Automating the Design of Embodied Agent Architectures

Jian Zhou **Sihao Lin** **Jin Li** **Shuai Fu** **Gengze Zhou** **Qi Wu**
Australian Institute for Machine Learning, University of Adelaide, SA, Australia
{j.zhou, sihao.lin, shuai.fu, gengze.zhou, qi.wu01}@adelaide.edu.au
jinli0410.ai@gmail.com

Abstract: Embodied agents are typically built as hand-designed compositions of perception, memory, planning, and action modules. This modularity exposes a large architectural design space, but current systems still rely on researcher intuition to choose where information is stored, how observations are processed, and how model calls are connected. Agent Architecture Search (AAS) automates such design for text-domain agents, but has not been systematically evaluated on perceptual embodied agents through simulator rollouts. We study this transfer. We introduce AGENTCANVAS, a typed-graph runtime that hosts embodied executors as editable node-and-wire programs with simulator-aware execution and episode-level logs, and KDLOOP, a coding-agent search procedure that cycles through proposal, critique, experiment, and distillation, with triggered reflection after stalls. We evaluate three AAS variants across four embodied executors spanning vision-language navigation, embodied question answering, and language-conditioned manipulation. The resulting 3×4 matrix shows that architecture-level search can produce deployable and directional success-rate gains on embodied tasks, while one apparent high-scoring candidate is rejected as leak-bearing. At the same time, the experiments expose constraints that are muted in text-domain AAS: optimization signals can be masked by rollout noise, search can become trapped in local edit basins, and episode-level credit assignment only partially emerges even when detailed logs are available. These results characterize both the promise and the current limits of automated architecture search for embodied agents.

Keywords: Embodied Agents, Agent Architecture Search, LLM Agents

Project page: https://jianzhou0420.github.io/src/works/AgentCanvas/paper.html

1 Introduction

Embodied agents increasingly act by composing foundation models with perception, mapping, memory, planning, and control modules. This design pattern appears across vision-language navigation [1, 2, 3], embodied question answering [4, 5, 6], and language-conditioned manipulation [7, 8], where a system observes the world, builds or queries an internal state, calls language or vision-language models for reasoning, and converts the result into actions. The advantage of this agentic paradigm is that the architecture is explicit. Unlike end-to-end policies whose structure is absorbed into weights, these systems expose where information flows and which modules can be edited.

This explicit structure also creates a design problem. Each agent fixes choices about sensor abstractions, map representations, memory state, prompt structure, planner topology, model placement, and action interfaces. These choices are usually made by hand for a single benchmark. As foundation models and embodied tools multiply, the space of plausible architectures grows faster than manual iteration can cover. The natural question is whether architecture design itself can be automated.

Agent Architecture Search (AAS) offers one route. In text-domain agents, AAS uses an optimizer to propose and evaluate alternative LLM-agent workflows, automating choices that would otherwise

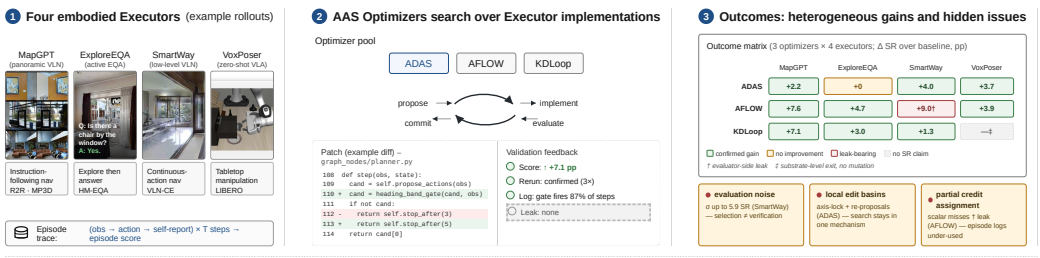

Figure 1: **Embodied Agent Architecture Search.** (1) Four seed Executors emit per-episode traces. (2) ADAS/AFlow/KDLoop edit Executor code through a shared coding-agent harness and only the proposer differs. (3) The 3×4 ΔSR matrix shows several deployable or directional improvements plus one leak-bearing apparent gain, and surfaces three constraints of embodied AAS: evaluation noise, local edit basins, and partial credit assignment.

be hand-designed [9, 10, 11, 12]. However, moving AAS to embodied agents is not a direct transfer. Text-domain AAS operates over cheap, stateless calls and a mature vocabulary of runnable primitives such as chain-of-thought, debate, or self-refinement. Embodied agents instead interact with stateful simulators, produce noisy multi-episode evaluations, and depend on long execution traces involving observations, actions, tool calls, and intermediate planner outputs. Moreover, there is no small palette of composable embodied reasoning primitives: the natural starting points are published systems such as MapGPT [3], SmartWay [13], ExploreEQA [5], and VoxPoser [8].

We therefore study embodied AAS in a *method-seeded* form. Each search session starts from a published embodied-agent architecture and searches nearby graph-level modifications, rather than assembling an agent from scratch. To make this possible, we introduce an executor substrate and an optimizer comparison harness. On the executor side, AGENTCANVAS represents an embodied agent as a typed node-and-wire graph whose modules can be edited, executed, and logged inside a simulator. On the optimizer side, KDLOOP runs each search iteration through THINK, CRITIC, EXPERIMENT, and DISTILL phases implemented with a coding-agent orchestrator, with REFLECT triggered when progress stalls. We also port ADAS [9] and AFlow [10] into the same harness for comparison.

Across four executors and three embodied task families, we find that AAS can improve embodied agents: several searched graphs obtain confirmed success-rate gains over their seeded baselines. These gains indicate functional changes in agent behavior, contrasting with recent critiques that attribute text-domain AAS gains largely to superficial workflow restructuring [14]. At the same time, the experiments show that embodied AAS is constrained by the evaluation regime. Rollout noise can obscure the effect of graph edits, search dynamics can repeatedly exploit a local edit basin rather than discovering qualitatively different mechanisms, and episode-level credit assignment only partially emerges even though detailed logs are available to the agents. Thus, the contribution is not only a set of improved executors, but a characterization of what automated architecture search must handle when moved from text programs to perceptual agents acting in 3D environments.

2 Related Work

Embodied Agent Systems. Embodied AI has studied agents that connect perception, language, and action in interactive environments, including vision-language navigation (VLN) [1], embodied question answering (EQA) [4], and instruction-following household tasks such as ALFRED [15]. A growing line of systems composes frozen LLMs and VLMs with memory, tools, geometric modules, and low-level controllers. In language-conditioned manipulation, foundation models rank skills, generate code, or produce geometric constraints for control (SayCan [16], Code-as-Policies [7], VoxPoser [8], ReKep [17]). In vision-and-language navigation, LLM-based agents reason over textual, topological, or metric scene representations (NavGPT [2], MapGPT [3], SmartWay [13]); in embodied question answering, frozen VLMs are coupled with exploration, memory, and tool use (Explore-EQA [5], GraphEQA [6], ToolEQA [18]). These agentic systems differ from embodied foundation models and end-to-end VLA policies trained or adapted on robot-action data (PaLM-

E [19], RT-1 [20], RT-2 [21], OpenVLA [22], π_0 [23]) in that their architectures remain explicit. This explicit structure exposes a search space, but existing systems still rely on hand-designed module choices for each benchmark.

Agent Architecture Search. Agent Architecture Search (AAS) automates the design of LLM-agent workflows by running an external optimizer over candidate architectures at development time. It is distinct from self-improving agents that adapt their behavior during deployment [24, 25]. ADAS introduced the analogy to neural architecture search [9], and subsequent work has explored stronger optimizers, structured workflow spaces, and multi-agent compositions (AFlow [10], AgentSquare [11], MaAS [12], GPTSwarm [26]), as well as general optimization frameworks and platforms (Trace [27], EvoAgentX [28]). This line is related to prompt, program, and LM-pipeline optimization methods such as DSPy [29], OPRO [30], and TextGrad [31]. However, AAS treats the agent architecture itself—rather than a prompt, module, or pipeline parameter—as the object of search. Recent studies show that text-domain gains can be fragile: tuned single-agent baselines can match searched workflows [14], and generated workflows may be unstable under paraphrase [32]. Our work tests AAS in a different regime: perceptual agents acting inside simulators, where architectures connect perception, memory, planning, and action modules and evaluations produce noisy multi-episode rollouts.

Coding Agents and Graph Substrates. Applying AAS to embodied agents requires candidate generation and execution substrates beyond text-domain workflow search. Prior AAS systems often generate each candidate with a single LLM completion, sufficient for compact Python executors but not multi-file embodied-agent code. We therefore use a coding-agent harness, drawing on systems that edit codebases, run tests, and iterate (SWE-agent [33], OpenHands [34], Claude Code [35]). Typed graph substrates such as LangGraph [36], AutoGen [37], and Trace [27] provide related abstractions for composing LLM calls, but primarily target human-authored workflows or text-domain optimization. In contrast, embodied AAS needs a graph substrate whose nodes are bound to stateful simulator execution, action interfaces, and episode-level logs (§3.2). Unlike repository-level coding-agent benchmarks, we use off-the-shelf coding agents as embodied-AAS optimizers: each patch changes an executable agent graph that must remain compatible with simulator APIs, module input–output interfaces, and diagnostic logging.

3 Method

Following prior AAS work [9, 10], we separate the task-performing *Executor* from the external *Optimizer* that edits it across development-time iterations. An *Executor* is the task-performing agent: a typed graph of perception, memory, planning, and action modules that runs inside a simulator and returns a task score. An *Optimizer* is a separate LLM-agent system that edits Executors across iterations. It never directly acts in the task environment. It proposes graph edits, triggers evaluation, and uses the resulting code diffs, scores, and logs to choose subsequent edits.

3.1 Problem Formulation

Let $\mathcal{C}$ be the substrate-supported space of executable embodied-agent graphs in §3.2. Each candidate $c \in \mathcal{C}$ receives fitness $f(c)$, measured by success rate over a multi-episode simulator evaluation, following embodied navigation, question answering, and manipulation benchmarks [1, 4, 15, 8]. Given a seed Executor c_0 and budget B, an AAS variant generates a trajectory $\{c_t\}_{t=0}^{B}$ and returns the best evaluated candidate $\arg\max_{t \le B} f(c_t)$. The transition rule producing c_{t+1} from the trajectory is the search policy, and is the component that differs across the three variants in §3.3.

Unlike text-domain AAS, we do not search from a small library of generic LLM-call operators. Embodied agents do not yet have a comparable set of runnable, task-agnostic primitives: navigation, question answering, and manipulation systems each contain task-specific perception, state, and action interfaces. We therefore use a *method-seeded* setting. Each c_0 is a published embodied-agent method, such as MapGPT [3], SmartWay [13], ExploreEQA [5], or VoxPoser [8], and search explores graph-level modifications around that method. This setting matches how embodied sys-

tems are currently engineered and lets us ask whether AAS can improve real executors rather than synthetic workflow sketches.

3.2 Substrates

Executor Substrate: AgentCanvas Embodied AAS requires candidates that are both editable by an Optimizer and runnable inside stateful simulators. We introduce AGENTCANVAS, a typed-graph runtime that represents each Executor as a node-and-wire JSON graph backed by Python node modules. Nodes expose typed input–output ports, and graph edits—adding edges, swapping nodes, or replacing subgraphs—are applied as structured patches whose type compatibility is checked before expensive rollouts.

AgentCanvas also instruments every rollout. Node firings record external inputs and outputs, while nodes may write internal breadcrumbs such as planner decisions, prompts, tool calls, or self-reports. These episode-level records connect candidate edits to downstream behavior and are exposed to all Optimizer variants through the shared harness, although §4.4 shows that access alone does not ensure systematic use.

For scale, simulator-dependent nodes are replicated across workers, while shared foundation-model nodes remain singleton services that batch requests. This enables multi-episode evaluation without reloading model weights per worker. Additional implementation details and runtime contracts are in Appendix A.

Optimizer Substrate: Coding-agent Harness Text-domain AAS often uses a single LLM proposer conditioned on a compact archive, because candidates are usually small text workflows or compact Python executors. Embodied executors instead expose multi-file workspaces containing graph JSON, node code, prompts, evaluation outputs, and episode logs. Useful proposals must often inspect files, diagnose failed rollouts, and modify both graph structure and node implementations. A key part of our port is therefore a repository-level coding-agent substrate that makes existing AAS optimizers runnable on embodied executors: ADAS and AFlow preserve their proposer and memory rules, while candidate edits for all variants are implemented through the same off-the-shelf coding-agent session with file, shell, and tool access [33, 34, 35].

The harness is shared across variants. An outer orchestrator runs the loop: the variant-specific proposer selects an edit direction, the implementer applies it to the active graph and code workspace, and the evaluator runs the simulator suite and records scores and logs. Only proposer logic and persistent memory differ; implementation, evaluation, validation, logging, and file-access contracts are fixed. Thus, performance differences should reflect search policy and memory structure rather than unequal tool, code-editing, or log access.

This substrate is necessary for embodied AAS, but not sufficient for good attribution. It records proposals, patches, tool traces, and episode logs; the Optimizer must still decide which evidence to inspect and how to convert it into future edits.

3.3 Variant Implementations

All variants use the same Executor substrate, coding-agent harness, evaluation suite, and iteration budget. They differ only in how the proposer selects the next edit and what information persists across iterations.

ADAS. We port ADAS [9] by preserving its Reflexion-style proposal structure, bootstrap-CI fitness, and flat append-only archive. Each of the original LLM proposal calls is implemented as an independent tool-augmented sub-agent, so the port retains the sampling diversity of the original method while giving each proposal access to the shared codebase and logs.

AFlow. We port AFlow [10] by preserving score-softmax parent selection, anti-replay memory, and success/failure experience injection. The main adaptation is the search space: the original AFlow

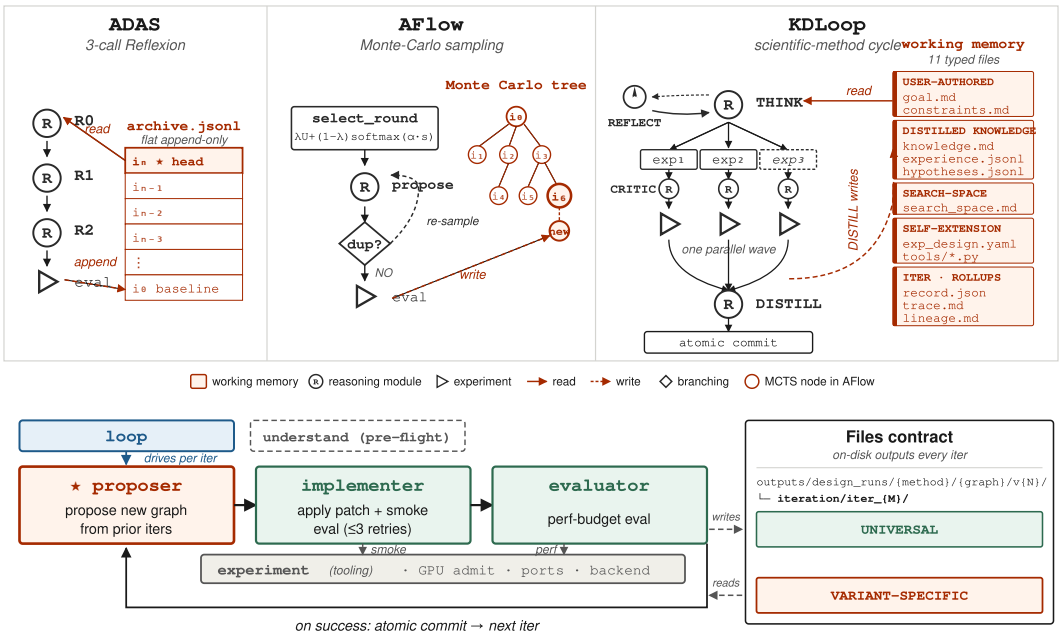

Figure 2: *Top*: AAS-method diagrams. Each variant differs in proposer logic and persistent memory. *Bottom*: the shared coding-agent harness. The outer loop invokes a proposer, implementer, and evaluator. Only the proposer is variant-specific, while implementation, evaluation, validation, and file access are shared.

composes over a curated text-workflow operator library, whereas embodied AAS must edit the seed graph directly. Thus, our port searches free-form structural changes over typed Executor graphs.

KDLoop. Knowledge Distill Loop (KDLoop) is designed for the embodied setting, where each iteration produces more evidence than a scalar score. It runs a four-phase cycle. THINK reads the distilled memory and proposes up to three experiments, each tagged with an intervention axis such as prompt content, topology, observation pipeline, state-memory, or model configuration. CRITIC checks the proposed edits against previous failed patches and constraints before execution. EXPERIMENT applies and evaluates the selected edit. DISTILL writes the outcome back into typed memory, including confirmed findings, refuted edits, open conjectures, and search-space coverage. A triggered REFLECT phase audits stalled progress, updates coverage, and may mark parts of the space as exhausted.

This memory structure is intended to address two embodied-AAS failure modes. It tracks what kinds of edits have already been attempted, reducing repeated local sampling, and it preserves mechanism-level evidence that would otherwise be lost in a flat archive. KDLoop is not guaranteed to dominate scalar SR: its coverage bias can leave a productive basin before deeper exploitation is complete. The experiments therefore evaluate it both as an optimizer and as a probe of what search structure is needed for embodied AAS.

4 Experiments

To our knowledge, this is the first systematic evaluation of AAS-style search that edits published perceptual embodied-agent architectures and validates the resulting candidates through simulator rollouts, rather than text-only workflow benchmarks. We use the 3×4 variant–executor matrix to ask whether graph-level search improves embodied agents, what edits survive, and where scalar SR gives insufficient credit. Across navigation, question answering, and manipulation, several cells improve over their seeded baselines, while others expose local-optimum and credit-assignment failures that are specific to embodied executors.

Setup. All experiments use a single pinned AgentCanvas build (§3.2), where each executor is represented as a forward node-and-wire graph with the backbone fixed to GPT-5-MINI (except VoxPoser).

Table 1: Yield across the 3×4 optimizer–executor matrix. Baseline/Best are raw SR mean±sd (%) over three post-selection passes; Δ is the mean gain in percentage points. Small variance-overlapping gains are directional, not certified. † marks a leak-bearing SmartWay run excluded from deployable claims; ‡ marks the VoxPoser logging-fault diagnosis (§4.4).

Executor	Optimizer	Baseline	Best	Δ	Surviving change (axis)
MapGPT	ADAS	46.9±3.1	49.1±3.2	+2.2	Step-Back distill + STOP-advisor wire + landmark gate (state-mem, topology)
	AFlow		54.5±3.1	+7.6	Stop/anti-revisit rules + elevation deadband (prompt, obs)
	KDLoop		54.0±2.3	+7.1	Heading-band action gate + `stop_after` 3→5 (obs, control)
ExploreEQA	ADAS	43.0±1.7	—	—	No iter beat baseline — lever not at the graph layer
	AFlow		47.7±2.1	+4.7	VLM single-letter prompt rewrite (prompt)
	KDLoop		46.0±1.0	+3.0	Bayesian prior + sentinel MCQ filter + "A/B/C/D-only" (state-mem, obs, prompt)
SmartWay	ADAS	29.7±2.1	33.7±2.5	+4.0	Plurality-vote planner (n=3) + Stop-Gate (model-cfg, topology)
	AFlow†		38.7±5.9	+9.0	Progress-tracker wired to evaluator (**leak**) + stall hint (topology, state)
	KDLoop		31.0±4.6	+1.3	STOP-lexicon history accumulator (state-mem)
VoxPoser	ADAS	9.0±0.0	12.7±0.0	+3.7	Composer rewrites + execution-loop tuning (model-cfg, action)
	AFlow		12.9±0.5	+3.9	GPT-4o composer swap + retry loop + workspace bounds (model-cfg, control, obs)
	KDLoop‡		—	—	Detected shared VoxPoser logging fault; diagnosed below graph layer; no mutation

The same search harness is used throughout: Claude Code provides the coding-agent session, and Claude Opus 4.7 with a 1M-token context window acts as orchestrator. We seed search from four published embodied-agent methods spanning three families: MapGPT and SmartWay for VLN, ExploreEQA for EQA, and VoxPoser for zero-shot VLA. Each cell starts from the corresponding method baseline, and search-time fitness is measured by task success rate (SR). We report raw SR mean±sd over three post-selection passes to separate search-time selection from rerun variance. For the VoxPoser executor, all optimizer runs include the same controlled substrate-level logging fault: dynamically dispatched LMP sub-calls are omitted from the voluntary self-report channel. This fault does not prevent scalar SR optimization, but it creates a diagnostic test of whether an optimizer inspects episode-level evidence rather than only scalar outcomes.

4.1 Does AAS improve embodied executors?

Table 1 shows that architecture-level search can improve embodied executors when the dominant failure mode lies inside the editable graph. Most selected candidates improve over the baseline mean after rerun, but variance-overlapping deltas are treated as directional rather than certified. Gains are clearest on MapGPT, where AFlow and KDLoop reach similar ∼54% SR from a 46.9±3.1 baseline through different stopping, revisitation, and action-gating edits. ExploreEQA improves more modestly through answer-formatting and sentinel-handling changes. The SmartWay–AFlow cell is excluded from deployable-gain counts because it uses evaluator-side information. We retain it as a diagnostic failure mode.

No optimizer dominates. ADAS tends to concentrate on topology and model configuration, AFlow often edits the observation pipeline, and KDLoop spreads edits across observation, prompt, control, and state-memory axes. The two cells without an SR claim are also informative: ADAS on ExploreEQA finds no improving graph-level lever, while KDLoop surfaces the shared VoxPoser logging fault and terminates with a below-graph diagnosis.

Figure 3 visualizes three representative search regimes. The complete 12-cell grid is deferred to Appendix B.

4.2 Evaluation Noise

The headline numbers in Table 1 are selected by single-pass fitness during search, but reported after three independent reruns. This distinction matters because a best iteration is the maximum over a

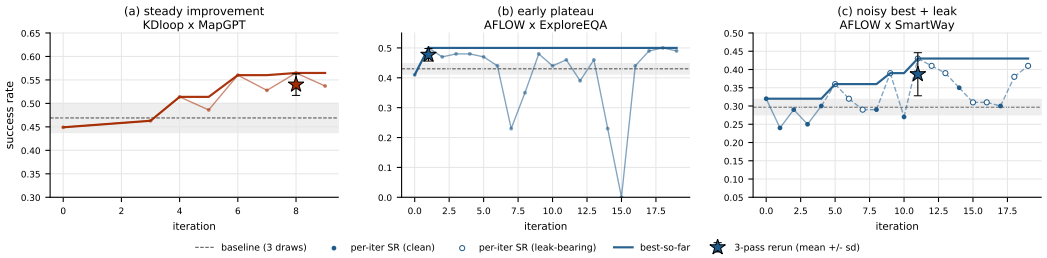

Figure 3: Representative search trajectories. Each panel shows single-pass SR, best-so-far SR, the shared baseline, and the rerun-confirmed best iteration. The three cells illustrate improvement, plateau, and noisy best-selection; full trajectories are in Appendix B.

Table 2: Per-cell attempted intervention-axis counts. Bold marks the repeated edit family analyzed in §4.3. Counts include KDLoop–VoxPoser attempts before the injected logging fault was surfaced.

Optimizer	Executor	PC	TO	CF	OP	SM	MC	AS	NP
ADAS	MapGPT	1	**10**	3	4	1	1	–	–
	ExploreEQA	–	–	1	**9**	–	–	–	**10**
	SmartWay	1	7	1	1	–	**9**	–	1
	VoxPoser	–	–	6	–	–	**10**	3	1
AFlow	MapGPT	4	2	6	5	2	–	1	–
	ExploreEQA	5	–	2	**12**	–	–	–	–
	SmartWay	3	2	2	7	1	4	–	–
	VoxPoser	1	–	4	1	–	**8**	5	–
KDLoop	MapGPT	2	2	2	3	–	–	–	–
	ExploreEQA	3	1	1	1	3	–	–	1
	SmartWay	2	2	–	2	3	–	1	1
	VoxPoser	–	–	4	2	–	1	–	1

PC prompt; TO topology; CF control-flow; OP observation; SM state-memory; MC model-config; AS action-space; NP no-change probe.

sequence of noisy estimates, and therefore can overstate the true value of a candidate architecture. The high-variance SmartWay cells illustrate the issue. AFlow's selected SmartWay design reruns at $43/32/41\%$, giving $38.7\pm5.9\%$ SR. KDLoop shows the same variance pattern at a lower mean, $32/35/26\%$, giving $31.0\pm4.6\%$ SR. In these cells, selection and verification are not interchangeable: the single-pass score identifies a candidate, but repeated evaluation is needed to interpret it.

Lower-variance cells behave differently. KDLoop on MapGPT (54.0 ± 2.3 vs. 46.9 ± 3.1) and the two ExploreEQA improvements (47.7 ± 2.1 for AFlow and 46.0 ± 1.0 for KDLoop, from 43.0 ± 1.7) remain separated from their baselines after rerun confirmation. The resulting lesson is methodological rather than tied to a particular executor: embodied AAS needs to distinguish search-time selection from post-hoc certification. KDLoop partially internalizes this requirement through a three-pass evaluation floor, whereas the ported methods rely on single-pass selection and require additional reruns to validate their selected iterations. This makes the reported gains credible, but also shows why naive best-iteration selection is insufficient for scaling embodied AAS under realistic evaluation budgets.

4.3 Local Optima in Search Dynamics

The local-optimum effect reflects the search dynamics rather than only the seeded executor. Once `ADAS-subagent` or `AFlow` finds a high-scoring modification pathway, later proposals often remain near the same mechanism: the search keeps moving, but mostly within one basin. Although both ports edit executable graphs, neither records which intervention axes have already been explored, whether an axis is saturated, or whether the editable space has been broadly covered. KDLoop addresses this by tagging proposals by intervention axis and using the typed history to redirect search after stalls.

Table 2 shows three symptoms of basin-local convergence. First, effort is concentrated on the most-used axis, averaging $60\%/43\%/38\%$ for ADAS/AFlow/KDLoop. Second, ADAS repeatedly rediscovers the same MapGPT score, $\mathrm{SR}{=}0.4769$, across six iterations, suggesting revisits to an

already-tested neighborhood. Third, without a space-level stopping predicate, ADAS and AFlow run to the iteration cap in all eight cells, using 84 and 82 committed iterations. These patterns suggest that embodied AAS needs memory not only over scalar scores, but also over the structure of attempted edits.

KDLoop makes the opposite trade-off. Its typed history reduces blind local resampling and improves coverage of the editable graph space, but this coverage bias can also under-exploit useful basins. On SmartWay, ADAS remains in a promising region long enough to find a useful graph-level refinement, whereas KDLoop redirects earlier. AFlow also concentrates on SmartWay, but its best result exploits a ground-truth evaluator affordance rather than improving the intended reasoning pathway, as discussed in §4.4. Thus, KDLoop is not uniformly better. Tt shifts the exploration–exploitation balance by reducing unproductive repetition while sometimes leaving productive basins under-explored.

4.4 Episode-level Credit Assignment Only Partially Emerges

Prior AAS systems usually optimize opaque LLM modules against scalar metrics such as accuracy or success rate. Embodied AAS provides richer evidence: each candidate executor produces observations, actions, tool calls, planner outputs, self-reports, and simulator traces that can explain why a score changed. Our harness exposes these logs, and all search agents know they are available. However, episode-wise credit assignment only partially emerges. In practice, the ported AAS variants still behave mainly as scalar-metric optimizers unless the search procedure explicitly directs attention to mechanism-level evidence.

The robustness probes expose this gap. In all VoxPoser runs, we inject a silent framework-level defect: the logs of dynamically dispatched language-model program (LMP) sub-calls inside the VoxPoser planner [8] are missing from the voluntary self-report channel. Although all agents can inspect episode logs, ADAS and AFlow do not surface the missing traces and remain focused on scalar evaluation outcomes. KDLoop is the only variant that actively inspects episode-wise logs and detects the absence, but it does so late rather than as a routine attribution step. In SmartWay, AFlow obtains its best score by wiring a ground-truth distance-to-goal signal from the Habitat evaluator into the executable graph. The scalar metric rewards this change, but episode-level inspection is needed to recognize that the mechanism depends on evaluator-side information rather than a deployable reasoning pathway.

These cases show that log access alone is insufficient. Coding-agent-based AAS does not reliably use episode-wise evidence for credit assignment, even when the evidence is available and discoverable. Embodied AAS therefore needs explicit attribution mechanisms that check whether a proposed graph mechanism actually executed, whether the information it uses is deployable, and whether a failure comes from the editable graph or the underlying substrate.

5 Limitations and Future Work

This study is an initial characterization of embodied AAS rather than a complete solution. **1) Executable scope:** AgentCanvas represents executors as typed node-and-wire workflows, covering many current embodied agents but not systems whose control flow, tools, or memory are constructed dynamically during deployment. Future work should extend AAS to richer executable representations and safer edit operators beyond workflow-shaped agents. **2) Optimization signal:** Costly, noisy rollouts can obscure the effect of a graph edit, so the best single-pass score may not reliably estimate architecture quality. Search dynamics can also induce local basins, where variants repeatedly modify one promising mechanism rather than exploring qualitatively different edits. KDLoop takes initial steps through rerun floors, typed intervention history, and space-exhaustion reflection, but future systems need more adaptive, substrate-portable strategies for selection, coverage tracking, and stopping. **3) Credit assignment:** Success rate alone is insufficient for embodied credit assignment. Although our harness exposes episode-level logs, detailed episode-wise analysis only partially emerges. Future embodied AAS systems should make attribution a first-class part of the loop, checking whether proposed mechanisms execute, whether information sources are deployable, and whether failures arise from the editable graph or the underlying substrate.

References

[1] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sünderhauf, I. Reid, S. Gould, and A. Van Den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3674–3683, 2018.

[2] G. Zhou, Y. Hong, and Q. Wu. Navgpt: Explicit reasoning in vision-and-language navigation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

[3] J. Chen, B. Lin, R. Xu, Z. Chai, X. Liang, and K.-Y. Wong. Mapgpt: Map-guided prompting with adaptive path planning for vision-and-language navigation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9796–9810, 2024.

[4] A. Das, S. Datta, G. Gkioxari, S. Lee, D. Parikh, and D. Batra. Embodied question answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–10, 2018.

[5] A. Z. Ren, J. Clark, A. Dixit, M. Itkina, A. Majumdar, and D. Sadigh. Explore until confident: Efficient exploration for embodied question answering. *arXiv preprint arXiv:2403.15941*, 2024.

[6] S. Saxena, B. Buchanan, C. Paxton, P. Liu, B. Chen, N. Vaskevicius, L. Palmieri, J. Francis, and O. Kroemer. Grapheqa: Using 3d semantic scene graphs for real-time embodied question answering. *arXiv preprint arXiv:2412.14480*, 2024.

[7] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[8] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

[9] S. Hu, C. Lu, and J. Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025.

[10] J. Zhang, J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025.

[11] Y. Shang, Y. Li, K. Zhao, L. Ma, J. Liu, F. Xu, and Y. Li. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations*, 2025.

[12] G. Zhang, L. Niu, J. Fang, K. Wang, L. Bai, and X. Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025.

[13] X. Shi, Z. Li, W. Lyu, J. Xia, F. Dayoub, Y. Qiao, and Q. Wu. Smartway: Enhanced waypoint prediction and backtracking for zero-shot vision-and-language navigation. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2025.

[14] J. Xu, A. Koesdwiady, S. Bei, Y. Han, B. Huang, D. Wang, Y. Chen, Z. Wang, P. Wang, P. Li, et al. Rethinking the value of multi-agent workflow: A strong single agent baseline. *arXiv preprint arXiv:2601.12307*, 2026.

[15] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749, 2020.

[16] A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, D. Ho, J. Ibarz, A. Irpan, E. Jang, R. Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, pages 287–318. PMLR, 2023.

[17] W. Huang, C. Wang, Y. Li, R. Zhang, and L. Fei-Fei. Rekep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation. *arXiv preprint arXiv:2409.01652*, 2024.

[18] M. Zhai, H. Liang, X. Fan, Z. Gao, C. Li, C. Sun, X. Bin, Y. Wu, and Y. Jia. Multi-step reasoning for embodied question answering via tool augmentation. *arXiv preprint arXiv:2510.20310*, 2025.

[19] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. Palm-e: An embodied multimodal language model, 2023. URL https://arxiv.org/abs/2303.03378.

[20] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, T. Jackson, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, K.-H. Lee, S. Levine, Y. Lu, U. Malla, D. Manjunath, I. Mordatch, O. Nachum, C. Parada, J. Peralta, E. Perez, K. Pertsch, J. Quiambao, K. Rao, M. Ryoo, G. Salazar, P. Sanketi, K. Sayed, J. Singh, S. Sontakke, A. Stone, C. Tan, H. Tran, V. Vanhoucke, S. Vega, Q. Vuong, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich. Rt-1: Robotics transformer for real-world control at scale, 2023. URL https://arxiv.org/abs/2212.06817.

[21] B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

[22] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[23] K. Black, N. Brown, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, L. Groom, K. Hausman, B. Ichter, S. Jakubczak, T. Jones, L. Ke, S. Levine, A. Li-Bell, M. Mothukuri, S. Nair, K. Pertsch, L. X. Shi, J. Tanner, Q. Vuong, A. Walling, H. Wang, and U. Zhilinsky. π_0: A vision-language-action flow model for general robot control, 2024. URL https://arxiv.org/abs/2410.24164.

[24] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[25] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36: 8634–8652, 2023.

[26] M. Zhuge, W. Wang, L. Kirsch, F. Faccio, D. Khizbullin, and J. Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

[27] C.-A. Cheng, A. Nie, and A. Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *Advances in Neural Information Processing Systems*, 37:71596–71642, 2024.

[28] Y. Wang, S. Liu, J. Fang, and Z. Meng. Evoagentx: An automated framework for evolving agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 643–655, 2025.

[29] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

[30] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

[31] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[32] S. Xu, J. Zhang, S. Di, Y. Luo, L. Yao, H. Liu, J. Zhu, F. Liu, and M.-L. Zhang. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.

[33] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[34] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, 2025.

[35] Anthropic. Claude code, 2025. https://www.anthropic.com/product/claude-code.

[36] LangChain. LangGraph, 2024. https://github.com/langchain-ai/langgraph.

[37] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Appendix Contents

- **Appendix A – AgentCanvas.** The typed-graph Executor substrate (§A).
- **Appendix B – Per-Cell Search Trajectories.** Full search trajectories across the 12-cell grid (§B).
- **Appendix C – Coding-Agent Harness.** The method-agnostic Optimizer-side substrate (§C).
- **Appendix D – Implementation Details.** Per-variant proposer and memory (§D).
- **Appendix E – Experiments Setup.** Executors, eval tiers, models, and the rerun protocol (§E).
- **Appendix F – Problem Formulation.** The formal search space, method-seeded neighborhood, admission, and objective (§F).

A AgentCanvas: A Typed-Graph Executor Substrate for Embodied AAS

This appendix documents AGENTCANVAS, the **Executor substrate** introduced in §3.2: the runtime that represents each embodied agent as an editable, runnable, instrumented program and so makes the space $\mathcal{C}$ of candidate architectures (§3.1) something an Optimizer can actually traverse. Its scope here is deliberately narrow. We describe only the designs that an AAS Optimizer relies on, not the human-facing visual editor, the node-authoring API, or the wider platform. The full implementation is in the released repository. The Optimizer-side substrate, the coding-agent harness that proposes and applies edits, is described in §3.2. This appendix is its Executor-side counterpart, and §A.6 ties the two together.

§A.1 states the two interface gaps in current embodied agents that motivate the substrate, together with the class of agents the substrate represents. §A.2–§A.3 describe the *inference* interface: the agent as an editable typed graph and the static checks that gate an edit before a rollout. §A.4 describes the *evaluate* interface, how an Optimizer autonomously scores a candidate at benchmark scale, with guaranteed termination. §A.5 describes the episode-level evidence available for credit assignment, and its honest limit. §A.6 summarizes the substrate as an AAS interface: the little the Optimizer must do, and the scaffolding it is spared.

Released code. The full implementation is available at: `https://github.com/jianzhou0420/AgentCanvas`.

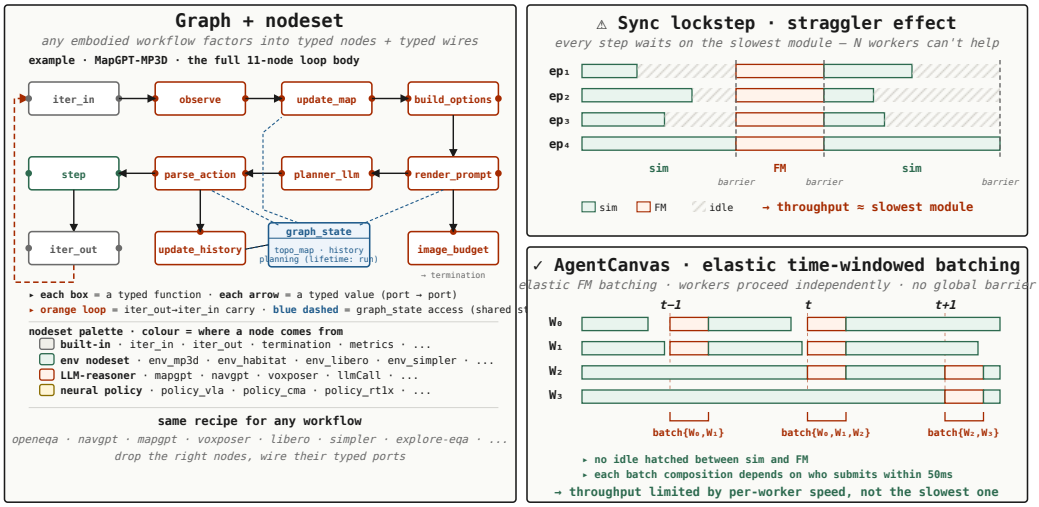

Figure 4: **AgentCanvas as an embodied-AAS substrate.** *Left:* an agent factors into typed nodes, typed-port wires, and shared state in a container reached by access grants (shown: the MapGPT-MP3D loop body; node colour = source nodeset; the loop is an `iter_out`→`iter_in` carry, not a back-edge; §A.2–§A.3). *Right:* the batch-eval optimisation (§A.4). Naive synchronous lock-step (top) stalls every worker on the slowest module, while AgentCanvas (bottom) decouples workers and batches whichever FM calls arrive within a short window (default 50 ms), so throughput tracks per-worker speed.

A.1 Two interface gaps in current embodied agents

Published embodied agents are typically shipped as bespoke, workflow-shaped forward passes implemented in custom multi-file code: a navigation or question-answering system hard-wires its own perception calls, prompt assembly, memory updates, model placements, and action decoding into Python that is specific to one method and one benchmark. For a *human* researcher this is legible enough. For an *Optimizer* it presents two missing interfaces.

(i) No unified inference interface. Because each agent's structure lives in idiosyncratic code, there is no common surface on which to express a structural edit. "Move the stop decision after the landmark check", "add a voting node over three planner calls", or "re-point the observation encoder" are uniform graph-level operations in principle, but in practice each requires reading and rewriting method-specific code, and a malformed edit is only discovered by running it. An Optimizer cannot apply structural edits uniformly, nor cheaply reject an ill-formed one.

(ii) No unified evaluate interface. Each agent also carries its own evaluation scaffolding, namely its own episode loop, its own simulator wiring, its own metric bookkeeping. There is no standard call by which an external system hands the agent a benchmark split and receives a success rate, and no standard way to run that evaluation at the throughput a search loop needs. An Optimizer cannot autonomously trigger scoring, and certainly cannot fan it out across workers without re-engineering each agent's harness.

AGENTCANVAS closes both gaps with a single substrate that is simultaneously *visual for a human* (a node-and-wire canvas, Figure 5) and *standard and batch-optimized for an Optimizer* (a typed-graph data model with one structured-edit surface and one batch-eval surface). The substrate represents an agent as a fixed forward graph over perception, memory, planning, and action modules: this covers the published embodied methods we seed from (§3.1) and is what makes method-seeded search tractable, but it deliberately stops short of agents whose control flow, tool set, or memory are *constructed dynamically during deployment*, the boundary that becomes Limitation 1 in §5. The remainder of this appendix describes the parts of that substrate that the Optimizer touches.

Figure 5: **The AgentCanvas UI, for a human.** The typed graph an Optimizer edits as JSON is, for a researcher, a node-and-wire canvas with an inspectable state panel (shown: the SmartWay-CE graph and its run state). The visual editor and the Optimizer's edit/evaluate interface are two views of one artifact.

A.2 The Executor as an editable typed graph

An Executor in AGENTCANVAS is a pure-data `GraphDefinition`: a JSON document of typed `NodeDefs`, `EdgeDefs`, state containers, and access grants that fully specifies one agent's topology

and configuration with no hidden runtime state (Figure 4, left).[1] Each `NodeDef` is a *reference*, not an instance: its `type` is a string key resolved at run time against the node registry (`NODE_HANDLERS` for built-ins, `${nodeset}__${node}` for nodeset-provided modules), with all per-instance knobs in a `config` dictionary. Edges connect *named* ports through `sourceHandle`/`targetHandle`, and a region of the graph can be wrapped into a reusable composite via `NodeDef.subgraph` with `portIn`/`portOut` boundary nodes. The runtime engine (`ReactiveExecutor`) expands composites with `flatten_graph()` before running, so nesting is an authoring convenience, not a runtime concept.

Because the whole agent is one referential artifact, the structural edits an AAS proposal needs are exactly the legal mutations of this JSON: add, remove, or retype a `NodeDef` (swap one module for another by changing its `type`); rewire an `EdgeDef` between named ports; tune any node's `config`; edit the state containers, access grants, `step_budget`, or `terminationCondition`; or wrap/unwrap a subgraph. This is the surface the Optimizer's implementer (§3.2) writes against, and the nodes that travel along it are themselves a uniform, swappable unit: every node (a tool, a model call, an environment step, a whole sub-agent) is a single class with declared `input_ports` and `output_ports` of typed `PortDef`s and one `execute(inputs, ctx)` method, so module substitution is type-shaped rather than code-shaped.

What this gives the Optimizer: one artifact whose every structural choice is an addressable JSON field, so a proposal is a structured patch rather than a multi-file code rewrite, and "one graph = one agent" makes a candidate portable, diffable, and loggable as a unit.

A.3 Typed ports and pre-rollout validation

Ports carry a fixed wire-type catalog: perception payloads (images, depth maps), control payloads (actions, poses), scalar and text payloads (text, booleans, metric bundles), structured observation and step-result records, an untyped escape hatch, and a `LIST[T]` modifier. Types are not decorative: they gate execution and, more importantly for search, they gate *admission*. Before any rollout, `validate_graph_connectivity()` runs at graph load and rejects (with an HTTP 400, never a silent never-fires) any candidate that wires incompatible types, leaves a required (non-optional) input unconnected, mis-uses a `LIST[T]` coercion, or violates the iteration-pivot wiring rules (the `Initialize`/`iterIn`/`iterOut` pairing that expresses a cyclic agent loop without graph back-edges). The legal edit and connection space is itself machine-readable: a node-schema endpoint exposes every node's ports and config schema, so an Optimizer can enumerate what may be wired to what rather than guessing.

What this gives the Optimizer: a cheap, deterministic filter that discards a structurally invalid proposal in milliseconds at load time, before it spends a multi-episode GPU rollout, turning "did my edit even type-check" from a post-hoc rollout failure into a pre-flight static check.

A.4 An autonomous, batch-optimized evaluate interface

The batch stack is shaped by an asymmetry between the two node kinds a run contains. A foundation-model node (the LLM/VLM backbone) is *stateless* (inputs in, outputs out) and natively batchable, so its efficient deployment is a single weights-loaded replica serving many callers at once. A simulator node is *stateful*: each parallel episode needs its own simulator, so K-way evaluation spawns K replicated simulator subprocesses whose model calls all route to the one shared backbone. The naive way to batch those calls (a fixed, lock-stepped batch that waits for all K workers to submit before firing one forward pass) reintroduces a straggler (short-board) effect: every worker stalls on the slowest simulator step, so one slow or long episode throttles the whole batch (Figure 4, right).

AgentCanvas avoids this with *decoupled workers and an opportunistic, time-windowed batch*. The K workers are never lock-stepped: each drives its own episode stream asynchronously

[1] The data model lives in `graph_def.py`, and it round-trips through `GraphDefinition.from_dict()` / `to_dict()`, so an Optimizer can read, mutate, and write a candidate as plain JSON.

(`EnvWorkerPool`), so a fast worker advances to its next step or episode without waiting for the others. Inside the shared backbone subprocess, a `BatchedInferenceServer` keeps one queue per `(function, config)` key. Each caller submits a single sample and awaits a future, and the queue flushes *whatever has arrived* after a short restart-on-submit debounce (`flush_timeout_ms`, default 50 ms) rather than after a fixed count. The batch size thus flexes to whoever is ready in the window (a slow worker is simply absent from the current batch and joins a later one instead of holding it open), and a single in-flight lock bounds peak GPU memory to one batch while the next batch keeps accumulating. The model's stateless contract is what makes this safe: any set of concurrent callers can be stacked into one pass, with recurrent state, where present, carried explicitly on the wire (`hidden_in`/`hidden_out`) rather than held in the server.

A run is launched with one `POST /api/eval/v2/start` (graph, split, episode count, `worker_count`). A VRAM-admission scheduler queues concurrent search sessions rather than letting them out-of-memory each other, and the single-worker path is bit-identical to the parallel one, so throughput scales without changing the measured score. The run returns aggregate success rate (and SPL / nDTW where the benchmark defines them) in `summary.json`, plus a frozen `graph.json` and self-contained per-episode records.

What this gives the Optimizer: a one-call, headless way to turn an edited graph into a benchmark score that scales across workers without per-worker weight reloads and without a straggler tax. This is what turns large-scale embodied search from possible into routine.

A.5 Episode-level evidence, and its honest limit

Every rollout is logged so that a score change can be *read off* rather than reproduced, and the log has two kinds of entries. The first is **automatic**: for each node firing the framework records the node's inputs and outputs, its timing and any error, and (for a model call) the model name and token cost, with no effort from the node author. The second is **optional**: a node may also record its own internals, the prompt it assembled, the raw model reply, a planner decision, by calling `_self_log`. Each episode is written to its own self-contained `log.jsonl`, so an Optimizer can open episode 47 and see exactly what fired, in order, without re-running or untangling it from other episodes.

The catch is that the second kind is opt-in. If a node does its real work *inside* itself (dispatching a tool call or a nested language-model program) and does not `_self_log` it, that inner step never appears: the log still shows the node's outer inputs and outputs, but not what happened between them. So the evidence reliably pins a score change to a *node*, but cannot always see *inside* one.

What this gives the Optimizer: per-node, per-episode evidence that localizes a score change to the firing that moved, with one stated blind spot, an unlogged internal dispatch, where that evidence runs out.

A.6 Summary: what the substrate asks of the Optimizer

Taken together, the preceding designs reduce embodied architecture search to a loop with a tiny surface. To search, the Optimizer needs only three things from the substrate: a typed JSON graph it can edit (§A.2–§A.3), a single call that scores any edited graph over a benchmark split (§A.4), and the per-episode logs that call returns (§A.5). Edit the graph, issue one evaluate call, read the result. That is the whole contract, and it is what makes AAS over real embodied agents possible at all.

What the Optimizer therefore does *not* have to do is the part that would otherwise dominate. It does not read or maintain each agent's bespoke, multi-file implementation. It does not write glue code to drive a simulator. And it does not build an evaluation harness per method. This matters because the Optimizer is itself a context-bound coding agent: when the agent under search is ordinary code with its own runner, every iteration burns scarce context on *how to execute and score* a candidate. Collapsing the agent to a data artifact behind a standard evaluator lets the Optimizer spend its context on *what to change* (the search itself) instead of on scaffolding.

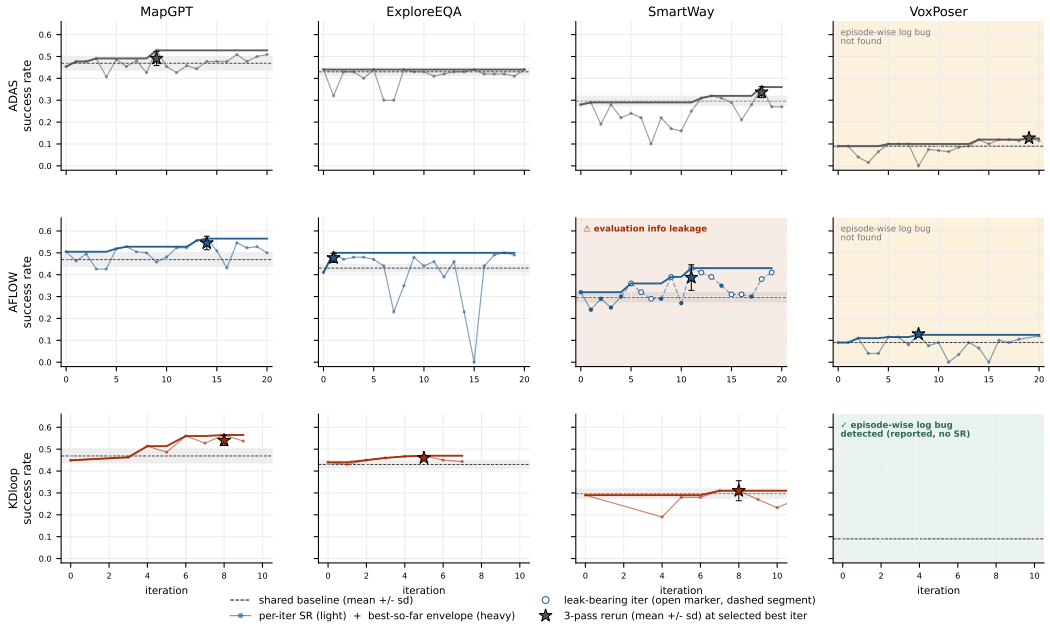

Figure 6: Full 3×4 per-cell search trajectories (rows: optimizers, columns: executors), same conventions as Fig. 3: per-iter single-pass SR (light line + markers), best-so-far envelope (heavy line), shared baseline mean $\pm$ sd (dashed line + band), and three-pass rerun mean $\pm$ sd at the selected best iter ($\star$). The success-rate axis is shared across all panels. The iteration axis is 0–20 for the ADAS / AFlow rows and 0–10 for KDLoop. In-cell shading marks the substrate-level findings: KDLoop detecting the episode-wise logging bug (green), ADAS / AFlow leaving it unfound (amber), and AFlow's SmartWay gain riding on evaluation-information leakage (red). AFlow $\times$ SmartWay's leak iters use dashed segments and open markers (§4.4).

And because that evaluator is the batched, straggler-avoiding stack of §A.4, each candidate is scored at low wall-clock, so the loop can afford to evaluate many of them. A minimal edit surface, an autonomous and fast evaluator, and readable per-episode evidence are jointly what turn embodied AAS from a per-method engineering effort into a search an Optimizer can simply run.

B Full Per-Cell Search Trajectories

ADAS and AFlow were designed as single-shot, non-agentic optimizers: one meta-prompt emits an entire agent in a single shot, scored by cheap exact-match on a text benchmark. That formulation does not run natively inside a coding-agent session, and its text-domain assumptions (no runtime, no episodes, near-free evaluation) do not transfer to embodied executors. We lift all three optimizers onto one shared coding-agent harness (§3.2) with embodied adaptations, and treat them not as competitors for a single ranking but as three *search paradigms* for embodied AAS (§3.3): 3-call Reflexion hill-climb (ADAS), softmax-mix parent sampling (AFlow), and knowledge-distillation search (KDLoop). The grid below is a *qualitative* comparison. It shows all three paradigms yield working search on embodied executors, each with a characteristic strength and failure mode, and we do not expect KDLoop to dominate search efficiency, effectiveness, and robustness at once. ADAS and AFlow concentrate within one edit basin, which buys precise within-mechanism tuning, e.g. AFlow on ExploreEQA spends eight of its nineteen search iters refining a single answer-aggregator (relevancy-threshold $\rightarrow$ rel-weighted sum $\rightarrow$ majority-vote $\rightarrow$ rel-gated mean-pool $\rightarrow$ Top-K confident-step), plateauing near 0.50. KDLoop instead searches several intervention axes in parallel and designs its own targeted sub-experiments (e.g. failure-mode episode subsets that accumulate reusable knowledge), giving more efficient coverage. Its knowledge-driven search also avoids the evaluation-information leakage AFlow falls into on SmartWay and surfaces the framework logging bug on VoxPoser (§4.4).

Figure 6 is the full 3×4 grid behind the three representative cells of Fig. 3. Plot conventions, axes, and the in-cell shading are defined in the caption.

Special cells. AFlow × SmartWay: 11 leak-bearing iters wire the evaluator's ground-truth distance-to-goal into the planner (§4.4), and the headline-selected iter_11 is leak-bearing. ADAS × ExploreEQA: no iter beats iter_0 in 20 iters, so no rerun star. KDLoop × VoxPoser: exits with a substrate bug report, no SR.

Across the grid. *Steady growth* on MapGPT for all three: AFlow peaks at iter_14 (0.545 ± 0.031 rerun), KDLoop at iter_8 (0.540 ± 0.023), ADAS at iter_9 (0.491 ± 0.032), all above the 0.469 baseline. *SmartWay* separates selection from verification: ADAS $0.360\rightarrow0.337\pm0.025$ and AFlow $0.430\rightarrow0.387\pm0.059$ both shrink on rerun, and AFlow's higher figure is leak-bearing (§4.4), whereas KDLoop holds a leak-free 0.310 ± 0.046. *VoxPoser*: ADAS and AFlow post marginal substrate-adjacent gains ($+0.037$ / $+0.039$), while KDLoop diagnoses the episode-wise logging bug underlying the cell (reported, no SR).

C Coding-Agent Harness as AAS Substrate

This appendix documents the **method-agnostic coding-agent harness** that hosts the AAS family. It is the substrate an AAS algorithm runs on, not any algorithm itself. It is method-agnostic by construction: swapping the algorithm replaces a single skill (`proposer`) and reuses the rest. Concrete variants (ADAS, AFlow, ...) and their variant-specific choices are out of scope. Our `ADAS` port is in Appendix D.

C.1 Pipeline overview

One iteration factors into an orchestrator (`loop`) and four workers: `understand` (a read-only context loader), `proposer` (the one AAS-specific skill), `implementer`, and `evaluator`. The latter two delegate evaluation to a shared runner (`/experiment:run`). Skills are Markdown prompts run by a single coding-agent conversation per iteration. The decomposition exploits a capability shift: where original ADAS [9] needed bespoke implement-and-debug machinery (a typed edit language, a deterministic replayer, repair loops) because models could not edit a multi-file codebase from a spec, recent coding agents [35, 33] can. We collapse that whole branch into `implementer` (§C.3). The proposer emits a prose change spec and a sub-agent edits the code natively.

Files contract. All variants share an on-disk contract (`_common/files-contract.md`). Runs live under a method-scoped root `outputs/design_runs/{method}/{graph}/v{N}/`, one dir per iteration at `iteration/iter_n/`. The one mandatory per-iteration artifact is an `active_workspace/` overlay (§C.5). Other artifacts are typed by a fixed taxonomy and declared per variant in a manifest, so the substrate fixes placement/mutability/lifecycle by type while each variant names only its own files.

C.2 The `loop` skill: per-iter orchestration

`loop` owns one per-iteration conversation and drives `proposer` → `implementer` → `evaluator`, with each worker writing into a transient staging directory. On evaluator success it runs the atomic-commit transaction (§C.5). On any worker failure it discards staging, increments `consecutive_skips`, and treats the iteration as never having happened. It terminates on an iteration cap, a user `STOP` sentinel, or too many consecutive skips. Where the algorithm needs independent samples or fresh tool contexts, a worker issues *sub-agent spawns*, self-contained sub-conversations with full tool access that return a JSON block. N spawns are N independent samples, which a single thinking-pass (one shared autoregressive trace) cannot provide. The substrate supplies the spawn primitive, and how many to issue and how to combine them is the variant's choice (Appendix D).

C.3 Implementer: native editing and the smoke gate

The proposer hands over a change spec, a prose `intent` plus a `targets` file list, not a typed op list. A deterministic helper (`_common/lib/overlay.py`) seeds each target into the overlay and enforces

the edit whitelist (a hard wall around framework and vendored code, and a soft warn for edits outside the iteration's graph and nodesets), but never edits. A tool-augmented sub-agent then edits the seeded files natively. The implementer validates the result (JSON still parses, Python still compiles) and pins every model-call node to one enforced inference profile (`pin_llm_profile.py`) so the search cannot vary the underlying model. It then runs a small smoke evaluation (default 5 episodes, profile `smoke_{graph}`) whose gate is **runtime correctness only**: pass iff the run exits cleanly, every episode completes, takes ≥ 1 step, and returns a valid numeric metric. The metric *values* are not consulted, so a clean zero-scoring run is data, not failure. On any failure the overlay resets to the parent and a *fresh* sub-agent retries (default `retry_max`=3), seeing only the `intent` plus failure traces as read-only context, with no inherited message history. On exhaustion the implementer reverts and signals a skip, and frozen `workspace/` is never touched.

C.4 Evaluator and the fitness signal

The evaluator runs once per successful implementer on a separate profile (`perf_{graph}`, typically 100 episodes) and writes a **neutral** metrics record (run id, episode count, per-episode accuracy list, primary-metric mean, secondary metrics). It is **method-free**: how those numbers become the optimizer's scalar fitness (a mean, a t-interval, or our ADAS port's bootstrap confidence interval) is a variant choice computed in the variant's `loop` during commit (Appendix D), which is what keeps the evaluator reusable across variants. The cheap-smoke / expensive-perf split is cost-driven: a 100-episode run costs 1–2 hours, so retries are confined to the smoke tier, and a low-but-valid perf result is recorded, not retried.

C.5 Run-directory contract and atomic commit

All state is filesystem-based. Frozen `workspace/` is never written by any skill, and an iteration's edits land only in its `active_workspace/` (bootstrapped from the parent, then patched), which the backend overlays over the frozen baseline at evaluation time (overlay wins on conflict). This overlay is the single point coupling the file contract to runtime behaviour. It lets each iteration, and concurrent runs over one baseline, evaluate independent edits without a per-iteration checkout. Workers write to staging (`v{N}/.staging/iter_n/`). On success the orchestrator moves it into `iteration/` and *then* appends one entry to the variant's working memory (for ADAS, one archive line enriched with the fitness statistic). The move-then-append order makes a failed iteration a clean no-op: a half-written iteration never enters the working memory.

D Variant Implementation Details

The three variants share the substrate of Appendix C and differ only in the proposer skill and the memory that persists across iterations (§3.3). Where §3.3 states *what* each port preserves from upstream, this appendix gives *how* our port realizes it. The released code for all three lives under `.claude/commands/architect/`, and each algorithm's caption names its specific files, so the prose below stays at the level of mechanism. All three algorithms share one skeleton: a one-line setup then a flat per-iteration loop. In the loop body, IMPLEMENT (native editing + smoke gate) and EVALUATE (neutral metrics) are the shared substrate, and everything else is variant-specific.

D.1 ADAS

We realize each of upstream's three Reflexion calls [9] as an *independent* tool-augmented sub-agent spawn, recovering per-call sampling diversity while giving every sample full read/grep/shell access. The three spawns share one message history rendered as text, and the deliverable is a prose change spec (an *intent* plus a list of target files). We pre-seed the archive with the baseline plus seven reference patterns as text-only, null-fitness entries, a design palette to adapt, not benchmarked alternatives. Two upstream mechanisms are deliberately dropped: the stateless-API sampling knobs (no analogue once a call is a conversation), and the smoke gate's "mean $<0.01 \Rightarrow$ debug" trigger (a clean

zero-scoring embodied run is data, not a bug, so the gate checks runtime correctness only). Fitness is a bootstrap CI computed in the variant loop at commit, keeping the shared evaluator method-free. The Reflexion-only reflection field is stripped before the archive append.

Algorithm 1 ADAS port. Files: `adas-subagent/{loop,proposer}.md`, `lib/helpers.py`; substrate `_common/{implementer,evaluator}.md`.

PRESEED: baseline *iter*$_0$ + 7 reference patterns (*fitness*=null) ▷ *loop §3*
for each iteration n **until** cap / STOP / skips **do** ▷ *loop §4*
  analysis ← ANALYZEHEAD(*archive*[−1]) ▷ *proposer §2*
  base ← BUILDPROMPT(*archive*, *analysis*) ▷ *§3–4: full archive injected*
  s_0 ← SPAWN(*base*) ▷ *§6 call #1: propose*
  s_1 ← SPAWN(*base*, s_0, REFLEXION$_1$) ▷ *call #2 (shared history)*
  s_2 ← SPAWN(*base*, s_0, s_1, REFLEXION$_2$) ▷ *call #3 → patch*
  if any spawn fails **then** rollback; **continue**
  if ¬IMPLEMENT(s_2.*patch*) **then** rollback; **continue** ▷ *smoke, ≤ 3 retries*
  acc ← EVALUATE(`perf`) ▷ *neutral metrics*
  fit ← BOOTSTRAPCI(*acc*) ▷ *in loop, not evaluator*
  COMMIT: append *archive* (strip *reflection*)

D.2 AFlow

Each iteration first selects a parent from the archive by score-softmax mixing, so any prior iteration can seed the next edit. We follow upstream's *code* defaults (α=0.2, λ=0.3, K=4), not the paper's Eq. 3 (α=0.4, λ=0.2) [10]. The parent's prior accepted and rejected modifications are injected as "avoid-repeating" experience. A duplicate (under whitespace-normalized matching) resamples the parent, and the resample loop (unbounded upstream) is capped at five. The search space is free-form structural edits over the typed Executor graph (no operator library), and the selected parent's overlay is checked out into staging before IMPLEMENT. Two further deviations, both on the evaluation side. (i) Cost rules out the five-pass validation average, so each iteration is scored on a *single* performance pass. Because that score feeds both the softmax ranking and the convergence test, the unmodeled run-to-run variance propagates into the search itself, not just the final report, and we calibrate it with post-loop verification reruns on the top iterations. (ii) Under single-pass scoring every per-round variance is zero, so upstream's top-3 convergence predicate collapses to exact equality regardless of z, and we keep it advisory. Memory is the ADAS archive plus a parent id, the verbatim modification string, and a bare numeric score (the bootstrap median) for the softmax.

Algorithm 2 AFlow port. Files: `aflow/{loop,proposer}.md`, `lib/aflow_helpers.py`; substrate as ADAS.

PRESEED: baseline *iter*$_0$ only ▷ *no reference palette*
for each iteration n **until** cap / STOP / skips / converged **do**
  experience ← EXPERIENCEMAP(*archive*) ▷ *parent ↦ succ/fail mods*
  for *attempt* = 1 . . . 5 **do** ▷ *anti-replay*
    top ← TOPROUNDS(*archive*, K=4) ▷ *iter*$_0$ *forced in*
    p ← SELECTROUND(*top*, α=.2, λ=.3) ▷ *softmax-mix*
    r ← SPAWN(OPTIMIZEPROMPT(p, *experience*[p])) ▷ *single call*
    if ¬PARSED(r) **then continue** ▷ *regex fallback*
    if CHECKMOD(r, *experience*[p]) **then continue** ▷ *duplicate*
    break ▷ *novel modification*
  if none accepted **then** rollback; **continue**
  write `proposal.md`(p, *r.mod*); CHECKOUT(p) ▷ *loop §4b*
  if ¬IMPLEMENT(r) **then** rollback; **continue**
  acc ← EVALUATE(`perf`) ▷ *every iteration*
  score ← median(*acc*) ▷ *feeds* SELECTROUND
  COMMIT(p, *mod*, *score*); CHECKCONVERGENCE ▷ *advisory*
VERIFYRERUNS(top-1, top-2, baseline) ▷ *loop §8*

D.3 KDLoop

KDLoop is the variant we design for the embodied setting, where each iteration yields more evidence than a scalar score. It replaces the single propose step with a four-phase cycle plus a triggered meta-phase. THINK (one tool-augmented spawn) reads the typed memory and emits up to three experiment specs, each tagged with an intervention axis. An empty THINK result never stops the run but escalates to REFLECT. CRITIC spawns once per patched spec to predict whether it would re-introduce a previously refuted pathology, with one rebuttal round back through THINK. EXPERIMENT (inline, no separate implementer skill) applies each surviving spec in its own overlay, then submits all *(spec, pass)* pairs as *one* parallel wave whose per-submission worker counts are set by a makespan minimizer under the performance-tier worker cap. DISTILL (one spawn over all specs) writes lessons back to memory. REFLECT is a meta-phase that fires only on a three-iteration heartbeat, on the last three iterations sharing one axis, or on a THINK escalation, and it audits search-space coverage. Memory is eleven typed files rather than a flat archive. The load-bearing ones are pure facts, an open-hypothesis queue, an append-only log of closed-case lessons (confirmed *and* refuted), a coverage map, an eval-profile registry, and self-authored utilities. These files are mutated by the explicit eager writes shown in Alg. 3: THINK appends new conjectures to the hypothesis queue, and for every hypothesis that its experiment resolves, DISTILL appends a lesson to the experience log and line-deletes the entry from the queue, so a refuted patch is committed as a lesson rather than discarded. Termination is goal-driven or a REFLECT "space-exhausted" verdict, not a fixed iteration cap.

Algorithm 3 KDLoop port. Files: `myloop/loop.md` and phase skills {`proposer`,`critic`,`distill`,`reflect`}`.md`, `lib/multi_spec_eval.py`.

Typed memory M: `knowledge.md`, `hypotheses.jsonl`, `experience.jsonl`, `search_space.md`, ...
($+=$ append, $-=$ line-delete)

for each iteration n **until** goal met / space exhausted **do** ▷ *§3a: poll `goal.md`*
  if REFLECTTRIGGER **then** ▷ *heartbeat-3 / axis-3*
    $v \leftarrow$ REFLECT(M); `search_space.md` $+=$ coverage ▷ *eager*
    if v=exhausted **then stop** (SATURATED)
  specs $\leftarrow$ THINK(M) ▷ *$K \leq 3$ axes*
    `hypotheses.jsonl` $+=$ new conjectures ▷ *THINK eager*
    `knowledge.md`, `experiment_design.yaml`, `tools/` $+= \ldots$
  if *specs*$=\emptyset$ **then** REFLECT(M); retry / skip ▷ *never stops*
  for *spec* $\in$ *specs* **with** patch **do** ▷ *§3b.5 CRITIC*
    $c \leftarrow$ CRITIC(*spec*)
    if $c \in$ {REVISE, BLOCK} **then** *spec* $\leftarrow$ THINK(*spec*, c); recheck ▷ *1 rebuttal*
  ready $\leftarrow$ surviving patched specs $\cup$ no-patch specs
  for *spec* $\in$ *ready* **with** patch **do** ▷ *§3c: apply, own overlay*
    seed overlay; IMPLEMENT(*spec*) ▷ *no stacking*
  meta $\leftarrow$ MULTISPECEVAL(*ready*); ALLOCATEWORKERS ▷ *ONE wave; makespan-min*
  verdicts $\leftarrow$ DISTILL(*ready*, *meta*) ▷ *one spawn*
  for each hypothesis h resolved this iter **do** ▷ *distill.md eager*
    `experience.jsonl` $+=$ lesson(h) ▷ *confirmed / refuted / inconcl.*
    `hypotheses.jsonl` $-= h$ ▷ *resolved $\rightarrow$ off queue*
  `hypotheses.jsonl` $+=$ new; `knowledge.md` $+=$ facts ▷ *DISTILL eager*
  COMMIT(`record.json`) ▷ *§5*

E Experiments Setup Detail

This appendix expands the setup of §4. Every cell of the 3×4 matrix runs on the same harness (Appendix C) and the same per-executor evaluation profile, and variants differ only in the proposer and memory (Appendix D).

E.1 Executors and benchmarks

Each cell seeds search from a published embodied-agent method and scores candidates by task success rate (SR) on that method's benchmark (Table 3). The performance tier is the paper-comparable

eval set and supplies every reported SR. The navigation cells additionally log SPL and (n)DTW as secondary metrics, EQA logs step count, and manipulation logs step count and cumulative reward.

Executor	Domain	Simulator	Split	Perf. episodes
MapGPT	VLN (discrete)	Matterport3D	`MapGPT72`	216 (val_unseen subset)
SmartWay	VLN (continuous)	Habitat (VLN-CE)	`rand100`	100 (val-unseen subset)
ExploreEQA	Embodied QA	HM3D	`val`	100 (of 500 HM-EQA val)
VoxPoser	VLA manipulation	LIBERO	VAS	200 (4 suites $\times 10 \times 5$)

Table 3: The four executors and their performance-tier evaluation sets. VAS is the four LIBERO suites (`spatial / object / goal / 10`), ten tasks each, five episodes per task. SR is `success` on each benchmark's own scorer.

E.2 Evaluation tiers

Each executor profile defines a cheap deterministic *smoke* tier (3–8 episodes) and a paper-comparable *performance* tier (Table 3). Every candidate is smoke-gated before it is ever measured. The smoke tier checks runtime correctness only (the run exits cleanly, each episode takes ≥ 1 step and returns a valid metric) and a candidate that fails it is repaired or skipped, never scored (Appendix C.3). The two ADAS-family ports then measure each surviving candidate on the *full* performance tier: AFlow on every iteration, because its score-softmax parent sampling needs a comparable per-iteration score, and ADAS once per successful candidate, for its archive and report. KDLoop instead *designs* its own experiment each iteration. Its Think phase composes a targeted subset, and a ≥ 30-episode failure-mode set is the default measurement tier. It escalates to the full performance tier only once a custom run shows non-degenerate lift (≥ 0.05 SR over the prior best), to amortize the 1–2 h cost. Episode budgets and worker counts are pinned per executor (e.g. MapGPT 15 env steps at 120 s each, SmartWay 20 at 180 s, and VoxPoser 200 waypoints).

E.3 Models and iteration budget

The searched agents use a single pinned backbone, GPT-5-MINI at temperature 1, deterministically enforced on every model-call node (§C.3). The VoxPoser composer is the exception, run on GPT-4O. The search harness is fixed throughout: Claude Code provides the coding-agent session and Claude Opus 4.7 with a 1M-token context is the orchestrator. ADAS and AFlow run 20 search iterations per cell and KDLoop runs 10, not counting the shared `iter_0` baseline each evaluates first. Because each KDLoop iteration designs and runs several experiments, its ten iterations are roughly equivalent in compute to twenty ADAS/AFlow iterations, so the per-cell budget is comparable across optimizers. There is a single run per (optimizer, executor) cell, with no multi-seed averaging, and variance is reported within a run (next).

E.4 Baseline and rerun protocol

To separate search-time selection from rerun variance we report two reruns, both on the performance tier. (i) *Shared baseline*: the three optimizers each evaluate the same baseline graph as their `iter_0`, giving three independent SR draws per cell, and their mean and sample standard deviation are the Baseline column of Table 1. (ii) *Best-iter rerun*: each cell's selected best iteration is re-evaluated three times under the identical performance config. We report the mean$\pm$sd and the gain Δ against the shared-baseline mean.

E.5 VoxPoser logging fault

All VoxPoser runs share one controlled substrate-level fault: dynamically dispatched LMP subcalls are omitted from the agent's voluntary self-report channel. The fault does not block scalar SR optimization, so it acts as a diagnostic of whether an optimizer inspects episode-level evidence rather than only the scalar outcome. In our runs this cleanly separates the three optimizers: all share the same log access, yet ADAS and AFlow never surface the missing traces and keep optimizing

the scalar SR, while KDLoop is the only variant to inspect the per-episode logs and report the fault (§4.4).

F Problem Formulation for Embodied AAS

Automated agent design has produced almost as many problem formulations as methods, because a formulation tends to fold its method's own commitments into the statement of the problem, so what reads as the problem is really one method's move-set. And because the field is text-agent dominant, those formulations are shaped for prompt-and-tool pipelines, not for the closed perception–action loops of embodied agents, so they do not transfer. We therefore give a formulation that is both embodied and method-agnostic. We keep the template ADAS [9] adopts by analogy to neural architecture search (a *search space*, a *search algorithm*, and an *evaluation function*) but draw the line the template leaves implicit, between the problem and the method. The *problem* is the search space together with the objective over it: fixed, method-agnostic, and shared by every variant (§F.1, §F.3). The *method* is the search algorithm, the transition rule that produces the next candidate, the only component that varies (§3.1). We formalize it for the three variants we study: ADAS, AFlow, and our KDLoop (§F.2).

F.1 AgentCanvas as substrate: the search space

The substrate fixes what a candidate is and which candidates are well-formed. It is what *defines* the search space rather than shrinking a pre-existing one (§A.1). We give the candidate object and the operators that edit it, then the space in two views (its *open form*, the set of all expressible candidates independent of any starting point, and its *method-seeded* region, what the search reaches from a published baseline) and close with what the space can express.

Candidate. A candidate is a typed, config-bearing graph,

$$c = (V, \Sigma, E, \gamma), \qquad E = E_{\text{flow}} \sqcup E_{\text{mem}},$$

with four constituents: the *nodes* V, drawn from an open, optimizer-extensible node library $\mathcal{N}$; the *memory cells* Σ, the candidate's declared shared state; the *edges* E, of two kinds, namely dataflow edges E_{flow} and memory edges E_{mem}; and the *configuration* γ, a per-node parameter map. We expand each in turn.

Nodes. A single graph holds finitely many nodes ($V \in \mathcal{N}^*$), but $\mathcal{N}$ is not a fixed palette: the optimizer may author a new node type and modify or delete an existing one, realized as patches that add or rewrite a `BaseCanvasNode` subclass. Formally $\mathcal{N} = \mathcal{N}_0 \cup \mathcal{N}_\Delta(c)$, a seed library $\mathcal{N}_0$ plus the node types defined within c's own overlay, so the library is endogenous to c rather than an exogenous constant.

Memory cells. Each cell $\sigma \in \Sigma$ holds a typed value with a *reducer* (accumulator, last-write, or counter) that combines concurrent writes and a *lifetime* that fixes when a reset signal clears it.

Edges. Dataflow edges E_{flow} wire nodes to nodes and must be type-compatible. Memory edges $E_{\text{mem}} \subseteq V \times \Sigma$ connect a node to a cell: $(v, \sigma) \in E_{\text{mem}}$ lets node v read and write cell σ. A memory edge is of a different kind from a dataflow edge (it carries no payload and does not trigger firing), and the two kinds are disjoint by endpoint type, so a dataflow payload may not live in a cell.

Configuration. The per-node map γ, with $\gamma(v) \in \text{Cfg}(t(v))$, gives each node its prompt text, sampling parameters, and `persist` flags. (Σ and E_{mem} are the *declared, shared* memory. A node's private runtime scratch and an `iterIn` `persist` slot (carried dataflow, held in γ) are not part of it.)

Everything is a node. Two things that a formulation often leaves outside the candidate are here ordinary nodes in V. The first is control flow: the loop and its stopping rule are not an external harness but the nodes INITIALIZE, ITERIN, ITEROUT, and TERMINATION. The second is the environment: the simulator the agent acts in is itself a node, joined to the agent by ordinary dataflow edges. The agent reads observations from it and sends actions back. A candidate is therefore one self-contained,

closed-loop graph holding the agent, its control flow, and its environment together, with no hidden code running outside it. (The environment node supplies the task, observations in and actions out, and is held fixed across a run. What the search varies is the agent and how it is wired to that node.) A candidate's *coarse procedure*, how it plans and how it acts each step, is thus just its topology, interior to c and part of what is searched, not a knob of the algorithm. This is the data model of Appendix A (repro anchor `graph_def.py`).

Operators. A candidate of the shape just defined comes with its move-set for free. The constituents that can be edited are fixed (the node-type library $\mathcal{N}_\Delta$, the node instances V, the memory cells Σ, the two edge kinds E_{flow} and E_{mem}, and the configuration γ), so the high-level operators are simply one add / modify / remove family per constituent, a finite alphabet $\mathcal{O}$:

constituent	add	modify	remove
node type $\mathcal{N}_\Delta$	DEFINE-TYPE	MODIFY-TYPE	DELETE-TYPE
node instance V	INSERT-NODE	RETYPE-NODE	REMOVE-NODE
memory cell Σ	ADD-CELL	EDIT-CELL	REMOVE-CELL
dataflow edge E_{flow}	CONNECT	REWIRE	DISCONNECT
memory edge E_{mem}	GRANT	—	REVOKE
config γ	SET-CONFIG (fields: prompt / param / persist)		

DEFINE-TYPE extends the *vocabulary* $\mathcal{N}_\Delta$, while INSERT-NODE instantiates an existing type into the *graph* V. These are distinct operators, since authoring a new node and using it are separate moves. EDIT-CELL retypes a cell or changes its reducer or lifetime, and a memory edge has no modify form because it confers read and write together.

This alphabet is *complete* at the structural level. Any finer, implementation-level edit (rewording a line of a prompt, flipping one `persist` flag, swapping a tool inside a node) is a special case of one of these operators (here, SET-CONFIG). It changes a constituent's contents without introducing a new *kind* of move. The alphabet closes for a structural reason: it is generated by the constituents of c themselves, and an edit touching nothing in $\{\mathcal{N}_\Delta, V, \Sigma, E, \gamma\}$ would not be an edit to a candidate at all. The operators are afforded by the substrate's data model, and the search policy (§F.2) is what realizes each as a code diff and sequences them.

The open-form space. Let $\mathcal{W}(V, \Sigma)$ denote the well-typed edge sets over $V \cup \Sigma$ (type-compatible dataflow edges, plus memory edges to existing cells) and $\mathrm{Valid}(\cdot)$ the static well-formedness check of §A.3 (well-typed edges of both kinds, three-pivot coherence, reachable graph in/out). The open-form search space is the set of all valid candidates:

$$\mathcal{C} = \Big\{ c = (V, \Sigma, E, \gamma) \;\Big|\; V \in \mathcal{N}^*,\ E \in \mathcal{W}(V, \Sigma),\ \gamma \in \prod_{v \in V} \mathrm{Cfg}\big(t(v)\big),\ \mathrm{Valid}(c) \Big\}. \tag{1}$$

It is fixed by membership alone (a candidate is in $\mathcal{C}$ if it is well-formed), with no reference to a starting point or to how one candidate is reached from another. $\mathcal{C}$ is countably infinite (node count, prompt strings, and node-type definitions are all unbounded) yet strongly regularized: not by a closed library ($\mathcal{N}$ is open) but by the type discipline its nodes and edges must respect, which is what keeps Valid a cheap static check rather than a rollout.

The method-seeded space. The open-form $\mathcal{C}$ is far too large to enumerate, and AAS does not try to. It *seeds* from a published method, ported into a graph as a baseline $c_0 \in \mathcal{C}$, and explores the candidates reachable by applying the operators above to that seed:

$$\mathcal{C}(c_0) = \Big\{ c_0 \oplus \Delta \;\Big|\; \Delta \in \mathcal{O}^*,\ \mathrm{Valid}(c_0 \oplus \Delta) \Big\}, \tag{2}$$

where $\oplus$ applies the operator patch Δ to the seed. Because every operator edits only the candidate's own constituents, the framework runtime and the vendored environment stay frozen by construction. The search reaches agents, never the substrate they run on. Every member of $\mathcal{C}(c_0)$ lies in the open-form $\mathcal{C}$ (it meets the same Valid), so the seed and the operator budget carve out a tractable region of $\mathcal{C}$ rather than the whole of it.

An expressive, structurally validatable space. Each node is a Python class, hence Turing-complete on its own, and the graph layer adds loops, branching, and shared state over nodes. Together that is enough that current embodied agents, workflow-shaped ones in particular, can almost all be represented cleanly as candidates. What keeps the space searchable is that validity is a structural, graph-level check: a candidate's wiring (typed edges, three-pivot coherence, reachable in/out) is screened by the cheap static Valid of (1), no rollout, and only authoring a new node type drops to opaque Python. The space is also not a product of independent coordinates: an edit to one constituent constrains the others through Valid, so $\mathcal{C} \neq \prod_k \Pi_k(\mathcal{C})$. The one form it cannot represent, control flow built dynamically at deployment, is Limitation 1 (§5).

F.2 The search policy: ADAS, AFlow, and KDLoop

The search policy is the method, the only component that differs across variants, since the space (§F.1) and the objective (§F.3) are shared. Each variant carries its own run memory H_t, a summary of the evaluated trajectory whose structure is itself part of the method. The three fall into two shapes: ADAS and AFlow are one family, a single-proposal archive hill-climb that differs only in how the parent is chosen, while KDLoop takes a structurally different step.

F.2.1 ADAS and AFlow: single-proposal archive hill-climb

Both keep a flat archive $H_t = \{(c_i, f(c_i), \ell_i)\}_{i \le t}$ (ℓ_i the rollout log), draw one parent, and apply the operator set produced by the shared coding-agent proposer g:

$$c_{t+1} = c_{\mathrm{par}} \oplus g(c_{\mathrm{par}}, H_t), \qquad c_{\mathrm{par}} \sim P(\cdot \mid H_t). \tag{3}$$

The generator g (an opaque LLM call emitting $\Delta \in \mathcal{O}^*$ as a native code diff) and the realize / validate / evaluate harness around it (Appendix C) are identical, and every committed candidate is measured on the full objective f. The two methods differ only in the parent law P:

ADAS: $P = \delta_{c_t}$ (archive head), AFlow: $P = \lambda\,\mathrm{unif} + (1-\lambda)\,\mathrm{softmax}\big(100\,\alpha\,s(c)\big)$ over top-K,

with $s(c)$ the candidate's bootstrap-median success rate and AFlow's anti-replay memory steering g off edits already tried on the parent. ADAS edits the last committed candidate, while AFlow stochastically reselects an archived one.

F.2.2 KDLoop: a filtered wave over typed memory

KDLoop departs from the single-proposal template. Over a typed memory H_t (facts, open hypotheses, closed-case lessons including refuted edits, and an intervention-space coverage map), THINK emits up to three axis-tagged operator sets, CRITIC filters them, and the survivors run as one wave on the incumbent c_{par} (the previous iteration's best spec), which advances greedily to the wave's winner:

$$\begin{gathered} \{(a_j, \Delta_j)\}_{j=1}^{m} = \textsc{Think}(H_t)\ \ (m \le 3), \qquad \mathcal{D} = \{\Delta_j : \textsc{Critic}(a_j, \Delta_j, H_t)\}, \\ c_{t+1} = \arg\max_{\Delta \in \mathcal{D}}\ \hat{f}\big(c_{\mathrm{par}} \oplus \Delta\big). \end{gathered} \tag{4}$$

Here a_j is the intervention axis that the operator set Δ_j targets, one lever from the coverage map in H_t, and $\mathcal{D}$ collects the proposals CRITIC admits. After the wave, DISTILL/REFLECT write the outcomes back into H_t. The selection signal $\hat{f}$ is a THINK-composed measurement tier, by default a custom ≥ 30-episode subset, escalating to the full objective f only on demonstrated lift, so KDLoop is the one variant that scores wave candidates more cheaply than the shared evaluator.

F.3 The evaluation function

No policy builds the evaluation function f. It is absorbed into the AgentCanvas substrate that already defines the space. Because a candidate is a closed-loop graph carrying its own environment node (§F.1), *running* it *is* evaluating it: the batch rollout path (§A.4) executes the graph over a held episode suite and returns the success rate $f(c)$. The same path serves every variant (KDLoop's cheap signal

$\hat{f}$ is just that path at a smaller episode budget, the full f being it at the held-suite budget), so f is method-free, and the objective $\max_{c \in \mathcal{C}} f(c)$ belongs to the problem, not to any policy.

Absorbing the evaluator is what makes the representation load-bearing. Unlike text-domain AAS, where scoring is a cheap deterministic check, here f is an expensive, noisy multi-episode rollout, so the static Valid filter of (1) screens malformed candidates *before* any rollout is spent. And each rollout returns more than a scalar: the logs ℓ recorded alongside $f(c)$ are the evidence a policy's memory H_t distils, so a candidate is scored with its trace, not as a bare number.