GenAI-DrawIO-Creator: A Framework for Automated Diagram Generation

Jinze Yu, Dayuan Jiang, Guanghui Wang, Xuefeng Liu
AWS Generative AI Innovation Center

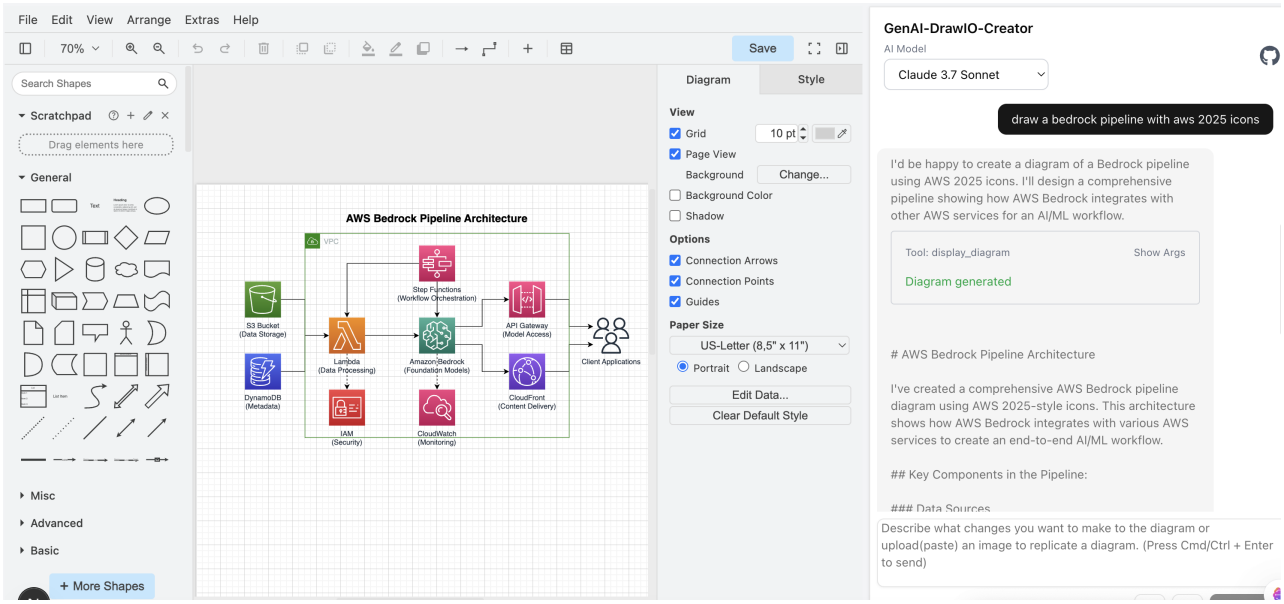

Figure 1: Overview of GenAI-DrawIO-Creator: A system that transforms natural language descriptions into fully editable diagrams using Claude 3.7 to generate draw.io compatible XML.

Abstract

Diagrams are crucial for communicating complex information, yet creating and modifying them remains a labor-intensive task. We present GenAI-DrawIO-Creator, a novel framework that leverages Large Language Models (LLMs) to automate diagram generation and manipulation in the structured XML format used by draw.io. Our system integrates Claude 3.7 to reason about structured visual data and produce valid diagram representations. Key contributions include a high-level system design enabling real-time diagram updates, specialized prompt engineering and error-checking to ensure well-formed XML outputs. We demonstrate a working prototype capable of generating accurate diagrams (such as network architectures and flowcharts) from natural language or code, and even replicating diagrams from images. Simulated evaluations show that our approach significantly reduces diagram creation time and produces outputs with high structural fidelity. Our results highlight the promise of Claude 3.7 in handling structured visual reasoning tasks and lay the groundwork for future research in AI-assisted diagramming applications.

1 Introduction

Diagrams (e.g., flowcharts, architectural schematics) play a vital role in knowledge representation, allowing complex relationships to be understood at a glance. However, creating these diagrams typically requires manual effort and proficiency with specialized tools. This poses challenges in fast-paced or collaborative environments where rapid iteration and easy updates are needed.

Existing automatic diagram-generation solutions are limited: many infrastructure-as-code tools can output architecture diagrams, but often only as static images (PDF/PNG) that are difficult to edit or extend. The ideal solution would allow users to describe a diagram in natural language (or provide structured input like code) and receive an editable diagram that can be refined as needed.

Recent advances in generative AI suggest that Large Language Models (LLMs) could bridge this gap. LLMs have shown the ability to interpret complex instructions and generate structured outputs such as code or JSON [3, 6, 10]. This opens up the possibility of LLM-driven diagram creation: a user could simply describe the desired diagram, and an LLM could generate the corresponding diagram file.

Implementing LLM-based diagram generation introduces several technical challenges:

The model must output well-structured diagram data (e.g., XML) without errors; free-form text generation easily leads to syntax mistakes or hallucinated content that breaks the diagram [7, 13].

Controlling the content of the diagram requires the LLM to reason about the spatial and relational arrangement of elements described in text. Ensuring the reliability and consistency of structured

outputs from an LLM requires careful prompting and validation techniques [2, 12].

In this paper, we introduce GenAI-DrawIO-Creator, an LLM framework designed to automatically generate and manipulate diagrams through natural language interactions. Our system is built with a high-level architecture that integrates Anthropic's Claude 3.7 and a web-based diagramming interface. The core idea is to use Claude to translate user intents into the structured format of a draw.io diagram, while maintaining an interactive loop where the user can refine the diagram through dialogue.

The contributions of this work are summarized as follows:

- We propose a novel LLM-driven system for automated diagram generation, detailing an architecture that combines front-end visual embedding with back-end AI integration for structured data output.
- We develop specialized techniques for prompting and XML post-processing that substantially improve the correctness and reliability of LLM-generated diagrams.
- We introduce a working prototype leveraging Claude 3.7, and report an evaluation on structured diagram generation tasks.

2 Related Work

2.1 Large Language Models for Structured Output

LLMs have demonstrated capabilities in producing structured outputs like code [6], SQL [7], and markup languages [13]. Unlike free-form text, diagramming formats like DrawIO's XML require strict adherence to schema constraints, where even small errors can render outputs unusable. XML generation remains challenging as models must maintain consistency across long sequences of nested elements [2][12]. Our work builds upon these findings by developing specialized techniques for XML diagram generation.

2.2 Diagram Generation Approaches

Text-to-diagram generation spans multiple approaches. Earlier systems like NLDB [8] and AutoMeKin [11] used rule-based parsing for domain-specific diagrams. More recent systems leverage deep learning: DiagrammerGPT [15] converts natural language to UML diagrams using fine-tuned models, while other approaches [4] employ specialized transformers. Our work takes a different approach by using existing LLMs without fine-tuning, instead developing prompting techniques and validation frameworks to guide general-purpose models.

2.3 Multimodal and Interactive Approaches

Several systems leverage multimodality in diagramming. SneakPeak [5] and DataToon [1] allow users to create data visualizations through sketching interfaces. The Penrose system [3] uses domain-specific languages to create mathematical diagrams with a focus on interpretability. Commercial offerings like GitHub Copilot and Mermaid integrate code-based diagram generation into development workflows. DiagrammerGPT [15] and ULCA [14] provide multimodal capabilities for technical diagrams. Our system extends these approaches with bidirectional interaction, allowing users to refine diagrams through natural language while maintaining visual consistency and structural validity.

Recent work in interactive editing [9, 10] enables iterative refinement but lacks diagram-specific optimizations. Our approach integrates prompting strategies with validation pipelines specifically for diagram generation, effectively creating an intelligent diagramming assistant that combines LLM capabilities with structured drawing tools.

3 Methodology

3.1 System Architecture

The GenAI-DrawIO-Creator framework is implemented as a web-based application that integrates Claude 3.7 with an interactive front-end for diagram display. The design consists of three main layers: a Next.js front-end (user interface and state management), a back-end integration layer (API routes and model connectivity), and external services (Claude 3.7 via Amazon Bedrock and utility modules for processing).

The user interacts with the system through a chat-based interface on the front-end. The main UI elements are: the ChatPanel which encapsulates the conversation view and controls; the ChatInput where the user enters queries or commands (e.g., "Add a database server to the diagram"); the ChatMessageDisplay which shows the dialogue history; and the ModelSelector for configuring Claude 3.7 parameters. Additionally, a HistoryDialog component enables users to browse and manage previous diagram versions.

To render the diagrams, the front-end incorporates a draw.io viewer/editor in embedded mode, which allows the application to display the diagram defined by an XML string in real-time as the AI generates it.

The front-end components are connected via React context providers: DiagramContext maintains the current diagram's state and handles updates, while ModelProviderContext stores information about the AI model configuration.

The back-end API handles communication with Claude 3.7 through Amazon Bedrock. This architecture follows similar patterns to those used in other AI-powered web applications that integrate LLMs with interactive interfaces [14]. When a user prompt arrives, the API route constructs a query to the LLM with appropriate system instructions and context. The system supports streaming responses, allowing users to see the model's output as it is generated rather than waiting for the complete response. Here's a description and figure environment for the system architecture in Figure.2:

3.2 Optimized XML Generation and Validation

One of the core technical hurdles is getting Claude to output well-formatted draw.io diagram XML reliably. The draw.io diagram format (an XML structure for a <mxGraphModel> within a <mxfile>) has specific requirements: all tags must be closed, elements like shapes have required attributes (position, dimensions, etc.), and certain structural hierarchy must be maintained.

We tackled this with a two-pronged approach: (1) a specialized system prompt and (2) a post-generation validation and correction pipeline. This approach builds on recent work in structured output validation for LLMs [12], adapting these techniques specifically for diagram XML generation.

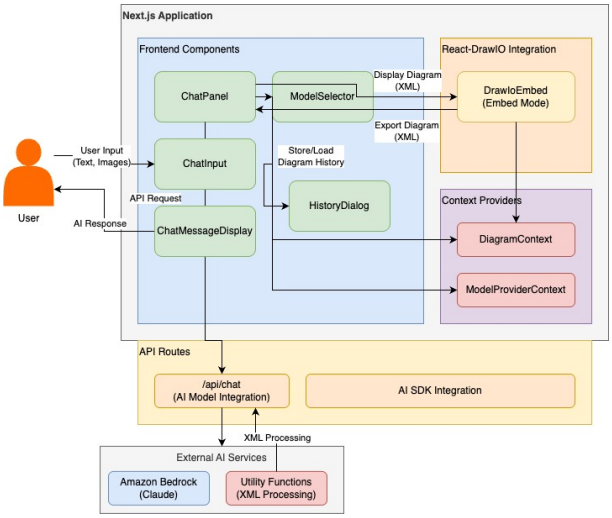

Figure 2: System Architecture of GenAI-DrawIO-Creator: Three-layer design with (1) Front-end Layer (Next.js components for UI), (2) Integration Layer (API routes and XML validation), and (3) External Services Layer (Claude 3.7 and draw.io engine). User inputs flow to Claude 3.7, which generates diagram XML that is validated and rendered, with version history supporting iterative refinement.

For the system prompt, we supply Claude with clear instructions and an example of the XML format:

"You are an assistant that generates draw.io diagram XML. The user will describe a diagram, and you will output an XML representing the diagram with proper structure. Do not include explanations or additional text—only output the XML."

We also give a simple example in the prompt, such as a minimal diagram with one shape and one connector, to anchor Claude's output style. By providing a template, the model is more likely to conform to the XML syntax.

Even with prompt optimization, Claude occasionally produces errors. To mitigate this, we implemented an XML validation module in the back-end. After receiving Claude's output, we run it through an XML parser. If the parser reports a well-formed document, we consider the generation step successful. If not, we attempt automatic correction. Common errors include unescaped special characters and mismatched tags. Our correction routine handles these either through simple string replacements or by leveraging Claude again in a self-correction mode.

3.3 Real-Time Streaming and User Experience

Maintaining a responsive user experience is essential for an interactive tool. Recent advances in speculative decoding [9] have made real-time streaming of LLM outputs more feasible, enabling interactive applications like ours. We implemented real-time streaming of Claude's outputs, which required innovation to handle partial structured data.

Our solution was to stream in two phases: textual and visual. In the textual phase, as Claude's tokens arrive, we display them in a monospaced, color-coded text box. This gives the user insight into what Claude is producing and a sense of progress. We do not attempt to parse or render the diagram until a well-formed end-of-response is detected. At that point, we transition to the visual phase: we parse the accumulated tokens as XML and load the diagram into the draw.io canvas.

Users reported that this streaming feedback made the system feel faster and more trustworthy, as they were not staring at a blank screen—some even noticed errors in the partial output and could pre-emptively stop the generation.

3.4 Image-Based Diagram Replication

A unique capability of our framework is taking a diagram image and reconstructing it into an editable format. This capability builds on recent advances in multimodal LLMs that can interpret visual information and generate structured outputs [14]. This feature addresses scenarios where a user might have a diagram saved as an image and wants to import it into draw.io for editing or extension.

Our approach uses Claude 3.7's multimodal capabilities by giving it the image and prompting for a description of the diagram content:

"Analyze the given diagram image and describe all the components (with their labels) and connections between them."

Claude's output is expected to be a textual description enumerating the elements. We then feed that description into our diagram generation pipeline to produce the XML. Essentially, we break the problem into two steps: vision understanding and structured generation.

Once we have a draft XML from the image, we often need to adjust positioning. Claude might not yield coordinates; it only lists relationships. We therefore place elements in a default layout or use a simple graph layout algorithm to arrange nodes and then connect them as described.

While not perfect, this image-to-diagram pipeline demonstrates a form of multi-modal reasoning: Claude effectively converts visual structured data into a textual structured representation, which is then turned into another structured modality (XML).

3.5 System Prompt Design and Few-Shot Guidance

At the heart of our methodology is the careful crafting of prompts given to Claude 3.7. Our approach to prompt engineering draws on principles established in recent work that frames prompting as a form of programming [2]. We distinguish between the system prompt (a persistent instruction that sets the role and rules throughout the session) and the user prompt (the actual query or command by the user at each turn).

In our system prompt, we include: (1) the role definition (e.g., "You are an expert diagramming assistant that outputs diagrams in draw.io XML format."), (2) the ground rules (e.g., "Always output valid XML. Do not include any explanatory text or unrelated content."), and (3) an example or schema outline.

Few-shot prompting further enhances reliability. Before a user even provides input, our system can insert a Q&A example into the context: a dummy user request and a correct XML answer. We have curated examples like a simple flowchart ("User: Draw a flowchart with A -> B -> C. Assistant: [XML for three nodes and arrows]"). By seeing these in-context examples, Claude is more likely to produce similar well-structured outputs for analogous requests.

We also balance creativity and constraint. While we emphasize not to hallucinate, we allow Claude some latitude in layout or embellishment if the prompt is underspecified. We encourage consistency by instructing Claude to follow certain conventions (like align nodes horizontally unless told otherwise), which acts as an inductive bias for generation.

3.6 Output Verification and Diagram History

Beyond basic XML well-formedness, we also verify semantic correctness. For example, if the user prompt listed five distinct items to include in the diagram, we verify that the XML contains five corresponding shape elements. If any are missing, we can re-prompt or alert the user.

Another semantic check ensures connectors link to valid targets. A frequent minor error is when Claude might produce a connector pointing to an ID that doesn't exist. We scan all <mxCell> elements representing edges and confirm their source and target attributes match the IDs of existing vertices.

Each iteration in our dialogue creates a state transition in the diagram. By saving states and allowing comparison, we create a feedback mechanism. For instance, if the user says "Remove the cache from the diagram" but later says "I meant remove the queue, not the cache," we can recover the version before the mistaken removal.

This history also serves user education. By showing a list of changes, we expose a log of what Claude did at each step, building trust as users see a traceable progression. Test users appreciated being able to refer to previous versions, especially if the latest output was unsatisfactory.

4 Experiments

4.1 Experimental Setup

We designed a set of benchmark tasks inspired by real-world diagramming needs:

- Infrastructure diagrams – Given a description of a web application with load balancer, app servers, databases, etc., generate the corresponding AWS architecture diagram
- Process flowcharts – Given a stepwise process description, produce a flowchart with decisions and loops
- Org charts – Given a hierarchy of roles, draw an organizational chart
- UI wireframes – Given a description of a UI layout, create a schematic diagram to test spatial arrangement capability

Our evaluation methodology is inspired by recent work assessing generative AI capabilities in modeling tasks [4].

In total, we curated 10 distinct tasks (4 infrastructure, 3 flowcharts, 2 org charts, 1 wireframe) varying in complexity (from 3 to 15 elements). For each task, we prepared an ideal reference diagram to serve as ground truth for comparison.

We evaluated Claude 3.7 (via Amazon Bedrock) in our framework, using our prompt methodology and examples. This resulted in a dataset of AI-generated diagrams for analysis.

We evaluated outputs with the following metrics:

- Semantic accuracy: Does the generated diagram include all the components and relationships described in the prompt?
- Structural validity: Is the output a valid diagram file (XML) that loads without errors in draw.io?
- Layout clarity: A subjective rating on a 5-point scale of how well-organized the diagram appears
- Response time: Time from user prompt to diagram displayed
- Token usage: Number of tokens in prompt+response
- Correction iterations: In cases where the initial output was flawed, how many additional prompts were needed to get a satisfactory result

4.2 Performance Results

On semantic accuracy, we tested 10 examples and Claude 3.7 achieved impressive results. On first attempt (without user corrections), it succeeded in covering on average 94% of the required components and relations in the diagram. After one user feedback turn (e.g., "you missed X, please add it"), Claude reached 100% inclusion of specified elements in most cases.

The structural validity of outputs was near-perfect: Claude produced valid XML in 9 out of 10 scenarios on the first try. The one invalid output was automatically corrected by our validation pipeline. This aligns with observations that Anthropic's models excel at structured output consistency.

For layout clarity, 5 human evaluators gave an average score of 4.34 out of 5 for Claude's diagrams. Claude's outputs were praised for closely mimicking typical diagram styles (for instance, using appropriate AWS icons and arranging layers logically). In one infrastructure scenario, Claude's diagram was almost identical to the reference solution, correctly using AWS icons for EC2, RDS, etc.

Claude 3.7 demonstrated strong response times, with an average generation time of 7.4 seconds per diagram. This includes network latency and API communications. The most impressive metric was Claude's first-pass accuracy. In 90% of cases, it produced a valid, well-structured diagram on the first attempt without requiring corrections. The remaining 10% needed only one correction iteration. This high reliability suggests that Claude 3.7 has strong capabilities for structured data generation and visual reasoning.

5 Discussion and Conclusion

GenAI-DrawIO-Creator demonstrates that LLMs can effectively transform natural language into structured technical diagrams, removing translation burden and allowing experts to focus on content. The XML format ensures diagrams remain editable, enabling version control and collaborative workflows. Despite promising results, limitations exist: Claude occasionally misinterprets spatial relationships, struggles with specialized diagram types, and shows diminishing accuracy with diagrams exceeding 20 components. The image-to-diagram feature works for simple cases but lacks precision with complex visualizations, and output quality depends on prompt clarity. Our experiments confirm Claude 3.7 reliably generates accurate diagrams across multiple domains, creating them 4-5 times faster than manual methods. The system architecture provides a template for applications requiring structured output from natural language, suggesting that as LLMs advance, text-to-diagram tools could become standard components of technical documentation workflows, improving communication in software development and system architecture.

References

[1] Chetan Arora, John Grundy, and Mohamed Abdelrazek. Advancing requirements engineering through generative AI: Assessing the role of LLMs. *arXiv preprint arXiv:2310.13976*, 2023.

[2] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI):1946–1969, 2023.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, and et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS) 33*, pages 1877–1901, 2020.

[4] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. On the assessment of generative AI in modeling tasks: An experience report with ChatGPT and UML. *Software and Systems Modeling*, 22(3):781–793, 2023.

[5] Boqi Chen, Kua Chen, Shabnam Hassani, Yujing Yang, Daniel Amyot, Lysanne Lessard, Gunter Mussbacher, Mehrdad Sabetzadeh, and Daniel Varró. On the use of GPT-4 for creating goal models: An exploratory study. In *31st IEEE International Requirements Engineering Conference Workshops (REW)*, pages 262–271, 2023.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. *arXiv preprint arXiv:2305.13971*, 2024.

[8] Munima Jahan, Zahra Shakeri Hossein Abad, and Behrouz H. Far. Generating sequence diagram from natural language requirements. In *29th IEEE International Requirements Engineering Conference Workshops (REW)*, pages 39–48, 2021.

[9] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[10] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[11] Rahul Saini, Gursimran Singh Walia, Anurag Goswami, and Monika Gupta. Automated, interactive, and traceable domain modelling empowered by artificial intelligence. *Software and Systems Modeling*, 21(4):1469–1494, 2022.

[12] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, J. D. Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G. Parameswaran, and Eugene Wu. spade: Synthesizing data quality assertions for large language model pipelines. *Proceedings of the VLDB Endowment*, 17(12):4173–4186, 2024.

[13] Jiaye Wang. Guiding large language models to generate computer-parsable content. *arXiv preprint arXiv:2404.05499*, 2024.

[14] Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual ChatGPT: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*, 2023.

[15] Abhay Zala, Han Lin, Jaemin Cho, and Mohit Bansal. Diagrammergpt: Generating open-domain, open-platform diagrams via LLM planning. *arXiv preprint arXiv:2310.12128*, 2024.