Documentation Helps, But Leaves a Gap: Diagnosing LLM Generation for Evolving DSLs

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are increasingly used to generate executable artifacts in evolving domain-specific languages (DSLs), including Mermaid.js diagrams. Current documentation often suppresses stale syntax and improves renderability, but many outputs still expose a *render–constraint gap*: they render while violating task-critical structural constraints, or fail because models mix current evidence with stale or cross-formalism priors.

We study this gap as a measurable problem in documentation-aware DSL generation. We introduce DIAGRAM-FAIL-BENCH, a diagnostic benchmark of 630 Mermaid tasks across 21 diagram types, plus a 120-task PlantUML transfer set. Each instance records conflict targets, forbidden legacy patterns, required facets, machine-checkable constraints, and semantic checkpoints, enabling evaluation of executable success, stale syntax, facet coverage, evidence compliance, and render–semantic gaps.

We further propose RA-RAG, a render-aware mitigation framework combining task-aware retrieval, evidence-grounded syntax contracts, deterministic render validation, structured failure diagnosis, and repair-oriented retrieval. We use RA-RAG primarily as an experimental probe. Results show that current documentation helps but remains insufficient, while syntax/facet anchoring and machine-check repair substantially improve executable constraint success, with runtime and renderability trade-offs.

1 Introduction

Domain-Specific Languages (DSLs) such as Mermaid.js are widely used in technical documentation, software design, research writing, and educational materials (Mermaid, 2026; Mermaid Contributors, 2026b). Their appeal is practical: a user can describe a diagram in natural language and obtain compact diagram code that can be rendered directly. As LLM-based assistants enter writing and development workflows, users increasingly expect these models to produce diagram code that is both semantically plausible and syntactically valid under the current rendering engine.

This expectation is difficult to satisfy because Mermaid is an *evolving* DSL. Across versions, its grammar and recommended usage have changed through renamed roots, altered structural constraints, newly introduced diagram families, and stricter configuration requirements (Mermaid, 2026; Mermaid Contributors, 2026c). A model may therefore have learned historical Mermaid syntax during pretraining while being asked, at inference time, to obey a newer specification. Providing current documentation helps: it can reduce obvious legacy roots, improve diagram-type selection, and raise render success. But documentation alone does not guarantee that the model retrieves every task-critical syntax facet or instantiates the retrieved rule correctly.

We call the legacy-prior component of this problem a **stale knowledge conflict**. Unlike ordinary factual errors, stale knowledge conflicts in DSL generation often become *executable failures*: a deprecated root, unsupported token, or cross-formalism drift can make the generated program fail validation or rendering. Our broader focus is the **render–constraint gap**: even after documentation-aware generation improves renderability, many outputs still fail deterministic machine checks or omit structural and semantic requirements. Such failures arise from incomplete facet retrieval, evidence non-compliance, structural violations, semantic loss, and repair drift.

Why standard solutions are insufficient. Fine-tuning or continual learning can refresh model knowledge, but must be repeated whenever the target DSL evolves. Standard Retrieval-Augmented Generation (RAG) injects current documentation (Lewis et al., 2020), yet usually optimizes prompt-level relevance rather than coverage of

the syntax facets required by a specific diagram. Generic self-correction and self-debugging can use execution feedback (Madaan et al., 2023; Chen et al., 2024; Gou et al., 2024), but raw error logs often induce shallow retries, broad rewrites, or new syntax drift.

The main novelty of this paper is therefore not simply another RAG pipeline. Instead, we study *when and why* LLMs fail to follow evolving formal specifications after documentation has been supplied. We ask: (i) how much current documentation reduces stale syntax and improves renderability; (ii) how large the remaining render–constraint gap is; (iii) whether failures come from missing facet evidence, evidence non-compliance, or structural/semantic loss; and (iv) which render-aware mitigation components reduce the gap without excessive runtime or repair drift.

To answer these questions, we introduce DIAGRAM-FAIL-BENCH, a diagnostic benchmark for evolving Mermaid generation. It contains 630 tasks across 21 diagram types. Each instance includes conflict targets, forbidden legacy forms, required syntax facets, machine-checkable rules, and semantic checkpoints; hard and extreme items additionally include audited reference answers. We also propose RA-RAG (Render-Aware RAG), not as a general-purpose new RAG architecture, but as a controlled diagnostic intervention for isolating retrieval miss, evidence non-compliance, and repair drift. Its distinguishing interventions are facet-coverage-oriented retrieval rather than ordinary top-k relevance, an explicit syntax/facet contract rather than a generic instruction, and structured render/checker traces that trigger repair-specific retrieval rather than raw-error retry. We evaluate these mechanisms through ablations, adversarial variants, and oracle controls so that the contribution is measured as failure decomposition rather than as a monolithic system claim.

Contributions.

1. We formulate the **render–constraint gap** in evolving DSL generation: documentation improves renderability, but residual failures arise from stale syntax, missing facets, evidence non-compliance, and structural/semantic constraint loss.
2. We introduce DIAGRAM-FAIL-BENCH, a diagnostic Mermaid benchmark with 630 tasks across 21 diagram types, metadata for conflict targets, forbidden legacy syntax, required facets, machine checks, and semantic checkpoints, plus a 120-task PlantUML transfer set.
3. We propose diagnostic metrics for evolving DSL generation, including executable machine success, stale syntax rate, facet coverage, evidence compliance, repair localization, and render–semantic gap.
4. We propose RA-RAG as a render-aware mitigation framework and use controlled experiments, component ablations, and oracle variants to identify which mechanisms reduce the residual gap.

2 Related Work

Code, DSL, and structured artifact generation. Code LLMs are commonly evaluated with executable benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CodeXGLUE (Lu et al., 2021), and Spider (Yu et al., 2018). These benchmarks use execution as a correctness signal, but mostly target stable languages, unit-testable functions, or fixed schemas. Diagram DSLs such as Mermaid are renderer-centered, version-sensitive, and often non-Turing-complete: correctness depends on parser/renderer acceptance and diagram-specific structural constraints. Constrained decoding and schema-enforced generation, such as JSON-SchemaBench (Geng et al., 2025), reduce output-format violations, but evolving diagram DSLs additionally require version-sensitive syntax grounding, renderer-level validation, and task-specific constraint checks.

Retrieval, knowledge conflicts, and repair. Retrieval-Augmented Generation injects non-parametric evidence into generation (Lewis et al., 2020), but retrieved evidence may conflict with parametric knowledge. Prior work formalizes conflicts in QA (Longpre et al., 2021), surveys context-memory conflicts (Xu et al., 2024), and studies the tug-of-war between model priors and external evidence (Wu et al., 2024). Recent systems address imperfect retrieval and conflicts through adaptive decoding, soft prompting, or hidden-state intervention (Wang et al., 2025a; Yuan et al., 2024; Wang et al., 2025b; Choi et al., 2025; Ma et al., 2026). Our setting differs because the question is not only whether a statement follows evidence, but whether a generated DSL artifact is accepted by a versioned renderer and satisfies formal constraints. Work on self-feedback and tool-interactive repair, including Self-Refine, Reflexion, Self-Debugging,

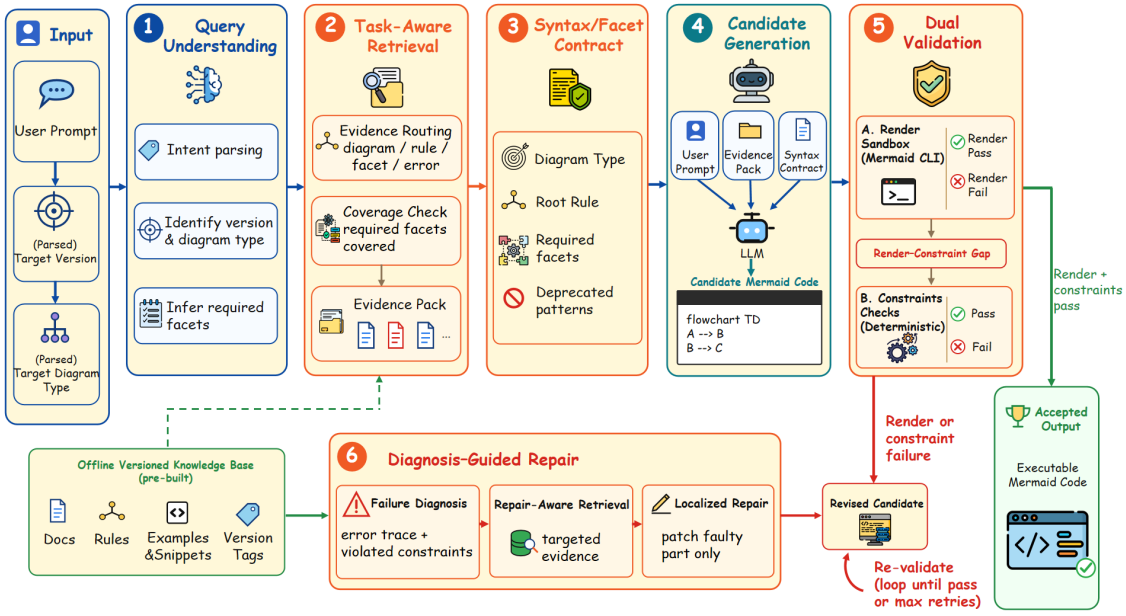

Figure 1: RA-RAG overview. The framework parses task intent, retrieves version-tagged syntax evidence with facet coverage, builds an evidence-grounded syntax contract, validates generated Mermaid code with render and machine checks, and uses structured failure traces for repair-specific retrieval and localized revision.

CRITIC, and LEVER (Madaan et al., 2023; Shinn et al., 2023; Chen et al., 2024; Gou et al., 2024; Ni et al., 2023), is also closely related. We differ by treating validation failures as diagnostic evidence: renderer and machine-check feedback are mapped to violated syntax facets, used for repair-specific retrieval, and tested through ablations and oracle controls.

Diagram generation and evaluation. DiagrammerGPT studies text-to-diagram generation through LLM planning and rendering (Zala et al., 2024). MermaidSeqBench evaluates NL-to-Mermaid sequence diagrams with human verification and LLM-as-a-judge dimensions such as syntax correctness, activation handling, error handling, and usability (Shbita et al., 2025). DIAGRAM-FAIL-BENCH is complementary: it covers 21 Mermaid diagram families and explicitly targets evolving, version-sensitive syntax. We separate deterministic renderer/machine success from human-facing semantic adequacy, enabling analysis of render–constraint gaps, stale syntax, facet coverage, evidence compliance, and semantic audit outcomes.

3 Problem Formulation

Let $\mathcal{M}_\theta$ be an LLM with frozen parameters θ, and let $\mathcal{D}_t$ denote the official syntax specification of a DSL at version t. A stale knowledge conflict arises when the model's prior over syntax patterns is concentrated on an outdated specification $\mathcal{D}_{t'}$ with $t' < t$, while the inference-time context provides evidence for the current specification $\mathcal{D}_t$ (Longpre et al., 2021; Wu et al., 2024). Given a user prompt u and retrieved context c_t, the model emits a candidate program $s \sim P_\theta(s \mid u, c_t)$. The candidate is executable only if $s \in \mathcal{V}_t$, where $\mathcal{V}_t$ is the set of programs accepted by the target renderer.

Let $\mathcal{C}_{\text{mach}}(u, t)$ denote deterministic, machine-checkable constraints such as required roots, axis declarations, dependency slots, nesting closure, configuration keys, or countable markers. Let $\mathcal{C}_{\text{sem}}(u)$ denote human-facing semantic intent constraints such as relation direction, grouping intent, or high-level topology. We define the render–constraint gap as:

$$s \in \mathcal{V}_t \quad \text{and} \quad s \not\models \mathcal{C}_{\text{mach}}(u, t). \tag{1}$$

Semantic adequacy is stricter:

$$s \in \mathcal{V}_t \wedge s \models \mathcal{C}_{\text{mach}}(u, t) \wedge s \models \mathcal{C}_{\text{sem}}(u). \tag{2}$$

In practice, $\mathcal{C}_{\text{mach}}$ is evaluated deterministically over the full benchmark, while $\mathcal{C}_{\text{sem}}$ is evaluated separately through semantic checkpoints and audited human annotation. Stale conflict is one source of non-executability or constraint failure: $s \notin \mathcal{V}_t$ and $\exists p \in \mathcal{P}_{old} : p \subset s$, where $\mathcal{P}_{old}$ contains deprecated, forbidden, or version-incompatible patterns.

Table 1: DIAGRAM-FAIL-BENCH composition. Each diagram type follows the same difficulty schedule.

Split	Easy	Med.	Hard	Ext.	Total
Per type	5	9	9	7	30
All 21 types	105	189	189	147	630

This separates four failure sources: stale syntax, retrieval miss, evidence non-compliance, and repair drift, which the experiments measure separately.

4 DIAGRAM-FAIL-BENCH: A Diagnostic Benchmark

DIAGRAM-FAIL-BENCH evaluates documentation-aware evolving DSL generation rather than general diagram generation alone. Each prediction is rendered in a deterministic headless Mermaid environment using Mermaid CLI (Mermaid Contributors, 2026a) and evaluated with machine-checkable constraints plus human-facing semantic checkpoints.

Coverage and schema. The benchmark contains 630 tasks across 21 Mermaid.js diagram types, with 30 tasks per type. Each type follows a fixed difficulty allocation: 5 easy, 9 medium, 9 hard, and 7 extreme tasks (Table 1). The types cover both long-standing diagrams (Flowchart, Sequence, Class, State, ER, Gantt) and newer or more version-sensitive families (Architecture, Block, Packet, Kanban, Treemap, Venn, XY Chart, Sankey). Each item contains a prompt and diagnostic fields: `diagram_type`, `difficulty`, `conflict_target`, `forbidden_legacy_patterns`, `required_facets`, machine checks, and semantic checkpoints. Easy tasks target lexical conflicts, medium tasks target local structural constraints, hard tasks introduce cross-formalism interference, and extreme tasks test renderable but semantically incomplete outputs.

Construction and validation. Conflict targets are derived from official documentation, release notes, changelog entries, migration notes, and version-diff artifacts (Mermaid, 2026; Mermaid Contributors, 2026c). We script-extract candidate rules, examples, deprecated forms, and version markers, then manually convert them into natural-language prompts, reference Mermaid solutions, required facets, forbidden patterns, deterministic checks, and semantic checkpoints. Items are reviewed for answerability under the target Mermaid version, diagram-type correctness, difficulty alignment, checker polarity, and whether forbidden patterns reject reference solutions. Reference solutions are used for construction and audit only; they are not supplied to generation, retrieval, or repair.

All hard/extreme reference solutions are executed in the same sandbox used for model outputs; experiments use `mmdc` v11.12.0 with a 30-second timeout. A reference-backed item is retained only if its reference renders and satisfies all deterministic machine checks. The released set contains 336 hard/extreme reference solutions; all 210 unique reference programs render and pass their checks under the target sandbox. This prevents "dead" tasks that ask for unsupported functionality.

Semantic audit and controls. Machine checks cover formalizable requirements such as root declarations, forbidden legacy tokens, required fields, and countable structural patterns. Semantic checkpoints are human-facing rubrics for outputs that render and may pass syntax but still lose task intent. We audit semantic adequacy with 210 stratified outputs over diagram type and difficulty. Two Mermaid-aware annotators review each case using the prompt, code, rendering, reference when available, and semantic checkpoints; disagreements are adjudicated and Cohen's κ is reported. We also include a 30-task non-conflict sanity split built from stable Mermaid syntax, and a 120-task PlantUML transfer set covering sequence, class, component, activity, state, use-case, object, and deployment diagrams. These controls test whether improvements reflect conflict resolution rather than longer prompts, more repair iterations, or Mermaid-specific artifacts.

5 RA-RAG: Render-Aware Mitigation

5.1 Overview

Figure 1 summarizes the online inference flow of RA-RAG, while Figure 2 describes the offline construction of its versioned rule inventory. RA-RAG is a closed-loop diagnostic intervention with three stages: task-aware planning, render-in-the-loop validation, and diagnosis-guided recovery. We do not claim novelty from simply chaining retrieval and repair; instead, the controlled variables are (i) whether retrieval covers required syntax facets, (ii) whether the model is anchored by an explicit syntax/facet contract, and (iii) whether failures are repaired from structured traces linked back to versioned evidence.

RA-RAG first parses the prompt into diagram-type candidates and required syntax facets, re-

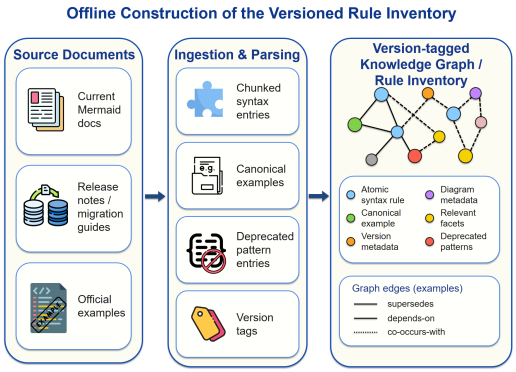

Figure 2: Offline construction of the versioned rule inventory. Current Mermaid documentation, release notes, migration guides, and official examples are parsed into syntax entries, canonical examples, deprecated-pattern records, and version tags.

trieves evidence through diagram-, root-, facet-, and error-aware routes, and selects evidence until required facets are covered. It then builds a compact syntax contract and generates code. A candidate is accepted only if it renders and passes deterministic machine checks; otherwise, renderer or checker feedback is converted into a structured trace, mapped to a likely violated constraint, and used for repair-specific retrieval and localized revision. This design compares relevance-based retrieval, raw-feedback retry, contract anchoring, and repair-specific evidence within one executable loop.

5.2 Version-Tagged Rule Inventory

Figure 2 illustrates the offline construction of the versioned rule inventory. We implement this inventory as a version-tagged knowledge graph $\mathcal{G}$ constructed from Mermaid documentation, release notes, changelog entries, and migration guides (Mermaid, 2026; Mermaid Contributors, 2026c). Each node stores an atomic syntax rule, a canonical example, version metadata, diagram metadata, relevant facets, deprecated patterns, and severity labels. Edges encode *supersedes*, *depends-on*, and *co-occurs-with* relations. The inventory supports both initial retrieval and repair-time retrieval. In our formulation, it is not merely a source of current documentation; it is a rule inventory used to test whether failures come from missing facets or from failure to comply with available evidence.

5.3 Task-Aware Retrieval and Syntax Contracts

Instead of querying with the raw prompt alone, RA-RAG first predicts diagram-type candidates $\hat{y}$ and syntax facets $\mathbf{f}$ such as `root`, `dependency`, `milestone`, `service`, or `classDef`. It then performs broad, diagram-specific, root-rule, and facet-specific retrieval. Candidate nodes are reranked by semantic similarity, diagram-type match, facet hits, root bonuses, example bonuses, and deprecated-pattern penalties. Coverage-based selection prioritizes evidence that fills missing required facets.

From the selected evidence, the model constructs a syntax contract a recording the target diagram type, reference version, required root, required facets, deprecated patterns to avoid, and evidence-grounded syntax prototypes extracted from retrieved grammar fragments and examples. We treat the syntax contract not as a standalone novelty claim, but as an explicit intervention whose effect is measured through executable machine success, evidence compliance, and ablation experiments.

5.4 Structured Render Feedback and Repair Retrieval

Generated code is executed using a local headless Mermaid renderer (Mermaid Contributors, 2026a) and evaluated with deterministic machine checks. If rendering fails, RA-RAG converts raw renderer output into a structured trace containing error type, line/column, unexpected and expected tokens, faulty line, local context, diagram hint, and inferred error facets. If rendering succeeds but machine checks fail, the trace records missing required patterns, forbidden patterns, or structural constraints. A diagnosis module maps the trace to a likely violated constraint, confidence score, and minimal repair target. Repair retrieval then queries the rule inventory using the trace, diagnosis, syntax contract, and local code context, rather than the raw error string alone. We evaluate repair under a conservative acceptance policy because repair can improve constraints while increasing runtime or harming renderability.

6 Experimental Design and Results

The experimental section is organized as a diagnostic study rather than only a leaderboard: we quantify how much current documentation helps, how much gap remains after documentation-aware generation, and which mitigation mechanisms re-

duce that gap.

6.1 Setup and Metrics

For the primary full-benchmark results, we report GPT-5.1 and GPT-5.4 runs with temperature 0.0, and evaluate outputs with a fixed local `mmdc` v11.12.0 renderer. We additionally run Mimo v2.5 Pro as an open-model replication using the same benchmark, renderer, retrieval index, and evaluation scripts. The primary conditions are: **Direct**, **Old Docs**, **Current Docs**, **Standard RAG**, **Self-Refine**, **Self-Debug**, **Syntax Contract**, **Machine-check Repair**, and RA-RAG. All retrieval methods use the same corpus/index; Standard RAG and Syntax Contract share the initial evidence budget; repair methods share the renderer, checker, timeout, and repair budget. Direct, Old Docs, Current Docs, Standard RAG, and Syntax Contract use one generation attempt; Machine-check Repair and RA-RAG allow up to three render/checker-guided attempts; Self-Refine and Self-Debug allow up to five self-revision attempts. To avoid conflating retry budget with mechanism design, component ablations preserve the same maximum attempt budget whenever the removed component does not make the condition undefined. Appendix C summarizes the information available to each condition.

We report render success, executable machine success (render success plus deterministic machine checks), stale syntax rate (SSR), facet coverage (FCR), evidence compliance (ECR), render–semantic gap (RSG), average iterations, and runtime. Semantic adequacy is reported separately through the human-audited validation protocol rather than used as a primary full-benchmark metric.

Validity and reproducibility controls. Because the benchmark, rule inventory, and checkers are all derived from evolving DSL documentation, we explicitly separate construction artifacts from generation-time evidence. Reference solutions are used only for audit and never supplied to generation, retrieval, or repair; checkers are identical across methods and use only recorded deterministic constraints. We treat machine success as executable correctness rather than semantic correctness, and use semantic audit, non-conflict controls, and cross-DSL transfer as partial checks against checker-source bias. For auditability, the supplement includes prompt templates, retrieval parameters, checker templates, renderer logs, sampled semantic annotations, and per-diagram confidence intervals. These details are intended to support both the responsible NLP checklist and independent replication.

A 70-task pilot over newer Mermaid families, reported in Table 4, motivates the final framing: current documentation raises renderability, but many renderable outputs still fail deterministic checks; syntax/facet anchoring and repair reduce this gap while exposing cost–quality trade-offs.

The pilot also shows that Old Docs can underperform No Docs. We attribute this to retrieval-induced negative transfer: for recent or version-sensitive diagram families, old documentation often lacks the target specification and retrieves incompatible rules. Such context can override the model's conservative prior, whereas No Docs more often falls back to generic Mermaid syntax that is less faithful but renderable.

6.2 Main Benchmark Results

Table 2 reports deterministic primary-run results on the completed 630-task DIAGRAM-FAIL-BENCH run. Current Docs improves over Old Docs on executable machine success for both completed closed-model runs (37.62 vs. 29.84 for GPT-5.1; 49.21 vs. 44.44 for GPT-5.4) and for the open-

Table 2: Main deterministic results on the 630-task DIAGRAM-FAIL-BENCH run. R denotes render success; Exec. denotes executable machine success, requiring render success plus deterministic machine checks.

Method	GPT-5.1		GPT-5.4		Mimo v2.5 Pro	
	R ↑	Exec. ↑	R ↑	Exec. ↑	R ↑	Exec. ↑
Direct	73.17	28.57	85.08	45.87	52.54	22.54
Old Docs	75.24	29.84	86.51	44.44	56.03	26.35
Current Docs	79.52	37.62	89.21	49.21	63.97	34.44
Standard RAG	85.40	40.16	89.05	43.97	64.60	33.33
Self-Refine	85.40	31.90	95.56	49.52	88.25	40.63
Self-Debug	78.89	34.29	91.43	49.68	85.87	37.62
Syntax Contract	84.29	38.89	93.33	50.95	68.73	34.76
Machine-check Repair	85.40	77.62	89.05	85.24	83.97	45.08
RA-RAG	**93.02**	**83.65**	**99.21**	**93.65**	**91.75**	**54.92**

Table 3: Component ablations and oracle variants on the full benchmark using GPT-5.1. Δ reports percentage-point change in Exec. Machine relative to RA-RAG.

Setting	Exec. Machine	Δ Exec.	Mechanism tested
Standard RAG	40.16	-43.49	Retrieval without contract/repair
Machine-check Repair	77.62	-6.03	Repair without syntax/facet anchoring
RA-RAG	**83.65**	0.00	Complete pipeline
RA-RAG w/o syntax contract	44.29	-39.36	Syntax/facet anchoring
RA-RAG w/o coverage selection	41.11	-42.54	Facet-complete retrieval
RA-RAG w/o structured trace	42.86	-40.79	Error interpretation
RA-RAG w/o repair retrieval	41.43	-42.22	Repair-specific evidence
Oracle Retrieval	52.38	-31.27	Upper bound for retrieved evidence only
Oracle Diagnosis	**84.92**	$+1.27$	Upper bound for failure diagnosis

Table 4: Diagnostic pilot on 70 newer Mermaid tasks. Exec. Machine requires render success plus deterministic machine checks.

Condition	Render	Exec.	Gap	SSR
No Docs	65.71	14.29	51.43	4.29
Old Docs	58.57	17.14	41.43	5.71
Current Docs	87.14	41.43	45.71	2.86
Current Docs + forbidden	84.29	35.71	48.57	0.00
Machine-check Repair	82.86	42.86	40.00	0.00
Syntax Contract	80.00	51.43	28.57	0.00
Contract + Repair	81.43	51.43	30.00	0.00

model Mimo docs baseline (34.44 vs. 26.35), confirming that up-to-date documentation is useful. However, Current Docs remains far below repair-based methods, showing that document injection alone does not eliminate the render–constraint gap. Standard RAG improves renderability over Direct, but its executable machine success remains much lower than repair-based methods, suggesting that retrieved documentation often helps syntax while leaving structural constraints unsatisfied.

Machine-check Repair is the largest source of executable correctness, raising Exec. Machine from Standard RAG's 40.16 to 77.62 for GPT-5.1 and from 43.97 to 85.24 for GPT-5.4. RA-RAG achieves the best deterministic trade-off in both completed closed-model runs: compared with Machine-check Repair, it further improves Exec. Machine from 77.62 to 83.65 for GPT-5.1 and from 85.24 to 93.65 for GPT-5.4. On Mimo, the completed RA-RAG run raises render success from Standard RAG's 64.60 to 91.75 and Exec. Machine from 33.33 to 54.92, providing an open-model replication of the render-aware trend. This suggests that syntax/facet anchoring and repair-aware retrieval improve deterministic executability, while semantic adequacy still needs separate human validation.

The lower absolute Mimo score is informative rather than contradictory. Mimo RA-RAG attains high render success but much lower executable success, indicating that the remaining bottleneck is not basic syntax but exact constraint instantiation. Its frequent failures include root/diagram-family mismatches and missing structural markers. This supports our framing that documentation-aware retrieval improves renderability, but weaker instruction following and evidence compliance still limit executable correctness.

6.3 Diagnostic Decomposition

Table 3 reports component ablations and oracle variants. These are dependency tests for a coupled closed-loop system, not claims that each module contributes independently and additively. Removing syntax contracts, coverage-based selection, structured traces, or repair retrieval drops Exec. Machine by roughly 39–43 points, bringing performance close to Standard RAG. This pattern is expected because the removed components break different links in the same evidence–diagnosis–repair chain: without coverage selection, required facets are often absent; without structured traces, failures cannot be localized; without repair retrieval, the model repairs from stale context rather than targeted evidence. Oracle Retrieval improves over ordinary ablations but remains far below RA-RAG, indicating that stronger evidence alone is insufficient. Oracle Diagnosis slightly exceeds RA-RAG (84.92 vs. 83.65), suggesting that failure interpretation is the main remaining headroom.

Semantic, non-conflict, and transfer controls. Table 5 reports validation beyond the main executable metric. The non-conflict split separates ordinary stable Mermaid generation from the conflict-heavy benchmark: Direct and Current Docs already render all non-conflict prompts, while closed-loop RA-RAG preserves renderability and raises executable machine success to 100.00. The semantic audit compares an LLM adjudicator with independently audited human labels on 210 sampled outputs, yielding 82.86% agreement. The judge

Table 5: Semantic validation, non-conflict controls, and PlantUML transfer. For non-conflict rows, results are Render / Exec.

Evaluation	N	Result
LLM–human agreement	210	82.86% (κ=0.60)
Judge precision / recall / F1	210	94.07 / 81.94 / 87.59
Human semantic adequacy	210	73.81%
Non-conflict Direct	30	100.00 / 50.00
Non-conflict Current Docs	30	100.00 / 63.33
Non-conflict RA-RAG	30	100.00 / 100.00
PlantUML Direct	120	92.50 / 35.00
PlantUML RAG	120	99.17 / 22.50
PlantUML RA-RAG	120	100.00 / 100.00

remains high-precision but imperfect, so we report human semantic adequacy separately rather than merging it into executable machine success.

For cross-DSL transfer, PlantUML RAG improves renderability over Direct but has low executable machine success, again showing that document injection alone does not guarantee constraint satisfaction. PlantUML RA-RAG reaches 100.00 Exec. Machine on this 120-task transfer set. We interpret this as a failure-mode transfer check, not as proof of full DSL generality or semantic perfection, because the set is smaller than DIAGRAM-FAIL-BENCH and its deterministic checks cover only recorded executable constraints.

What remains unsolved after render-aware repair. Although RA-RAG substantially improves executable machine success, remaining failures show that rendering feedback alone is insufficient. We observe three recurring modes. First, some outputs render but instantiate only shallow structure, omitting a grouping, initialization block, or multi-hop relation; this motivates separating machine success from semantic adequacy. Second, repair can be locally correct but globally disruptive, introducing layout, naming, or dependency violations. Third, evidence availability does not imply compliance: models may copy examples partially, mix them with older syntax, or apply them to the wrong diagram family. These patterns suggest that the residual gap concerns grounding, constraint instantiation, and failure interpretation, not retrieval recall alone.

Why the gains are not only from extra attempts. Extra attempts alone do not explain the gains: Self-Refine and Self-Debug allow more revisions but remain far below Machine-check Repair and RA-RAG. Component ablations retain the repair loop yet drop close to Standard RAG, showing that structured diagnosis and versioned evidence, rather than retry count, drive the improvements.

Scope of the claims. We use Mermaid as a controlled executable testbed, not as evidence that RA-RAG is a universal diagram-generation solver. The non-conflict and PlantUML splits are validity checks rather than claims of full generality. Render and machine success are necessary for executable correctness, but do not replace human assessment of intended structure.

7 Discussion

Current Docs improves renderability but leaves much lower executable machine success, so the problem is not merely missing context. If task-aware retrieval has high facet coverage but low evidence compliance, the bottleneck is how models instantiate available evidence. If oracle retrieval or oracle diagnosis reduces the remaining gap, the bottleneck is coverage or failure interpretation. If RA-RAG improves executable machine success but increases runtime or harms renderability, then render-aware repair has a real trade-off that should be reported rather than hidden.

8 Conclusion

We studied documentation-aware generation for evolving DSLs. Current documentation reduces overt stale syntax and improves renderability, but does not eliminate the render–constraint gap: many outputs remain non-executable or violate deterministic checks, and semantic adequacy requires separate human validation. We introduced DIAGRAM-FAIL-BENCH, a diagnostic benchmark for Mermaid generation that records conflict targets, forbidden legacy syntax, required facets, machine checks, and semantic checkpoints. We further proposed RA-RAG, a render-aware mitigation framework, and used controlled conditions to separate retrieval misses, evidence non-compliance, conflict sensitivity, structural/semantic loss, and repair drift. A 120-task PlantUML transfer set extends the evaluation beyond Mermaid without changing the primary benchmark scope. The resulting framing shifts the contribution from a monolithic RAG pipeline to a diagnostic study of how LLMs use documentation for evolving formal specifications.

Limitations

This work focuses primarily on Mermaid.js, which provides a controlled executable testbed but does not prove generalization to all evolving DSLs. The

PlantUML transfer set is smaller than DIAGRAM-FAIL-BENCH and should be read as transfer evidence rather than as a full second benchmark. The benchmark is manually constructed and may reflect annotation bias. Moreover, the benchmark, rule inventory, and checkers all draw from official documentation, so semantic audit, non-conflict controls, and cross-DSL transfer partially mitigate but do not eliminate checker-source bias. Absolute executable rates may be sensitive to checker design; we therefore interpret machine success as a diagnostic executable metric rather than a complete measure of semantic correctness. Render success is necessary but not sufficient for semantic correctness, motivating machine checks and sampled human validation. Finally, RA-RAG requires a local renderer plus a version-tagged rule inventory and introduces runtime overhead; selective repair, early stopping, and cheaper retrieval policies remain deployment optimizations.

Ethical Considerations

This work uses public DSL documentation and synthetic/user-style diagram prompts rather than private user data. The main risks are indirect: over-reliance on automatically generated technical diagrams, benchmark overfitting to deterministic checkers, and misuse of repair loops to produce plausible but semantically incomplete artifacts. We mitigate these risks by separating render, machine-check, and human semantic evaluation, reporting limitations of the transfer set, and treating RA-RAG as a diagnostic intervention rather than a replacement for human review in safety-critical documentation workflows.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *Proceedings of the Twelfth International Conference on Learning Representations*.

Eunseong Choi, June Park, Hyeri Lee, and Jongwuk Lee. 2025. Conflict-aware soft prompting for retrieval-augmented generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 26981–26995, Suzhou, China. Association for Computational Linguistics.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. 2025. Generating structured outputs from language models: Benchmark and studies. *arXiv preprint arXiv:2501.10868*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *Proceedings of the Twelfth International Conference on Learning Representations*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Shayne Longpre, Kartik Perisetla, Anthony Chen, Nikhil Ramesh, Chris DuBois, and Sameer Singh. 2021. Entity-based knowledge conflicts in question answering. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7052–7063, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

Xuhua Ma, Richong Zhang, and Zhijie Nie. 2026. CoRect: Context-aware logit contrast for hidden state rectification to resolve knowledge conflicts. *arXiv preprint arXiv:2602.08221*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

Mermaid. 2026. Mermaid documentation. Accessed 2026-05-19.

Mermaid Contributors. 2026a. Mermaid CLI. Accessed 2026-05-19.

Mermaid Contributors. 2026b. Mermaid: Generation of diagrams like flowcharts or sequence diagrams from text in a similar manner as markdown. Accessed 2026-05-19.

Mermaid Contributors. 2026c. Mermaid releases. Accessed 2026-05-19.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Basel Shbita, Farhan Ahmed, and Chad DeLuca. 2025. MermaidSeqBench: An evaluation benchmark for LLM-to-mermaid sequence diagram generation. *arXiv preprint arXiv:2511.14967*.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Fei Wang, Xingchen Wan, Ruoxi Sun, Jiefeng Chen, and Sercan O. Arik. 2025a. Astute RAG: Overcoming imperfect retrieval augmentation and knowledge conflicts for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30553–30571, Vienna, Austria. Association for Computational Linguistics.

Han Wang, Archiki Prasad, Elias Stengel-Eskin, and Mohit Bansal. 2025b. AdaCAD: Adaptively decoding to balance conflicts between contextual and parametric knowledge. 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 11636–11652, Albuquerque, New Mexico. Association for Computational Linguistics.

Kevin Wu, Eric Wu, and James Zou. 2024. ClashEval: Quantifying the tug-of-war between an LLM's internal prior and external evidence. In *Advances in Neural Information Processing Systems*, volume 37.

Rongwu Xu, Zehan Qi, Zhijiang Guo, Cunxiang Wang, Hongru Wang, Yue Zhang, and Wei Xu. 2024. Knowledge conflicts for LLMs: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8541–8565, Miami, Florida, USA. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Xiaowei Yuan, Zhao Yang, Yequan Wang, Shengping Liu, Jun Zhao, and Kang Liu. 2024. Discerning and resolving knowledge conflicts through adaptive decoding with contextual information-entropy constraint. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3903–3922, Bangkok, Thailand. Association for Computational Linguistics.

Abhay Zala, Han Lin, Jaemin Cho, and Mohit Bansal. 2024. DiagrammerGPT: Generating open-domain, open-platform diagrams via LLM planning. In *First Conference on Language Modeling*.

A Trace-to-Diagnosis Mapping Examples

Table 6 shows representative rule templates used by the diagnosis module to convert heterogeneous Mermaid renderer/checker feedback into repair-retrieval queries. The module is rule-based and auditable: renderer traces provide line-local parser evidence, deterministic checks provide missing or forbidden constraint labels, and the syntax contract supplies the expected diagram type and required facets. This appendix supports auditability of the repair process; we do not treat it as a separate diagnosis-accuracy experiment. Diagnosis quality is evaluated indirectly through the structured-trace ablation and Oracle Diagnosis comparison in Table 3.

B Runtime Attribution

Table 7 provides an auxiliary runtime attribution on a 100-instance GPT-5.4 subset stratified by diagram type and difficulty. This analysis is intended to characterize cost and latency, not to replace the full-benchmark results in Table 2. The main cost of closed-loop methods is not deterministic checking alone: repair-capable methods increase wall-clock time because they issue additional LLM calls and repeat retrieval, rendering, and checking until an executable constraint state is reached or the repair budget is exhausted. On this subset, RA-RAG has nearly the same average runtime as Machine-check Repair (73.91s vs. 73.37s) while achieving substantially higher executable success (97.00 vs. 78.00).

Table 6: Representative trace-to-diagnosis mappings used for repair retrieval. "Signal" denotes normalized renderer or checker evidence; "diagnosis facet" is the constraint label used to retrieve repair-specific rules and examples.

Failure class	Normalized signal	Diagnosis facet	Repair-retrieval query	Confidence
Forbidden legacy root	First non-comment line matches `stateDiagram` while the contract requires `stateDiagram-v2`, or a checker fires `forbidden_legacy_root`.	`root.version`	Target diagram + required root + deprecated root replacement example.	High
Root or diagram-type mismatch	Parsed intent predicts architecture, packet, or XY chart, but the output root is `flowchart`, missing, or reported by the renderer as an unknown diagram type.	`root.diagram_type`	Target diagram root + minimal current-version skeleton.	High
Unexpected token near local syntax	Renderer gives a line/column and unexpected token; the faulty line contains a known cross-formalism marker such as an unsupported arrow, relation marker, or bracket form.	`local.syntax`	Diagram type + faulty token class + canonical transition/edge syntax.	Medium
Missing configuration block	Rendering succeeds or partially parses, but machine checks report a missing required frontmatter/configuration, axis declaration, or initialization field.	`config.required`	Diagram type + required config facet + placement example.	High
Missing required facet after render	Rendering succeeds, but deterministic checks report absent required facets such as dependency, service/port/group, class/style directive, or semantic marker count.	`facet.missing`	Diagram type + missing facet name + compatible insertion example.	High

Table 7: Runtime attribution on a 100-instance GPT-5.4 subset stratified by diagram type and difficulty. Avg/P50/P90 are wall-clock seconds. Calls denotes average LLM invocations per instance. R/C denotes rendering plus deterministic checking; Ret./Diag. denotes retrieval plus structured diagnosis.

Method	Exec.	Avg	P50	P90	Calls	LLM%	R/C%	Ret./Diag.%
Direct	47.00	21.71	11.02	34.25	1.00	89.3	10.7	0.0
Standard RAG	53.00	39.50	28.87	44.25	1.00	38.9	20.4	40.7
Machine-check Repair	78.00	73.37	39.72	200.98	1.38	27.7	34.5	37.8
RA-RAG	**97.00**	73.91	43.15	199.96	2.84	36.2	25.6	38.2

Although RA-RAG uses more LLM calls on average, its localized repair and early stopping partly offset the additional call count in wall-clock time. The high P90 values show that latency is dominated by a small number of difficult repair cases, supporting selective repair and early stopping as deployment-time optimizations.

C Baseline Information Access

Table 8 summarizes the information available to each primary condition. The goal is to make baseline comparisons auditable rather than to introduce an additional method. This matrix also clarifies that methods differ not only in retry budget, but also in whether they receive retrieval evidence, syntax contracts, raw renderer/checker feedback, structured traces, or repair-specific retrieval.

Table 8: Information access of primary conditions. “Raw feedback” denotes renderer or checker messages without structured diagnosis.

Method	Cur. docs	Retrieval	Contract	Render fb.	Check fb.	Structured trace	Repair retrieval
Direct	–	–	–	–	–	–	–
Old Docs	–	–	–	–	–	–	–
Current Docs	✓	–	–	–	–	–	–
Standard RAG	✓	✓	–	–	–	–	–
Self-Refine	✓	–	–	self	–	–	–
Self-Debug	✓	–	–	raw	raw	–	–
Syntax Contract	✓	✓	✓	–	–	–	–
Machine-check Repair	✓	✓	–	raw	raw	–	–
RA-RAG	✓	✓	✓	✓	✓	✓	✓