vSpeedUI: Boosting Mobile GUI Agents via Reusable Transition Graphs

Xiaohan Zheng[1] , **Yihong Chen**[1] , **Haiquan Qiu**[1] , **Quanming Yao**[1]
[1]Department of Electronic Engineering, Tsinghua University

Abstract

LLM-based mobile GUI agents automate smartphone tasks, but the step-by-step perception and reasoning loop causes high latency. We present v**SpeedUI**, an experience-driven system that reuses past trajectories to accelerate execution. vSpeedUI builds a Reusable Transition Graph (RTG) whose nodes represent discriminative UI states and whose edges store actions as Semantic Step Summaries capturing step intents and explicit state-level preconditions rather than brittle UI coordinates. At task start, vSpeedUI executes Global Look-ahead Planning to retrieve candidate RTG steps and batch-validates and batch-ranks candidate transitions to compile a pre-verified plan. Execution then follows lightweight graph traversal with state localization and action remapping, falling back to a base agent when reuse is infeasible. On HarmonyOS, vSpeedUI reduces inference latency and total time while preserving strong success rates. Code, extended paper and **Demo Video** are available at https://github.com/lgssstsp-gmail/vSpeed_repo.

1 Introduction

LLM-based mobile GUI agents automate smartphone tasks by mapping multimodal UI observations to executable actions across interaction paradigms, ranging from vision-grounded action prediction to structured text/graph-based element selection [Jiang *et al.*, 2025; Zhou *et al.*, 2025; Zhang *et al.*, 2025; Qin *et al.*, 2025; Ye *et al.*, 2025]. Their progress is commonly assessed in realistic interactive environments and robustness benchmarks [Zhou *et al.*, 2023; Rawles *et al.*, 2024; Chen *et al.*, 2025]. Despite significant optimizations in model-side efficiency [Yang *et al.*, 2025; Christianos *et al.*, 2024], RL-driven prediction [Lu *et al.*, 2025; Yan *et al.*, 2025; Hu *et al.*, 2025], and verifier-driven pipelines [Dai *et al.*, 2025], most current systems still follow a rigid step-wise perception-reasoning-action loop: they perceive the screen, invoke LLMs for the next micro-action, and repeat.

Latency thus scales linearly with task length, limiting real-time deployment; to break this, *memory matters*. Like human "muscle memory", agents should reuse successful trajectories to avoid repeated perception and reasoning [Memon *et al.*, 2003; Amalfitano *et al.*, 2011; Yang *et al.*, 2018; Lee *et al.*, 2023]. However, effective reuse is non-trivial: existing approaches like in-context demonstrations [Lee *et al.*, 2024], or per-step retrieval [Guan *et al.*, 2025; Chen *et al.*, 2025] are brittle under UI variation and hard to validate for new goals, relying on superficial similarities rather than semantic preconditions.

Thus, we present vSpeedUI, boosting efficiency and performance via *structured experience reuse*. We decouple high-level planning from device-level perception using Semantic Step Summaries and explicit state preconditions. We organize them into a Reusable Transition Graph (RTG) with nodes as discriminative UI states and edges as reusable actions.

On top of the RTG, vSpeedUI provides robust execution by combining state matching with action remapping when element positions shift. Crucially, at task initialization, vSpeedUI performs Global Look-ahead Planning. Unlike exhaustive search, this acts as a parallel feasibility check; it retrieves candidate transitions and performs batch-validation and batch-ranking to compile a plan that aligns with both explicit preconditions and implicit task logic. During runtime, the agent follows this plan, with safe fallback when reuse is infeasible. Our contributions are summarized as follows.

- We propose an experience-centric execution paradigm for mobile GUI agents, replacing step-by-step LLM reasoning with reusable semantic transitions and enabling Global Look-ahead Planning at initialization.
- We introduce an RTG where actions are stored as semantic intents with state preconditions rather than fixed UI coordinates, allowing cached transitions to be reused across tasks and interface variations.
- Experiments on real devices show substantial reductions in LLM inference latency and task total time, with high reuse rates and maintained or improved success rates under dynamic perturbations.

2 System Design

Figure 1 illustrates the overall framework of vSpeedUI. The system consists of two core modules:

1). *RTG Construction*: It organizes historical trajectories into an RTG for robust reuse under interface variations.

2). *Experience-Driven Execution*: It reuses RTG transitions with state matching and action remapping. At task initialization, the Global Look-ahead Planning batch-validates

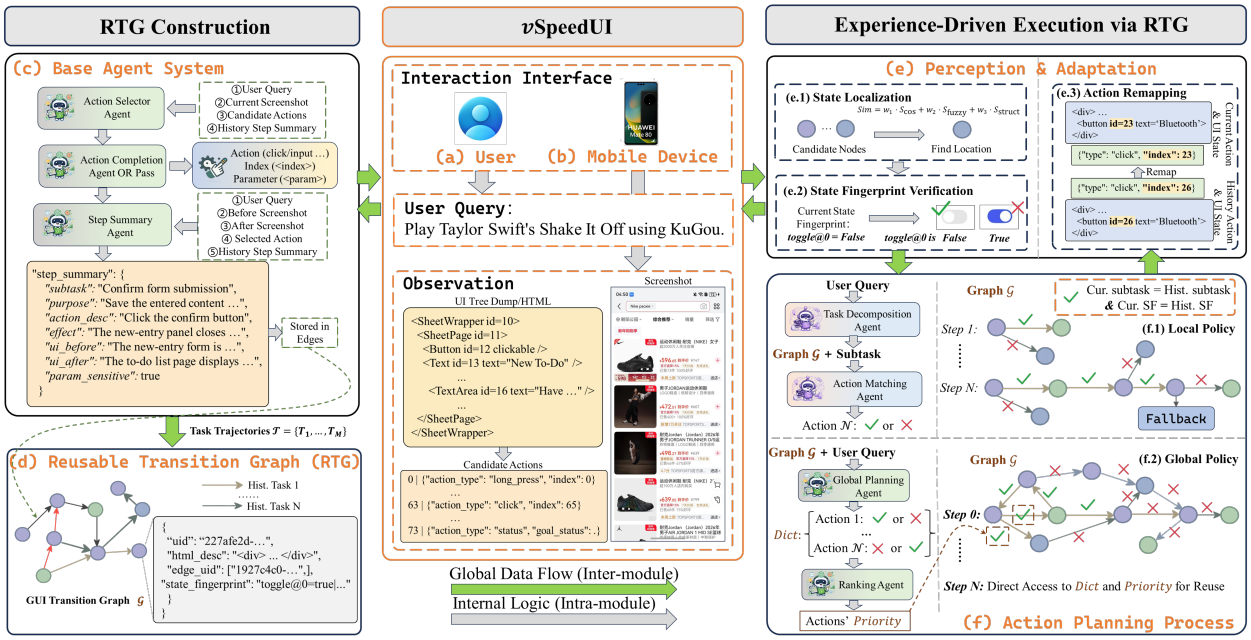

Figure 1: Overview. vSpeedUI boosts mobile GUI agents via structured experience reuse with RTG. **Left (RTG Construction):** historical trajectories are converted into RTG transitions annotated with Semantic Step Summaries. **Middle (Environment):** collects user queries and device observations (UI dump tree/screenshot). **Right (Experience-Driven Execution via RTG):** to make RTG reuse reliable under UI changes, we perform state localization, state fingerprint verification, and context-aware action remapping. The Global Policy executes Look-ahead Planning to batch-validate and batch-rank reusable steps at Step 0 to compile a pre-verified plan,

candidate steps to compile a pre-verified plan, reducing runtime to lightweight graph execution.

Reusable Transition Graph (RTG) Construction. We organize historical execution trajectories into a *Reusable Transition Graph (RTG)* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (Fig. 1d). Trajectories are collected by our Base Agent (Fig. 1c) that interacts with the device and user to produce step sequences [Dai *et al.*, 2025]. For each action a_t taken at state s_t, we generate a *Semantic Step Summary* to decouple replayable operations from raw UI details. Specifically, it abstracts raw events into (i) `subtask`, a step intent for cross-task matching, and (ii) `ui_before`, explicit state-level preconditions that specify the required UI context for valid execution.

Given a collected trajectory, we incrementally insert it into the RTG. Each node $v \in \mathcal{V}$ denotes a UI state and stores its UI tree/HTML snapshot plus a *State Fingerprint* that captures task-relevant dynamic attributes (Fig. 1d). This fingerprint enables robust deduplication across repeated runs: a new node is created only when the fingerprint or semantic evidence indicates a meaningful state change. Each directed edge $e \in \mathcal{E}$ denotes a transition induced by an action, and stores its *Semantic Step Summary*, executable parameters, and *origin-task metadata*. When ingesting new trajectories, we merge transitions that match existing nodes/edges and append unseen branches, so graph coverage grows continuously with accumulated experience.

Experience-Driven Execution. We treat the constructed RTG as a library of reusable semantic transitions. At runtime, vSpeedUI relies on three key mechanisms: grounding current observations to the RTG, ensuring execution safety via fallback mechanisms, and policies for reuse decision.

1) Perception & Adaptation (Grounding). This layer (Fig. 1e) ensures reliable reuse under dynamic UI changes. We first perform *State Localization* to map the current UI to candidate RTG nodes via stable UI-structure similarity. Then, *State Fingerprint Verification* is applied to confirm task-critical micro-states (e.g., whether the Bluetooth switch is already toggled in a "Turn on Bluetooth" task). During execution, we utilize *Context-Aware Action Remapping* to locate targets under UI drift, enabling stored transitions to remain executable beyond brittle coordinates.

2) Reuse vs. Fallback (Safety). Grounding serves as a safety gate. If the localized state mismatches or step preconditions do not hold (e.g., no valid outgoing transition exists), vSpeedUI aborts reuse and falls back to the Base Agent. This conditional acceleration prevents error accumulation rather than enforcing reuse as a hard constraint.

3) Local vs. Global (Reuse Decision Policy). While the Local Policy (Fig. 1 f.1) performs reactive step-wise matching, the Global Policy (Fig. 1 f.2) shifts reasoning to **Step 0**. It retrieves candidate edges and performs **Global Look-ahead Planning** to batch-validate their semantic alignment with the user goal. If ambiguity arises where multiple valid edges coexist at the same state (e.g., opening different apps on the home screen for cross-app tasks), we invoke a *Ranking Agent* to perform *parallel batch-ranking* to align the execution with the *implicit task logic* based on the global goal and *origin-task context*. By shifting logic to a parallelized batch process, the Global Policy enables vSpeedUI to resolve long-term intent at initialization, avoiding the linear latency accumulation inherent in traditional step-wise perception-reasoning loops.

3 Experiments

Settings. We evaluate vSpeedUI on a Huawei Mate 80 smartphone running HarmonyOS 6. Our testbed consists of 10 mobile tasks spanning lifestyle social media (Xiaohong-

shu), music (QQ Music), e-commerce (JD.com), system applications (Memo), and so on, with an average optimal trajectory length of 8.5 steps, and each task is repeated for 15 trials. We compare vSpeedUI against SOTA LLM-based GUI agents, including UI-TARS [Qin *et al.*, 2025], Mobile-Agent-v3 [Ye *et al.*, 2025], and GPT-5-based naïve vision/text agents (Vision: predicting coordinates from screenshots; Text: selecting elements via UI tree dump/HTML). We report four key metrics in Table 1 and Table 2: (i) Success Rate (SR): Percentage of tasks completed within $1.5\times$ the ideal step count. (ii) LLM Inference Latency (IL): The time latency incurred by LLM inference. (iii) Reuse Rate (RR): The ratio of steps driven by the graph versus the optimal step count. (iv) Total Time (TT): The end-to-end time to complete the tasks.

Table 1: Main Results.

Method	SR (%)↑	IL (s/step)↓	TT (s/task)↓	RR (%)↑
UI-TARS (72B)	44.7	17.5	191.4	-
MA-v3 (32B)	50.7	27.7	276.9	-
GPT-5 (Vision)	47.3	18.1	189.1	-
MA-v3 (GPT-5)	54.0	56.4	515.0	-
GPT-5 (Text)	64.0	20.6	215.3	-
vSpeedUI (GPT-5)	93.3	21.4 /task	79.5	93.2

Notes. Total time accounts for system overhead, averaging 4.55s/step, which covers hardware-level operations such as screenshotting, UI dumping, and environment settling time. And method "MA-v3" is "Mobile-Agent-v3".

Benchmarking Comparison. Table 1 shows that existing SOTA agents remain limited by step-wise decision making: while their LLM IL is moderate, their end-to-end Success Rates stay below 64% and Total Time is still in the 190–515s range. In contrast, vSpeedUI (GPT-5) achieves a 93.3% Success Rate and reduces Total Time to 79.5s. This gain is enabled by structured experience reuse, which replaces repeated per-step reasoning with pre-validated transitions and yields a high Reuse Rate of 93.2%.

Table 2: Efficiency with Reuse policies under Same-query Reuse.

vSpeedUI Reuse Policy	SR (%)↑	IL (s) ↓	TT (s/task)↓	RR (%)↑
None (GPT-5)	76.7	59.5 /step	545.4	–
Local (Qwen2.5-7B)	82.7	1.1 /step	140.5	72.9
Global (GPT-4)	86.0	11.7 /task	106.8	84.8
Global (GPT-5)	93.3	21.4 /task	79.5	93.2

Effectiveness of Reuse Policies. Table 2 ablates reuse policies within vSpeedUI. Without reuse (*None*, i.e., our step-wise Base Agent), the agent attains 76.7% SR but incurs prohibitive per-step LLM latency (59.5s/step) and 545.4s per task. Local reuse improves efficiency (140.5s) with moderate SR (82.7%), while Global reuse further boosts both reliability and speed. Using **Global (GPT-5)**, vSpeedUI reaches 93.3% SR and 79.5s total time, by shifting LLM inference to a one-time initialization (s/task) and achieving 93.2% reuse.

Handling Query Variations. Table 3 evaluates four query perturbations: *Synonym Paraphrasing (Para.)*, *Step Addition/Deletion (Decr./Incr.)*, and *Parameter Perturbation (Param.)*. And "Qwen2.5" is "Qwen2.5-7B-Instruct". Across both tasks, RTG reuse remains robust: Local and Global policies retain high SR while substantially reducing TT versus the Base Agent. Global policy is the most stable, achieving perfect SR on Memo and the lowest TT (e.g., ∼42–45s under Para./Decr./Param.). The main stress case is *step increase* on XHS, where RR drops and Local policy SR degrades to 60.0%, yet Global still preserves 85.7% SR with far lower TT.

Table 3: Robustness results under query perturbations.

		Memo (Simple)			XHS (Hard)		
Method	**Type**	**RR↑**	**SR↑**	**TT↓**	**RR↑**	**SR↑**	**TT↓**
Local (Qwen2.5)	Para.	85.7	88.7	80.2	72.7	73.3	180.0
	Decr.	85.7	93.3	78.9	72.7	86.7	179.7
	Incr.	71.4	88.7	124.3	54.6	60.0	264.0
	Param.	85.7	93.3	76.8	63.6	73.3	234.4
Global (GPT-5)	Para.	100.0	100.0	44.3	84.6	85.7	135.0
	Decr.	100.0	100.0	44.0	76.9	92.9	171.7
	Incr.	71.4	100.0	128.8	69.2	85.7	207.6
	Param.	100.0	100.0	42.1	76.9	92.9	170.1
Base Agent	Mixed	n/a	93.3	334.6	n/a	73.3	603.0

Note: RR: Reuse Rate (%), **SR**: Success Rate (%), **TT**: Total Time (s).

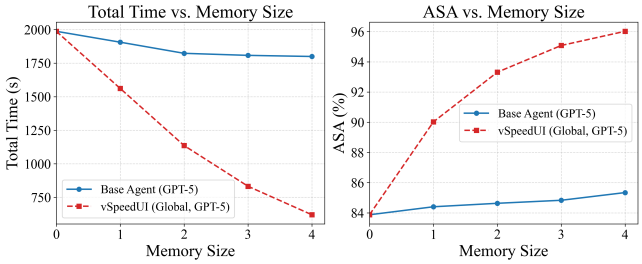

Figure 2: Action Selection Accuracy and time vs. memory size M.

Scaling with More Experience. We evaluate how vSpeedUI scales when the RTG accumulates four diverse tasks (M=1 . . . 4): social media search (Xiaohongshu), task recording (Memo), music playback (QQ Music), and e-commerce filtering (JD.com). We test the agent on a new, long-horizon cross-app query (e.g., deciding whether to purchase Michael Jackson's *Thriller*). Fig. 2 shows that as M increases, Total Time drops from 1987s to 620s and Action Selection Accuracy (ASA) rises to 96.02%.

Demonstration. Our **Demo Video** illustrates vSpeedUI solving unseen, complex tasks by dynamically "stitching" transitions from different tasks, showcasing semantic recombination and acceleration over rote replay.

4 Conclusion

In this paper, we present vSpeedUI, a structured experience reuse framework for mobile GUI agents. By organizing past trajectories into a Reusable Transition Graph and validating reusable plans at task start, it replaces repetitive step-wise reasoning with lightweight graph execution. Experiments on real devices show substantial reductions in latency and task time while maintaining strong success rates.

References

[Amalfitano *et al.*, 2011] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261. IEEE, March 2011.

[Chen *et al.*, 2025] Weizhi Chen, Ziwei Wang, Leyang Yang, Sheng Zhou, Xiaoxuan Tang, Jiajun Bu, Yong Li, and Wei Jiang. Pg-agent: An agent powered by page graph, 2025.

[Christianos *et al.*, 2024] Filippos Christianos, Georgios Papoudakis, Thomas Coste, Jianye Hao, Jun Wang, and Kun Shao. Lightweight neural app control, 2024.

[Dai *et al.*, 2025] 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, 2025.

[Guan *et al.*, 2025] Ziyi Guan, Jason Chun Lok Li, Zhijian Hou, Pingping Zhang, Donglai Xu, Yuzhi Zhao, Mengyang Wu, Jinpeng Chen, Thanh-Toan Nguyen, Pengfei Xian, Wenao Ma, Shengchao Qin, Graziano Chesi, and Ngai Wong. Kg-rag: Enhancing gui agent decision-making via knowledge graph-driven retrieval-augmented generation. In *Proc. EMNLP*, pages 5396–5405. Association for Computational Linguistics, 2025.

[Hu *et al.*, 2025] Zhiyuan Hu, Shiyun Xiong, Yifan Zhang, See-Kiong Ng, Anh Tuan Luu, Bo An, Shuicheng Yan, and Bryan Hooi. Guiding vlm agents with process rewards at inference time for gui navigation, 2025.

[Jiang *et al.*, 2025] Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, Joey Tianyi Zhou, and Chi Zhang. Appagentx: Evolving gui agents as proficient smartphone users, 2025.

[Lee *et al.*, 2023] 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, 2023.

[Lee *et al.*, 2024] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 1119–1133, 2024.

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

[Memon *et al.*, 2003] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. Dart: a framework for regression testing "nightly/daily builds" of gui applications. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, ICSM-03, pages 410–419. IEEE Comput. Soc, 2003.

[Qin *et al.*, 2025] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. Ui-tars: Pioneering automated gui interaction with native agents, 2025.

[Rawles *et al.*, 2024] 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, 2024.

[Yan *et al.*, 2025] Haolong Yan, Yeqing Shen, Xin Huang, Jia Wang, Kaijun Tan, Zhixuan Liang, Hongxin Li, Zheng Ge, Osamu Yoshie, Si Li, Xiangyu Zhang, and Daxin Jiang. Gui exploration lab: Enhancing screen navigation in agents via multi-turn reinforcement learning, 2025.

[Yang *et al.*, 2018] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. *Automated Software Engineering*, 25(4):833–873, June 2018.

[Yang *et al.*, 2025] Zhen Yang, Zi-Yi Dou, Di Feng, Forrest Huang, Anh Nguyen, Keen You, Omar Attia, Yuhao Yang, Michael Feng, Haotian Zhang, Ram Ramrakhya, Chao Jia, Jeffrey Nichols, Alexander Toshev, Yinfei Yang, and Zhe Gan. Ferret-ui lite: Lessons from building small on-device gui agents, 2025.

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

[Zhang *et al.*, 2025] Zhong Zhang, Yaxi Lu, Yikun Fu, Yupeng Huo, Shenzhi Yang, Yesai Wu, Han Si, Xin Cong, Haotian Chen, Yankai Lin, Jie Xie, Wei Zhou, Wang Xu, Yuanheng Zhang, Zhou Su, Zhongwu Zhai, Xiaoming Liu, Yudong Mei, Jianming Xu, Hongyan Tian, Chongyi Wang, Chi Chen, Yuan Yao, Zhiyuan Liu, and Maosong Sun. Agentcpm-gui: Building mobile-use agents with reinforcement fine-tuning, 2025.

[Zhou *et al.*, 2023] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2023.

[Zhou *et al.*, 2025] Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. Mai-ui technical report: Real-world centric foundation gui agents, 2025.

A Appendix

B Ethical Statement

There are no ethical issues.

C Acknowledgments

This work is supported by ...

D Detailed Introduction of Method

This appendix details how vSpeedUI is deployed in practice, focusing on the implementation-oriented components that are only briefly described in the main text.

Importantly, our method is orthogonal to existing agent designs: it is a modular, plug-and-play layer between UI observation and action execution, reusing actions while falling back to an arbitrary, swappable base agent (any underlying model/system) when reuse is uncertain.

Following a top-down narrative, we introduce (i) grounding (state alignment and verification), (ii) safety mechanisms (fallback and termination decisions), (iii) reuse decision policies (reuse/skip/parameter edits), (iv) global look-ahead planning and ambiguity resolution, and (v) RTG construction and update.

D.1 Grounding: Matching (State Localization), Re-localization (State Re-Localization), and Fingerprint Verification

Page representation. Each UI state is represented by a UI tree dump (denoted as h); we optionally store a screenshot for future multi-modal extensions. We maintain a Reusable Transition Graph (RTG) whose nodes store the page dump h and whose edges store actions. Grounding answers: "which RTG node matches the current on-device page?"

Matching score used in practice. Given two UI dumps h_a, h_b, our implementation computes three similarity terms: (i) an embedding cosine similarity s_{emb} from a pluggable text embedder; (ii) a fuzzy string match score s_{fuzzy} over extracted stable texts; and (iii) a structure token Jaccard similarity s_{struct} (tag+cls tokens). We fuse them as

$$S(h_a, h_b) = w_{\text{emb}} \cdot s_{\text{emb}} + w_{\text{fuzzy}} \cdot s_{\text{fuzzy}} + w_{\text{struct}} \cdot s_{\text{struct}},$$

and accept a match only if all three pass thresholds (cosine distance, fuzzy ratio, and structure ratio). This corresponds to the `compare_html_similarity()` routine in our code. In practice, we assign a relatively larger weight to s_{struct}, since structural similarity better reflects the stable page skeleton, whereas text and embedding cosine similarity are more easily affected by dynamic content.

(1) Localization along the expected trajectory (fast check). Suppose the agent has already matched the current RTG node v_t and has selected a reusable edge e_t. We predict the expected next node $\hat{v}_{t+1}$ as the endpoint of e_t and verify whether the current UI dump h_{cur} matches the stored dump $h(\hat{v}_{t+1})$ using the above score. In deployment we set *looser* thresholds for this along-trajectory check, because reuse makes $\hat{v}_{t+1}$ relatively certain and we mainly need to rule out gross mismatches. This realizes the "page matching along a trajectory is looser" principle.

(2) Re-localization in the whole RTG (top-k retrieval + strict verify). If the expected-node verification fails, we run *graph-wide* retrieval to obtain top-k candidate nodes:

$$\mathcal{C} = \text{TopK}\left(\texttt{VectorStore.search}(\text{emb}(h_{\text{cur}}))\right),$$

and then apply a stricter verification on each candidate with `compare_html_similarity()`. This re-localization is stricter because it must disambiguate among many similar pages in the entire RTG, matching the "state localization is a stricter, advanced version of page matching" requirement.

(3) Task-relevant state fingerprint (hard gate). Some pages share an almost identical UI skeleton but differ in a task-relevant hidden state (e.g., a toggle is ON/OFF, a favorite button is active/inactive). Our implementation extracts a *state fingerprint* $f(h)$ from checkable widgets in the UI dump, e.g., mapping row labels to `Toggle.checked` values, producing a deterministic string like `Bluetooth=true|...`. The core rule is a *hard gate*:

$$\text{if } f(h_{\text{cur}}) \neq f(h(v)),$$

then do NOT merge/match even if

$$S(h_{\text{cur}}, h(v)) \text{ is high.}$$

This logic is used during node de-duplication and matching (`old_state_fp != new_state_fp` $\Rightarrow$ split node).

Third-party apps without state exposure (and our current protocol). For some third-party apps, task-relevant states are not exposed in the UI tree (e.g., "liked" vs "not liked" may produce the same dump), making $f(h)$ incomplete. In our current experiments we *avoid* such cases to ensure fair evaluation, but the issue is engineering-solvable: when we detect "same UI dump" but the action is semantically effective, we can attach a lightweight *visual state fingerprint* computed from a small Region of Interest (ROI) patch around the acted widget (e.g., perceptual hash on a cropped region) and inject it into $f(\cdot)$ as an additional component. This preserves our conservative reuse policy: reuse only when both skeleton and task-relevant state match.

D.2 Safety: Fallback

Decision categories. At runtime, the agent's decision is not about selecting one action from many; instead, each step provides a *single candidate action* (from the retrieved RTG edge), and the agent decides *whether this candidate can be safely reused* (or should be rejected and handled by the none policy (the base agent)). In addition, the agent must decide "whether to fallback to a more general (vSpeedUIwith None Policy/Base Agent)". Concretely, whenever grounding/verification/remapping fails, we abort reuse and fallback to the base agent that performs stronger per-step reasoning and perception.

Why fallback is essential. Experience reuse is intentionally conservative: if a step is uncertain (e.g., page mismatch, ambiguous candidates, or remap failure), the system prefers correctness over speed by switching to the base agent (vSpeedUIwith None Policy), which provides robustness on unseen layouts and dynamic content. This corresponds to the "reuse vs fallback" safety framing in the writing.

D.3 Reuse Decision: Implemented Decisions and Extensible Options

For each candidate reusable step, the planner outputs one of: (i) **Reuse** (execute as-is), (ii) **Skip** (logically redundant for the new goal), (iii) **Fallback** (cannot reuse and defer to the base agent), or (iv) **Parameter-slot edit** (reuse the same action schema but modify input parameters). [1]

Skip for goal-truncation and same-page redundancy. Some historical steps are task-related but unnecessary for the new goal (e.g., historical trajectory includes both "like" and "collect," while the new goal only requires "collect"). These steps occur on the same functional page and should be skipped without hurting the final outcome.

Parameter-slot editing. If a step is structurally reusable but contains goal-dependent parameters (e.g., search keywords, input text), we keep the executable action schema unchanged and only replace the parameter values. This supports robust reuse across "same subtask, different arguments".

D.4 Global Policy: Global Look-ahead Planning and Ambiguity Resolution

Why global planning. A key engineering advantage is shifting expensive reasoning to Step 0 by validating many historical steps *in parallel*, instead of doing step-wise sequential deliberation.[2] This reduces online decision overhead and improves stability under long-horizon tasks.[3]

Parallel step validation at initialization. At task start, the planner (the Global Planning Agent) evaluates each historical step against the new goal and produces a per-edge reuse decision (reuse / reject / skip / parameter edit). The main paper describes this as "global look-ahead"; here we emphasize it is implemented as batched LLM calls (one prompt per step), enabling near-constant decision latency w.r.t. horizon.

Ranking agent for multi-valid edges. When multiple valid outgoing edges coexist at the same RTG node (e.g., several icons or entry points on the home screen), we invoke an auxiliary ranker (the Ranking Agent) to parallelly decide an execution order consistent with the *implicit task logic*.

Why ranking is necessary (beyond state-only validation). Our parallel Step 0 validation assumes that the verified UI representation (UI dump + state fingerprint) is a sufficiently informative *state summary*, so that the reuse feasibility of an edge can be judged mainly from the current state and the new goal. However, some tasks impose *history-dependent* or *order-sensitive* constraints that are not fully observable from the current page alone (e.g., "do A before B", where whether A has been satisfied is not explicitly encoded in the UI dump). In the RTG, this often appears as a single node having multiple outgoing edges that are all individually reusable under state matching, yet only a subset (or a specific order) leads to successful task completion.

How we resolve the ambiguity. We therefore apply a ranking step that scores candidate edges under the *global task context* and returns a priority order list for execution. This ranking is implemented as parallel scoring (one lightweight scoring prompt per candidate edge), and serves as an additional safeguard that complements localization/fingerprint verification by enforcing goal-consistent action ordering.

D.5 Local Policy: Step-by-Step Reuse Decision

Unlike the global policy, the local policy performs reuse decision at each step. Its main advantage is efficiency: since it only needs to reason over the current local context, it can be deployed with a lighter-weight model, making it more suitable for resource-constrained settings and reducing runtime latency. Concretely, the local policy follows a two-stage hierarchical procedure. A *Task Decomposition Agent* first decomposes the user query into the current subtask to be completed. Given the current subtask, an *Action Matching Agent* determines whether the current reusable action matches the subtask, and whether the subtask has already been completed. Once the agent judges that the current subtask is finished, control returns to the Task Decomposition Agent, which outputs the next subtask. The two agents therefore alternate throughout execution in a step-by-step manner, forming a lightweight local reuse policy.

D.6 Reusable Transition Graph Construction and Update

Unified build/update logic. RTG construction and update share the same procedure: we insert each recorded trajectory into the graph one node at a time; insertion already includes

[1] In principle, the decision space can be further extended with additional options that are not implemented in the current system: **(1) State rollback**: when a task contains repeated reversible operations (e.g., toggling the same setting multiple times), the agent could reuse a stored reverse transition to return to a previously verified state instead of re-planning from scratch. For example, if a task repeatedly toggles the same UI control, the system could roll back to an earlier verified state rather than replaying the entire sequence. **(2) Delayed subtask completion**: some tasks may require completing an additional subtask in the middle of execution, and the actions for this subtask are not present in memory. (In the query perturbation experiments reported in the main text, the inserted step is added at the end of the task.) For example, a "collect" action may depend on first completing a "like" action. In such cases, the agent could temporarily defer the current reusable step, complete the required subtask, and then resume the original goal.

[2] We currently implement the *deformable prompt* formulation (one prompt per step/edge) to enable parallel validation. Further system optimizations are orthogonal and left for future work, including (i) prefix-sharing for prompts with long common prefixes, and (ii) stream-style execution that starts acting once early validated steps return while remaining validations complete in the background.

[3] In practice, we do not validate *all* edges in a large RTG at Step 0 since many are irrelevant to the current goal. Two resource-saving variants are natural: (i) a coarse pre-filter that selects a small candidate set of edges/nodes before LLM validation; and (ii) partial look-ahead around the currently localized node (e.g., validating actions within depth-d neighborhoods) in a walk-and-validate manner. If the agent falls back to the base policy and "teleports" to an unexpected node, the local neighborhood changes and validation may be re-triggered. To mitigate this, we can use a fast **Local Policy** (small model) for quick, on-the-fly reuse decisions, while overlapping large-model validation with inevitable system overheads (e.g., screenshot capture and UI-tree dumping), which already take several seconds per step in real-device execution.

Table 4: Main results on a HUWEI MATE 80.

Method	SR (%)↑	IL (s/step)↓	Total Time (s/task)↓	Reuse (%)↑
UI-TARS (72B)	44.7	17.5	191.4	–
Mobile-Agent-v3 (32B)	50.7	27.7	276.9	–
GPT-5 (Naïve Vision)	47.3	18.1	189.1	–
Mobile-Agent-v3 (GPT-5)	54.0	56.4	515.0	–
GPT-5 (Naïve Text)	64.0	20.6	215.3	–
Our Base Agent (GPT-5 Text)	76.7	①26.3; ②31.3; ③14.4†	545.437	–
Results with Same-query Reuse				
Method	**Δ SR (%)↑**	**IL (s/step OR s/task)**	**Total Time**	**Reuse Rate**
Our Reuse (Local, Qwen2.5)	+ 6.0	④ 0.4, ⑤ 1.4†	140.5	72.9
Our Reuse (Global, GPT-4)	+ 9.3	⑥ 6.9♣, ⑦ 4.8♣	106.8	84.8
Our Reuse (Global, GPT-5)	+ 16.6	⑧ 13.7♣, ⑨ 7.7♣	79.5	93.2

Notes.
Total time accounts for **system overhead**, averaging 4.55s/step, which covers hardware-level operations such as screenshotting, UI dumping, and environment settling time. †: sporadic — triggered only when the current action requires completion (i.e., not invoked at every step). ♣: one-time upfront — invoked mainly for global planning and global ranking at the beginning of a task, rather than per step. **Different agents from (Fig.1)**: ① Action Selection Agent (Fig.1 c); ② Step Summary Agent (Fig.1 c); ③ Action Completion Agent (Fig.1 c); ④ Action Matching Agent (Fig.1 f.1); ⑤ Task Decomposition Agent (Fig.1 f.1); ⑥ and ⑧ Global Planning Agent (Fig.1 f.2); ⑦ and ⑨ Global Ranking Agent (Fig.1 f.2);

the "update" behavior because every new node is matched against existing nodes before creating a new one.

Node merge and redundancy pruning. For each incoming node, we first retrieve similar nodes and compare structure; if candidates are highly similar, we further call an LLM to decide whether they represent the same page/state under task context. If an action results in no meaningful UI change (e.g., redundant click), we treat it as a redundant step and prune it instead of adding a new transition.

Extensibility beyond text skeletons. Although our current implementation uses text-based HTML skeletons as the page invariant, the same pipeline generalizes to visual invariants (stable layout regions) and even hybrid judgments. This supports future integration of vision-anchored matching and simplifies remapping when UI trees are incomplete.

E Experiments

E.1 Datasets

We evaluate on 10 real-device tasks, each specified by a natural-language user query. Since our deployment environment is Chinese, the agent receives the **original Chinese** instructions at runtime. For readability and reproducibility under standard `pdflatex` compilation, we report the **English translations** of all queries in Table 5.

Starting state and app launching. All tasks start from the phone home screen (no app is pre-opened). To mimic realistic user behavior and avoid relying on privileged app-launching APIs (e.g., direct package-level launching that may require device-specific registries or extra preprocessing), the agent navigates the home screens to locate the target app icon and then enters the app. In our setup, apps may appear on different home-screen pages, so launching may involve up to two swipe operations before tapping the app icon.

Main Results.

E.2 Fine-grained Latency Breakdown of Reuse Policies

To better understand where the runtime savings come from, Table 4 provides a fine-grained latency breakdown of the major agents used by the Base (None Policy), Local Policy, and Global Policy. This table decomposes the inference cost into individual agent calls and distinguishes between three invocation patterns: *per-step*, *sporadic*, and *one-time upfront*.

For the Base Agent (None Policy), the dominant cost comes from repeated step-wise reasoning. In particular, the *Action Selection Agent* and *Step Summary Agent* are invoked frequently during execution, while the *Action Completion Agent* is triggered only when the current action requires explicit completion checking. This explains why the Base Agent, despite being robust, incurs the highest total runtime.

For the Local Policy (Here, we do not report the time that fall back to the Base Agent, since their cost is identical to that of the Base Agent itself.), the expensive general-purpose reasoning loop is replaced by lightweight step-wise reuse decisions. Concretely, the system mainly relies on the *Action Matching Agent* and the *Task Decomposition Agent*, which are substantially cheaper than the Base Agent's per-step reasoning modules. This leads to a large reduction in total task time while maintaining reasonable success rates, However, the current performance remains limited, likely due to both prompt design and the capability ceiling of the lightweight model, which we consider an important direction for future improvement.

For the Global Policy, the key design choice is to shift expensive reasoning from step-wise execution to task initialization. The *Global Planning Agent* and *Ranking Agent* are invoked only once at the beginning of the task, rather than at every step. Although this introduces a visible upfront cost, the resulting plan enables the subsequent execution to proceed mostly through lightweight graph traversal and action remapping. As a result, the global policy achieves the best overall trade-off between efficiency and reliability. In particular, the

Table 5: Evaluation queries (English translations shown). All tasks are executed with Chinese instructions in our deployment.

English query (shown)
1. Open JD.com, search for "Nike running shoes", filter results to the price range 600–1000 RMB, enter the first product, and stay on its product detail page.
2. Open QQ Music, search for "Elvis Presley", enter the artist page from the results, and view the artist's albums on the artist page.
3. Open Amap (Gaode Maps), search for directions to "Universal Beijing Resort", and stay on the route details page.
4. Open Fliggy, go to the "Flights" page, search for flights from Beijing to Shenzhen, and stay on the flight details page.
5. Open Huawei Music, search for the artist Taylor Swift, and play her song "Shake It Off".
6. Open Dianping, go to the local food/restaurant list, switch sorting to "distance", and enter one restaurant from the list.
7. Open Xiaohongshu, search for "MacBook Air M2 real experience", use the filter entry (typically in the top toolbar) to select image-text posts with the most likes, open the first post, then like and save it.
8. Open Notes, create a new to-do item with the content: "Remind myself to prepare the PPT".
9. Turn on Bluetooth.
10. Open the Weather app, search for "guangzhou" in city management, add Guangzhou to the city list, and return to the home screen.

GPT-5-based global policy has a higher one-time planning cost than the GPT-4 variant, but this additional upfront reasoning yields a higher reuse rate and the lowest end-to-end task time.

Overall, the main efficiency gain of vSpeedUI does not come from making each individual reasoning step cheaper, but from reducing the number of reasoning steps that must be performed online. By converting repeated per-step deliberation into reusable transitions plus one-time global look-ahead planning, vSpeedUI substantially reduces total execution time while improving task success.

F Case Study

Please refer to the 5-9 minute segment of our **demo video**.