CeLoR: Inference-Time Verification and Repair for LLM-Generated Artifacts

Jaehyun Sim
simjay@mit.edu

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02115

Abstract

Large language models (LLMs) are increasingly used to generate and edit structured artifacts such as source code and configuration files, but real systems are governed by many local policies: internal registries, security baselines, and style guides. These policies often cannot be fully expressed in each prompt and sometimes cannot safely be sent to a hosted LLM at all. In practice, developers improvise "LLM + tools" workflows, repeatedly asking the model to fix artifacts using test and linter output, which can be token-hungry, non-deterministic, and hard to reuse. We explore an alternative based on counterexample-guided inductive synthesis (CEGIS). CeLoR (CEGIS-in-the-Loop Reasoning) is a client-side framework that treats an LLM as a template generator, runs a bounded CEGIS loop over the hole space, uses oracle failures as counterexamples to prune bad candidates, and stores successful repairs in a FixBank for reuse. We instantiate CeLoR for Kubernetes manifests, where LLM-generated YAML must satisfy cluster-specific policies, and evaluate it on 30 repair tasks (113 violations). CeLoR matches a pure-LLM re-prompt baseline in success rate (100%) while using 21% fewer model calls in cold-start mode and eliminating LLM calls entirely with over 30× speedups once the FixBank is populated.

1 Introduction

Large language models (LLMs) are increasingly used to generate structured artifacts such as source code, CI pipelines, and configuration files [1]. Developers routinely ask models to "write a deployment," or "fix this function," and often obtain artifacts that compile or apply. However, real systems are governed by many *local* rules: internal registries, environment naming schemes, security baselines, style guides, and test suites. These rules rarely fit into every prompt, and some cannot safely be sent to a hosted LLM at all.

In practice, this leads to an ad-hoc "LLM + tools" workflow: generate a candidate, run tests or linters and policy checkers, paste error messages back into the prompt, and repeat until the result seems acceptable. Recent work on self-debugging formalizes this pattern for code, where the model runs tests and then tries to repair its own output [2]. While effective in many settings, this approach is token-hungry (multiple rounds of prompting), non-deterministic (the same input can lead to different repairs), and does not reuse what is learned about recurring mistakes. It also does not fully address privacy, because high-level policy descriptions still need to be exposed to the model.

Program synthesis offers a different lens. In counterexample-guided inductive synthesis (CEGIS), a synthesizer searches over a constrained program space while a verifier checks candidates against a specification and returns counterexamples that refine the search. Sketch is a classic instance: the user

Preprint. Under review.

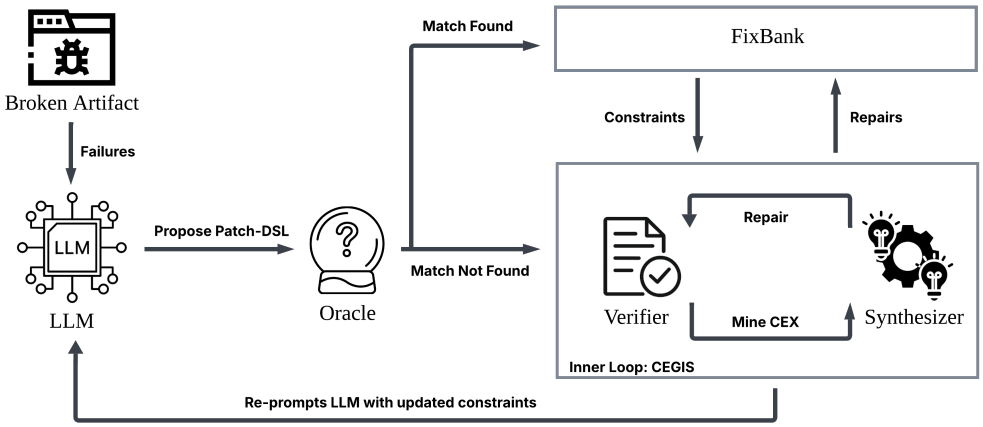

Figure 1: System Design of CEGIS-in-the-Loop Reasoning Framework (CeLoR)

writes a partial program with "holes," and a solver-backed engine fills those holes so that assertions hold [3]. Syntax-guided synthesis (SyGuS) makes the syntactic space explicit via grammars [4]. These systems cleanly separate (i) how programs are structured, (ii) how we search over parameters, and (iii) how counterexamples prune the search.

We explore how to bring this style of reasoning into LLM-assisted development without placing the model inside the refinement loop. We propose CeLoR (CEGIS-in-the-Loop Reasoning), a client-side framework that treats an LLM as a *template generator* and uses a small CEGIS loop with existing tools to repair the template's instantiations. Instead of asking the model to directly "fix" an artifact, CeLoR asks it to emit a program in a lightweight PatchDSL: a tiny domain-specific language describing how to transform an artifact, with parameters ("holes") left unspecified. External tools (tests, linters, policy checkers) act as oracles: given a candidate instantiation, they either pass or produce error information. CeLoR searches over a finite hole space, uses tool failures as counterexamples to rule out families of bad candidates, and records successful patterns in a *FixBank* so that recurring violations can be repaired without additional LLM calls.

We instantiate this pattern for Kubernetes manifests, a domain where LLMs already generate plausible YAML but where deployments must obey cluster- and organization-specific policies enforced by existing tools. Our aim is not to introduce a new synthesis algorithm or to compete with large-scale benchmarks, but to evaluate whether this "LLM-as-template, CEGIS-as-repair" pattern is practical and useful in realistic LLM+tools workflows.

2 Background and Related Work

CeLoR builds on ideas from program synthesis, configuration repair, automated program repair, and LLM-with-tools workflows.

Program synthesis and CEGIS In program synthesis, systems like Sketch allow users to write partial programs with unknowns ("holes"), and then automatically fill those holes so that the completed program satisfies a specification expressed as assertions [3]. Counterexample-guided inductive synthesis (CEGIS) frames this as an iterative dialogue between a synthesizer and a verifier: the synthesizer proposes a candidate, the verifier either accepts it or returns a counterexample, and the synthesizer refines its search space accordingly. Syntax-guided synthesis (SyGuS) makes the syntactic space explicit by defining a grammar for allowed programs and pairing it with a logical specification [4]. CeLoR adopts this pattern in a restricted, client-side setting: a PatchDSL defines a space of patch programs, tools act as oracles, and a small CEGIS loop searches over a finite hole space to find a satisfying instantiation.

Configuration repair and policy enforcement A line of work targets repair of configuration artifacts rather than general code. Tortoise, for example, repairs Puppet manifests when the resulting system configuration does not match the administrator's intent, using constraints and SMT solving to find edits that reconcile declarative configuration with observed state [5]. Other systems analyze how configuration changes affect execution and suggest repairs accordingly. CeLoR's Kubernetes instantiation is similar in spirit: manifests are the artifact, and tools encode the spec. However, CeLoR intervene with an LLM-generated PatchDSL layer and focuses on interactive, LLM-driven editing sessions with regressions, rather than one-shot repair of misconfigured systems.

Automated program repair and templates Automated program repair (APR) tools generate patches from test failures, contracts, or other specifications. Template-based APR methods define a library of edit patterns and instantiate them at suspicious locations, to achieve strong results on benchmarks such as Defects4J [6]. Recent work revisits template-based APR in the era of LLMs, where they used large models to fill in missing code within predefined patch templates [7]. Conceptually, CeLoR's PatchDSL and FixBank resemble a lightweight, synthesis-friendly version of template-based APR: templates describe structured edits, and local search selects instantiations. The difference is that CeLoR *learns* templates from an LLM on-demand and uses tool-defined oracles, rather than shipping with a fixed, hand-crafted template library and relying solely on tests.

LLMs with tests, tools, and self-debugging Recent work on self-debugging teaches LLMs to inspect execution results and revise their own code, by prompting the model with failing tests, error messages, or traces and asking for improved versions of the program [2]. Many empirical studies include a "re-prompt with errors" baseline, in which failed tests or tool diagnostics are appended to the prompt and the model is asked to fix the artifact. These pipelines can substantially improve success rates, but they keep the model inside the refinement loop: every counterexample triggers another model query, and the search process remains black-box. CeLoR takes a different approach. It uses the LLM once per violation pattern to generate a *parameterized* patch in a DSL, and then runs all subsequent search and verification locally. We trade some flexibility for three benefits: fewer LLM calls, deterministic behavior once the template and hole space are fixed, and an easy way to cache and reuse repairs through the FixBank.

Compared to configuration repair, CeLoR inserts an LLM-generated, syntax-guided patch layer in front of existing tools. Compared to template-based APR, it learns patch templates on demand instead of relying on a fixed library. Compared to self-debugging LLM pipelines, it moves the model to the boundary of the loop and uses a small CEGIS engine to handle refinement. The rest of the paper builds on this perspective, describing the CeLoR framework, its Kubernetes instantiation, and our initial evaluation.

3 Framework

We formalize artifact repair as a structured synthesis task over a constrained space of edits. CeLoR wraps a large language model (LLM) with an inference-time synthesis loop guided by counterexamples from local verification tools. Given an artifact $a_0 \in \mathcal{A}$ (e.g., a broken deployment YAML), CeLoR queries an LLM once to emit a patch program$p \in \mathcal{L}_{\text{PatchDSL}}$. This program encodes a fixed structure of transformation steps, each parameterized by symbolic placeholders or holes $h_1, \ldots, h_n$, which remain uninstantiated during generation.

Each hole h_i has an associated discrete domain D_i, manually defined or contextually inferred. The full hypothesis space of patch instantiations is the Cartesian product:

$$H = D_1 \times D_2 \times ... \times D_n$$

PatchDSL operations describe edits like setting field values, rewriting container image tags, or enforcing formatting. The DSL is kept minimal to ensure tractable and solvable synthesis, while expressive enough to capture common LLM mistakes.

To define correctness, CeLoR assumes access to a set of black-box tools $\mathcal{O} = \{O_1, \ldots, O_k\}$ that act as oracles. Each oracle evaluates an artifact and returns either $PASS$ or structured failure reports. The specification is implicitly defined by these tools:

$$\forall i \in [1, k], \quad O_i(a) = PASS$$

The core repair mechanism is a bounded CEGIS loop. CeLoR enumerates candidate assignments$\vec{h} \in H$, applies $p(\vec{h})$to a_0, and evaluates the result with all oracles. If any fail, error messages are used to extract constraints that prune future assignments (e.g., forbidding specific values or value combinations). The loop proceeds until a satisfying instantiation is found or the space is exhausted. Most importantly, this loop is deterministic, LLM-free, and guided entirely by observable tool feedback.

To enable reuse, CeLoR caches successful repairs in a FixBank, indexed by error signatures, which are canonicalized summaries of oracle failures. When a new artifact fails validation, CeLoR checks for a matching signature. If found, it reuses the corresponding patch template and hole domains, to skip the LLM usage and to enter the local synthesis loop directly.

This architecture supports several properties absent in prompt-based repair: token efficiency, reproducibility, and local policy encapsulation. Rather than rely on repeated model queries, CeLoR offloads refinement to symbolic search over a constrained program space.

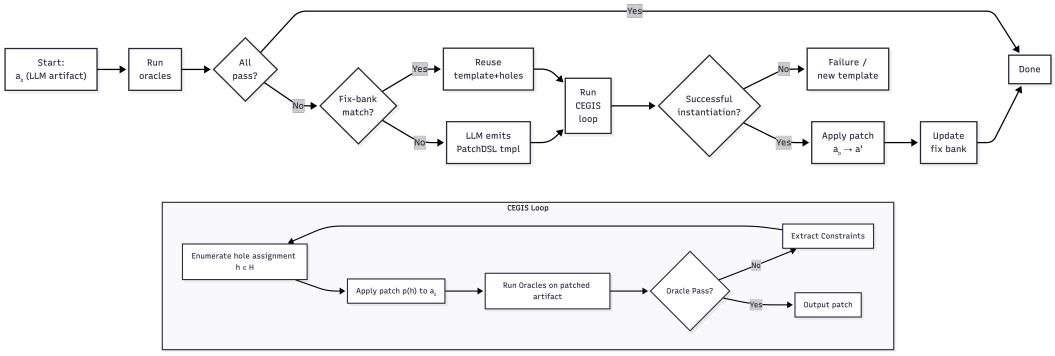

Figure 2: Flow chart of CEGIS-in-the-Loop Reasoning Framework (CeLoR)

4 Prototype: Kubernetes Manifest Repair

We instantiate CeLoR in the domain of Kubernetes manifests, where LLM-generated YAML often violates environment-specific policies. The artifacts are Deployment and Service specs evaluated by two local oracles:

1. a policy checker enforcing rules like private image registries and requiring production rules
2. a formatting oracle ensuring YAML file format complies with internal style guides

These policies are enforced entirely through local tools; none are visible to the LLM.

CeLoR uses a PatchDSL tailored to this domain, with operations like setting labels, rewriting image references, and formatting YAML. Patch templates emitted by the LLM contain symbolic holes (e.g., allowed environments, tags), and CeLoR locally enumerates valid hole assignments using CEGIS.

```
# Example Patch Templates with holes
EnsureEnvLabel(scope="deployment", value=?env)
EnsureImageRegistry(container="web", registry=?reg, tag=?tag)
NormalizeFormatting(mode=?fmt)
```

A small Python-based prototype for Kubernetes manifest repair implements patch interpretation, oracle execution, constraint-based pruning, and a FixBank to cache successful repairs keyed by error signatures.

We evaluate the system on a two-step scenario. In Round 1, an LLM-generated deployment violates security policies. CeLoR queries the LLM for a patch template and repairs the manifest via local

synthesis. In Round 2, the same violation recurs in a sidecar container. This time, CeLoR reuses the previous template from the Fix Bank and repairs the issue without querying the LLM again.

This prototype demonstrates how CeLoR achieves efficient and reusable policy repair by separating structure generation (via LLM) from correctness enforcement (via tools), all while keeping sensitive rules local.

5 Experiments and Results

We evaluate CeLoR on a controlled set of Kubernetes manifest repair tasks to answer three questions: (i) can CeLoR reliably repair LLM-generated artifacts under tool-defined policies; (ii) how do its LLM calls and latency compare to a direct re-prompting baseline; and (iii) how much benefit the FixBank provides for recurring violation patterns.

Setup We construct 30 repair tasks from the prototype domain in Kubernetes, where an LLM is asked to generate or modify Deployment manifests that intentionally violate registry, tag, or labeling policies and/or formatting rules. Each task contains one or more violations (113 in total across 30 cases), and is passed to one of three configurations:

- **CeLoR Cold Start**: CeLoR with an empty FixBank. For each task, we call the LLM once to generate a PatchDSL template, then run the local CEGIS loop over the hole space.
- **CeLoR Warm Start**: CeLoR after the FixBank has been populated from the same 30 tasks in cold-start mode. For each task, CeLoR first attempts to match the error signature against the FixBank and, if successful, skips the LLM and runs only the local synthesis loop.
- **Pure-LLM Baseline**: a self-debugging-style re-prompt baseline [2] that directly asks the model to output a fixed manifest. When tools report failures, we append the error messages to the prompt and re-prompt, up to a fixed retry budget.

We use GPT-4.1-mini as the LLM, a maximum of 10 candidates in the CEGIS loop, and a 10 s timeout per case. For each method we measure: (i) success rate (all oracles pass); (ii) total and average number of LLM calls; (iii) wall-clock time per case (including tool invocations); and (iv) number of candidate instantiations tried in CeLoR.

Success rate All three approaches successfully repaired all 30 cases (113/113 violations fixed), yielding a 100% success rate for each configuration (Table 1). This confirms that both CeLoR and the pure-LLM baseline are sufficiently expressive for the policies and PatchDSL used in our prototype.

Table 1: Success rates over 30 Kubernetes manifest repair tasks.

Approach	Successful cases	Total cases	Success rate
CeLoR Cold Start	30	30	100%
CeLoR Warm Start	30	30	100%
Pure-LLM Baseline	30	30	100%

LLM usage Table 2 summarizes LLM call counts. CeLoR Cold Start uses exactly one LLM call per case (30 in total), because the model is only used to generate a PatchDSL template. CeLoR Warm Start uses no LLM calls: once the FixBank is populated, all 30 cases can be repaired by reusing cached templates. In contrast, the pure-LLM baseline makes 38 calls (1.27 per case on average), because 8 of the 30 cases (26.7%) require two iterations to satisfy all tools.

Table 2: LLM call efficiency over 30 tasks.

Approach	Total LLM calls	Avg per case	Cases with > 1 call
CeLoR Cold Start	30	1.00	0
CeLoR Warm Start	0	0.00	0
Pure-LLM Baseline	38	1.27	8

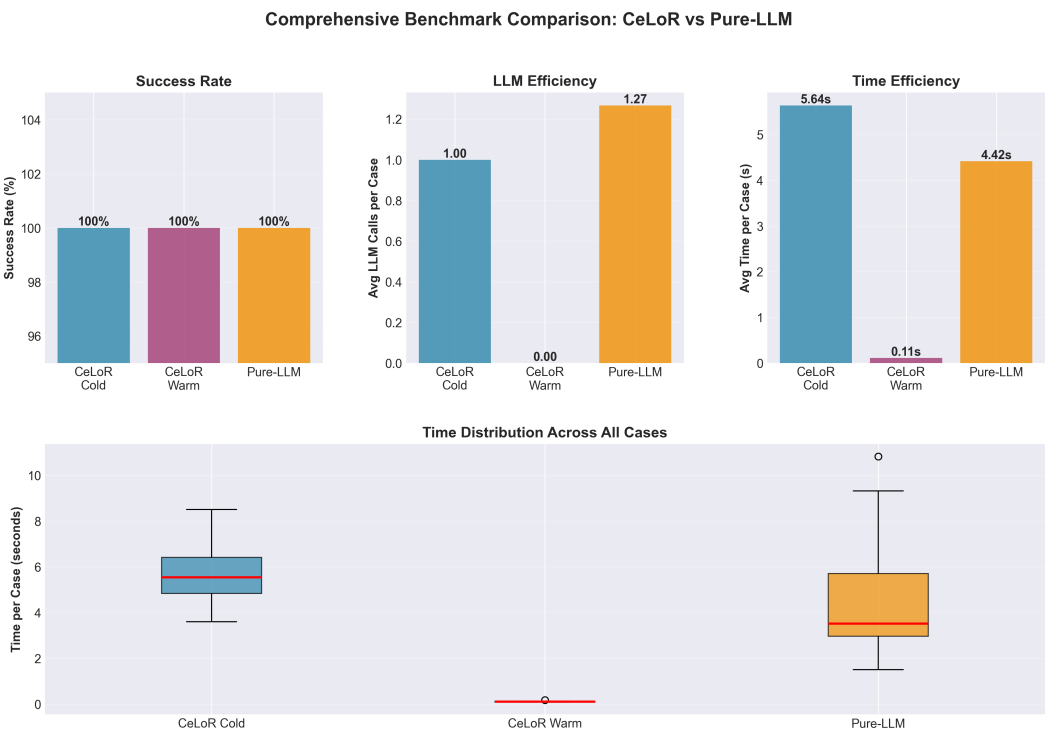

Figure 3: Comprehensive benchmark plots for CeLoR and Pure-LLM. **Top left**: Bar chart of success rate for CeLoR Cold, CeLoR Warm, and Pure-LLM. **Top middle**: Bar chart of LLM efficiency, showing average number of LLM calls per case for each method. **Top right**: Bar chart of time efficiency, showing average wall-clock time per case for each method. **Bottom**: Boxplots showing the distribution of time per case across all 30 tasks for CeLoR Cold, CeLoR Warm, and Pure-LLM.

CeLoR Cold Start thus uses 21% fewer model calls than the pure-LLM baseline (30 vs. 38), while CeLoR Warm Start eliminates them entirely on this benchmark. The latter illustrates the potential benefit of CeLoR's FixBank: once a violation pattern has been seen, later instances can be repaired without further LLM interaction.

Latency Table 3 reports wall-clock time. Pure-LLM is 26% faster than CeLoR Cold Start on average (4.0 s vs. 5.4 s per case), because it does not pay the cost of local synthesis. In Essence, Pure-LLM trades extra model calls for lower client-side computation. However, CeLoR Warm Start is dramatically faster than both: with only FixBank lookup and a single candidate per case, it repairs all 30 tasks in 3.2 s total (0.11 s per case), yielding a 50.6× speedup over CeLoR Cold Start and a 36.4× speedup over the pure-LLM baseline.

Table 3: Latency over 30 tasks (including oracle execution).

Approach	Total time (s)	Avg per case (s)	Speedup vs. Cold
CeLoR Cold Start	162.0	5.40	1.0×
CeLoR Warm Start	3.2	0.11	50.6×
Pure-LLM Baseline	120.0	4.00	0.74×

Synthesis cost and reuse In CeLoR, synthesis itself is cheap: Cold Start tries 30 candidates in total (1 per case on average), with only a single harder case requiring multiple candidates and learning three constraints that are then reused in Warm Start. Warm Start replays the same successful instantiations with zero constraint learning. In our setting, oracle runtime dominates total cost; simple constraint-based pruning is enough to avoid redundant oracle calls.

Summary All three methods solve all tasks, but they differ in how they trade LLM usage for local computation and reuse. For one-off repairs, the pure-LLM baseline is slightly faster in our prototype, at the cost of additional model calls and non-deterministic behavior across retries. CeLoR Cold Start is modestly slower but uses fewer LLM calls and has an explicit, deterministic search space once a template is fixed. When violations recur, CeLoR Warm Start shows the main advantage of the architecture: by reusing FixBank templates, it can repair all 30 tasks with zero LLM calls and two orders of magnitude lower latency. These results are limited to a small benchmark and a single domain, but they support the claim that a CEGIS-style controller can make LLM+tool workflows more token-efficient, deterministic, and amortizable over repeated patterns.

6 Discussion

CeLoR takes a different path from the usual "re-prompt until tests pass" pattern. Instead of keeping the LLM inside the loop, it uses the model once to propose a structured patch in a PatchDSL, then relies on a small CEGIS-style search and existing tools to decide which instantiations are acceptable. This makes the roles of the components explicit: the LLM provides a prior over patch structure, tools define the specification, and local search explores a finite parameter space.

Treating the LLM as a template generator has two practical consequences. First, model usage is bounded and predictable: in our experiments, CeLoR Cold Start uses exactly one LLM call per task (30 total), and once the FixBank is populated, CeLoR Warm Start repairs all 30 tasks without any LLM calls, whereas a pure re-prompt baseline needs 38 calls over the same tasks. Second, once a template and hole space are fixed, behavior is deterministic: the same artifact and oracle outputs lead to the same patch, in contrast to re-prompting pipelines whose behavior can vary across retries.

The PatchDSL and hole space capture a bias-capacity tradeoff. A small DSL with tiny domains keeps synthesis trivial and encourages reuse via the FixBank, but it cannot express arbitrary edits. In contrast, richer DSLs would increase coverage but require stronger search. The FixBank, although simple, already shows how recurring violation patterns can be turned into local "repair macros": after a cold-start phase, warm-start repair is both LLM-free and much faster (over $30\times$ in our prototype). We see this combination as the main conceptual takeaway of CeLoR: LLM-generated templates, tool-defined oracles, local synthesis, and cached patterns.

7 Limitations and Future Work

Our prototype makes several simplifying assumptions. The PatchDSL and hole spaces are hand-designed and very small, which keeps the CEGIS loop cheap (on average a single candidate per case in our experiments) but restricts what kinds of repairs can be expressed without asking the LLM for a new template. Scaling to richer domains such as non-trivial code refactorings or CI pipelines will require more expressive operations and larger, possibly adaptive, hole domains.

CeLoR also relies on manually chosen oracles and simple constraint extraction. We assume someone has already decided which tools matter and how they are configured, and we mostly turn specific error codes into "do not use this value again" constraints. This is sufficient for the discrete policies in our Kubernetes setting, but it does not exploit the full structure of tool feedback. Learning richer constraints or ranking candidates by error information is a natural next step.

The FixBank is currently a best-case cache: we evaluate it on the same 30 tasks that populate it, so hit rates are 100%. Real deployments will see evolving policies, changing error messages, and only partially overlapping patterns, which will require more robust matching and mechanisms to retire or refine stale entries. Finally, our evaluation is narrow: a single domain, 30 tasks, one model configuration, and a basic re-prompt baseline. A broader study across artifact types, PatchDSLs, and LLMs is needed to understand where CeLoR provides the most benefit.

We view CeLoR primarily as a design pattern rather than a finished system: LLM-generated templates, tool-defined specs, local CEGIS-style search, and cached repairs. Future work includes instantiating this pattern for source code with AST-level PatchDSLs and oracles such as `black`, `ruff`, `mypy`, and `pytest`, and plugging stronger synthesis backends in behind the same interface to handle richer templates without changing the overall architecture.

8 Conclusion

We asked whether ideas from counterexample-guided synthesis can help structure "LLM + tools" workflows so they are more predictable, efficient, and reusable. CeLoR is a first step toward an affirmative answer. Rather than keeping the LLM inside a trial-and-error repair loop, CeLoR uses the model once to emit a structured patch in a small PatchDSL and then performs all refinement locally via a CEGIS-style search guided by existing tools. Tools provide the specification, the PatchDSL and hole space expose an explicit search region, and a FixBank caches successful patterns for reuse.

Our Kubernetes prototype shows that this architecture is practical in a realistic, policy-rich setting. On 30 manifest repair tasks (113 violations), CeLoR matches a pure re-prompt baseline in success rate while using fewer model calls in cold-start mode and eliminating them entirely once the FixBank is populated, with over 30$\times$ speedups in warm-start repair. These are small-scale results in a single domain, but they illustrate the main benefits of separating generation from refinement and treating the LLM as a template generator rather than the primary search engine.

We view CeLoR less as a finished system and more as a design pattern: LLM-generated templates, tool-defined specs, local CEGIS-style search, and cached repairs. Extending this pattern to richer PatchDSLs, stronger synthesis backends, and other artifact types such as source code, CI pipelines, or data configurations is promising future work. The core insight, however, remains simple: we can often get more reliable and reusable behavior not by calling the model more, but by combining a single model-generated hypothesis with principled local search and verification.

9 Data and Reproducibility

Entire code, prototype, data, results, and instructions can be found in the git repository at `https://github.com/simjay/celor`. The benchmark entrypoints and results are available in the project repository at `https://github.com/simjay/celor/blob/main/benchmark`.

References

[1] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 09 2024.

[2] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv (Cornell University)*, 04 2023.

[3] Rastislav Bodík and Armando Solar-Lezama. Program synthesis by sketching. 01 2008.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. 10 2013.

[5] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. pages 625–636, 10 2017.

[6] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: revisiting template-based automated program repair. pages 31–42, 07 2019.

[7] Yuwei Zhang, Zhi Jin, Ying Xing, Ge Li, Fang Liu, Jiaxin Zhu, Wensheng Dou, and Jun Wei. Patch: Empowering large language model with programmer-intent guidance and collaborative-behavior simulation for automatic bug fixing. *arXiv (Cornell University)*, 01 2025.