HiOS: A Device-Specific Agent Framework for Intent Graph Execution

Jiahe Tian | HiOS Labs Inc.
ipo@stanford.edu | ORCID: 0009-0007-8750-287X

Abstract

An agent procedure can remain semantically valid while becoming ineligible on the device that must execute it. HiOS is a device-specific agent framework that separates a portable intent graph from a plan bound to supported tools, model interfaces, adapter schemas, authorized scope, and verification contracts. The runtime rechecks live resource and service conditions before inference, action, and resumption, preserving verified work while rebinding or deferring affected nodes. iOS provides the motivating application context. A Python reference implements the specialization and admission layer for synthetic API and foreground-web adapters. Across 27 selected conformance scenarios, it matches the declared terminal outcomes with zero ineligible dispatches or wrong-payload writes; 11 cases complete and ten remain pending. A bind-once control makes 16 ineligible attempts. Supporting studies check all 65,536 four-node directed graphs on macOS and iOS Simulator and execute 6,804 recovery traces under declared service semantics. These results establish bounded component behavior, not end-to-end agent accuracy or physical resource savings. The framework reserves generative reasoning for unresolved decisions and makes device compatibility an explicit condition of reuse. Native integration, real-model comparisons, and device-specific cost calibration remain necessary to test its efficiency hypothesis.

1 Introduction

An agent's next action depends on the device executing it. A procedure that can send a receipt through a supported API may need a foreground web adapter on another installation, and may become ineligible after a permission change, application suspension, or disconnection. Even when an on-device foundation model is available, the model cannot create a missing tool, restore network reachability, or guarantee that an interrupted write did not commit. A mobile agent framework must therefore connect semantic planning to the capabilities and current conditions of a specific execution environment.

We propose HiOS as a *device-specific agent framework*. It separates a portable intention—the user's goal, arguments, dependencies, and required outcomes—from a device-bound execution plan. The framework specializes that plan to supported adapters, software interfaces, permissions, and verification evidence, then rechecks its eligibility as device and service conditions change. iOS is the motivating platform and prototype context; the specification does not establish implementation on other operating systems.

Consider "save this receipt and send the total to my team." Once the receipt fields and authorized recipient are explicit, local persistence and transmission are ordinary operations with checkable outcomes. The same intended outcome can require different adapters, foreground conditions, and retry rules on different devices. HiOS aims to execute these resolved parts without repeated generative decisions while returning to inference only when semantics, observation, or adapter binding remains unresolved. Device-specific execution is the central mechanism; reduced inference expenditure is a hypothesis to test.

We study a setting in which a compatible on-device foundation model is available and remote model credentials are already configured. These assumptions remove installation and credential

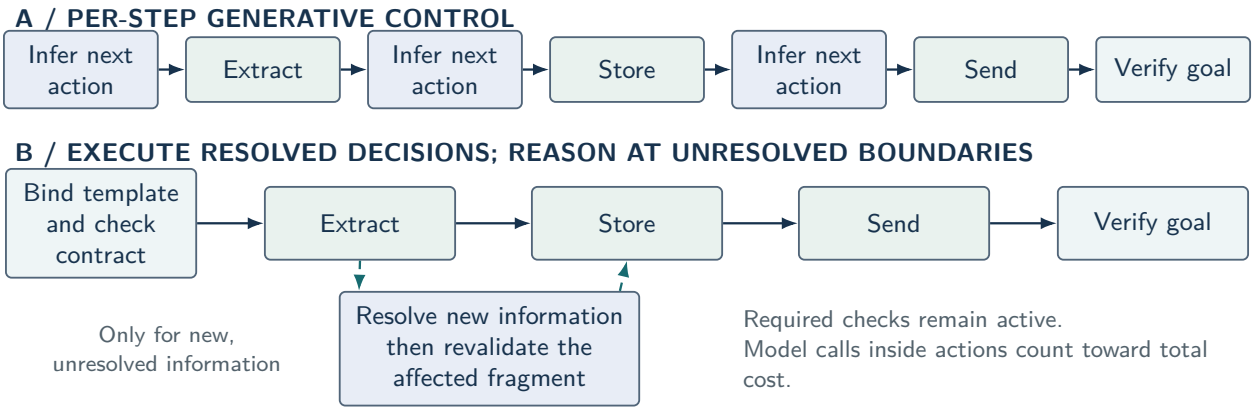


Figure 1: Where new reasoning enters execution. This is a conceptual comparison, not a timing trace or a measured token reduction. Spacing does not encode cost. Per-action checks are omitted from the drawing for clarity; the compared systems must satisfy the same verification obligations.

acquisition from the research problem. They do not imply that either model can solve every task, that a remote endpoint is reachable, or that an application is permitted to control another application. The operating system's supported interfaces remain a hard boundary.

Our proposed unit of execution is a finite *intent graph*: a dependency graph of typed actions and explicit verification obligations, instantiated from the user's request. A graph can be constructed using inference, bound from a reusable template, or assembled from known operations. Its executor performs resolved actions without requiring a new generative decision at every step. An outer control loop observes execution and resource changes, requests additional reasoning when necessary, and constructs a new graph fragment when an observation reveals information that was unavailable earlier.

Figure 1 illustrates the proposed change in control flow. The intended reduction is in repeated decisions whose inputs and contracts are already resolved; extraction, verification, or repair can still require inference and must remain in the cost accounting.

This separation is not, by itself, a new research contribution. ReAct established an influential reasoning–action loop [29]; LLMCompiler organizes function calls using dependencies [20]; mobile and computer agents already learn or retrieve reusable procedures [12, 22, 27, 30]. More recent systems reduce the frequency of large-model decisions [17, 31], while HybridFlow routes subtasks in a dependency graph across edge and cloud resources [15]. A defensible contribution must therefore go beyond "use a DAG," "reuse workflows," or "choose local versus cloud."

The central question is: *Which parts of a resolved agent procedure remain valid on a particular device as its capabilities, permissions, resources, and reachable services change?* A second question asks whether maintaining this validity is cheaper than repeatedly replanning, after accounting for binding, monitoring, verification, and repair.

The paper specifies three connected elements: a device execution profile and compatibility contract; a specialization and preflight procedure that preserves verified work while rebinding or deferring affected nodes; and an executable conformance study with trace-based checks. Supporting graph and recovery studies test dependency handling and uncertain side effects. The components constitute a framework specification and bounded reference implementation. General semantic planning, calibrated resource optimization, and end-to-end iOS integration require further evaluation.

The native HiOS application supplies a planner, graph representation, sequential executor, verification paths, and workflow records. The new Python reference specializer makes the device contract executable for controlled fixtures; it is separate from that application. This distinction identifies both what can now be reproduced and what must still be integrated.

2 Related Work and Research Position

Reasoning and dependency graphs. ReAct interleaves reasoning with environment actions, making observations available to subsequent decisions [29]. LLMCompiler separates planning, dependency handling, and function execution [20]. These ideas motivate graph execution but also establish that dependency-aware orchestration is prior work. Our proposed executor differs in its target decision: whether a fragment requires new inference under a phone's current constraints, including the consequences of delaying or repeating a state-changing action.

Workflow memory and skill reuse. MobileGPT organizes mobile task knowledge into reusable subtasks [22]. Mobile-Agent-E adds evolving guidance and executable shortcuts [27]. CUA-Skill develops parameterized skills and compositions for computer use [12], and MobiAgent includes a record-and-replay component [30]. These systems substantially overlap with the motivation to avoid rediscovering procedures. We do not claim workflow memory as novel. The proposed distinction is that reuse eligibility depends on typed bindings, interface versions, current preconditions, and explicit verification obligations, and that the reuse decision participates in resource scheduling.

Less frequent generative control. ClawMobile separates probabilistic reasoning from deterministic smartphone execution backends [17]. Jev-Mobile delegates repeated low-level decisions to a lightweight typed executor under a higher-level goal [31]. V-Droid uses generative verification to select from candidates [13]. These approaches support the importance of execution structure and are direct intellectual neighbors. Their mechanisms are not automatically available through an opaque on-device model API: for example, an application that receives generated text cannot assume access to arbitrary scoring or prefill-only interfaces. We also avoid equating lower cost among successful trajectories with lower cost at matched task success.

Adaptive inference and hardware scheduling. FrugalGPT studies model cascades that trade inference cost against quality [11]. HybridFlow is the closest overlap in graph-level resource adaptation: it constructs dependency-aware subtasks and routes them between edge and cloud under budget considerations [15]. Its evaluated reasoning workloads and edge GPU configuration differ from state-changing application workflows on an iPhone. HeRo studies orchestration of agentic retrieval pipelines on heterogeneous mobile hardware [23]. Its hardware-mapping focus differs from the public-interface constraints of an application using a system-managed model. Carbon-aware function-call routing adds another resource objective to edge–cloud selection [24]. Taken together, these works rule out a broad novelty claim for adaptive routing. The device-specific framework makes compatibility and current eligibility explicit at each of these choices: *whether to reason, where to reason, and when an observable action is safe to execute or resume.* The reference contribution is the specified binding/admission boundary and its selected conformance probes; no measured routing advantage is asserted.

Evaluation environments and platform capabilities. AndroidWorld supplies parameterized tasks with state-based evaluation [25]. iOSWorld extends phone-agent evaluation to native

Table 1: Closest research directions and the additional question posed here. Entries describe published focus; they are not experimentally established advantages of HiOS.

Direction	Established focus	Question for this proposal
LLMCompiler [20]	Dependencies and function-call orchestration	When can a phone execute a resolved fragment without new inference?
MobileGPT, Mobile-Agent-E, CUA-Skill, MobiAgent [12, 22, 27, 30]	Reusable mobile/computer procedures	Does versioned, contract-checked reuse pay for its own validation and repair?
ClawMobile, Jev-Mobile, V-Droid [13, 17, 31]	Alternative reasoning/execution or verification structures	Do savings persist with supported iOS interfaces and independent success checks?
HybridFlow [15]	Adaptive edge–cloud routing over dependency graphs	Can reuse, side effects, thermal pressure, and intermittent action services be handled jointly?
HeRo [23]	Mobile hardware orchestration for agentic RAG	What adaptation remains useful when model hardware scheduling is opaque?

iOS simulator applications [19]. Both inform our use of independently checked final states. Neither substitutes for energy measurements on physical deployment devices, and benchmark control interfaces must not be assumed to be privileges available to a shipped iOS application. Apple's Foundation Models framework supplies model integration and tool-calling support [1, 8]; its evolving capabilities, including dynamic profiles, must be treated as a baseline dependency rather than an original contribution [6].

Side-effect recovery. RIFL implements exactly-once RPC semantics using durable completion records and associated server mechanisms [21]. Our weaker application-level reconciliation depends on each service's identity, query, and retry contracts; it supplies no generic exactly-once guarantee or new distributed-transaction primitive.

Table 1 summarizes the comparison as a difference in research focus, not a claim that every neighboring system lacks every proposed feature. The targeted primary-source review was updated through 25 October 2026; it is not an exhaustive systematic review.

3 Device Model and Intent Contracts

3.1 Availability is not executability

Let u denote a user request, z_t the observed application state, and r_t the resource observation at decision time t. We represent resources as

$$r_t = (b_t, \ell_t, \theta_t, n_t, m_t, d_t), \tag{1}$$

where b_t is battery level, ℓ_t is low-power mode, θ_t is the operating system's thermal category, n_t summarizes network conditions, m_t is a memory-pressure signal, and d_t is the remaining deadline.

These observations are partial and sometimes delayed. A network path marked available does not prove that an authenticated service request will succeed. Thermal categories are not direct temperature measurements.

The local model is assumed installed and available on the chosen supported devices. Remote credentials are assumed configured. Model inference is distinct from application actions: cloud reasoning can produce a valid local calendar operation, and local reasoning can propose an upload that still requires a remote service. We therefore associate every backend with a capability descriptor covering permissions, supported action schemas, network requirements, foreground requirements, and interface version.

"Without an application API" means that an action may use another supported control surface, such as an app-owned web view or an exposed system integration. It does not mean arbitrary native-app automation. iOS sandbox restrictions and explicitly exposed services bound the action space [7]. Unsupported operations terminate with a clear handoff or an infeasibility result. They are not silently removed from the task.

3.2 Intent graphs and contracts

An execution epoch is a directed acyclic graph $\mathcal{G} = (V, E)$ with bound inputs x, a user-goal specification g, and a set of required terminal predicates. Each node $v \in V$ is

$$v = (a, I, O, P, Q, C, R, \nu), \tag{2}$$

where a is a registered action, I and O are input and output types, P and Q are preconditions and observable postconditions, C is its capability requirement, R is a retry and side-effect policy, and ν identifies its schema and implementation version. Edges carry typed output-to-input references or explicit ordering constraints. A node becomes ready only when its dependencies have committed verified outputs and its guard is satisfied.

A request may initially be underspecified. The compiler must identify missing slots, conflicting constraints, and actions requiring user authorization. It may ask for clarification or produce a bounded partial plan. It must not infer a recipient or destructive scope solely to make a graph executable. Goal predicates also require review: a perfectly executed graph can still encode the wrong interpretation of the user's intent.

The DAG describes one finite epoch, not the entire lifetime of an agent. Unknown UI state, newly returned service data, and conditional branches can require another epoch. Retrying an action is an executor state transition, not an edge that introduces a cycle into the current DAG. This distinction permits acyclic dependency validation without pretending that real interaction has no feedback.

Figure 2 instantiates these definitions for a receipt task. The recipient is explicit, a saved-record check precedes sending, and separate local and remote predicates are needed to satisfy the complete goal. These dependencies do not imply that graph branches are currently executed in parallel.

3.3 Device-specific execution contract

The framework separates a portable intention from the program that one device is permitted and able to execute. Define a device execution profile

$$\mathcal{D} = (\kappa, K, A, V, \rho), \tag{3}$$

where κ identifies relevant hardware-class and software-interface versions; K describes model and tool capabilities; A maps actions to supported API or UI adapters; V specifies available verification

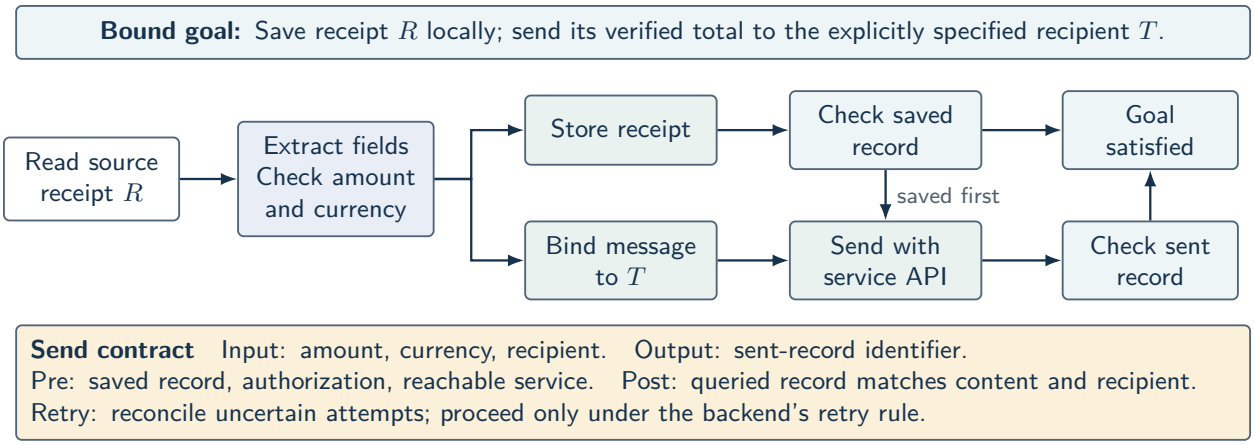


Figure 2: Worked intent graph for an illustrative fixture using app-owned storage and a supported messaging service. Arrows express data or ordering dependencies. A verified sent record is the stated observable goal; it does not establish that a human read the message. The example does not assume arbitrary control of native iOS apps.

and retry evidence; and ρ identifies optional device-specific cost calibration. A profile describes an execution environment, not a person's identity or a permission grant. Two phones with the same chip can have different profiles because their software, integrations, or account scopes differ. Device specificity therefore includes supported behavior as well as physical resources.

The profile changes less often than the live state $s_t = (z_t, r_t, p_t, f_t)$, which adds current permissions p_t and foreground/background eligibility f_t to the application and resource observations. Static support and current permission are distinct: a registered messaging adapter does not establish an authorized destination or a live session. Similarly, an installed model does not make every inference mode or action feasible at the current deadline and resource limit.

Specialization binds a portable graph to the current profile:

$$(G_{\mathcal{D}}, \sigma) = \text{Specialize}(G, x, \mathcal{D}), \tag{4}$$

where x contains explicit task arguments and σ records the relevant profile fingerprint, adapter schemas, and authorized scope. Specialization chooses concrete adapters, encodes arguments, and attaches their preflight and postcondition checks. It does not re-interpret the user's recipient or relax the required goal. An unsupported action yields infeasibility or handoff; a missing semantic argument yields clarification.

Reuse eligibility is device-bound. A template preserves a portable procedure and parameter schema. Its previously bound execution plan is reusable only when its compatibility stamp and current preconditions still hold. New arguments are rebound even on the same device; account-specific identifiers and old recipients are never reusable parameters by default. A schema, adapter, model-interface, or authorized-scope change invalidates the affected binding. Rebinding need not invoke a model when the new adapter and its parameter transformation are already registered. Unknown mappings return to a reasoning or clarification boundary.

Revalidate at dispatch and resume. Immediately before inference or a state-changing action, the executor checks the relevant profile stamp and live eligibility. On resumption it retains verified

outputs whose contracts still hold, rebinds affected future nodes, and reconciles uncertain earlier side effects before considering another write. This gives a checkable admission invariant:

$$\begin{aligned} \text{Dispatch}(v,t) \Rightarrow\ & \text{Compatible}(v, \mathcal{D}_t) \\ & \wedge\ \text{Authorized}(v, p_t) \\ & \wedge\ \text{Eligible}(v, s_t). \end{aligned} \tag{5}$$

The invariant concerns the last observed state. A service or permission can change after preflight; platform enforcement, tool errors, postcondition verification, and unknown-commit recovery remain necessary. No atomic snapshot across a phone and remote services is assumed.

A minimal executable policy. The reference specializer in Section 6.1 implements versioned binding and the admission gate for synthetic API and foreground-web adapters. Its routing rule prefers eligible local reasoning, otherwise eligible cloud reasoning, otherwise deferral. Local eligibility checks configured thermal, memory, local-budget, and profile-specific low-power limits; cloud eligibility checks connectivity, model-service reachability, and data-egress permission. The rule makes the framework executable without pretending to know optimal device costs. A calibrated selector using ρ is a later extension. Model-interface availability is assumed; the conformance study checks dispatch decisions without invoking a model.

3.4 Costs, quality, and feasibility

At a decision boundary the controller chooses an option o from direct execution, validated reuse, local inference, cloud inference, or deferral. An option also identifies its action backend and verification procedure. Feasibility is a mask

$$\mathcal{F}_t = \{o : C(o, z_t, r_t) = 1\}, \tag{6}$$

which excludes unsupported permissions, unreachable required services, invalid contracts, and prohibited resource states. Conditions that cannot be established become uncertainty, not evidence of feasibility. The mask is checked again immediately before a side effect.

Among feasible execution options, a candidate controller minimizes

$$\widehat{J}_{\mathcal{D}}(o) = w_E \frac{\widehat{E}(o)}{E_0} + w_L \frac{\widehat{L}(o)}{L_0} + w_T \frac{\widehat{T}(o)}{T_0} + w_C \frac{\widehat{C}_{\$}(o)}{C_0}, \tag{7}$$

subject to a target lower confidence bound on task completion, a deadline, and the user's resource limits. $\widehat{E}$ estimates device energy, $\widehat{L}$ elapsed latency, $\widehat{T}$ token expenditure, and $\widehat{C}_{\$}$ remote inference charges. Positive reference quantities E_0, L_0, T_0, C_0 make the units explicit. Weights and limits are fixed before evaluation and sensitivity is reported. Device energy and cloud charges are distinct quantities; neither represents total environmental impact.

The estimates belong to calibration ρ for a declared device/software configuration. Reusing a task graph does not authorize transferring another device's latency or energy estimates. Missing or stale calibration triggers a declared fallback rule and an "uncalibrated" trace field, not an invented cost prediction. The executed reference policy tests eligibility and rebinding; it does not instantiate this calibrated objective.

For a fragment, estimates include its continuation, observation, verification, and likely repair costs. Greedy selection on only the next model call could prefer an apparently cheap action that makes the rest of the graph expensive or impossible. Our proposed initial policy uses a bounded

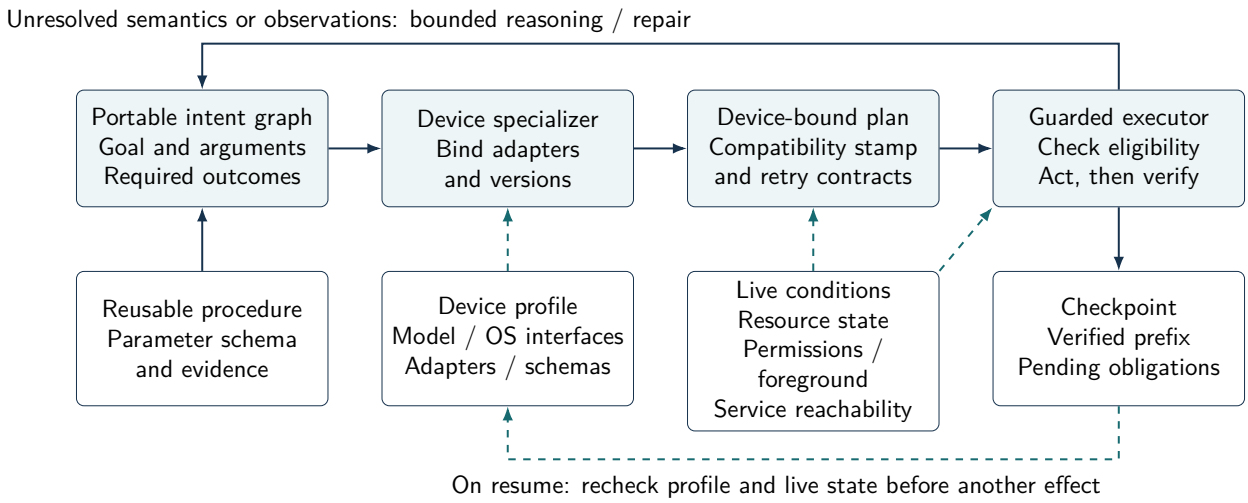


Figure 3: Device-specific framework. A reusable intention becomes an executable plan only after adapter binding and compatibility checks. Live conditions are rechecked at inference, action, and resumption boundaries. The specialization and admission layer has a synthetic reference implementation; native integration and calibrated resource selection remain pending.

remaining-graph estimate; a one-step policy is an ablation. Predictions and confidence bounds are empirical approximations. This formulation does not prove a real-world completion guarantee.

Deferral is represented separately from successful execution. It requires a reason, an expiry or resumption condition, and a durable description of outstanding obligations. A task cannot become "successful" merely because the controller avoided its expensive part. Evaluation distinguishes complete, failed, infeasible, pending, and handed-off outcomes under a common deadline.

3.5 Token accounting

We count local and remote input and output tokens separately, including planning, verification, repair, and repeated context. Counts are associated with model and tokenizer versions; tokens from different tokenizers are not interchangeable units of compute. When a platform does not expose token counts, an estimate is labeled as an estimate and character/byte counts and model-call counts are also retained. Reused cached inputs and provider billing categories are recorded when exposed. "No new generated tokens" for a deterministic action does not mean zero computation or zero energy.

4 Framework Specialization and Execution

Figure 3 separates portable intent, device specialization, and guarded execution. This section specifies the complete framework; Section 5 distinguishes the native prototype, the executable reference layer, and unimplemented extensions.

4.1 Compile intent once, resolve uncertainty when needed

The compiler first normalizes explicit arguments, retrieves candidate templates, and checks whether a registered deterministic interpretation applies. A deterministic path is appropriate only when it

resolves the requested semantics with sufficient specificity; a keyword match alone is insufficient for ambiguous requests. A compact local-model call can fill typed slots or select a candidate. A remote model is an available escalation when the connection, user constraints, and estimated value justify it.

Generated output enters the runtime through a schema validator. Guided generation can reduce structural failures, but it cannot establish that an action is semantically appropriate [8]. Validation checks registered tool names, argument types, required dependencies, capability requirements, finite size, and acyclicity. It rejects undeclared output references and checks that required goal predicates have a producer or an explicit unresolved obligation. A model cannot grant itself a permission by writing a capability into its output.

Plans contain stable decisions and symbolic references, not guesses about unseen screen coordinates or future service results. For example, a file identifier returned by an extraction step is bound to a later persistence operation. If a website changes its form or a service returns several possible recipients, the executor stops at that observation boundary and requests a small repair or clarification. It need not regenerate already verified work.

4.2 Reuse with explicit invalidation

A reusable fragment contains a parameter schema, dependencies, action contracts, compatible interface versions, a provenance record, and validation statistics stratified by relevant conditions. Retrieval produces candidates; it does not authorize execution. Eligibility requires valid argument bindings, current preconditions, appropriate permissions, and compatible versions. A semantically similar request can still require a different destination, time zone, account, or side-effect policy.

Successful execution can nominate a fragment for reuse, but successful tool return values alone do not qualify it. Promotion requires independent postcondition checks and removal of instance-specific state. The runtime retains the distinction between a template and a past trace. It invalidates or quarantines a template when a schema changes, its oracle becomes unavailable, or drift causes repeated verification failures. A bounded cache prevents unlimited growth and reports retrieval overhead.

Reuse has an amortization cost. Let C_b be one-time construction and qualification cost, C_f the cost of executing a comparable task with fresh reasoning, and C_r the cost of binding, validating, and executing a reused fragment, including expected repair. Under stationary costs and comparable completion quality, reuse is cheaper after n executions only if

$$C_b + nC_r < nC_f, \qquad n > \frac{C_b}{C_f - C_r}, \quad C_f > C_r. \tag{8}$$

The inequality is an accounting identity, not a measured result. Frequent interface changes or rare tasks can prevent amortization. Evaluation must therefore include cold construction, warm reuse, and drift separately.

4.3 A controller with bounded overhead

The first native integration should use a transparent rule policy and measured cost tables. Thermal and low-power notifications, path changes, execution outcomes, and explicit phase boundaries trigger reconsideration. The controller should not query a language model merely to decide whether the phone is hot. It samples public resource signals at a bounded rate and uses hysteresis to avoid repeatedly switching modes when a signal oscillates.

At each boundary, the controller updates the feasibility mask, then compares the remaining costs of eligible options. It can continue a validated local operation, request local interpretation,

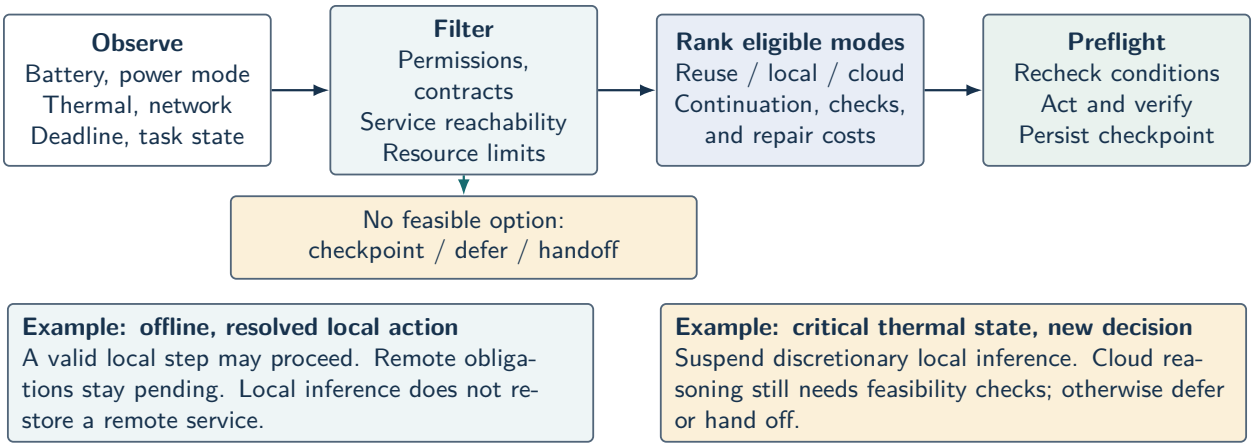


Figure 4: Proposed decision sequence and two conditional examples. Battery level, thermal state, or connectivity alone does not uniquely determine the best mode. The examples specify eligibility rules, not empirically established optimal choices or measured energy savings.

use remote interpretation, or checkpoint and defer. Remote reasoning is not an automatic response to thermal pressure: connection quality, radio activity, remaining deadline, data constraints, and whether the action itself requires the network all matter. Likewise, low battery does not imply that local inference is the cheapest choice.

A learned cost or success estimator is a later, separately evaluated extension. It may use held-out calibration data and conservative bounds; unsupported contexts fall back to the rule policy. Updates from completed tasks must be versioned and restricted to training or explicitly declared online-adaptation protocols. Benchmark test outcomes must not quietly populate a reusable skill library for later test cases. Neither exploration on real side effects nor a claim of optimality is required for the initial paper.

For a concrete first policy, unsupported actions are rejected before cost ranking; remote options are removed when their required service is unavailable; and new discretionary local inference is suspended in the critical thermal category. Among the remaining options, a valid bound fragment competes with fresh reasoning using the same estimated continuation costs. Missing semantic information triggers a bounded local call, remote escalation, or clarification according to calibrated capability and budget checks. If no option meets those checks, the obligation remains pending or is handed off. Thresholds for battery reserve, hysteresis duration, prediction confidence, and maximum repair count are explicit configuration values fixed before the final evaluation. The reference implementation tests the eligibility subset and device rebinding. The calibrated remaining-cost selector and its thresholds remain to be implemented and measured.

Figure 4 makes the ordering explicit: feasibility filtering precedes cost ranking, and a preflight check protects against conditions changing between selection and dispatch.

4.4 Ordering, side effects, and recovery

Graph independence is necessary but not sufficient for parallel execution. Two nodes can be logically independent and still share a single foreground page, a UI navigation stack, or a mutable record. Nodes therefore declare resource locks or conflict domains. The initial executor may remain sequential; any parallel extension must enforce both dependency readiness and resource exclusion.

Algorithm 1 Execution at observation boundaries (proposed)

Input: Request u, capability registry K, budgets B

$(G, g) \leftarrow$ CompileOrBind(u, K)
Validate(G,g,K)
while required obligations remain **do**
    $(z, r) \leftarrow$ Observe( )
    Reconcile(G,z)
    $V_r \leftarrow$ ReadyNodes(G, z)
    $F \leftarrow$ FeasibleOptions(V_r, z, r, K, B)
    if F has no executable option **then**
        return CheckpointOrStop(G,g,z,B)
    end if
    $o \leftarrow$ SelectBoundedCost(F, G, r, B)
    if o requires reasoning **then**
        $\Delta G \leftarrow$ ResolveBoundary(o, z)
        ValidateAndSplice$(G, \Delta G, K)$
    else
        PreflightAndLog(o,z)
        $y \leftarrow$ Execute(o)
        $q \leftarrow$ Verify(o, y)
        CommitOrRepair(G,o,q)
    end if
    DebitCosts(B)
end while
return VerifyGoal(g)

Higher concurrency also requires a fresh cost estimate because it can increase memory contention or thermal pressure.

Before a state-changing action, the runtime persists an attempt identifier and, where supported, an idempotency key. After execution, it records a verified commit and the resulting object identifier. If the application is suspended or the connection fails after dispatch, the action enters an *unknown-commit* state. The runtime queries the destination for the expected effect before retrying. An empty current snapshot is insufficient if an earlier request can still commit: reconciliation needs a settled result, an effective cancellation or fencing mechanism, or another backend-specific retry guarantee. A verified present record also needs the relevant outstanding-attempt obligations resolved before a stable completion claim. When reconciliation is impossible, it requests a handoff rather than assuming that the action failed safely. There is no general exactly-once guarantee across arbitrary external services.

Figure 5 separates an interrupted attempt from an established failure. A missing response cannot by itself authorize a repeated write.

Retries have explicit limits on attempts, time, and inference. Repair preserves verified outputs whose dependencies and assumptions remain valid. A compensating operation, when available, is itself an observable action with its own failure modes; it is not described as a universal rollback. Pending work resumes only after rechecking permissions, versions, freshness requirements, and the relevant resource condition. iOS background-execution limits mean that a checkpoint is a resumable obligation, not a promise of unattended continuous execution.

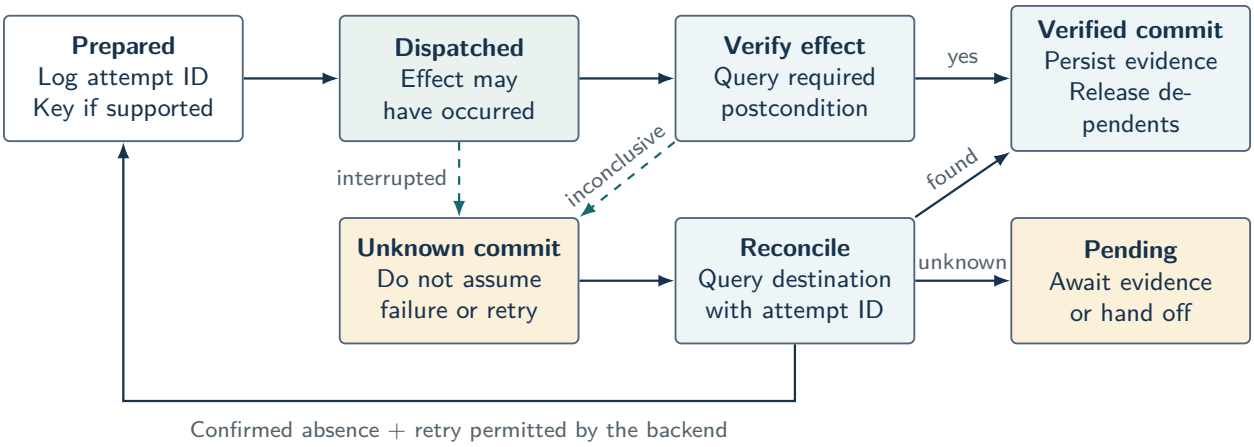


Figure 5: Proposed recovery states for an external side effect. Dashed transitions indicate loss or insufficiency of evidence. A retry requires both an established absence and permission under the backend's retry semantics; services without adequate reconciliation support cannot provide a general exactly-once guarantee.

4.5 Verification remains mandatory

Every required action has an evidence level: an independently queried state, a validated tool receipt, or weaker observational evidence such as a UI change. The runtime records this level and does not silently replace a required state check with a cheaper model judgment. Generated assertions of success are not independent evidence. When only visual evidence exists, the benchmark needs an external oracle or must report a weaker evaluation category.

Content obtained from documents, pages, and tool responses is observation data. It cannot alter the user's authorized destination, expand permissions, or waive a required check. Any proposed change to those fields returns to intent validation. This boundary matters for ordinary extraction errors as well as adversarial content, and should be exercised with synthetic fixtures.

An elementary invariant explains the value of this discipline. If a node is marked committed only after its postcondition is established, and every successor consumes only committed typed outputs, then, by induction over the executed topological order, each consumed output has the declared verification evidence. This does not prove that the oracle is complete, that external state remains unchanged afterward, or that the graph correctly expresses the user's intent. Those remain distinct validation obligations.

4.6 Illustrative offline-to-online workflow

Consider a synthetic receipt already available to the app, with an explicitly specified local archive and remote destination. Extraction and local persistence may be feasible offline. Transmission is not. The runtime can commit the archive operation, record its identifier, and checkpoint the upload obligation. On reconnection it checks whether a prior transmission committed, validates the destination, and resumes only the outstanding action. The user-visible result remains partial until the remote postcondition is observed. This is a design example, not an observed trace or an existing end-to-end capability claim.

5 Implementation and Evidence Boundaries

We inspected selected working-tree files of the existing HiOS iOS application on 22 October 2026. The repository HEAD was `9fb8d526bc6f`; a separate manifest records full content hashes because a commit identifier alone does not identify uncommitted file contents. This is a static source audit. It does not establish successful execution, benchmark accuracy, or measured power behavior.

The device-contract reference added for this study is a separate implementation. It encodes versioned API/web bindings, profile fingerprints, explicit argument rebinding, local/cloud eligibility, and preflight after a verified prefix. Its source, selected scenarios, and trace analyzer are packaged under `experiments/device_contract/`. It makes the central specialization layer executable for synthetic fixtures; it does not implement an iOS resource monitor, a semantic model, or a calibrated cost estimator. Table 2 continues to describe the native application rather than attributing the reference layer to that app.

The planner contains deterministic native-tool paths, on-device model classification and planning, and cloud fallback. The Foundation Models wrapper primarily consumes generated strings; the inspected paths should not be described as a fully typed intent compiler. TaskGraph contains nodes and dependencies, cycle checks, and topological ordering. Its main executor iterates over an ordered node list sequentially. A DAG representation therefore exists, but graph parallelism is not an implemented performance result.

Workflow recording stores tool sequences and arguments. Similarity lookup uses string containment, and prior workflows can inform planning. This is useful scaffolding but does not constitute versioned, contract-checked graph instantiation. Battery level appears in context. The inspected source did not establish a policy driven by thermal state, low-power mode, measured energy, or network-path changes. Fixed offline/online modes and fallback logic should not be presented as continuous resource adaptation.

Verification combines browser predicates, tool receipts, and model judgments. The cloud goal verifier prioritizes negative verdicts, whereas the Foundation Models helper accepts positive substrings such as `ACHIEVED`, including when negated. Native actions can rely on tool-reported outcomes; the headless harness combines completion state with response keywords. These paths do not independently establish all goal predicates. This audit identifies possible false positives without estimating their frequency. Appendix H records the expanded source review, including observation coverage, usage counters, and build-specific capabilities.

Native integration should proceed in dependency order. First, define contracts and independent fixture oracles, and ensure that all compared systems receive the same verdicts. Second, add tracing around model calls, tool attempts, verification, and suspension boundaries. Third, implement typed reuse and a rule controller. Only then should a learned estimator or parallel scheduler be introduced. This order makes it possible to attribute savings to specific mechanisms.

Public Apple APIs expose thermal and low-power state, and Network framework exposes path observations [3, 5]. These are inputs to the proposed application-level controller, not direct controls over the system model's accelerator scheduling. MetricKit supplies aggregate diagnostic and performance information and is not a substitute for a per-decision resource oracle [2]. SDK, OS, model availability, tool registration, and release/debug configuration must be frozen and recorded for each experiment. Newly available model features require a fresh baseline audit rather than a retrospective claim that the original prototype implemented them.

Subsequent bounded verification. Section 6 reports device-contract conformance, graph checks, and the expanded recovery study executed on 25 October. These component results do not validate the adaptive controller, model quality, or resource savings.

Table 2: Implementation status from static inspection. "Present" means identified in source, not experimentally validated.

Component	Present in the inspected application	Work required for this proposal
Intent routing	Deterministic paths, local model, cloud fallback	Typed bindings; ambiguity handling; comparable routing instrumentation
Graph execution	TaskGraph, dependencies, topological sequence	Typed dataflow; guards; conflict domains; durable checkpoints
Workflow reuse	Successful sequence records; text-based lookup	Parameterized templates; contracts; version checks; drift invalidation
Resource policy	User-selected modes; battery context	Thermal, low-power, network and deadline observations; bounded controller
Verification	Tool results and model-based checks	Independent state oracles; explicit evidence levels; unknown-commit recovery
Evaluation	Opt-in headless harness and timing utilities	Controlled fixtures; matched baselines; physical-device energy protocol

6 Executed Component Verification

The executed evidence has three layers: device-specific binding and admission, native graph structure, and recovery of uncertain side effects. These are component studies; none is a full-agent benchmark.

6.1 Device-contract conformance

A new Python reference implements the specialization and preflight layer from Section 3.3. Its fixed workflow binds an explicit synthetic receipt, archives it, and sends it through a registered adapter; selected probes also include a reasoning-boundary marker. Four synthetic profiles represent an API adapter, a foreground-web adapter, a changed API schema with a low-power restriction, and an installation without the required send adapter. They are execution profiles, not measurements of four physical phones. All assume an available local model and configured remote credentials.

Twenty-seven selected scenarios exercise initial constraints and changes after binding: network or service loss, permission revocation, foreground loss, schema and adapter changes, account-scope changes, missing or changed task arguments, local resource limits, egress restrictions, deadline expiry, and an unresolved earlier commit. The workflow preserves its verified archive before the send boundary. A reasoning marker tests route eligibility only: no model is called and no generated output is simulated. The complete scenarios, expected terminal states, and 81 policy traces are included in the local artifact.

The diagnostic controls are *portable replay*, which reuses a previously bound API plan and its old arguments, and *bind once*, which binds current arguments and the initial profile but retains its initial eligibility decisions. The *device contract* policy rebinds incompatible adapters and rechecks live conditions before dispatch. All use the same fixture-side admission checks and final-state verification. The fixture records inadmissible attempted calls even when it refuses their effects; a

Table 3: Device-contract conformance over 27 selected scenarios per policy. These are diagnostic counts, not workload success rates. Ineligible attempts and wrong writes can overlap terminal outcomes. "Other" includes failure, infeasibility, handoff, and clarification.

Policy	Verified complete	Ineligible attempts	Wrong writes	Pending	Other
Portable replay	1	23	3	0	26
Bind once	5	16	0	3	19
Device contract	11	0	0	10	6

separate analysis reconstructs exact goal satisfaction from the recorded writes.

The device-contract policy matches the declared terminal state in all 27 probes: 11 complete, ten remain pending, three require handoff, one is infeasible, one needs clarification, and one reaches its expired deadline. It makes no ineligible dispatch or wrong-payload write. Bind-once completes five and makes 16 ineligible attempts. Portable replay completes one and makes 23 ineligible attempts and three wrong-payload writes. These deliberately limited controls isolate stale bindings and stale eligibility; they are not competitive agent baselines. Nine focused tests pass, including corrupted success claims and acceptance flags. The result establishes conformance of this reference layer under visible state changes, not optimal scheduling, model quality, or device resource savings.

6.2 Native graph structure

The Swift harness checks every four-node directed adjacency matrix on host and iOS Simulator: 65,536 cycle classifications and 8,688 readiness checks agree with a separately implemented oracle. Five malformed-input and completion probes improved from 2/5 to 5/5 after repairing duplicate-ID validation and completion tracking. Appendix A preserves the original failures and the initial recovery pilot.

6.3 Expanded recovery study

The expanded Python reference fixture crosses 14 write conditions, six observation conditions, three idempotency contracts, and three restart boundaries. This gives 756 constructed cases for each policy and query budget. Three policies at query budgets of one, three, and six produce 6,804 traces. These are nested sensitivity runs, not 6,804 independent user tasks. The executor reserves at most three dispatch attempts per case. SQLite records service effects; a file checkpoint reconstructs executor state. No language model, native application, physical phone, or actual network is involved.

The controls isolate recovery decisions. *Acknowledgement retry* accepts an acknowledgement and retries a lost response. *Snapshot retry* verifies the complete payload against current authoritative state, but treats an empty snapshot as permission to retry. *Reconciliation* additionally requires evidence that no outstanding request can still commit. A query can describe current state accurately without establishing this latter condition. The fixture exposes the condition explicitly; a real backend that cannot establish it must remain unresolved or provide another safe retry contract. These controls are reference ablations, not competitive agent baselines.

An analyzer that does not import the executor checks exactly one full-payload match both at executor return and after draining outstanding requests. A success declaration failing either check is *unsafe success*. Duplicates are counted even when the runtime detects them and reports failure.

Table 4: Expanded recovery results at the primary query budget of three. Each row covers the same 756 deterministic conditions. Percentages describe this matrix, not field rates. Duplicate effects can overlap any terminal category; the other four outcome columns partition the cases.

Policy	Verified completion	Unsafe success	Duplicate effects	Pending	Failed
Ack. retry	234 (31.0%)	360 (47.6%)	120 (15.9%)	126	36
Snapshot retry	166 (22.0%)	0 (0.0%)	18 (2.4%)	468	122
Reconciliation	174 (23.0%)	0 (0.0%)	0 (0.0%)	474	108

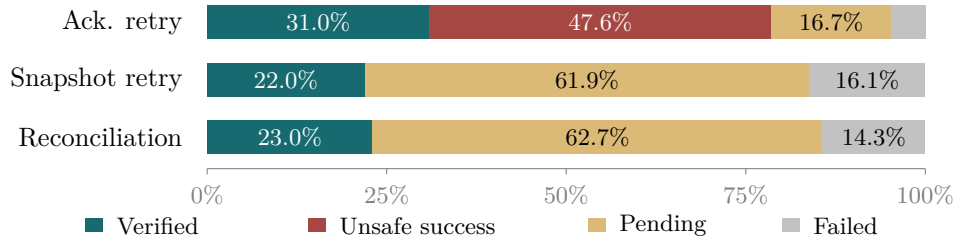


Figure 6: Measured fixture outcomes, generated from the analysis output. The large pending fraction is part of the result: withholding an unverified success claim does not complete the user's task. No confidence interval is attached to this fully enumerated artificial domain.

All assigned cases, including permanent outages, remain in the denominator. Appendix B gives the schedules, budget sensitivity, costs, and per-fault results.

6.4 Findings and costs

At three queries, reconciliation verifies 174/756 cases (23.0%), with zero unsafe successes or duplicates, 474 pending cases (62.7%), and 108 detected failures. Of the pending cases, 132 have a correct settled effect that the runtime cannot verify within its budget. Snapshot retry verifies 166 cases and also avoids unsafe success, but produces 18 duplicate effects (2.4%). Checking a final state can detect damage without preventing it. Reconciliation therefore improves this safety outcome without establishing a large completion advantage over the stronger control.

Acknowledgement retry verifies 234 cases but declares unsafe success in 360 (47.6%); 48 of these appear correct at return and are invalidated by a later commit. It also produces 120 duplicate-effect cases. Its higher completion count is not a matched-quality efficiency result. Reconciliation uses 828 dispatches plus 2,025 queries, compared with 1,332 dispatches and no queries for acknowledgement retry. Query overhead cannot be omitted: the totals are 16.40 and 5.69 operations per verified completion, respectively, under a simple unweighted operation count.

Increasing the query budget from one to three to six raises reconciliation's verified completion from 18 to 174 to 348 cases (2.4%, 23.0%, and 46.0%), while total operations rise from 1,317 to 2,853 to 4,002. At six queries, 240 cases remain pending and 168 fail; observed unsafe-success and duplicate counts remain zero. These results expose a completion–observation trade-off under the stipulated service semantics. They do not measure latency, inference cost, energy, generalization, or full-agent reliability.

7 Validation Targets and Remaining Evaluation

The executed conformance study addresses whether the specified device admission rules are applied at the correct boundaries. It does not answer whether language interpretation is accurate or whether specialization reduces total resource use. The framework therefore separates three validation targets.

Contract conformance. Given explicit arguments and observable conditions, test compatibility, binding, eligibility, and preservation of verified work. The current selected scenarios expose stale-plan and stale-state errors against fixture oracles. Extend this layer with larger graph compositions, independently authored scenarios, adapter faults, and races after preflight. Synthetic device profiles are useful for these checks, provided their limits and all blocked outcomes remain visible. The current reference does not calibrate cost or implement physical-device monitors.

Semantic quality and inference expenditure. Integrate the same specialization layer with real models and supported tools. Compare per-step generative control, graph execution without reuse, static qualified reuse, and device-aware execution on matched tasks, observations, model versions, retry budgets, and state oracles. Hold out workflow families, compositions, and interface versions; freeze the reusable library before final evaluation. Separate cold construction, warm reuse, cross-profile rebinding, and drift. An adapted HybridFlow-style router is useful if its assumptions can be matched to the same action contracts [15].

Count input and output tokens by model and tokenizer, including interpretation, verification, repair, failed attempts, and construction. The primary digital endpoint is a predeclared inference-expenditure measure per verified completion subject to a fixed completion and deadline criterion. A lower count caused by more deferral is not an efficiency result. Report false success, wrong effects, pending work, handoff, and first-useful-action and completion timing. Choose the success margin and final sample size from a feasibility pilot before inspecting held-out comparisons; cluster uncertainty by workflow family and session. A selected conformance matrix is not a substitute for that sampled study.

Physical-device calibration and benefit. Estimate costs separately for each declared hardware, OS, model-interface, adapter, and resource stratum. Re-evaluate calibration when those conditions change. Match the task and verification obligations across physical devices, randomize run order, and retain failed-attempt energy and controller overhead. Synthetic thermal or battery flags can check a policy branch but cannot establish energy savings. Use calibrated measurements or explicitly labeled supported proxies; do not convert token counts or logical fixture ticks into joules or milliseconds.

Appendix C retains the full protocol, external-benchmark options, device conditions, measurement boundaries, and all-attempt accounting. That protocol remains prospective. The immediate research claim is the specified device contract and its bounded reference behavior. Resource efficiency is the next hypothesis to validate, not a result inferred from the conformance counts.

8 Limitations and Open Questions

The device specialization and preflight layer is executable in a synthetic reference, while the complete native framework remains unevaluated. Section 6 establishes selected conformance and bounded graph/recovery findings. Device profiles, compatibility stamps, and admission checks are

established software techniques; their organization here defines the framework, while any broader efficiency advantage needs comparative evidence. A simpler static policy may suffice on stable devices and workflows.

Intent ambiguity remains an upstream problem. Types and postconditions can catch malformed actions or missing effects but cannot establish that a recipient, deadline, or inferred preference matches what the user intended. Clarifications and permissions therefore remain explicit costs and outcomes. A system that asks more questions may improve correctness while increasing elapsed time; both effects must be measured.

Model availability is a simplifying assumption, not a claim of universal device support. A system-managed model can change with the operating system, and its hardware placement and exact accounting may be opaque. Public thermal categories and network observations may be too coarse for an accurate predictor. A policy trained on one phone, one battery condition, or one service can miscalibrate elsewhere. Version pinning, conservative fallback, and per-condition reporting reduce but do not eliminate these problems.

The feasible action surface limits generality. Supported native integrations and app-owned web views do not cover all third-party app workflows. Remote service state may be only partially observable, and some destinations provide neither an idempotency mechanism nor a reliable reconciliation query. In those cases the system cannot promise safe automatic retry. Explicit uncertainty or handoff is a legitimate result, but it reduces autonomous completion.

Template reuse also creates a distributional risk. A workflow may remain syntactically valid after its meaning or surrounding application state changes. Version checks alone cannot detect this. Contract quality, test coverage, and drift signals determine how much reuse is justified, while overly conservative checks can erase its economic benefit. Equation 8 makes this trade-off explicit but does not resolve it.

9 Reproducibility and Research Practice

The local artifact contains the device specializer, 27 selected conformance scenarios, graph snapshots, recovery fixtures, raw traces, separate outcome analysis, and provenance. It is not an end-to-end runtime benchmark. The application-derived Swift snapshots retain their existing MIT notice; licensing for new research code and public release remains to be finalized. The full study must additionally release capability manifests, frozen libraries, baseline configurations, and documented proprietary dependencies.

Experiments should initially use synthetic data and test destinations. A later user study or analysis of non-public human data requires the applicable ethics review before collection. For reproducibility, anonymized aggregate outcomes are insufficient on their own: release the task definitions, inclusion rules, and failure categories needed to reconstruct denominators. Credential values, personal content, and unrelated application files are not research artifacts.

Use of AI assistance. The initial prose throughout this manuscript, its proposed formalization and evaluation protocol, its reference experiment code and analysis scripts, its figures, and its LaTeX assembly were prepared with assistance from OpenAI Codex. The author(s) must review the technical statements, sources, and final wording before submission. No AI tool is listed as an author, and reported pilot counts are computed from executable traces rather than invented by a model. This disclosure identifies drafting assistance rather than implying completed human validation.

10 Conclusion

A device-specific agent framework must preserve the user's intended outcome while specializing how that outcome can be reached. HiOS makes this boundary explicit through device profiles, adapter bindings, compatibility stamps, live admission checks, and verified checkpoints. The reference layer demonstrates the specified behavior under selected capability and state changes; the graph and recovery studies support separate dependency and side-effect obligations. The remaining test is whether this structure lowers all-attempt inference expenditure at matched success when integrated with real models and supported device interfaces. Physical energy benefits require their own calibrated measurements.

References

[1] Apple. Foundation models framework, 2026. URL https://developer.apple.com/foundation-models-framework/. Developer documentation; accessed 22 October 2026.

[2] Apple. MetricKit, 2026. URL https://developer.apple.com/documentation/metrickit. Developer documentation; accessed 22 October 2026.

[3] Apple. NWPathMonitor, 2026. URL https://developer.apple.com/documentation/network/nwpathmonitor. Developer documentation; accessed 22 October 2026.

[4] Apple. Measuring your app's power use with power profiler, 2026. URL https://developer.apple.com/documentation/xcode/measuring-your-app-s-power-use-with-power-profiler. Xcode documentation; accessed 22 October 2026.

[5] Apple. ProcessInfo, 2026. URL https://developer.apple.com/documentation/foundation/processinfo. Thermal state and low-power-mode APIs; accessed 22 October 2026.

[6] Apple. LanguageModelSession.DynamicProfile, 2026. URL https://developer.apple.com/documentation/foundationmodels/languagemodelsession/dynamicprofile. Developer documentation; availability and beta status must be checked against the experimental SDK; accessed 22 October 2026.

[7] Apple. Security of runtime process in iOS, iPadOS, and visionOS, 2026. URL https://support.apple.com/guide/security/sec15bfe098e/web. Apple Platform Security; accessed 22 October 2026.

[8] Apple. Expanding generation with tool calling, 2026. URL https://developer.apple.com/documentation/foundationmodels/expanding-generation-with-tool-calling. Developer documentation; accessed 22 October 2026.

[9] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

[10] Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks, 2024. URL https://arxiv.org/abs/2407.05291.

[11] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. arXiv:2305.05176, 2023. URL https://arxiv.org/abs/2305.05176.

[12] Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Leon Xu, Suzhen Zheng, Hao Fan, Pashmina Cameron, Justin Wagle, and Kazuhito Koishida. CUA-Skill: Develop skills for computer using agent. arXiv:2601.21123v2, 2026. URL https://arxiv.org/abs/2601.21123v2.

[13] Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. Advancing mobile GUI agents: A verifier-driven approach to practical deployment. arXiv:2503.15937v5, 2026. URL https://arxiv.org/abs/2503.15937v5. V-Droid; the arXiv record also lists an accepted-title variant.

[14] Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Massimo Caccia, Alexandre Drouin, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Graham Neubig, Quentin Cappart, Russ Salakhutdinov, and Nicolas Chapados. The browsergym ecosystem for web agent research. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=5298fKGmv3.

[15] Jiangwen Dong, Jiayu Li, Tianhang Zheng, and Wanyu Lin. HybridFlow: Resource-adaptive subtask routing for efficient edge–cloud LLM inference. arXiv:2512.22137v5, 2026. URL https://arxiv.org/abs/2512.22137v5.

[16] Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 11642–11662. PMLR, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

[17] Hongchao Du, Shangyu Wu, Qiao Li, Riwei Pan, Jinheng Li, Youcheng Sun, and Chun Jason Xue. ClawMobile: Rethinking smartphone-native agentic systems. arXiv:2602.22942v2, 2026. URL https://arxiv.org/abs/2602.22942v2.

[18] Amine El hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. Webarena verified: Reliable evaluation for web agents. In *Workshop on Scaling Environments for Agents*, 2025. URL https://openreview.net/forum?id=94tlGxmqkN.

[19] Lawrence Keunho Jang, Mareks Woodside, Geronimo Carom, Andrew Keunwoo Jang, Jing Yu Koh, and Ruslan Salakhutdinov. iOSWorld: A benchmark for personally intelligent phone agents. arXiv:2606.09764, 2026. URL https://arxiv.org/abs/2606.09764.

[20] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. arXiv:2312.04511, 2023. URL https://arxiv.org/abs/2312.04511.

[21] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015. doi: 10.1145/2815400.2815416. URL

https://sigops.org/s/conferences/sosp/2015/current/2015-Monterey/printable/126-lee.pdf.

[22] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steven Y. Ko, Sangeun Oh, and Insik Shin. Explore, select, derive, and recall: Augmenting LLM with human-like memory for mobile task automation. arXiv:2312.03003v3, 2024. URL https://arxiv.org/abs/2312.03003v3.

[23] Maoliang Li, Jiayu Chen, Zihao Zheng, Ziqian Li, Xinhao Sun, Guojie Luo, Chenchen Liu, and Xiang Chen. HeRo: Adaptive orchestration of agentic RAG on heterogeneous mobile SoC. arXiv:2603.01661v2, 2026. URL https://arxiv.org/abs/2603.01661v2.

[24] Aikaterini Maria Panteleaki, Varatheepan Paramanayakam, Spyros Tragoudas, and Iraklis Anagnostopoulos. Carbon-aware routing for function calling in edge–cloud LLM systems. arXiv:2609.13559, 2026. URL https://arxiv.org/abs/2609.13559.

[25] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. arXiv:2405.14573, 2024. URL https://arxiv.org/abs/2405.14573.

[26] Pranay Tummalapalli, Sahil Arayakandy, Ritam Pal, and Kautuk Kundan. LLM inference at the edge: Mobile, NPU, and GPU performance efficiency trade-offs under sustained load. arXiv:2603.23640, 2026. URL https://arxiv.org/abs/2603.23640.

[27] Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-Agent-E: Self-evolving mobile assistant for complex tasks. arXiv:2501.11733, 2025. URL https://arxiv.org/abs/2501.11733.

[28] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A distributed asynchronous reinforcement learning framework for efficient large-scale LLM training, 2025. URL https://arxiv.org/abs/2505.24034v2. Version 2; used as a systems-evaluation methodological comparison.

[29] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. arXiv:2210.03629v3, 2023. URL https://arxiv.org/abs/2210.03629v3. ICLR 2023.

[30] Cheng Zhang, Erhu Feng, Xi Zhao, Yisheng Zhao, Wangbo Gong, Jiahui Sun, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. MobiAgent: A systematic framework for customizable mobile agents. arXiv:2509.00531, 2025. URL https://arxiv.org/abs/2509.00531.

[31] Linghua Zhang. Jev-Mobile: Jev as an executor for mobile GUI agents. arXiv:2609.30186v1, 2026. URL https://arxiv.org/abs/2609.30186v1.

Table 5: Five targeted graph-contract probes, run on both host and simulator. These are regression checks, not estimates of a field failure rate.

Required behavior	Before	After
Reject a missing dependency	Pass	Pass
Reject an unknown dependency owner	Pass	Pass
Reject duplicate node IDs	Fail	Pass
Unknown completion IDs cannot finish a graph	Fail	Pass
Required IDs finish a graph despite unrelated completion input	Fail	Pass

A Exploratory Verification Pilot

This pilot was executed on 25 October 2026. It tests graph structure and a small recovery reference implementation. It is neither a preregistered study nor an end-to-end agent benchmark. Source snapshots, hashes, the complete graph matrix, 72 recovery traces, and an independent outcome-analysis script accompany the local verification artifact.

A.1 Native graph checks

The harness concatenates the application's `Tool.swift`, `TaskNode.swift`, and `TaskGraph.swift` verbatim, adds the required Combine import, and compiles a standalone executable. It does not replace the application's graph algorithm. We used Xcode 27.1 (27A9275), macOS 27.0 (26A5353q), and an iPhone 17 Pro simulator running iOS 26.2 (23C54). The simulator runs the compiled graph code, not the full HiOS application or an on-device language model.

The finite input domain contains every 4×4 binary adjacency matrix: 65,536 directed graphs on four labeled nodes, including self edges. An independently written iterative elimination oracle classifies 543 as acyclic and 64,993 as cyclic. The application's DFS-based classifier agreed on all cases. Every accepted order contained each node exactly once and placed each prerequisite before its dependent. For the 543 acyclic graphs, all 16 completed-node subsets were checked, giving 8,688 readiness cases with no disagreement. Host and simulator produced identical matrix files before and after the patch. This bounded check does not prove correctness for larger graphs, malformed inputs outside the probes, concurrency, or downstream actions.

The original validator did not reject duplicate IDs. The execution-state container also compared set size with node count, allowing unrelated IDs to cause either false completion or failure to recognize completion. The patch rejects duplicate IDs, computes completion from required IDs, and ignores unknown IDs passed to `markCompleted`. Table 5 retains the initial failures rather than reporting only the repaired result. This patch does not replace the separate model-based verification path discussed in the source audit.

A.2 Recovery fixture and controls

A separate Python implementation executes a single, fully specified synthetic receipt-write action. A SQLite service fixture stores the actual effects; a file-backed JSON checkpoint is reread after each dispatch to model lost executor state. This is object reconstruction, not an operating-system crash or a test of power-loss durability. The code is not integrated into the iOS application and uses no language model.

The fixed Cartesian matrix combines six first-dispatch conditions (normal operation, timeout before commit, timeout after commit, an incorrect payload with acknowledgement, acknowledge-

Table 6: Recovery fixture outcomes. Counts cover 36 deterministic cases per policy. These frequencies describe the constructed matrix, not a real-user workload distribution. Duplicate effects are a subset of false success here.

Policy	Verified complete	False success	Duplicate effects	Pending	Failed
Acknowledgement-only retry	15	15	3	0	6
Reconciliation	16	0	0	10	10

ment without a write, and explicit rejection), three query modes (authoritative, one stale empty response followed by authoritative responses, and unavailable), and server idempotency enabled or disabled. This yields 36 cases per policy. Later dispatches operate normally. Both policies have a maximum of two dispatches and three queries. The fixture guarantees no delayed write can commit after an authoritative absence result; real services without this guarantee cannot use that retry rule.

The *acknowledgement-only retry* control accepts acknowledgement as success and retries a timeout. It is an intentionally limited diagnostic ablation, not a competitive ReAct, graph-reuse, or adaptive-routing baseline. The *reconciliation* policy checks the complete effect payload, treats stale absence as inconclusive, retries only after authoritative absence, and remains pending when evidence is unavailable. An independent analyzer reads the fixture rows and requires exactly one matching effect. A runtime success declaration is counted as false success if that state predicate fails. Every matrix case remains in the denominator.

The reconciliation policy made 44 dispatches and 68 queries across its 36 cases; the control made 48 dispatches and no queries. Its zero observed false-success and duplicate counts therefore do not establish lower total cost. Of its ten pending cases, four already contained a correct effect that the executor could not verify. The other ten cases ended in explicit rejection or detected payload mismatch. Eight contract tests passed, including a deliberately corrupted recipient checked by the independent analyzer.

A.3 Interpretation and remaining study

These observations support two narrow statements: the inspected graph methods satisfy the enumerated structural checks after the reported repairs, and the reference reconciliation policy preserves the tested effect predicate under the declared fixture semantics. They do not measure language understanding, graph synthesis, reuse generalization, adaptive routing, token savings, real network latency, physical energy, or thermal behavior. No confidence interval is assigned to a fully enumerated artificial matrix. A subsequent study needs held-out workflow families, real model calls, matched competitive baselines, all-attempt accounting, and external benchmark tasks. Physical-phone measurements remain necessary for the energy and thermal claims in the main protocol.

B Expanded Recovery Methods and Sensitivity

The exploratory version 0.2 protocol and executable fixture are stored separately from the unchanged version 0.1 pilot. The protocol was written before executing this expanded matrix, but was not preregistered and does not define a held-out confirmatory study. The Python 3.14.7 implementation ran on the macOS host. It makes no model calls. Fifteen targeted contract tests exercise delayed commits, expired keys, incorrect payloads, restart boundaries, budget exhaustion,

and deliberately corrupted inputs to the independent analyzer. All passed. Independence here means separate code paths, not external authorship or peer validation.

B.1 Complete condition matrix

The 14 write schedules are normal operation; dropped responses before commit once, twice, or persistently; dropped responses after commit once or persistently; acknowledgement without a write once or persistently; an incorrect amount, destination, or currency; explicit rejection; a delayed commit with a lost response; and an acknowledgement before a delayed commit. Conditions described as "once," payload errors, rejection, and delayed commits affect the first actual service dispatch; later dispatches are normal. The persistent schedules affect every dispatch.

The six query schedules are immediately authoritative; one stale empty response; three stale empty responses; one unavailable response; three unavailable responses; and permanent unavailability. The finite stale/unavailable schedules become authoritative afterward. Stale absence never authorizes a retry for either verifying policy. An authoritative snapshot reports current rows separately from a *settled* flag, which certifies that no outstanding request for the action key can later commit. There are no external writers in this fixture.

Server idempotency has three settings: absent, durable for the entire run, and expiring two logical ticks after an accepted commit. Dispatches and queries each advance the logical clock by one tick. A delayed write commits three ticks after its dispatch. Deduplication is checked at commit, with an expiring key valid strictly before its expiry tick. These are declared sensitivity parameters, not estimates of any real service's retention period or network latency. The audit drains outstanding writes after executor return; these final ticks do not consume controller queries and do not represent measured response time.

The restart factor is no interruption, interruption after checkpointing but before the first dispatch, or interruption after dispatch before the response can update the checkpoint. The first interruption consumes a reservation but performs no service write. Each interrupted continuation rereads its checkpoint. No actual operating-system kill, power failure, storage corruption, concurrent writer, or background iOS execution is tested. Three reservation attempts and the same query budget apply to all policies. Query budgets of one, three, and six are reported separately; the acknowledgement-only control does not use queries and consequently repeats identical behavior across these budget settings.

B.2 Outcomes by write condition

Table 7 retains every write stratum. Each row contains 54 combinations of query schedule, idempotency, and restart per policy at three queries. The delayed-write rows illustrate why an accurate current snapshot alone does not authorize a non-idempotent retry. Permanent outages and exhausted observation budgets remain pending rather than being excluded from success denominators.

With immediately authoritative queries, reconciliation verifies 66/126 cases (52.4%). With one stale or unavailable response it verifies 54/126 (42.9%) in each stratum. Three initial stale responses, three initial unavailable responses, and permanent unavailability each produce zero verified completions within a three-query budget. Each of these strata has 123 pending cases and three observed rejections. Thus the pooled 23.0% completion rate depends strongly on the chosen uniform weighting of observation failures. It must not be interpreted as deployment accuracy.

Table 7: Per-write-condition outcomes at three queries; 54 cases per row and policy. Reconciliation has zero unsafe successes and duplicates in every row. Snapshot duplicates and acknowledgement unsafe successes are shown as additional diagnostics; complete outcomes for all policies are in the CSV artifact.

Write condition	Reconciliation			Snapshot	Ack. retry
	Verified	Pending	Failed	duplicates	unsafe
Normal	27	27	0	0	6
Drop before, once	21	33	0	0	0
Drop before, twice	6	48	0	0	0
Drop before, persistent	0	54	0	0	0
Drop after, once	27	27	0	0	18
Drop after, persistent	27	27	0	0	0
Ack. without write, once	21	33	0	0	36
Ack. without write, persistent	0	54	0	0	54
Wrong amount	0	27	27	0	54
Wrong destination	0	27	27	0	54
Wrong currency	0	27	27	0	54
Explicit rejection	9	18	27	0	0
Delayed commit	18	36	0	9	36
Ack. before commit	18	36	0	9	48

Table 8: All-attempt operation counts at three queries. The per-completion denominator is verified completions, while the numerator includes operations from every assigned case. These counts are not measured computational cost.

Policy	Dispatches	Queries	Total operations	Per case	Per verified completion
Ack. retry	1,332	0	1,332	1.76	5.69
Snapshot retry	882	2,013	2,895	3.83	17.44
Reconciliation	828	2,025	2,853	3.77	16.40

B.3 Resource accounting and budget sensitivity

Table 8 counts all actual dispatches and queries, including pending and failed cases. Dispatch reservations are logged separately; a reservation interrupted before dispatch is not counted as a network operation. One query and one dispatch are given equal weight only for this transparent operation-count diagnostic. Their real latency, price, and energy need not be equal. The analysis supplies no basis for converting these counts to tokens, milliseconds, or joules.

At six queries, snapshot retry verifies 338 cases, with 228 pending, 190 detected failures, and 22 duplicate-effect cases; it declares no unsafe success. Reconciliation verifies ten additional cases, avoids these 22 duplicate effects, and uses 4,002 total operations compared with 4,038. This is a conditional fixture result, not evidence that reconciliation is universally cheaper. The strict settlement contract, fixed delay, bounded fault schedules, single action, and known payload limit external validity. Workflow composition, real API behavior, human-intent ambiguity, and inference-based verification remain outside the executed study.

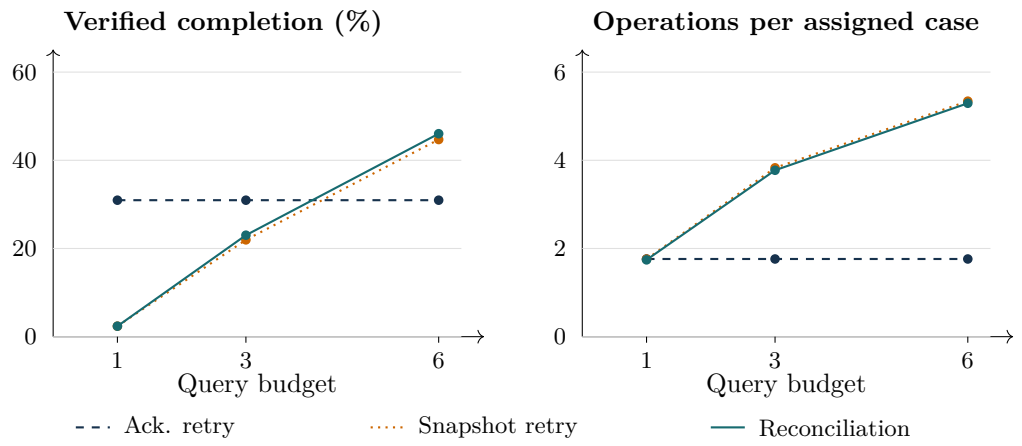


Figure 7: Sensitivity to the query budget over the same 756 conditions per policy. Lines connect the three evaluated budgets; they do not imply a fitted continuous response. Costs include every case. The left panel alone would hide the observation overhead shown on the right.

B.4 Reproduction

From an extracted verification artifact, run the following commands. The runner and analyzer use only the Python standard library. Rendering emits vector TikZ figures and LaTeX tables from the resulting counts; compilation additionally needs the manuscript's LaTeX packages.

```
python3 -m unittest discover -s experiments/expanded -v
python3 experiments/expanded/recovery.py --out rerun
python3 experiments/expanded/analyze.py --out rerun
python3 experiments/expanded/render.py --results rerun --paper rendered
```

The artifact includes the protocol, full traces, one outcome row per condition and policy, factor-wise summaries, budget summaries, environment information, and file hashes. No confidence intervals or significance tests are reported for this fully enumerated artificial domain. A future sample of workflows must use family-aware uncertainty rather than treating these factorial or nested budget runs as independent observations.

C Detailed Full-System Evaluation Protocol

Figure 8 summarizes the proposed digital study. As a methodological comparison, LlamaRL separates execution diagrams and system design from matched efficiency comparisons, quality checks, and component ablations [28]. Its distributed RL-training setting differs from mobile action execution; we adopt this separation of mechanisms and evidence, not its training algorithm or reported speedups.

C.1 Questions and hypotheses

The evaluation must answer four questions. RQ1 asks whether typed graph execution and qualified reuse reduce inference and end-to-end cost at comparable task success. RQ2 asks whether adaptation improves outcomes under network changes, low battery, and thermal pressure beyond a static local-first policy. RQ3 asks whether verification and recovery prevent false success and duplicate

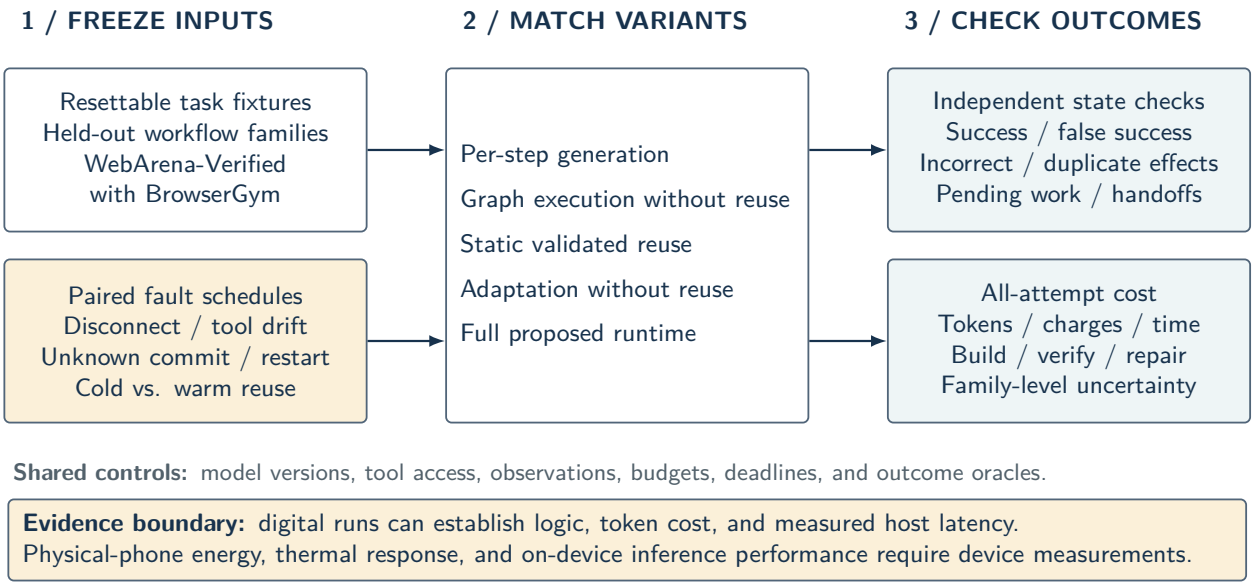


Figure 8: Planned evaluation flow, with all results pending. The static and adaptive variants isolate reuse and resource adaptation under shared controls. The accounting includes failed attempts and construction, validation, and recovery costs. Digital benchmarks support a narrower evidence claim than a physical-phone deployment study.

side effects. RQ4 asks how much workload repetition is needed before construction, lookup, and validation amortize.

The first, digital study will use all-attempt inference expenditure per verified completion as its primary resource endpoint, with model-specific token counts and charges kept separate, subject to a preregistered success non-inferiority criterion and deadline-completion threshold. The exact endpoint, margin, and analysis must be frozen before the held-out comparison. A subsequent physical-device study tests energy per verified completion under the same quality constraints. Token savings alone cannot establish energy savings. If energy cannot be calibrated, retain supported measurement units rather than treating a relative power score as joules. Neither stage has yet been executed.

C.2 Task population and splits

Begin with a feasibility pilot of roughly 20–30 workflow templates. This is a scoping step, not a statistically sufficient final benchmark. Tasks should include local record operations, extraction into local storage, application-service interactions, supported web flows, mixed local/remote dependencies, and recovery after interruption. Each instance has seeded initial state, an explicit goal, a permitted action set, an independent final-state oracle, and a teardown procedure. Synthetic accounts and fixture data allow state-changing operations to be evaluated without involving real recipients.

Partition workflows into disjoint families for template construction, policy calibration, and final evaluation. Hold out paraphrases and arguments, but also hold out compositions and interface versions; paraphrase-only separation is inadequate for testing generalization. Fix the skill library before a conventional test run. A separate continual-learning experiment may permit growth, but

its order, reset policy, and construction costs must be reported. Cold and warm caches are separate conditions, not pooled without explanation.

Use AndroidWorld and iOSWorld as methodological references and possible supplementary environments [19, 25]. Report any ported or restricted subset explicitly. A simulator experiment can test task logic; it cannot establish physical-phone power savings or ordinary application access to privileged benchmark controls. The primary deployment study uses only interfaces the evaluated iOS build can actually invoke.

C.3 Matched baselines and ablations

All main comparisons use the same permitted tool registry, action schemas, oracle, model versions, context access, deadlines, and retry budgets. The following baselines separate mechanisms:

1. **Per-step generative control.** A ReAct-style or native structured-tool loop, with fixed-local and fixed-cloud variants where feasible.
2. **Static local-first routing.** The current architecture's routing principle, evaluated with the common oracle and instrumentation. Unmodified prototype behavior is reported separately if retained.
3. **Graph execution without reuse.** Compile each request and execute a validated graph using fixed routing.
4. **Static validated reuse.** Use the same templates and contracts as the proposed system but disable resource adaptation.
5. **Rule-based adaptation.** Enable resource rules and bounded cost tables over the same actions and templates.
6. **Learned adaptation, if implemented.** Replace the rule selector with a calibrated estimator while keeping the remaining system unchanged.

An adapted HybridFlow-style router is a valuable additional control if its decision assumptions can be matched to the mobile action setting [15]. It must be labeled as an adaptation, not a reproduction of published results. Ablations remove reuse, resource signals, continuation-cost estimates, and adaptive inference placement one at a time. Verification remains active in deployed comparisons; an offline replay analysis can estimate its overhead without authorizing unchecked side effects.

Digital-first evaluation. Before physical-device measurements, we propose an entirely digital stage that isolates contract-checked reuse and adaptive inference. Resettable service fixtures will test held-out workflow compositions, interface drift, and interrupted writes against independent state oracles. External browser evaluation will use WebArena-Verified through BrowserGym, with WorkArena++ providing a compositional extension when instance access is available [10, 14, 16, 18]. Optional tool-coordination experiments will use an explicitly pinned τ-bench release [9]; current and legacy benchmark frames will not be pooled. We will compare per-step generation, graph execution without reuse, fixed validated reuse, and adaptive variants under matched observations and budgets, reporting verified success, all-attempt token cost per completion, latency, and duplicate side effects. iOSWorld and AndroidWorld may provide simulator transfer checks, but synthetic resource signals and simulator timings cannot establish iPhone battery or thermal benefits. These full-system comparisons remain pending. The narrower verification pilot in Appendix A is a separate preliminary result; physical energy measurements constitute a subsequent validation stage.

C.4 Devices and controlled conditions

Use at least two supported physical iPhone generations if available, with exact device model, OS build, model version or exposed identifier, app revision, and battery health recorded. This is a planned target; no device inventory is asserted in this manuscript. Fix brightness, display state, audio state, background load, account fixtures, and starting battery range. Record whether low-power mode is enabled independently of battery percentage. Separate cold model loading from warm steady-state operation.

Network conditions include stable connectivity, controlled latency and bandwidth limits, disconnection before a task, and interruption after a remote side effect may have committed. Record actual observed path and service behavior rather than assuming the configured emulator profile was achieved. Resource tests include nominal operation and sustained workloads that reach documented thermal categories. Ambient conditions, cool-down criteria, and ordering are recorded. Battery depletion and thermal history make simple sequential baseline runs confounded; randomized paired blocks and repeated sessions reduce this bias.

Token throughput under sustained load can differ from short-run behavior, reinforcing the need for long enough measurement windows [26]. The proposed study does not infer a universal device ranking from one model or one prompt. A separate controller-only microbenchmark measures monitoring, lookup, schema validation, and policy-decision overhead.

C.5 Energy and timing instrumentation

Record monotonic timestamps for request receipt, first useful action, each model call, each tool attempt, each verification, and terminal outcome. Human waiting is retained as a separate component; it is neither silently charged as model compute nor omitted from user-facing elapsed time. Include decoding, retrieval, OCR where used, screen activity, transport, verification, and retries within the declared device measurement boundary.

Apple's Power Profiler supports app power investigation, but its measurements need careful interpretation: charging can invalidate the reported battery-discharge measure, device-relative impact scores are not cross-device energy units, and profiling connections can perturb activity [4]. Use controlled unplugged runs and matched idle windows where appropriate. Report the tool, sampling interval, measured units, uncertainty, and whether results are device-total or attributed to the app. A calibrated external setup can support joule estimates if its charging and device-state effects are characterized; otherwise retain the supported proxy units. Battery percentage alone is too coarse for a short individual task. MetricKit is supplementary aggregate evidence, not the primary per-task measurement [2].

C.6 Outcomes and statistical analysis

The primary correctness outcome is satisfaction of all required terminal predicates within the declared horizon. Also report incorrect writes, duplicate side effects, false-success declarations, partial completion, pending work, and handoffs. Feasible and infeasible task strata use a predeclared classification; a difficult task cannot be relabeled infeasible after a failure. For interrupted tasks, report both success before the original deadline and eventual success after a separately bounded recovery window.

Let N be the number of assigned attempts, S_i indicate verified success, and E_i the measured

energy for attempt i. Report both

$$\overline{E}_{\text{attempt}} = \frac{\sum_{i=1}^{N} E_i}{N}, \qquad E_{\text{completion}} = \frac{\sum_{i=1}^{N} E_i}{\sum_{i=1}^{N} S_i}. \tag{9}$$

The latter includes resources spent on failed attempts; it is undefined if no task succeeds. Apply analogous all-attempt accounting to tokens and charges. Report p50 and p95 time to first useful action, completion-time distributions, and deadline-success curves. Successful-only latency may be included as a diagnostic but must not replace the complete outcome distribution.

Estimate paired differences using matched task instances and randomized run blocks. Cluster uncertainty by workflow family and device/session where applicable, so that repeated paraphrases or runs are not treated as independent tasks. Report confidence intervals, per-device results, and heterogeneity across network and thermal strata. Choose the success non-inferiority margin, the primary resource endpoint, multiplicity handling, and the final sample size after a pilot variance estimate but before inspecting final comparisons. A failure to reject a difference is not evidence of equivalence. A pooled average must not hide failures in the offline or sustained-load conditions the system is meant to handle.

C.7 Result reporting template

No experiments measuring the complete controller's resource benefits have been run for this manuscript. The device-contract conformance study tests a narrower eligibility and rebinding layer. Table 9 in Appendix D is an explicitly empty reporting template; its dashes are not zeros, measured ties, or omitted negative results. The current evidence consists of the source audit, targeted related-work analysis, and the component studies in Section 6; none supplies a measurement of the proposed controller's resource benefit.

A favorable outcome would show a resource reduction with the preregistered completion criterion satisfied, including the cost of failed attempts and controller overhead. A narrower outcome—for example, savings only for repeated workflows on stable networks—would support a narrower claim. If the rule policy matches a learned policy, the simpler controller is preferable. If verification erases token savings or resource adaptation degrades completion under drift, that negative result should determine the paper's conclusion rather than be excluded from the workload.

D Pending Result Reporting Template

This table is an unmeasured reporting template. It contains no benchmark results.

E Example Action Contract

This illustrative schema shows an action after argument binding. It is a proposed representation, not the current Swift serialization. The output reference is resolved only after the producing node has committed. The example does not establish an implemented remote service.

```
{
  "id": "send_total",
  "action": "fixture.message.send",
  "schema_version": 1,
  "inputs": {
    "destination": "test-team",
    "total": {"from": "extract.total"}
```

Table 9: Pending evaluation. All entries are unmeasured. Energy units must be fixed by the validated instrumentation before filling this table. Success must include uncertainty; local and cloud tokens must be broken out in the final report.

System	Success (%)	Energy/ completion	Tokens/ attempt	Deadline success (%)	False success (%)
Per-step control	—	—	—	—	—
Static local-first	—	—	—	—	—
Graph, no reuse	—	—	—	—	—
Static validated reuse	—	—	—	—	—
Rule-based adaptation	—	—	—	—	—
Learned policy (conditional)	—	—	—	—	—

```
  },
  "requires": ["network", "test-account"],
  "preconditions": ["destination_confirmed"],
  "postconditions": ["message_found_by_key"],
  "effect": "remote_write",
  "retry": {
    "mode": "reconcile_before_retry",
    "max_attempts": 2,
    "idempotency_key": "task-id/node-id"
  },
  "conflict_domain": "test-team-outbox"
}
```

F Trace and Outcome Requirements

Each trace should retain a task and instance identifier, workflow family, device/OS/build identifiers, model identifiers, tool-registry version, skill-library version, and assignment to baseline and condition. Each event records monotonic time, the active node and attempt, capability decisions, resource observations, inference location, token-count provenance, tool outcome, verification evidence, and checkpoint state. Sensitive content can be replaced with fixture identifiers; necessary scientific fields should not be dropped silently.

The task outcome record distinguishes *complete*, *failed*, *infeasible*, *pending*, and *handed off*. It also retains the separate oracle verdict and whether the runtime itself claimed success. This supports direct measurement of false success. A successful retry contributes its entire attempt history to costs. A lost trace is a missing observation whose cause and frequency must be reported, not a failed or successful task by default.

G Source Audit Map

The audit uses paths relative to the iOS repository. A machine-readable manifest in the manuscript project records full SHA-256 hashes and the observed repository HEAD. The paths identify evidence for the implementation discussion; they do not constitute a public source release.

Planner. `HiOS/Services/HiOS/HiOSPlannerService.swift`: native classification, deterministic paths, model planning, and fallback.

Executor. `HiOS/Services/HiOS/HiOSAgentLoop.swift`: workflow lookup, topological execution, verification paths, and workflow recording.

Graph types. `HiOS/Services/HiOS/TaskGraph.swift` and `HiOS/Services/HiOS/TaskNode.swift`: dependencies, node fields, and verification criteria.

Local model. `HiOS/Services/FoundationModelService.swift`: generation wrapper and text-based verification helper.

Memory. `HiOS/Services/WorkflowManager.swift` and `HiOS/Models/WorkflowTemplate.swift`: stored tool sequences and text-based matching.

Context. `HiOS/Services/SystemContextService.swift`: device context including battery level.

Evaluation scaffold. `HiOSTests/HeadlessAgentRunnerTests.swift`: opt-in headless execution and response-based pass criteria.

H Additional Project Context

A broader folder review on 25 October 2026 compared architecture diagrams, browser-harness proposals, improvement logs, pitch material, and the working-tree source. A separate manifest records the technical files inspected. These materials explain the design's provenance; historical estimates and product demonstrations are not substituted for experimental evidence.

Observation and verification are partially implemented. `ObservationEngine` captures browser URL, title, visible text, and interactive elements; the inspected path caps text at 2,000 characters and elements at 100. Its native-state observer returns empty collections. `GoalVerifier` implements browser predicates as well as model-based checks, and its cloud goal-verdict parser gives negative words precedence. This is distinct from the positive-substring issue in the Foundation Models helper. The next experiment therefore needs action-specific native or service-state oracles, rather than assuming either that verification is absent or that the current observer covers every effect.

Recovery and concurrency have narrower scope than the proposal. `RecoveryPlanner` already selects retries, waits, alternative selectors, and replanning. These mechanisms do not establish persisted unknown-commit reconciliation. The graph executor remains a sequential topological loop. Separate conversation loops can be cloned while sharing services, including the browser; this does not demonstrate parallel graph scheduling or isolation of conflicting browser actions. The April parallel-execution analysis is a proposal, not a performance result.

Accounting exists but needs complete coverage. Provider usage events, planner input/output counters, loop-level aggregation, and a CloudKit logging helper are present. The inspected aggregation copies planner counts; it does not establish coverage of every classification, verification, recovery, fallback, or action-internal model call. Existing model-family price constants also require a pinned, dated rate source before reporting monetary cost. Token/latency budgets in the older flowchart are explicitly estimates and remain excluded from the results tables.

Capability manifests must follow the evaluated build. `AppShortcutsProvider` is guarded by `DEBUG`. The tool registry also gates some contact, maps, messaging, and payment tools, while retaining other native and browser facilities. Thus a Debug test's available actions cannot be assumed to match a Release build. The benchmark must freeze registration, permissions, and direct-service access as well as the source revision. Historical App Store readiness notes concern distribution and are not scholarly-publication evidence.

Roadmaps remain separate from observed mechanisms. The May browser-harness documents explicitly propose domain-skill files, screenshot-based verification, and learned playbooks. A screenshot helper alone does not establish that proposed verification pipeline. Similarly, external-account orchestration notes and pitch slides supply product goals, not measurements or universal website-access guarantees. The harness slides' separation of action from checking is consistent with the pilot's independent fixture oracle, but their broader reliability and efficiency claims require their own evaluation.

I From Architecture Diagrams to Executable Claims

The existing architecture documents communicate an intent–plan–execute–verify loop. That loop is a useful conceptual basis, but a diagram alone does not establish typed contracts, parallel scheduling, or adaptation to changing resources. Older flow descriptions and newer source paths are not fully aligned: a local helper's historical role differs from current native routing, and a graph can be executed sequentially. For this manuscript, source inspection determines implementation status; the new runtime diagram describes the proposed design. These two evidence levels should remain separate as the application evolves.