# GUI-GENESIS: AUTOMATED SYNTHESIS OF EFFICIENT ENVIRONMENTS WITH VERIFIABLE REWARDS FOR GUI AGENT POST-TRAINING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Post-training GUI agents in interactive environments is critical for developing generalization and long-horizon planning capabilities. However, training on real-world applications is hindered by high latency, poor reproducibility, and unverifiable rewards relying on noisy visual proxies. To address the limitations, we present GUI-GENESIS, the first framework to automatically synthesize efficient GUI training environments with verifiable rewards. GUI-GENESIS reconstructs real-world applications into lightweight, task-conditioned web environments using multimodal code models and equips them with *code-native rewards*, executable assertions that provide deterministic reward signals and eliminate visual estimation noise. Extensive experiments on the WeChat mini-app platform (serving over one billion active users) show that GUI-GENESIS reduces interaction latency by $10\times$ and cuts estimated recurring rollout and verification costs by over 98% compared with training on real mini-apps, excluding one-time preparation costs. Notably, agents trained with GUI-GENESIS-synthesized environments achieve performance comparable to agents post-trained in real-world environments, demonstrating robust zero-shot sim-to-real transfer and establishing automated environment synthesis as a scalable and cost-effective alternative to direct real-world training. We open-source GUI-GENESIS on our anonymous website `https://anonymous.4open.science/r/gui_genesis-B23B/`.

## 1 INTRODUCTION

GUI agents powered by vision-language models have achieved remarkable success in automating digital tasks, yet they increasingly require interactive post-training to master long-horizon planning and dynamic error recovery (Hong et al., 2024; Wang et al., 2025a; Xu et al., 2025b; Ran et al., 2024). While training on static datasets allows models to clone basic behavioral patterns, such approaches often fail when agents encounter out-of-distribution states or complex workflows that necessitate trial-and-error (Torabi et al., 2018; Rawles et al., 2023; Zhang et al., 2024). To bridge the gap, the community is shifting toward reinforcement learning (RL), where agents refine their decision-making policies through continuous exploration within interactive GUI applications (Yue et al., 2025; Zhang et al., 2025). This transition highlights the need for interactive environments that can support high-throughput sampling required for policy optimization.

However, utilizing real-world GUI apps as interactive environments fundamentally constrains the efficacy and scalability of GUI agent post-training regarding computational efficiency and reward verification (Rawles et al., 2025; Zhou et al., 2024; Xie et al., 2024). First, real-world applications often mandate active user login sessions and depend on remote backend synchronization (Ran et al., 2023; 2025b). These heavy external dependencies not only introduce stochastic instability into the training loop but also impose prohibitive infrastructure burdens that render large-scale parallel training expensive (Deng et al., 2023). Second, real-world apps inherently lack precise and verifiable rewards (Wang et al., 2025a), as the ground-truth task state is often encapsulated within the backend or rendered implicitly in the interface (Kaelbling et al., 1998; Rawles et al., 2025). In the absence of a verifiable oracle, training pipelines are forced to employ VLMs as proxy judges to estimate task progress (Gu et al., 2025; Xu et al., 2025a; Wang et al., 2025a). These heavy-weight evaluators not



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

only introduce additional inference costs but also stochastic noise due to visual hallucinations and alignment errors, undermining the potential of the RLVR recipe (Guo et al., 2025; He et al., 2025).

To overcome the efficiency and verifiability challenges, we introduce GUI-GENESIS, the first framework that simultaneously addresses both challenges by synthesizing lightweight applications equipped with embedded reward oracles from real mobile applications. Driven by the insight that full backend complexity is unnecessary for agent training, GUI-GENESIS leverages VLMs and code LLMs to reverse-engineer user traces into standalone web applications that preserve logical fidelity while eliminating network and resource overhead. Beyond structural reconstruction, GUI-GENESIS explicitly addresses the reward verification problem by identifying task success conditions during synthesis and embedding executable assertions directly into the source code as shown in Section 4.4. Such code-native reward provides state-based supervision, transforming noisy visual reward estimates into deterministic feedback for policy optimization.

To evaluate GUI-GENESIS, we conduct experiments on a benchmark of 149 challenging real-world tasks on the WeChat mini-app platform, which serves over one billion monthly active users. We post-train a Qwen-2.5-VL-32B (Bai et al., 2025b) model with standard Multistep GRPO (Shao et al., 2024) in GUI-GENESIS-synthesized environments against both the base model without post-training and the model trained directly in the real-world environment. Experiment results show that GUI-GENESIS substantially boosts training and cost efficiency, reducing interaction latency by $10\times$ and cutting estimated recurring rollout and verification costs by over 98% per epoch compared to real-world-based training with VLM rewards, excluding one-time preparation costs, including code synthesis and trace collection. Beyond training efficiency and cost saving, experiment results show that agents post-trained solely in our synthesized environments outperform the base model by a relative margin of 14.54% and achieve performance comparable to agents trained directly in real-world applications under the same protocol, slightly exceeding them by a relative margin of 3.27%. These results demonstrate that GUI-GENESIS achieves robust zero-shot sim-to-real transfer, establishing environment synthesis as a scalable and cost-effective alternative to training GUI agents in real-world environments.

Our main contributions are as follows:

- We identify and analyze the efficiency and verifiability bottlenecks of training GUI agents in real-world applications, and propose *environment synthesis* as a scalable, white-box alternative to interaction with black-box apps.
- We introduce GUI-GENESIS, the first framework that leverages multimodal code models to synthesize lightweight web environments from real mobile applications equipped with **code-native rewards**, which eliminate visual estimation noise for reward signals.
- Extensive experiments demonstrate that GUI-GENESIS reduces environment latency by $10\times$ and saves 98.69% of estimated recurring rollout and verification costs per epoch while achieving comparable performance against training in real-world applications.

## 2 BACKGROUND AND RELATED WORK

**GUI agents and the shift to RL.** Recent advancements in VLMs have empowered agents to automate tasks across web and mobile platforms (Bai et al., 2025a; Qin et al., 2025; Gou et al., 2024; You et al., 2024). While early approaches primarily utilized Supervised Fine-Tuning (SFT) on static datasets (Wu et al., 2024; Lin et al., 2025; Lu et al., 2024; Chen et al., 2024; Yang et al., 2025), they often struggle with error recovery and long-horizon planning. Consequently, the field is pivoting toward Reinforcement Learning (RL) to enable agents to learn from interaction (Ye et al., 2025; Xu et al., 2025a; Wang et al., 2025a; Lu et al., 2025). Nowadays, GUI agents typically train on real-world applications (e.g., Android emulators), which are computationally expensive, slow, and prone to network instability, severely limiting the scale of data collection.

**Environment synthesis for GUI and web agents.** Very recent and concurrent work has explored synthetic environments as a way to mitigate the scarcity and stability challenges of collecting real-world GUI interaction data, primarily targeting data scalability rather than training-time efficiency. InfiniteWeb (Zhang et al., 2026) and AutoWebWorld (Wu et al., 2026b) synthesize large-scale functional web environments from abstract specifications, while VeriEnv (Chae et al., 2026) focuses on

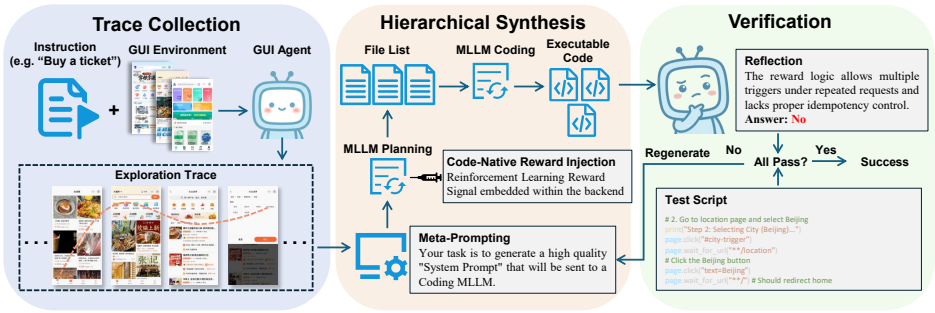


Figure 1: Overview of GUI-GENESIS

the safety of online exploration by generating synthetic copies from online websites. MobileGym (Wu et al., 2026a) provides browser-based mobile environments with structured state and deterministic task verification.

GUI-GENESIS differs from these concurrent efforts in three ways. First, rather than performing web-to-web synthesis, it bridges real-world mobile mini-apps to lightweight web environments, using the browser stack as a high-throughput training surrogate. Second, it grounds generation in real agent execution traces rather than abstract specifications or static screenshots, preserving task-relevant application logic. Third, instead of treating rewards as post-hoc artifacts or relying on external validators, it synthesizes deterministic, sub-goal-decomposed code-native rewards directly in the environment backend.

## 3 PROBLEM FORMULATION

We consider training GUI agents through environment interactions. The target application is modeled as a POMDP $\mathcal{M}_{\text{real}} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega \rangle$, where states $s \in \mathcal{S}$ are latent (e.g., backend databases), observations $o \in \Omega$ are visual screenshots, and rewards $\mathcal{R}(s, a)$ depend on the unobservable true state. Agents learn policies $\pi(a|o)$ to maximize expected cumulative reward. Training on $\mathcal{M}_{\text{real}}$ brings two major bottlenecks:

**Efficiency bottleneck.** Each real-world interaction incurs latency from network, rendering, and I/O—often exceeding seconds per step. At the scale required for effective RL, wall-clock training time becomes intractable. Parallelization needs expensive mobile devices which is financially constrained (e.g., ∼$24K/day for 100 concurrent cloud instances (Services, 2025a;b)).

**Verification bottleneck.** Ground-truth rewards $\mathcal{R}(s, a)$ are latent, forcing training to rely on noisy visual proxies $\hat{R}(o, a)$ from VLMs (Wang et al., 2025a). Estimation error $\epsilon(o)$ risks reward hacking, while heavy VLM inference overhead compounds the efficiency bottleneck.

**Synthesis objective.** Given traces $\mathcal{D}$, construct task-conditioned $\mathcal{M}_{\text{syn}}$ satisfying:

- **Fidelity:** $\pi^*_{\text{syn}} \approx \pi^*_{\text{real}}$ on the task manifold.
- **Verifiability:** deterministic code-assertion rewards.
- **Efficiency:** $\tau_{\text{step}}(\mathcal{M}_{\text{syn}}) \ll \tau_{\text{step}}(\mathcal{M}_{\text{real}})$.

Critically, unlike learned neural simulators like world models (Luo et al., 2025; Li et al., 2025), $\mathcal{M}_{\text{syn}}$ is *synthesized* as executable code, ensuring the white-box verifiability and near-instantaneous simulation.

## 4 METHODOLOGY

### 4.1 OVERVIEW

While environment synthesis offers a principled solution to the latency and verification bottlenecks of real-world apps, generating full-fidelity replicas of production applications remains beyond the

capability of current code LLMs (Ran et al., 2025a). However, training GUI agents does not require globally complete simulation: since agents are optimized for specific instructions, the environment need only preserve behavioral fidelity along the task-relevant state trajectory. Building on this insight, we propose a *task-conditioned synthesis* strategy. Rather than reconstructing an entire application, GUI-GENESIS synthesizes a dedicated, standalone web environment (implemented with Flask (Pallets, 2024)) for each RL task, encapsulating only the interaction logic related to the task.

As illustrated in Figure 1, GUI-GENESIS has three stages: (1) **trace-driven context acquisition** (§4.2), which captures visual and logical context from real-world interactions; (2) **hierarchical code synthesis** (§4.3), which constructs the environment and injects code-native rewards using a multimodal code model; and (3) **automated verification** (§4.5), which checks that the synthesized environment is valid and executable.

### 4.2 TRACE-DRIVEN CONTEXT ACQUISITION

We ground synthesis in actual execution traces to preserve real-world visual and interaction dynamics. For each training task, a baseline GUI agent attempts the task on the target application, yielding an exploration trace $\mathcal{T} = \{(I_0, u_0), \ldots, (I_T, u_T)\}$, where $I_t$ is the raw screenshot and $u_t$ is the user action at step $t$.

Crucially, we utilize these traces *regardless of task success*. Even failed attempts effectively capture the *visual design patterns* (e.g., color schemes, widget styles) and the *logical flow* (e.g., page transitions). This trace $\mathcal{T}$ serves as the multimodal context for the synthesizer, enabling the model to reconstruct the features of the target app.

### 4.3 HIERARCHICAL CODE SYNTHESIS

We use a multimodal code model (Kimi K2.5 (Moonshot AI, 2026)) as the synthesis backbone and address the complexity of full-stack development with a hierarchical strategy that combines *meta-prompting* (Suzgun & Kalai, 2024) for system design and a *plan-and-execute* (Zhou et al., 2022) workflow.

**Meta-prompting for system design.** We initialize synthesis with a high-level meta-instruction, where the model analyzes the input trace $\mathcal{T}$ and task description to generate a refined, task-specific system prompt. To mitigate common synthesis pitfalls, we enforce four strict design constraints via the meta-prompt:

1. **Viewport resolution:** Mandating specific viewport dimensions (e.g., $375 \times 812$) to enforce alignment with mobile interface standards.
2. **Visual fidelity:** Prescribing the use of modern CSS frameworks (e.g., Tailwind) to accurately emulate the native styling observed in the traces.
3. **Functional isolation:** Mocking external network requests with local database logic (e.g., SQLite) to create self-contained, offline-executable environments.
4. **Adversarial distractors:** Explicitly requiring the generation of distractor elements alongside target items to prevent trivializing retrieval tasks.

**Plan-and-execute implementation.** To reduce attention degradation in long-context generation, we separate file-structure planning from implementation. The model first generates a file manifest (e.g., `app.py` and `templates/index.html`), then implements each component sequentially through a multi-turn dialogue, allowing it to handle frontend fidelity and backend logic separately.

### 4.4 CODE-NATIVE REWARD INJECTION

A critical innovation of GUI-GENESIS is the embedding of verifiable rewards directly into the environment source code, addressing the sparse and noisy reward problem of real-world apps.

During the backend generation phase, we instruct the model to synthesize a deterministic reward function, implemented as `calculate_reward()`. This function inspects the application's backend state variables rather than relying on noisy visual observations. Consider the task "order a beef

burger without onions at 10". The synthesized function checks:

$$r = \alpha_1 \cdot \mathbb{I}(\text{cart.has("burger")}) + \alpha_2 \cdot \mathbb{I}(\neg\text{cart.has("onion")}) + \alpha_3 \cdot \mathbb{I}(\text{orderTime} = \text{"10:00"}) \quad (1)$$

where $\mathbb{I}(\cdot)$ is the indicator function and $\alpha_i \in [0, 1]$ are sub-goal weights with $\sum_i \alpha_i = 1$. An example can be found in Appendix B.

This approach transforms the reward signal from a noisy probabilistic estimation to a **deterministic executable assertion**. The signal is reproducible for a fixed state and implementation, while its semantic correctness depends on the generated reward logic.

### 4.5 Automated Self-Verification

To filter syntax errors, inconsistent state updates, and logic hallucinations, GUI-Genesis verifies each generated environment through static self-reflection and dynamic execution. A candidate that fails either stage is discarded and resampled; we make up to $K = 5$ attempts and accept the first candidate that passes both stages.

**Static Self-Reflection.** The model first reviews the generated code, focusing on reward logic. Because RL success conditions are often expressible as simple boolean predicates such as Equation 1, this stage filters apparent reward-logic inconsistencies before execution, independently of UI layout, event handling, and other application mechanisms.

**Dynamic Testing.** Because static correctness does not guarantee runtime feasibility, the model generates a companion Playwright script (Microsoft, 2025) that attempts the task in the synthesized environment. A candidate passes only if the script terminates without uncaught runtime errors and obtains a reward greater than 0, verifying that the widgets are interactable and backend state transitions trigger the reward function.

## 5 Experiment Setup

We design our experiments to evaluate the efficiency of training in GUI-Genesis-synthesized environments and their efficacy in facilitating sim-to-real transfer.

**Datasets and Tasks.** Our evaluation is conducted on a proprietary dataset focused on the *WeChat Mini-App* ecosystem (Tencent, 2024), one of the largest super apps serving over one billion active users. This platform represents a challenging testbed due to its dynamic mini-app structures and diverse service domains (e.g., retail, food delivery, and public services). We curated two strictly non-overlapping task sets to prevent data leakage: a *training set* containing 969 instructions across 411 unique mini-apps based on real-world tasks and a *evaluation set* containing 149 instructions across 112 unseen mini-apps, spanning more than 23 distinct usage categories.

**Baselines and Training Environments.** To isolate the impact of environmental fidelity on learning efficiency, we maintain a fixed agent architecture and vary the training source and reward mechanism. **1) Base Model (Lower Bound):** We adopt Qwen-2.5-VL-32B (Bai et al., 2025b), an open-source 32B-parameter multimodal vision-language model with strong GUI-grounding capabilities, as the base model. This checkpoint serves as the initialization for all subsequent RL pipelines. **2) Real-World Environment (VLM-Reward):** The agent is trained through interacting with live WeChat mini-apps. Since system-level states are inaccessible in production, we employ a commercial VLM as the reward model. This represents the current state-of-the-art framework but is constrained by high latency and inference costs. **3) GUI-Genesis Environment (VLM-Reward):** The agent trains within our synthesized, lightweight web environments but utilizes the same VLM as the real-world baseline for reward signals. **4) GUI-Genesis Environment (Code-Native Reward):** The agent trains within our synthesized environments utilizing deterministic, code-based execution feedback.

After training, we evaluate all policies *zero-shot* on the real WeChat mini-apps to measure sim-to-real transfer; agents trained on GUI-Genesis have no interaction with the real-world environment during RL post-training.

**Agent and RL Pipeline.** All experiments employ a unified VLM-based GUI agent executing a ReAct-style (Yao et al., 2022) strategy. We strictly control the training protocol across environments using **Multistep GRPO** (Group Relative Policy Optimization) (Shao et al., 2024) within the ROLL framework (Wang et al., 2025b). This control ensures that performance differences are attributable solely to environment quality and reward reliability.

**Evaluation Metrics** Our primary metric is the *Success Rate (SR)*, measuring the proportion of tasks successfully completed when deploying the trained agent into the live WeChat mini-app environment. We report SR verified through three distinct sources: *Human Annot. SR* (specialized human annotators), *VLM Eval SR* (commercial VLM judgment), and *Code-native SR* (executable assertions where a reward of 1.0 indicates task success). Human Annot. SR and VLM Eval SR are reported on real-world mini-apps, while VLM Eval SR and Code-native SR are reported on synthesized environments.

# 6 RESULTS

## 6.1 MAIN RESULTS

### 6.1.1 EFFICIENCY AND COSTS

We first compare the computational and financial costs of training in Table 1 in details. In terms of latency, executing a single interaction takes $10\times$ longer in a real-world environment than in a synthetic one. Taking model inference and other overhead in consideration, a complete rollout in a real-world environment takes an average of $0.22$ hours due to network instability and rendering overhead while our synthesized environments execute locally without remote-backend dependencies, reducing the per-rollout latency to just $0.10$ hours, with $2\times$ speedup.

Beyond time savings, GUI-GENESIS avoids VLM-as-judge rewards and uses inexpensive CPU-based simulation instead of costly real mobile devices. Verifying each rollout with a VLM costs $\$0.005$. To quantify the resource savings from replacing mobile devices with CPU-based web applications, we use the corresponding cloud-device rental prices from AWS Device Farm (Services, 2025b). Consider a standard RL epoch with $1,000$ environments and 12 rollouts per environment, yielding $12,000$ trajectories. As shown in Table 1, training in GUI-GENESIS-synthesized environments cuts aggregate rollout time by more than half and reduces recurring rollout and verification costs by $98.69\%$. Section 6.2.1 separately accounts for the one-time environment-preparation cost and the total first-epoch cost.

Table 1: Comparison of computational efficiency and equivalent rental costs in training. Costs are estimated for one RL epoch ($1,000$ envs, 12 rollouts per env).

| Metric | Real-World Env | GUI-GENESIS (Ours) |
|---|---|---|
| **Latency** | | |
| Time per Interaction (s) | 4.81 | $\mathbf{0.42}$ ($> 10\times$) |
| Time per Rollout (h) | 0.22 | $\mathbf{0.10}$ ($\approx 2.2\times$) |
| **Unit Costs** | | |
| Verification API Cost (per trajectory) | $\$0.005$ (VLM) | $\$\mathbf{0.00}$ (code-native) |
| Equivalent Rental Cost (per min) | $\$0.17$ | $\$\mathbf{0.005}$ ($-97.06\%$) |
| **Aggregate Rollout Time (h) (1 Epoch)** | 2726.4 | 1215.6 |
| **Rollout and Verification Cost (1 Epoch)** | $\$27{,}869.28$ | $\$\mathbf{364.68}$ ($-98.69\%$) |
| **One-Time Preparation Cost** | — | $\$2506.37$ |
| **Total Equivalent Cost (Preparation + 1 Epoch)** | $\$27{,}869.28$ | $\$\mathbf{2871.05}$ ($-90\%$) |

### 6.1.2 SIM-TO-REAL MODEL PERFORMANCE

As shown in Table 2, agents post-trained in GUI-GENESIS environments exhibit strong transfer performance when evaluated on real-world mini-apps. With the proposed code-native reward, the agent achieves a Human Annotation SR on real world environments of $42.28\%$, improving over the baseline model without reinforcement learning by $5.37$ absolute percentage points, corresponding to a relative gain of $14.54\%$. Surprisingly, when compared with agents trained directly in the real-world environment, which achieve $40.94\%$, our method obtains a slightly higher success rate in

Table 2: Main Results on Sim-to-Real Transfer. The synthetic/VLM-reward condition holds the reward source fixed when comparing real and synthetic training environments.

| Training Method | Real-World Mini-apps | | Synthetic Environments | |
|---|---|---|---|---|
| | **Human Annot. SR** | VLM Eval SR | VLM Eval SR | Code-native SR |
| **Base Model** | 36.91% | 52.34% | 63.76% | 38.93% |
| **Real-World Environment** | | | | |
| VLM reward | 40.94% | 59.73% | 63.76% | 44.30% |
| **Synthetic Environment** | | | | |
| VLM reward (Ours) | 41.61% | **61.74%** | 69.80% | 47.65% |
| Code-native reward (Ours) | **42.28%** | **61.74%** | **71.81%** | **48.99%** |

this evaluation setting, with a relative gain of $3.27\%$. A similar trend is observed under VLM-based evaluation, where the model trained in GUI-GENESIS environments also achieve competitive performance.

With VLM rewards fixed, synthetic-environment training achieves a real-world Human Annot. SR of $41.61\%$, compared with $40.94\%$ for real-environment training. Code-native rewards raise SR to $42.28\%$ while eliminating VLM-judge calls.

We interpret these results as evidence that synthesized environments can serve as an effective and efficient substitute for real-world training rather than as a general claim that simulation is inherently superior to real environments. One possible explanation for this performance gain is that real-world training introduces additional sources of variance, including network latency, application instability, and noise from VLM-based reward estimation, which may affect policy optimization. In contrast, GUI-GENESIS provides lower-latency execution and deterministic code-native rewards, leading to a more stable post-training signal while still enabling transfer to real-world applications.

#### 6.1.3 SCALING PROPERTIES OF TRAINING IN SYNTHESIZED ENVIRONMENTS

To investigate how the diversity of synthesized training environments affects agent generalization, we conduct a dataset-scaling analysis with quarter-scale ($N = 240$), half-scale ($N = 480$), and full-scale ($N = 969$) training subsets. We evaluate the post-trained agents on the 149 held-out synthesized environments described in Section 5 using both code-native SR and VLM Eval SR. Figure 5 shows a strong positive relationship between environment scale and agent performance, suggesting that broader synthetic experience helps agents acquire transferable interaction patterns. Since our framework allows for the low-cost, automated synthesis, we can further investigate this trend by scaling up the generation of synthetic environments.

### 6.2 ANALYSIS OF SYNTHESIZED ENVIRONMENTS

#### 6.2.1 SYNTHESIS COST

Constructing $969 + 149 = 1,118$ environments incurs approximately \$2.5K for trace collection, based on Qwen API charges and equivalent machine costs, and \$302.12 in synthesis API charges. We use Qwen-2.5-VL-32B for trace collection because the traces serve only as synthesis context. Together, these costs total approximately \$2.8K, or \$2.51 per environment, and constitute a *one-time fixed cost*. For the $1,000$-environment workload in Table 1, the normalized preparation cost is $(2500 + 302.12) \times 1000/1118 \approx \$2506.37$. Adding \$364.68 for one RL epoch yields approximately \$2871.05, about $90\%$ below the \$27,869.28 real-environment cost. Subsequent epochs reuse the environments without repeating trace collection or synthesis.

#### 6.2.2 SELF VERIFICATION PASS-RATE

Table 3: Distribution of generation attempts.

| **Attempt** | 1st | 2nd | 3rd | 4th | 5th | Fail |
|---|---|---|---|---|---|---|
| **Rate** | 27.73% | 22.09% | 14.67% | 9.93% | 8.23% | 17.35% |

GUI-GENESIS synthesizes environments that are faithful to the original applications and equipped with code-native reward oracles. As shown in Table 3, we generate environments for $1,118$ RL tasks,

of which $82.65\%$ pass self-verification; most successful environments are generated within the first two attempts. For tasks that do not pass self-verification, we retain the final iteration to align with the real-world training setting. Appendix I provides further analysis of these failures.

#### 6.2.3 Visual Fidelity of Synthesized Environments

To assess whether our synthesized environments preserve the visual and task-relevant structure of their real-world counterparts, we evaluated all models on the $149$ held-out synthetic evaluation environments generated during the setup phase (Section 5). As reported in the two synthetic-environment columns of Table 2, the relative ranking of training methods remains unchanged across evaluation settings. This consistency suggests that the synthesized environments retain task-relevant affordances, object configurations, and interaction cues, although this comparison is indirect evidence of fidelity.

We additionally compare the homepages of $100$ randomly sampled synthetic environments with their real counterparts using Kimi K2.5. The overall similarity score is $44.03/100$, compared with $99.87$ for same-app real-image pairs and $10.20$ for different-app pairs; $69\%$ of synthetic–real pairs are judged to depict the same app. This measures visual recognizability rather than functional equivalence; full scores are in Appendix G.

Using the same visual proxy on $20$ randomly sampled apps, removing trace grounding, meta-prompting, plan-and-execute, or self-verification reduces overall similarity from $45.95$ to $12.50$, $37.95$, $22.80$, or $36.40$, respectively. Appendix H reports the complete component ablation.

Crucially, we observe that standard VLM Eval SR exceeds human-annotated SR on real mini-apps and code-native SR on synthetic environments as shown in a case study in Appendix C. This case illustrates that VLM judges can yield false positives on incomplete tasks. Conversely, our code-native reward analyzes the underlying program state, providing a verification standard based directly on the implemented task conditions.

#### 6.2.4 Complexity of Synthesized Environment

To assess whether GUI-Genesis environments retain the interactive complexity of their real-world counterparts, we use the GUI agent from the trace collection pipeline to collect successful trajectories and compare their lengths. We exclude trajectories longer than 20 steps to reduce noise from cases where the agent becomes trapped in repetitive behavior and to focus on meaningful interactions.

Figure 2 shows the trajectory-length distributions after filtering. Synthesized environments have a spread comparable to that of real environments and require $5.63$ steps on average, indicating that they preserve nontrivial multi-step interaction depth and can support training for long-horizon planning.

#### 6.2.5 Asymmetric Capabilities in Coding and Navigation

We further investigate the relationship between the underlying code model's ability to *synthesize* the environment and the agent's ability to *navigate* it. As illustrated in Figure 3, we observe a distinct divergence across several generated cases. Although application synthesis is intuitively a more complex task than navigation, we note that the relationship between the two is not merely hierarchical in difficulty. Instead, the model exhibits asymmetric capabilities in these domains. Notably, there exist specific instances where the model successfully synthesizes a functional application but fails to navigate it. This intriguing finding suggests a potential pathway for self-evolutionary training, wherein a model could acquire novel navigation skills by exploring the very environments it generates. Evaluating such an iterative self-training process remains future work.

### 6.3 Analysis of Code Native Reward

To investigate the fidelity and granularity of our proposed code-native reward mechanism, we conducted a comparative analysis against a standard VLM-based reward model (VLM-as-Judge). We sampled $1,000$ instances classified as failure (score $0$) and $1,000$ instances classified as success (score $1$) by the VLM from our synthesized environments. We then computed the corresponding code-native rewards for these instances and present the cumulative distribution functions (CDF) of the code-native rewards for both groups are visualized in Figure 4. Our code-native reward demonstrates

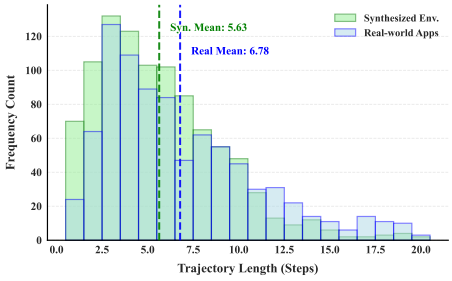


Figure 2: Successful trajectory lengths.

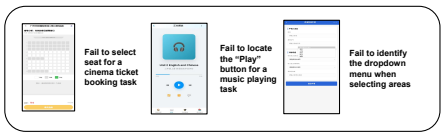


Figure 3: A coding–navigation mismatch.

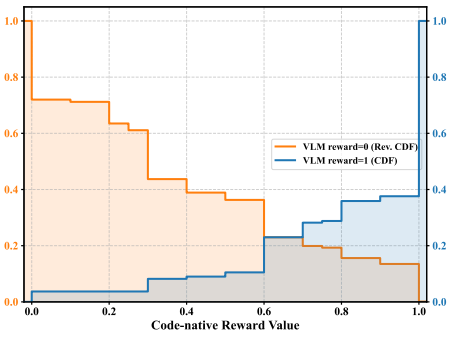


Figure 4: Code-native rewards for trajectories labeled by the VLM judge.

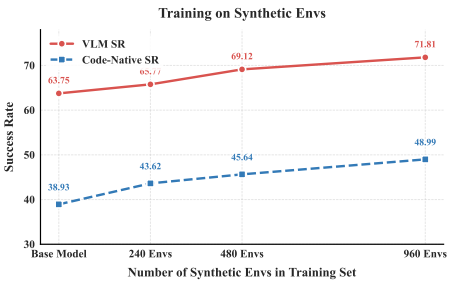


Figure 5: Scaling synthetic training environments yields continuous improvement.

a strong consistency with the visual semantic judgments of the VLM. For the group classified as *failures* by the VLM (orange curve), we observe that $75\%$ of the corresponding code-native rewards are concentrated in the range $[0, 0.6]$, consistent with a low completion status under the code-native reward. Conversely, for the group VLM-classified as *successes*, over $60\%$ of the code-native rewards exceed $0.8$, with a significant logical peak at $1.0$. This alignment suggests consistency between visual judgments and the underlying code logic, although agreement with a VLM judge is not an independent test of reward correctness.

## 7 Conclusion

In this paper, we have proposed GUI-Genesis, a framework that addressed the efficiency and verifiability bottlenecks of post-training GUI agents in real-world interactive environments by leveraging multimodal code model to reverse-engineer real mobile applications into lightweight web environments with verifiable code-native rewards. Experiments on 149 challenging WeChat mini-app tasks show that agents post-trained solely in synthesized environments transfer effectively to real-world applications, matching or slightly exceeding agents trained directly on target apps while reducing interaction latency by $10\times$ and cutting estimated recurring rollout and verification costs by $98.69\%$ per epoch. These results establish environment synthesis as a scalable and cost-effective paradigm for GUI agent post-training.

## REPRODUCIBILITY STATEMENT

We release the GUI-GENESIS synthesis pipeline, synthesized web environments, code-native reward assertions, and the curated training and evaluation task sets through our anonymous website. Appendix E documents the datasets, software dependencies, licenses, and artifact contents; Appendix A provides the prompts used throughout environment generation; and Section 5 reports the training protocol and evaluation setup. Together, these materials specify the environment-construction, verification, post-training, and evaluation procedures needed to reproduce our main results.

## ETHICS STATEMENT

The interaction traces used in this work were obtained through a partnership with the platform operator under appropriate data-use agreements. We retain only de-identified, task-relevant information, remove account identifiers from released instructions, and populate synthesized environments with mocked data rather than real user data, as detailed in Appendix E. Human evaluation was conducted by consenting professional QA annotators compensated at standard local industry rates (Appendix F). Because application synthesis can also facilitate deceptive replicas or phishing interfaces, the released artifacts are intended for research on agent training and evaluation and should be used with appropriate access controls, safety review, and human oversight.

## AI USE STATEMENT

In this work, generative AI was used in two ways. First, multimodal and code language models are components of GUI-GENESIS: they transform collected interaction traces into synthetic web environments, including application code and code-native reward logic. The generated environments and reward logic were checked using the verification procedure described in Section 4.5. Second, generative AI tools assisted with editing author-written passages for clarity and readability and with preparing and formatting figures. Generative AI was not used for research ideation, methodology or experiment design, retrieval or selection of related work, formulation of mathematical claims or proofs, or interpretation of experimental results. The authors reviewed all AI-assisted work, verified the final text, citations, figures, numerical values, and conclusions against the underlying experiment records, and take responsibility for all content and artifacts in this work.

## REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-vl technical report, 2025a. URL https://arxiv.org/abs/2511.21631.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025b.

Hyungjoo Chae, Jungsoo Park, and Alan Ritter. Safe and scalable web agent learning via recreated websites, 2026. URL https://arxiv.org/abs/2603.10505.

Wentong Chen, Junbo Cui, Jinyi Hu, Yujia Qin, Junjie Fang, Yue Zhao, Chongyi Wang, Jun Liu, Guirong Chen, Yupeng Huo, et al. Guicourse: From general vision language models to versatile GUI agents. *arXiv preprint arXiv:2406.11317*, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *NeurIPS*, 36:28091–28114, 2023.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for GUI agents. *arXiv preprint arXiv:2410.05243*, 2024.

Jihao Gu, Qihang Ai, Yingyao Wang, Pi Bu, Jingxuan Xing, Zekun Zhu, Wei Jiang, Ziming Wang, Yingxiu Zhao, Ming-Liang Zhang, et al. Mobile-r1: Towards interactive reinforcement learning for vlm-based mobile agent via task-level rewards. *arXiv preprint arXiv:2506.20332*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, and et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Haoran He, Yuxiao Ye, Qingpeng Cai, Chen Hu, Binxing Jiao, Daxin Jiang, and Ling Pan. Random policy valuation is enough for llm reasoning with verifiable rewards, 2025. URL https://arxiv.org/abs/2509.24981.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for GUI agents. In *CVPR*, pp. 14281–14290, 2024.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

Dacheng Li, Yunhao Fang, Yukang Chen, Shuo Yang, Shiyi Cao, Justin Wong, Michael Luo, Xiaolong Wang, Hongxu Yin, Joseph E Gonzalez, et al. Worldmodelbench: Judging video generation models as world models. *arXiv preprint arXiv:2502.20694*, 2025.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for gui visual agent. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 19498–19508, 2025.

Quanfeng Lu, Wenqi Shao, Zitao Liu, Fanqing Meng, Boxuan Li, Botong Chen, Siyuan Huang, Kaipeng Zhang, Yu Qiao, and Ping Luo. GUI odyssey: A comprehensive dataset for cross-app GUI navigation on mobile devices. *arXiv preprint arXiv:2406.08451*, 2024.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Guanjing Xiong, and Hongsheng Li. Ui-r1: Enhancing efficient action prediction of gui agents by reinforcement learning. *arXiv preprint arXiv:2503.21620*, 2025.

Dezhao Luo, Bohan Tang, Kang Li, Georgios Papoudakis, Jifei Song, Shaogang Gong, Jianye Hao, Jun Wang, and Kun Shao. Vimo: A generative visual gui world model for app agents. *arXiv preprint arXiv:2504.13936*, 2025.

Microsoft. Playwright. https://github.com/microsoft/playwright, 2025. Framework for Web Testing and Automation.

Moonshot AI. Kimi k2.5: Visual agentic intelligence. https://www.kimi.com/blog/kimi-k2-5.html, 2026. Accessed: 2026-01-29.

Pallets. Flask, 2024. URL https://flask.palletsprojects.com/. The Python micro framework for building web applications.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In *ICSE*, pp. 894–905, 2023.

Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. Guardian: A runtime framework for LLM-based UI exploration. In *ISSTA*, pp. 958–970, 2024.

Dezhi Ran, Yuan Cao, Mengzhou Wu, Simin Chen, Yuzhe Guo, Jun Ren, Zihe Song, Hao Yu, Jialei Wei, Linyi Li, Wei Yang, Baishakhi Ray, and Tao Xie. Appforge: From assistant to independent developer – are gpts ready for software development?, 2025a. URL https://arxiv.org/abs/2510.07740.

Dezhi Ran, Zihe Song, Wenyu Wang, Wei Yang, and Tao Xie. TaOPT: Tool-agnostic optimization of parallelized automated mobile UI testing. In *ASPLOS*, pp. 1251–1265, 2025b.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for Android device control. *NeurIPS*, 36:59708–59728, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William E Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *ICLR*, 2025.

Amazon Web Services. Aws device farm, 2025a. https://aws.amazon.com/device-farm/.

Amazon Web Services. Aws device farm pricing, 2025b. https://aws.amazon.com/cn/device-farm/pricing/.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Tencent. Weixin mini program platform capabilities, 2024. URL https://developers.weixin.qq.com/miniprogram/dev/platform-capabilities/miniapp/intro/. Accessed: 2026-01-29.

Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Weixun Wang, Shaopan Xiong, Gengru Chen, et al. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library, 2025b. URL https://arxiv.org/abs/2506.06122.

Dingbang Wu, Rui Hao, Haiyang Wang, Shuzhe Wu, Han Xiao, Zhenghong Li, Bojiang Zhou, Zheng Ju, Zichen Liu, Lue Fan, and Zhaoxiang Zhang. MobileGym: A verifiable and highly parallel simulation platform for mobile GUI agent research, 2026a. URL https://arxiv.org/abs/2605.26114.

Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, Liang Chen, Yuyao Zhai, Bang Liu, Chenglin Wu, and Yuyu Luo. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines, 2026b. URL https://arxiv.org/abs/2602.14296.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Wenyi Zhao, and Yuxiao Dong. Mobilerl: Online agentic reinforcement learning for mobile gui agents. *arXiv preprint arXiv:2509.18119*, 2025a.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous GUI interaction. In *ICML*, 2025b.

Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. Aria-ui: Visual grounding for gui instructions. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22418–22433, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Foundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pp. 240–255. Springer, 2024.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.

Charlie Zhang, Graham Neubig, and Xiang Yue. On the interplay of pre-training, mid-training, and rl on reasoning language models, 2025. URL https://arxiv.org/abs/2512.07783.

Jiwen Zhang, Jihao Wu, Yihua Teng, Minghui Liao, Nuo Xu, Xiao Xiao, Zhongyu Wei, and Duyu Tang. Android in the zoo: Chain-of-action-thought for GUI agents. *arXiv preprint arXiv:2403.02713*, 2024.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training, 2026. URL https://arxiv.org/abs/2601.04126.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *ICLR*, 2024.

# A PROMPT

We show the prompts used at different stages of GUI-GENESIS below. The Meta Prompt (Figure 6) is invoked at the meta-prompting stage to derive a task-specific system prompt from the input trace and instruction; Figure 7 shows a concrete instantiation of such a system prompt for a weather-query task. Conditioned on this system prompt, the Plan-and-Execute Prompt (Figure 8) decomposes generation into a file manifest followed by per-file implementation. Once the website is implemented, the test-script prompt (Figure 9) generates a Playwright test that verifies end-to-end task completion, and the Self-Reflection Prompt (Figure 10) audits the generated code with a focus on the reward logic.

**Figure 6: Meta Prompt**

```
You are an expert Prompt Engineer. Your task is to generate a high-
    quality "System Prompt" that will be sent to a Coding LLM (
    Multimodal). The Coding LLM will use your prompt (and attached
    images) to build a fully functional website.
{instruction}
In the prompt you generate, you must strictly require the following:
\#\#\# 1. Visual Alignment (High Priority)
The prompt must explicitly instruct the Coding LLM to handle visual
    requirements based on the availability of screenshots:
*   **Target Device**: Force the website to render specifically for a
    mobile resolution of **410x858**. Do NOT scale specifically for
    desktop viewing; maintain mobile layouts.
*   **Native App Simulation**: Design the interface to mimic a native
    mobile application (iOS/Android style) rather than a standard
    mobile webpage. Use app-like navigation (e.g., sticky bottom tab
    bars, top navigation bars with back buttons), rounded layouts, card
    -based lists, and touch-friendly interaction zones.
*   **Screenshot-trace-based Design**: The Coding LLM will also receive
     a trace of screenshots when finishing the task but the trace might
     be werong. Coding LLm donnot have to faithfully reproduce the
    screenshots but take them, especially the design style, for a
    reference.
\#\#\# 2. Technical \& Functional Stack
*   **Tech Stack**: The website must be built using **Python (Flask or
    FastAPI)** for the backend and **vanilla HTML/CSS/JS** for the
    frontend.
*   **Mock Backend**: All data persistence must be mocked in-memory (no
     external database requirements).
*   **UI/UX**: The frontend must be aesthetically pleasing, modern, and
     responsive.
\#\#\# 3. Data Diversity \& Distractors
*   **Input Generality**: For any text input fields, the implementation
     must allow free-text entry. Do not restrict inputs to the specific
     text seen in the screenshots or provide default recommendation.
*   **Distractor Options**: If the task involves selecting an item from
     a list (e.g., dropdowns, radio buttons, search results), **you
    must generate realistic "distractor" items** alongside the correct
    target item. Do not only implement the single correct option shown
    in the screenshot. Include at least 3-5 plausible but incorrect
    alternatives.
\#\#\# 4. Hidden RL Reward Mechanism
The generated prompt must insist on a rich, multi-layered "
    Reinforcement Learning Reward Signal" embedded within the backend:
*   **State Tracking**: The backend must maintain an internal state
    tracking the progress specifically for the `target\_task`.
*   **Action Explanation**: Explain every action with "ACTION\
    _EXPLANATION=xxx" printed to the server system output.
*   **Reward Output Format**: The output string must strictly follow: `
    RL\_REWARD=xxx, NEXT=xxx` printed to stdout (NOT displayed on the
    webpage).
*   **Sparse \& Multi-Step Rewards**:
    *   Do not just give a binary 0 or 1 reward. Distribute rewards
        across critical steps.
    *   `RL\_REWARD` must be a normalized value ($0.0$ to $1.0$)
        representing the cumulative completion status.
    *   Do not give reward at the first stage for successfully
        launching the app.
```

```
\#\#\# 5. Task-Specific Page Flow \& Termination
The prompt must instruct the Coding LLM to construct the final page
    logic based on the task type:
*   **Ordering/booking/recharging**: The final page must be an **Order
    Confirmation** or **Payment Confirmation** page.
*   **Information Query**: The final page must clearly display the **
    Query Results**.
*   **Date-related tasks**: The final page must show the **Date
    Selection** interface.
*   **Routing/Start-End tasks** (e.g., transit, navigation): The final
    page must visualize the **Start Point, End Point, and Direction**.
*   **Terminal State**: Once the final page is reached, the flow stops.
     **Do NOT** automatically restart or redirect. **Do NOT** show a
    generic "Task Completed" popup; simply stay on the relevant
    confirmation/result UI.
**Based on these requirements, please generate the final, cohesive
    System Prompt that we will send to the Multimodal Coding LLM.**
% \end{promptlisting}

    % \caption{A specialized prompt designed for reasoning tasks.}
    % \label{pmt:example}
% \end{figure*}
```

**Figure 7: Meta Prompt response, used to generate code directly**

```
## **Role & Objective**

You are an elite Full-Stack Developer and UI/UX Designer. Your
    objective is to build a fully functional, high-fidelity mobile web
    application based on the user's specific request and provided
    reference screenshots.
**User Task**: (Check the weather in Lvliang City for tomorrow).
**App Type**: Weather Information Application.
You must generate a complete project structure using **Python (Flask)**
     for the backend and **Vanilla HTML/CSS/JS** for the frontend. The
    application must run immediately via `python app.py`.

## **1. Visual & UI/UX Requirements (Strict Enforcement)**
*   **Target Resolution**: You are building specifically for a mobile
    viewport of **410px width x 858px height**.
    *   *Constraint*: Do not optimize for desktop. Ensure all CSS uses
        flexible units suitable for this specific mobile ratio.
*   **Language**: All text, labels, and UI elements must be in **
    Chinese (Simplified)**.
*   **Native App Simulation**:
    *   The web interface must feel like a native iOS/Android app, not
        a standard website.
    *   Use a fixed bottom navigation bar (sticky), a top app bar with
        back navigation (where applicable), and card-based layouts with
         rounded corners (border-radius: 12px+).
    *   Ensure touch targets are large (minimum 44x44px).
*   **Design Autonomy & Screenshots**:
    *   Use the provided screenshots as a **style reference** (color
        palette, font hierarchy, icon style).
    *   **However**, if the screenshots contradict the Specific User
        Task (e.g., showing a shopping app when the user asks for
        weather), prioritize the **User Task** for layout and
        functionality.
    *   Self-infer a modern, clean weather dashboard layout if
        screenshots are unclear.

## **2. Technical Stack & Implementation**
*   **Backend**: Python (Flask).
*   **Frontend**: HTML5, CSS3, JavaScript (No external frameworks like
    React/Vue; use Jinja2 templates or inline HTML strings).
*   **Data Persistence**: Use **In-Memory** storage (global Python
    dictionaries/lists). Do not use SQL or external databases.

## **3. Data Logic, Inputs & Distractors**
*   **Free-Text Input**: The search bar for city selection must allow
    **free-text entry**. Do not hardcode the input to "Lvliang". Allow
    the user to type.
```

```
*   **Realistic Distractors**:
    *   When displaying search results, history, or "Nearby Cities",
        you must include **3-5 plausible distractor options** (e.g.,
        Taiyuan, Datong, Beijing, Xi'an) alongside the correct target (
        Lvliang).
    *   Do not present "Lvliang" as the only option.

## **4. Reinforcement Learning (RL) Reward Mechanism**
You must embed a logic tracking system in the backend to evaluate the
    user's progress toward the goal: "Check weather in Lvliang for
    tomorrow."
**Output Requirements**:
*   Every time the user performs a significant action (clicks, searches
    , page loads), the backend must print two lines to `stdout`:
    1.  `ACTION\_EXPLANATION= [Brief description of what happened]`
    2.  `RL\_REWARD=[0.0 to 1.0], NEXT=[Next logical step]`
**Reward Scoring Logic**:
*   **0.0**: App launch / Home page load. (Do not reward simple startup
    ).
*   **0.3**: User successfully inputs (Lvliang) into the search field
    OR selects "Lvliang" from a list of cities.
*   **0.6**: User navigates to the specific Detail Page for "Lvliang".
*   **1.0**: User selects/toggles the date to "Tomorrow" or the view
    specifically confirms "Lvliang Weather: Tomorrow" is visible.

## **5. Page Flow & Termination**
1.  **Home Page**: Search bar or City List.
2.  **Interaction**: User searches for "Lvliang City".
3.  **Result/Detail Page**: Displays weather details.
    *   Since this is an **Information Query** task, the flow **
        terminates** at the Result Page.
    *   The final view must be the Weather Detail interface showing the
         forecast for Lvliang.
```

**Figure 8: Plan-and-Execute Prompt after meta-prompting**

```
Given above instruction, output a list of file paths necessary to
    build this project.
Rules:
1. Include files like app.py, index.html, styles.css, script.js, and
     specific feature pages.
2. Do NOT include image assets (jpg, png) as you cannot generate
    them. Use placeholders in code later.
3. Return ONLY a JSON object with a single key "files" containing an
     array of strings. We will ask you tio fill in these files later
    .

{Meta Prompt response,}

Now output code for the file {file\_path}.
Return ONLY the raw code content for this file. No markdown
    backticks at the start/end.
```

**Figure 9: Testscrpt generation prompt**

```
Finally, generate a javascript test script to test your website with
     playwright in python. Use sys.exit(1) to return testing error.
Return ONLY the raw code content for this file. No markdown
    backticks at the start/end.'
```

**Figure 10: Self-Reflection Prompt**

```
{
    'app.py': ...,
    'templates/index.html': ...,
}
Here is my code for accomplishing following requirements {
    instruction}; is my implementation right or not? please answer
    only with "yes" or "no".
You can focus on: is the reward rule same to the requirement? can
    website work normally for interaction?
```

## B GENERATION EXAMPLES

We show some code snippets that demonstrate how the generated code fulfills our design constraints in a specific case, "to purchase an iPhone 15 Pro (256GB) at a price below 4500". The viewport-related CSS variables (Figure 11) and the body styles (Figure 12) reproduce the prescribed mobile resolution and native-app aesthetic. The reward computation snippet (Figure 13) reads internal state variables and prints a deterministic, sub-goal-decomposed signal. The mocked product database (Figure 14) populates the catalog with realistic distractors alongside the target item, and the Playwright test script (Figure 15) verifies end-to-end task completion in the synthesized environment.

**Figure 11: How generated code use specefied resolution**

```
:root {
    --primary-blue: #4a90e2;
    --text-main: #333;
    --text-secondary: #888;
    --bg-light: #f7f8fa;
    --card-bg: #ffffff;
    --border-color: #eee;
    --app-width: 410px;
    --app-height: 858px;
}
```

**Figure 12: How generated code imitate mobile app GUI**

```
body {
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI",
        Roboto, "Helvetica Neue", Arial, sans-serif;
    background-color: #2c3e50; /* Background for desktop view */
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    color: var(--text-main);
...
```

**Figure 13: How generated code calculate code-native rewards**

```
 if product_id == 1:
        state.current_step = 2
        print("ACTION\_EXPLANATION=User viewing product id=1", flush
            =True)
        print("RL_REWARD=0.5, NEXT=ORDER PRODUCT", flush=True)
```

**Figure 14: How generated code mock database interactions**

```
# Mock Database
PRODUCTS = [
    {
        "id": 1,
        "title": "iPhone 15 Pro 256G Natural Titanium",
        "model": "iPhone 15 Pro",
        "storage": "256G",
        "price": 4299,
        "condition": "95% New",
        "battery": 91,
        "repair": "No repairs",
        "desc": "First-hand personal use, all original with no
            disassembly or repairs, 91% battery, excellent condition
            .",
        "is_target": True
    },
    {
        "id": 2,
        "title": "iPhone 14 Pro 256G Space Black",
        "model": "iPhone 14 Pro",
        "storage": "256G",
        "price": 3800,
        "condition": "95% New",
        "battery": 88,
        "repair": "No repairs",
        "desc": "Very good condition, normal signs of use.",
        "is_target": False
    },
    {
        "id": 3,
        "title": "iPhone 15 Pro 128G Blue Titanium",
        "model": "iPhone 15 Pro",
        "storage": "128G",
        "price": 3999,
        "condition": "99% New",
        "battery": 95,
        "repair": "No repairs",
        "desc": "Storage capacity is not enough for me, selling at a
             low price.",
        "is_target": False
    },
    {
        "id": 4,
        "title": "iPhone 15 Pro 256G White Titanium",
        "model": "iPhone 15 Pro",
        "storage": "256G",
        "price": 5500,
        "condition": "99% New",
        "battery": 100,
        "repair": "No repairs",
        "desc": "Almost brand new, all accessories included.",
        "is_target": False
    },
    {
        "id": 5,
        "title": "iPhone 15 Pro 256G Black Titanium",
        "model": "iPhone 15 Pro",
        "storage": "256G",
        "price": 3500,
        "condition": "90% New",
        "battery": 79,
```

```
        "repair": "No repairs",
        "desc": "Scratches on the screen, low battery health,
            selling cheap.",
        "is_target": False
    }
]
```

**Figure 15: How generated code write testscripts**

```
import sys
import os
import time
from playwright.sync_api import sync_playwright

def run_test():
    port = os.environ.get("PORT", 5000)
    url = f"http://0.0.0.0:{port}"

    with sync_playwright() as p:
        # Launch browser with the specified mobile dimensions
        browser = p.chromium.launch(headless=True)

        context = browser.new_context(
            viewport={'width': 410, 'height': 858},
            user_agent="Mozilla/5.0 (iPhone; CPU iPhone OS 14_7_1
                like Mac OS X) AppleWebKit/605.1.15 (KHTML, like
                Gecko) Version/14.1.2 Mobile/15E148 Safari/604.1"
        )

        page = context.new_page()

        try:
            # 1. Load the Application
            print(f"Navigating to {url}...")
            page.goto(url)
            page.wait_for_selector("#product-search-input", timeout
                =5000)

            # 2. Search for "iPhone 15 Pro"
            print("Action: Searching for 'iPhone 15 Pro'...")
            search_input = page.locator("#product-search-input")
            search_input.fill("iPhone 15 Pro")
            search_input.press("Enter")

            ...
        except Exception as e:
            print(f"TEST ERROR: {str(e)}")
            sys.exit(1)

if __name__ == "__main__":
    run_test()
```

## C SUPPLEMENTARY CASE STUDIES

In Figure 16, booking a ride with intermediate stops in a ride-hailing app illustrates a failure case for the VLM judge, which cannot recognize that the destination and intermediate stop are reversed, while a code-native approach successfully identifies the issue by simply checking two variables. Our anonymous website shows additional case studies and demonstrates that our synthesized environment has strong fidelity.




Figure 16: Case: booking a ride with intermediate stops on a ride-hailing app

## D Anonymous Website

We release our code for environment generation and the full synthesized dataset on anonymous website `https://anonymous.4open.science/r/gui_genesis-B23B/`, which also incorporates a detailed analysis of mini-app distribution (`https://anonymous.4open.science/r/gui_genesis-B23B/synthesis/distribution.md`) and more case studies of synthesized mini-app (`https://anonymous.4open.science/r/gui_genesis-B23B/case_study/overview.md`).

## E Scientific Artifacts

**Artifacts Used.** We use Qwen-2.5-VL-32B (released under its respective open-weights license), the ROLL training framework, and standard open-source software (PyTorch, Transformers, Playwright). We use these artifacts strictly for research purposes consistent with their licenses and intended use. WeChat mini-app interaction traces are obtained through a partnership with the platform operator under appropriate data-use agreements; only de-identified, task-related screenshots and DOM snapshots are used.

**Existing Packages.** We rely on publicly available packages, including PyTorch, Transformers, ROLL (Wang et al., 2025b), and Playwright (for in-browser environment execution and code-native reward verification).

**Artifacts Created.** We release (i) the GUI-Genesis synthesis pipeline, (ii) the synthesized lightweight web environments and associated code-native reward assertions, and (iii) the curated 969-instruction training and 149-instruction evaluation task sets on an anonymous website `https://anonymous.4open.science/r/gui_genesis-B23B/`. We will release them under a permissive research license alongside documentation describing the schema, dependencies, and example invocations.

**PII and Sensitive Content.** The training and evaluation tasks describe generic mini-app interactions (e.g., querying weather, ordering, ride-hailing) and do not contain personally identifiable information. The synthesized environments use mocked in-memory data with realistic-looking but fabricated values (e.g., placeholder city/store names). No real user data is embedded. We manually audited the released task instructions to remove any incidental references to real account identifiers.

**Documentation and Statistics.** The dataset comprises 969 training and 149 evaluation instructions over 411 and 112 unique mini-apps respectively, spanning more than 23 service categories (retail, food delivery, public services, etc.). Languages covered are Chinese (the primary language of the WeChat ecosystem) and English (a smaller subset and instruction templates).

## F HUMAN ANNOTATION

**Annotator Recruitment.** Sim-to-real success rates reported as *Human Annot. SR* are produced by professional QA annotators recruited through the platform operator and compensated at standard local industry rates; annotators were informed of the research use of the labels and consented in writing.

**Annotation Protocol.** Annotators were given the natural-language task instruction together with the recorded final-state screenshot(s) of the agent's trajectory and asked to provide a binary judgment (success/failure) of whether the task was completed. Cases of ambiguity were resolved by majority vote among at least two annotators. Annotators were not asked to annotate any subjective attributes (e.g., demographic inferences); we therefore do not collect or report demographic information about the agents in the screenshots.

**Annotator Demographics.** Annotators are based in mainland China and are native Mandarin speakers, which matches the primary language of the WeChat mini-app evaluation set. No additional demographic data was collected.

## G Visual Similarity Evaluation

To measure visual reconstruction directly, we randomly sample 100 synthesized environments and pair their homepages with those of the corresponding real mini-apps. Kimi K2.5 rates each pair along layout, components, text, color, and imagery, and assigns an overall similarity score, all on a 0–100 scale. It also makes a binary judgment of whether the two screenshots depict the same app. We contextualize these scores using two reference settings evaluated by the same judge: distinct homepage screenshots from the same real app, and 100 pairs of different real apps. Table 4 reports the complete results.

Table 4: VLM-rated homepage similarity. The synthesized–real setting uses 100 sampled environments, and the different-app reference uses 100 pairs. Visual scores are on a 0–100 scale; Same-app is the fraction of pairs judged to depict the same app.

| Pair type | Layout | Components | Text | Color | Imagery | Overall | Same-app |
|---|---|---|---|---|---|---|---|
| Synthetic–real | 50.13 | 44.38 | 55.19 | 54.62 | 30.81 | 44.03 | 69% |
| Real–real (same app) | 99.88 | 99.86 | 99.95 | 99.91 | 99.93 | 99.87 | 100% |
| Real–real (different apps) | 17.06 | 17.56 | 4.41 | 13.85 | 8.97 | 10.20 | 4% |

The synthesized homepages score above the different-app reference across all dimensions, but remain well below the same-app reference. Imagery has the lowest similarity score (30.81), consistent with the absence of original app-specific image assets in the synthesis pipeline. The 69% same-app rate measures visual recognizability under this judge, not functional correctness. Because Kimi K2.5 is also used in synthesis, these scores are a model-based diagnostic rather than an independent human assessment. Homepage comparisons do not establish equivalence of state transitions, task outcomes, or recovery paths; those aspects require separate evaluation.

## H Component Ablations

We ablate four components of environment construction on 20 randomly sampled apps: trace grounding, meta-prompting, plan-and-execute synthesis, and self-verification. We use the homepage-similarity protocol described in Appendix G to assess each setting. This 20-app study is separate from the 100-environment analysis above, so its full-pipeline scores should be compared with the ablated settings in Table 5.

Table 5: Component ablations on 20 sampled apps. Scores measure homepage visual similarity under the same VLM-based protocol, not downstream RL success. Same-app denotes the judge's binary app-identity decision aggregated over the sample.

| Setting | Layout | Components | Text | Color | Imagery | Overall | Same-app |
|---|---|---|---|---|---|---|---|
| Full pipeline | 47.95 | 46.90 | 58.25 | 50.30 | 34.20 | 45.95 | 65% |
| w/o trace grounding | 13.75 | 15.50 | 12.00 | 20.00 | 8.25 | 12.50 | 5% |
| w/o meta-prompting | 45.20 | 30.15 | 40.10 | 32.80 | 38.65 | 37.95 | 50% |
| w/o plan-and-execute | 17.30 | 32.60 | 15.85 | 34.10 | 25.75 | 22.80 | 20% |
| w/o self-verification | 34.50 | 45.80 | 45.20 | 40.15 | 25.10 | 36.40 | 55% |

Trace grounding has the largest effect on overall similarity and same-app recognition in this sample. The remaining components also improve overall similarity relative to their ablated settings, although the effect varies across visual dimensions. These results characterize reconstruction quality on a small sample and do not isolate each component's effect on RL performance. Adversarial distractors address the availability of alternative choices rather than homepage appearance, and are illustrated qualitatively in Appendix B. The effect of the reward source on agent performance is evaluated separately in Table 2.

## I Audit of Unverified Environments

To understand why candidates remain unverified, we manually inspect 10 cases that did not pass self-verification within the five-attempt budget. Table 6 separates failures of the checking procedure from defects in the generated environments.

Table 6: Manual audit of 10 candidates that remained unverified after five attempts. Counts describe the inspected cases and are not estimates of the failure proportions in the full dataset.

| Failure source | Count |
|---|---|
| Static-stage false negative | 4 |
| Dynamic-test script fault | 3 |
| Environment defect | 3 |
| Total inspected | 10 |

The audit shows that an unverified candidate need not be an unusable environment: the static checker may reject a functional candidate, and the generated dynamic test may itself fail. The sample is too small to establish how often these causes occur across all unverified candidates. Our training setup retains the final candidate for each task when the budget is exhausted, rather than dropping tasks according to verification status. The effect of filtering these candidates from RL training remains unevaluated.

Dynamic verification requires the generated test script to terminate without uncaught runtime errors and obtain a reward greater than $0$. Positive reward may reflect partial progress, so this check does not certify full task completion.