MOBILESKILLBENCH: AUTOMATICALLY CONSTRUCTING MEMORY-DEPENDENT MOBILE-AGENT TASKS FROM REUSABLE SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Personal mobile assistance requires agents to combine reusable operations with user information that determines which actions are appropriate. Evaluating this ability requires tasks in which personal records influence the workflow and its expected state. Existing work provides interactive environments, personalized tasks, and structured task synthesis, but constructing the dependencies between operations and user records remains a separate design problem. We present MOBILESKILLBENCH, a framework for constructing mobile-agent tasks from raw skill documents and de-identified user memories. A Skill–Memory Graph represents operation dependencies and the records that supply inputs, initialize application data, or constrain valid results. Joint Skill-Memory Search (JSMS) builds candidate subgraphs containing skill workflows and records from one user, then balances candidate quality against repeated use of the same graph resources. Auditable Task Compilation translates each selected subgraph into an instruction, initial state, expected state, and evaluator, with source evidence linking these components. Controlled recompilation checks whether removing a record changes task requirements or leaves essential information unresolved. The resulting MOBILESKILLBENCH comprises 281 tasks that evaluate LLM agents' ability to use user memory to complete complex, multi-step tasks. Across eight current LLM agents, the best model achieves only 28.8% strict success and 67.9% Completion, with strict success falling to 10.9% on multi-app tasks. These results reveal substantial room for improvement in personalized mobile assistance.

1 INTRODUCTION

Large language model (LLM) agents are becoming part of everyday mobile assistance, enabling users to delegate tasks across applications through natural-language requests. By August 2026, Google's Gemini app had surpassed one billion monthly users across platforms and supported automation across over 40 Android apps (Google, 2026). Apple likewise introduced Siri AI with personal-context understanding and cross-app actions (Apple, 2026). These developments place mobile agents in a setting where their decisions can affect users' purchases, schedules, and personal communications. Reliable assistance therefore requires both the ability to operate applications and the ability to use the right personal information. Evaluating these abilities together is increasingly important as users entrust mobile assistants with everyday tasks.

A growing body of mobile-agent benchmarks has made progress toward measuring these systems. ANDROIDWORLD (Rawles et al., 2024), ANDROIDLAB (Xu et al., 2025), MOBILEWORLD (Kong et al., 2025), MOBILEGYM (Wu et al., 2026), KNOWU-BENCH (Chen et al., 2026), and MYPHONEBENCH (Tang et al., 2026) provide increasingly diverse environments, task suites, app coverage, and evaluation protocols. These benchmarks are essential for measuring whether an LLM agent can execute a specified task, interact across applications, or infer preferences.

However, execution-focused evaluation can overlook conflicts with user habits and prior context when actions are not explicitly linked to memory. For the request in Figure 1, *Reschedule tomorrow's team meeting and send my talking points*, moving an event and sending a message may satisfy action-based checks while violating the user's focus-time preference or using the wrong talking

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

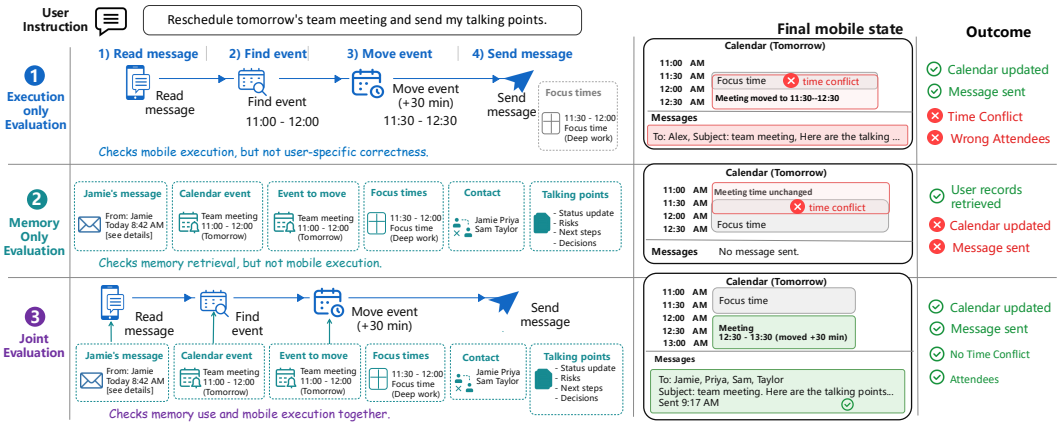


Figure 1: Running example contrasting three evaluation views of the same scheduling request: execution-only, memory-only, and joint evaluation (ours).

points. Memory-focused benchmarks such as MobileMem instead emphasize retaining, retrieving, and reasoning over personal experiences, with question-answering evaluations leaving complex execution largely untested. Recalling the correct meeting and notes does not demonstrate that an agent can coordinate the calendar update and notification under the user's constraints. The challenge is therefore to automatically construct tasks that require both memory understanding and challenging multi-step execution, with success criteria that verify their joint correctness.

To address this challenge, we introduce MOBILESKILLBENCH, which automatically constructs multi-step tasks from reusable mobile skills and de-identified user memories. The key idea is a Skill–Memory Graph (SMG) that captures both dependencies among operations and their connections to personal records. This structure allows task generation to compose operations whose correct execution depends on memory. In Figure 1, the graph links rescheduling to the intended meeting and focus-time preference, and notification to attendee contacts and the talking-points note. These links make the records determine which event to update, which times are valid, and what to send to whom. A successful task must therefore satisfy the user's constraints across the entire workflow. Briefly, our pipeline has three stages: Skill–Memory Graph induction extracts operations, user records, and their dependencies from raw skill documents and memories; Joint Skill-Memory Search (JSMS) selects compatible skill workflows and records from the same user to form task subgraphs; and Auditable Task Compilation converts each subgraph into an instruction, initial state, expected result, and evaluator, with record-removal checks to verify memory dependence. The resulting benchmark contains 281 tasks. Even state-of-the-art LLM agents struggle to use user memory to complete these complex, multi-step tasks, highlighting substantial room for improvement in personalized mobile assistance. In summary, our contributions include:

- We introduce MOBILESKILLBENCH, an automatic task-construction framework and benchmark that connects user memory with complex, multi-step mobile execution, making personal records part of both task requirements and success criteria.
- We propose a Skill–Memory Graph and joint search method that compose reusable operations with the user records they depend on. Auditable compilation preserves these dependencies in task components and tests memory necessity through controlled record removal.
- Experiments on our benchmark reveal that even existing LLMs struggle to use memory to complete complex mobile workflows, exposing a substantial gap in reliable personalized assistance.

2 RELATED WORK

Mobile execution benchmarks. ANDROIDWORLD generates parameterized tasks with initialization and state-based success checks; ANDROIDLAB provides a systematic Android evaluation setting (Rawles et al., 2024; Xu et al., 2025). MOBILEWORLD extends evaluation to agent–user interaction and tool access (Kong et al., 2025). These environments establish how mobile actions can

be executed and scored. MOBILESKILLBENCH addresses task construction upstream of execution, selecting operation dependencies and user records together.

Personalized mobile and personal-data benchmarks. PSPA-BENCH evaluates personalized GUI behavior, while KNOWU-BENCH and FINGERTIP study personalized and proactive assistance (Nie et al., 2026; Chen et al., 2026; Yang et al., 2025). KNOWU-BENCH includes preference elicitation during interaction; FINGERTIP includes personalized execution trajectories. MCP-PERSONA uses simulated personal applications, and MOBILEMEM studies long-term mobile experiences (Wang et al., 2026; Deng et al., 2026). Personalization and execution thus already overlap in prior work. Our distinction concerns construction: selected records are explicitly connected to operation inputs or constraints, allowing their role in a generated task to be inspected.

Agent skills and agent memory. SKILLSBENCH measures the effect of reusable skills on task solving, and SKILLDAG supports skill selection through typed inter-skill relationships (Li et al., 2026; Bai et al., 2026). MEMARENA evaluates personal memory assistance over on-device histories (Zhang & Ma, 2026). These studies motivate structured procedural knowledge and persistent personal information. Here, raw documents and records are construction inputs: their extracted relationships determine which operations and memories can form a task.

Structured task synthesis. AGENTSYNTH scales computer-use task generation, SKILLSYNTH composes terminal tasks through skill graphs, and SYNTHAGENT combines synthetic tasks, simulated environments, and rubric-based rewards (Xie et al., 2025; Fan et al., 2026; Lyu et al., 2026). The significant distinction is what the intermediate structure represents and which dependencies it preserves during generation. MOBILESKILLBENCH extends this construction perspective to dependencies between mobile operations and persistent user records.

3 PRELIMINARY

3.1 MOBILE-AGENT TASK INTERFACE

Mobile-Agent Task. A mobile-agent task tests whether an agent can complete a user request in a mobile environment. Formally, a task is $t = (q, \sigma^0, \sigma^*, \varepsilon)$, where q is the instruction, σ^0 is the initial state, σ^* is the expected state, and ε is the evaluator. At interaction index k, the agent observes the exposed part of σ^k, acts, and eventually returns a final state σ' and, when requested, an answer. The execution trajectory is the recorded sequence of tool calls and returned observations. The expected state specifies task-relevant conditions, including required answer content, rather than an exact copy of the whole system. The evaluator checks the trajectory, final state, and answer against q and these conditions. Both σ^* and ε remain hidden; user records are accessible only through the environment or its permitted memory interface.

Skills. Raw skill documents summarize reusable experience for completing mobile-assistant operations. Reusable skills can improve task solving, and typed skill graphs can support skill selection (Li et al., 2026; Bai et al., 2026). The document set is $\mathcal{D}_S$, where each $d_S \in \mathcal{D}_S$ may contain an operation description, execution traces, app or API notes, and example task templates. The documents summarize real mobile-assistant service workflows. See Appendix B for details.

Memories. Raw memory data provide user-specific values, preferences, and constraints. The record set is $\mathcal{D}_M$, with each record written as $d_M = (u, \mathcal{X}, \lambda)$. Here, u is a stable alias identifying the user whose records are grouped together, $\mathcal{X}$ is de-identified content such as a message, calendar event, note, or contact, and λ is its associated time when available. Direct personal identifiers in the real user data are removed or replaced while retaining task-relevant relationships.

3.2 PROBLEM FORMULATION

Given $\mathcal{D}_S$ and $\mathcal{D}_M$, our goal is to construct a set of mobile-agent tasks $\mathcal{T}$ from reusable operations and user records. We represent each task as $t = (q, \sigma^0, \sigma^*, \varepsilon)$: q is the instruction shown to the agent, σ^0 is the starting state, σ^* describes the task-relevant conditions that should hold after execution, and ε specifies how those conditions are checked. A valid task must describe a coherent workflow and a checkable outcome; it is memory-dependent when a required value or constraint comes from records belonging to one user, identified by u.

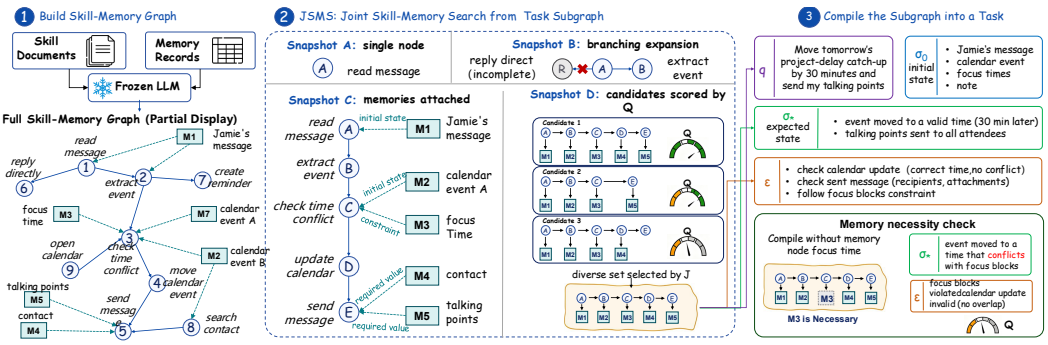


Figure 2: Overview of construction. The full SMG contains alternative skills and memories. JSMS grows task subgraphs from single skill nodes, attaches relevant memories. The memory necessity check removes a record and recompiles the task under the same instruction.

4 Construction of MobileSkillBench

The construction proceeds in three steps (Figure 2). First, it extracts skill nodes, memory nodes, and their dependencies from $\mathcal{D}_S$ and $\mathcal{D}_M$ to form $\mathcal{G}$. Then, it searches $\mathcal{G}$ for compatible task subgraphs and selects a set $\mathbb{G}^*$. Finally, it compiles each selected $\mathcal{G}^* \in \mathbb{G}^*$ into a task $t = (q, \sigma^0, \sigma^*, \varepsilon)$.

4.1 Stage I: Skill–Memory Graph Induction

This stage turns raw skill documents and raw memory records into a Skill–Memory Graph:

Definition 1 (Skill–Memory Graph). *A Skill–Memory Graph (SMG) is a directed typed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{R})$, where $\mathcal{V} = \mathcal{V}_S \cup \mathcal{V}_M$ is the node set, $\mathcal{V}_S$ contains skill nodes, $\mathcal{V}_M$ contains memory nodes, and $\mathcal{R}$ is the set of edge types. An edge is $e = (v_1, v_2, r, \xi_e, w_e)$, where $v_1, v_2 \in \mathcal{V}$ are its endpoints, $r \in \mathcal{R}$ is its edge type, ξ_e is the source evidence supporting the edge, and $w_e \in [0, 1]$ is a relation strength derived from that evidence.*

Skill Nodes. Let f_θ denote the frozen LLM, with fixed parameters θ, used for node extraction and task compilation. Each document $d_S \in \mathcal{D}_S$ yields a skill node $v_S \in \mathcal{V}_S$ described by $(\mathcal{I}, \mathcal{O}, \mathcal{C}, \delta)$. The sets $\mathcal{I}$ and $\mathcal{O}$ describe input and output slots using consistent names and types; $\mathcal{C}$ describes states permitting execution, and δ describes the state change. If $\sigma^k \in \mathcal{C}$ and α assigns valid input values, the described operation gives $\sigma^{k+1} = \delta(\sigma^k; \alpha)$ and may return values for $\mathcal{O}$. For a calendar update, they specify the new time, target entry, editable initial entry, and updated time. They describe operations rather than execution code.

Memory Nodes. Each record $d_M = (u, \mathcal{X}, \lambda) \in \mathcal{D}_M$ yields a memory node $v_M \in \mathcal{V}_M$ described by $(u, \mathcal{F}, \lambda)$. The LLM extracts task-relevant values and their types into $\mathcal{F}$ while preserving the owner alias and associated time. A memory node can supply an input, initialize application data, or constrain the expected state. Jamie's message identifies the request, the calendar record identifies the meeting, and the talking-points note supplies message content. Focus time specifies an interval reserved for uninterrupted work, during which meetings should not be scheduled.

Edges. The edge set $\mathcal{E}$ has two groups with three edge types each, giving six types in $\mathcal{R}$. Skill–skill edges connect $v_1, v_2 \in \mathcal{V}_S$ through `produces-input`, `enables-precondition`, or `orders-before`, describing information transfer, state requirements, and execution order. Memory–skill edges connect $v_1 \in \mathcal{V}_M$ to $v_2 \in \mathcal{V}_S$ through `binds-private-slot`, `seeds-environment`, or `imposes-constraint`, describing how a record supplies an input, initializes application data, or constrains the expected state.

How to construct SMG? After node extraction, we check each ordered skill–skill and memory–skill pair against the source records. Each of its three applicable edge types is tested independently, so a pair may have multiple supported edges or none.

For skill–skill pairs, we add `produces-input` when an output matches a downstream input in name and type, `enables-precondition` when an operation establishes a state required by the

next, and `orders-before` when the source specifies their order. For memory–skill pairs, we add `binds-private-slot` when a record supplies an input unavailable from the instruction, public state, or earlier outputs; `seeds-environment` when it supplies required initial data; and `imposes-constraint` when it restricts valid outcomes. For example, a contact supplies the notification recipient, while focus time constrains rescheduling. These checks establish local dependencies; complete workflows still require execution validation.

Each edge stores its source text and the relevant information extracted from it as evidence ξ_e (e.g., a note reserving 2–3 p.m. for focus time and the extracted time interval). It receives a weight $w_e \in [0, 1]$ reflecting how strongly this evidence supports the dependency. These weights guide subgraph scoring during search; scoring details appear in Appendix E.

4.2 Stage II: Joint Skill-Memory Search

Given the SMG $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{R})$, JSMS identifies valid task subgraphs $\mathbb{G}^*$ within the graph.

Definition 2 (Task subgraph)**.** *A task subgraph is $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$, where $\mathcal{V}^* = \mathcal{V}_S^* \cup \mathcal{V}_M^*$, $\mathcal{V}_S^* \subseteq \mathcal{V}_S$, $\mathcal{V}_M^* \subseteq \mathcal{V}_M$, and $\mathcal{E}^* \subseteq \mathcal{E}$. Every edge in $\mathcal{E}^*$ has both endpoints in $\mathcal{V}^*$. The skill nodes form a directed path specifying workflow order; we call this ordered sequence a skill workflow. Multiple memory nodes may connect to different skill nodes. All selected records share one owner alias, meaning they belong to one user's data, not that all people mentioned are the same person.*

Our proposed Joint Skill-Memory Search (JSMS) algorithm grows each candidate task subgraph from one skill node. Appendix F gives the complete pseudocode.

a) Growing candidate subgraphs. JSMS starts from each skill $v \in \mathcal{V}_S$ with a candidate $\hat{\mathcal{G}} = (\{v\}, \emptyset)$. It extends the current skill workflow by following an outgoing skill–skill edge from the last node. If several next skills are available, each creates a separate path. A skill is added only if it has not appeared in the path and does not violate the required operation order. JSMS tries to attach memories to the starting skill and after each extension, so both short and long paths can produce tasks. Extension stops when reaching the maximum length $L_{\max}$ or no valid next skill remains.

b) Attaching memory nodes. For a current path with skill nodes $\hat{\mathcal{V}}_S$, let $R(\hat{\mathcal{V}}_S)$ be the private inputs that public information and earlier skill outputs cannot provide. A private input is a value the agent must recover from a user memory record, such as the talking points needed to compose the message in Figure 2. JSMS follows memory–skill edges to find a record set $\hat{\mathcal{V}}_M$ that supplies these inputs, together with the required initial data and constraints. For example, rescheduling and notifying require the focus time, contacts, and talking-points note shown in Figure 1. All selected records must belong to the same user u. Each record set meeting these requirements is combined with the skill workflow and its supporting edges to form a separate candidate subgraph.

c) Accepting and selecting subgraphs. Candidates with consistent skill order and complete same-user memory support enter the pool $\mathcal{K}$. We score each candidate by

$$Q(\mathcal{G}^*) = \mathrm{Dep}(\mathcal{G}^*)\,\mathrm{Ground}(\mathcal{G}^*)\,\mathrm{Comp}(\mathcal{G}^*). \tag{1}$$

Dep takes the lowest weight among necessary dependency edges, preventing strong connections from hiding a weak but essential one. *Ground* averages the strongest memory support for each required private input, measuring how well those inputs are grounded. *Comp* exponentially penalizes skills beyond a preferred path length and the number of memories, favoring compact candidates; Multiplication lowers the score when any factor is weak.

Starting with an empty selected set $\mathbb{G}$, JSMS adds the candidate with the largest positive gain:

$$\Delta(\mathcal{G}^* \mid \mathbb{G}) = Q(\mathcal{G}^*) - \sum_{a \in \mathcal{D}(\mathcal{G}^*)} \eta_a\, c_a(\mathbb{G}), \tag{2}$$

where $\mathcal{D}(\mathcal{G}^*)$ contains the candidate's skills, memories, and edges counted as reusable graph resources, c_a counts selected subgraphs using resource a, and $\eta_a \geq 0$ controls its reuse penalty. Thus, an equally good candidate using less-repeated resources is preferred. Candidates containing or contained in a selected subgraph are excluded. Selection stops at the task budget B or when no positive gain remains, returning $\mathbb{G}^*$. Appendix F gives the score definitions and equivalent set objective.

Table 1: Comparison with existing mobile-agent benchmarks. Multi-app denotes tasks involving at least two apps; Memory denotes explicit evaluation using user history or saved preferences. – indicates that the source does not report the value.

Benchmark	#Tasks	#Apps	Multi-app	Memory	Skill	Generation
ANDROIDWORLD (Rawles et al., 2024)	116	20	9.5%	✗	✗	Templates
ANDROIDLAB (Xu et al., 2025)	138	9	–	✗	✗	Manual
SPA-BENCH (Chen et al., 2025)	340	66	11.8%	✗	✗	Manual
MOBILEWORLD (Kong et al., 2025)	201	20	62.2%	✗	✗	Manual
MOBILEGYM (Wu et al., 2026)	256	28	36.3%	✗	✗	Templates
KNOWU-BENCH (Chen et al., 2026)	192	23	–	✓	✗	Templates
MYPHONEBENCH (Tang et al., 2026)	300	10	–	✓	✗	Auto. + review
MOBILESKILLBENCH (ours)	281	67	35.9%	✓	✓	Automatic

4.3 STAGE III: AUDITABLE TASK COMPILATION

For each selected subgraph $\mathcal{G}^* \in \mathbb{G}^*$, the frozen LLM f_θ uses its skill descriptions, user records, and dependency edges to compile $t = (q, \sigma^0, \sigma^*, \varepsilon)$. The instruction q states the requested operations but leaves memory-dependent values for the agent to recover. The initial state σ^0 contains the records needed to perform these operations, including existing app data and user constraints. The expected state σ^* specifies the required changes and, where applicable, answer content. Finally, the evaluator ε translates these requirements into checks on the tool trajectory, final state, and answer. In Figure 1, memory records initialize σ^0; checks verify rescheduling, message content, and recipients.

Connecting tasks to the mobile environment. Our evaluation represents mobile operations through JSON tools rather than a graphical phone interface, following the API-based evaluation paradigm of APPWORLD and τ-BENCH (Trivedi et al., 2024; Yao et al., 2025). The task specification supplies q and σ^0 to the harness, which exposes tool schemas for reading and updating the structured app state. The agent accesses user records through these tools and receives JSON observations after each call; σ^* and ε remain hidden. Thus, compilation specifies the task data and success conditions, while the harness provides the interaction interface used in Section 6.

Checking memory dependence. We remove one selected record at a time, keeping q, the skill workflow, other records, and generation settings fixed, and recompile the affected initial state and success checks. A record is necessary if its removal leaves a required input unresolved or changes the required outcome or checks; wording changes alone do not count. For example, removing the talking points leaves the message content unspecified, whereas removing a focus time may change which meeting times are acceptable. Unnecessary records and their edges are pruned, and the task is regenerated and checked for support and consistency.

5 BENCHMARK STATISTICS

Benchmark scope. Table 1 summarizes the scale of the benchmark: 281 tasks covering 67 primary-app labels. These tasks span 11 user profiles and 8 functional domains. Instructions average 8.83 English words, while workflows average 4.77 reference steps and 4.80 executable actions, reflecting real-world mobile assistance where brief user requests require complex, multi-step execution.

Coverage and structure. Figure 3 shows the primary-app distribution and the number of skills per task. The benchmark contains 214 distinct skills used 717 times, averaging 2.55 skills per task. It includes 101 multi-app tasks (35.9%) and 77 tasks with at least six executable actions (27.4%). The source tasks comprise 268 execution tasks and 13 hybrid tasks combining execution with answer checks (4.6%). Tasks may cover multiple domains, yielding 400 domain assignments.

Comparison with existing benchmarks. Table 1 highlights the benchmark's combination of multi-app workflows, user memory, reusable skills, and automatic construction. Its multi-app proportion exceeds those of ANDROIDWORLD and SPA-BENCH and closely matches MOBILEGYM. Most importantly, MOBILESKILLBENCH is the only benchmark in this comparison to combine user memory, explicit reusable skills, and fully automatic task generation, supporting diverse, memory-dependent workflows while reducing reliance on manual authoring and predefined templates.

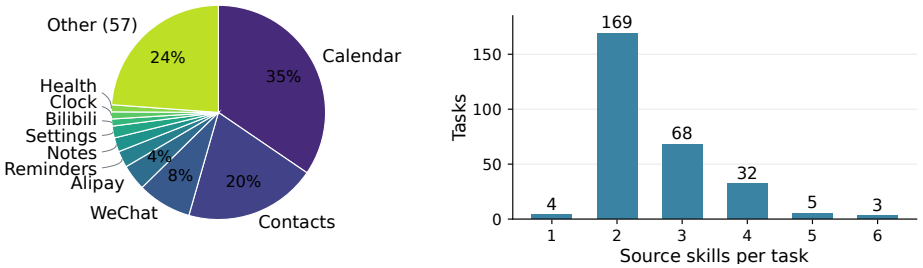


Figure 3: Composition of the final benchmark. Left: the ten most frequent primary-app labels; the remaining 57 labels are grouped as "Other". Right: number of source skills used per task.

6 EXPERIMENTS

Evaluation harness. We use a shared evaluation harness to evaluate all LLM backbones through JSON tool calls. For each synthesized task, the harness initializes σ^0 and provides the agent with the instruction q and available tool schemas. It handles tool calls, returns JSON observations, and records the interaction trajectory. At termination, the hidden evaluator ε checks the actions, final state, and any required answer against σ^*. The same task specifications, tool interface, and scoring procedure are used across models. Appendix G explains the data flow and JSON representations.

LLM backbones. We evaluate eight current LLM backbones with the same tool interface and task order: GPT-5.5, GLM-5.2, Gemini 3.1 Pro, Claude Opus 5, Doubao-Seed-2.0-Pro, Qwen3.8-Max, DeepSeek-V4-Pro, and Kimi K3. We freeze the system prompt, tool schemas, action budget, temperature, context policy, and termination rules across models. The comparison therefore measures differences in tool-based task execution while keeping the environment and evaluator fixed.

Metrics. *Success rate (SR)* is the primary metric and counts a task as successful only when all required actions and final-state checks pass. *Completion* is the mean fraction of required evaluator conditions satisfied per task, so it captures partial progress. We additionally report SR on single-app and multi-app subsets, mean tool-call steps, and reported token usage. Steps and tokens include failed attempts and are interpreted together with SR rather than as standalone quality measures. For construction ablations, we additionally report the number of generated tasks (#Tasks), the percentage of reviewed tasks judged valid by human evaluators (*Valid*), the reported rate of state checks supported by environment data (*Supported*), and the mean per-task action process error rate (*Errors*).

6.1 MAIN RESULTS

Table 2 shows that even strong LLM agents struggle with the mobile scenarios in MOBILESKILL-BENCH: overall SR ranges from 15.3% to 28.8%, leaving more than 70% of tasks unsolved even by the best-performing model. Completion consistently exceeds SR; GPT-5.5, for example, reaches 67.9% Completion but only 28.8% SR. This gap indicates that agents often make substantial partial progress without satisfying all requirements, underscoring the importance of evaluating the complete task outcome. Multi-app tasks are harder for every model: GPT-5.5 drops from 38.9% SR on single-app tasks to 10.9% on multi-app tasks, and no model exceeds 10.9% on the latter. This consistent gap points to coordinating operations across applications as a major challenge in personalized mobile assistance. Execution cost also does not consistently track success: Doubao-Seed-2.0-Pro uses the most tokens and steps but remains below the best-performing model. Thus, longer execution alone does not ensure reliable completion. Together, these results motivate agents that maintain task requirements throughout execution and verify the final outcome before declaring success.

6.2 DETAILED ANALYSIS

Ablation Studies. We compare task-construction methods using the same source skill and memory pools, and evaluate the resulting task sets with GPT-5.5. Against the full method's joint SMG search, **a) Skill-only** selects skill workflows first and attaches supporting memories afterward, testing the role of memory in workflow selection; **b) Memory-only** selects records from one user first and then

Table 2: Main results on MOBILESKILLBENCH. SR, Completion, Single, and Multi are percentages; Steps is the mean number of actions per attempt.

Model	Overall		App SR		Efficiency	
	SR ↑	Completion ↑	Single ↑	Multi ↑	Steps ↓	Tokens ↓
Proprietary models						
GPT-5.5 (OpenAI, 2026)	**28.8**	**67.9**	**38.9**	**10.9**	11.1	4.28M
Gemini 3.1 Pro (Google DeepMind, 2026)	20.6	57.8	28.3	6.9	**10.0**	4.03M
Claude Opus 5 (Anthropic, 2026)	18.5	46.5	27.2	3.0	11.6	4.53M
Doubao-Seed-2.0-Pro (ByteDance Seed, 2026)	19.9	48.6	28.3	5.0	27.7	13.17M
Qwen3.8-Max (Qwen Team, 2026)	22.1	55.8	32.2	4.0	10.7	**3.58M**
Open-weight models						
GLM-5.2 (Z.ai, 2026)	15.3	45.2	22.2	3.0	**8.8**	**3.49M**
DeepSeek-V4-Pro (DeepSeek, 2026)	18.2	**49.1**	25.0	**5.9**	17.3	11.41M
Kimi K3 (Moonshot AI, 2026)	**19.2**	46.1	**27.8**	4.0	8.9	3.85M

Table 3: Ablation of task-construction methods using the same source skills and memories.

Method	#Tasks	SR ↑	Completion ↑	Valid ↑	Supported ↑	Errors ↓
Full MOBILESKILLBENCH	**52**	**35.0**	**56.3**	**86.7**	**94.8**	**38.8**
Skill-only	37	34.3	55.6	61.7	86.7	41.6
Memory-only	41	12.2	32.7	68.3	68.6	72.3
Topic-random	38	21.1	44.1	63.3	74.0	63.8

adds compatible skills, testing the role of skill dependencies in task construction; **c) Topic-random** randomly combines topically related skills and memories instead of selecting them through joint dependency search, testing whether topical relevance alone is sufficient. Table 3 reports construction quality and downstream agent performance.

The full method achieves the highest validity (86.7%) and evaluator support (94.8%), with the lowest process error rate (38.8%). Skill-only yields similar SR and Completion but lower validity (61.7%), suggesting that attaching memories after workflow selection weakens task coherence. Memory-only has the lowest evaluator support (68.6%) and the highest error rate (72.3%), highlighting the importance of skill dependencies for constructing executable tasks. Topic-random also reduces validity and support, indicating that topical relevance does not replace explicit dependencies. Together, these results support jointly selecting skills and memories to improve task quality and checkability.

Performance by Skill Count. To examine the challenge of composing operations, we group tasks by skill count in Figure 4(a,b). All eight models show lower SR and Completion when moving from two skills to three. For GPT-5.5, SR drops from 39.1% to 11.8%, while Completion falls from 75.4% to 57.0%. The much steeper loss in strict success suggests that partial execution does not reliably translate into end-to-end completion as workflows grow. This highlights skill coordination as a key challenge: agents must preserve dependencies across operations and satisfy their requirements.

Performance by Memory Count. To examine how execution performance varies with memory requirements, we group tasks by memory count in Figure 4(c,d). Performance is not monotonic in record count, but all eight models achieve lower SR and Completion on three-memory tasks than on two-memory tasks. GPT-5.5's SR falls from 34.5% to 13.0%, and Completion from 72.1% to 52.2%. This shared decline suggests that jointly using multiple records to guide execution remains challenging. The key requirement is to integrate the relevant information and constraints into a coherent sequence of actions, beyond retrieving individual records.

Case Study. To understand why agents fail, we manually analyze all failed runs across the eight models and group them by their primary failure cause (Figure 5a). We identify six categories: **Tool/action error**, where a tool call fails because the target cannot be found or the action or arguments are invalid; **Loop / step budget**, where repeated exploration or ineffective actions exhaust

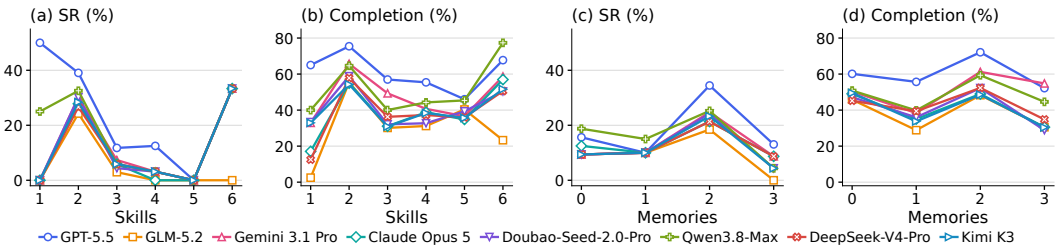


Figure 4: SR and Completion (%) by skill count (a,b) and memory count (c,d).

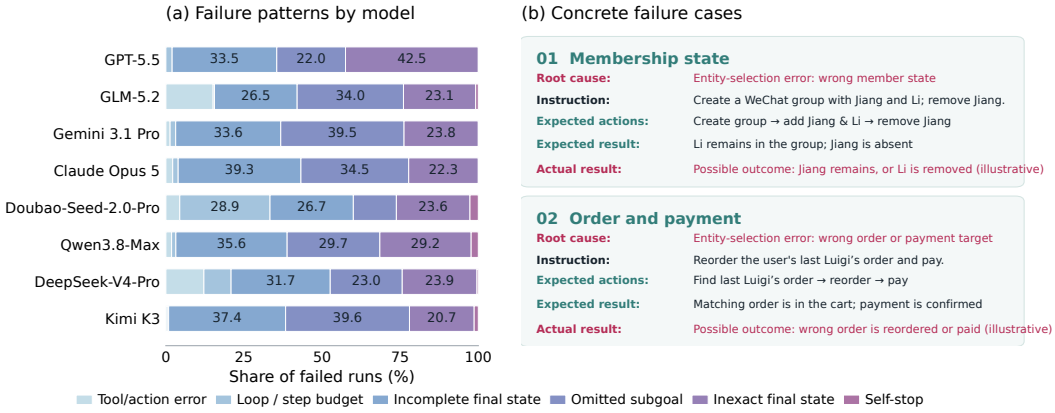


Figure 5: Manual analysis of all failed runs and representative cases. Left: primary failure causes, normalized within each model's failed runs. Right: illustrative possible failure outcomes organized by root cause, instruction, expected actions, and expected result.

the interaction budget; **Incomplete final state**, where operations are attempted but the required result is not saved, submitted, confirmed, or maintained; **Omitted subgoal**, where the agent declares completion while leaving a required part of the task undone; **Inexact final state**, where the resulting entity, value, membership, time, or status differs from the requested outcome; and **Self-stop**, where the agent gives up before completion. Each failed run receives one primary label, and proportions are normalized over the failed runs of each model.

The analysis highlights a recurring gap between performing operations and ensuring their intended effects. Inexact final states account for 42.5% of GPT-5.5's failures, omitted subgoals for 39.5% of Gemini's, and loops or exhausted budgets for 28.9% of Doubao's, suggesting distinct needs for outcome verification, subgoal tracking, and effective progress monitoring. Moreover, 185 tasks fail for all eight models, yet 89.1% of these 1,480 runs terminate with `completed`. Figure 5b highlights two examples: the WeChat task requires Li to remain while Jiang is removed, and the Luigi's task requires the matching order and confirmed payment. All eight agents report completion on both, yet none passes; mean Completion is only 21.3% and 18.8%, respectively. The cases show that executing actions or announcing success is insufficient without verifying the final state.

7 Conclusion

We presented MobileSkillBench, a framework and benchmark for evaluating how LLM agents use user memory to complete complex mobile tasks. Its Skill–Memory Graph connects reusable operations to the personal records that determine their inputs and constraints. Joint search and auditable compilation turn these dependencies into tasks with consistent instructions, states, and evaluators, while record-removal checks assess memory necessity. The resulting 281-task benchmark challenges even state-of-the-art LLM agents, exposing a gap in reliable personalized assistance. By making memory-dependent execution central to task construction and evaluation, MobileSkillBench provides a basis for developing agents that carry user context through multi-step workflows.

ETHICS STATEMENT

MOBILESKILLBENCH is constructed from de-identified real user data. Direct identifiers are removed or replaced with aliases before graph construction, and raw user records are not included in the benchmark release. Because messages, calendars, contacts, and notes may still contain sensitive structure, access controls and applicable data-governance requirements must be maintained throughout collection, processing, and release. The benchmark is intended for evaluating assistants with authorized access to a user's data, not for inferring information about non-consenting individuals.

REPRODUCIBILITY STATEMENT

Section 4 defines the graph, search objectives, and task compilation procedure. Appendices A–F document the notation, input construction, benchmark composition, and pseudocode. Frozen prompts, lookup tables, task specifications, and harness settings will be released with the evaluation implementation.

AI USE STATEMENT

Generative AI tools were used to assist with language editing and figure drafting; all technical claims and final presentation were reviewed by the authors. The use of a frozen LLM inside the proposed construction method is part of the research methodology and is described in Section 4.

REFERENCES

Anthropic. Introducing Claude Opus 5. Model announcement and system card, 2026. URL https://www.anthropic.com/news/claude-opus-5.

Apple. Apple introduces Siri AI, a profoundly more capable and personal assistant. Apple Newsroom, June 8, 2026. URL https://www.apple.com/newsroom/2026/06/apple-introduces-siri-ai-a-profoundly-more-capable-and-personal-assistant/.

Tong Bai, Zhenglin Wan, Pengfei Zhou, Xingrui Yu, Yang You, and Ivor W. Tsang. SkillDAG: Self-evolving typed skill graphs for LLM skill selection at scale. *arXiv preprint arXiv:2606.03056*, 2026. URL https://arxiv.org/abs/2606.03056.

ByteDance Seed. Seed2.0 model card: Towards intelligence frontier for real-world complexity. *arXiv preprint arXiv:2607.00248*, 2026. URL https://arxiv.org/abs/2607.00248.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Yixing Li, Xurui Zhou, Weiwen Liu, Shuai Wang, Kaiwen Zhou, Rui Shao, Liqiang Nie, Yasheng Wang, Jianye Hao, Jun Wang, and Kun Shao. SPA-Bench: A comprehensive benchmark for smartphone agent evaluation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OZbFRNhpwr. App count follows the official release at https://github.com/ai-agents-2030/SPA-Bench.

Tongbo Chen, Zhengxi Lu, Zhan Xu, Guocheng Shao, Shaohan Zhao, Fei Tang, Yong Du, Kaitao Song, Yizhou Liu, Yuchen Yan, Wenqi Zhang, Xu Tan, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. KnowU-Bench: Towards interactive, proactive, and personalized mobile agent evaluation. *arXiv preprint arXiv:2604.08455*, 2026. URL https://arxiv.org/abs/2604.08455.

DeepSeek. DeepSeek-V4-Pro general availability release. DeepSeek API documentation, 2026. URL https://api-docs.deepseek.com/news/news260813/.

Xinle Deng, Yida Xue, Xiangyuan Ru, Yijun Chen, Buqiang Xu, Mingjun Mao, Xinjie Liu, Haoming Xu, Shuofei Qiao, Mengru Wang, Chen Jiang, Yuchen Eleanor Jiang, Lizhong Wang, Jason Wang, Li Zeng, Haofen Wang, Guilin Qi, Huajun Chen, and Ningyu Zhang. MobileMem:

Learning from a year of mobile experiences. *arXiv preprint arXiv:2608.13606*, 2026. URL https://arxiv.org/abs/2608.13606.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs. *arXiv preprint arXiv:2604.25727*, 2026. URL https://arxiv.org/abs/2604.25727.

Google. More than 1 billion people are using the Gemini app every month. Google Blog, August 11, 2026. URL https://blog.google/innovation-and-ai/products/gemini-app/one-billion-monthly-users/.

Google DeepMind. Gemini 3.1 Pro model card. Model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-1-pro/.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive and MCP-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025. URL https://arxiv.org/abs/2512.19432.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Han-chung Lee, and Dawn Song. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Yuanjie Lyu, Chengyu Wang, Lei Shen, Jun Huang, and Tong Xu. Mock worlds, real skills: Building small agentic language models with synthetic tasks, simulated environments, and rubric-based rewards. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12529–12545, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.570. URL https://aclanthology.org/2026.acl-long.570/.

Moonshot AI. Kimi K3: Open frontier intelligence. Technical report and open weights, 2026. URL https://github.com/MoonshotAI/Kimi-K3.

Hongyi Nie, Xunyuan Liu, Yudong Bai, Yaqing Wang, Yang Liu, Quanming Yao, and Zhen Wang. PSPA-Bench: A personalized benchmark for smartphone GUI agent. *arXiv preprint arXiv:2603.29318*, 2026. URL https://arxiv.org/abs/2603.29318.

OpenAI. GPT-5.5 system card. OpenAI system card, 2026. URL https://openai.com/index/gpt-5-5-system-card/.

Qwen Team. Qwen3.8. Model release, 2026. URL https://github.com/QwenLM/Qwen3.8.

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 preprint arXiv:2405.14573*, 2024. URL https://arxiv.org/abs/2405.14573.

Zhengyang Tang, Ke Ji, Xidong Wang, Zihan Ye, Xinyuan Wang, Yiduo Guo, Ziniu Li, Chenxin Li, Jingyuan Hu, Shunian Chen, Tongxu Luo, Jiaxi Bi, Zeyu Qin, Shaobo Wang, Xin Lai, Pengyuan Lyu, Junyi Li, Can Xu, Chengquan Zhang, Han Hu, Ming Yan, and Benyou Wang. Do phone-use agents respect your privacy? *arXiv preprint arXiv:2604.00986*, 2026. URL https://arxiv.org/abs/2604.00986.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Wenhao Wang, Peizhi Niu, Gongyi Zou, Xiyuan Yang, Jingxing Wang, Haoting Shi, Yaxin Du, Jingyi Chai, Xianghe Pang, Shuo Tang, Yanfeng Wang, and Siheng Chen. MCP-Persona: Benchmarking LLM agents on real-world personal applications via environment simulation. *arXiv preprint arXiv:2606.02470*, 2026. URL https://arxiv.org/abs/2606.02470.

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

Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. AgentSynth: Scalable task generation for generalist computer-use agents. *arXiv preprint arXiv:2506.14205*, 2025. URL https://arxiv.org/abs/2506.14205.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. AndroidLab: Training and systematic benchmarking of Android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2144–2166, 2025. doi: 10.18653/v1/2025.acl-long.107. URL https://aclanthology.org/2025.acl-long.107/.

Qinglong Yang, Haoming Li, Haotian Zhao, Xiaokai Yan, Jingtao Ding, Fengli Xu, and Yong Li. FingerTip 20K: A benchmark for proactive and personalized mobile LLM agents. *arXiv preprint arXiv:2507.21071*, 2025. URL https://arxiv.org/abs/2507.21071.

Shunyu Yao et al.
tau-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2406.12045.

Z.ai. GLM-5.2 model release. Official model repository, 2026. URL https://github.com/zai-org/GLM-5.

Jiadong Zhang and Xiaosong Ma. MemArena: An ego-centric benchmark for on-device agentic personal memory assistants at scale. *arXiv preprint arXiv:2608.02613*, 2026. URL https://arxiv.org/abs/2608.02613.

A Notation

Calligraphic symbols denote sets or structured descriptions; $\mathbb{G}$ denotes a set of task subgraphs. Lowercase symbols denote individual objects or functions. Table 4 lists the main symbols used in Sections 3–4.

B Input Construction

Skill collection. The raw skill-document set $\mathcal{D}_S$ is built from real mobile-assistant service scenarios, including execution traces, app or API descriptions, and recurring task templates. We first collect documents at the granularity of one meaningful mobile operation, such as reading a message, extracting an event time, checking a calendar conflict, creating a calendar event, or sending a message. We discard documents that are too specific to one screen layout, duplicate another operation, or describe a whole task rather than a reusable operation. The frozen LLM f_θ then maps each accepted document d_S to a skill node v_S with description $(\mathcal{I}, \mathcal{O}, \mathcal{C}, \delta)$. The skill nodes are checked for duplicate meanings, missing inputs or outputs, inconsistent runnable states, and app-domain coverage before graph induction.

Memory collection. The raw memory-data set $\mathcal{D}_M$ is built from de-identified real user data. Before records enter the benchmark, direct personal identifiers are removed or replaced with stable aliases, and sensitive free text is filtered or rewritten while preserving the task-relevant structure. Eligible sources include messages, calendar events, notes, contacts, preferences, reminders, bills, and other app data that persists across sessions. We exclude transient interface state, public runtime

Table 4: Main notation.

Symbol	Meaning
$\mathcal{D}_S$, d_S	Raw skill-document set and one raw skill document.
$\mathcal{D}_M$, d_M	Raw memory-data set and one raw memory record.
u, $\mathcal{X}$, λ	Owner alias, de-identified content, and optional time in raw memory record d_M.
f_θ, θ	LLM used for extraction and compilation, and its fixed parameters.
v_S, v_M	Skill node extracted from d_S and memory node extracted from d_M.
$\mathcal{I}$, $\mathcal{O}$, $\mathcal{C}$, δ	Input/output slot descriptions, runnable states, and described state update.
α, k	Assignment of skill inputs and interaction index.
$\mathcal{F}$	Task-relevant fields extracted from $\mathcal{X}$ for a memory node.
t	One mobile-agent task.
q, σ^0, σ^k, σ^*, σ', ε	Instruction, initial state, intermediate state, expected state, final state, and evaluator.
$\mathcal{G}$	Skill–Memory Graph.
$\mathcal{V}$, $\mathcal{V}_S$, $\mathcal{V}_M$	All graph nodes, skill nodes, and memory nodes.
$\mathcal{E}$, e	Graph edge set and one edge.
$\mathcal{R}$	Edge-type set.
ξ_e, w_e	Source evidence stored on edge e and its relation strength in $[0, 1]$.
Rel, Sup, Spec, Val	Fixed factors for edge type, evidence support, endpoint specificity, and edge validation.
$\hat{\mathcal{G}}$, $\mathcal{G}^*$	Partial task subgraph and one complete task subgraph.
$\mathcal{V}_S^*$, $\mathcal{V}_M^*$, $\mathcal{E}^*$	Skill nodes, memory nodes, and edges in task subgraph $\mathcal{G}^*$.
$R(\mathcal{V}_S^*)$, z	Private slots required by a skill workflow and one such slot.
$L_{\max}$, $L^\star$, β_L, β_M	Maximum and preferred workflow lengths and compactness coefficients.
$\mathcal{E}^*_{\text{core}}$, $\mathcal{E}_{\mathcal{G}^*}(z)$	Required edges and memory–skill edges supporting slot z.
Q, Dep, Ground, Comp	Candidate quality and its dependency, grounding, and compactness factors.
$\mathcal{K}$, $\mathbb{G}$, $\mathbb{G}^*$, B	Complete candidate pool, a candidate subset, the selected set, and the task budget.
$\mathcal{D}_{\text{res}}$, c_a, η_a, J	Reusable resources, reuse count, reuse penalty, and set-selection objective.
$\mathcal{T}$, $\mathcal{T}_{\text{eval}}$	Constructed task set and its subset with evaluators.
ρ_i, SR	Agent trajectory on task t_i and strict success rate.

objects, and records without a stable owner. The frozen LLM f_θ then maps each accepted record $d_M = (u, \mathcal{X}, \lambda)$ to a memory node v_M with description $(u, \mathcal{F}, \lambda)$: $\mathcal{F}$ stores typed fields such as person, time, location, amount, preference, or constraint. We then check that every memory node has a valid owner alias and enough typed content to initialize a mobile state, fill a private slot, or constrain a skill node.

C Benchmark Composition

The final release contains 281 tasks. All counts in this appendix refer to this release rather than to earlier construction exports. A task may involve several apps and functional domains; therefore, multi-label domain counts are not expected to sum to 281.

Table 5: Final benchmark composition.

Quantity	Count
Tasks	281
Primary-app labels	67
User/environment profiles	11
Functional domains	8
Multi-app tasks	101 (35.9%)
Distinct skill IDs	214
Skill uses	717

The benchmark contains 1,350 executable actions, 1,676 fine-grained evaluation conditions, and 474 active final-state scoring points. The average task contains 2.55 skills, 4.77 reference workflow

Table 6: Skill and memory counts in the final 281-task release.

Count per task	0	1	2	3	4	5	6
Skills (tasks)	0	4	169	68	32	5	3
Memories (tasks)	32	20	206	23	–	–	–

Table 7: Functional-domain distribution in the final release. A task may receive multiple domain labels.

Domain	Tasks
Time, schedule, and reminders	146
Contacts and communication	121
Files, documents, and notes	31
Local services, shopping, and travel	30
Media and content	28
Payments, bills, and finance	21
Health and appointments	12
Location, weather, and navigation	11

steps, 4.80 executable actions, and 1.78 memory records. The main text reports the corresponding skill and memory distributions in Figure 3. The 13 source hybrid tasks (tasks combining execution and an answer check) are retained in the release, but their active evaluator type is execution; any answer-quality checks are stored separately.

Counting conventions. A *skill* is one distinct reusable operation selected in a task subgraph; a emphskill use counts each occurrence of a skill in a task, so repeated use across tasks contributes repeatedly. A *reference workflow step* is an operation described in the compiled workflow, whereas an *executable action* is one tool operation available to the harness; the two counts need not match. A *memory record* is one selected user record attached to a task subgraph, regardless of how many fields it supplies. "Multi-app" means that the task's executable actions involve at least two applications. Domain counts are multi-label assignments and therefore may exceed the number of tasks.

D Evaluation and Audit Definitions

The main evaluation metrics are defined over the active evaluator checks in each task. SR is one only when every required active check passes; Completion is the fraction of active checks satisfied, averaged over tasks. Single and Multi are SR restricted to tasks involving one or at least two applications, respectively. Steps count tool calls in an execution trajectory, including failed calls; Tokens use the reported model-token count for the same trajectory.

For construction ablations, *Valid* is the fraction of manually reviewed generated tasks judged coherent and executable. *Supported* is the reported fraction of evaluator checks linked to supporting environment data, and *Errors* is the mean task-level process-error rate computed from the exported audit. These audit measures evaluate task construction; SR and Completion evaluate an agent running the resulting tasks and should not be interpreted as interchangeable.

Failure-analysis protocol. The failure study manually reviews the failed runs and assigns each run one primary error label using the observed stopping reason, action coverage, and final-state evidence. The six labels and their meanings are defined in Section 6. Figure 5b contains illustrative possible outcomes based on the aggregate reports; it is not a substitute for a per-trajectory final-state log.

E Graph Induction Details

Algorithm 1 summarizes Section 4.1. Each edge condition is evaluated from the structured node descriptions.

Edge weights. For an edge e of type r, its weight is

$$w_e = \mathrm{Rel}(r)\,\mathrm{Sup}(e)\,\mathrm{Spec}(e)\,\mathrm{Val}(e). \tag{3}$$

The four factors represent the prior strength assigned to the edge type (Rel), support from the source evidence (Sup), how specifically that evidence identifies the two connected nodes (Spec), and the strength of validation of the proposed dependency (Val). Each factor uses fixed grades in $[0, 1]$. Their product lowers the weight when any factor is weak and gives zero weight when any factor is zero. The factors are selected from versioned lookup tables and rounded to four decimals; their entries and the supporting fields in ξ_e are stored with the edge. These weights are subsequently used in the dependency and memory-support scores during search.

Algorithm 1 Skill–Memory Graph Induction

Require: Raw skill documents $\mathcal{D}_S$, raw memory data $\mathcal{D}_M$, frozen LLM f_θ
Ensure: Skill–Memory Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{R})$

$\mathcal{V}_S \leftarrow \textsc{ExtractSkillNodes}(f_\theta, \mathcal{D}_S)$
$\mathcal{V}_M \leftarrow \textsc{ExtractMemoryNodes}(f_\theta, \mathcal{D}_M)$
$\mathcal{V} \leftarrow \mathcal{V}_S \cup \mathcal{V}_M$
$\mathcal{E} \leftarrow \emptyset$
for all $v_1, v_2 \in \mathcal{V}_S$ and each skill–skill type $r \in \mathcal{R}$ **do**
    if the condition for r in Section 4.1 holds **then**
        store ξ_e, compute w_e, and add $(v_1, v_2, r, \xi_e, w_e)$ to $\mathcal{E}$
    end if
end for
for all $v_1 \in \mathcal{V}_M$, $v_2 \in \mathcal{V}_S$, and each memory–skill type $r \in \mathcal{R}$ **do**
    if the condition for r in Section 4.1 holds **then**
        store ξ_e, compute w_e, and add $(v_1, v_2, r, \xi_e, w_e)$ to $\mathcal{E}$
    end if
end for
return $(\mathcal{V}, \mathcal{E}, \mathcal{R})$

F Search and Compilation Details

F.1 Candidate Quality

A candidate $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ contains skill nodes $\mathcal{V}_S^*$, memory nodes $\mathcal{V}_M^*$, and their connecting edges. Only candidates with consistent operation order and complete same-user record support enter the feasible pool $\mathcal{K}$. Scoring ranks these feasible candidates; it does not replace feasibility checks. Their quality is

$$Q(\mathcal{G}^*) = \mathrm{Dep}(\mathcal{G}^*)\,\mathrm{Ground}(\mathcal{G}^*)\,\mathrm{Comp}(\mathcal{G}^*). \tag{4}$$

Each factor lies in $[0, 1]$, so Q also lies in $[0, 1]$. The product reduces the overall score when any factor is low; a zero factor makes the score zero. These are construction scores, not calibrated probabilities of agent success.

Dependency strength. Let $\mathcal{E}^*_{\mathrm{core}} \subseteq \mathcal{E}^*$ contain the edges required to establish workflow order or provide necessary inputs, initial data, and constraints. Using the edge weights $w_e \in [0, 1]$ assigned during graph induction, we define

$$\mathrm{Dep}(\mathcal{G}^*) = \begin{cases} \min_{e \in \mathcal{E}^*_{\mathrm{core}}} w_e, & \mathcal{E}^*_{\mathrm{core}} \neq \emptyset, \\ 1, & \mathcal{E}^*_{\mathrm{core}} = \emptyset. \end{cases} \tag{5}$$

The minimum identifies the weakest necessary connection: several strong edges cannot compensate for one poorly supported dependency on which the task relies. For example, a reliable calendar workflow can still receive a low score if the connection between a selected note and the required notification content is weak. The empty-set value is neutral and introduces no dependency penalty.

Private-input grounding. Let $R(\mathcal{V}_S^*)$ contain the private inputs that public information and earlier skill outputs cannot provide. For each input z, let $\mathcal{E}_{\mathcal{G}^*}(z)$ be its supporting memory–skill edges in

the candidate. Feasibility requires this set to be nonempty. We first take the strongest support for each input, then average across inputs:

$$\mathrm{Ground}(\mathcal{G}^*) = \begin{cases} \dfrac{1}{|R(\mathcal{V}_S^*)|} \displaystyle\sum_{z \in R(\mathcal{V}_S^*)} \max_{e \in \mathcal{E}_{\mathcal{G}^*}(z)} w_e, & R(\mathcal{V}_S^*) \neq \emptyset, \\ 1, & R(\mathcal{V}_S^*) = \emptyset. \end{cases} \tag{6}$$

Taking the maximum avoids rewarding an input merely for having many supporting records; averaging gives each required input equal weight. As an illustrative calculation, best-support weights of 0.9 for a recipient and 0.7 for message content yield $\mathrm{Ground} = 0.8$. Unlike Dep, which checks the weakest necessary dependency across the subgraph, Ground measures the average strength of the available evidence for private inputs. The factors can share edges but aggregate different aspects of their support. A task with no private inputs receives the neutral value 1.

Compactness. We discourage skill workflows exceeding a preferred skill count and penalize the number of selected memories:

$$\mathrm{Comp}(\mathcal{G}^*) = \exp[-\beta_L \max\{0, |\mathcal{V}_S^*| - L^\star\} - \beta_M |\mathcal{V}_M^*|]\,. \tag{7}$$

Here, $L^\star$ is the preferred skill count, and $\beta_L, \beta_M \geq 0$ control the two penalties. Skill workflows at or below $L^\star$ incur no length penalty; each additional skill above it multiplies Comp by $e^{-\beta_L}$. Each additional memory multiplies it by $e^{-\beta_M}$. Setting either coefficient to zero disables that penalty. For example, with $L^\star = 3$, a five-skill candidate is penalized for two excess skills. This is a size preference, not a test of redundancy: even necessary memories incur the memory penalty. Stage III separately checks necessity by removing records and recompiling. The hard search limit $L_{\max}$ bounds which skill workflows are considered, whereas $L^\star$ affects their ranking.

F.2 Diversity and Greedy Selection

Let $\mathcal{D}_{\mathrm{res}}$ be the set of skill nodes, memory nodes, and edges whose reuse is penalized, and let $\mathcal{D}(\mathcal{G}^*) \subseteq \mathcal{D}_{\mathrm{res}}$ contain those present in candidate $\mathcal{G}^*$. For a selected set $\mathbb{G} \subseteq \mathcal{K}$, the resource-use count is

$$c_a(\mathbb{G}) = \sum_{\mathcal{G}^* \in \mathbb{G}} \mathbf{1}[a \in \mathcal{D}(\mathcal{G}^*)]\,, \tag{8}$$

where the indicator $\mathbf{1}[\cdot]$ equals one when its condition holds and zero otherwise. We score the selected set by

$$J(\mathbb{G}) = \sum_{\mathcal{G}^* \in \mathbb{G}} Q(\mathcal{G}^*) - \sum_{a \in \mathcal{D}_{\mathrm{res}}} \eta_a \binom{c_a(\mathbb{G})}{2}. \tag{9}$$

The nonnegative weight η_a controls the penalty for each pair of tasks sharing resource a. Since $\binom{c}{2} = c(c-1)/2$, a resource used by zero or one task incurs no penalty; use by two and three tasks creates one and three sharing pairs, respectively. This discourages repeatedly selecting tasks built from the same resources while allowing reuse when candidate quality outweighs the penalty.

For an unselected candidate, its marginal gain is

$$\begin{aligned} \Delta(\mathcal{G}^* \mid \mathbb{G}) &= J(\mathbb{G} \cup \{\mathcal{G}^*\}) - J(\mathbb{G}) \\ &= Q(\mathcal{G}^*) - \sum_{a \in \mathcal{D}(\mathcal{G}^*)} \eta_a c_a(\mathbb{G}). \end{aligned} \tag{10}$$

This is exactly the formula used in the main text: adding the candidate creates $c_a(\mathbb{G})$ new sharing pairs for each resource it uses, because $\binom{c+1}{2} - \binom{c}{2} = c$. An unused resource adds no penalty; a frequently selected resource reduces the gain more. As an illustrative calculation, quality 0.8 and one penalized resource already used twice with $\eta_a = 0.1$ give a gain of 0.6.

Starting from $\mathbb{G} = \emptyset$, JSMS repeatedly adds the admissible candidate with the largest positive Δ. A candidate is inadmissible if its node and edge sets contain, or are contained in, those of an already selected candidate. This containment exclusion, denoted $\mathrm{AntiChain}$ in Algorithm 2, prevents nested task variants from being selected together. Gains are updated after each addition, and ties are resolved by stable node identifiers. Selection stops when $|\mathbb{G}| = B$, where B is the task budget, or when no admissible candidate has positive gain. The result is $\mathbb{G}^*$. Greedy selection is an approximation and does not guarantee the globally optimal set.

F.3 Search Procedure and Compilation Checks

Algorithm 2 summarizes JSMS over the candidate pool $\mathcal{K}$. A prefix in the algorithm is the current partial skill workflow from its starting skill to any intermediate skill, including the single starting node. For that path, $\hat{\mathcal{V}}_S$ denotes its skill nodes, and $R(\hat{\mathcal{V}}_S)$ has the same private-input meaning as above. The scores apply before compilation. If record removal changes a selected subgraph, the reduced structure requires renewed support and consistency checks; its previous score and containment relationships cannot automatically be attributed to the reduced collection. Execution in the mobile environment is still needed to establish executability.

G Evaluation Harness and JSON Examples

The harness connects a compiled task $t = (q, \sigma^0, \sigma^*, \varepsilon)$ to an LLM agent: it supplies the request, mediates access to app data through tools, and passes execution evidence to the evaluator. We explain these responsibilities through a reorder task: *"Put my last Luigi's order in the cart."* The instruction and evaluator excerpts below come from the available task-construction export. The initial-state, tool-schema, and interaction examples are schematic, with illustrative identifiers and dates; they explain the interface rather than reproduce a recorded evaluation episode or prescribe the runtime's exact field names.

G.1 Task Loading and Information Boundaries

Linking the task to its checks. The construction export stores task instructions and evaluation plans separately. The field `plan_id` links them; `source_case_id` links a task back to its source subgraph. Listing 1 reproduces these fields and the instruction. The full task plan also contains `reference_workflow` and `agent_reference_answer`. These are construction and evaluation artifacts, not instructions to expose to the tested agent. In particular, a reference response such as "the order is in the cart" is not evidence that the agent actually changed the cart.

What the agent can see. The agent receives q, the tool schemas, and observations returned by its tool calls. The initial state σ^0 holds the data those tools access; it need not be copied wholesale into the prompt. The target conditions σ^*, evaluator ε, reference response, and construction diagnostics remain outside the agent input. This separation makes the historical order something to identify through interaction, rather than an answer supplied by the evaluator. The current benchmark export identifies the agent-facing instruction as `agent_view.user_query`.

G.2 Initial State and Tool Interaction

Representing app data. For the reorder example, σ^0 needs a history of orders and the current cart. Listing 2 shows an illustrative state with two orders from the same restaurant. Their identifiers distinguish the objects, and their timestamps resolve "last." The correct order contains two pizzas, garlic bread, and a drink. Having an older order with different contents explains why merely finding the restaurant is insufficient. The JSON represents application data; no screenshot or touch coordinate is required.

Describing an operation. A tool schema specifies the operation name and the arguments the agent must supply. Listing 3 illustrates a cart-update tool. Its required argument is an order identifier, so the agent must first identify the relevant historical order through the available read operations. The schema describes the call interface; it does not itself implement the state change. A runtime implementation must separately define how that call reads the order and updates the cart.

Connecting calls to observations. At step k, the agent uses its instruction and previous observations to select a tool call. The harness passes that call to the tool execution layer and returns its observation. Read operations expose relevant data; update operations may change σ^k to σ^{k+1}. Listing 4 illustrates one successful update after the target order has been identified. The request and response should be read together: the first states what the agent asked to do, and the second reports what the operation returned. They are distinct from the final state used for scoring. The harness records the interaction trajectory so that a stated intention, a tool request, and an accomplished change can be distinguished.

Algorithm 2 Joint Skill-Memory Search

Require: Skill–Memory Graph $\mathcal{G}$, task budget B, maximum number of skill nodes $L_{\max}$
Ensure: Selected task subgraphs $\mathbb{G}^*$

$\mathcal{K} \leftarrow \emptyset$
for all start nodes $v \in \mathcal{V}_S$ **do**
  initialize $\hat{\mathcal{G}} \leftarrow (\{v\}, \emptyset)$
  branch from the last skill node along eligible outgoing edges without repetition
  for all prefixes of length at most $L_{\max}$, including the starting node **do**
    compute $R(\hat{\mathcal{V}}_S)$
    enumerate alternative memory sets through incoming memory–skill edges
    for all sets covering required inputs, initial data, and constraints with at most one owner **do**
      allow the empty set only when no record is required
      add $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ to $\mathcal{K}$
    end for
  end for
end for
compute $Q(\mathcal{G}^*)$ for each $\mathcal{G}^* \in \mathcal{K}$
initialize $\mathbb{G} \leftarrow \emptyset$
add the feasible candidate with largest positive gain under J and $\mathrm{AntiChain}$
repeat until $|\mathbb{G}| = B$ or no feasible addition improves J
$\mathbb{G}^* \leftarrow \mathbb{G}$
return $\mathbb{G}^*$

Listing 1: Actual task-export fields, with other fields omitted.

```
{
  "schema_version": "paper-stage4-task-plan.v3",
  "plan_id": "plan_1516637e8984ceee",
  "source_case_id": "task_subgraph.lr_nwpa.454a5726cc8d8424",
  "user_query": "Put my last Luigi's order in the cart."
}
```

G.3 Hidden Checks and Success Conditions

Checking the requested outcome. The expected state σ^* describes task-relevant conditions rather than an exact copy of all app data. For the reorder request, the relevant outcome is that the cart contains the items of the latest Luigi's order. Opening the order history or producing a reassuring final answer is insufficient. Conversely, the evaluator need not demand that the agent reproduce every navigation step in a reference workflow.

Reading an evaluator specification. Listing 5 extracts one action gate and one leaf criterion from the available construction export, retaining their original field names and values. The gate specifies the required action type and the evidence sources; `on_missing` assigns zero action credit when no matching completed action is found. The criterion's `target` selects the evidence object, `field` selects a value within that object, and `eval` specifies the comparison. Here, the check compares `order.restaurant` with `Luigi's Pizzeria`. Other criteria in the same plan check whether the order is the most recent and whether its items appear in the cart. The displayed restaurant check alone therefore cannot establish task success.

Construction checks versus active scoring. An exported evaluation plan may contain more checks than the runtime activates. For the current 281-task release, the benchmark statistics identify 474 active final-state scoring points, whereas 14 read-only points and 36 answer/report-quality points are listed separately. The 13 tasks originally labeled hybrid retain answer checks under `report_quality_checks`, while their active scoring type is execution. Consequently, the presence of a reference response or a report check must not be interpreted as evidence that it contributes

Listing 2: Schematic initial state; dates and identifiers are illustrative.

```
{
  "orders": [
    {
      "id": "order_17", "restaurant": "Luigi's Pizzeria",
      "placed_at": "2026-09-20T18:00:00",
      "items": [
        {"name": "Margherita Pizza", "quantity": 2},
        {"name": "Garlic Bread", "quantity": 1},
        {"name": "Coca-Cola 1.5L", "quantity": 1}
      ]
    },
    {
      "id": "order_12", "restaurant": "Luigi's Pizzeria",
      "placed_at": "2026-09-18T18:00:00",
      "items": [{"name": "Margherita Pizza", "quantity": 1}]
    }
  ],
  "cart": {"items": []}
}
```

Listing 3: Schematic JSON tool schema, not an extracted runtime definition.

```
{
  "name": "add_order_to_cart",
  "description": "Add the items of an existing order to the cart.",
  "parameters": {
    "type": "object",
    "properties": {
      "order_id": {
        "type": "string",
        "description": "ID of the selected historical order."
      }
    },
    "required": ["order_id"], "additionalProperties": false
  }
}
```

to the reported SR or Completion. The current release also reports no tasks requiring an additional LLM judge; the richer earlier construction format should not be mistaken for the final scoring policy.

G.4 From an Episode to the Reported Metrics

The evaluator associates execution evidence with the task's active checks. SR records whether all required active checks pass; Completion measures partial satisfaction according to the active scoring policy. Thus, choosing the right restaurant but the wrong historical order can satisfy a local check while failing the task. The weights stored in a construction-time criterion tree do not, by themselves, establish how the final runtime normalizes or aggregates those checks.

For reproducibility, the execution record needs to identify the task and model, preserve tool requests and responses, and associate the final state with the scoring output. Tool-call counts and token usage describe the cost of that same episode. The final harness configuration is also needed to specify the action limit, termination conditions, invalid-call handling, and token-accounting convention. These settings and the executable tool environment are separate from the static task/evaluator JSON illustrated above.

Listing 4: Illustrative tool request and response; this is not a measured trajectory.

```
{
  "request": {
    "name": "add_order_to_cart",
    "arguments": {"order_id": "order_17"}
  },
  "response": {
    "status": "ok", "order_id": "order_17",
    "cart_items": [
      {"name": "Margherita Pizza", "quantity": 2},
      {"name": "Garlic Bread", "quantity": 1},
      {"name": "Coca-Cola 1.5L", "quantity": 1}
    ]
  }
}
```

Listing 5: Extracted construction-time gate and criterion. This is a fragment of the earlier evaluation-plan format, not the final runtime policy.

```
{
  "plan_id": "plan_1516637e8984ceee",
  "action_id": "action_01",
  "action_type": "add_order_to_cart",
  "gate": {
    "action_type": "add_order_to_cart",
    "evidence_sources": ["agent_trajectory", "final_device_state"],
    "completion_required": true,
    "match_policy": "best_matching_completed_action",
    "consume_match": true,
    "on_missing": "zero_action_score"
  },
  "criterion": {
    "criterion_id": "criterion_01_01_01",
    "description": "The order is from Luigi's Pizzeria",
    "target": "matched_action",
    "field": "order.restaurant",
    "weight": 0.16666667,
    "eval": {
      "method": "exact_match",
      "type": "equals",
      "content": "Luigi's Pizzeria"
    },
    "children": []
  }
}
```