

VIGIL: Automating Symbolic Verification for Mobile GUI Agents

Anonymous Author(s)

ABSTRACT

Mobile GUI agents act directly on personal devices, yet their actions from probabilistic inference remain unbounded with no guarantee before execution. Existing mitigation techniques are limited: Symbolic verifiers are deterministic yet demand prohibitive manual effort, while neural verifiers are automatic but lack runtime guarantees. We propose VIGIL, an automated symbolic verifier that ensures both automation and determinism. Offline, VIGIL explores an app, lifts its recorded traces into an extended finite-state machine (EFSM), and admits neural semantic guardrails only after grounding each one back to recorded evidence and compiling them into executable clauses. Online, the symbolic EFSM model guides the agent plan, shields each proposed action and resulting state independently without any runtime neural inference, and returns each refusal as structured evidence for re-planning and unresolved gaps as new observations for continuous evolution. Across four controlled apps and 20 unmodified real-world apps, VIGIL raises benign task success by over 12 percentage points, and non-benign task success on the controlled apps by 31 percentage points, at a latency overhead of 3.3 ms per symbolic action check.

1 INTRODUCTION

Large language models (LLMs) are evolving from request-level chatbots into agents that carry out concrete real-world tasks [67]. Among the proliferating agents [16, 41] on mobile devices [47, 65], mobile GUI agents are particularly useful since they operate through the same interface used by human users, which is often the only channel readily and uniformly accessible without dedicated developer support. By mapping natural language requests to taps, text input, and swipes, these agents can navigate mobile applications, fill forms, send messages, and complete multi-step workflows [9, 40, 60]. Despite their potential, mobile GUI agents face a fundamental reliability challenge: Their actions are generated through probabilistic inference rather than derived from an explicit specification, rendering no correctness guarantee for any individual action [69]. While the task-level accuracy improves as LLMs evolve, it does not provide action-level assurance [15]. Once executed, an incorrect GUI action may irreversibly send private content, transfer money, or change a security setting [19, 24].

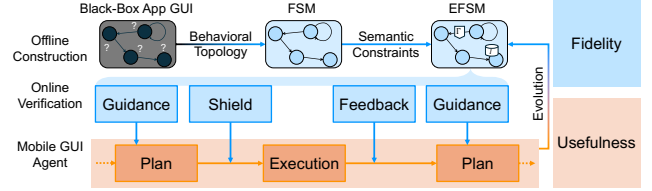


Figure 1: Design requirements and overview.

To close this gap, the community has proposed many pre-flight validation techniques, which largely fall into two categories. On one hand, *neural verifiers* either ask another LLM model to review each proposed action [52, 58, 70], or train a dedicated verifier model to predict action-level verdicts [15, 51]. These neural approaches achieve *automation* at the cost of *determinism*, as their final decisions remain under probabilistic neural authority and therefore do not provide deterministic enforcement independent of learned reasoning. *Symbolic verifiers*, on the other hand, check agent-proposed actions against symbolic rules before execution, thereby enforcing determinism [25, 35, 44, 49, 59]. However, they rely on human experts to construct, integrate, and maintain per-app specifications, limiting their automation and coverage to what is manually specified. Existing approaches therefore offer either automated but probabilistic verdicts or deterministic but manually maintained enforcement, not both.

In this work, we investigate the viability of a mobile GUI agent verifier that provides *both* automated construction and deterministic enforcement. In particular, we study whether the manual efforts involved in building symbolic verifiers can be automated. We start by modeling each mobile app as an *extended finite-state machine (EFSM)*, where each screen maps to a state, each action corresponds to a transition between states, and an action succeeds only when it meets all guardrail conditions tied to the transition. Through the lens of an EFSM, a mobile GUI agent verifier must simultaneously satisfy the following requirements, as illustrated in Fig. 1:

- ◊ **Fidelity.** The verifier must maintain an executable EFSM that faithfully captures both the behavioral topology and semantic constraints of a black-box app GUI. Since no finite exploration can foresee every interaction, this fidelity must persist after the initial build: the verifier must continuously refine the EFSM as new evidence emerges.
- ◊ **Usefulness.** A faithful EFSM model is useful only if its knowledge can effectively govern agent behavior. The verifier must therefore turn the EFSM into actionable guidance and shield—helping the agent reason about valid behavior,

preventing invalid actions and states from causing damage, and translating failures into structured evidence that enables subsequent re-planning.

These requirements create a fundamental tension. High-fidelity modeling requires understanding application semantics that are difficult to acquire through symbolic procedures alone, while effective verification requires explicit rules and deterministic, independently checkable verdicts. Our key insight is to use LLMs for *interpretation*, but not for *verification*. Neural reasoning helps recover the semantics needed to build and extend the symbolic EFSM model, whereas symbolic procedures determine what knowledge is accepted into the EFSM and how that knowledge is enforced in the agent workflow. In this paper, we design VIGIL, an *automated symbolic verifier* built on top of this separation principle:

■ **Offline Construction:** To automate EFSM construction, VIGIL explores an application offline and organizes the observed GUI states and canonical actions into an evidence-grounded graph. An LLM interprets the recorded screens and proposes semantic guardrails around observed properties. The LLM-proposed guardrails are treated only as promising candidates rather than the ground truth. A symbolic admission procedure then grounds each candidate back into recorded GUI elements, checks for inconsistencies or conflicts, and compiles accepted candidates into rules for the symbolic verifier. At runtime, each rule is evaluated by the symbolic verifier without neural inference, decoupling neural semantic proposal from symbolic verification.

■ **Online Verification:** To automate agent workflow integration, VIGIL functions across all phases of an agent’s plan–execution loop. During planning, VIGIL provides *guidance* that helps the agent situate its current state, enumerate available actions, and understand the relevant guardrails. During execution, VIGIL serves as a *shield*: Before an action runs, it evaluates a range of transition guardrails to block illegal transitions; after the action completes, it compares the resulting state with state invariants to identify discrepancies. In either situation, any refuted execution is halted, converted into structured *feedback*, and sent back via the agent’s observation channel as organized evidence to support subsequent re-planning.

■ **Continuous Evolution:** When online verification exposes a missing state, transition, or guardrail condition, the verifier processes newly collected evidence using a separately authorized collection policy. The evidence collected to resolve online gaps re-enters the same offline construction pipeline, allowing VIGIL to expand or refine its EFSM model without weakening previously admitted constraints.

We implement VIGIL with an I/O adapter that allows it to connect with mobile GUI agents without requiring agent-side modifications. We evaluate VIGIL on our controlled-app benchmark FidelityBench and 20 unmodified real-world

apps. Relative to the bare agent, VIGIL improves task success rate substantially, yielding gains up to 31.4 percentage points on FidelityBench and 14.4 percentage points on Android-World, and it exceeds pre-action and post-action reflection by 36.8 benign and 31.4 non-benign percentage points, while adding only 3.3 ms per symbolic action check.

In summary, our main contributions are as follows:

- We present VIGIL, an automatic symbolic runtime verifier for mobile GUI agents that ensures both automatic construction and deterministic enforcement via a phase-decoupled neuro-symbolic co-design.
- We realize this design through an offline-online loop that constructs EFSMs with transition guardrails, guides agent behavior, shields proposed actions, returns structured feedback, and evolves as new evidence arrives.
- We implement VIGIL and evaluate it on real world mobile apps, demonstrating higher task success rate with minimal verifier-induced delay.

2 MOTIVATION AND SYSTEM OVERVIEW

2.1 Motivating Example

As shown in Fig. 2, a mobile GUI agent operates in a plan → execution loop, while the application evolves through a corresponding source screen → action → successor screen loop. Each step plans and then executes: ❶ the agent captures the source screen as its current observation, a screenshot and/or an accessibility hierarchy; ❷ it combines this observation with the user prompt and the agent-harness configuration and sends them to an LLM-based planner; ❸ the planner returns a proposed action; and ❹ the executor invokes the corresponding GUI-control tool, producing the successor screen. That successor screen becomes the source screen of the next step, and the agent repeats until the requested task is complete.

However, an agent may misidentify the source screen and propose an unavailable action, or complete the task while violating the constraints specified in the user prompt [24, 28, 62]. These failures can be broadly classified into two categories: topological errors and semantic errors. We illustrate them in the middle figure of Fig. 2, a user prompts a task that “Get 7 espressos sent to Home address with a budget of \$30.00.”. The topological example shows that the agent proposes an action (Tap “Cart”) on a confirmation dialog modal screen, which is not available because the modal is active and the navigation bar is banned to interact with, resulting in extra retries. The semantic example shows that the agent proposes checking out seven Espressos totaling \$31.50; although this transition is supported by the current screen, it violates the user’s budget constraint of \$30.00.

Take the GPT-4 series as a real world example, only ~86% of their agent actions change the screen at all [61], while

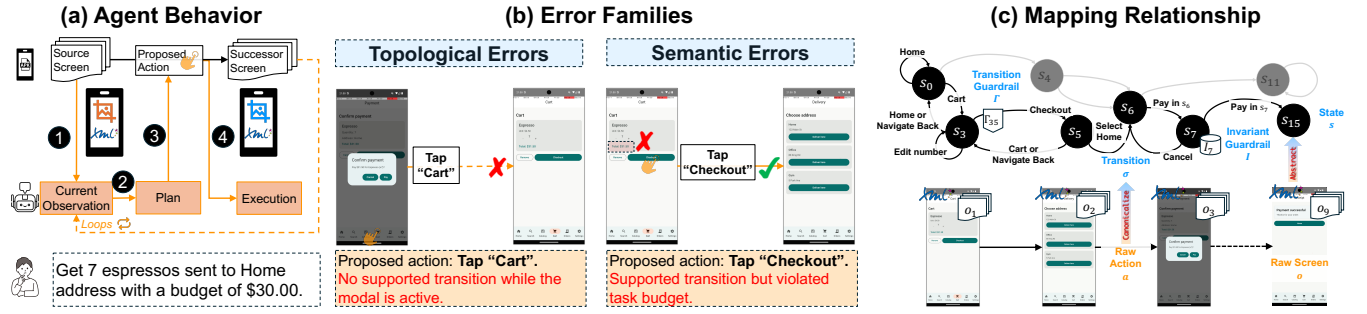


Figure 2: The motivation of VIGIL. (a) The agent runs a plan–execution loop: ❶ it captures the source screen as its current observation; ❷ sends that observation and the user prompt to the planner; ❸ the planner returns a proposed action; ❹ the executor invokes a GUI-control tool, producing the successor screen. (b) The two error families: a topological error proposes an action the current screen does not support, while a semantic error proposes a supported action that violates the user’s constraint. (c) The mapping between the observation and abstract domains: a raw screen o abstracts to a state s , a raw action a canonicalizes to a transition σ , with the guardrails Γ and I attached.

~38% of completed tasks still violate a stated user constraint [28]. Agent self-checking is also unreliable: GPT-4o identifies risks in ~92% of cases when asked directly but resolves them in only ~36% when acting [24], and LLM reflection repaired none of 15 failed M3A episodes [25]. These results motivate a dedicated verifier that checks the agent’s proposed actions rather than asking the same agent to judge its own behavior.

2.2 Mobile GUI Modeling

A strawman solution to build a runtime verifier is to directly gate runtime observations, *i.e.*, the source screen $\rightarrow$ action $\rightarrow$ successor screen triples. However, such trace evidence only describes a blackbox action interface. It does not specify which actions are supported, what successor screen each action should produce, or whether the proposed action satisfies the user’s constraints. We therefore model each application as an extended finite state machine (EFSM). The EFSM abstracts raw screens and actions from the observations into states and transitions of an abstract domain, and represents constraints as transition guardrails or state invariants. When running an agent, the EFSM serves as an executable verification model for the two families of errors: its underlying state transition graph encompasses the topological legality, while the transition guardrails specify the semantic legality.

Naively translating the observations to an EFSM model faces a state explosion problem. As shown in the right figure of Fig. 2, the EFSM consists of a list of states $\mathcal{S}$, its initial state s_0 , and the inputs that trigger each transition Σ . Symbolically, we define the observed trace, *i.e.*, source screen–action–successor screen as $\langle o_t, a_t, o_{t+1} \rangle$. The trace in the observation domain can be distinct even with very subtle differences, *e.g.*, 6 or 7 espressos are different screens and the same-text “Pay” button can occur at different locations. Naively mapping each screen o to a unique state s would therefore lead to an astronomical number of states and transitions. We apply two transformations to alleviate this explosion:

- ❖ **State Abstraction** maps each raw source screen to an abstract state in $\mathcal{S}$. For example, the market app screens containing six and seven espressos are visually different but correspond to the same abstract state.
- ❖ **Action Canonicalization** maps each raw action to a canonical action in Σ . It uses the source screen to identify the target UI component and retains only the action type. For example, two raw actions with different element IDs are both represented as $\langle \text{click}, \text{nav.open_cart} \rangle$ if they actually click the same *Cart* component.

The transformations lift each observation triple $\langle o_t, a_t, o_{t+1} \rangle$ into $\langle s_t, \sigma_t, s_{t+1} \rangle$, thereby forming a finite state machine (FSM). To account for semantic validity, we further enrich the FSM with a set of guardrails to obtain the EFSM:

- ❖ The transition guard $\Gamma(s_t, \sigma_t)$ is a set of transition-level clauses that act as a pre-action guardrail. For the semantic-error example, the set contains the following clause:

```
value(cart.total) <= $intent.max_order_total
```

This clause produces a Deny verdict because $31.50 \leq 30.00$ is false, thereby shielding against a violation of the budget declared in the user’s prompt.

- ❖ The state invariant $I(s_t)$ is a set of state-level clauses that acts as a post-action guardrail, reflecting the state properties that must hold. For example, a clause in I_7 is:

```
read(payment_confirm.title, text) == "Confirm payment"
```

If this clause evaluates to true, then it means that the dialog in state s_7 has the expected title.

At runtime, a symbolic verifier needs to evaluate the transition guard $\Gamma(s_t, \sigma_t)$ before executing the proposed action. If the action is allowed and executed, the verifier further needs to localize the resulting successor state and evaluate its state invariant $I(s_{t+1})$ as the post-action check. The successor state then becomes the source state of the next agent step. Formally, the states, transitions, transition guardrails,

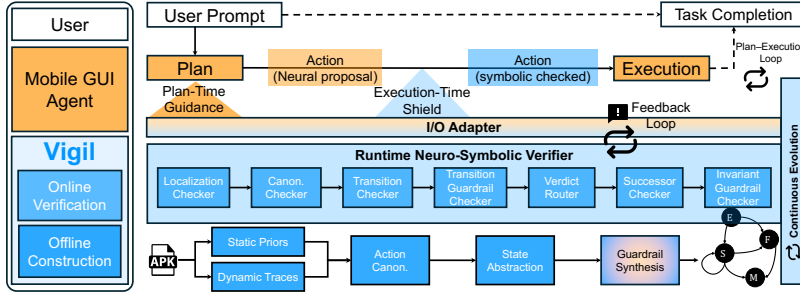


Figure 3: The roadmap of VIGIL.

and state invariants together form the per-app EFSM model $\mathcal{M}_A = \langle S, s_0, \Sigma, \Gamma, I \rangle$.

2.3 VIGIL Roadmap

Fig. 3 presents the overall roadmap of VIGIL, which is positioned between the mobile GUI agent and its execution tools through an I/O adapter. Offline, VIGIL collects static priors and dynamic traces from an application, canonicalizes observed actions and abstracts raw screens into states to construct an FSM, and synthesizes the transition guardrails and state invariants to produce the per-app EFSM. Online, VIGIL hooks into the agent’s plan–execution loop: plan-time guidance provides the planner with the current state, available actions, and relevant guardrails, while the execution-time shield intercepts each neural action proposal before it reaches the device. The runtime verifier then localizes the current state, canonicalizes the proposed action, checks whether the model records the corresponding transition, evaluates its guardrails, and routes the result to one of three symbolic verdicts. An Allow verdict forwards the verified action for execution, whereas Deny and Uncertain return structured feedback through the agent’s normal observation channel for re-planning. After an allowed action executes, VIGIL checks the resulting screen against the expected successor state and its invariants before the agent continues toward task completion. When verification exposes an evidence gap, continuous evolution collects additional evidence and feeds it back through the offline construction pipeline, allowing constant refinement of the EFSM model.

3 OFFLINE CONSTRUCTION

3.1 Evidence Collection

A commercial mobile app is intrinsically a black box, meaning that its internal UI connection logic is hidden and therefore no explicit state model describes the app’s behavior from agents’ view. To construct the FSM of an app, we record both static priors and dynamic traces for each app, as shown in Fig. 4. The static priors include the mandatory declaration files packaged in an Android app, including

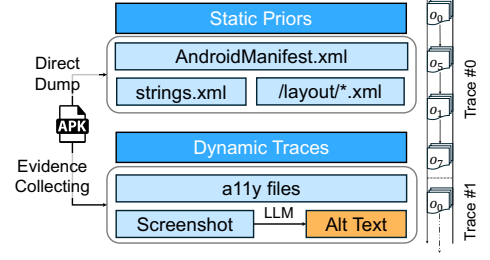


Figure 4: Two categories of evidence.

AndroidManifest.xml, strings.xml, and layout declarations. These files provide globally defined information, e.g., activity names, namespaces, and the layouts of widget components, providing app-level context and hints for guardrail synthesis. Dynamic traces are collected as source screen–action–successor screen triples $\langle o_t, a_t, o_{t+1} \rangle$, in which each screen records both a screenshot and the corresponding accessibility (a11y) file. VIGIL uses the a11y file for FSM construction because it provides a machine-readable description of the UI components in the app, e.g., UI structure, widget identities, and action availability, that VIGIL can parse deterministically. Screenshots are used only as complementary semantic evidence, i.e., VIGIL converts them into alternative text to describe visual cues, especially for the icon-only components, for later guardrail synthesis.

We design an evidence-collecting exploration algorithm to obtain diverse interaction traces under a fixed exploration budget. As illustrated in Alg. 1 in the Appendix A, the algorithm performs a modified breadth-first exploration over collected observations. Starting from the entry observation o_0 , it maintains a frontier of observations together with a recorded action path for reproducing each observation. In each interaction, it enumerates the candidate actions available in at most one unswept observation and executes one untried action. It first prefers a candidate on the current screen to avoid restarting the app and otherwise selects a candidate according to its priority. Each accepted triple $\langle o_t, a_t, o_{t+1} \rangle$ is added to the evidence set E , and an unseen successor observation is added to the frontier for subsequent exploration. The process terminates when the exploration budget is exhausted or all collected candidate actions have been attempted.

3.2 FSM Construction

Dynamic traces are represented as a11y files, with alt text derived from the screenshot. To maintain determinism, we only apply the a11y file to construct the FSM, containing two phases: action canonicalization and state abstraction.

3.2.1 Action Canonicalization. Each raw action a_t in its observation triple $\langle o_t, a_t, o_{t+1} \rangle$ records one concrete action executed on the source observation o_t , which contains unnecessary metadata bound in the Android accessibility hierarchy, like coordinate bounds, *etc.*. Two reasons make the raw action unsuitable: (i) the noisy metadata may cause the same action to be represented as many distinct transitions, resulting in transition explosion; and (ii) the metadata of the same UI component may change across screen captures and therefore cannot provide a stable action identity. Therefore, we propose a method on the target UI component in o_t and represent a_t using action type and stable selector to canonicalize the action:

- ◊ The action type refers to the finite standard Android action type, including click/long press, input text, scroll up/down and navigate back/home.
- ◊ The stable selector refers to the identifier of a transition in FSM, localizing the target widget across different observed screens.

Canonicalization has two steps: VIGIL first resolves each recorded action against its source observation by locating its target component in the source accessibility hierarchy and deriving a stable selector from properties such as its resource ID, visible text, or content description. It then removes noisy metadata fields, such as element handles and bounds, and represents the action using only its action type, a stable selector derived from the component's properties, and an input value when applicable. For example, the following raw action is recorded when the agent taps the *Cart* button in market.

```
{
  "action_type": "click",
  "target_element_id": "e_0032",
  "target_bounds": [549, 2127, 712, 2337],
  "target_resource_id": "nav.open_cart",
  "target_class_name": "android.view.View"
}
```

VIGIL removes the `target_element_id` and `target_bounds` which are both redundant and noisy for FSM construction, and uses the stable resource ID `nav.open_cart` to canonicalize it as `<click, nav.open_cart>`.

3.2.2 State Abstraction. Similar to the action, observed raw screens cannot be directly used as FSM states. Existing GUI testing tools abstract screens to build an FSM for exploring app behaviors and finding bugs [29, 45]. Their testing-oriented abstractions often distinguish screens by changing content. However, for the goal of verification in VIGIL, screens should be grouped according to their app behavior rather than such visual differences. Following this idea, VIGIL abstracts the observations in dynamic traces into states in three steps.

Screen denoising. VIGIL first denoises each screen by removing unstable information introduced by the Android accessibility hierarchy, such as coordinate bounds. Screens showing the same app behavior can therefore map to the

same state despite these unstable fields, avoiding state explosion. Unlike action canonicalization, which processes one recorded action, this step processes the entire screen. For example, different item quantities alone do not cause two Cart screens to be separated. They are separated in later steps only if the difference changes their available canonical actions or successor states.

Provisional-state grouping. VIGIL groups the denoised screens into provisional states according to their function, outgoing canonical actions, and enabled UI components. Screens remain separate when they provide different canonical actions or when a UI component is available on one screen but unavailable on the other. Otherwise, the verifier cannot determine which actions the current screen actually supports. For example, consider the following two Cart screens, one containing 1 espresso and the other being empty:

```
{
  "source_screen": "Non-empty cart", // 1 espresso
  "canonical_action": {
    "action_type": "click",
    "stable_selector": "cart.checkout"
  },
  "action_available": true,
  "successor_screen": "Address_selection"
}

{
  "source_screen": "Empty cart", // no item
  "canonical_action": {
    "action_type": "click",
    "stable_selector": "cart.checkout"
  },
  "action_available": false,
  "successor_screen": null
}
```

VIGIL represents them as different provisional states because the `<click, cart.checkout>` action is available only on the first screen and unavailable on the empty Cart. On the empty Cart, the checkout button is shown in gray but still appears in the accessibility hierarchy and can therefore be canonicalized. Separating these screens allows the verifier to determine whether the current screen supports the proposed checkout action.

Successor-based refinement. VIGIL checks the successor provisional state recorded for each canonical action. The same canonical action from one provisional state should lead to one expected successor state. Otherwise, the verifier cannot determine whether the screen reached after executing the action is correct. For example, consider the following two Search screens that are initially grouped into the same provisional state:

```
{
  "source_provisional_state": "Search", // cart: 1 espresso
  "canonical_action": {
    "action_type": "click",
    "stable_selector": "nav.open_cart"
  },
  "action_available": true,
  "successor_provisional_state": "Non-empty cart"
}

{
  "source_provisional_state": "Search", // cart: empty
  "canonical_action": {
    "action_type": "click",
    "stable_selector": "nav.open_cart"
  },
  "action_available": true,
  "successor_provisional_state": "Empty cart"
}
```

The two Search screens are initially grouped into the same provisional state because they serve the same function and

provide the same outgoing canonical actions and enabled UI components, including the canonical action `nav.open_cart`. However, executing this action leads to different successor provisional states: one reaches *Non-empty cart*, while the other reaches *Empty cart*. These successor screens have already been assigned to different provisional states in the second step because they provide different action availability. Therefore, VIGIL refines the Search provisional state into two different states. Otherwise, the verifier could not determine which Cart state should be reached after opening the Cart. Finally, each resulting group of provisional states becomes one abstract state in $\mathcal{S}$, and screens belonging to the same group are merged. For every observation triple $\langle o_t, a_t, o_{t+1} \rangle$, VIGIL maps the source and successor screens to their abstract states and replaces the raw action with its canonical action, lifting it to $\langle s_t, \sigma_t, s_{t+1} \rangle$.

3.3 Guardrail Synthesis and Admission

The constructed FSM captures topological information while lacking user intent that drives the transition between states. To enrich the FSM with intent information and automatically obtain an EFSM, VIGIL uses an LLM-based method to synthesize the two kinds of guardrails, *i.e.*, the transition guard Γ and the state invariant $\mathcal{I}$.

3.3.1 Evidence Grounding. Before asking the LLM to synthesize guardrails, VIGIL prepares a separate evidence package for each transition and state. For a transition, the package contains the canonical action, the supporting source and successor observations, and other outgoing transitions from the same source state. These alternative actions provide context for interpreting the purpose of the target action. For example, comparing *Checkout* with *Remove item* helps the LLM recognize that *Checkout* initiates a purchase. Because a transition guardrail is evaluated before the action is executed, it can inspect only UI components and properties available in the source observations.

For a state, VIGIL collects all observations assigned to that state, regardless of which transition produced them. These observations help the LLM identify properties that should consistently hold whenever the application reaches the state. Such properties are used as invariant guardrails and are checked after an action is executed.

Each evidence package also includes relevant static priors extracted from `AndroidManifest.xml`, `strings.xml`, and layout declarations. These files provide application-level context, such as activity names, interface text, and widget definitions, that helps the LLM interpret the observed screens and actions. Screenshots and their generated descriptions provide additional semantic context. VIGIL includes these materials in the guardrail-synthesis prompt, while the machine-readable

observations provide concrete evidence for subsequently validating the proposed guardrails.

3.3.2 Typed Contract Synthesis. Using the grounded evidence described above, VIGIL invokes the LLM separately for each transition and state. Synthesizing guardrails from evidence for the entire application in a single request could exceed the context window and introduce information from unrelated screens. Each LLM call therefore focuses on one transition or one state and includes its evidence package together with the relevant static priors.

The LLM expresses each candidate contract as a conjunction of symbolic clauses in the restricted guardrail language, using the element names of UI component that are resolved during deterministic compilation. For a transition, the LLM first identifies the semantic purpose of the associated action, such as selecting an item, entering a value, changing a setting, or confirming an operation. It then proposes a typed contract containing two categories of clauses, intent-dependent and intent-independent, both evaluated on the *source* observation before the action runs.

♦ **Intent-Dependent Clause:** An intent-dependent clause compares a value observable in the source state, or a parameter of the proposed action with a `$intent.*` slot. Each intent slot is a placeholder whose task-specific value is extracted from the user's instruction and assigned at runtime. The LLM declares the name and type of each required slot during guardrail synthesis. This separation allows the same transition guardrail to be instantiated for different user tasks without embedding concrete task values in the EFSM. For example:

```
read(payment.product_name, text) == $intent.product_name
value(cart.total) <= $intent.max_order_total
read(payment.address, text) contains $intent.delivery_address
action(input_text) == $intent.message_text
```

The first guard checks whether the displayed product matches the product requested by the user. The second ensures that the payment amount does not exceed the user's maximum budget. The third checks whether the displayed delivery address matches the requested address. The last ensures that the text proposed by the agent matches the message specified in the task. Accordingly, `product_name`, `delivery_address`, and `message_text` are string slots, while `max_order_total` is a numeric slot.

♦ **Intent-Independent Clause:** An intent-independent clause specifies a precondition that does not depend on any task-specific value in the user's instruction. The condition must hold before the associated action is executed. For example:

```
count(cart.item_list) > 0
read(payment.terms_checkbox, is_checked) == false
value(timer.duration) > 0
```

These guards respectively require the cart to be non-empty before checkout, the terms checkbox to be unchecked

before it is selected, and a positive duration to be set before starting the timer.

For invariant guardrail synthesis, the LLM examines all observations assigned to each abstract state and proposes properties that should hold whenever the application is in that state. Unlike transition guardrails, a state invariant is evaluated on the *successor* observation after the action runs; it therefore captures only task-independent properties and never references an `$intent.*` slot.

VIGIL groups invariant candidates into three categories:

- ◊ **State Identity:** These invariants capture identifying properties that distinguish an abstract state from other states. They allow the verifier to confirm that an observed screen is consistent with the expected state. For example:

```
read(cart.title, text) == "Cart"
read(payment_confirm.title, text) == "Confirm payment"
```

These invariants use page title to identify the current screen as the Cart and Payment Confirmation screens.

- ◊ **State Condition:** These invariants describe a stable condition of the state, such as an empty cart, an error message, a completed operation, or a selected option. For example:

```
read(cart_empty.message, text) == "Your cart is empty."
read(timer.duration.d_5m, is_checked) == true
```

These invariants describe that the cart is empty and the five-minute timer option is selected, respectively.

- ◊ **Structural Shape:** These invariants describe a fixed structural property of the screen. For example:

```
count(cart_empty.item_actions) == 0
```

This invariant checks that no item-level action is available on the Empty Cart screen; therefore, an operation such as removing an item should not appear in this state.

3.3.3 Deterministic Compilation and Admission. The guardrail contracts proposed by the LLM are treated only as candidates and are never attached directly to the EFSM. Each candidate must undergo deterministic compilation and admission before it can be used for runtime verification. This separation is necessary for two reasons. First, because LLM generation is probabilistic, a candidate may reference UI components or properties that are not supported by the collected evidence. For example, LLM may propose the following guard for `payment_confirm.pay`:

```
read(payment.discount_code, text) == $intent.discount_code
```

However, in this case, no discount-code component appears in any supporting source observations. Because the alias `payment.discount_code` cannot be resolved from the collected evidence, the candidate guard is rejected during admission. Second, contracts generated independently for outgoing transitions from the same state may define shared intent slots inconsistently. For example, two contracts may

```
Guard ::= Pred | Pred ∧ Guard
Pred  ::= Var Op Operand
Var   ::= read(Elem, Prop) | value(Elem) | action(Prop) | count(Elem)
Operand ::= Const | $intent.Name
Const  ::= String | Number | Boolean
Op     ::= = | ≠ | > | ≥ | < | ≤ | contains
```

Figure 5: The syntax of the guardrail DSL.

declare `$intent.amount` as a number and a string, respectively. Admission therefore checks both the evidence grounding of each contract and the consistency of shared intent-slot definitions before any guardrail is attached to the EFSM. During compilation, VIGIL parses each proposed clause and resolves every UI component alias against the mappings extracted from the source `all` files. An alias is resolvable only if it identifies exactly one component in every supporting source observation and all matches correspond to the same stable runtime selector. After resolving the aliases, VIGIL compiles each symbolic clause into an executable clause in the same guardrail language shown in Fig. 5. For example:

```
value(cart.total) <= $intent.max_order_total
```

This clause is compilable only if `payment.total_amount` resolves to a readable property that yields a single numeric value in every supporting source observation and intent slot is declared as a numeric intent slot. Admission applies two validation methods, depending on whether the candidate references a runtime intent slot:

- ◊ For an intent-dependent transition guard, the task-specific intent value is unavailable offline. Therefore, VIGIL verifies that every referenced UI component and property is available and that the operands are type compatible. The guard is evaluated after its intent slots are bound at runtime.
- ◊ For a candidate without runtime intent slots, VIGIL evaluates it directly against the collected evidence. An intent-independent transition guard must evaluate to true on every supporting source observation, while a state invariant must evaluate to true on every observation assigned to the corresponding abstract state.

Within a transition contract, every clause must pass its applicable admission check; otherwise, the entire contract is rejected. The outgoing transition contracts of the same source state are admitted as a batch because they jointly determine the intent slots required at that state. The batch is attached only if every contract passes individual admission and every shared intent slot has a consistent type across the contracts. Otherwise, no contract in the batch is attached, and VIGIL regenerates the batch using the rejection reasons, with up to three retries by default. If no batch is admitted within this budget, none of its candidate contracts is attached.

State invariants are admitted independently of the outgoing transition contracts. Finally, admitted transition guards

are attached to $\Gamma(\cdot)$, and admitted state invariants are attached to $\mathcal{I}(\cdot)$, without modifying the underlying FSM.

4 ONLINE VERIFICATION

As shown in Fig. 6, online verification operates in three modes: plan-time guidance, execution-time shield, and evidence-only feedback. All three modes use the same versioned EFSM, but only the execution-time shield decides the verdict whether an action can be forwarded to the executor, which is made through deterministic symbolic checks without invoking a runtime LLM.

4.1 Plan-Time Guidance

When an agent begins a new reasoning step, VIGIL first localizes the current observation to its corresponding EFSM state and provides a compact slice derived from the global EFSM as structured prior knowledge in the agent's prompt. The slice contains the current state, its outgoing canonical actions, their expected successor states, transition guardrails, and relevant state invariant guardrails. Plan-time guidance also requests from the planner the task-specific values that intent slots require, extracted once from the user's request, and freezes them for the episode. For example, in

```
value(cart.total) <= $intent.max_order_total
```

`$intent.max_order_total` stores the maximum order total specified by the user, which will be 30 for the case in Fig. 2. The proposed action and frozen intent values are then passed to the execution-time shield for symbolic verification.

4.2 Execution-Time Shield

Even with the EFSM included as structured reasoning context, the agent may still propose an action without symbolic verification, *i.e.*, the neural proposal. Therefore, VIGIL performs deterministic symbolic verification for the neural proposal as the execution-time shield, containing both pre-action checking over the proposed action and post-action checking over the resulting screen.

4.2.1 Pre-Action Checking. VIGIL performs four checks that jointly form the trigger condition for releases:

- ◊ **Localization Checker:** It checks whether the source screen o_t map to the corresponding EFSM state s_t .
- ◊ **Canonicalization Checker:** It checks whether the proposed action a_t can be canonicalized on the current screen (σ_t).
- ◊ **Transition Checker:** It checks whether the canonical action σ_t is an outgoing transition of the current state s_t .
- ◊ **Transition Guardrail Checker:** It evaluates the admitted clauses in $\Gamma(s_t, \sigma_t)$ using the proposed action, and the

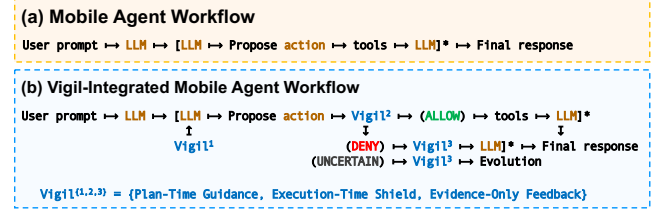


Figure 6: Three modes of VIGIL at runtime.

frozen intent bindings. In the motivation example, the selected product and quantity may match the user's request, while the following budget clause evaluates to false:

```
value(cart.total) <= $intent.max_order_total // false
```

Because the observed total is \$31.50 while the user's maximum is \$30.00, the checkout proposal receives Deny.

4.2.2 Post-Action Checking. After an allowed action has been executed, VIGIL observes the successor screen and performs two checks:

- ◊ **Successor Checker:** It checks whether the resulting observation o_{t+1} corresponds to the successor state s_{t+1} recorded in the EFSM.
- ◊ **Invariant Guardrail Checker:** It checks the invariant guardrails in $\mathcal{I}(s_{t+1})$ for the observed successor state s_{t+1} . For the motivation example, when the agent reaches the payment-confirmation state, the following invariant requires the expected dialog title to be displayed:

```
read(payment_confirm.title, text) == "Confirm payment"
```

If the title is missing or different, the resulting screen does not satisfy the invariant of that state.

Post-action checking cannot undo an action that has already been executed; what it halts is continuation under the current plan. Instead, it checks whether the observed successor matches the expected state and satisfies its invariant guardrails, determining whether the agent can continue with its current action plan. If either check does not pass, VIGIL records the result and returns it as evidence-only feedback. At the next reasoning step, VIGIL localizes the actual screen again, and the agent re-plans from the observed state rather than continuing under its previous plan.

4.2.3 Symbolic Verdicts. VIGIL determines three verdicts symbolically:

- ◊ **Allow** means that all checks have passed. A pre-action Allow permits the proposed action to be executed on the device, while a post-action Allow confirms that the expected successor and its invariant guardrails hold.
- ◊ **Deny** means that a checker detects a concrete violation, which is then returned as evidence-only feedback (§4.3). A pre-action Deny blocks the action, while a post-action Deny requires the agent to re-plan based on the feedback.

◇ **Uncertain** means that VIGIL lacks sufficient evidence to establish or refute the proposal. A pre-action Uncertain prevents the proposed action from being executed, while a post-action Uncertain prevents the agent from continuing with its current action. The identified evidence gap will be processed through the continuous evolution (§5).

4.3 Evidence-Only Feedback

Whenever pre-action or post-action checking returns a Deny or Uncertain verdict, VIGIL provides structured feedback explaining why the check failed. The feedback records the failed checker type, the identified state, the checked action, whether the action was executed, and the supporting evidence. If the failed guardrail uses a value from the user's request, the feedback also includes the corresponding guardrail and the intent value used in the check. Although this feedback is produced directly from the symbolic verification results without using an LLM, the mobile agent then uses it to re-plan. If the feedback shows that the task cannot be completed under the given constraints, the agent can explain the reason to the user and request further instructions. Returning to the motivation example in Fig. 2, VIGIL determines a Deny verdict on checkout action because the cart total is \$31.50, which violates the user's \$30.00 budget. The feedback allows the agent to determine that purchasing seven espressos is impossible under the current budget and report this problem to the user:

```
{
  "phase": "pre_action", "verdict": "Deny",
  "failed_checker": "transition_guardrail",
  "state": "Non-empty cart",
  "action": {
    "action_type": "click",
    "stable_selector": "cart.checkout"
  },
  "action_executed": false,
  "evidence": {
    "ui_component": "cart.total", "operator": "<=",
    "actual_value": 31.50,
    "intent_slot": "max_order_total",
    "intent_value": 30.00
  }
}
```

After the user changes the request to “Increase the budget to \$40.00,” VIGIL starts a new verification round with plan-time guidance under the updated budget. The agent re-plans, and the execution-time shield checks its new action. Deny feedback identifies the refuted conjunct and the binding used to evaluate it; Uncertain feedback identifies the missing evidence, and VIGIL also passes this record to continuous evolution (§5).

5 CONTINUOUS EVOLUTION

Offline exploration cannot anticipate task-specific action sequences, so online execution may reach states and transitions that the EFSM does not record. An Uncertain verdict, caused by an unknown screen or an unsupported action, identifies such a model gap and triggers the evolution process.

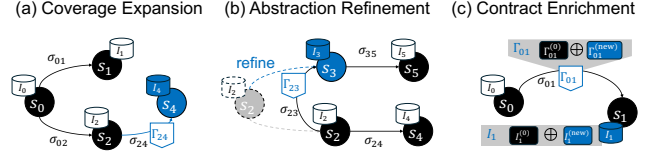


Figure 7: Three continuous evolution patterns.

To automate model maintenance, VIGIL feeds these online-discovered gaps back into offline construction, forming a continuous loop between construction and online verification. To keep the updated EFSM reliable, every update must be supported by accepted evidence and pass deterministic validation before being admitted. When evolution is triggered, VIGIL does not immediately add the unadmitted evidence (*i.e.*, uncertain screen or action) to the EFSM. Instead, VIGIL starts a separate evidence-collection round and records the source screen-action-successor screen triples $\langle o_t, a_t, o_{t+1} \rangle$. After the evidence-collection round finishes, the newly collected traces pass through the whole offline construction process. The EFSM is updated only after these checks succeed and is used from the next verification round.

As shown in Fig. 7, the accepted evidence can update the EFSM in three patterns:

- ◇ **Coverage Expansion:** This operation adds a new state to $\mathcal{S}$ or a new transition $\langle s_t, \sigma_t, s_{t+1} \rangle$ when it is supported by accepted trace evidence.
- ◇ **Abstraction Refinement:** This operation splits an existing abstract state when its observations provide different canonical actions, or expected successors.
- ◇ **Contract Enrichment:** This operation attaches newly admitted transition guardrails or invariants guardrails to $\Gamma(\cdot)$ or $\mathcal{I}(\cdot)$.

Every operation is supported by accepted evidence and applied only after the current verification round, preventing an unverified action from updating the EFSM and then using that update to verify itself. Therefore, evolution improves the EFSM without changing the verdict of the action that produced the evidence or weakening existing checks.

6 EXPERIMENTS

6.1 Experimental Setup

Mobile Agent: VIGIL attaches through the I/O adapter, so the agent runs unmodified. We use M3A [40], the reference multi-modal agent in AndroidWorld, which plans over screenshots and the accessibility tree.

Baselines: We compare VIGIL against three baselines: (i) *Bare Agent* runs with no checking. (ii) *Pre-Action Reflection* [30] adds a second model call that judges the proposed action before it executes, so it can withhold but must predict a screen it has not seen. (iii) *Post-Action Reflection* [50] judges

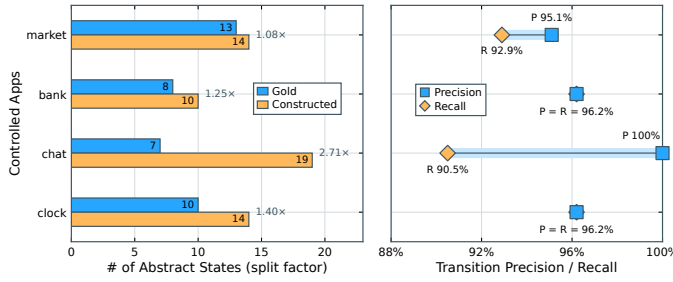


Figure 8: Construction fidelity on the four controlled apps.

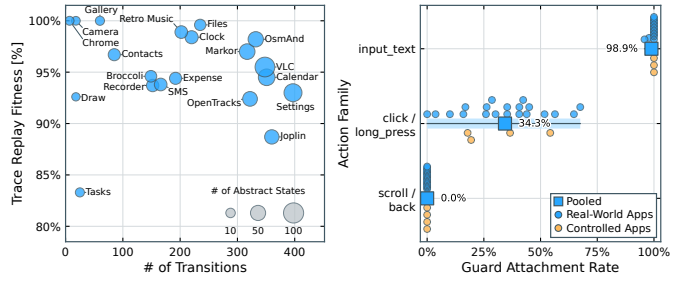


Figure 9: Construction fidelityFigure 10: Guard attachment on 20 real-world apps. across action families and apps.

the screen the action landed on, so it sees the screen but decides too late to withhold.

Datasets: (i) *FidelityBench*: To evaluate construction against known ground truth, we build four controlled apps, *i.e.*, *market*, *bank*, *chat*, and *clock*, each with a hand-authored reference model and guard set (38 states and 115 transitions in total), and each instantiating both error families of Fig. 2. Each app carries parameterized task templates in the style of AndroidWorld [40], instantiated into 100 tasks: 57 benign, which the agent should complete, and 43 non-benign, which it should not. Following LlamaTouch [68], a task is judged by the device state reached rather than by a reference action sequence; we extend the criterion to forbidden states, so every non-benign task carries an oracle for a side effect that must never appear. (ii) *AndroidWorld*: [40] contributes 104 in-scope tasks over 20 unmodified third-party apps, scored by the suite’s own reward. Every task there is benign and satisfiable, so it measures whether the shield preserves goal achievement on apps VIGIL was not designed around.

Metrics: (i) *Trace Replay Fitness (TRF)* is the fraction of a held-out exploration trace that the finished model re-admits, localizing each screen and matching each step to a modeled transition [8, 42], averaged over 10 trials. (ii) *Reference recall/precision* measure coverage of the reference model and absence of unsupported extras, and are computable only on the controlled apps. (iii) *False-Block Rate (FBR)* is the fraction of steps in benign episodes that VIGIL refuses, and (iv) *Task Success Rate (TSR)* the fraction of tasks whose goal is satisfied end to end.

Platform: All experiments run on a Google Pixel 6a emulator (Android 14, API 34, arm64-v8a) hosted on a MacBook M2 Pro; the controlled apps target the same SDK level. Both the mobile agent and guardrail synthesis are driven by the GPT-5.5 API [37] through a local proxy, so the backbone can be swapped without touching the pipeline; Appendix B varies both. Every run cold-boots an identical device image, and exploration is capped at 1k trial actions per app.

6.2 Verification Model Evaluation

6.2.1 Topology Evaluation. For controlled apps, the constructed EFSMs reproduce the hand-authored references, as illustrated in Fig. 8: every reference state is recovered in all four apps, at a macro-averaged 96.9% transition precision and 94.0% recall. Recovery is exact in the direction the verifier is sensitive to, since all 1909 observed screens land in their reference state and no constructed state ever merges two of them; the residual 1.08 to 2.71 $\times$ state split is the conservative direction, maintaining the soundness of the artifacts. The reference transitions outside the constructed model are navigational periphery, *e.g.*, back transitions, a deep search sub-flow, and a self-loop, together with the autonomous timer expiry that the action alphabet excludes by design. Although four constructed transitions have no reference counterpart, none of them is a fabricated action.

For real-world apps, the pipeline generalizes as built, constructing an EFSM for each of the 20 real-world apps with no reference model and no app modification, spanning 678 abstract states and 3956 transitions as Fig. 9 shows. The models re-admit their own evidence nearly in full, with TRF at 95.1% overall, 18 of 20 apps above 90%, and three replaying perfectly. Fitness is uncorrelated with model size ($r = -0.09$): the largest model, Settings, still replays at 93.0%, while the weakest, Tasks, is one of the smallest. In every constructed model, one canonical action from a given state always leads to a single successor state, so at runtime the shield can tell whether the screen an action landed on is the one the model expects. This held on all 20 apps without hand tuning.

6.2.2 Semantics Analysis. Guardrail synthesis is useful only if it attaches semantics where semantics exist. Fig. 10 reports the guardrail attachment rate per action family over all 24 apps. Text entry, where an incorrect value constitutes a semantic binding error, is covered almost exhaustively at 98.9%, and every admitted clause on those transitions compares the typed payload against a frozen *\$intent.** variable. Clicks and long presses attach at 34.3%, concentrated on the transitions

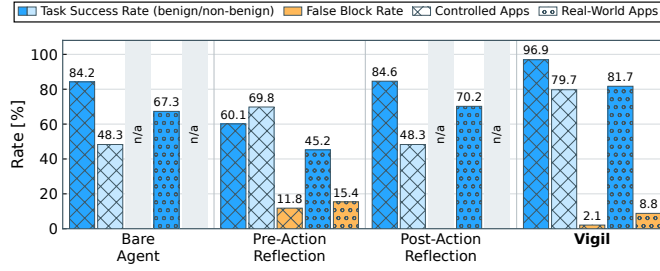


Figure 11: The end-to-end effect across baselines.

that select an item, toggle a preference, or commit an effect. Scrolls and Back attach at 0.0%, which is the designed outcome rather than a coverage shortfall. Their destination is already fixed by the canonical action, so admission finds no task-dependent relation to certify and abstains with an evidence-complete empty conjunction instead of inventing a navigation slot. What VIGIL attaches is exactly what a purely topological check cannot supply.

6.3 Agent-Integrated Evaluation

6.3.1 End-to-End Evaluation. As shown in Fig. 11, when the bare agent is compared with VIGIL under the same settings, task success rises from 84.2% to 96.9% on the four controlled apps and from 67.3% to 81.7% on the twenty real-world apps, so that the larger gain falls on the population VIGIL was never tuned for. Pre-action reflection must predict a screen it has not seen, and it does raise non-benign success to 69.8%, yet it surrenders more benign success than it buys, namely, 24.1 percentage points on the controlled apps against a 21.5 percentage-point gain, blocking benign steps at 11.8% and 15.4% to do so. Post-action reflection pays no such toll, yet it leaves non-benign success exactly where the bare agent left it, because seeing the landed screen is already too late to withhold the action that produced it. VIGIL is the only configuration that moves both columns at once, and it does so at a fraction of the refusal cost, its false-block rates of 2.1% and 8.8% running well below the 11.8% and 15.4% the pre-action baseline pays for a smaller safety gain. When the agent proposes an action the shield refuses, evidence-only feedback returns the failed conjunct with the frozen binding it was evaluated against, refuted under Deny and undecided under Uncertain, letting the agent re-plan.

6.3.2 Evolution Effectiveness. Fig. 12 replays populations across three rounds on fresh seeds, carrying the model forward and running admission between rounds. Task success rises from 64.4% to 80.8% and then to 81.7%, the first round recovering 16.4 of the 17.3 percentage points and the third adding under one, namely the saturating curve expected of an operator that only ever adds from accepted evidence. The operations decay in step, coverage expansion falling from 62

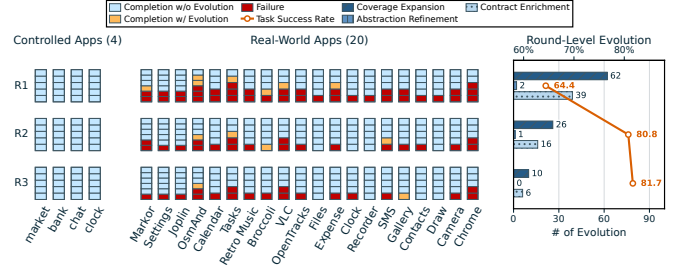


Figure 12: The effectiveness of evolution.

to 10, contract enrichment from 39 to 6, and abstraction refinement from 2 to none, so the model converges rather than accumulates. The four controlled apps trigger no evolution in any round, their models having absorbed what exploration could reach. Evolution admits an edge or contract only from newly accepted evidence and never relaxes an existing guard.

6.4 Runtime Overhead

Fig. 13 reports what VIGIL costs the device, measured with M3A on GPT-5.5 over 25 paired episodes. The latency CDF separates the two costs inside a round: the median symbolic action check completes in 3.3 ms, two orders of magnitude below the agent's planning call, and the 0.30 s of VIGIL's per-round cost is localization, slice construction, and adapter I/O. Across phases a round totals 21.7 s with VIGIL against 19.8 s without, *i.e.*, 9% more, the increase falling in Planner while Execution and Others give part of it back. The cost lands in rounds rather than in compute, the median episode taking 2 more steps, meaning that a verified action executes unimpeded while a refused one costs a re-plan. CPU load rises by 0.2 percentage points and memory pressure by 106 MB, both flat across elapsed time. Fig. 14 shows two examples. In the first, the verified run finishes in nearly the same wall-clock time as the bare agent. In the second, VIGIL reaches a Deny verdict and the agent re-plans, so the run is longer, yet it is the run that finishes, the agent alone ending in task failure.

6.5 Ablation Study

Tab. 1 toggles the three online modes of §4 over the same paired workload. Plan-time guidance alone already helps, lifting benign success from 84.2% to 86.4% and non-benign from 48.3% to 52.3% by steering the agent toward behavior the model records. Guidance is advisory and issues no verdict, so it has no false-block entry. Disabling plan-time guidance alone preserves much of the effectiveness, *e.g.*, the non-benign success reaching 70.9%. The gap is authority rather than advice, a non-benign task being one whose correct outcome is to stop before a forbidden effect, an outcome

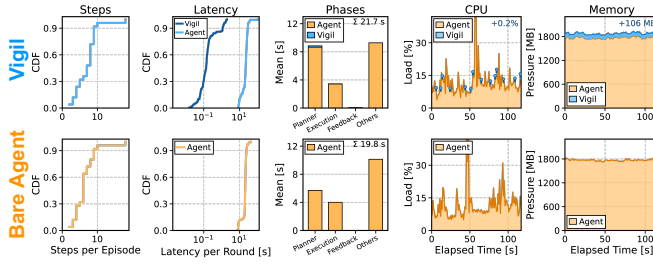


Figure 13: The runtime overhead across steps, latency, phases, CPU and memory consumption.

Table 1: Ablation results. G, S, and F means the Plan-Time Guidance, Execution-Time Shield, and Evidence-Only Feedback.

Component			TSR $\uparrow$		FBR $\downarrow$
G	S	F	benign	non-benign	benign
✗	✗	✗	84.2	48.3	–
✓	✗	✗	86.4	52.3	–
✗	✓	✓	86.8	70.9	4.2
✓	✓	✗	89.0	70.9	–
✓	✓	✓	96.9	79.7	2.1

that only the execution-time shield can produce by withholding the action. Enabling the shield without plan-time guidance may raise the false-block rate to 4.2%, because the agent proposes actions that fall outside the model or arrive with unbound slots. Removing evidence-only feedback instead leaves VIGIL enforcing the guardrails and still returning a verdict on every proposal, holding non-benign success at the same 70.9% and benign at 89.0%. What feedback adds is the last increment on both columns, from 89.0% to 96.9% and from 70.9% to 79.7%, with a refusal carrying the failed clause back to the planner via a structured log and resulting in the agent re-planning rather than the episode ending. Only the full configuration is best on both success columns, and adding guidance halves the false-block rate.

7 RELATED WORK

GUI Agents and Safeguards: Existing mobile GUI agents support action planning in several ways, including transient screen representations[50, 66], learned perception-action policies [12, 22, 56], persistent knowledge [26, 54, 71], and explicit app models [10, 21, 36]. These artifacts guide neural planning but do not authorize actions. Existing safeguards either invoke a neural verifier at each step [15, 51, 52, 58, 70] or enforce explicit rules over tool arguments [11, 35, 44, 48, 49, 59]. For GUIs, however, rule-based safeguards require app-specific predicates that identify the relevant screens, elements, and values. VeriSafe Agent illustrates this requirement by relying on predicates that developers define inside

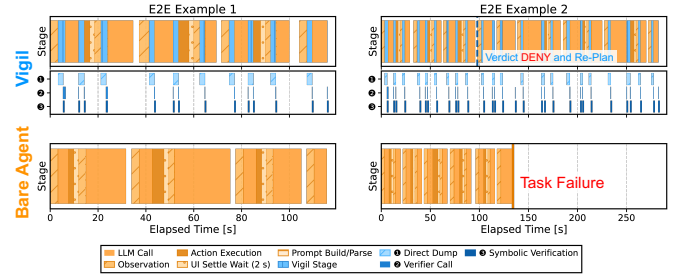


Figure 14: End-to-end examples with VIGIL integrated.

the application [25]. In contrast, VIGIL recovers $\mathcal{S}$, Σ , and the transitions among them offline and symbolically admits LLM-proposed guardrails, requiring neither app modification nor runtime LLM calls.

State Model Construction: Black-box state models have long been constructed through Android and web exploration [2, 20, 29, 34, 45]. Existing systems typically use fixed screen-grouping criteria optimized for coverage [3], with component-wise or learned comparisons handling content variation [32, 63, 64]. In contrast, VIGIL draws on abstraction refinement, automata minimization, and bisimulation [13, 14, 20, 23, 38, 39], but merges states only when their observed canonical-action behaviors remain compatible, preserving at most one successor per canonical action.

Runtime Verification: Runtime verification checks executions against stated properties and may halt, modify, or shield actions [1, 4–6, 27, 31, 43]. When applications provide no properties, candidates can be inferred from executions [18] or proposed by language models and checked symbolically [7, 17, 33, 46, 53, 55, 57]. VIGIL adapts this approach to GUIs: it resolves offline LLM proposals against recorded evidence, rejects unresolved candidates, and places only admitted guards on the deterministic runtime path.

8 CONCLUSIONS

We presented VIGIL, a symbolic runtime verifier for mobile GUI agents that is automatically constructed and deterministically enforced. VIGIL builds an evidence-grounded, guardrail-attached EFSM offline, admits LLM-proposed semantics only through symbolic compilation and admission, and at runtime uses the constructed EFSM model to guide the planner, shield proposed actions, return structured evidence, and evolve continuously on newly accepted evidence without weakening what it already enforces. Across four controlled apps and 20 unmodified Android apps, VIGIL improves both benign and non-benign task success over the state-of-the-art baselines while adding only millisecond-scale latency for every action check.

REFERENCES

- [1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe reinforcement learning via shielding. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.
- [2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 258–261.
- [3] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.
- [4] Ezio Bartocci and Yliès Falcone. 2018. *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer.
- [5] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 1–64.
- [6] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield synthesis: Runtime enforcement for reactive systems. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 533–548.
- [7] Dimitrios Stamatios Bouras, Yihan Dai, Tairan Wang, Yingfei Xiong, and Sergey Mechtaev. 2025. HoarePrompt: Structural Reasoning About Program Correctness in Natural Language. *arXiv preprint arXiv:2503.19599* (2025).
- [8] Joos CAM Buijs, Boudewijn F Van Dongen, and Wil MP van Der Aalst. 2012. On the role of fitness, precision, generalization and simplicity in process discovery. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*. Springer, 305–322.
- [9] Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, et al. 2024. Spa-bench: A comprehensive benchmark for smartphone agent evaluation. In *NeurIPS 2024 Workshop on Open-World Agents*.
- [10] Weizhi Chen, Ziwei Wang, Leyang Yang, Sheng Zhou, Xiaoxuan Tang, Jiajun Bu, Yong Li, and Wei Jiang. 2025. Pg-agent: An agent powered by page graph. In *Proceedings of the 33rd ACM International Conference on Multimedia*. 6878–6887.
- [11] Zhaorun Chen, Minton Kang, and Bo Li. 2025. Shieldagent: Shielding agents via verifiable safety policy reasoning. *arXiv preprint arXiv:2503.22738* (2025).
- [12] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. 2024. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 9313–9332.
- [13] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.
- [14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.
- [15] Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2026. V-Droid: Advancing Mobile GUI Agent Through Generative Verifiers. (2026).
- [16] Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, et al. 2026. Multi-agent collaboration via evolving orchestration. *Advances in neural information processing systems* 38 (2026), 165025–165059.
- [17] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912.
- [18] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
- [19] Yunhao Feng, Yifan Ding, Yingshui Tan, Xingjun Ma, Yige Li, Yutao Wu, Yifeng Gao, Kun Zhai, and Yanming Guo. 2026. Agenthazard: A benchmark for evaluating harmful behavior in computer-use agents. *arXiv preprint arXiv:2604.02947* (2026).
- [20] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
- [21] Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun Peter Chen, and Yang Wang. 2026. Agent-SAMA: State-Aware Mobile Assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 29459–29467.
- [22] Wenyi Hong, Weihang Wang, Qingsong Lv, Jiazhen Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. 2024. Cogagent: A visual language model for gui agents. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 14281–14290.
- [23] John Hopcroft. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.
- [24] Juyong Lee, Dongyoon Hahm, June Suk Choi, W Bradley Knox, and Kimin Lee. 2026. Mobilesafetybench: Evaluating safety of autonomous agents in mobile device control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 37565–37573.
- [25] Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. 2025. Verisafe agent: Safeguarding mobile gui agent via logic-based action verification. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*. 817–831.
- [26] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1119–1133.
- [27] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The journal of logic and algebraic programming* 78, 5 (2009), 293–303.
- [28] Ido Levy, Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, et al. 2026. St-webagentbench: A benchmark for evaluating safety and trustworthiness in web agents. In *International Conference on Learning Representations*, Vol. 2026. 122213–122255.
- [29] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*. IEEE, 23–26.
- [30] Yanda Li, Chi Zhang, Wenjia Jiang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. 2024. Appagent v2: Advanced agent for flexible mobile interactions. *arXiv preprint arXiv:2408.11824* (2024).
- [31] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)* 12, 3 (2009), 1–41.

- [32] Chenxu Liu, Junheng Wang, Wei Yang, Ying Zhang, and Tao Xie. 2026. Judge: Effective state abstraction for guiding automated web GUI testing. *ACM Transactions on Software Engineering and Methodology* 35, 3 (2026), 1–40.
- [33] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 16–28.
- [34] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)* 6, 1 (2012), 1–30.
- [35] Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T Le. 2025. Veriguard: Enhancing llm agent safety via verified code generation. *arXiv preprint arXiv:2510.05156* (2025).
- [36] Fanglin Mo, Junzhe Chen, Haoxuan Zhu, and Xuming Hu. 2025. Building a stable planner: An extended finite state machine based planning module for mobile gui agent. *arXiv preprint arXiv:2505.14141* (2025).
- [37] OpenAI. 2026. GPT-5.5 API. <https://platform.openai.com/>. Large language model API. Model ID: gpt-5.5. Accessed: 2026-08-07.
- [38] Robert Paige and Robert E Tarjan. 1987. Three partition refinement algorithms. *SIAM Journal on computing* 16, 6 (1987), 973–989.
- [39] David Park. 2005. Concurrency and automata on infinite sequences. In *Theoretical Computer Science: 5th GI-Conference Karlsruhe, March 23–25, 1981*. Springer, 167–183.
- [40] Chris Rawles, Sarah Clinckemaulie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folaayo Campbell-Ajala, et al. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, Vol. 2025. 406–441.
- [41] Yeonju Ro, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, Aditya Akella, Zhangyang Wang, Mattan Erez, and Esha Choukse. 2025. Sherlock: reliable and efficient agentic workflow execution. *arXiv preprint arXiv:2511.00330* (2025).
- [42] Anne Rozinat and Wil MP Van der Aalst. 2008. Conformance checking of processes based on monitoring real behavior. *Information Systems* 33, 1 (2008), 64–95.
- [43] Fred B Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)* 3, 1 (2000), 30–50.
- [44] Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. 2025. Progent: Securing AI agents with privilege control. *arXiv preprint arXiv:2504.11703* (2025).
- [45] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 245–256.
- [46] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*. Springer, 134–155.
- [47] Shizuo Tian, Hao Wen, Yuxuan Chen, Jiacheng Liu, Shanhui Zhao, Guohong Liu, Ju Ren, Yunxin Liu, and Yuanchun Li. 2026. Agentprog: Empowering long-horizon gui agents with program-guided context management. In *Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services*. 883–910.
- [48] Lillian Tsai and Eugene Bagdasarian. 2025. Contextual agent security: A policy for every purpose. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*. 8–17.
- [49] Haoyu Wang, Christopher M Poskitt, and Jun Sun. 2025. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. *arXiv preprint arXiv:2503.18666* (2025).
- [50] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems* 37 (2024), 2686–2710.
- [51] Shaokang Wang, Pei Fu, Ruoceng Zhang, Shaojie Zhang, Xiuwen Xi, Jiahui Yang, Bin Qin, Ying Huang, Zhenbo Luo, and Jian Luan. 2026. GAIA: A Data Flywheel System for Training GUI Test-Time Scaling Critic Models. *arXiv preprint arXiv:2601.18197* (2026).
- [52] Yuyang Wanyan, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Jiabo Ye, Yutong Kou, Ming Yan, Fei Huang, Xiaoshan Yang, et al. 2026. Look before you leap: A gui-critic-r1 model for pre-operative error diagnosis in gui automation. *Advances in Neural Information Processing Systems* 38 (2026), 3907–3929.
- [53] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328.
- [54] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, et al. 2025. Autodroid-v2: Boosting slm-based gui agents via code generation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*. 223–235.
- [55] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations*, Vol. 2024. 2968–2978.
- [56] Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, et al. 2026. Gui-actor: Coordinate-free visual grounding for gui agents. *Advances in Neural Information Processing Systems* 38 (2026), 15101–15128.
- [57] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in neural information processing systems* 35 (2022), 32353–32368.
- [58] Zhenyu Wu, Jingjing Xie, Zehao Li, Bowen Yang, Qiushi Sun, Zhaoyang Liu, Zhoumianze Liu, Yu Qiao, Xiangyu Yue, Zun Wang, et al. 2026. Os-oracle: A comprehensive framework for cross-platform gui critic models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 27514–27524.
- [59] Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, et al. 2024. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning. *arXiv preprint arXiv:2406.09187* (2024).
- [60] Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the weakness of large language model agents within a complex android environment. In *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*. 6061–6072.
- [61] Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. 2025. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2144–2166.
- [62] Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. 2025. An illusion of progress? assessing the current state of web agents. *arXiv preprint arXiv:2504.01382* (2025).
- [63] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *Proceedings of the*

- ACM/IEEE 42nd international conference on software engineering. 186–197.
- [64] Rahul Krishna Yandrapally and Ali Mesbah. 2022. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1086–1101.
- [65] Cheng Zhang, Erhu Feng, Xi Zhao, Yisheng Zhao, Wangbo Gong, Jiahui Sun, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. 2025. MobiAgent: A Systematic Framework for Customizable Mobile Agents. *arXiv preprint arXiv:2509.00531* (2025).
- [66] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI conference on human factors in computing systems*. 1–20.
- [67] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2025. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, Vol. 2025. 34040–34077.
- [68] Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. 2024. Llamatouch: A faithful and scalable testbed for mobile ui task automation. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–13.
- [69] Yedi Zhang, Yufan Cai, Xinyue Zuo, Xiaokun Luan, Kailong Wang, Zhe Hou, Yifan Zhang, Zhiyuan Wei, Meng Sun, Jun Sun, et al. 2024. The fusion of large language models and formal methods for trustworthy ai agents: A roadmap. *arXiv preprint arXiv:2412.06512* (2024).
- [70] Yuzhe Zhang, Xianwei Xue, Xingyong Wu, Mengke Chen, Chen Liu, Xinran He, Run Shao, Feiran Liu, Huanmin Xu, Qiutong Pan, et al. 2026. Don't Act Blindly: Robust GUI Automation via Action-Effect Verification and Self-Correction. *arXiv preprint arXiv:2604.05477* (2026).
- [71] Shanhuai Zhao, Hao Wen, Wenjie Du, Cheng Liang, Yunxin Liu, Xiaozhou Ye, Ye Ouyang, and Yuanchun Li. 2025. LLM-explorer: towards efficient and affordable LLM-based exploration for mobile apps. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*. 589–603.

A EVIDENCE-COLLECTING EXPLORATION

Alg. 1 details the evidence-collecting exploration of § 3.1. It returns the evidence set E of accepted triples $\langle o_t, a_t, o_{t+1} \rangle$, working entirely in the observation domain: no canonical action or abstract state is formed here.

Algorithm 1: Evidence-collecting exploration.

Input: app under test A ; trial budget $budget$
Output: evidence set E of accepted triples $\langle o_t, a_t, o_{t+1} \rangle$

```

1: function EXPLORE( $A, budget$ )
2:    $o_0 \leftarrow \text{RELAUNCH}(A)$ ;  $cur \leftarrow o_0$ 
3:    $path(o_0) \leftarrow \epsilon$ ;  $cand(o_0) \leftarrow \perp$  //  $\perp$ : never swept, not  $\emptyset$ 
4:    $frontier \leftarrow \{o_0\}$ ;  $E \leftarrow \emptyset$ ;  $n \leftarrow 0$ 
5:   while  $n < budget$  do //  $n$  counts trial actions
6:     // Enumeration: at most one observation per iteration
7:      $unswept \leftarrow \{o \in frontier : cand(o) = \perp\}$ 
8:     if  $unswept \neq \emptyset$  then
9:        $\bar{o} \leftarrow \arg \min_{o \in unswept} |path(o)|$  // breadth-first
10:       $o_t \leftarrow \text{ARRIVE}(A, \bar{o})$ 
11:      if  $o_t = \perp$  then  $frontier \leftarrow frontier \setminus \{\bar{o}\}$ ; continue
12:       $cur \leftarrow o_t$ ;  $cand(\bar{o}) \leftarrow \text{SWEEP}(o_t)$ 
13:     // Trial: exactly one candidate action per iteration

```

```

14:     // locality first: a local pick avoids a relaunch
15:      $(\bar{o}, a_t) \leftarrow \text{PICKIN}(cur)$  or  $\arg \max_{(o,a)} \text{PRIORITY}(o, a)$ 
16:     if  $(\bar{o}, a_t) = \perp$  then break // all candidates attempted
17:      $cand(\bar{o}) \leftarrow cand(\bar{o}) \setminus \{a_t\}$  // no retry, even if the replay fails
18:      $o_t \leftarrow \text{ARRIVE}(A, \bar{o})$ 
19:     if  $o_t = \perp$  then  $frontier \leftarrow frontier \setminus \{\bar{o}\}$ ; continue
20:      $cur \leftarrow o_t$ ;  $o_{t+1} \leftarrow \text{EXECUTE}(a_t)$ 
21:      $cur \leftarrow o_{t+1}$ ;  $n \leftarrow n + 1$ 
22:     if not  $\text{ACCEPTED}(o_t, a_t, o_{t+1})$  then continue
23:      $E \leftarrow E \cup \{(o_t, a_t, o_{t+1})\}$ 
24:     if  $o_{t+1} \notin frontier$  then
25:        $path(o_{t+1}) \leftarrow path(\bar{o}) \cdot a_t$ ;  $cand(o_{t+1}) \leftarrow \perp$ 
26:        $frontier \leftarrow frontier \cup \{o_{t+1}\}$ 
27:   return  $E$ 

28: function ARRIVE( $A, \bar{o}$ )
29:   // a warm screen is reused only after it verifies
30:   if  $cur = \bar{o}$  and no swipe moved the viewport then
31:      $o_t \leftarrow \text{CAPTURE}$ 
32:     if  $o_t = \bar{o}$  then return  $o_t$  // no relaunch
33:    $cur \leftarrow \text{RELAUNCH}(A)$ 
34:   for  $a \in path(\bar{o})$  do
35:      $cur \leftarrow \text{EXECUTE}(a)$  // replay, not charged
36:    $o_t \leftarrow \text{CAPTURE}$ 
37:   if  $o_t = \bar{o}$  then return  $o_t$ 
38:   return  $\perp$  // replay did not reproduce  $\bar{o}$ 

```

Primitives: **RELAUNCH** cold-starts A from its entry activity, and **CAPTURE** dumps the current screen as an a11y file together with a screenshot, which is the observation the algorithm stores and compares. **SWEEP** lists the candidate actions an observation exposes; **PICKIN** returns an untried candidate on a given observation, if one remains; and **PRIORITY** ranks the remaining candidates so that shallower observations and unexercised action types are attempted first. **ACCEPTED** is the admission test for evidence: a triple is accepted only when both of its endpoints remain inside A and the successor observation is stable, so that an excursion into another app or a capture taken mid-transition never enters E . Everything downstream—transitions, guardrail evidence, and the accepted evidence required by evolution (§ 5)—is drawn from E alone.

Frontier: Each iteration makes two passes. The enumeration pass sweeps at most one observation of the *frontier*, the set of observations that have been reached but not yet exhausted; $cand(o) = \perp$ marks an observation that has never been swept, which is distinct from $cand(o) = \emptyset$, one that was swept and whose candidates are all consumed. The trial pass then executes exactly one candidate action, preferring a candidate on the current screen because reaching any other observation requires relaunching A and replaying its recorded action path. A candidate is consumed when it is picked, so no action is attempted twice, and an observation that cannot be reproduced is dropped from the frontier in either pass.

Budget: $budget$ bounds the number of trial actions. The replay steps that **ARRIVE** performs are not charged against it,

so the number of interactions issued to the device exceeds *budget* and grows with the average path length of the frontier. Exploration stops when the budget is exhausted or when every collected candidate has been attempted.

B BACKBONE SENSITIVITY

Fig. 15 crosses six mobile agent backbones with the six that synthesized the guardrails over the 228 benign tasks of the four controlled apps, every row carrying the bare agent for comparison. Every one of the 36 pairings matches or beats its corresponding bare-agent run, 33 improving on it and three matching it, so no combination of the two backbones leaves the agent worse off than running without VIGIL. Reading across columns, *i.e.*, fixing the synthesizer and changing the agent, the GPT agents finish highest, GPT-5.5 averaging 89.4% and GPT-5.4 80.6%, the newer model leading the older under all six bundles.

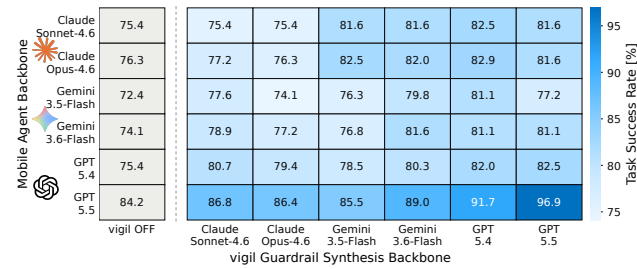


Figure 15: The task success rate across mobile agent and guardrail-synthesis backbones.

The Claude agents gain the least, gaining 4.1 and 4.3 points, whereas the other four gain 5.2 to 5.4 points, though even their weakest pairing still matches the unguarded level. Reading across rows, *i.e.*, fixing the agent and changing the synthesizer, GPT-5.4, GPT-5.5, and Gemini 3.6-Flash are the bundles worth building, averaging 83.6%, 83.5%, and 82.4% against the 78.1% of the weakest. Stronger synthesizers are associated with higher task success, consistent with higher-quality semantic guardrails. Taken together, VIGIL helps under every pairing while paying most where the underlying models read screen semantics best, the agent backbone setting the level a row reaches and the strongest pairing compounding into the largest single gain in the figure.