BRIGADE: BAYESIAN RUNTIME INFERENCE GUIDING AGENTIC DAG EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Decomposing a complex task into a workflow of subtasks lets large language model (LLM) agents work on focused contexts instead of one long context that hurts understanding or exceeds the context window. However, multi-agent workflows are rarely cost-effective: they consume much more budget without a matching gain in performance, because the execution path is unclear, outputs are verified repeatedly, and no principled rule says when to stop. Effective orchestration therefore requires deciding not only which subtask to execute, but also when to acquire verification evidence or stop. To this end, we introduce BRIGADE, a Bayesian decision-theoretic framework for budget-constrained orchestration over a fixed task directed acyclic graph. BRIGADE combines prior knowledge of the model's ability with posterior beliefs over artifact correctness updated by evidence obtained at runtime, and controls the workflow by solving the constrained optimization problem. To make this problem tractable, we derive a lightweight two-step lookahead controller that selects execution, verification, and submission actions online under the budget, without calling an LLM planner at each round. By conditioning the second decision on possible observations, the controller values verification for guiding later actions, even when verification does not directly improve an artifact. Controlled simulations show gains from two-step lookahead when budgets permit additional decisions beyond the cheapest completion. On AppWorld with Doubao executors, BRIGADE shows favorable task-success-cost trade-offs at low and intermediate operating points.

1 INTRODUCTION

Large language model (LLM) agents increasingly tackle complex tasks through multi-agent collaboration: dividing work among specialized agents distributes context and expertise and enables parallel execution (Tran et al., 2025). This decomposition creates a workflow-control problem (Yu et al., 2025): under limited budget and uncertain outputs, the system must decide which subtask to advance, which agent to use, and when to check or revise results.

Existing approaches improve orchestration policies or workflow structures. Learned orchestrators acquire delegation and tool-use strategies, then interpret runtime feedback through LLM calls (Ruan et al., 2026; Su et al., 2025). Workflow optimization instead searches for reusable programs or task-specific communication graphs (Zhang et al., 2024a;b; Zhuge et al., 2024); adapting the plan during execution can require another planner call (Niu et al., 2025). A lightweight alternative would use evidence about completed work to weigh intervention value against cost, without an LLM call for each decision.

Bayesian decision theory connects prior experience with online control: offline calibration supplies initial beliefs, observations update them, and posterior expected utility guides action selection (Papamarkou et al., 2026a). Recent applications control a code candidate or gather evidence about a single latent state (Papamarkou et al., 2026b; Amin, 2026). Extending this principle to a workflow requires locating uncertainty across dependent subtasks and choosing where, how, and with which agent to intervene.

We introduce **BRIGADE** (Bayesian Runtime Inference Guiding Agentic DAG Execution), a lightweight Bayesian controller for budget-constrained multi-agent workflows. A planner decomposes the task once into a directed acyclic graph (DAG) of subtasks and output dependencies.

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

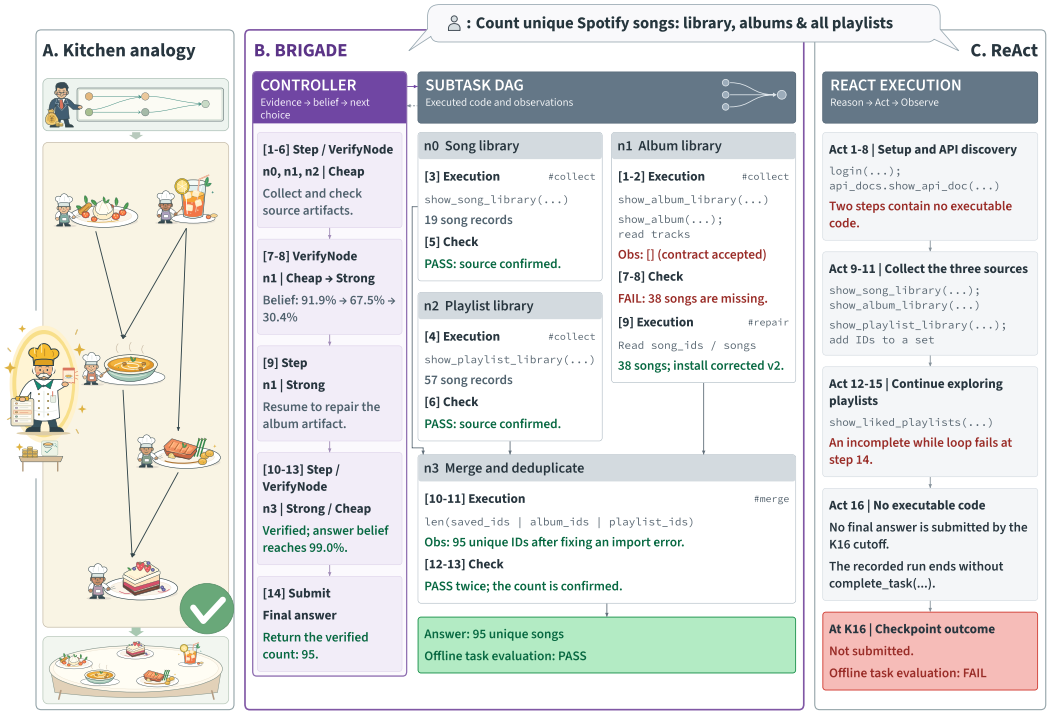


Figure 1: **Overview of BRIGADE and an illustrative AppWorld task. (A) Kitchen analogy.** The planner acts as an executive chef who designs the overall workflow; the BRIGADE controller acts as a head chef coordinating service; node-specific executors act as station chefs preparing individual dishes. Completing the task corresponds to serving the full meal. **(B) Adaptive workflow execution.** To count unique Spotify songs across the song library, albums, and all playlists, the planner constructs a four-node DAG: three collection nodes followed by merging and deduplication. Using runtime feedback, the controller decides which node to advance or verify and whether to use a cheap or strong model, allocating budget as execution unfolds. Each executor maintains its node's context and implements the selected action through concrete API calls and checks. Verification reveals missing album songs, prompting the controller to allocate a strong-model Step to repair the result. **(C) Comparison with ReAct.** BRIGADE submits the correct count of 95 songs at action 14, whereas the illustrated ReAct run has not submitted an answer at the 16-action cutoff.

BRIGADE adapts execution within this fixed graph. Figure 1 illustrates the planner, controller, and executor roles through a kitchen analogy and a Spotify task. Offline runs calibrate correctness priors, action effects, observation likelihoods, and costs; runtime feedback updates task-specific beliefs. The controller selects the subtask, model, and intervention—execution, refinement, verification, or submission—as supported by the agent harness. Evidence is tied to output versions and context, so revising an upstream result can invalidate dependent work while preserving unrelated progress.

Dependencies also make information gathering valuable: checking an upstream result can reveal whether downstream execution or repair is worthwhile. Verification can therefore improve subsequent decisions without changing an artifact. BRIGADE captures this benefit through two-step lookahead in belief space: it scores each feasible intervention using its possible observations and the best continuation after each, executes the first action, and recomputes from the posterior. Numerical belief updates and action scoring reserve model-call budget for execution and evidence gathering.

We evaluate BRIGADE in controlled simulations and on AppWorld (Trivedi et al., 2024). Simulations show higher success for BRIGADE-Q2 than Fixed, Frozen-Prior, and BRIGADE-Q1 at budgets 7.5, 10, and 14. On AppWorld, BRIGADE achieves the highest task goal completion among the compared methods at $K{=}8$ and $K{=}16$. At $K{=}16$, it reaches 70.18% completion versus 47.37%

for ReAct, with 44.0% lower mean execution cost. These are preliminary system-level observations from a single evaluation run; protocol details and complete results appear in Appendix E.

Our contributions are twofold:

- **A lightweight Bayesian controller for MAS workflows.** We combine execution-quality beliefs, workflow dependencies, and two-step lookahead to allocate computation without an LLM scheduling policy.
- **Feasibility and budget efficiency.** Simulations validate the decision-theoretic design, while AppWorld demonstrates improved completion–cost trade-offs in several budget regimes.

2 Related Work

Multi-agent systems. LLM-based MAS research studies agent roles, interaction protocols, and communication structures (Tran et al., 2025). AutoGen enables programmable conversations (Wu et al., 2023), while MetaGPT organizes collaboration through role-specific procedures and shared artifacts (Hong et al., 2023). BRIGADE addresses resource allocation within such collaboration rather than designing a new communication protocol.

Agent workflow control. Workflow coordination involves both policy learning and structural design (Yu et al., 2025). AOrchestra dynamically instantiates instruction–context–tools–model tuples and improves orchestration through supervised fine-tuning or trajectory-based prompt optimization (Ruan et al., 2026); ToolOrchestra uses reinforcement learning with outcome, efficiency, and preference rewards (Su et al., 2025). Both condition runtime choices on feedback through an LLM orchestrator. For structural design, AFlow searches executable workflows using Monte Carlo tree search (Zhang et al., 2024b), G-Designer generates task-conditioned communication topologies with a variational graph autoencoder (Zhang et al., 2024a), and GPTSwarm optimizes prompts and graph connectivity (Zhuge et al., 2024). ADAS similarly searches code-defined agent designs (Hu et al., 2024). Flow revises DAGs through planner calls informed by execution state (Niu et al., 2025). These methods optimize policies or structures; BRIGADE explicitly estimates execution quality within a fixed subtask graph and selects interventions by their posterior value and cost.

Bayesian methods for agent control. Papamarkou et al. (2026a) advocate Bayesian reasoning at the orchestration layer. Papamarkou et al. (2026b) use a posterior over code correctness to select critics, generation, verification, and stopping. Amin (2026) aggregates LLM-elicited likelihoods to update a belief about candidate qualification and selects screening or terminal decisions by expected cost and information value. The latter controls evidence acquisition about a fixed state; neither work allocates execution across a DAG of dependent subtasks. PETS allocates sampling across questions (Liu et al., 2026), while Early Experience trains agent policies from prior interactions (Zhang et al., 2025). BRIGADE instead uses offline executions to calibrate beliefs and action models, then updates task-specific posteriors to jointly select the subtask, agent, and intervention while respecting output dependencies.

3 Method

BRIGADE allocates computation across a subtask workflow using a Bayesian controller. Its decision rule operates on beliefs and predicted outcomes, while an agent harness determines how the selected intervention is executed. We first formulate this separation as a budget-constrained partially observable decision process, then describe the workflow representation, offline calibration, and runtime control loop.

Figure 2 illustrates how calibrated action models and task-specific beliefs jointly guide the next intervention.

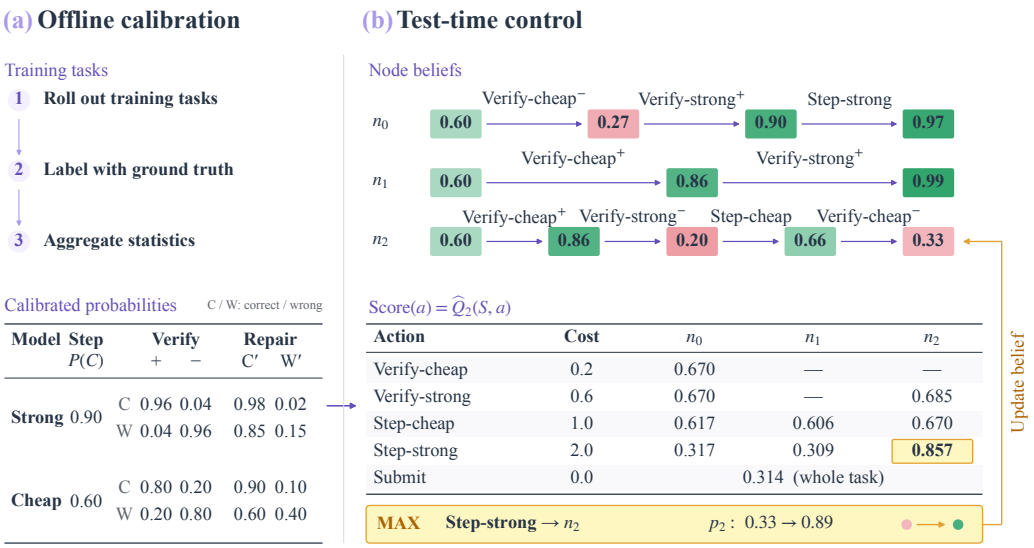


Figure 2: **From offline calibration to online action selection. (a)** Training rollouts are labeled with ground truth to estimate Step correctness, Verify verdict likelihoods, and repair transitions. **(b)** Rows show the observation-conditioned belief history of each node; colors encode correctness probability. Superscripts $+$ and $-$ denote positive and negative verification feedback. The table compares two-step expected task-success scores $\widehat{Q}_2(S, a)$, including Submit; dashes denote unavailable actions. A strong Step on n_2 has the largest score (0.857). The return arrow illustrates the subsequent belief update. Values are illustrative node-only simulator calculations with a remaining budget of 2.0; node-type conditioning and dependency effects are omitted. Repair is a Step on an existing output.

3.1 Budget-Constrained Bayesian Decision Process

Consider a task x executed through an agent harness $\mathcal{H}$. At each decision boundary, ξ_t records the public execution state, including progress, current outputs, and relevant interaction history. The latent state z_t describes task-relevant execution quality, such as whether intermediate results are correct. We model execution as a POMDP (Kaelbling et al., 1998): the controller observes feedback but cannot directly inspect z_t. Given the recorded history H_t, its information state is

$$b_t(z) = P(z_t = z \mid H_t), \qquad S_t = (\xi_t, b_t, B_t), \tag{1}$$

where B_t is the remaining budget.

The harness exposes a finite set of admissible interventions $\mathcal{A}_{\mathcal{H}}(S_t)$, including termination when permitted. An intervention may change a result, obtain information, or do both. After executing a_t, the harness returns an outcome y_{t+1} containing observable feedback, public state changes, and realized cost c_t, giving $B_{t+1} = B_t - c_t$. To evaluate an intervention, the controller must predict both the resulting execution quality and the feedback it will receive. We describe these outcomes with a conditional distribution $K_{\mathcal{H}}(z', y \mid z, \xi, a)$: the probability of next quality state z' and observation y, given current quality z, public state ξ, and action a. This accommodates both repairs that can change correctness and checks that provide evidence about an unchanged result.

For terminal task reward $R(z_T, \xi_T) \in [0, 1]$, our objective is

$$\max_{\pi} \ \mathbb{E}_{\pi}[R(z_T, \xi_T)] \quad \text{s.t.} \quad \sum_{t=0}^{T-1} c_t \le B, \tag{2}$$

where $\pi(a \mid S)$ is the controller's policy for selecting an admissible action given information state S. The budget constraint applies to each execution. The terminal reward is available for offline evaluation, not as runtime feedback.

Verification can improve task success by changing what the controller does next: favorable evidence may support downstream execution, whereas unfavorable evidence may favor repair. To isolate this

benefit, fix a set $\mathcal{C}$ of continuations affordable under the same remaining budget. Let $F(a, y)$ be the expected terminal reward of continuation a after observing y. Choosing a continuation after observing y gives

$$\mathbb{E}_y\left[\max_{a\in\mathcal{C}} F(a, y)\right] \;\geq\; \max_{a\in\mathcal{C}} \mathbb{E}_y[F(a, y)].$$

The left-hand side allows the choice to depend on the observation; the right-hand side selects one continuation before seeing it. When informative feedback changes which continuation is preferable, conditioning the choice on that feedback improves expected terminal reward. In execution, verification consumes budget and can make some continuations unaffordable. Our lookahead accounts for this trade-off by updating both beliefs and remaining budget before evaluating subsequent actions.

3.2 Bayesian Control of Agent Workflows

Task planning. Given task x, a planner constructs a fixed DAG $G = \mathrm{Planner}(x) = (\mathcal{N}, \mathcal{E})$. Each node specifies a subtask and an output contract, and edges identify the upstream outputs it requires. A node can execute only when these inputs are available. If an upstream output is revised, downstream results that used its previous version must be revalidated or regenerated before reuse.

Offline calibration. Offline calibration provides prior outcome probabilities for two main actions: Step, which advances subtask execution, and Verify, which checks the correctness of an existing output. We run these actions on training tasks, label the resulting outputs, and estimate their outcome probabilities. For Step, we estimate output correctness and repair success; for Verify, we estimate the probability of each verdict conditioned on whether the checked output is correct. These calibration tables specify the probabilities of Step outcomes and Verify verdicts. We denote the resulting transition and observation probabilities by $\widehat{K}_{\mathcal{H}}$. We also estimate action costs $\bar{c}_{\mathcal{H}}(S, a)$ from the same training runs. Other datasets may define their own actions while using the same estimation procedure on their respective training tasks.

Runtime posterior updates. During execution, the controller chooses which node to execute or verify and which model to use, or decides to submit the task. The executor maintains the selected node's context and determines the concrete operations, such as generating code or verification queries. It returns observations and actual cost to the controller, which updates its beliefs and remaining budget before choosing the next action.

After action a_t, the controller uses the observed feedback y_{t+1} and the calibration tables $\widehat{K}_{\mathcal{H}}$ to update its belief $b_t(z)$ about the current execution quality:

$$b_{t+1}(z') = \frac{\sum_z \widehat{K}_{\mathcal{H}}(z', y_{t+1} \mid z, \xi_t, a_t) b_t(z)}{\sum_{\tilde{z}, z} \widehat{K}_{\mathcal{H}}(\tilde{z}, y_{t+1} \mid z, \xi_t, a_t) b_t(z)}. \tag{3}$$

The numerator sums over possible previous states z, weighting the prior belief $b_t(z)$ by the calibrated probability of reaching state z' and receiving the observed feedback y_{t+1}. The denominator normalizes these weights across all possible resulting states, so their posterior probabilities sum to one. In practice, we apply this update to individual node outputs as an approximation, avoiding enumeration of all combinations of correct and incorrect nodes; implementation details appear in Appendix B.

Two-step lookahead. Using the updated beliefs and remaining budget, BRIGADE compares candidate actions over two predicted decisions. For each candidate first action, we predict its possible feedback and the resulting states. Within each resulting state, we compare feasible second actions, allowing the second choice to depend on the first feedback. States reached after the second action are scored by a leaf-value estimate $\widehat{V}_{\mathrm{tail}}$ of the remaining task. Only the selected first action is executed; the controller then observes actual feedback and repeats the procedure from the updated state.

Let $\widehat{P}_{\mathcal{H}}(y \mid S, a)$ be the calibrated outcome distribution and $\widehat{\tau}(S, a, y)$ the predicted successor, including updated beliefs and remaining budget. The planning model may use coarser outcomes than runtime inference. For predicted feedback y_1, y_2, write $S_1 = \widehat{\tau}(S, a_1, y_1)$ and $S_2 = \widehat{\tau}(S_1, a_2, y_2)$.

The nonterminal action values and the controller's decision are

$$\widehat{Q}_1(S_1, a_2) = \mathbb{E}_{y_2 \sim \widehat{P}_{\mathcal{H}}(\cdot \mid S_1, a_2)} \left[\widehat{V}_{\text{tail}}(S_2) \right], \tag{4}$$

$$\widehat{Q}_2(S, a_1) = \mathbb{E}_{y_1 \sim \widehat{P}_{\mathcal{H}}(\cdot \mid S, a_1)} \left[\max_{a_2 \in \mathcal{A}_{\mathcal{H}}(S_1)} \widehat{Q}_1(S_1, a_2) \right], \tag{5}$$

$$a_t \in \arg\max_{a \in \mathcal{A}_{\mathcal{H}}(S_t)} \widehat{Q}_2(S_t, a). \tag{6}$$

Here, $\widehat{Q}_1$ averages leaf values over the second action's possible feedback. $\widehat{Q}_2$ selects the best feasible second action separately for each first observation and averages these values; the controller executes the action with the largest $\widehat{Q}_2$. All choices respect dependencies and remaining budget. At either depth, Submit terminates the branch with value $\widehat{U}_{\text{sub}}(S) \approx \mathbb{E}_{z \sim b}[R(z, \xi)]$. The scores require no model calls; actual costs are charged after execution. Appendix B.3 details the decision loop.

Leaf-value estimation. A task may remain unfinished after the two predicted actions. To estimate the value of completing it within the remaining budget, we consider a finite set $\mathcal{R}(S)$ of bounded continuation routes from each leaf state S. Let $\widehat{P}_r(S' \mid S)$ be the terminal-state distribution obtained by propagating calibrated outcomes along route r while deducting predicted costs. We estimate

$$\widehat{V}_{\text{tail}}(S) = \max_{r \in \mathcal{R}(S)} \sum_{S'} \widehat{P}_r(S' \mid S) \widehat{U}_{\text{sub}}(S'). \tag{7}$$

This supplies the largest expected submission value among the candidate routes as the boundary value for two-step planning (Ross et al., 2008). In AppWorld, the routes complete missing work, optionally after revising one existing output. Their construction and evaluation are detailed in Appendix B.4.

4 Experiments

We evaluate the proposed controller in two complementary settings. A fully controlled simulation isolates the value of Bayesian sequential decision making when the data-generating process is known, while AppWorld evaluates the same principle with language-model agents acting in a stateful application environment.

4.1 Simulation

Experimental setup. We use a controlled simulation with no language-model calls. Each episode contains a five-node AND-DAG (chain, fork–join, or layered), executed by three synthetic agents with different cost–accuracy profiles. The controller observes execution feedback and verifier verdicts but cannot access hidden output correctness. We evaluate budgets $B \in \{5, 7.5, 10, 14\}$, with 200 episodes per graph–budget–method configuration (9,600 episodes in total). Each plotted point pools the three graphs, giving 600 episodes. Success requires submitting a workflow with all five outputs correct. Appendix D gives the action model, calibration procedure, controller settings, and statistical protocol.

Compared methods. The four policies use paired instances under the same budget. *Fixed* follows a topological Solve–Verify schedule and conditionally Refines after failed verification. *Frozen-Prior* uses one-step action selection and predictive output probabilities without conditioning them on observed feedback. *BRIGADE-Q1* performs one-step Bayesian action selection. *BRIGADE-Q2* uses two-step planning, allowing the second action to depend on the predicted feedback from the first.

Results. Figure 3 shows that BRIGADE-Q2 achieves the highest observed success rate at budgets 7.5, 10, and 14. At budget 7.5, BRIGADE-Q2 reaches 26.0% success, compared with 18.5% for the strongest baseline; it also exceeds every baseline evaluated with budget 10. At budget 14, BRIGADE-Q2 reaches 45.2%, compared with 39.8% for BRIGADE-Q1. At the lowest budget of 5, Fixed achieves 7.8%, while the three adaptive policies have zero successes. Thus, the benefit of two-step planning appears when the budget permits additional decisions beyond the cheapest completion, rather than uniformly across all budgets.

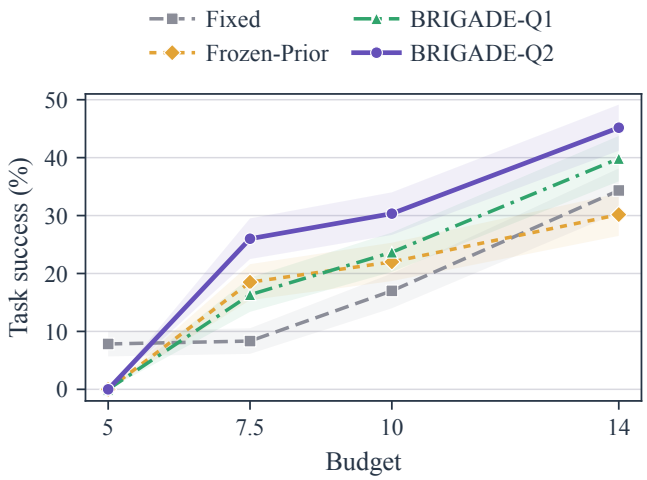


Figure 3: Task success versus budget (600 episodes per point). Shading: 95% normal-approximation intervals. Details in Appendix D.

4.2 AppWorld

Benchmark and metrics. We evaluate on AppWorld v0.1.3.post1, a benchmark of stateful API tasks covering read-only queries, state-changing operations, and multi-application workflows. We report Task Goal Completion (TGC), the fraction of tasks completed successfully, and mean execution cost. Figure 4 focuses on the lower budget settings; Appendix E provides the data split, evaluation protocol, and complete results.

Models and calibration. We use Doubao-Seed-2.1-Pro as the Planner and strong executor, and Doubao-2.0-Lite as the cheap executor. The Planner produces a task DAG that remains fixed during execution. Correctness priors, repair transitions, verifier likelihoods, and action costs are estimated on AppWorld training tasks; Appendix A gives the calibration details.

AppWorld control interface. The controller selects Step, Verify, or Submit and, for Step or Verify, a target node and the cheap or strong executor. Step advances execution or repair by one interaction round; Verify assesses an existing node output. Executors maintain node-level interaction contexts and generate the concrete code or API queries. BRIGADE allocates computation, while executors carry out the selected intervention.

Repeated-action penalty. In this instantiation, we subtract an additional repetition penalty from each nonterminal action value in Eqs. (4)–(5):

$$\rho(S,a) = n_{\mathrm{rep}}(S,a)\frac{\bar{c}_{\mathcal{H}}(S,a)}{B_{\mathrm{rem}}}\widehat{V}_{\mathrm{tail}}(S). \tag{8}$$

For Step, n_{rep} counts preceding execution rounds that have not replaced the current node output; for Verify, it counts previous checks of that output by the selected model tier. The penalty is zero before an output exists, on a first attempt, and for Submit. Repeating an intervention must therefore justify a larger deduction as repetitions accumulate or the remaining budget shrinks. This is an additional decision rule for AppWorld, not an implication of the Bayesian belief update.

Compared methods and budgets. We compare BRIGADE with BFW, Fix, and ReAct. BFW uses a two-step controller with frozen correctness posteriors; Fix executes nodes in topological order with the strong model, without verification or deliberate revision of installed outputs. ReAct operates in a single task-level session without a fixed DAG. We focus on $K \in \{8, 16\}$, where K denotes the ReAct round cap and the corresponding reference execution budget for DAG methods. Appendix E specifies the full budget grid and cost accounting.

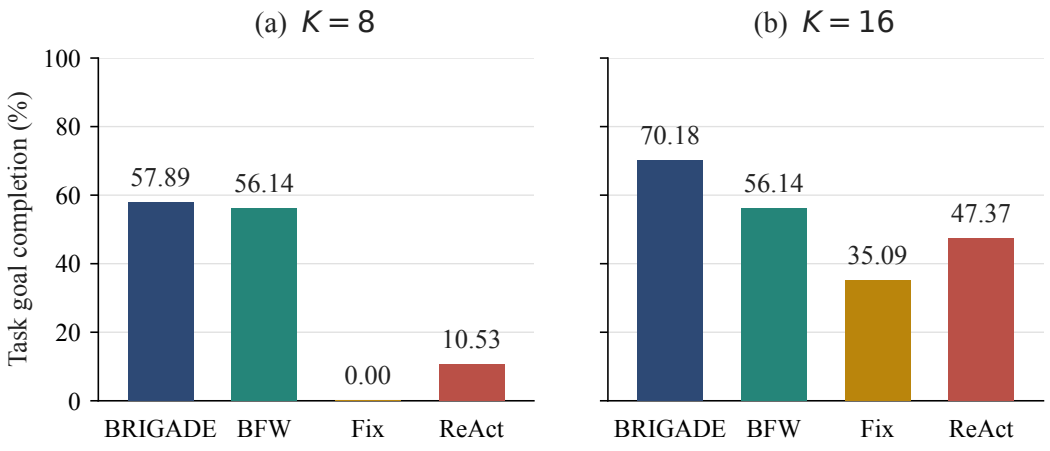


Figure 4: AppWorld task goal completion at $K{=}8$ and $K{=}16$. Complete results and the cost–success curve appear in Appendix E.

Results under limited budgets. **BRIGADE achieves the highest TGC at both low-budget settings** (Figure 4). At $K{=}8$, BRIGADE reaches 57.89% TGC versus 10.53% for ReAct while using 37.6% less mean execution cost; BFW reaches 56.14% and Fix completes no tasks. At $K{=}16$, BRIGADE reaches 70.18% TGC, exceeding BFW, the strongest baseline at this setting, by 14.04 percentage points. Compared with ReAct, it combines **22.81 percentage points higher TGC with 44.0% lower mean execution cost**. Appendix E reports all budget settings and their cost–success trade-offs.

Appendix C illustrates three selected execution traces showing how runtime feedback guides repair, completion checking, and target-set selection; these traces complement the aggregate results.

5 Limitations and Future Work

Evaluation scope. Our experiments with LLM agents are limited to AppWorld and one fixed model pair.

High-budget completion. ReAct reaches higher task completion at the most expensive observed operating points. Future work will examine how the controller can use additional budget for task-level verification and finalization when further computation is valuable. One possibility is a continuation stage that uses remaining budget for additional verification or targeted revision when the base controller would otherwise submit. Such a stage is not part of the reported results.

6 Conclusion

BRIGADE treats execution, verification, and submission within a fixed task DAG as budget-constrained Bayesian decisions. Offline calibration and runtime belief updates support a two-step controller that accounts for how feedback changes the next action, without an LLM call for scheduling. Controlled simulations and AppWorld results support this approach in several budget regimes. Improving high-budget continuation and task-level finalization, and evaluating across seeds and held-out tasks, remain directions for further work.

AI USAGE DISCLOSURE

The research idea originated with the authors. Generative AI tools, including OpenAI Codex, assisted with code implementation and debugging, data processing, result interpretation, figures and tables, and manuscript writing and editing. The authors reviewed all AI-assisted work and take responsibility for the final content.

REFERENCES

Danial Amin. Bayesian Orchestration of Multi-LLM Agents for Cost-Aware Sequential Decision-Making. *arXiv preprint arXiv:2601.01522*, 2026. URL https://arxiv.org/abs/2601.01522v1.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. *arXiv preprint arXiv:2308.00352*, 2023. URL https://arxiv.org/abs/2308.00352.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. *arXiv preprint arXiv:2408.08435*, 2024. URL https://arxiv.org/abs/2408.08435.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Zhangyi Liu, Huaizhi Qu, Xiaowei Yin, He Sun, Yanjun Han, Tianlong Chen, and Zhun Deng. PETS: A Principled Framework Towards Optimal Trajectory Allocation for Efficient Test-Time Self-Consistency. *arXiv preprint arXiv:2602.16745*, 2026. URL https://arxiv.org/abs/2602.16745v2.

Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. Flow: Modularized Agentic Workflow Automation. *arXiv preprint arXiv:2501.07834*, 2025. URL https://arxiv.org/abs/2501.07834.

Theodore Papamarkou, Pierre Alquier, Matthias Bauer, Wray Buntine, Andrew Davison, Gintare Karolina Dziugaite, Maurizio Filippone, Andrew Y. K. Foong, Vincent Fortuin, Dimitris Fouskakis, Jes Frellsen, Eyke Hüllermeier, Theofanis Karaletsos, Mohammad Emtiyaz Khan, Nikita Kotelevskii, Salem Lahlou, Yingzhen Li, Fang Liu, Clare Lyle, Thomas Möllenhoff, Konstantina Palla, Maxim Panov, Yusuf Sale, Kajetan Schweighofer, Artem Shelmanov, Siddharth Swaroop, Martin Trapp, Willem Waegeman, Andrew Gordon Wilson, and Alexey Zaytsev. Position: Agentic AI Orchestration Should Be Bayes-Consistent. *arXiv preprint arXiv:2605.00742*, 2026a. URL https://arxiv.org/abs/2605.00742.

Theodore Papamarkou, Vladislav Smirnov, Viktor Mazanov, Artem Vazhentsev, Preslav Nakov, Timothy Baldwin, and Artem Shelmanov. Bayesian control for coding agents. *arXiv preprint arXiv:2606.24453*, 2026b. URL https://arxiv.org/abs/2606.24453v1.

Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008. doi: 10.1613/jair.2567.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. AOrchestra: Automating Sub-Agent Creation for Agentic Orchestration. *arXiv preprint arXiv:2602.03786*, 2026. URL https://arxiv.org/abs/2602.03786v2.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. ToolOrchestra: Elevating Intelligence via Efficient Model and Tool Orchestration. *arXiv preprint arXiv:2511.21689*, 2025. URL https://arxiv.org/abs/2511.21689.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-Agent Collaboration Mechanisms: A Survey of LLMs. *arXiv preprint arXiv:2501.06322*, 2025. URL https://arxiv.org/abs/2501.06322.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. *arXiv preprint arXiv:2407.18901*, 2024. URL https://arxiv.org/abs/2407.18901.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155*, 2023. URL https://arxiv.org/abs/2308.08155.

Chaojia Yu, Zihan Cheng, Hanwen Cui, Yishuo Gao, Zexu Luo, Yijin Wang, Hangbin Zheng, and Yong Zhao. A Survey on Agent Workflow – Status and Future. *arXiv preprint arXiv:2508.01186*, 2025. URL https://arxiv.org/abs/2508.01186.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-Designer: Architecting Multi-Agent Communication Topologies via Graph Neural Networks. *arXiv preprint arXiv:2410.11782*, 2024a. URL https://arxiv.org/abs/2410.11782.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. *arXiv preprint arXiv:2410.10762*, 2024b. URL https://arxiv.org/abs/2410.10762.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, Jian Xie, Yuxuan Sun, Boyu Gou, Qi Qi, Zihang Meng, Jianwei Yang, Ning Zhang, Xian Li, Ashish Shah, Dat Huynh, Hengduo Li, Zi Yang, Sara Cao, Lawrence Jang, Shuyan Zhou, Jiacheng Zhu, Huan Sun, Jason Weston, Yu Su, and Yifan Wu. Agent Learning via Early Experience. *arXiv preprint arXiv:2510.08558*, 2025. URL https://arxiv.org/abs/2510.08558.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language Agents as Optimizable Graphs. *arXiv preprint arXiv:2402.16823*, 2024. URL https://arxiv.org/abs/2402.16823.

A CALIBRATION EXPERIMENTS

We estimate three quantities from AppWorld training tasks: the correctness of an initial subtask output, the reliability of verifier judgments, and the effect of repair after a negative judgment. Output correctness is labelled offline, separately from the verifier's prediction. Let $Z \in \{C, W\}$ denote a correct or incorrect output. All estimates below use training data; evaluation-task labels are excluded.

A.1 PRIOR SUCCESS PROBABILITY

The success prior measures semantic correctness conditional on producing an output that satisfies the output contract. We collect 34 executions of 17 subtasks from eight training tasks, using both executor tiers. Of these, 29 produce labelled outputs. With a Beta$(1, 1)$ prior, k_m correct outputs among n_m labelled outputs for executor m give

$$\widehat{p}_m = \widehat{P}(Z = C \mid \text{output produced}, m) = \frac{k_m + 1}{n_m + 2}. \tag{9}$$

Because the sample is small, we pool subtask types within each executor tier rather than estimate a separate prior for each type.

Table 1: Prior success probabilities. Counts include only labelled outputs; probabilities use Beta$(1, 1)$ smoothing.

Executor	Labelled outputs n_m	Correct outputs k_m	Prior success (%)
Cheap	14	13	87.5
Strong	15	14	88.2

A.2 VERIFIER RELIABILITY

We evaluate both verifier tiers on 114 labelled outputs from 73 training tasks: 82 correct and 32 incorrect. This yields 227 valid judgments; one invalid response is excluded. A verifier returns PASS, FAIL, or UNKNOWN. Taking correctness as the positive class, we define the conditional rates

$$\begin{aligned} \mathrm{TP} &= P(\mathrm{PASS} \mid C), & \mathrm{TN} &= P(\mathrm{FAIL} \mid W), \\ \mathrm{FP} &= P(\mathrm{PASS} \mid W), & \mathrm{FN} &= P(\mathrm{FAIL} \mid C). \end{aligned} \tag{10}$$

For verifier tier v, subtask type e, and judgment d, we estimate the three-way likelihood with a Dirichlet$(1, 1, 1)$ prior:

$$\widehat{P}(d \mid Z = z, e, v) = \frac{n_{z,d} + 1}{N_z + 3}, \qquad N_z = \sum_{d \in \{\mathrm{PASS,FAIL,UNKNOWN}\}} n_{z,d}. \tag{11}$$

We pool subtask types within a verifier tier when either truth class has fewer than ten samples. Data retrieval has sufficient samples for its own estimates; data processing, state mutation, and control use pooled estimates.

Table 2: Verifier conditional probabilities (%). U_C and U_W are UNKNOWN probabilities given correct and incorrect outputs. Thus $\mathrm{TP} + \mathrm{FN} + U_C = 100\%$ and $\mathrm{TN} + \mathrm{FP} + U_W = 100\%$, up to rounding. Other subtask types share the same pooled estimates.

Subtask type	Verifier	N_C/N_W	TP	TN	FP	FN	U_C	U_W
Data retrieval	Cheap	48/28	62.7	64.5	25.8	11.8	25.5	9.7
Data retrieval	Strong	48/28	68.6	83.9	12.9	17.6	13.7	3.2
Other types	Cheap	82/32	72.9	60.0	31.4	11.8	15.3	8.6
Other types	Strong	81/32	77.4	77.1	20.0	14.3	8.3	2.9

Incorrect outputs are deliberately included to estimate error detection. Their frequency in this diagnostic sample therefore does not replace the success prior. UNKNOWN is retained as a separate outcome, so the reported TP/TN/FP/FN rates are not renormalized over decisive judgments alone.

A.3 REPAIR AFTER A FAIL JUDGMENT

We measure repair from 45 distinct outputs receiving a FAIL judgment across 34 training tasks. Each is retried by both executor tiers after restoring the state preceding the affected operations. The 90 repair trajectories produce 77 labelled outputs within eight rounds; 13 produce no output. We estimate both the probability of correcting an incorrect output and the probability of preserving a correct output that was rejected by the verifier.

Let A denote production of a new labelled output, Z' its correctness, and m, e the repair executor and subtask type. For each initial truth z, let N_z count trajectories with $A = 1$, and K_z count those whose resulting output is correct. Beta$(1, 1)$ smoothing gives

$$\widehat{r}_z^{(e,m)} = \widehat{P}(Z' = C \mid Z = z, \text{FAIL}, A = 1, e, m) = \frac{K_z + 1}{N_z + 2}. \tag{12}$$

If either initial truth class has fewer than three trajectories, we pool subtask types within the repair executor tier under the same intervention. This criterion is applied before restricting to trajectories that produce an output. Data processing requires pooling; the other two types use their own samples.

Table 3: Repair probabilities after a FAIL judgment (%). Repair success is $W \to C$; preservation is $C \to C$. Counts and probabilities condition on a new labelled output. Data-processing estimates use pooled samples, not only data-processing trajectories.

Subtask type	Executor	N_W/N_C	Repair success	Preservation
Data retrieval	Cheap	10/10	58.3	91.7
Data retrieval	Strong	9/10	63.6	91.7
Data processing (pooled)	Cheap	26/15	42.9	88.2
Data processing (pooled)	Strong	20/16	54.5	83.3
State mutation	Cheap	14/4	37.5	83.3
State mutation	Strong	11/4	46.2	66.7

The 13 trajectories without an output are outside the conditional estimate in Eq. (12); the table is not an unconditional repair-success rate. Paired attempts and pooled rows reuse observations, and small truth groups limit precision. These training estimates specify the controller's probability model rather than establish held-out calibration accuracy.

B Runtime Belief Updates and Framework Pseudocode

This appendix describes evidence assignment and posterior updates for node outputs, the two-step controller, and its leaf-value calculation.

B.1 Output Versions and Evidence Scope

The controller maintains a correctness belief for each current node output. For output $o_i^{(k)}$ of node i at version k, this belief is $p_i^{(k)} = P(Z_i^{(k)} = C \mid H_t)$, where $Z_i^{(k)} \in \{C, W\}$ denotes correctness. Each observation concerns a specific output version and its supporting inputs. Reproducing the same output in the same context preserves its belief; evidence about an earlier version is not reapplied as a new observation of its replacement.

An upstream revision invalidates downstream outputs that depend on its previous version. These outputs must be regenerated with the current inputs before they support submission. Dependencies are therefore enforced through output validity, while probabilistic inference operates on individual node outputs. This local representation avoids enumerating joint correctness assignments, but does not capture all correlations among nodes.

B.2 Posterior Updates

Observed feedback updates the correctness belief by Bayes' rule. Let p^- be the belief about the current output before incorporating new feedback y, and let $\ell_z(y) = \widehat{P}(y \mid Z = z, H_t, a_t)$ be its likelihood given correctness z, the previous history, and the selected action. Then

$$p^+ = \frac{p^- \ell_C(y)}{p^- \ell_C(y) + (1 - p^-)\ell_W(y)}. \tag{13}$$

The numerator weights correctness by how likely the feedback is under a correct output; the denominator normalizes over correct and incorrect outputs. The calibration tables provide these probabilities without using evaluation-time correctness labels.

Step. A first output receives the calibrated prior for its executor tier. When a Step replaces an existing output, an available correctness transition gives

$$p^- = p\, T_{CC}(a) + (1 - p)\, T_{WC}(a), \tag{14}$$

where p is the previous output's belief and $T_{zC}(a)$ is the calibrated probability that the replacement is correct given previous correctness z. For repair after FAIL, these are the preservation and repair probabilities in Appendix A. Execution evidence, such as the number of rounds, execution errors, and output-contract feedback, then updates the new output's belief through Eq. (13). This evidence is drawn from the rounds that produced the output and incorporated once. Its likelihoods are estimated from training traces; combining the trajectory signals assumes conditional independence given correctness.

Verify. Verification leaves the output unchanged and updates its current belief using the observed PASS, FAIL, or UNKNOWN verdict. For the first check, the likelihoods come from the verifier calibration table in Appendix A. For repeated checks, the predictive likelihoods also condition on verification history, using a Dirichlet model of verdict probabilities. Thus, repeated judgments are not treated as independent applications of the same fixed likelihood. Across output revisions, past checks can inform verifier reliability, but do not count as fresh evidence of the replacement's correctness.

B.3 TWO-STEP CONTROLLER

Algorithm 1 compares two successive controller decisions and executes only the first. Each predicted first observation induces a separate comparison of second actions, as illustrated in Figure 5. Their scores call $\mathrm{Leaf}(S_2) = \widehat{V}_{\mathrm{tail}}(S_2)$, computed by Algorithm 2. In the pseudocode, p_1 and p_2 are the calibrated probabilities of the corresponding predicted observations; $\widehat{\tau}$ updates the predicted state and remaining budget. For AppWorld, the nonterminal scores at both depths also subtract the repetition penalty in Eq. (8).

Algorithm 1 BRIGADE: two-step control with repeated replanning.

Require: Task x, calibration tables, predicted costs, and budget B.
Build the task DAG and initialize state S with budget B.
while a feasible action remains and execution limits allow **do**
1. Compare two successive decisions
$\widehat{Q}_2(S, \mathrm{Submit}) \leftarrow \widehat{U}_{\mathrm{sub}}(S)$.
for all nonterminal first actions $a_1 \in \mathcal{A}_{\mathcal{H}}(S)$ **do**
$\widehat{Q}_2(S, a_1) \leftarrow 0$.
for all predicted first observations (y_1, p_1) **do**
$S_1 \leftarrow \widehat{\tau}(S, a_1, y_1)$.
$\widehat{Q}_1(S_1, \mathrm{Submit}) \leftarrow \widehat{U}_{\mathrm{sub}}(S_1)$.
for all nonterminal second actions $a_2 \in \mathcal{A}_{\mathcal{H}}(S_1)$ **do**
$\widehat{Q}_1(S_1, a_2) \leftarrow \sum_{(y_2, p_2)} p_2 \,\mathrm{Leaf}(\widehat{\tau}(S_1, a_2, y_2))$.
end for
$\widehat{Q}_2(S, a_1) \leftarrow \widehat{Q}_2(S, a_1) + p_1 \max_{a_2 \in \mathcal{A}_{\mathcal{H}}(S_1)} \widehat{Q}_1(S_1, a_2)$.
end for
end for

2. Execute the selected first action
$a \leftarrow \arg\max_{a \in \mathcal{A}_{\mathcal{H}}(S)} \widehat{Q}_2(S, a)$.
if $a = \mathrm{Submit}$: submit the current result and stop.
$(y, c) \leftarrow \mathrm{Execute}(a)$. ▷ Actual feedback and cost

3. Update and replan
Update outputs, validity, and beliefs from y as described above.
Record the feedback and deduct actual cost c from the budget in S.
end while

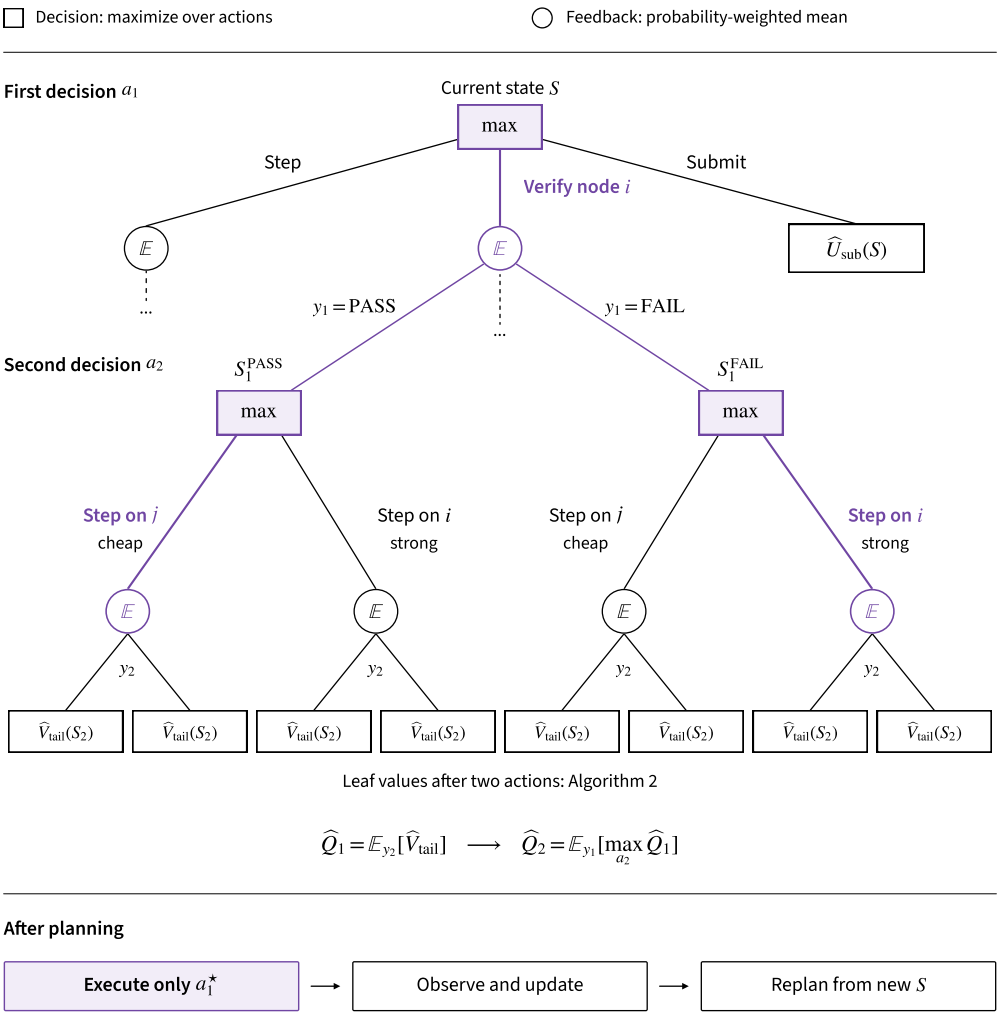


Figure 5: **Two-step decisions conditioned on predicted feedback.** Squares maximize over actions; circles average over possible feedback. The expanded Verify branch illustrates how PASS and FAIL can favor different second actions: a Step on downstream node j or a revision Step on node i. Highlighted action edges show illustrative choices based on the predicted scores; additional actions and feedback branches are omitted. Each state after the second action and feedback receives a leaf estimate from Algorithm 2. Propagating these values upward yields $\widehat{Q}_1$ and $\widehat{Q}_2$ in Eqs. (4)–(5). The controller executes only the selected first action, then updates the state from actual feedback and replans.

B.4 Practical Leaf-Value Estimation

Algorithm 2 supplies the leaf values used after the second predicted action in Algorithm 1. Its base route r_0 retains valid outputs and processes missing nodes in dependency order. For each missing node, it selects the model with the lowest predicted cost and continues Step rounds with that model until an output is produced or resources run out. A finite round allowance bounds the completion policy.

Additional routes $r_{i,m,h}$ first revise one existing output i using model m for at most h rounds. At the first replacement or the revision limit, they follow the same completion policy, including any invalidated downstream outputs. The finite candidate set varies the eligible output, model, and revision length; it introduces no further revision choices or Verify actions within the tail.

Each route propagates calibrated probabilities for producing a new output or continuing execution, deducting predicted costs at each round. A branch ends when the prescribed work finishes, the next Step is unaffordable, or its round limit is reached. Terminal submission values are averaged within each route before maximizing across routes, as in Eq. (7). No additional model calls are made during this evaluation.

Algorithm 2 $\mathrm{Leaf}(S)$: evaluate bounded continuation routes.

Require: Leaf state S, calibration tables, and predicted costs.

1. Form candidate routes
$\mathcal{R}(S) \leftarrow \{r_0\} \cup \{r_{i,m,h}\}$, over eligible nodes, models, and lengths.

2. Compute terminal-state probabilities
for all $r \in \mathcal{R}(S)$ **do**
  Start at S with probability weight 1.
  for all nonterminal branches $(\bar{S}, w)$ generated along r **do**
    Let a be the next Step prescribed by r.
    For each outcome y, add branch $(\widehat{\tau}(\bar{S}, a, y),\ w\widehat{P}_{\mathcal{H}}(y \mid \bar{S}, a))$.
  end for
  Sum weights ending at each S' to obtain $\widehat{P}_r(S' \mid S)$.
end for

3. Compute each route's expected submission value
For each terminal S', set $\widehat{U}_{\mathrm{sub}}(S')$ to the product of required-node correctness beliefs, or zero if any required output is missing or invalid.
$v_r \leftarrow \sum_{S'} \widehat{P}_r(S' \mid S)\widehat{U}_{\mathrm{sub}}(S')$ for each route r.

4. Return the best route value
return $\widehat{V}_{\mathrm{tail}}(S) = \max_{r \in \mathcal{R}(S)} v_r$.

C AppWorld Case Studies

We present three AppWorld case studies comparing BRIGADE with ReAct: repayment and confirmation, note backup, and filtered song downloading. The diagrams separate controller decisions from node execution and show how feedback guides subsequent actions within each workflow. These are selected execution traces; final outcomes are determined by offline task evaluation.

C.1 Repayment and Confirmation

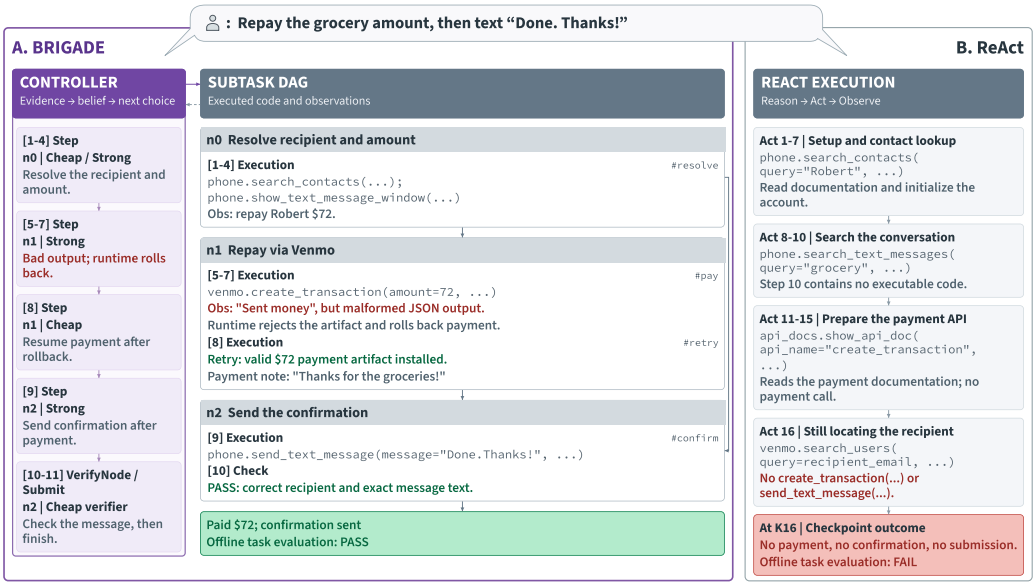


Figure 6: **Repayment and confirmation.** A three-node workflow identifies the recipient and amount, executes a $72 payment, and sends the requested confirmation. The runtime rolls back a payment attempt with invalid output; the controller then resumes the payment node with the cheap model before advancing to confirmation. BRIGADE completes the task at action 11. In the displayed ReAct run, neither payment nor confirmation has been performed at the 16-action cutoff.

C.2 NOTE BACKUP

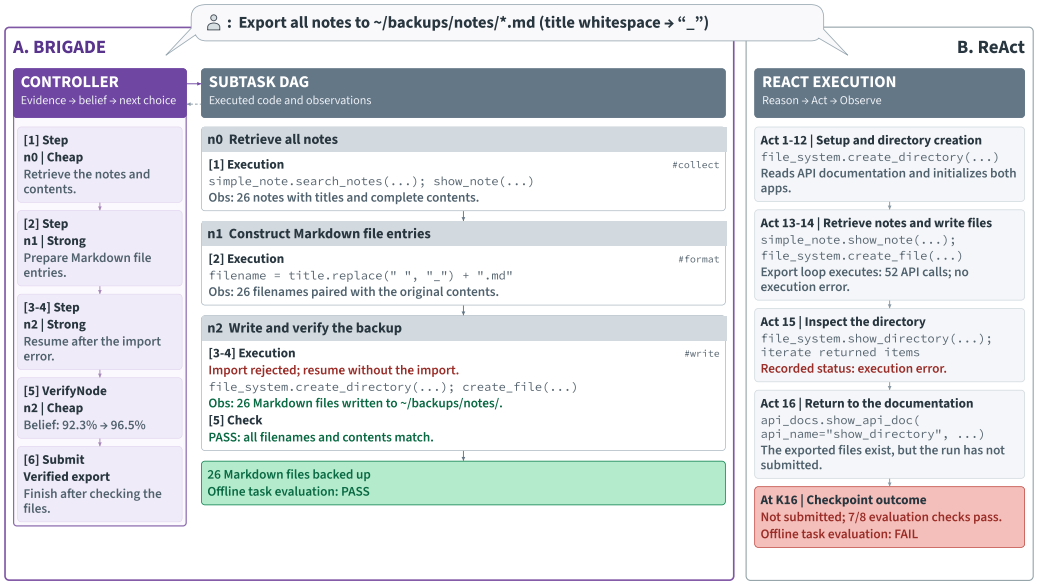


Figure 7: **Note backup and completion checking.** BRIGADE retrieves 26 notes, constructs Markdown file entries, and writes the backup with normalized filenames. After recovering from an import error, it verifies the filenames and contents and submits at action 6. ReAct creates the files but encounters an error during inspection and has not submitted at the 16-action cutoff. Seven of its eight evaluation checks pass, while the overall task evaluation fails.

C.3 FILTERED SONG DOWNLOADING

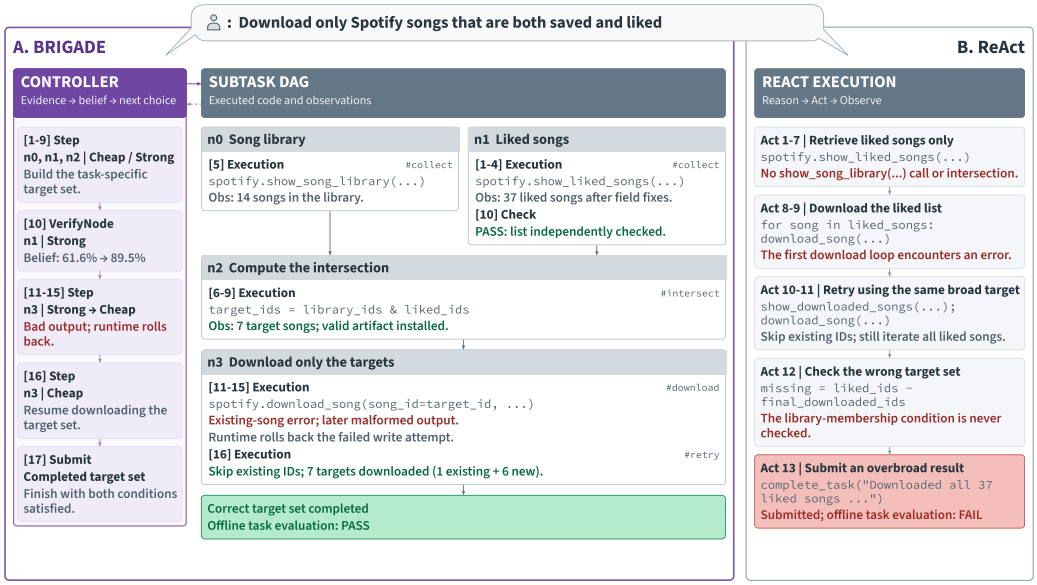


Figure 8: **Downloading the correct target set.** BRIGADE intersects 14 saved songs with 37 liked songs to obtain seven targets. After runtime rollback of a failed download attempt, execution resumes and completes the target set: one existing download and six new ones. BRIGADE then submits at action 17. ReAct instead targets all 37 liked songs and submits at action 13 without enforcing the library-membership condition, failing task evaluation.

D Simulation Experimental Details

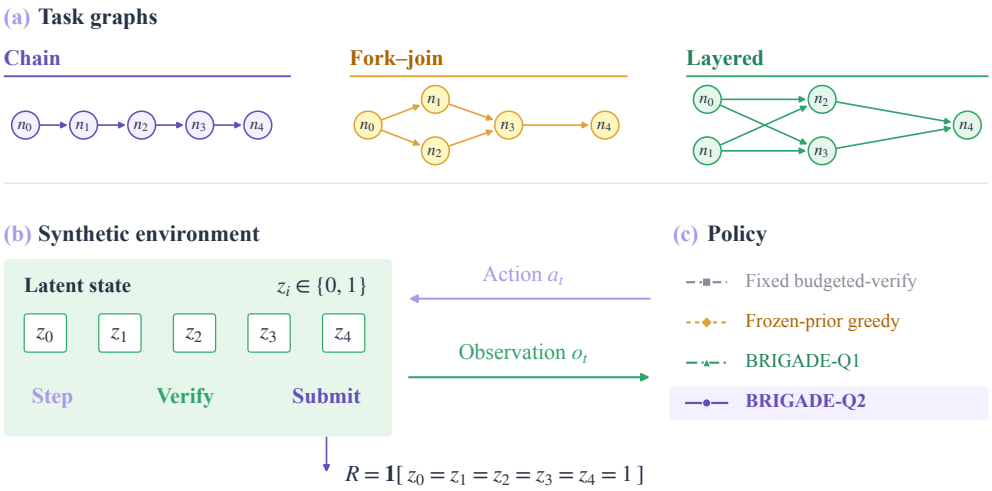


Figure 9: **Controlled simulation setup.** The three five-node task graphs share the same node-type assignment. The controller selects Step, Verify, or Submit and observes feedback, while node correctness remains hidden. Step comprises Solve or Refine. Terminal success requires submitting with all five outputs correct. The policy labels correspond to Fixed, Frozen-Prior, BRIGADE-Q1, and BRIGADE-Q2 in Figure 3.

D.1 Environment and Action Model

The simulation isolates uncertainty about node correctness in a fixed five-node AND-DAG. Nodes $0, \ldots, 4$ have task types reasoning, code, data, code, and reasoning, respectively; this assignment is identical across all three graphs in Figure 9 and Table 4. A node can be solved once its parents have outputs. Edge compatibility is fixed to one, so there is no additional edge uncertainty or edge verification. Hidden correctness is never exposed to the controller during evaluation.

Table 4: The three simulation workflows. Each ordered pair denotes a directed edge.

Graph	Edges
Chain	$(0,1), (1,2), (2,3), (3,4)$
Fork–join	$(0,1), (0,2), (1,3), (2,3), (3,4)$
Layered	$(0,2), (0,3), (1,2), (1,3), (2,4), (3,4)$

Step has two forms: Solve generates an output for an unfinished node, and Refine revises an existing output. Verify checks an existing output without changing it. Submit ends the episode and has zero cost. Each episode allows at most 20 actions, at most one Refine per node, and at most one check by each verifier for a given output version. Every action must fit the remaining budget. Costs are synthetic resource units, not monetary or wall-clock costs.

For node i, a hidden difficulty $d_i \sim \mathrm{Uniform}[-0.6, 0.6]$ is sampled once per episode. If agent m solves a node of type τ_i, its output correctness $z_i \in \{0, 1\}$ is sampled with probability

$$P(z_i = 1 \mid m, \tau_i, d_i) = \sigma(\mathrm{logit}(p_{m,\tau_i}) - d_i)\,,$$

where σ is the logistic function and $p_{m,\tau}$ is given in Table 5. Thus, dependencies constrain execution order, while local correctness is sampled from the agent–type difficulty model. Refine can repair an incorrect output or damage a correct one. In this configuration its transition probabilities are

$$P(z_i' = 1 \mid z_i = 0, m, \tau_i) = p_{m,\tau_i} + 0.05,$$
$$P(z_i' = 0 \mid z_i = 1, m, \tau_i) = 0.18 - 0.12 p_{m,\tau_i}.$$

Table 5: Synthetic agent parameters. The three probability columns give base Solve success probabilities before the difficulty adjustment.

Agent	Code	Data	Reasoning	Solve cost	Refine cost
Generalist	0.62	0.60	0.58	1.0	0.8
Technical specialist	0.88	0.84	0.56	3.0	2.4
Reasoning specialist	0.68	0.70	0.90	4.0	3.2

Both Solve and Refine return binary execution feedback, with $P(\text{positive} \mid z_i = 1) = 0.98$ and $P(\text{positive} \mid z_i = 0) = 0.60$. Explicit verification provides the more discriminative evidence in Table 6; FAIL probabilities are the complements of the corresponding PASS probabilities.

Table 6: Verifier reliability and cost. Correctness is hidden from the controller.

Verifier	$P(\text{PASS} \mid z_i = 1)$	$P(\text{PASS} \mid z_i = 0)$	Cost
Cheap	0.80	0.25	0.25
Strong	0.96	0.05	1.00

D.2 Calibration and Compared Controllers

Solve priors are estimated from a separate calibration sample of 50 labeled executions for each agent–task-type pair, with difficulty drawn from the same distribution as evaluation. A $\mathrm{Beta}(1,1)$ prior gives $\widehat{p}_{m,\tau} = (k_{m,\tau}+1)/52$, where $k_{m,\tau}$ is the number of correct calibration outputs. These estimates are shared across methods and are not refitted on evaluation outcomes. Execution-feedback likelihoods, verifier reliability, Refine transitions, and action costs are specified parameters of this controlled experiment. The Bayesian controllers update local output beliefs from feedback without accessing hidden correctness or updating the shared calibration parameters.

Fixed uses the cheapest agent in topological order. It verifies with the strong verifier while reserving the cheapest remaining completion cost, and Refines after FAIL only when repair, reverification, and the cheapest completion of the remaining nodes fit the budget. Frozen-Prior uses one-step planning and predicts the effect of Solve and Refine on output quality, but ignores observations in both planning and runtime belief updates. BRIGADE-Q1 and BRIGADE-Q2 instead condition on execution feedback and verifier verdicts and use one and two explicit lookahead actions, respectively. Each adaptive policy executes only its selected first action before replanning.

All three adaptive controllers share the same completion estimate at the lookahead boundary. Let M denote the nodes without outputs, b_i the correctness belief for an existing output, and $\boldsymbol{m}$ an assignment of agents to missing nodes. The simulation estimates completion by solving these nodes in a valid order, without further verification or refinement:

$$\widehat{V}^{\mathrm{sim}}_{\mathrm{tail}}(S) = \max_{\boldsymbol{m}:\, \sum_{i\in M} c^{\mathrm{solve}}_{m_i} \le B_{\mathrm{rem}}} \left[\left(\prod_{i \notin M} b_i \right) \left(\prod_{i \in M} \widehat{p}_{m_i,\tau_i} \right) - 0.02 \sum_{i \in M} c^{\mathrm{solve}}_{m_i} \right].$$

The value is zero if no assignment can complete the missing nodes within budget. If all outputs exist, it reduces to their estimated joint success probability. Verification and refinement are evaluated in the explicit lookahead, rather than in this completion estimate.

In this simulation, action scores subtract 0.02 times the immediate action cost in addition to enforcing the budget constraint; the leaf formula applies the same coefficient to future completion cost. Recorded utility is $R - 0.02C$, where R is terminal success and C is total cost. Figure 3 reports success itself, without this cost penalty.

D.3 Evaluation Protocol

We evaluate four methods at budgets $\{5, 7.5, 10, 14\}$ on each of the three graphs, with 200 episodes per configuration, for 9,600 episodes in total. The master random seed is 12345. Calibration uses a separate random stream; within each graph–budget–episode combination, all methods share the

same environment seed. Different budgets use distinct episode streams. This is one evaluation with a shared calibration sample, not an average over independently recalibrated repetitions.

An episode succeeds only if it submits with all five outputs correct. Unfinished or budget-exhausted episodes count as failures and remain in the denominator. Each plotted method–budget point pools 200 episodes from each graph, so $N = 600$ with equal graph weights. We report the mean binary success rate $\widehat{p} = N^{-1} \sum_{j=1}^{N} R_j$. The shaded interval is $\widehat{p} \pm 1.96 s/\sqrt{N}$, where s is the sample standard deviation of these episode outcomes. This normal approximation describes episode-level variation conditional on the fixed calibration sample; it does not measure variation across calibration seeds or provide exact binomial coverage.

E COMPLETE APPWORLD DEVELOPMENT RESULTS

We report a single run on the AppWorld development split (`dev`), with all 57 listed tasks retained in the TGC denominator. BRIGADE is labeled STEP in the original logs. For DAG methods, $K \in \{8, 16, 24, 32\}$ indexes the execution budget matched to a reference ReAct round cap, not the number of controller actions; equal K does not imply equal realized cost. ReAct uses $K \in \{4, 8, 16, 24, 32, 50\}$, evaluated from saved trajectory prefixes with earlier termination respected. The added $K{=}24$ point uses the official checkpoint evaluator and requires no new model calls.

Table 7: Complete single-run AppWorld development results. TGC is the success count divided by 57 listed tasks. Cost is the mean execution cost per recorded episode, in CNY.

Method	K	Successes / tasks	TGC (%) ↑	Cost (CNY) ↓
BRIGADE	8	33 / 57	57.89	0.1876
	16	40 / 57	70.18	0.3881
	24	42 / 57	73.68	0.4587
	32	43 / 57	75.44	0.5082
BFW	8	32 / 57	56.14	0.1859
	16	32 / 57	56.14	0.3516
	24	34 / 57	59.65	0.3807
	32	33 / 57	57.89	0.4193
Fix	8	0 / 57	0.00	0.1651
	16	20 / 57	35.09	0.4502
	24	28 / 57	49.12	0.5623
	32	35 / 57	61.40	0.5825
ReAct	4	0 / 57	0.00	0.1219
	8	6 / 57	10.53	0.3005
	16	27 / 57	47.37	0.6929
	24	46 / 57	80.70	0.9499
	32	49 / 57	85.96	1.0870
	50	54 / 57	94.74	1.2056

There are 56 recorded episodes per DAG-method setting and 57 per ReAct checkpoint. The unrecorded DAG task counts as a failure in TGC. Costs exclude reused planning, calibration, and attempts outside retained terminal records. ReAct caps share one run; independent-run variability and significance are unassessed. Code revisions are not fully controlled across batches, so these results do not isolate individual controller components.

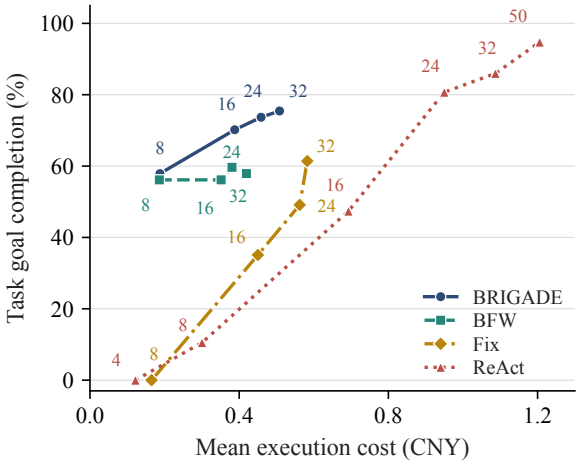


Figure 10: The complete cost range for all four methods, including the higher-cost ReAct checkpoints $K{=}32$ and $K{=}50$. Labels denote K.