, even diverse models yield comparably reliable diagram artifacts, differing mainly in speed and first-attempt stability rather than in their ultimate ability to satisfy the diagram specification.

4. LLM Performance Evaluation for Text-to-Diagram Generation

We evaluated ten large language models (LLMs) on 30 text inputs: 20 short **easy** prompts (easy001–easy020) and 10 structurally more demanding **complex** specifications. For each model–prompt pair, the multi-agent pipeline executed Agent 5 (diagram generator) followed by Agent 7 (deterministic validator). Agent 7 attempted to render the generated diagram code up to three times and recorded diagnostics such as syntax violations, missing nodes/edges, and invalid attributes.

Each run is logged in `agent7_stats.jsonl` with: – the LLM identifier (`model`), – the proposed diagram type (`diagram_type_proposed`), – the number of validation iterations (`validation_iterations`), – whether the diagram passed on the first attempt (`passed_first_attempt`), – the final validation status (`final_validation_status`), – a structured error profile (`error_profile`), and – end-to-end latency (`latency_ms`). In total, the dataset contains **300 runs** (30 prompts times 10 models).

4.1. Overall model comparison

Table 1 summarizes validation performance and latency for all models across all prompts.

- **Final success:** fraction of runs where Agent 7 reported `final_validation_status = OK` within at most three iterations.
- **First-attempt:** fraction of runs that already passed in the very first validation call (no regeneration from Agent 5 required).
- **Mean iters:** mean number of Agent 7 validation iterations per run.

Table 1. Validation performance and latency for all models across all prompts.

Model	Runs	Final success	First-attempt	Mean iters	Mean latency (s)
anthropic/claude-sonnet-4.5	30	30	28	1.10	44.52
deepseek/deepseek-v3.2-exp	30	28	24	1.17	105.06
glm-4.6	30	25	19	1.13	100.87
google/gemini-2.5-pro	30	29	24	1.23	91.06
google/gemini-3-pro-preview	30	30	29	1.03	84.62
moonshotai/kimi-k2-thinking	30	25	21	1.03	613.12
openai/gpt-5	30	29	24	1.30	158.99
openai/gpt-5-codex	30	28	25	1.23	92.93
openai/gpt-5.1-codex	30	28	28	1.07	49.69
x-ai/grok-4-fast	30	28	23	1.27	29.48
x-ai/grok-code-fast-1	30	26	23	1.37	34.07

- **Mean latency:** average end-to-end time per run (Agent 5 generation + Agent 7 validation).

Overall, all models achieve high final success rates ($\geq 85\%$), but they differ in how reliably they succeed on the **first attempt** and in their **latency**. For example, some models trade slightly lower first-attempt accuracy for shorter runtimes, while others are slower but almost always produce validator-clean code on the first try.

4.1.1. Success rates

The bar chart in Figure 2 shows the final validation success rate for each model. All LLMs eventually produce valid diagrams for the majority of prompts, but models such as `openai/gpt-5-codex`, `anthropic/claude-sonnet-4.5`, and `openai/gpt-5.1-codex` reach **near-perfect final success**.

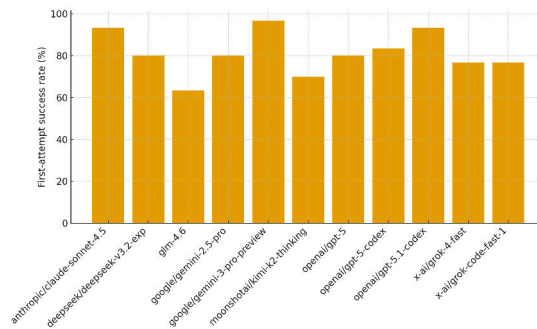
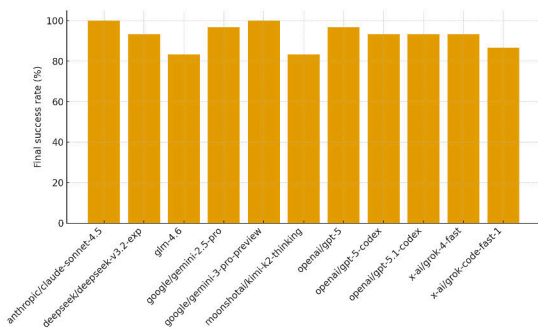


Figure 2. Final validation success rate per model

Figure 3. First-attempt validation success per model

Figure 3 focuses on the stricter metric of **first-attempt success**, i.e., whether the very first diagram produced by Agent 5 passed validation. This metric is particularly relevant for interactive usage, where regeneration loops are undesirable.

4.1.2. Validation iterations

The mean number of validation iterations (Figure 4) is close to 1 for all models, indicating that the pipeline rarely needs to use the full budget of three attempts. Nevertheless, some models systematically require more retries, which suggests that they occasionally produce code with minor syntax or style issues that Agent 7 can catch and correct via regeneration.

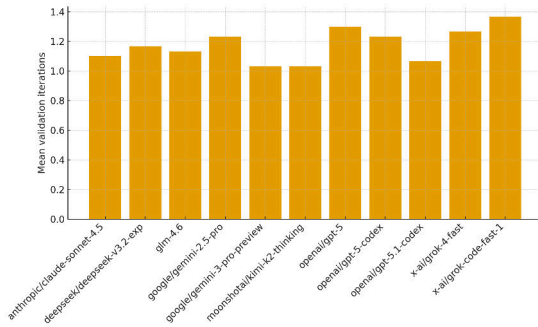


Figure 4. Mean Agent 7 validation iterations per model

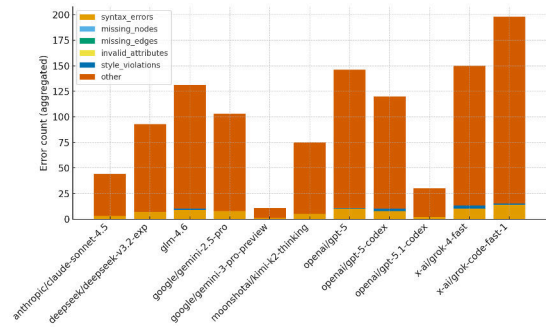


Figure 5. Error profile per model (Agent 7 diagnostics)

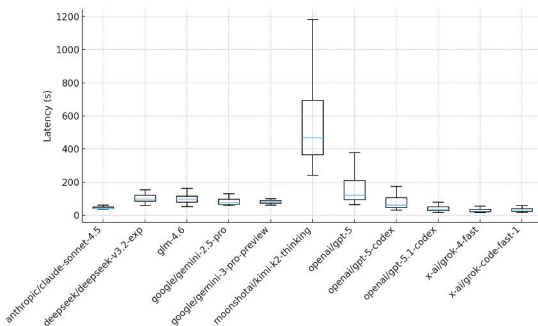


Figure 6. Latency distribution per model

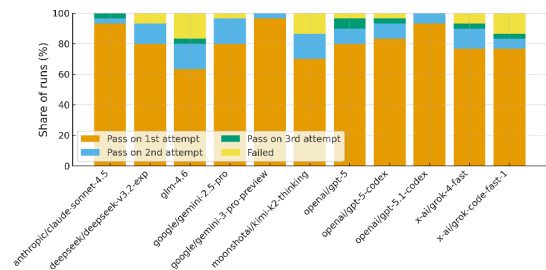


Figure 7. Cumulative success rates by validation iteration (Agent 7)

4.1.3. Latency

Figure 6 reports the latency distribution per model as a boxplot (median, quartiles, and central spread; outliers omitted for readability). We observe that latency varies significantly across providers. Some models provide **fast but slightly less stable** first-attempt behavior, while others are **slower but more consistent**. This highlights a practical trade-off when embedding LLMs into an interactive diagramming tool.

4.2. Error profiles from Agent 7 diagnostics

Agent 7’s structured `error_profile` field allows us to distinguish between different failure modes:

- `syntax_errors`: the rendering engine rejected the diagram due to malformed syntax.
- `missing_nodes/missing_edges`: references to undefined entities or incomplete relationships.
- `invalid_attributes`: invalid or ill-typed attributes for the given diagram language.
- `style_violations`: violations of style or flavor constraints (e.g., forbidden attribute formats).

– **other**: remaining engine errors and miscellaneous diagnostics.

Figure 5 aggregates these errors across all prompts for each model. The stacked bars show that most non-successful runs are dominated by **syntax errors** and a large residual "other" category. The absence of `missing_nodes`, `missing_edges`, and `invalid_attributes` in most models suggests that once the syntax is correct, the structural semantics of the diagrams are usually consistent with the specification.

The relatively small number of `style_violations` indicates that integrating **global** and **local flavors** into the prompt (Section 2.3 of the paper) is generally robust, and that LLMs rarely break purely aesthetic constraints once they have been learned.

4.3. Prompt complexity: easy vs. complex

To assess sensitivity to input complexity, we distinguish between the 20 short **easy** prompts (`easy001`–`easy020`) and the 10 structurally richer **complex** prompts. Table 2 reports success metrics for both groups.

Table 2. Success rates for easy prompts vs. complex prompts

Dataset	Model	Runs	Final success	First-attempt
easy	anthropic/claude-sonnet-4.5	20	100.0%	100.0%
easy	deepseek/deepseek-v3.2-exp	20	100.0%	85.0%
easy	glm-4.6	20	90.0%	65.0%
easy	google/gemini-2.5-pro	20	95.0%	85.0%
easy	moonshotai/kimi-k2-thinking	20	90.0%	80.0%
easy	openai/gpt-5	20	95.0%	85.0%
easy	openai/gpt-5-codex	20	100.0%	90.0%
easy	openai/gpt-5.1-codex	20	100.0%	95.0%
easy	x-ai/grok-4-fast	20	100.0%	85.0%
easy	x-ai/grok-code-fast-1	20	95.0%	85.0%
complex	anthropic/claude-sonnet-4.5	10	100.0%	100.0%
complex	deepseek/deepseek-v3.2-exp	10	100.0%	100.0%
complex	glm-4.6	10	0.0%	0.0%
complex	google/gemini-2.5-pro	10	100.0%	100.0%
complex	moonshotai/kimi-k2-thinking	10	100.0%	0.0%
complex	openai/gpt-5	10	100.0%	100.0%
complex	openai/gpt-5-codex	10	100.0%	0.0%
complex	openai/gpt-5.1-codex	10	100.0%	100.0%
complex	x-ai/grok-4-fast	10	0.0%	0.0%
complex	x-ai/grok-code-fast-1	10	100.0%	0.0%

On the **easy** prompts, most models achieve **final success rates of 95–100%** and high first-attempt rates, indicating that generating syntactically valid flowchart-style diagrams is a largely solved problem for modern LLMs in this pipeline. The more demanding **complex** prompts are represented by ten runs per model, so the numbers should still be interpreted with appropriate caution; however, they serve as an initial indicator that complex specifications can trigger additional failures or retries in some models, especially those optimized for speed.

4.4. Interpretation and implications for the architecture

From the perspective of the multi-agent architecture, the experiment supports three main observations:

1. **Agent 7 as a robustness amplifier.**

Even for models with imperfect first-attempt behavior, the combination of deterministic validation and bounded regeneration significantly increases the final success rate. This confirms the effectiveness of the “generate–validate–regenerate” loop as a guardrail around high-variance generative models.

2. **Models differ more in stability than in raw capability.**

All tested models are capable of producing valid diagram code for the majority of prompts. The main differences lie in how often they produce such code **without** needing retries, and how quickly they do so. For interactive settings, high first-attempt accuracy and low latency are more important than marginal differences in ultimate success rate.

3. **Error structure matters for prompt and flavor design.**

The dominance of syntax-related errors and the scarcity of structural errors (missing nodes/edges) suggest that syntax templates and explicit grammar reminders in the prompts are crucial. Flavor-related violations are rare, indicating that stylistic guidance integrates well with the structural generation step and does not destabilize the pipeline.

Overall, the results demonstrate that the **multi-agent LLM pipeline is robust across a heterogeneous set of LLM back ends**, and that deterministic validation via Agent 7 effectively homogenizes quality: once wrapped in the same generate–validate–refine loop, even diverse models yield comparably reliable diagram artifacts, differing mainly in speed and first-attempt stability rather than in their ultimate ability to satisfy the diagram specification.

5. Summary

We surveyed the evolution of diagram generation from manual sketches and templates to formal model–driven pipelines and, most recently, LLM-based text-to-diagram systems. Building on these foundations, we introduced a modular, multi-agent architecture that operationalizes a reason–generate–validate–refine loop: Agent 3 selects the appropriate diagram type from a catalog; Agent 5 synthesizes a structured, executable specification while integrating global and local flavor guidance; Agent 7 deterministically validates and materializes artifacts; and Agent 9 performs optional, user-guided refinements with feasibility checks and minimal edits.

We implemented this pipeline end-to-end and provided a thin Tkinter GUI that delegates execution to the same CLI entry point, ensuring consistent behavior and simplified maintenance. In an empirical study across 30 diagrams (20 easy, 10 complex) covering 300 runs (30 prompts \times 10 models), most models achieved high final validation success, while differing primarily in first-attempt stability and latency. Agent 7 served as a robustness amplifier by catching syntax issues and enabling bounded regeneration, and flavor constraints integrated cleanly without destabilizing structural correctness.

Future work will extend the diagram catalog and grammars, strengthen semantic grounding through DSLs and graph-transformation back ends, and deepen human-in-the-loop refinement (provenance, safety, and usability studies). We also plan to explore multimodal inputs and visual QA, broaden benchmarks for complex specifications beyond ER diagrams, and release artifacts (prompts, logs, and metrics) to facilitate reproducibility and community adoption.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data and code availability

We have shared our data in a archive available at <https://zenodo.org/records/17741490>.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sector

Declaration of AI assistance in the writing process

During the preparation of this manuscript, the authors used AI-based tools, including ChatGPT and Google Gemini, to support grammar checking, language refinement, and clarity of expression. After using these tools, the authors carefully reviewed and edited the text as necessary and accept full responsibility for the final content of the publication.

References

- [1] E. Brito, R. Póvoas, and P. Providência, “Drawing in the engineering design process,” in *Structures and Architecture*. CRC Press, 2016, pp. 871–878.
- [2] A. Taraszkiwicz, “Freehand drawing versus digital design tools in architectural education,” *Global Journal of Engineering Education*, Vol. 23, No. 2, 2021, pp. 107–112.
- [3] K.D.D. Willis, Y. Li, Y. Sun, and T. Funkhouser, “Engineering sketch generation for computer-aided design,” 2021, arXiv:2104.09621.
- [4] R. Agrawal, K. Jagtap, and R. Kantipudi, “An overview of hand-drawn diagram recognition methods and applications,” 2023, researchGate Preprint.
- [5] L. Donnici, “Rethinking sketching: Integrating hand drawings, digital tools, and generative AI,” *Infrastructures*, Vol. 9, No. 5, 2025, p. 119.
- [6] Autodesk Inc., “AutoCAD software,” 2024. [Online]. <https://www.autodesk.com/products/autocad/overview>
- [7] Trimble Inc., “SketchUp software,” 2024. [Online]. <https://www.sketchup.com>
- [8] F. Luleci, F.N. Catbas, and O. Avci, “Generative adversarial networks for acceleration data augmentation in structural monitoring,” *Journal of Civil Structural Health Monitoring*, Vol. 13, No. 1, 2023, pp. 181–198.
- [9] G.C. Marano, M.M. Rosso, A. Aloisio, and G. Cirrincione, “Review of GANs in earthquake engineering,” *Bulletin of Earthquake Engineering*, Vol. 22, No. 7, 2024, pp. 3511–3562.
- [10] Z. Rastin, G.G. Amiri, and E. Darvishan, “GAN for damage identification in civil structures,” *Shock and Vibration*, 2021, p. 3987835.
- [11] Y. Feng, Y. Fei, Y. Lin, W. Liao, and X. Lu, “Intelligent generative design using rule-embedded GAN,” *Journal of Structural Engineering*, Vol. 149, No. 11, 2023, p. 04023161.
- [12] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea et al., “A prompt pattern catalog to enhance prompt engineering,” 2023, arXiv:2302.11382.

- [13] R.J.A. Buhr, G.M. Karam, C.J. Hayes, and C.M. Woodside, "Software CAD: A revolutionary approach," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, 1989, pp. 235–249.
- [14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [15] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. Benjamin/Cummings, 1994.
- [16] Object Management Group, "Unified modeling language (UML) specification, version 2.5.1," 2017. [Online]. <https://www.omg.org/spec/UML/2.5.1>
- [17] A. Schürr, A. Winter, and A. Zündorf, "The PROGRES approach: Language and environment," in *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999, pp. 487–550.
- [18] R. France and B. Rumpe, "Model-driven development: A research roadmap," in *Future of Software Engineering (FOSE 2007)*, 2007, pp. 37–54.
- [19] M. Mernik, J. Heering, and A.M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, Vol. 37, No. 4, 2005, pp. 316–344.
- [20] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, "Story diagrams: A new graph transformation language based on UML and Java," in *Theory and Application of Graph Transformation*. Springer, 2000, pp. 296–309.
- [21] S. Ghosh, P. Mukherjee, B. Chakraborty, and R. Bashar, "Automated generation of E–R diagram from a given text," in *International Conference on Machine Learning and Data Engineering (iCMLDE)*, 2018, pp. 91–96.
- [22] N. Rane, S. Choudhary, and J. Rane, "Integrating BIM with generative AI," *Buildings*, Vol. 14, No. 1, 2023, p. 220.
- [23] L. Giray, "Prompt engineering with chatgpt: A guide for academic writers," *Annals of Biomedical Engineering*, Vol. 51, No. 12, 2023, pp. 2629–2633.
- [24] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, Vol. 35, 2022, pp. 24 824–24 837.
- [25] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths et al., "Tree-of-thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems*, Vol. 36, 2023, pp. 11 809–11 822.
- [26] X. Zhang, L. Iturburu, J.N. Villamizar, X. Liu, M. Salmeron et al., "LLM agent for structural drawing generation using ReAct prompting and RAG," 2025, arXiv:2507.19771.

Appendix A.

This section summarizes the outputs produced from a set of seven input applications for an IT Expert (complex007.txt). The figures illustrate qualitative differences across the evaluated language models.

Model glm-4.6 by z.ai generated a structured mind map highlighting key skill clusters (Figure A1). Model openai/gpt-5-codex by openai produced a tabular representation of the same information, emphasizing categorical organization (Figure A2). Model gemini-2.5-pro by google, in contrast, created a radar diagram capturing relative skill intensities (Figure A3). These differences demonstrate the variety of diagrammatic representations obtainable from identical textual input.



Figure A1. Mind map generated by model glm-4.6 for complex007 input

Candidate	Primary Focus	Signature Skills & Tools	Team Contributions
Alex Carter	Cloud infrastructure & reliability engineering	AWS, Azure, Terraform, Ansible, GitLab/GitHub CI, Prometheus, Grafana	Produces clear documentation, mentors engineers, embeds security and incident prevention
Priya Shah	Backend services & platform operations	Python, Java, Kubernetes, PostgreSQL, Redis, Kafka, observability tooling	Shapes APIs and SLAs with stakeholders, advances code reviews, streamlines workflows
Daniel Novak	Network architecture & IT security	Campus/DC networking, firewalls, VPNs, zero-trust design, SSO integration	Leads proactive monitoring, supports audits, enforces change management
Laura Meier	Enterprise integration & process automation	ERP/CRM connectors, REST APIs, message queues, data modeling	Bridges business requirements to designs, iterates delivery, establishes governance
Omar El-Sayed	DevOps & platform engineering	Docker, Kubernetes, Helm, Argo CD, IaC, blue-green/canary deployments	Builds self-service platforms, promotes blameless culture, optimizes cost and performance
Hannah Fischer	Data platforms & analytics infrastructure	Apache Airflow, Spark, cloud data warehouses, Python, SQL	Ensures data quality and lineage, secures access, operationalizes ML workloads
Miguel Alvarez	Endpoint management & collaboration services	Microsoft 365, Google Workspace, device management automation	Delivers user-centric support, documents processes, coordinates lifecycle management
Emily Zhang	Software architecture & quality engineering	Domain-driven design, event-driven systems, secure auth, automated testing	Guides design reviews, mentors on clean code, aligns agility with business goals

Figure A2. Table representation generated by model openai/gpt-5-codex for complex007 input

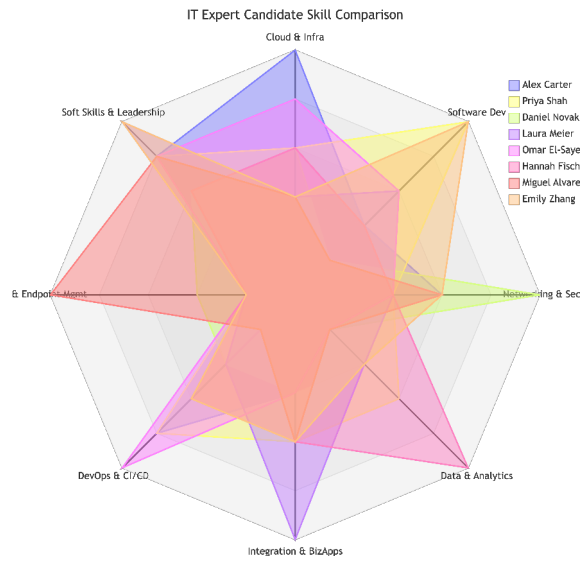


Figure A3. Radar generated by model gemini-2.5-pro for complex007 input

Authors and affiliations

Hartwig Grabowski
e-mail: hartwig.grabowski@hs-offenburg.de
ORCID: <https://orcid.org/0009-0001-4300-2626>
Institute of Machine Learning and Analysis (IMLA),
Offenburg University of Applied Sciences, Offenburg,
Germany