# Toward Executable Multi-framework Front-end Code Generation with Self-Correction

**Linxiao Li**[1*] **Haoran Ma**[2*] **Chenyue Wang**[1]
**Jiaye Lin**[3] **Haochen Sui**[4] **Jiechao Gao**[5†]
[1]The University of Sydney [2]Harvard University
[3]Meituan [4]University of Michigan [5]Stanford University

## Abstract

Generating codes from UI screenshots has recently benefited from MLLMs, yet existing methods centered on HTML targets and transfer poorly to multi-frameworks such as React, Vue, and Angular. However, unlike HTML, which does not entail compilation issues, generating executable code in multi-framework settings is more challenging due to framework-specific differences in syntax and failure modes, as well as the need to maintain global consistency across files. To address this problem, we study executable multi-framework front-end code generation and propose **MESCoder**, a tree-structured self-corrective generation framework. Our method first extracts a structural scaffold from the screenshot and then constructs sub-trees and assembly relations step by step, transforming large project generation into a sequence of localized decisions. On top of this scaffold, we introduce a unified Project Agent that continuously revises its own partial project under runtime sandbox feedback and dynamically chooses between local repair and cluster level repair. For training, we adopt a two-stage strategy that first initializes the policy offline to learn basic multi-framework generation and repair priors, and then refines it with online reinforcement learning to optimize long horizon executability and final page quality. The resulting framework models multi-framework executability as a conditioned project state transition process and improves the stability and quality of complex front end generation through self correction. Experiments show that our method improves executability, while narrowing the visual-fidelity gap to HTML-based generation. Codes are available at: Link.

## 1 Introduction

Recent advances in MLLMs have substantially improved the ability to generate front-end code directly from UI screenshots [27, 16, 34, 31]. This task requires models to jointly perform visual understanding, structural abstraction, and front-end implementation. As a result, generation quality depends not only on recognizing local UI elements, but also on whether the model can transform hierarchical layout and spatial organization into a coherent implementation [30, 20].

Existing work has made clear progress on this problem, yet the dominant formulation remains largely HTML-centered. Recent efforts have scaled HTML-screenshot training corpora [16, 34, 9] and established benchmarks for measuring how faithfully a generated page reproduces its reference [27]. On the method side, divide-and-conquer decomposition [29], layout-guided prompting [30], and structure-aware fine-tuning [20] further improve element recall and spatial fidelity. Common to all of these efforts, however, is a shared assumption: the generation target is a single self-contained HTML file that renders without compilation and without cross-file reasoning—an assumption that

*Equal contribution.
†Corresponding author: jiechao@stanford.edu.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

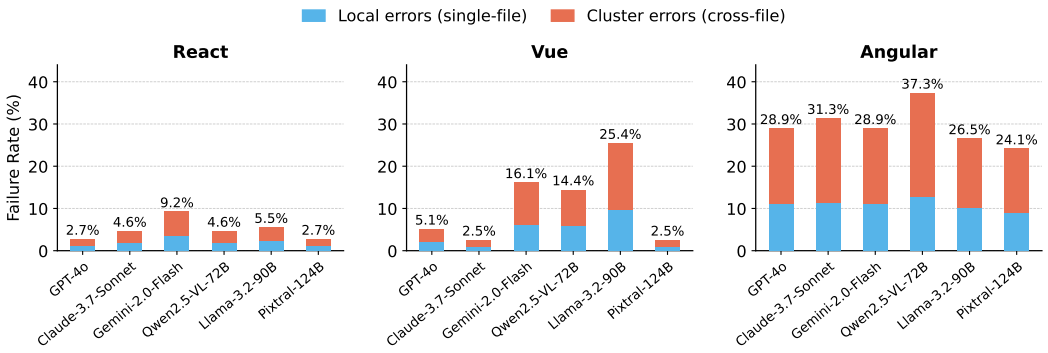



Figure 1: Compilation failure rates of six state-of-the-art MLLMs under one-shot generation on React, Vue, and Angular. Failures are decomposed into **local errors** and **cluster errors**.

does not hold in real front-end development, where production interfaces are assembled from multiple files within framework-specific project structures and become visible only after the entire project successfully compiles and runs.

However, extending screenshot-to-code generation from HTML to multi-frameworks such as React, Vue, and Angular is not merely a matter of changing prompts or rewriting syntax. Unlike HTML, multi-framework front-end generation requires models to produce an executable project within the framework-specific program space, where correctness depends not only on surface syntax but also on component boundaries, cross-file dependencies, parent-child assembly, and runtime consistency. Existing analysis shows that straightforward prompt transfer across frameworks causes a substantial drop in compilation success and is accompanied by highly framework-specific error modes [33]. In stricter frameworks, failures are often no longer limited to local token- or template-level mistakes, but instead emerge as engineering-level inconsistencies involving import/export relations, bindings, assembly, and project-level coherence. Meanwhile, one-shot screenshot-to-project generation entangles local implementation, global composition, and framework constraints in a single decoding process, making intermediate errors difficult to localize and even harder to correct reliably.

To address these challenges, we study executable multi-framework front-end code generation and propose MESCoder, a tree-structured self-corrective generation framework. Our key idea is to first extract a structural scaffold from the input screenshot, converting large project generation into a sequence of localized subtree expansion and assembly decisions, and then introduce a unified Project Agent that continuously revises its own partial project under runtime sandbox feedback. Rather than treating correction as a post-hoc repair stage, MESCoder integrates self-correction into the generation loop itself, enabling the model to dynamically decide whether to continue construction, perform local repair, or escalate to cluster-level repair when local edits become insufficient. To stably learn this behavior, we further adopt a two-stage training strategy: we first initialize the policy offline to acquire basic multi-framework generation and repair priors, and then refine it with online reinforcement learning to optimize long-horizon executability and final page quality. In this way, we formulate multi-framework front-end generation as a framework-conditioned project state transition process, with self-correction serving as the key mechanism for improving cross-framework executability and the stability of complex front-end generation.

Our contributions are summarized as follows:

- We propose MESCoder, a tree-structured self-corrective generation framework that decomposes project construction into localized subtree and assembly decisions and equips a unified Project Agent with runtime-feedback-driven self-correction.
- We develop a two-stage training strategy that combines offline policy initialization with online reinforcement learning, enabling to optimize long-horizon executability and generation quality.
- Extensive experiments demonstrate that MESCoder improves code executability across multiple frameworks while narrowing visual similarity comparable to HTML-based generation.

## 2 Related Work

### 2.1 WebUI Code Generation

Generating front-end code from visual inputs has been studied extensively. Early methods used classical vision pipelines to reverse-engineer UI elements [23, 22], while large-scale datasets such as Rico [6] later enabled data-driven approaches. Beltramelli [1] proposed pix2code, a CNN-LSTM model that produces code tokens from GUI screenshots, and Chen et al. [2] concurrently introduced a neural machine translator for GUI skeletons. The emergence of MLLMs has shifted this line of work toward fine-tuning on screenshot-code pairs at scale. WebSight [16] provides two million synthetic HTML–screenshot pairs, with parallel efforts such as Sketch2Code [19] extending inputs to hand-drawn sketches. Design2Code [27] introduces the first real-world benchmark of 484 webpages with fine-grained visual and structural metrics, complemented by IW-Bench [11] with DOM-based element and layout accuracy. Follow-up work scales training data to over one [34] and 2.56 million [9] instances, introducing TreeBLEU for structural recall, while Interaction2Code [32] extends evaluation to dynamic interactions and DesignBench [33] to React, Vue, and Angular. On the method side, recent approaches improve generation through divide-and-conquer decomposition [29], hierarchical generation [10], modular multi-agent pipelines [14], layout-guided prompting [30], and structure-aware fine-tuning [20]. Closer to our work, MRWeb [28] extends generation beyond single pages via explicit resource lists, and DesignCoder [4] couples hierarchical generation with self-correction. Despite this progress, existing work remains *HTML-centered*, targeting self-contained pages that render without compilation. Multi-framework generation for React, Vue, and Angular instead requires producing multi-file executable projects whose correctness depends on whether the project as a whole compiles and runs—a setting that prior methods do not address.

### 2.2 Self-correction in Code Generation

Self-correction has emerged as a central capability for code generation, allowing models to revise their own outputs based on feedback rather than committing to a single one-shot prediction [24]. Early prompting-based methods established the basic loop: Self-Refine [21] iteratively rewrites a program from natural-language critiques produced by the same model, Reflexion [26] adds an episodic memory of verbal reflections distilled from environment feedback, and Self-Debug [3] grounds the loop in execution traces by teaching LLMs rubber-duck-style debugging. However, intrinsic self-correction without external signals has been shown to plateau or even degrade reasoning quality [12], motivating a shift toward training-based correction with executable feedback. CodeRL [17] and StepCoder [7] optimize generation policies against unit-test or compiler rewards, while RLEF [8] reinforces multi-turn LLM agents directly with execution feedback inside the rollout. SCoRe [15] introduces a two-stage RL recipe to counter behavior collapse and distribution shift in self-correction, and $\mu$CODE [13] casts multi-turn code generation as a single-step decision problem with verifier-guided rewards. For small models, CoCoS [5] aligns reflective revision with parameter-efficient online RL. On the structural side, CodeTree [18] performs agent-guided tree search over generation and refinement, and AlphaCodium [25] reframes generation as a test-driven flow with iterative public-test refinement, while DesignCoder [4] couples hierarchical UI generation with a self-correction loop. Despite this progress, existing self-correction methods remain *single-file-centric*, treating each program as a self-contained unit revised in isolation. Multi-framework front-end projects, in contrast, demand correction over a graph of files whose compilability depends on cross-file imports, bindings, and component assembly—a setting where local edits often cannot restore project-level coherence and that prior methods do not address.

## 3 Methodology

### 3.1 Overview

Given a webpage screenshot $x$ and a target front-end framework $f \in \{\text{React}, \text{Vue}, \text{Angular}\}$, our goal is to generate a complete front-end project $P_T$ that is executable under $f$ and visually consistent with $x$. Fig 2 illustrates the learning process of MESCoder, which contains four components: A fixed scaffold pre-processor extracts a layout tree and a task queue from the screenshot. A policy module selects a high-level action according to the current project state. An LLM executor performs the selected action by generating or patching code. A runtime sandbox executes the intermediate project,

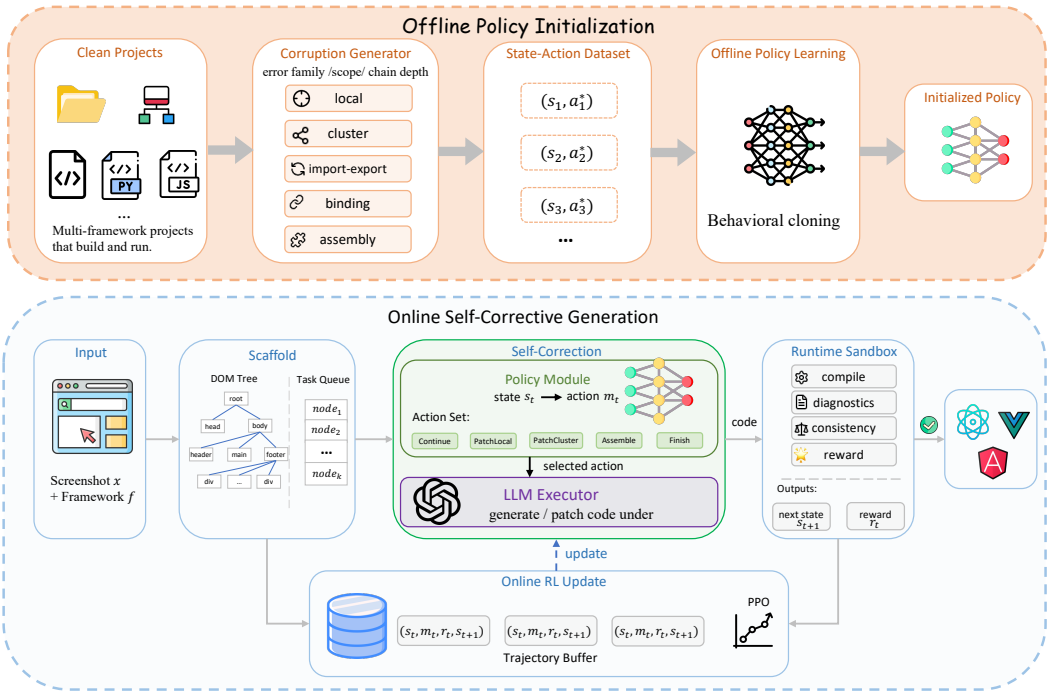



Figure 2: The two-stage learning process of MESCoder.

produces diagnostic feedback, and computes executability-oriented rewards. At step $t$, the interaction is written as

$$m_t \sim \pi_\phi(\cdot \mid s_t), \qquad \Delta_t = \mathrm{LLM}(s_t, m_t), \qquad (P_{t+1}, d_{t+1}, \ldots) = \mathcal{E}(P_t, \Delta_t, f), \tag{1}$$

where $s_t$ is the current state, $m_t$ is the high-level action, $\Delta_t$ is the concrete code edit, $d_{t+1}$ is the diagnostic summary, and $\mathcal{E}$ denotes the framework-aware execution environment.

The learning process has two stages. First, we initialize the policy offline using state-action data. Second, we refine the policy online with reinforcement learning under sandbox feedback. In the main setting, the LLM executor is kept fixed, and both offline initialization and online reinforcement learning optimize only the action policy and its value estimator.

### 3.2 Tree-Structured Self-Corrective Generation

Front-end pages are naturally hierarchical. A full page is composed of regions, containers, repeated blocks, and local components, and these units must eventually be assembled into a coherent project. Directly generating the entire project in one pass makes intermediate errors difficult to localize and repair. We therefore first extract a scaffold from the screenshot $\mathcal{S}(x) = \{T, U, Q\}$, where $T$ is a layout tree, $U$ is a content slot table, and $Q = \{\tau_1, \tau_2, \ldots, \tau_N\}$ is a task queue derived from the tree. The scaffold is fixed and not trained. It provides a structural interface that localizes generation and correction decisions.

At interaction step $t$, the current task pointer is $i_t$, and the active task is $q_t = \tau_{i_t} \in Q$. The agent observes

$$s_t = \{x, f, q_t, P_t, d_t, h_t, b_t\}, \tag{2}$$

where $P_t$ is the current partial project, $d_t$ is the diagnostic summary returned by the sandbox, $h_t$ stores recent action-feedback history, and $b_t$ is the remaining budget. The history term captures whether recent actions have improved the project, which is important for identifying repeated local failures and deciding whether broader correction is needed.

The policy selects one action from

$$m_t \in \{\textsc{Continue}, \textsc{PatchLocal}, \textsc{PatchCluster}, \textsc{Assemble}, \textsc{Finish}\}. \tag{3}$$

CONTINUE advances the current task when the partial project is sufficiently healthy. PATCHLOCAL invokes a local repair over a small scope. PATCHCLUSTER invokes a broader repair over a small

dependency or assembly cluster. ASSEMBLE integrates completed child units into the current parent node. FINISH terminates the episode when the project is complete and executable. This action set separates forward construction from self-correction. CONTINUE and ASSEMBLE construct the project, while PATCHLOCAL and PATCHCLUSTER correct intermediate failures. The policy therefore learns not only whether to repair, but also the appropriate scope of repair.

The policy is a lightweight neural controller:

$$\pi_\phi(m_t \mid s_t) = \mathrm{softmax}(g_\phi(z_t)), \tag{4}$$

where $z_t$ is a structured encoding of $s_t$ and $g_\phi$ is the actor network. After an action is selected, the LLM executor generates the concrete code edit:

$$\Delta_t = \mathrm{LLM}(s_t, m_t). \tag{5}$$

The executor is action-conditioned. Under PATCHLOCAL, it is instructed to modify only a local scope. Under PATCHCLUSTER, it may edit several related files or components. Thus, the LLM handles code realization, while the policy handles self-corrective control.

### 3.3 Runtime Sandbox and Executability Feedback

The runtime sandbox defines the execution environment. Given the current project $P_t$, an executor edit $\Delta_t$, the selected action $m_t$, and the target framework $f$, the sandbox applies the edit, runs framework-specific checks, and returns the updated project together with compact execution feedback:

$$(P_{t+1}, d_{t+1}, e_t, e_{t+1}, u_t) = \mathcal{E}(P_t, \Delta_t, f). \tag{6}$$

Here, $\mathcal{E}$ only applies and verifies the edit. The diagnostic summary $d_{t+1}$ is a compact representation of execution feedback. It records the dominant diagnostic family and the affected files or components, and is used as part of the next state rather than as a separate handcrafted reward term. The quantities $e_t$ and $e_{t+1}$ denote the numbers of sandbox error clusters before and after applying the edit, measured under the same action-relevant verification scope $\Omega_t = \Omega(s_t, m_t)$. The scope depends on the selected action: local repair is verified within the local edit scope, cluster repair within the related file cluster, assembly within the parent-child assembly scope, and FINISH on the full project. This avoids penalizing intermediate partial projects for missing future tasks that have not yet been generated. The binary signal $u_t \in \{0, 1\}$ indicates whether the current action successfully commits a new task. For example, $u_t = 1$ if a CONTINUE action completes the current subtree, or if an ASSEMBLE action successfully connects a completed child unit to its parent. Otherwise, $u_t = 0$. Since intermediate partial projects may not be fully renderable, visual quality is not used as a dense step-level signal. We compute a terminal quality score $Q_T$ only after a final renderable project is produced.

### 3.4 Two-Stage Policy Learning

**Offline policy initialization.** The offline stage initializes the action policy before online reinforcement learning. Its goal is not to train the LLM executor or to teach code generation. Instead, it provides a reasonable action prior for the policy module $\pi_\phi(m \mid s)$.

For each clean executable project $P^\star$, we construct normal project-construction states by replaying the tree-guided construction process. These states provide supervision for non-repair actions. If the current node is unfinished and no critical diagnostic is present, the label is CONTINUE. If a subtree is completed and should be integrated into its parent, the label is ASSEMBLE. If the root project is complete and strict checks pass, the label is FINISH. Thus, clean replay provides examples for the forward construction behavior. To initialize repair decisions, we generate corrupted states from clean projects. Given $P^\star$, an automatic corruption operator produces

$$\tilde{P} = \mathcal{C}(P^\star). \tag{7}$$

For a corrupted state $s_t$, we define a candidate repair set $\mathcal{M}(s_t)$, typically containing PATCHLOCAL and PATCHCLUSTER. For each candidate action $m \in \mathcal{M}(s_t)$, the LLM executor is called $K$ times to generate candidate edits:

$$\Delta_{m,k} = \mathrm{LLM}(s_t, m), \qquad k = 1, \ldots, K, \tag{8}$$

where $K$ is the number of executor samples used only for offline outcome estimation; it reduces labeling noise caused by stochastic LLM edits. Each candidate edit is evaluated by the sandbox. Let

$$\hat{R}(s_t, m) = \max_k R(s_t, m, \Delta_{m,k}) \tag{9}$$

be the best outcome score obtained under action $m$. The offline repair label is then

$$m_t^\star = \arg \max_{m \in \mathcal{M}(s_t)} \hat{R}(s_t, m). \tag{10}$$

This result-driven labeling procedure avoids reducing offline learning to a predefined error-action rule. It assigns action labels according to observed outcomes, so the same diagnostic family may still lead to different repair scopes under different contexts. The policy is initialized by minimizing

$$\mathcal{L}_{\text{off}} = - \sum_{(s_t, m_t^\star) \in \mathcal{D}_{\text{off}}} \log \pi_\phi(m_t^\star \mid s_t). \tag{11}$$

The resulting policy $\pi_{\phi_0}$ provides an action prior for subsequent online learning.

**Online reinforcement learning.** Offline initialization provides a policy prior, but executable project generation remains a long-horizon decision problem. During real generation, actions interact with LLM-generated edits, framework-specific constraints, and accumulated project states. Online reinforcement learning further optimizes the policy under actual sandbox feedback.

Starting from $\pi_{\phi_0}$, the agent interacts with the environment and collects trajectories $(s_t, m_t, r_t, s_{t+1})$. The online step reward is constructed from the sandbox feedback:

$$r_t^{\text{step}} = \alpha(e_t - e_{t+1}) + \beta u_t - \gamma c_t. \tag{12}$$

The first term rewards error reduction in the action-relevant verification scope. The second term rewards successful task commitment. The third term penalizes expensive or invalid actions. The cost $c_t$ depends on the selected action, edit size, verification scope size, and whether the edit can be validly applied. At the end of an episode, the agent receives

$$R_{\text{final}} = \begin{cases} r_{\text{pass}} + \mu Q_T, & \text{SUCCESS}, \\ r_{\text{partial}}, & m_T = \text{FINISH and ERROR}, \\ 0, & \text{otherwise}. \end{cases} \tag{13}$$

The complete optimization objective is

$$J(\phi) = \mathbb{E}_{\tau \sim \pi_\phi} \left[ \sum_{t=0}^{T-1} r_t^{\text{step}} + R_{\text{final}} \right]. \tag{14}$$

We optimize the policy using an actor-critic formulation. The actor is $\pi_\phi(m \mid s)$, while the critic estimates the state value $V_\psi(s)$. In our implementation, we use PPO-style clipped policy optimization for stability. The actor and critic parameters, $\phi$ and $\psi$, are updated during online learning, while the LLM executor remains fixed in the main setting. This keeps reinforcement learning focused on high-level self-corrective decision making rather than token-level code generation.

# 4 Experiment

## 4.1 Experimental Setup

**Benchmarks and Metrics.** We evaluate on **DesignBench** [33], the benchmark pairing real-world webpage screenshots with ground-truth executable multi-framework projects. Its *design generation* split contains 310 compilation-verified pages across React, Vue, and Angular. We stratify these by framework and complexity, holding out 200 pages as the test set and reserving the remaining 110 for training. We report CSR (Compilation Success Rate), the fraction of projects passing strict compilation and runtime verification, and CLIP [27], the visual similarity to the reference image.

**Training Data.** Our training corpus combines the 110 non-test DesignBench pages (compilation-verified, multi-framework) with 2,000 projects sampled from **WebCode2M** [9] (filtered, decomposed into multi-file hierarchies, and deduplicated against the test set). All clean projects are processed by a corruption pipeline $\mathcal{C}$ (Appendix B) that injects realistic LLM error modes across varying repair scopes, yielding a comprehensive offline corpus of 27,430 state-action pairs.

Table 1: Main results on DesignBench. *HTML Ref.* reports the CLIP of the same executor generating a single HTML file, serving as a compilation-free visual reference. Best multi-framework result per executor in **bold**. All metrics are averaged over three independent runs.

| | | React | | Vue | | Angular | | HTML |
|---|---|---|---|---|---|---|---|---|
| **Executor** | **Method** | CSR | CLIP | CSR | CLIP | CSR | CLIP | CLIP |
| GPT-4o | *HTML Ref.* | – | – | – | – | – | – | *0.784* |
| | One-Shot | 97.3 | 0.758 | 94.9 | 0.769 | 71.1 | 0.591 | – |
| | **MESCoder** | **99.1** | **0.766** | **97.8** | **0.779** | **89.2** | **0.661** | – |
| Claude-3.7-Sonnet | *HTML Ref.* | – | – | – | – | – | – | *0.853* |
| | One-Shot | 95.4 | 0.813 | 97.5 | 0.827 | 68.7 | 0.597 | – |
| | **MESCoder** | **98.3** | **0.825** | **99.1** | **0.832** | **87.1** | **0.674** | – |
| Gemini-2.0-Flash | *HTML Ref.* | – | – | – | – | – | – | *0.749* |
| | One-Shot | 90.8 | 0.754 | 83.9 | 0.694 | 71.1 | 0.606 | – |
| | **MESCoder** | **96.1** | **0.777** | **92.4** | **0.722** | **87.3** | **0.663** | – |
| Qwen2.5-VL-72B | *HTML Ref.* | – | – | – | – | – | – | *0.809* |
| | One-Shot | 95.4 | 0.783 | 85.6 | 0.681 | 62.7 | 0.519 | – |
| | **MESCoder** | **97.6** | **0.792** | **93.1** | **0.706** | **82.4** | **0.572** | – |
| Llama-3.2-90B | *HTML Ref.* | – | – | – | – | – | – | *0.724* |
| | One-Shot | 94.5 | 0.698 | 74.6 | 0.537 | 73.5 | 0.528 | – |
| | **MESCoder** | **97.2** | **0.710** | **86.8** | **0.577** | **88.1** | **0.573** | – |
| Pixtral-124B | *HTML Ref.* | – | – | – | – | – | – | *0.796* |
| | One-Shot | 97.3 | 0.744 | 97.5 | 0.776 | 75.9 | 0.629 | – |
| | **MESCoder** | **98.8** | **0.750** | **99.0** | **0.781** | **91.1** | **0.694** | – |

**Baselines.** We compare MESCoder against four baselines that span from direct prompting to heuristic and learned self-correction: (1) **One-Shot**: Generates the full multi-file project in a single LLM forward pass without structural decomposition or post-hoc repair, serving as the vanilla generation reference. (2) **Rule Repair**: Augments a scaffold-based generation pipeline with hard-coded, heuristic fix templates for the 20 most frequent compilation errors. This represents state-of-the-art non-learned execution repair. (3) **RLEF** (RL from Execution Feedback): Keeps LLM executor frozen and trains an external controller using feedback. The controller modulates the executor through generic generation instructions, but does not use our action space. It serves as a monolithic RL baseline for testing whether sandbox-feedback optimization alone can match MESCoder. (4) **Offline-Only**: An ablation baseline that uses our exact scaffold representation and MLP policy trained exclusively on the offline demonstration corpus, omitting the online PPO exploration stage.

### 4.2 Main Results

Table 1 reports end-to-end results across three frameworks and six LLM executors. MESCoder consistently raises CSR over every executor–framework combination. For React and Vue, where One-Shot baselines are already strong, the gains are moderate yet consistent across all executors. For Angular, where strict TypeScript module-registration constraints make single-pass generation most brittle, the improvements are markedly larger. Crucially, even the strongest executors benefit substantially, demonstrating that cluster-scope repair addresses failure modes that raw executor capability alone cannot resolve. The same policy architecture and training recipe are applied identically across all six executors, so the uniform pattern of improvement constitutes strong evidence that MESCoder's benefit is architectural rather than executor-specific. Alongside the CSR improvements, CLIP rises in proportion: Angular sees the largest visual-fidelity gains, consistent with its steeper executability improvement, while React and Vue show smaller but reliable increases, confirming that executability and visual quality improve together rather than trading off. The *HTML Ref.* column provides an executor-matched upper bound; MESCoder narrows the gap to this ceiling substantially on Angular while approaching it on React and Vue. A complexity-stratified breakdown (Appendix A, Table 3) further confirms that MESCoder's advantage is most pronounced on Hard samples, where cluster-level correction becomes critical when failure chains span multiple files.

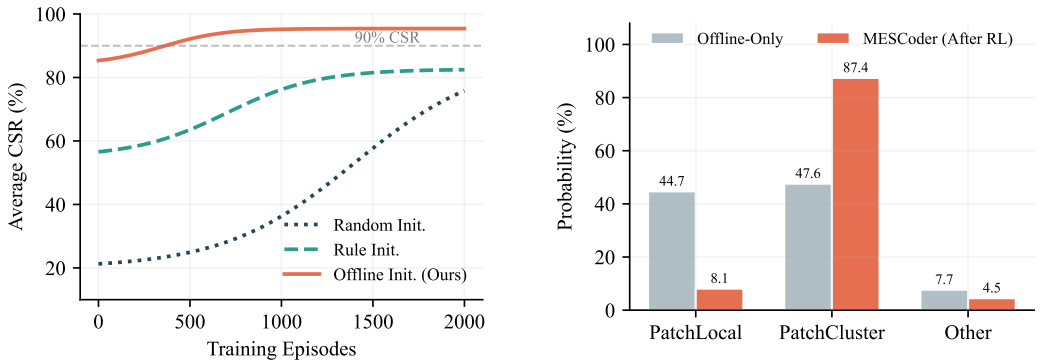



Figure 3: **Left**: Average CSR on the validation set during the online RL stage under three initialization conditions: behavior-cloning (offline) initialization, hand-coded rule initialization (rule policy used as a parameter seed, then fine-tuned with PPO), and random initialization. Offline initialization converges faster and plateaus higher, confirming the value of the offline prior. **Right**: Action distribution on stagnated-cluster states before and after RL fine-tuning. RL substantially increases the PATCHCLUSTER probability while reducing spurious PATCHLOCAL decisions on these hard states.

### 4.3 Policy Learning and Behavior

We analyze how the two-stage training procedure shapes the policy's self-correction behavior across training dynamics and scope escalation.

**Training dynamics and action shift.** Figure 3 illustrates training dynamics and the resulting action distribution shift. The offline-initialized policy reaches a stable, high-CSR plateau substantially faster than both random and rule-initialized baselines, confirming that the behavior-cloning stage provides a meaningful action prior that makes online RL tractable. The rule-initialized agent converges more slowly and stabilizes at a lower final CSR, as rigid heuristic rules cannot capture context-dependent scope decisions. On *stagnated-cluster* states—where repeated local repair has yielded no error reduction—the offline-only policy distributes probability nearly equally between PATCHLOCAL and PATCHCLUSTER; after online RL, PATCHCLUSTER rises to a clear dominant position while PATCHLOCAL drops sharply. Crucially, the shift is selective: this suggests that RL primarily increases cluster repair on states where local repair has stagnated, confirming that RL has learned to escalate scope *when and only when* local repair has demonstrably failed.

**Scope escalation analysis.** To directly quantify escalation behavior, we curate a diagnostic subset of stagnated-cluster episodes and compare methods on escalation rate, final CSR, average steps, and invalid patch count (Appendix D), together with two oracle variants: *Always-Local* (never escalates) and *Always-Cluster* (escalates unconditionally). *Always-Local* achieves the lowest CSR, unable to escape local stagnation; *Always-Cluster* achieves a perfect escalation rate but lower CSR than MESCoder, since indiscriminate broad edits disturb previously healthy files. MESCoder achieves the highest escalation rate among learned methods and the best final CSR, demonstrating that online RL teaches selective escalation.

### 4.4 Error-Dimension Analysis

Figure 4 characterizes MESCoder's gains across error families and frameworks. The left panel shows CSR gain over Rule Repair per error family (GPT-4o, injected-error subset): gains increase from syntactic failures (*Syntax & Template*) through intra-component errors (*Local Binding*, *Component Contract*) to cross-file coordination failures (*Assembly & Hierarchy*, *Module Dep. & Import/Export*, *Framework-Specific Engineering*)—validating that PATCHCLUSTER addresses the failure modes single-file repair cannot resolve. The right panel shows per-cell CSR across framework–error-family combinations; Angular exhibits the steepest difficulty gradient.

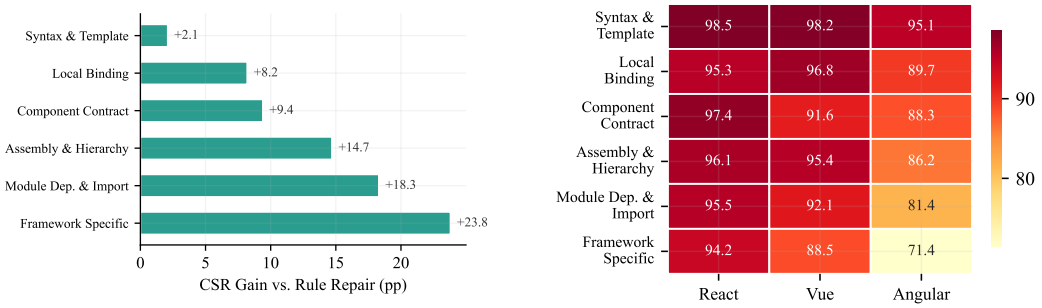



Figure 4: **Left**: CSR gain over Rule Repair per error family (GPT-4o, injected-error subset). **Right**: Per-cell CSR across framework–error combinations. Angular shows the steepest difficulty gradient.

### 4.5 Ablation Study

Table 2 presents a systematic ablation of MESCoder's key design choices, using GPT-4o as the executor and reporting average CSR across the three frameworks.

Table 2: Ablation study on average CSR (%) across React, Vue, and Angular (GPT-4o executor). Each row removes or replaces one design component from the full MESCoder system.

| **Variant** | Avg CSR | $\Delta$ vs. Full |
|---|---|---|
| Full MESCoder | 95.4 | – |
| *Architecture ablations* | | |
| w/o scaffold (flat generation) | 80.8 | $-14.6$ |
| replace MLP policy w/ rule policy | 82.6 | $-12.8$ |
| w/o PATCHCLUSTER action | 85.5 | $-9.9$ |
| w/o ASSEMBLE action | 89.7 | $-5.7$ |
| w/o history features $h_t$ | 91.5 | $-3.9$ |
| *Training ablations* | | |
| w/o offline init. (online RL only) | 86.2 | $-9.2$ |
| w/o online RL (= Offline-Only) | 84.1 | $-11.3$ |
| w/o action-relevant scope $\Omega_t$ | 88.3 | $-7.1$ |
| w/o PPO clipping | 92.2 | $-3.2$ |
| w/o entropy regularization | 92.7 | $-2.7$ |
| *Reward ablations* | | |
| w/o error-reduction reward | 81.2 | $-14.2$ |
| w/o task-commitment reward $u_t$ | 91.6 | $-3.8$ |
| w/o terminal quality $Q_T$ | 93.5 | $-1.9$ |

**Architecture and action space.** Removing the scaffold causes the largest CSR drop, confirming that decomposing project generation into localized subtree decisions is fundamental; replacing the MLP policy with a rule-based selector causes a comparably large drop, showing that context-dependent scope decisions cannot be captured by fixed heuristics. Removing PATCHCLUSTER leads to a substantially larger degradation than removing ASSEMBLE, validating that the ability to escalate to cross-file cluster repair is the more critical capability once local repair stagnates. Removing $h_t$ also incurs a penalty, confirming history-aware encoding is necessary for reliable stagnation detection.

**Training, scope, and reward.** Both removing online RL and removing offline initialization cause substantial CSR drops, with online RL contributing slightly more, reflecting complementary failure modes. Replacing $\Omega_t$ with full-project verification destabilizes training. Among rewards, $\alpha(e_t - e_{t+1})$ is dominant; $\beta u_t$ provides secondary support; $Q_T$ contributes a smaller but non-negligible gain, primarily by encouraging generation quality at episode termination. Inference efficiency (Appendix E, Table 8) and qualitative case studies (Appendix F) are provided in the appendix.

## 5 Conclusion

We presented MESCoder, a tree-structured self-corrective generation framework that addresses the challenge of executable multi-framework front-end code synthesis. By decomposing project construction into scaffold-guided subtree expansions and equipping a unified Project Agent with a learned policy for context-sensitive local and cluster-level repair under sandbox execution feedback, MESCoder transforms self-correction from a post-hoc step into an integral part of the generation loop. Experiments across React, Vue, and Angular with six LLM executors demonstrate consistent CSR improvements, most pronounced on Angular, where cross-file dependency failures are prevalent, while visual fidelity is preserved throughout.

## References


[1] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*, pages 1–6, 2018.

[2] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676, 2018.

[3] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[4] Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Liuqing Chen. Designcoder: Hierarchy-aware and self-correcting ui code generation with large language models. *arXiv preprint arXiv:2506.13663*, 2025.

[5] Jeonghun Cho, Deokhyung Kang, Hyounghun Kim, and Gary Geunbae Lee. Self-correcting code generation using small language models. *arXiv preprint arXiv:2505.23060*, 2025.

[6] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[7] Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4571–4585, 2024.

[8] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

[9] Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1834–1845, 2025.

[10] Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. Uicopilot: Automating ui synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1846–1855, 2025.

[11] Hongcheng Guo, Wei Zhang, Junhao Chen, Yaonan Gu, Jian Yang, Junjia Du, Shaosheng Cao, Binyuan Hui, Tianyu Liu, Jianxin Ma, et al. Iw-bench: Evaluating large multimodal models for converting image-to-web. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6449–6466, 2025.

[12] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

[13] Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. *arXiv preprint arXiv:2502.20380*, 2025.

[14] Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R Lyu, and Xiangyu Yue. Screencoder: Advancing visual-to-code generation for front-end automation via modular multimodal agents. *arXiv preprint arXiv:2507.22827*, 2025.

[15] Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024.

[16] Hugo Laurençon, Léo Tronchon, and Victor Sanh. Unlocking the conversion of web screenshots into html code with the websight dataset. *arXiv preprint arXiv:2403.09029*, 2024.

[17] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[18] Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. 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 3711–3726, 2025.

[19] Ryan Li, Yanzhe Zhang, and Diyi Yang. Sketch2code: Evaluating vision-language models for interactive web design prototyping. 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 3921–3955, 2025.

[20] Shanchao Liang, Nan Jiang, Shangshu Qian, and Lin Tan. Waffle: Fine-tuning multi-modal model for automated front-end development. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24786–24802, 2025.

[21] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

[22] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE transactions on software engineering*, 46(2):196–221, 2018.

[23] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*, pages 248–259. IEEE, 2015.

[24] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*, 2023.

[25] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

[26] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

[27] Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. 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 3956–3974, 2025.

[28] Yuxuan Wan, Yi Dong, Jingyu Xiao, Yintong Huo, Wenxuan Wang, and Michael R Lyu. Mrweb: An exploration of generating multi-page resource-aware web code from ui designs. *arXiv preprint arXiv:2412.15310*, 2024.

[29] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. Divide-and-conquer: Generating ui code from screenshots. *Proceedings of the ACM on Software Engineering*, 2(FSE):2099–2122, 2025.

[30] Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1123–1145, 2025.

[31] Jie Wu, Haoran Ma, Shisong Tang, Yulin Xu, Xiaoyu Kang, and Jiechao Gao. MulF-Coder: Framework-conditioned multi-agent for MLLM-based multi-framework front-end code generation. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=xUW3aH4ie3.

[32] Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 241–253. IEEE, 2025.

[33] Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R Lyu. Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*, 2025.

[34] Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Q Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, et al. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *Advances in neural information processing systems*, 37:112134–112157, 2024.

## A Difficulty-Stratified Results

**Complexity stratification protocol.** We partition the 200 test pages into three difficulty tiers using scaffold tree depth $D$ and inter-file dependency edge count $E$. A page is **Easy** if $D \leq 3$ and $E \leq 4$; **Hard** if $D \geq 6$ or $E \geq 10$; and **Medium** otherwise. This yields 68 Easy, 87 Medium, and 45 Hard.

**Aggregate difficulty results.** Table 3 reports CSR and CLIP averaged across all three frameworks. MESCoder's CSR advantage grows monotonically from Easy through Medium to Hard, directly reflecting the increasing importance of cluster-level repair as cross-file dependencies deepen. On Hard pages, Offline-Only drops sharply because its policy—trained without online exploration—consistently misclassifies stagnated cluster failures as local ones and exhausts its step budget.

Table 3: Results stratified by page complexity (GPT-4o executor, averaged across React / Vue / Angular). Complexity is defined by scaffold tree depth $D$ and inter-file dependency edge count $E$.

| **Method** | Easy CSR | Med CSR | Hard CSR | Easy CLIP | Med CLIP | Hard CLIP |
|---|---|---|---|---|---|---|
| One-Shot | 95.1 | 87.4 | 77.8 | 0.712 | 0.669 | 0.583 |
| Rule Repair | 96.3 | 90.1 | 80.3 | 0.726 | 0.683 | 0.601 |
| RLEF | 96.8 | 92.6 | 85.2 | **0.741** | 0.697 | 0.629 |
| Offline-Only | 96.6 | 81.6 | 70.0 | 0.737 | 0.674 | 0.598 |
| **MESCoder** | **98.6** | **95.3** | **91.7** | 0.739 | **0.701** | **0.632** |

**Per-framework difficulty breakdown.** Table 4 disaggregates Hard-tier results by framework. Angular shows the steepest difficulty gradient across all methods, reflecting that its strict module-wiring constraints make deep-hierarchy pages disproportionately failure-prone; React and Vue Hard pages are substantially easier, with all methods achieving higher CSR.

Table 4: Hard-tier CSR (%) per framework (GPT-4o executor).

| **Method** | Hard React | Hard Vue | Hard Angular |
|---|---|---|---|
| One-Shot | 88.6 | 86.3 | 58.4 |
| Rule Repair | 90.2 | 88.7 | 62.1 |
| RLEF | 93.4 | 91.8 | 70.3 |
| Offline-Only | 82.6 | 80.1 | 47.2 |
| **MESCoder** | **96.3** | **95.1** | **83.7** |

## B Offline Data Construction

**Clean project sources.** We use 110 DesignBench training pages plus 2,000 pages from WebCode2M. The WebCode2M pages are filtered for compilation correctness, decomposed into multi-file framework hierarchies, and deduplicated against the test set.

**Corruption operators.** The corruption pipeline $\mathcal{C}$ applies six operators, each targeting a distinct error family, to generate realistic error-inducing states from clean projects (Table 5):

**Result-driven repair labeling.** For each corrupted state, the repair label is assigned by calling the LLM executor $K{=}4$ times per candidate action and selecting the action that achieves the highest sandbox repair score. After filtering, we retain 27,430 labeled pairs.

## C Implementation Details

The scaffold preprocessor uses a pretrained visual detector and rule-based grouping to construct hierarchical layout trees. The lightweight MLP policy encodes the multimodal state $s_t$ via frozen encoders into a latent vector, then outputs an action distribution over the hierarchical action space. We

Table 5: Corruption operators used in offline data construction.

| Operator | Target Family | Description |
|---|---|---|
| $\mathcal{C}_{\text{syn}}$ | Syntax & Template | Removes closing tags, introduces invalid JSX attribute syntax, or misbalances template braces. |
| $\mathcal{C}_{\text{bind}}$ | Local Binding | Renames a prop or variable reference to an undefined name within a single component. |
| $\mathcal{C}_{\text{cont}}$ | Component Contract | Changes a component's exported interface without updating its parent call site. |
| $\mathcal{C}_{\text{asm}}$ | Assembly & Hierarchy | Removes or misroutes an import connecting a child component to its parent. |
| $\mathcal{C}_{\text{mod}}$ | Module Dep. & Import | Renames or removes an export without updating downstream import sites, producing missing-export or name-mismatch errors. |
| $\mathcal{C}_{\text{fw}}$ | Framework Specific | Removes an Angular component from `@NgModule`, or introduces a Vue `setup()` hook incompatible with Options API. |

evaluate across six frozen LLM executors: GPT-4o, Claude-3.7-Sonnet, Gemini-2.0-Flash, Qwen2.5-VL-72B, Llama-3.2-90B, and Pixtral-124B. Experiments are run on 8×A100 GPUs, and all results are averaged over three independent runs.

## D Scope Escalation Detail

Table 6 compares five strategies on escalation rate, final CSR, average steps, and invalid patch count. The two oracle variants bound the range: *Always-Local* stagnates without ever escalating; *Always-Cluster* escalates unconditionally but disturbs healthy files, yielding lower CSR than MESCoder. Among learned methods, MESCoder achieves the highest escalation rate and the best CSR, demonstrating that online RL teaches context-sensitive scope escalation.

Table 6: Scope escalation analysis on the stagnated-cluster diagnostic subset.

| Method | Escal. Rate (%) | CSR (%) | Avg Steps | Inv. Patches |
|---|---|---|---|---|
| *Always-Local* | 0.0 | 21.7 | 15.2 | 6.12 |
| *Always-Cluster* | 100.0 | 44.8 | 9.3 | 1.89 |
| Rule Repair | 26.3 | 39.2 | 12.4 | 3.71 |
| Offline-Only | 57.8 | 58.3 | 10.8 | 2.43 |
| **MESCoder** | **90.3** | **76.2** | **11.3** | **1.12** |

**Per-framework escalation breakdown.** Table 7 disaggregates escalation rate and CSR by framework for MESCoder and the two strongest baselines. Angular exhibits the highest stagnated-cluster rate (52 of the 87 episodes come from Angular), consistent with its strict module requirements. MESCoder's escalation rate on Angular stagnated-cluster episodes (91.4%) is the highest across all frameworks, while its CSR (69.8%) reflects that even correct escalation does not fully resolve all Angular failures—the deepest dependency chains remain beyond the episode horizon. React and Vue show lower stagnation frequencies (18 and 17 episodes respectively) and higher post-escalation CSR, as their dependency graphs are shallower and cluster edits more reliably sufficient.

## E Inference Efficiency Detail

Table 8 profiles the inference cost of each method (GPT-4o, averaged across frameworks). MESCoder's sandbox invocations exceed its step count because each PATCHCLUSTER action triggers pre- and post-patch verification. RLEF and Offline-Only operate at comparable interaction depth yet achieve notably lower CSR, confirming that structured two-stage training is the key differentiator.

Table 7: Per-framework escalation rate and CSR on the stagnated-cluster subset (GPT-4o executor). Episode counts reflect the frequency of stagnated-cluster episodes per framework.

| **Method** | **Metric** | React ($n$=18) | Vue ($n$=17) | Angular ($n$=52) |
|---|---|---|---|---|
| Rule Repair | Escal. Rate (%) | 27.8 | 29.4 | 25.0 |
| | CSR (%) | 44.4 | 47.1 | 34.6 |
| Offline-Only | Escal. Rate (%) | 66.7 | 64.7 | 52.3 |
| | CSR (%) | 72.2 | 70.6 | 49.4 |
| **MESCoder** | Escal. Rate (%) | **88.9** | **88.2** | **91.4** |
| | CSR (%) | **88.9** | **82.4** | **69.8** |

Table 8: Inference efficiency on DesignBench averaged across React / Vue / Angular.

| **Method** | Avg CSR | Avg Steps | Sandbox Calls | LLM Calls | Wall Time (s) |
|---|---|---|---|---|---|
| One-Shot | 87.8 | 1.0 | 1.0 | 1.0 | 18.3 |
| Rule Repair | 89.4 | 4.1 | 8.6 | 4.1 | 52.7 |
| RLEF | 91.8 | 7.3 | 7.3 | 7.3 | 94.2 |
| Offline-Only | 84.1 | 8.4 | 8.4 | 8.4 | 109.1 |
| **MESCoder** | **95.4** | **10.2** | **14.7** | **10.2** | **134.6** |

## F Qualitative Analysis

**Error injection and repair.** We present three concrete case studies drawn from DesignBench to illustrate how operator $\mathcal{C}$ injects representative errors and how MESCoder resolves each.

*Listing 1. Wrong React code ($\mathcal{C}_{bind}$).*

```jsx
import { useState } from "react";

export default function App() {
  const [todos, setTodos] = useState([
    { text: "Write a new blog post.", completed: false },
    { text: "Pick up laundry.",       completed: false },
  ]);

  const handleToggleComplete = (index) => {
    const updated = todos.map((todo, i) =>
      i === index
        ? { ...todo, complete: !todo.complete }  // renamed by C_bind
        : todo
    );
    setTodos(updated);
  };

  return (
    <ul>
      {todos.map((todo, i) => (
        <li key={i} onClick={() => handleToggleComplete(i)}
          style={{ textDecoration: todo.completed ? "line-through" : "none" }}>
          {todo.text}
        </li>
      ))}
    </ul>
  );
}
```

*Listing 2. Fixed React code.*

```jsx
import { useState } from "react";

export default function App() {
  const [todos, setTodos] = useState([
    { text: "Write a new blog post.", completed: false },
    { text: "Pick up laundry.",       completed: false },
  ]);

  const handleToggleComplete = (index) => {
    const updated = todos.map((todo, i) =>
      i === index
        ? { ...todo, completed: !todo.completed }  // Fixed
        : todo
    );
    setTodos(updated);
  };

  return (
    <ul>
      {todos.map((todo, i) => (
        <li key={i} onClick={() => handleToggleComplete(i)}
          style={{ textDecoration: todo.completed ? "line-through" : "none" }}>
          {todo.text}
        </li>
      ))}
    </ul>
  );
}
```

*Listing 3. Wrong Vue code ($C_{cont}$).*

```
const ResourceCard = (props: {
  iconName: string;  // renamed by C_cont
  title: string;
  description: string;
}) => (
  <div class="bg-white shadow rounded-lg p-6">
    <i class={`fas fa-${props.iconName} text-blue-500`}></i>  // renamed by
    C_cont
    <h2 class="text-lg font-medium">{props.title}</h2>
    <p class="text-sm text-gray-500">{props.description}</p>
  </div>
);

<div class="grid grid-cols-1 md:grid-cols-3 gap-6">
  <ResourceCard icon="file-text"
                title="Blog"
                description="Explore our learnings..." />
  <ResourceCard icon="clipboard-check"
                title="Changelog"
                description="Stay updated..." />
  <ResourceCard icon="users"
                title="Community"
                description="Join the discussion..." />
</div>
```

*Listing 4. Fixed Vue code.*

```
const ResourceCard = (props: {
  iconName: string;
  title: string;
  description: string;
}) => (
  <div class="bg-white shadow rounded-lg p-6">
    <i class={`fas fa-${props.iconName} text-blue-500`}></i>
    <h2 class="text-lg font-medium">{props.title}</h2>
    <p class="text-sm text-gray-500">{props.description}</p>
  </div>
);

<div class="grid grid-cols-1 md:grid-cols-3 gap-6">
  <ResourceCard iconName="file-text" // Fixed
                title="Blog"
                description="Explore our learnings..." />
  <ResourceCard iconName="clipboard-check" // Fixed
                title="Changelog"
                description="Stay updated..." />
  <ResourceCard iconName="users" // Fixed
                title="Community"
                description="Join the discussion..." />
</div>
```

*Listing 5. Wrong Angular code ($C_{fw}$).*

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { IotDashboardComponent } from
  './iot-dashboard/iot-dashboard.component';
// Missing: TemperatureDraggerComponent
// Missing: RoomSelectorComponent

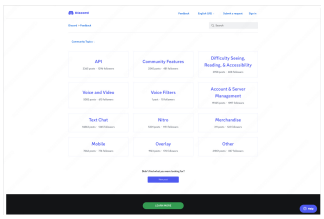
@NgModule({
  declarations: [
    IotDashboardComponent,
    // TemperatureDraggerComponent removed by C_fw
    // RoomSelectorComponent removed by C_fw
  ],
  imports: [
    CommonModule,
    // NbCardModule removed by C_fw
    // NbIconModule removed by C_fw
    // NbTabsetModule removed by C_fw
  ],
})
export class DashboardModule { }
```

*Listing 6. Fixed Angular code.*

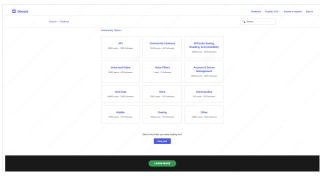
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NbCardModule, NbIconModule,
         NbTabsetModule } from '@nebular/theme'; // Fixed
import { IotDashboardComponent } from
  './iot-dashboard/iot-dashboard.component';
import { TemperatureDraggerComponent } from
  './iot-dashboard/temperature-dragger/
  temperature-dragger.component';           // Fixed
import { RoomSelectorComponent } from
  './iot-dashboard/room-selector/
  room-selector.component';                 // Fixed

@NgModule({
  declarations: [
    IotDashboardComponent,
    TemperatureDraggerComponent,  // Fixed
    RoomSelectorComponent,        // Fixed
  ],
  imports: [
    CommonModule,
    NbCardModule,    // Fixed
    NbIconModule,    // Fixed
    NbTabsetModule,  // Fixed
  ],
})
export class DashboardModule { }
```

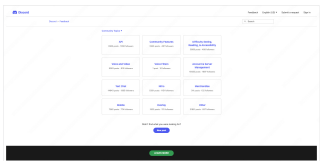
**Cross-framework rendering comparison.** Figure 5 shows qualitative rendering results for the same input UI across HTML, React, Vue, and Angular, alongside the original reference screenshot. The HTML output serves as a generation baseline, while MESCoder outputs for the three framework targets consistently match the reference layout.



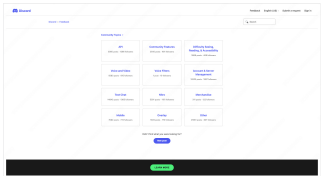
(a) Original



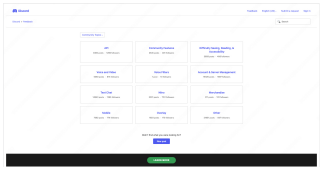
(b) HTML



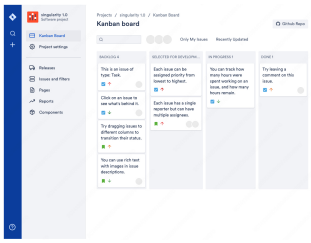
(c) React



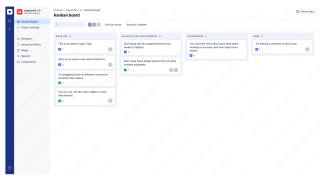
(d) Vue



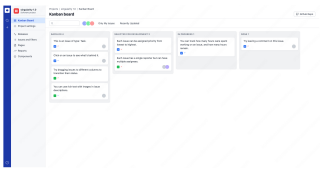
(e) Angular



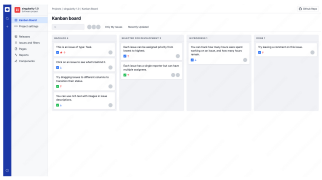
(f) Original



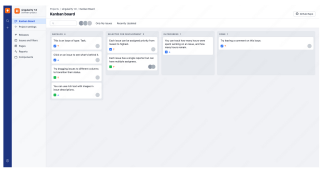
(g) HTML



(h) React



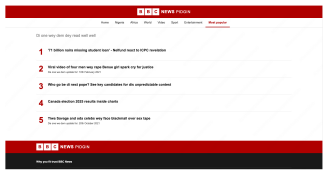
(i) Vue



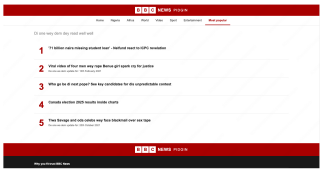
(j) Angular



(k) Original



(l) HTML



(m) React



(n) Vue



(o) Angular

Figure 5: Qualitative rendering results of MESCoder across three UI test cases. For each case, we show the original reference screenshot (Original), an HTML generation baseline (HTML), and MESCoder outputs for three component-based frameworks (React, Vue, Angular). MESCoder faithfully reproduces the reference layout across all target frameworks.

# G LLM Executor Prompt Templates

This appendix provides the complete prompt templates used for each action category in the MESCoder executor interface. All prompts share a common system preamble that establishes the task context; per-action sections then specify the concrete instruction, the project context, and the expected output format. Identical templates are used across all six LLM executors; no executor-specific prompt engineering is applied. Variables enclosed in ⟨⟩ are filled by the framework at inference time.

## G.1 System Preamble (Shared Across All Actions)

The following system message is prepended to every LLM call:

```
You are a senior front-end engineer specializing in ⟨framework⟩
(one of React, Vue, Angular). You are assisting an automated
code-generation agent that constructs a multi-file ⟨framework⟩
project from a UI screenshot. The project is built incrementally:
at each step you receive a partial project state, the current
compilation errors (if any), and an action instruction. Your
response must follow the structured output format exactly: produce
only the requested code edits, with no surrounding prose, no
markdown code fences outside the prescribed format, and no
commentary. Each edit is a JSON object: {''file'': ''<path>'', ''old'':
''<old_content>'', ''new'': ''<new_content>''}. If you are generating a
new file, set ''old'' to an empty string.
```

## G.2 Continuation Prompt (CONTINUE Action)

The CONTINUE action advances the current task when the partial project is sufficiently healthy, generating code for the next scaffold node. The prompt is structured as follows:

```
[TASK] Generate the ⟨framework⟩ component for the UI element
described below.

[SCAFFOLD NODE]
- Type: ⟨node_type⟩ (e.g., ''Card'', ''NavBar'', ''DataTable'')
- Bounding box (normalized): ⟨x_min, y_min, x_max, y_max⟩
- Tree depth: ⟨depth⟩
- Children to generate later: ⟨child_count⟩

[CONTENT SLOTS]
⟨slot_table⟩ (Each slot: {node_id, text_content,
img_crop_description})

[SCREENSHOT DESCRIPTION]
⟨screenshot image or visual description appended here⟩

[EXISTING PROJECT FILES]
⟨list of already-generated files with their current content⟩

[INSTRUCTION]
Generate the file ⟨target_filename⟩ for this component. Implement
the visual layout faithfully. Use only ⟨framework⟩-idiomatic
patterns. Do not import any external UI libraries unless they are
already present in package.json. Declare all props with correct
TypeScript types.

[OUTPUT FORMAT]
{"file": "⟨target_filename⟩", "old": "", "new": "<complete file
content>"}
```

### G.3 Assembly Prompt (ASSEMBLE Action)

The ASSEMBLE action integrates completed child units into the current parent node, resolving imports and type compatibility.

```
[TASK] Assemble the parent component ⟨parent_filename⟩ by integrating
the following child components that have already been generated.

[PARENT NODE]
- Type: ⟨parent_node_type⟩
- Expected children: ⟨list of child component names and filenames⟩

[CHILD COMPONENT INTERFACES]
⟨For each child: filename, exported component name, prop interface
definition⟩

[EXISTING PARENT FILE (if any)]
⟨current content of the parent file, or empty string if new⟩

[CURRENT DIAGNOSTIC]
⟨sandbox error messages for the current scope, or "No errors" if
clean⟩

[INSTRUCTION]
Write the complete ⟨parent_filename⟩ that: (1) imports all child
components with correct relative paths; (2) uses each child
component with correct props in the parent JSX/template; (3)
passes no incorrect prop types and omits no required props; (4) is
compatible with the framework's assembly conventions (e.g., Angular
@NgModule declarations, Vue components option, React named imports).

[OUTPUT FORMAT]
{"file": "⟨parent_filename⟩", "old": "⟨old content or empty⟩",
"new": "⟨new content⟩"}
```

### G.4 Local Repair Prompt (PATCHLOCAL Action)

The PATCHLOCAL action invokes a local repair over a small scope, targeting a single file identified as the primary error source.

```
[TASK] Fix the compilation error in a single file.

[ERROR SUMMARY]
⟨diagnostic summary: dominant error family, error messages, line
numbers⟩

[TARGET FILE]
Filename: ⟨target_filename⟩
⟨full file content⟩

[DIRECT IMPORT DEPENDENTS (read-only context)]
⟨for each file that imports target_filename: filename and content⟩

[CONSTRAINT]
You may ONLY edit ⟨target_filename⟩. Do NOT modify any dependent
files. Do NOT change the component's exported interface (prop names,
types, return shape) unless the error explicitly requires it.

[INSTRUCTION]
Produce a minimal, correct fix for the listed errors. Preserve all
existing functionality outside the error site.

[OUTPUT FORMAT]
```

{"file": "⟨target_filename⟩", "old": "⟨exact old content⟩", "new": "⟨corrected content⟩"}

### G.5 Cluster Repair Prompt (PATCHCLUSTER Action)

The PATCHCLUSTER action invokes a broader repair over a small dependency or assembly cluster identified by the sandbox.

```
[TASK] Fix a cross-file dependency error spanning multiple files.

[ERROR CLUSTER SUMMARY]
Error family: ⟨one of: Assembly & Hierarchy; Module Dep. &
Import/Export; Framework-Specific Engineering⟩
Affected files: ⟨list of filenames in the dependency cluster⟩
Error messages:
⟨full compiler/runtime error output for the cluster⟩

[CLUSTER FILE CONTENTS]
⟨for each file in the cluster: filename and full content⟩

[INSTRUCTION]
Produce a coordinated set of edits that resolves ALL listed errors
simultaneously. You MAY edit any subset of the files in the cluster.
Ensure that after your edits: (1) no new errors are introduced
in files outside the cluster, (2) previously passing files in the
project are not broken, (3) the overall UI layout and component
hierarchy is preserved.

[OUTPUT FORMAT]
Return a JSON array of edit objects, one per modified file:
[{"file": "⟨filename_1⟩", "old": "⟨old⟩", "new": "⟨new⟩"},
{"file": "⟨filename_2⟩", "old": "⟨old⟩", "new": "⟨new⟩"},
...]
```

### G.6 Finish Prompt (FINISH Action)

The FINISH action terminates the episode when the project is complete and executable, applying any remaining fixes in a final verification pass.

```
[TASK] Perform a final review and apply any remaining fixes to
ensure the project compiles and renders correctly as a whole.

[FULL PROJECT STATE]
⟨all project files listed with their content⟩

[FINAL DIAGNOSTIC]
⟨output of full-project verification (tsc / vue-tsc / ng build); or
"All checks pass"⟩

[INSTRUCTION]
If the final diagnostic reports errors, fix them with minimal
changes. If the diagnostic is clean, verify that: (1) package.json
contains all required dependencies; (2) the entry point (e.g.,
src/main.tsx, src/main.ts) correctly mounts the root component; (3)
all image or asset references resolve to valid paths. Return only
the files that require changes; return an empty array if the project
is already complete and correct.

[OUTPUT FORMAT]
JSON array of edit objects (may be empty [] if no changes needed).
```

## H Limitations

Our framework improves executable multi-framework front-end generation through execution feedback and self-correction, but it still has several limitations. First, the system depends on the quality of the scaffold preprocessor and the runtime sandbox. Incorrect structure extraction or incomplete framework-specific checks may mislead the policy and the LLM executor. Second, although the policy learns high-level corrective decisions, the actual code edits are still produced by the LLM executor; when the executor repeatedly fails to generate valid patches, the policy alone cannot fully recover the project. Third, online reinforcement learning introduces additional computational cost due to repeated LLM and sandbox calls, which may limit deployment in latency-sensitive settings. Finally, our method focuses on syntactic, structural, and build-time executability, while broader software engineering aspects such as security, accessibility, maintainability, and long-term code quality are not fully addressed.

## I Broader Impacts

This work may have positive societal impacts by reducing the cost of front-end development, enabling faster prototyping, and making UI implementation more accessible to designers, small teams, and developers with limited framework expertise. By explicitly emphasizing executable multi-framework projects, the method may also reduce debugging burden and improve the reliability of AI-assisted software generation. At the same time, the work has potential negative impacts. More capable automated code generation could contribute to low-quality mass-produced web interfaces, overreliance on generated code, or deployment of projects whose security, accessibility, or privacy properties have not been adequately reviewed. It may also affect demand for certain routine front-end implementation tasks. Therefore, generated projects should be treated as developer-assistance outputs rather than fully autonomous production-ready systems, and should undergo human review, security checks, and accessibility validation before deployment.

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the main scope, method, and empirical claims.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have dicussed three limitations in Appendix H.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: This paper does not include formal theoretical results or proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: See Section 4.1, and Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Our code is publicly available.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: In Section 4

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: In Section 4

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See Appendix C and Appendix E.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: The work is a front-end code generation method and appears consistent with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes].

    Justification: We discussed impacts in Appendix I.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risks

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Properly cited and stated

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The code we released in this paper contains related documents.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Not related

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: Not related

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: As described in the methodology, MESCoder uses an action-conditioned LLM executor to generate and patch code under policy-selected actions. Any additional use of LLMs for polishing or formatting did not affect the research methods or contributions.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.