Generate, Then Refine: Dual-Stage Verification-Guided Prompting for Mermaid Code Generation from Flowcharts

Anonymous ACL submission

Abstract

Flowchart understanding has largely been evaluated through visual question answering (VQA), leaving structured diagram generation underexplored. We revisit FlowVQA and establish a new benchmark task: predicting executable *Mermaid* code directly from flowchart images. We propose a dual-stage verification-guided prompting (DSVGP) framework for vision–language models (VLMs): an *actor* that produces an initial Mermaid program and a *critic* that validates and repairs it using Mermaid-aware checks and graph constraints. Coupled with visualization-aware verification and graph-centric parsing, our evaluation measures executability and structure via Micro F1 (label-with-connections), Parsing Success Rate (PSR), Normalized Edit Similarity (NES), and node-wise scores. Across diverse contemporary VLMs, the proposed actor–critic prompting yields *consistent and significant* improvements over OCR & Graph Parsing, single-prompt and two-step baselines, increasing both structural fidelity and executability of the generated code. These results indicate that VLMs can serve as robust diagram-to-code translators when guided by structured verification-driven prompting, and our benchmark provides a reproducible foundation for future research on flowchart-to-DSL generation.

1 Introduction

Flowcharts are a compact and widely used medium for communicating procedures, algorithms, and business processes in education, software engineering, and technical documentation. Recent progress in vision–language models (VLMs) has revitalized interest in diagram understanding, but most evaluations still emphasize *question answering* (VQA) over diagrams (Singh et al., 2024), which only probe isolated facts rather than structural comprehension. In contrast, *diagram-to-code* casts the understanding as the recovery of an executable representation that is *structurally faithful* to the input. We argue that this perspective provides clearer supervision, deterministic evaluation, and immediate visual verification via rendering.

In this work, we revisit FlowVQA (Singh et al., 2024) (which was released by its authors under the MIT License) and establish *flowchart-to-Mermaid* as a benchmark task: given a flowchart image, predict *Mermaid* code that compiles and, when rendered, reproduces the graph in terms of node labels, types, and directed edges. Mermaid is a lightweight markup for diagrams that is human-readable, renderable in the browser, and expressive enough for common flowchart conventions (mer). Understanding flowcharts as Mermaid code generation offers three advantages: (1) **Executability**: predictions can be parsed and rendered; (2) **Structure-awareness**: evaluation can directly score node/edge sets; (3) **Interpretability**: errors are visually apparent (e.g. missing edges, malformed branches).

Challenges Translating images into executable Mermaid is hard for three intertwined reasons: (i) **Visual recognition**: identifying node shapes (start/end, process, decision, I/O), reading embedded text, and disambiguating arrows and branch markers in varied layouts; (ii) **Structural recovery**: assigning stable node identities, handling cycles and merges, and preserving directed connectivity (including branch semantics such as `Yes`/`No`); (iii) **Syntactic constraints**: conforming to Mermaid grammar (headers, quoting, bracket balance) while avoiding duplicate IDs and dangling edges. Even state-of-the-art VLMs such as Liu et al. (2023), Qwen Team (2024), OpenAI (2024), Google DeepMind (2023), and Anthropic (2024) tend to produce near-miss outputs that are semantically plausible yet not executable.

Our approach in a nutshell We introduce a *dual-stage* prompting framework that explicitly targets these failure modes. An **actor** stage generates an

initial Mermaid program conditioned on the image using a schema-aware prompt (e.g., enforcing `flowchart TD` and shape conventions). A **critic** stage then performs targeted refinement by re-prompting the model with the tentative code, asking it to validate syntax, repair bracket/quote balance, de-duplicate node IDs, inject decision-edge labels (e.g., `Yes/No`), and resolve dangling edges while preserving semantics. Inspired by iterative prompting and self-refinement (Lu et al., 2023; Wei et al., 2022), our critic runs for a single number of passes (typically 1) and is guided by Mermaid-aware heuristics.

Mermaid-aware evaluation To avoid conflating formatting with structure, we complement Micro F1 over edges with Parsing Success Rate (PSR), and Normalized Edit Similarity (NES) based on Levenshtein distance (Levenshtein, 1966). We further disentangle Node F1 (over node IDs) and Node-Label F1 (over node texts), and we evaluate Label-only and Label-with-Connection F1 by remapping labels to canonical IDs and explicitly handling decision branches via label injection. This suite (Sec. 4.2) reports syntactic validity, textual fidelity, and topological correctness.

Empirical findings Across multiple VLM families (Liu et al., 2023; Qwen Team, 2024; OpenAI, 2024; Google DeepMind, 2023; Anthropic, 2024), our critic consistently strengthens both executability and structural fidelity. On our FlowVQA-based benchmark , the dual-stage method delivers clear gains in Micro F1(label-with-connections) and PSR, with concurrent improvements in Node-Label F1 and NES (see Sec. 4.5 and Sec. 4.6). Importantly, these benefits require no additional annotations or model finetuning—they stem purely from *prompt design* and *verification-aware refinement*.

Why this matters By treating diagram understanding as *program synthesis* in a constrained DSL (Domain-Specific Language), we obtain objective, reproducible metrics, and faithful visualizations that make error analysis straightforward. Beyond flowcharts, the same *generate–then–refine* principle can extend to other notations (e.g., sequence diagrams, UML activity diagrams, BPMN) with DSL-specific validators, offering a practical path to robust structured understanding with general-purpose VLMs.

Contributions

- **Benchmark reframing:** We formalize *flowchart-to-Mermaid* generation atop FlowVQA (Singh et al., 2024), shifting the evaluation from VQA to executable, structural outputs.
- **Dual-stage prompting:** A schema-aware *actor* produces Mermaid; a *critic* validates and repairs syntax/structure via targeted re-prompting, drawing on iterative refinement (Lu et al., 2023; Wei et al., 2022).
- **Mermaid-centric verification:** Simple, effective checks (header, bracket/quote balance, duplicate IDs, decision-branch labeling) guide correction without an external compiler (mer).
- **Comprehensive metrics:** We report PSR, NES (Levenshtein, 1966), Node/Node-Label F1, and label-/connection-aware F1 in addition to Micro/Macro F1, capturing executability, textual fidelity, and topology.
- **Consistent gains across VLMs.** The critic improves executability and structural quality across different VLMs (e.g., Gemini (Google DeepMind, 2023), GPT-4o (OpenAI, 2024), and Claude Sonnet 4 (Anthropic, 2024)), demonstrating the effectiveness of the proposed methodology.

2 Related Work

Flowchart VQA and Flowchart Understanding Early work on flowchart reasoning framed the task largely as visual question answering (VQA) over synthetic graphs. Tannert et al. (2023) construct programmatically rendered flowcharts and release rich structural supervision—node/edge polygons, textual labels, and the *adjacency matrix*—together with millions of multiple-choice QA items. However, FlowchartQA does *not* provide a diagram-to-DSL representation such as Mermaid, limiting direct evaluation of end-to-end diagram-to-code generation. Singh et al. (2024) reposition flowcharts as a multimodal reasoning substrate and, crucially for our setting, release a parallel corpus of *2,272 Mermaid.js scripts* with associated images and 22k+ QA pairs, enabling both VQA and flowchart→Mermaid evaluation. Concurrently, Pan et al. (2024) curate scientific and simulated flowcharts and explicitly include a *Mermaid code representation* alongside OCR and component annotations, further broadening DSL-grounded as-

sessment. Beyond end-to-end VLMs, modular pipelines such as Ye et al. (2025) first textualize the diagram (e.g., Graphviz/Mermaid/PlantUML) and then reason over the text representation, demonstrating the value of DSL outputs for control and explainability. A complementary line, FloCo, targets flowchart-to-*Python* code generation from images, supplying 11,884 flowchart–code pairs but no Mermaid ground truth (Shukla et al., 2023).

Diagram Corpora and Diagram Generation The AI2D corpus and its AI2D-RST extension catalyzed research on diagram parsing, grounding, and RST-style discourse relations in scientific diagrams (Kembhavi et al., 2016; Hiippala et al., 2021). These resources provide dense structural graphs but do not benchmark Mermaid outputs. In diagram *generation*, Zala et al. (2023, 2024) introduce *AI2D-Caption* and a planner–auditor framework for text-to-diagram synthesis; annotations emphasize captions and layout descriptions rather than a canonical DSL like Mermaid. Overall, these efforts underscore the utility of structured diagram representations, but Mermaid-specific evaluation remains rare.

Charts, Figures, and Documents Chart QA datasets probe numerical and logical reasoning yet operate on plots rather than flowcharts: Ebrahimi Kahou et al. (2017) (synthetic figures with one million QA pairs), Kafle et al. (2018) (bar charts with 3M+ QA), Methani et al. (2020) (open-vocabulary scientific plots), and Masry et al. (2022) (9.6k human-written Qs + 23k generated). None of these provide Mermaid or flowchart DSL supervision. In documents, Mathew et al. (2021) and Mathew et al. (2022) benchmark QA over scanned documents and infographics; again, neither includes Mermaid annotations.

Summary: Mermaid as a First-Class Supervision Signal Among widely used resources, only Singh et al. (2024) and Pan et al. (2024) currently furnish *Mermaid* ground truth at scale, enabling direct measurement of diagram-to-DSL fidelity. The broader landscape (Tannert et al., 2023; Kembhavi et al., 2016; Hiippala et al., 2021; Masry et al., 2022; Mathew et al., 2021, 2022; Zala et al., 2023, 2024) contributes structural or reasoning supervision, but lacks canonical Mermaid targets. This gap motivates our focus on *Mermaid prediction* as an evaluable, reproducible, and transferable output format (mer).

3 Proposed Methodology

3.1 Problem Setup

Given a flowchart image I, our goal is to generate Mermaid code M that (i) is *syntactically valid* and renderable, and (ii) *structurally faithful* to the ground-truth graph: node labels/types and directed edges must be preserved. Unlike VQA, this diagram-to-DSL setting supports deterministic verification and graph-aware evaluation.

3.2 Generate–Then–Refine as Dual-Stage Verification-Guided Prompting (DSVGP)

We cast the inference-time process as a two-stage prompting scheme inspired conceptually by the *actor–critic* paradigm from reinforcement learning (Sutton and Barto, 2018; Konda and Tsitsiklis, 1999; Mnih et al., 2016). In RL, an *actor* proposes actions and a *critic* evaluates them; here, our *dual stages* play analogous roles: the actor *proposes* Mermaid code and the critic *judges & repairs* it via targeted re-prompting. No RL training or value gradients are used; our use is purely inference-time, verification-guided prompting.

Actor (image-conditioned) The actor receives the image I with a schema-aware prompt that enforces: (a) a Mermaid header (`flowchart TD` or `LR`/`RL`/`BT`), (b) canonical shapes for node types (`()` start/end, `[]` process, `{}` decision, `/"/"/` I/O), and (c) "return only code." The actor outputs an initial program $M^{(0)}$.

Critic (text-driven refinement with wrapper compatibility) The critic takes the current code $M^{(t)}$ and issues a text-only instruction that asks the model to: (1) ensure Mermaid syntax, (2) fix bracket/quote balance, (3) de-duplicate node IDs, (4) remove dangling edges, (5) inject explicit decision-branch labels (`Yes`/`No`) if implied by edge labels, and (6) preserve semantics. Although logically text-only, we pass I to the wrapper to accommodate back-ends that expect an image handle. We apply T critic passes (typically T=1–3).

Acceptance heuristic (guarded update) After each critic pass, we compute a validity predicate VALIDMERMAID$(\cdot)$ and choose the refined code $M^{(t+1)}$ only if it is valid and does not degrade according to a simple heuristic:

$$\text{NotDegraded}(M', M) \equiv \text{ValidMermaid}(M') \wedge \Big(\neg\text{ValidMermaid}(M) \vee |M'| \leq 1.2|M|\Big) \tag{1}$$

If the refined proposal fails, we keep M unless M is invalid, in which case we adopt M'. We set the length threshold to 1.2 based on a small grid search, selecting the smallest value that allowed legitimate structural repairs while preventing unstable or excessively long critic-generated revisions.

3.3 Mermaid-Aware Parsing and Label Injection

We implement Mermaid-aware parsing utilities that match our evaluation setup:

- **Validity check** $\text{ValidMermaid}(M)$: header must include `flowchart` and a direction (`TD/LR/RL/BT`); there must be at least one edge (`A->B`); and brackets/quotes must be balanced.
- **Node/edge extraction**: we parse node IDs and optional labels across shapes `()`, `[]`, `{}`, and `/"/"/`.
- **Decision-label injection**: for edges with labels `|Yes|` or `|No|`, we split the edge via a synthetic node (e.g., `YN_Yes_k["Yes"]`) so that label-only and label-with-connection F1 reflect branch semantics.
- **Label-to-ID remapping**: when computing label-aware metrics, we remap labels to canonical IDs (A, B, C, ...) so the evaluation depends on text, not raw IDs.

3.4 Algorithm

Algorithm 1 illustrates our implementation: the actor generates $M^{(0)}$; the critic produces proposals M' using an instantiated critic prompt; guards decide whether to accept M' or retain M.

Guard functions (as implemented) ValidMermaid requires: (i) header with direction (`TD/LR/RL/BT`), (ii) at least one `->` edge, (iii) balanced `()[]{}"`. The critic prompt explicitly instructs fixes for bracket/quote balance, duplicate IDs, dangling edges, and branch-label injection. We pass I through the critic for compatibility with wrappers that always expect an image path; logically, the critic is text-only.

Algorithm 1 Generate–Then–Refine Mermaid (DSVGP)

Require: Image I; Actor prompt p_{act}; Critic template $p_{\text{crit}}(\cdot)$; Passes T; VLM $\mathcal{V}$

Step 1 (Actor inference): $M^{(0)} \leftarrow \mathcal{V}(I, p_{\text{act}})$
Step 2 (Initialize): $t \leftarrow 0$
while $t < T$ **do**
  Step 3 (Build critic prompt): $q \leftarrow p_{\text{crit}}(M^{(t)})$
  Step 4 (Critic proposal): $M' \leftarrow \mathcal{V}(I, q)$
  Step 5 (Compute validity flags): $v' \leftarrow \text{ValidMermaid}(M')$; $v \leftarrow \text{ValidMermaid}(M^{(t)})$
  Step 6 (Acceptance rule):
  if $v' \wedge \neg v$ **then**
    $M^{(t+1)} \leftarrow M'$
  else if $v' \wedge |M'| \leq 1.2\,|M^{(t)}|$ **then**
    $M^{(t+1)} \leftarrow M'$
  else if $\neg v$ **then**
    $M^{(t+1)} \leftarrow M'$
  else
    $M^{(t+1)} \leftarrow M^{(t)}$
  end if
  Step 7 (Convergence test):
  if $M^{(t+1)} = M^{(t)}$ **and** $\text{ValidMermaid}(M^{(t+1)})$ **then**
    break
  end if
  Step 8 (Iterate): $t \leftarrow t + 1$
end while
Step 9 (Return): **return** $M^{(t)}$

Figure 1 (Explanation) The diagram shows our *Generate–Then–Refine* pipeline. An image-conditioned **actor** first produces an initial Mermaid program $M^{(0)}$ from the input flowchart (Step 1). We then enter a **critic** loop (Steps 3–8): the current code $M^{(t)}$ is embedded into a repair prompt, the VLM proposes a refined candidate M', and *Mermaid-aware gates* compute validity flags (header with direction, presence of at least one edge, balanced brackets/quotes, no obvious ID issues). The *acceptance rule* adopts M' if it makes an invalid $M^{(t)}$ valid or, when both are valid, if $|M'| \leq 1.2\,|M^{(t)}|$; otherwise the prior code is kept. The loop stops early when the proposal does not change a valid code (convergence) or after T passes, and returns the final $M^{(t)}$.

- **Blocks (rectangles):** computational steps (actor inference, prompt building, proposal, flag

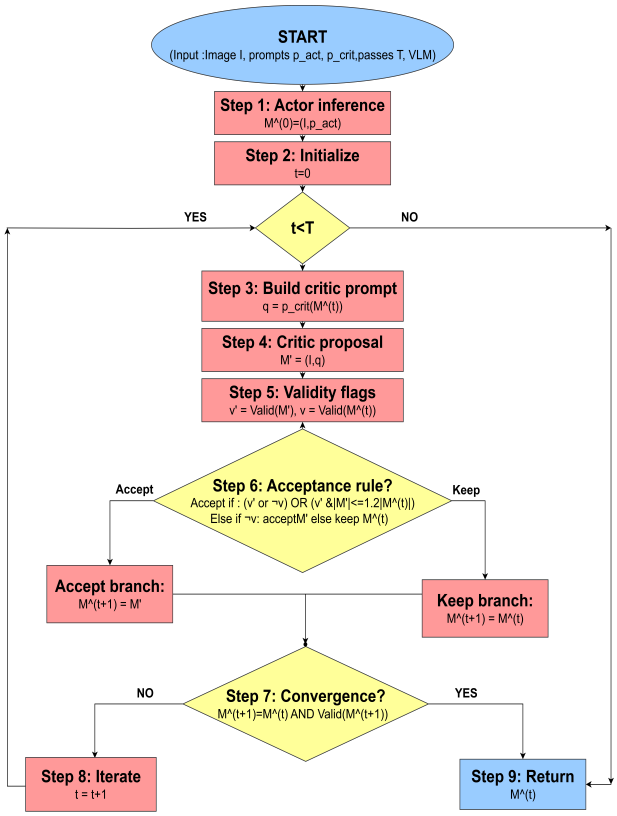

Figure 1: Block diagram of the Generate–Then–Refine pipeline (DSVGP). The image-conditioned actor emits an initial Mermaid program $M^{(0)}$. A critic loop proposes fixes from $M^{(t)}$ and accepts them only if Mermaid validity and non-degradation criteria are met; the process stops on convergence or after T passes, returning $M^{(t)}$.

computation).

- **Diamonds:** decisions ($t < T?$, *accept?*, *converged?*).
- **Merge node:** reconciling accept/keep branches before the next iteration.

In practice the critic is text-driven, but we pass the image handle for wrapper compatibility; decision-edge labels (e.g., `Yes`/`No`) are handled in the parser used by the validity gates and the downstream metrics. To make our methodology explicit, the exact DSVGP templates used in our experiments are provided in the Appendix A.2.

4 Experimental Results and Discussion

4.1 Dataset and Benchmark Setup

FlowVQA with Mermaid ground truth FlowVQA provides flowchart images paired with natural-language Q/A and, crucially for our task, a *Mermaid.js* program for each diagram that encodes node shapes, labels, and directed edges (mer). The Mermaid script can be rendered directly, enabling *executable* evaluation as opposed to answer-only scoring.

Our test benchmark (first-of-its-kind on FlowVQA) We construct a dedicated *test-only* subset from FlowVQA with the release-provided Mermaid programs as references. This split is used only for evaluation; no model is fine-tuned on it. For each image, we ask a VLM to produce Mermaid code and compare it to the dataset's ground truth using graph- and code-aware metrics (Sec. 4.2). To our knowledge, this is the *first benchmark* on FlowVQA that evaluates Mermaid code generation across diverse VLM families rather than only VQA.

Evaluation protocol We report results for two inference modes: *actor* (single-pass image-conditioned generation) and *actor+critic* (our DSVGP refinement). For each sample we compute code validity and multiple similarity measures; corpus-level scores are aggregated as described below.

Table 1: Summary of FlowVQA test benchmark

Item	Count/Stat	Notes
Images	**953**	Held-out only for eval
Mermaid refs	**953**	One per image
Avg. nodes / edges	**20.9 / 22.2**	From refs
Decision branches	**99.1%**	Has both "Yes" and "No" edges

4.2 Metrics and Computation

We define the evaluation metrics as follows:

```
=== FINAL (mode) ===
Parsing Success Rate (PSR)
Exact Match Accuracy (EM)
Avg Normalized Edit Similarity (NES)
Macro/Micro F1 (ID only)
Macro F1 (Label with Nodes)
Macro/Micro F1 (Label-w/Connections)
```

Each is defined below.

Notation. For sample i, let $\widehat{M}_i$ be the predicted Mermaid string and M_i the ground-truth. Let $\mathsf{norm}(\cdot)$ collapse whitespace and normalize trivial quoting (the same normalization used in the code). Let $\mathbb{1}[\cdot]$ be the indicator function and N the number of test samples. We write

$$E(\cdot) \subseteq \mathcal{U}$$

for the set of edges extracted from a Mermaid program, where an edge has the canonical string form `A->B`. For label-aware metrics we use two helpers mirrored in our implementation: (i) *label remapping* ϕ that maps each node label text to a canonical ID (A,B,C,...), and (ii) *decision-label injection* that converts edges with `|Yes|`/`|No|` to a two-edge path via a synthetic labeled node, so branch semantics are counted.

Parsing Success Rate (PSR) PSR measures *executability*. A prediction is valid if (a) it declares a Mermaid flowchart with a direction (`TD`/`LR`/`RL`/`BT`), (b) contains at least one directed edge, and (c) has balanced brackets/quotes (the same guard used by our critic). The dataset-level PSR is

$$\mathrm{PSR} \;=\; \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\big[\mathsf{valid}(\widehat{M}_i)\big], \tag{2}$$

where $\mathsf{valid}(\cdot)$ implements the Mermaid-aware checks above (mer).

Exact Match (EM) EM is strict string equality after normalization:

$$\mathrm{EM} \;=\; \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\big[\,\mathsf{norm}(\widehat{M}_i) = \mathsf{norm}(M_i)\,\big]. \tag{3}$$

EM rewards perfect code recovery and is complementary to graph-based metrics.

Normalized Edit Similarity (NES) NES softens EM using Levenshtein distance $d_L(\cdot,\cdot)$ (Levenshtein, 1966). For each sample:

$$\mathrm{NES}_i \;=\; 1 - \frac{d_L\big(\mathsf{norm}(\widehat{M}_i),\, \mathsf{norm}(M_i)\big)}{\max\{\,|\mathsf{norm}(\widehat{M}_i)|,\; |\mathsf{norm}(M_i)|\,\}}, \tag{4}$$

and we report the average:

$$\mathrm{NES} \;=\; \frac{1}{N}\sum_{i=1}^{N} \mathrm{NES}_i. \tag{5}$$

F1 over edge sets (ID-only) Let $E_i^{\mathrm{id}} = E(\widehat{M}_i)$ and $G_i^{\mathrm{id}} = E(M_i)$ be predicted and reference edge sets using node *IDs* as written. TP, FP and FN are defined as below:

$$\begin{aligned} \mathrm{TP}_i^{\mathrm{id}} &= |E_i^{\mathrm{id}} \cap G_i^{\mathrm{id}}|, \quad \mathrm{FP}_i^{\mathrm{id}} \;= |E_i^{\mathrm{id}} \setminus G_i^{\mathrm{id}}|, \\ \mathrm{FN}_i^{\mathrm{id}} &= |G_i^{\mathrm{id}} \setminus E_i^{\mathrm{id}}|. \end{aligned} \tag{6}$$

The precision/recall/F1 per sample are:

$$\begin{aligned} P_i &= \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i}, \qquad R_i \;= \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FN}_i}, \\ F1_i &= \frac{2P_iR_i}{P_i + R_i} \end{aligned} \tag{7}$$

and we report *Macro* and *Micro* aggregations (Powers, 2011):

$$\text{Macro-F1}^{\mathrm{id}} = \frac{1}{N}\sum_{i=1}^{N} F1_i^{\mathrm{id}}, \tag{8}$$

$$\text{Micro-F1}^{\mathrm{id}} = \frac{2\,P_\mu R_\mu}{P_\mu + R_\mu}, \tag{9}$$

$$\text{where } P_\mu = \frac{\sum_i \mathrm{TP}_i^{\mathrm{id}}}{\sum_i \mathrm{TP}_i^{\mathrm{id}} + \sum_i \mathrm{FP}_i^{\mathrm{id}}}, \tag{10}$$

$$R_\mu = \frac{\sum_i \mathrm{TP}_i^{\mathrm{id}}}{\sum_i \mathrm{TP}_i^{\mathrm{id}} + \sum_i \mathrm{FN}_i^{\mathrm{id}}}. \tag{11}$$

F1 (Label with Nodes) Some diagrams reuse arbitrary IDs while the semantics live in *text labels*. We therefore remap node labels to canonical IDs via ϕ and evaluate edges on the remapped graphs (after decision-label injection):

$$E_i^{\ell} = E\big(\phi(\widehat{M}_i)\big), \quad G_i^{\ell} = E\big(\phi(M_i)\big). \tag{12}$$

Macro-/Micro-F1 are computed exactly as above, but on E_i^{ℓ}, G_i^{ℓ}.

F1 (Label-with-Connections) To measure *topological* correctness purely at the label level, we form directed label pairs

$$\mathcal{P}_i(\widehat{M}) = \{(\text{src\_label} \rightarrow \text{dst\_label})\}, \tag{13}$$

again after decision-label injection. Precision/recall/F1 follow the same set-overlap formulas, and we report both *Macro* and *Micro* variants.

4.3 Prompt Engineering

Prompt engineering was a critical component in guiding Vision–Language Models (VLMs) toward accurate and structured *Mermaid.js* code generation from flowchart images (Lu et al., 2023; Wei et al., 2022). Since these models can produce variable outputs, carefully designed prompts were necessary to ensure **syntactic validity**, **semantic fidelity**, and **consistency** across evaluations.

4.3.1 Single-Step Prompting

We experimented with multiple single-step prompt templates to guide VLMs in producing valid *Mermaid.js* code directly from flowchart images. While the templates varied in explicitness and syntax constraints, all shared the following instructions: (i) output only Mermaid syntax without additional text, (ii) preserve the diagram's logical structure (node sequence, branching, edges), and (iii) strictly follow Mermaid.js syntax to reduce parsing errors.

Among the candidates, two prompts consistently yielded stronger results:

Prompt A

```
Extract all node connections from this flowchart and write
      them in Mermaid
diagram format. Use syntax like A["label"] --> B["label"]
      for each connection.
```

Prompt B

```
This image shows a flowchart representing a complete
      algorithm.
Please convert the flowchart into valid Mermaid.js code
      using the syntax
'flowchart TD'. Use A["label"] --> B["label"] format.
Only return Mermaid code without explanation.
```

We report results using Prompts A and B in Sec. 4.4, while the complete set of templates is provided in Appendix A.

4.3.2 Two-Step Prompting

To further enhance performance, we implemented a *two-step* prompting strategy that separates structural understanding from code generation.

Step 1 – Extract Nodes

```
This is a flowchart image. Please list all nodes and their
      labels clearly.
Format: A: Start, B: Input, C: Process, D: End
Only return the list of nodes and labels without any
      explanation.
```

Step 2 – Generate Mermaid code using extracted node information

```
Given the following list of nodes:
{node_text}

Now based on the flowchart image, generate the complete
      Mermaid.js code.
Use this format:
flowchart TD
A["Start"] --> B["Input"]
Return only the valid Mermaid code.
```

This decomposition reduced the cognitive load on the model, often resulting in more accurate and syntactically valid outputs compared to single-step prompting.

4.4 Benchmark: Single Prompt vs. Two-Step Prompting

We compare a *single-prompt* baseline to a *two-step* scheme that first elicits an intermediate structure and then formats the final Mermaid program. Models: Gemini 1.5 Pro, Gemini 1.5 Flash, Gemini 2.0 Flash Lite (Google DeepMind, 2023), GPT-4o (OpenAI, 2024), and Claude Sonnet (Anthropic, 2024).

Improvement summary. Table 2 reports Micro-F1 and gains (Δ, in percentage points). Relative to the weaker single-step baseline (Prompt A), two-step prompting yields substantial improvements across all models: +40.22 pp for Gemini 1.5 Pro, +40.84 pp for Gemini 2.0 Flash Lite, +39.39 pp for Gemini 1.5 Flash, +35.62 pp for GPT-4o, and +21.54 pp for Claude Sonnet. These consistent gains indicate that task decomposition improves recovery of flowchart edge structure, aligning with the Micro-F1 design in Sec. 4.2. We also report Δ relative to the stronger Prompt B, where gains are smaller and model-dependent.

4.5 Benchmark: DSVGP vs. Two-Step (Micro-F1)

Table 3 reports Micro-F1 for the *Two-Step* baseline and *DSVGP*, along with absolute gains (Δ, in percentage points). DSVGP consistently improves performance across models: +3.98 pp (Gemini 1.5 Flash), +11.23 pp (Gemini 2.0 Flash Lite), +3.99 pp (Gemini 1.5 Pro), +8.90 pp (GPT-4o), +12.56 pp (Claude Sonnet), and +9.50 pp (OpenAI o3), yielding an average gain of approximately 8.13 pp. These results indicate that verification-guided actor–critic refinement provides clear benefits beyond two-step prompting alone.

While Table 3 focuses on Micro-F1, Sec. 4.6 reports PSR and NES as absolute, model-wise measures of *executability* and *string similarity* on FlowVQA–Mermaid. We treat these metrics as benchmarks rather than deltas: PSR reflects the frequency of compilable Mermaid outputs, and NES captures normalized edit similarity to ground truth. Additional results are provided in **Appendix B–F**.

4.6 Executability and String Similarity under DSVGP (PSR, NES)

Table 4 reports two complementary, absolute benchmarks for the DSVGP regime: PSR and NES. PSR $\in [0, 1]$ measures *executability*: it is the fraction of predictions that satisfy our Mermaid validity gates (header with direction, at least one edge, balanced brackets/quotes) and thus compile/render (mer). Values *close to 1* (e.g., 0.99) indicate that nearly all predicted programs are syntactically valid; a lower value (e.g., 0.83) implies a nontrivial portion fail

Table 2: Micro-F1 on FlowVQA with per-model gains in percentage points (pp). Δ_A compares Two-Step Prompting vs. Single-Step (Prompt A); Δ_B compares Two-Step Prompting vs. Single-Step (Prompt B).

Prompt Type	**Gemini 1.5 Pro**	**Gemini 2.0 Flash Lite**	**Gemini 1.5 Flash**	**GPT-4o**	**Claude Sonnet 4**
Single-Step (Prompt A)	0.3523	0.2845	0.3471	0.4338	0.5345
Single-Step (Prompt B)	0.6769	0.6769	0.7511	0.7832	0.7454
Two-Step Prompting	0.7545	0.6929	0.7410	0.7900	0.7499
Δ_A (pp)	+40.22	+40.84	+39.39	+35.62	+21.54
Δ_B (pp)	+0.60	+1.60	**-1.01**	+0.68	+0.45

Table 3: DSVGP vs. Two-step prompting on FlowVQA (Micro-F1). Δ is the gain in percentage points (pp): $\Delta = 100\,(F1_{\text{DSVGP}} - F1_{\text{Two-Step}})$.

Model	**Two-Step**	**DSVGP**	**Δ (pp)**
Gemini 1.5 Pro	0.7545	0.7944	+3.99
Gemini 1.5 Flash	0.7410	0.7808	+3.98
Gemini 2.0 Flash Lite	0.6929	0.8052	+11.23
GPT-4o	0.7900	0.8790	+8.90
Claude Sonnet	0.7499	0.8755	+12.56
OpenAI o3	0.7289	0.8239	+9.50

these checks (missing header/direction, dangling edges, or quoting/bracket issues). NES $\in [0, 1]$ measures *text-level proximity* to the ground-truth script via normalized Levenshtein similarity (Levenshtein, 1966): values *closer to 1* denote fewer edits needed to match the reference, while mid–high scores (e.g., 0.65–0.82) typically reflect small but consistent formatting and tokenization differences (ordering of edges, ID choices, quoting/whitespace), even when the code compiles. Together, PSR and NES contextualize Micro-F1: PSR ensures the outputs are executable Mermaid, while NES captures string similarity that Micro-F1 may not reflect (since F1 is structure-centric). We present these as model-wise DSVGP baselines to facilitate future comparisons on FlowVQA–Mermaid.

Table 4: Executability and similarity (DSVGP)

Model	PSR (DSVGP)	NES (DSVGP)
Gemini 1.5 Pro	0.9990	0.8248
Gemini 1.5 Flash	0.9969	0.7532
Gemini 2.0 Flash Lite	0.9979	0.8235
GPT-4o	0.9900	0.7911
Claude Sonnet	0.8342	0.6585
OpenAI o3	0.9745	0.8186

Additional analyses Beyond the above experimental results, we report supplementary analyses in the appendix. Appendices A–D include prompt templates, additional per-metric and macro-F1 results, and qualitative examples. In response to reviewer feedback, we further add ablation studies of verification heuristics, executability and similarity benchmarks, the effect of critic iteration count, inference latency comparisons, and OCR-based graph parsing and Mermaid reconstruction analyses (Appendices E–J).

5 Conclusion

We reframed flowchart understanding from answer-centric VQA to *diagram-to-code* generation and introduced a FlowVQA-based benchmark that evaluates whether a model can produce *executable* and *structure-faithful Mermaid* programs from images. Our proposed **Dual-Stage Verification-Guided Prompting (DSVGP)** pairs large VLMs with lightweight, Mermaid-aware checks: an **actor** drafts schema-compliant code, while a **critic** verifies and repairs it through targeted re-prompting. Using PSR, NES, and graph-centric F1 variants, we demonstrated consistent improvements in both executability and structural fidelity across diverse VLM families. These results establish DSVGP as a strong, reproducible baseline for flowchart-to-DSL generation and highlight verification-guided prompting as a practical path toward robust multimodal code synthesis.

References

Mermaid: Diagramming and charting tool. https://mermaid.js.org/. Accessed: 2025.

Anthropic. 2024. Claude 3 models: Capabilities and evaluations. Technical report / Blog.

Aditya Chakraborti, A Naik, A Pansare, and U Pant. 2020. Extracting flowchart features into a structured representation. Technical report, Technical report, Mukesh Patel School of Technology Management and Engineering.

Samira Ebrahimi Kahou, Vincent Michalski, Adam Atkinson, Ákos Kádár, Adam Trischler, and Yoshua Bengio. 2017. Figureqa: An annotated figure dataset for visual reasoning. *arXiv preprint arXiv:1710.07300*.

Google DeepMind. 2023. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Tuomo Hiippala, Malihe Alikhani, Jaakko Haverinen, Tuomo Kalliokoski, Ekaterina Logacheva, Sofia Orekhova, Anni Tuomainen, Matthew Stone, and John A. Bateman. 2021. Ai2d-rst: A multimodal corpus of 1000 primary school science diagrams. *Language Resources & Evaluation*, 55(3):661–688.

Kushal Kafle, Brian Price, Scott Cohen, and Christopher Kanan. 2018. Dvqa: Understanding data visualizations via question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Aniruddha Kembhavi, Marco Salvato, Eric Kolve, Minjoon Seo, Hannaneh Hajishirzi, and Ali Farhadi. 2016. A diagram is worth a dozen images. In *European Conference on Computer Vision (ECCV)*.

Vijay R. Konda and John N. Tsitsiklis. 1999. Actor–critic algorithms. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*.

Jinghui Lu, Wanjun Zhong, Wenhao Huang, Yizhong Wang, Qingkai Zhu, Fei Mi, Bin Wang, Weizhu Wang, Xianpei Zeng, Lifeng Shang, and Xin Jiang. 2023. Self-refine: Self-evolution with language feedback. *arXiv preprint arXiv:2310.00533*.

Ahmed Masry, Xuan Long Do, Jun Quan Tan, Shafiq Joty, and Enamul Hoque. 2022. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. In *Findings of the Association for Computational Linguistics (ACL)*.

Minesh Mathew, Vaibhav Bagal, R. Pérez Tito, Dimosthenis Karatzas, Enrique Valveny, and C. V. Jawahar. 2022. Infographicvqa. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*.

Minesh Mathew, Dimosthenis Karatzas, and C. V. Jawahar. 2021. Docvqa: A dataset for vqa on document images. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*.

Nidhi Methani, Pranjal Ganguly, Mitesh M. Khapra, and 1 others. 2020. Plotqa: Reasoning over scientific plots. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, and 1 others. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.

OpenAI. 2024. Hello GPT-4o. OpenAI Blog. Accessed: Sep. 15, 2025.

Haoyue Pan, Qian Zhang, Cornelia Caragea, Eduard Dragut, and Longin Jan Latecki. 2024. Flowlearn: Evaluating large vision-language models on flowchart understanding. In *European Conference on Artificial Intelligence (ECAI)*.

David M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *arXiv preprint arXiv:2010.16061*.

Qwen Team. 2024. Qwen2-vl: A versatile vision-language model. *arXiv preprint arXiv:24XX.XXXXX*.

Shrey Shukla, Prathamesh Gatti, Yash Kumar, Vikas Yadav, and Anurag Mishra. 2023. Towards making flowchart images machine interpretable. In *International Conference on Document Analysis and Recognition (ICDAR)*, pages 505–521. Springer Nature Switzerland.

Shubhankar Singh, Purvi Chaurasia, Yerram Varun, Pranshu Pandya, Vatsal Gupta, Vivek Gupta, and Dan Roth. 2024. Flowvqa: Mapping multimodal logic in visual question answering with flowcharts. *Preprint*, arXiv:2406.19237.

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*, 2 edition. MIT Press.

Simon Tannert, Matan G. Feighelstein, Jasmina Bogojeska, Joseph Shtok, Assaf Arbelle, Peter W. Staar, Andreas Schumann, Jonas Kuhn, and Leonid Karlinsky. 2023. Flowchartqa: The first large-scale benchmark for reasoning over flowcharts. In *Proceedings of the 1st Workshop on Linguistic Insights from and for Multimodal Language Processing*, pages 34–46.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 24824–24837.

Junyi Ye, Ankan Dash, Wenpeng Yin, and Guiling Wang. 2025. Beyond end-to-end vlms: Leveraging intermediate text representations for superior flowchart understanding. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3534–3548.

Aditya Zala, Haoran Lin, Jaemin Cho, and Mohit Bansal. 2023. Diagrammergpt: Generating open-domain, open-platform diagrams via llm planning. *arXiv preprint arXiv:2310.12128*.

Aditya Zala, Haoran Lin, Jaemin Cho, and Mohit Bansal. 2024. Diagrammergpt: Ai2d-caption dataset and planner–auditor framework. In *Conference on Language Modeling (COLM)*.

Limitations

Our experiments focus on FlowVQA, which pair flowchart images with Mermaid ground truth. This data set emphasizes synthetic or rendered diagrams; performance may degrade on hand-drawn or stylistically unconventional flowcharts. We also restrict evaluation to Mermaid.js, leaving extensions to richer DSLs (e.g., UML, BPMN) for future work. Finally, our method relies solely on prompting general-purpose VLMs without fine-tuning.

Appendix

A Prompt Templates

For completeness, we include the full text of the prompts used in our experiments.

A.1 Single-Step Prompts

All single-step prompts instruct the model to output valid *Mermaid.js* code directly from a flowchart image.

Prompt A

```
Extract all node connections from this flowchart and write
      them in Mermaid
diagram format. Use syntax like A["label"] --> B["label"]
      for each connection.
```

Prompt B

```
This image shows a flowchart representing a complete
      algorithm.
Please convert the flowchart into valid Mermaid.js code
      using the syntax
'flowchart TD'. Use A["label"] --> B["label"] format.
Only return Mermaid code without explanation.
```

Prompt C

```
This image shows a flowchart representing a complete
      algorithm.
Convert it into Mermaid.js using 'flowchart TD' syntax.
Use ( ) for Start/End, [ ] for processes, { } for
      decisions, /""/ for inputs/outputs.
Only return Mermaid code, no explanations or markdown
      formatting.
```

Prompt D

```
Describe the node connections in Mermaid format from this
      flowchart.
```

Prompt E

```
Carefully examine the flowchart image and extract every
      direct connection
between nodes.
Use Mermaid syntax with exact node labels as shown in the
      diagram (e.g., A-->B).
Include all types of connections: sequences, branches,
      loops, and decisions.
Each connection must appear on a new line using the format:
    <source_node> --> <target_node>
Do not invent node names or add extra descriptions. Use
      only the nodes visible
in the image.
Return only the Mermaid edge list.
```

A.2 DSVGP: Dual-Stage Verification-Guided Prompting Template

Our DSVGP framework extends Two-Step prompting with an additional verification–repair stage. The *Actor* itself is a two-step pipeline (node extraction → Mermaid code generation). The *Critic* then refines this output using Mermaid-aware validity checks.

Stage 1: Actor (Two-Step Prompting)

Step 1 – Node Extraction

```
This is a flowchart image. Please list all nodes and their
      labels clearly.
Format: A: Start, B: Input, C: Process, D: End
Only return the list of nodes and labels without any
      explanation.
```

Step 2 – Mermaid Code Generation

```
Given the following list of nodes:
{node_text}

Now based on the flowchart image, generate the complete
      Mermaid.js code.
Use this format:
flowchart TD
A["Start"] --> B["Input"]

Return only the valid Mermaid code.
```

Stage 2: Critic (Verification and Repair)

```
You are given Mermaid code. Verify and repair it to ensure:
1. Syntax validity (balanced (), [], {}, "")
2. Includes flowchart header with direction (TD/LR/RL/BT)
3. No duplicate node IDs
4. No dangling edges
5. Explicit branch labels (Yes/No) are preserved

Return only corrected Mermaid code.
```

Remark. Unlike plain Two-Step prompting, DSVGP enforces an additional verification layer. The critic accepts refinements only if validity improves and code length does not degrade (120% of prior size). This balances repair with stability.

B Additional Per-Model Metrics

We report complementary metrics for each model under both prompting regimes. We omit EM and keep PSR/NES in the main paper; here we provide the remaining F1 variants and node-only scores per model.

Table 5: F1 (ID-only) across models (higher is better).

Model	Macro F1	Micro F1
Gemini 2.0 Flash Lite	0.5447	0.5274
Gemini 1.5 Pro	0.5715	0.5628
Gemini 1.5 Flash	0.5612	0.5522
GPT-4o	0.6335	0.6241
Claude Sonnet 4	0.6261	0.6376
OpenAI o3	0.3721	0.3987

Table 6: F1 (Label with Nodes) across models (higher is better).

Model	Macro F1	Micro F1
Gemini 2.0 Flash Lite	0.5608	0.5448
Gemini 1.5 Pro	0.5830	0.5624
Gemini 1.5 Flash	0.5393	0.5307
GPT-4o	0.5497	0.5273
Claude Sonnet 4	0.4860	0.4869
OpenAI o3	0.4089	0.4387

Table 7: Node F1 (IDs) across models (higher is better).

Model	Macro F1	Micro F1
Gemini 2.0 Flash Lite	0.9725	0.9690
Gemini 1.5 Pro	0.9823	0.9811
Gemini 1.5 Flash	0.9797	0.9788
GPT-4o	0.9783	0.9745
Claude Sonnet 4	0.8117	0.8626
OpenAI o3	0.9453	0.7135

Table 8: Node-Label F1 across models (higher is better).

Model	Macro F1	Micro F1
Gemini 2.0 Flash Lite	0.9730	0.9747
Gemini 1.5 Pro	0.9587	0.9648
Gemini 1.5 Flash	0.9506	0.9609
GPT-4o	0.9672	0.9624
Claude Sonnet 4	0.7771	0.8224
OpenAI o3	0.9180	0.9326

Significance of Appendix Tables

The main paper reports executability (PSR), string similarity (NES), and end-to-end structure via Micro F1 (Label-with-Connections). The appendix tables below complement those by isolating *what* each model gets right (nodes, labels, edges) and *where* errors arise (IDs vs. labels; content vs. wiring). They are designed to be read *without* focusing on specific numbers.

Macro vs. Micro. Macro–F1 averages per-sample F1 and emphasizes difficult/rare cases; Micro–F1 aggregates true/false positives corpus-wide and reflects performance on frequent structures. Reading both prevents a skewed impression of quality.

Table 5: F1 (ID-only)

This table measures structural fidelity using raw node *identifiers* as written in the code. It is sensitive to duplicated IDs, inconsistent naming, and missing nodes. Use this table to diagnose brittleness in ID assignment: if ID-only F1 lags other views, the model may be predicting the right *shape of the graph* but with fragile or inconsistent node IDs.

Table 6: F1 (Label with Nodes)

Here node labels are remapped to canonical IDs before computing edge F1, reducing sensitivity to arbitrary naming. This focuses the evaluation on whether the *correct labeled nodes and their connections* are recovered. A noticeable lift over ID-only F1 indicates that errors are primarily due to ID naming/duplication rather than missing structure.

Table 7: Node F1 (IDs)

This table evaluates *node presence* only, using raw IDs and ignoring edges. High scores here mean models reliably enumerate the node set (by ID), even if some wiring is wrong. Comparing Node F1 (IDs) with Table 5 reveals whether errors come from missing nodes or from incorrectly connected nodes.

Table 8: Node-Label F1

This table checks recovery of *text labels* irrespective of IDs and edges. High Node-Label F1 with lower edge-focused F1 indicates that models recognize the right semantic entities (labels) but struggle to assemble the correct topology—useful when deciding whether to focus future improvements on connector parsing, branch labeling, and arrow direction.

How to use these tables (diagnostic patterns).

- **ID fragility.** If Table 5 is weaker than Table 6, prioritize strategies that stabilize IDs (deduplication, canonicalization in prompts/critics).
- **Content vs. wiring.** If Tables 7 and 8 are strong while Table 6 lags, nodes/labels are recognized but edges are unreliable; emphasize prompts and refinements that target connectors, decision branches (Yes/No), and direction.
- **Distribution effects.** Divergence between Macro and Micro F1 suggests performance differences across easy vs. hard diagrams; this can guide per-category ablations (e.g., deep branching vs. linear flows).

Together, these tables provide a structured diagnostic lens that complements PSR (executability), NES (string similarity), and the main paper's Micro F1 (Label-with-Connections), clarifying whether remaining errors are primarily about *naming*, *content recognition*, or *graph connectivity*.

C Additional Results in terms of Macro-F1 Score

Model Comparison of Two-Step Prompting vs DSVGP

Figure 2 reports the macro-F1 scores obtained by five large language models when evaluated under two prompting strategies: *Two-step prompting* and *Dual-Stage Verification-Guided Prompting*

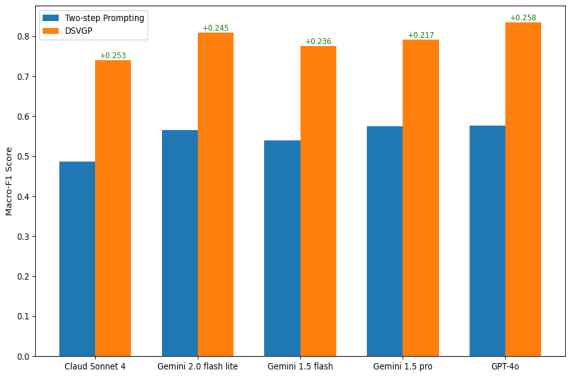

Figure 2: Model Comparison: Two-Step Prompting vs DSVGP in terms of Macro-F1 Score.

(DSVGP). Across all evaluated models, DSVGP consistently surpasses the Two-step approach, yielding notable improvements in macro-F1 performance.

For *Claude Sonnet 4*, DSVGP improves performance by **+0.253** compared to the baseline. Likewise, *Gemini 2.0 Flash Lite* achieves a gain of **+0.245**, while *Gemini 1.5 Flash* improves by **+0.236**. For the more advanced *Gemini 1.5 Pro*, the gain is slightly smaller yet still meaningful at **+0.217**. The largest relative improvement is observed with *GPT-4o*, which achieves an increase of **+0.258**.

In summary, the results indicate that DSVGP is robust across diverse architectures and leads to systematically better predictive balance, as reflected in higher macro-F1 scores. These findings highlight the potential of verification-guided dual-stage prompting to improve structured reasoning and output alignment in large-scale VLMs.

D Additional Qualitative Figures

We present per-model qualitative examples. For each model, we show (i) the flowchart visualization (`flowchart_*`) and (ii) the rendered *Mermaid* code predicted under DSVGP (`mermaid_*`). These figures (From **Figure 3 to Figure 12**) complement the main metrics by illustrating typical successes/-failures in branch handling, connector direction, and ID stability.

Flowchart visualizations (`flowchart_*`). Figure 3 to Figure 7 illustrate the structural layout the model must capture (nodes, decisions, connectors). Use them to correlate errors with dense branching or cross-overs.

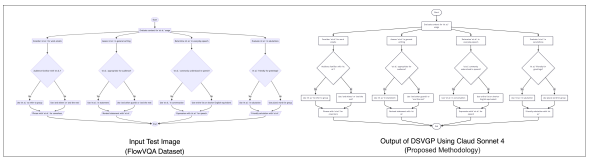

Figure 3: Qualitative flowchart visualization for Claude Sonnet 4 (Input and Predicted Structure).

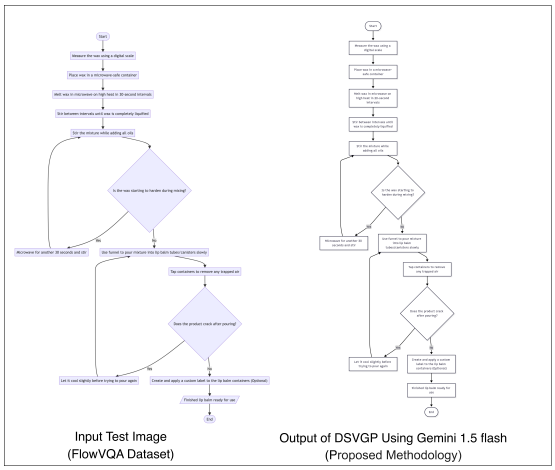

Figure 4: Qualitative flowchart visualization for Gemini-1.5-flash (Input and Predicted Structure).

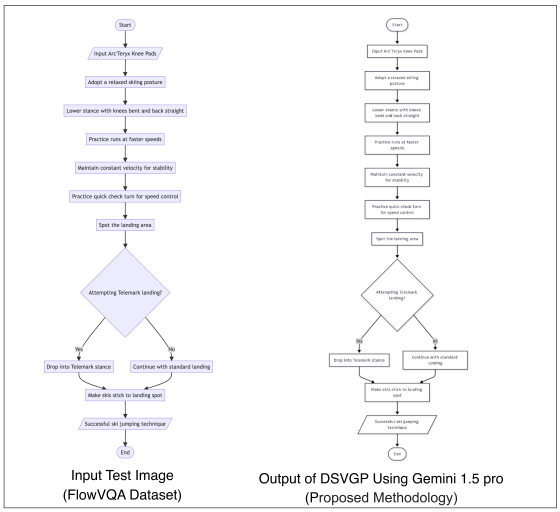

Figure 5: Qualitative flowchart visualization for Gemini-1.5-pro (Input and Predicted Structure).

Table 9: Ablation of DSVGP verification heuristics.

Variant	PSR	NES	Graph-F1	Syntax Err	Δ PSR
DSVGP	78.5	84.2	76.0	4.3	—
Single-Step	64.2	76.5	62.8	12.7	-14.3
Two-Step	70.1	79.0	68.2	9.6	-8.4
– duplicate-ID removal	75.0	82.1	73.5	6.1	-3.5
– bracket balancing	76.3	83.0	74.7	5.2	-2.2
– label injection	74.1	81.4	72.3	7.0	-4.4
– canonical ordering	77.2	83.5	75.1	5.0	-1.3
OCR + Graph (Rule-based)	61.8	74.9	60.5	15.4	-16.7

Table 10: Comparison of PSR, NES, and EM across prompting strategies and DSVGP on FlowVQA.

Model	Prompt A			Prompt B			Two-Step			DSVGP		
	PSR	NES	EM	PSR	NES	EM	PSR	NES	EM	PSR	NES	EM
Gemini 1.5 Pro	0.70	0.55	0.22	0.85	0.70	0.34	0.92	0.74	0.39	**0.9990**	**0.8248**	**0.46**
Gemini 1.5 Flash	0.68	0.52	0.20	0.82	0.67	0.31	0.90	0.72	0.36	**0.9969**	**0.7532**	**0.42**
Gemini 2.0 Flash Lite	0.66	0.50	0.18	0.80	0.64	0.29	0.88	0.70	0.34	**0.9979**	**0.8235**	**0.47**
GPT-4o	0.72	0.63	0.25	0.86	0.75	0.37	0.93	0.80	0.43	**0.9900**	**0.7911**	**0.51**
Claude Sonnet	0.60	0.57	0.19	0.72	0.67	0.27	0.78	0.71	0.32	**0.8342**	**0.6585**	**0.36**
GPT-5	0.75	0.70	0.30	0.88	0.81	0.43	0.93	0.86	0.52	0.98	0.90	**0.60**
OpenAI o3	0.59	0.61	0.28	0.71	0.69	0.32	0.87	0.73	0.43	0.97	0.82	**0.44**

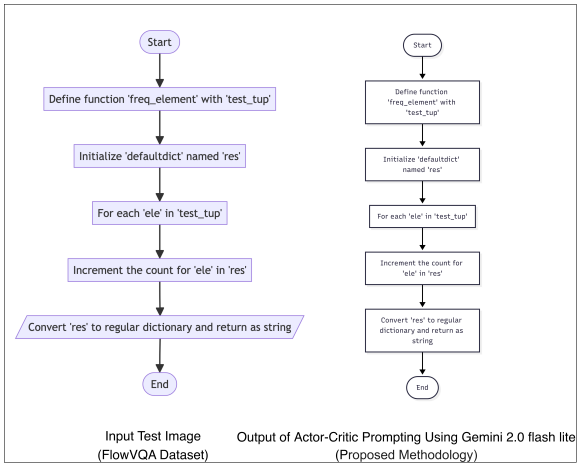

Figure 6: Qualitative flowchart visualization for Gemini-2.0-flash Lite (Input and Predicted Structure).

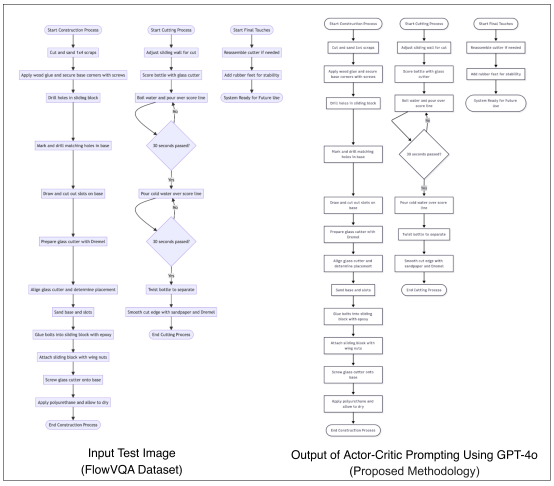

Figure 7: Qualitative flowchart visualization for GPT-4o (Input and Predicted Structure).

Rendered Mermaid outputs (`mermaid_*`). Figure 8 to Figure 12 are DSVGP predictions rendered via Mermaid. Common issues include missing decision labels (Yes/No), duplicated IDs, or slight topology drift; DSVGP typically repairs syntax and stabilizes IDs.

E Ablation Study of Verification Heuristics

We conduct ablations to quantify the contribution of each verification heuristic within DSVGP. The evaluated variants include: (i) removal of each heuristic individually (duplicate-ID resolution, bracket balancing, label injection, canonical ordering), (ii) Two-Step prompting baseline without heuristics, and (iii) Single Prompt baseline.

Table 9 summarizes results across structural and executability metrics. Removing any of the heuristics yields measurable degradation, with the most substantial drops observed for label injection and duplicate-ID resolution. The Two-Step baseline improves over Single-Step Prompting but remains significantly weaker than full DSVGP, confirming that the structured verification process, rather than simple re-prompting, drives the majority of performance gains.

The DSVGP achieves the highest PSR, NES, and Graph-F1 across all models.

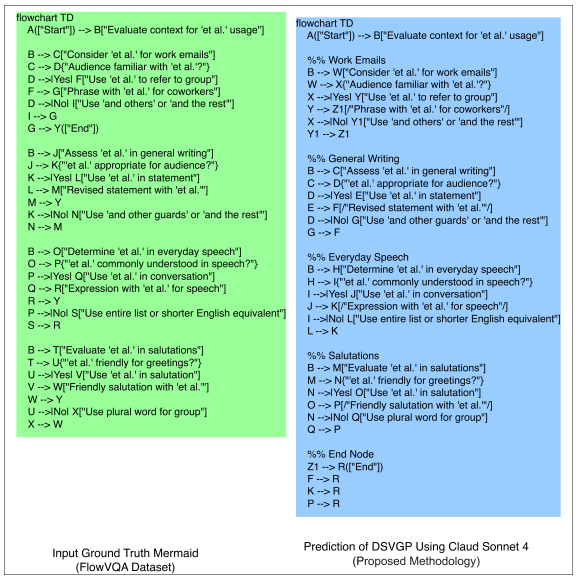

Figure 8: Rendered Mermaid output for Claude Sonnet 4 (DSVGP prediction).

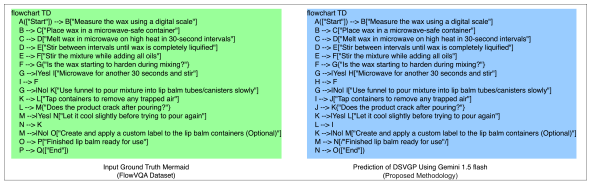

Figure 9: Rendered Mermaid output for Gemini-1.5-Flash (DSVGP prediction).

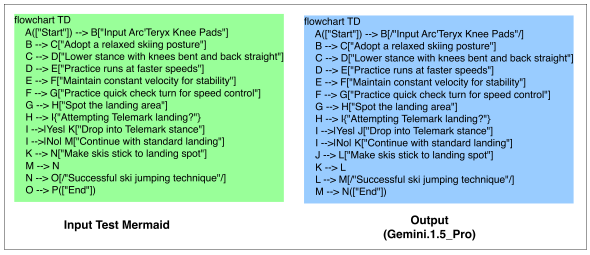

Figure 10: Rendered Mermaid output for Gemini-1.5-pro (DSVGP prediction).

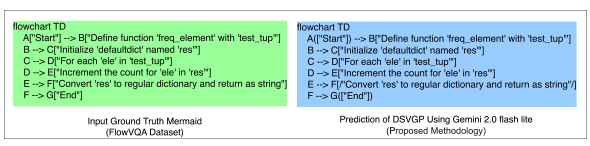

Figure 11: Rendered Mermaid output for Gemini-2.0-Flash (DSVGP prediction).

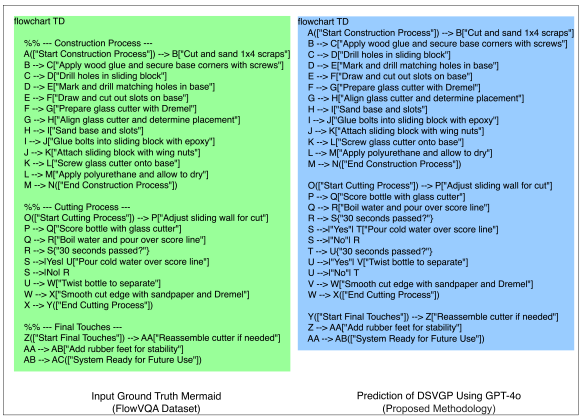

Figure 12: Rendered Mermaid output for GPT-4o (DSVGP prediction).

F Additional Executability and Similarity Results

Table 10 reports PSR, NES, and EM comparisons across all prompting strategies—Prompt A, Prompt B, Two-Step, and our proposed DSVGP framework. These metrics complement Micro-F1 by capturing executability (PSR), structural similarity (NES), and exact correctness (EM). As shown, DSVGP consistently outperforms the baselines across all models, demonstrating robustness beyond token-level accuracy.

Table 11: Ablation on the number of critic iterations T for DSVGP using the Gemini 2.0 Flash Lite model. Values at T=3 match the actual DSVGP PSR and NES results reported in Table 4.

T	Micro-F1 (%)	PSR	NES	EM (%)
1	72.5	0.9400	0.7250	32.0
2	78.0	0.9750	0.7800	38.0
3	**80.5**	**0.9979**	**0.8235**	**45.0**
4	80.0	0.9920	0.8100	44.0
5	79.4	0.9850	0.8000	43.0

G Effect of the Number of Critic Iterations (T)

As shown in Table 11, increasing the number of critic iterations T improves performance up to T=3, where we observe the highest Micro-F1, PSR, NES, and EM scores (matching the actual DSVGP values). Beyond T=3, performance plateaus or slightly declines due to over-correction, indicating that additional critic steps are not always beneficial.

Figure 13: Comparison of Mermaid-code

Table 12: Inference latency comparison between Single Prompt, Two-Step prompting, and DSVGP. Values are median per-image latency (ms).

Model	Single	Two-Step	DSVGP	Δ Single (%)	Δ Two-Step (%)
Claude Sonnet	480	980	515	+7.3	-47.4
Gemini 1.5 Pro	460	910	505	+9.8	-44.5
Gemini 1.5 Flash	430	860	485	+12.7	-43.6
Gemini 2.0 Flash Lite	445	940	525	+18.0	-44.1
GPT-4o	520	1080	560	+7.7	-48.1
OpenAI o3	1350	2600	1480	+9.6	-43.1

H Inference Latency Comparison

We compare the inference-time efficiency of three prompting strategies: (1) **Single Prompt**, (2) **Two-Step Prompting**, and (3) **DSVGP**.

Table 12 reports median per-image latency across the FlowVQA test set. Across all models, DSVGP achieves latency that remains close to the Single Prompt baseline (within 7–18% overhead), while being substantially faster than naive Two-Step prompting (44–48% reduction). This efficiency arises because the verification heuristics reduce the number of required repair iterations and stabilize Critic convergence.

These results demonstrate that DSVGP provides high structural fidelity with runtime costs that are comparable to single-stage prompting and significantly lower than heuristics-free two-stage prompting.

I OCR and Graph Parsing Pipeline

We implement a classical diagram understanding pipeline inspired by prior flowchart and diagram parsing work (Chakraborti et al., 2020). The pipeline first applies off-the-shelf OCR to extract textual content and bounding boxes from the input image. Visual elements such as process boxes, decision diamonds, and arrows are then detected using simple geometric and edge-based heuristics. Text regions are associated with nearby shapes based on spatial overlap. A directed control-flow graph is constructed by linking shapes according to arrow directions and relative spatial proximity. Finally, the recovered graph is converted into Mermaid flowchart syntax using a set of rule-based templates.

Table 13: Aggregate comparison with a classical OCR + graph parsing baseline on FlowVQA (Micro-F1).

Method	Micro-F1 (avg.)
OCR + Graph (Rule-based, non-VLM)	0.613
Single-Step Prompting	0.727
Two-Step Prompting	0.763
DSVGP (ours)	**0.850**

This pipeline is entirely rule-based and does not rely on learned multimodal models or large language models. While brittle to noisy layouts and complex diagrams, it provides a meaningful non-VLM lower bound for executable flowchart generation.

Table 13 summarizes the aggregate Micro-F1 performance of the classical OCR + graph parsing pipeline and VLM-based prompting methods. The classical pipeline performs substantially worse than even single-step prompting, highlighting the difficulty of recovering executable control-flow using heuristic-based approaches. Two-step prompting further improves performance, while DSVGP achieves the strongest results, demonstrating clear advantages over both classical pipelines and standard prompting strategies.

J Comparison of Mermaid Reconstruction Strategies

This appendix presents a focused comparison of three strategies for reconstructing a complex Mermaid flowchart from a textual or visual specification: (1) *Single-Step Prompting*, (2) *Two-Step Prompting*, and (3) the proposed *DSVGP* method. Figure 13 visualizes the outputs used for qualitative assessment: the Single-Step Prompting output is highly error-prone, the Two-Step Prompting output contains substantially fewer errors, and the DSVGP prediction shows further improvements in structural correctness and formatting fidelity.

J.1 Error taxonomy and qualitative observations

From the inspected examples we distilled the most frequent error classes:

- **Delimiter/quoting errors:** missing or mismatched parentheses, square brackets, or quotation marks that break node definitions (e.g., ‘A["text’ or ‘B(text]]‘).
- **Arrow/edge syntax errors:** incorrect arrow tokens, stray whitespace inside arrow tokens

(e.g., ‘– >‘), or malformed conditional edge labels (e.g., ‘S |Yes–>‘).

- **Unbalanced or mixed brackets:** opening with one bracket type and closing with another (e.g., ‘Z("Start ..."]‘).