# **HEXIS**: COMPILING AGENT SKILLS INTO EXTENDED FINITE STATE MACHINES

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Agent skills provide instructions and reusable knowledge, yet agents must still repeatedly infer which action to take next. This execution paradigm couples task reasoning with control decisions, which can cause agents to misapply or omit required steps. To improve *agent compliance* with skills, we introduce **HEXIS**, which compiles agent skills into extended finite state machines (FSMs) that separate knowledge from control flow. Skill knowledge is incorporated into local instructions that guide reasoning and generation within states. The machine records execution progress and intermediate results, while explicit transition conditions determine subsequent operations. Our incremental compiler first maps skill clauses and tool interfaces to state operations, local instructions, data bindings, and transitions. It then aligns development traces with existing states to identify missing operations and dependencies. These are incorporated by adding or reusing states and refining their connections. Updates are accepted only after static checks and replay of the current and all previously accepted traces. Across four benchmarks and four executors, HEXIS improves success over Skill + ReAct by 16.2 percentage points on average. With Qwen3.8-27B, HEXIS also reduces execution tokens by 38.4–88.9% across benchmarks.

## 1 INTRODUCTION

Agent skills (Agent Skills, 2026) package domain knowledge, instructions, and resources for reuse across tasks. A skill can combine explanations and worked examples with instructions for using scripts, reference materials, and templates. Agents load these materials when needed and use them to guide reasoning and tool use. SkillsBench (Li et al., 2026) reports that curated skills increase the average pass rate from 33.9% to 50.5% on 87 tasks spanning eight domains, illustrating the value of making specialized knowledge available during execution.

A central challenge in skill execution is that agents do not reliably follow skill requirements throughout a task (Li et al., 2025). In common practice, the skill document is supplied as context, and the model infers which actions to take from the instructions and interaction history. Even when the skill requires a particular operation, the model must infer when to perform it (Wu et al., 2024). For example, a model may report a failed check without carrying out the revision required by the skill. AgentIF documents difficulties in following complex conditional requirements and tool specifications (Qi et al., 2025), and SOPBench finds particularly poor procedural compliance among smaller models (Li et al., 2025). In longer tasks, the growing interaction history may further complicate the use of relevant instructions, given models' limitations in using long contexts (Liu et al., 2024). Consequently, requirements that are clearly stated in a skill can still be omitted or incorrectly applied during execution.

Existing research has pursued three relevant directions. First, skill optimization improves the documents available to agents. SkillOpt (Yang et al., 2026) revises skill documents using execution feedback and retains changes that improve validation performance. Second, reflection and memory help models infer appropriate actions from prior experience. Reflexion (Shinn et al., 2023) supplies feedback from earlier attempts, while Agent Workflow Memory (Wang et al., 2025) provides workflow descriptions extracted from experience. Third, programs organize the execution of models and tools. StateFlow (Wu et al., 2024) represents task stages and their transitions with a state machine, while Formal Skill (Zhang et al., 2026) uses a program to track progress and restrict available actions. Other methods construct executable workflows from different sources. AFlow (Zhang et al.,



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.

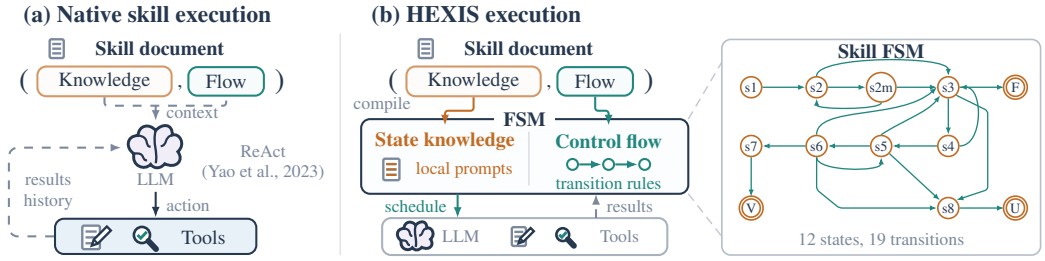


Figure 1: **Separating knowledge from control flow.** Existing works interpret the whole skill as context or memory. HEXIS separates knowledge and flow within an FSM that controls execution. Appendix B.4 details this FSM, whose guards are omitted here.

2025) searches for workflows using execution feedback, TraceCompiler (Yadouni & Li, 2026) derives programs from execution records, and Compile Then Page (Yu et al., 2026) compiles structured procedural requirements into programs.

Skill optimization improves the instructions available to the model, but the model must still determine which operation should follow from the skill and the current execution state. Reflection and memory provide additional experience, yet they likewise leave progress tracking and subsequent operation selection to model inference. StateFlow uses a state machine to organize task execution. The machine directs models and tools to perform the work assigned to the current stage, then determines the next stage from the results (Wu et al., 2024). This design reduces the need for the model to repeatedly infer the next step from instructions and interaction history. However, it does not address the prior problem of constructing such an executable representation from an existing agent skill. This compilation problem is nontrivial because a skill document interleaves task knowledge with procedural requirements over operation order, data dependencies, branching, repetition, and termination. A compiler must therefore decide which content remains available to the model for task-dependent reasoning and which execution relations the runtime can represent and enforce explicitly.

We introduce HEXIS, which compiles existing skills into extended finite state machines (FSMs) that execute tasks using language models and tools, as Figure 1 illustrates. Each machine records the current stage, executes its assigned operation, and applies transition conditions to determine the next stage. Models perform the reasoning and generation needed within each stage. This reduces the need for the model to repeatedly infer subsequent steps from the skill and interaction history, helping agents follow skill requirements more reliably. HEXIS constructs these machines from skill documents and execution traces by specifying operations, data dependencies, and conditions for branching, repetition, and termination, while retaining the skill's knowledge and instructions in model prompts.

Our contributions are as follows:

- We propose HEXIS, which compiles agent skills into extended FSMs that keep skill knowledge in state prompts and enforce control flow through explicit transitions.
- We develop a trace-guided incremental compiler that drafts a machine from the skill and accepts trace-driven updates only after static checks and replay of all accepted traces.
- We evaluate HEXIS on four benchmarks and four executors: machines compiled once exceed Skill + ReAct in 15 of 16 settings, by 16.2 percentage points on average.

## 2 Related Work

**Skill optimization.** Agent skills package instructions, scripts, and resources that agents load on demand (Agent Skills, 2026), and curated skills raise average pass rates across domains (Li et al., 2026). Skill optimization improves this document, as panel (a) of Figure 2 illustrates. SkillOpt (Yang et al., 2026) revises a skill from scored executions and keeps the edits that improve held-out validation performance. The revised text, however, still leaves every control decision to the executor, so the cumulative deviation analyzed in Section 3.1 can persist even with a better skill. Formal Skill (Zhang et al., 2026) makes control explicit by representing skills as programs with executors, hooks, and local runtime state. HEXIS also executes explicit control, but derives it from an existing

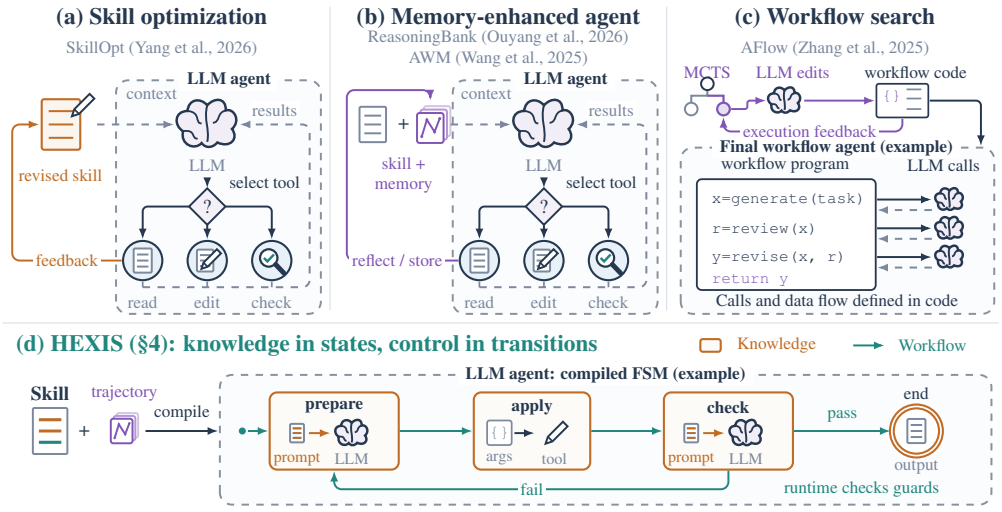


Figure 2: **Separating knowledge from workflow.** In panel (d), HEXIS retains task knowledge and local operations in orange FSM states, while green transitions define the workflow. Panels (a) and (b) provide revised skills or memory as context for the LLM to select tools and control execution. Panel (c) sketches an AFlow program that composes LLM calls.

skill document and refines it with execution traces, so an optimized skill can serve as its input, as Table 2 shows.

**Memory-enhanced agents.** Memory methods let agents reuse experience from earlier executions, as panel (b) of Figure 2 illustrates. Reflexion (Shinn et al., 2023) stores verbal feedback on failed attempts for later trials, Agent Workflow Memory (Wang et al., 2025) induces reusable workflows from successful trajectories and adds them to the context of later tasks, and ReasoningBank (Ouyang et al., 2026) distills strategies from both successful and failed trajectories and retrieves the relevant ones for each new task. These methods enlarge what the model knows but not how its execution is controlled: the model must still decide when and how to apply retrieved experience, and the added context competes with a growing interaction history (Liu et al., 2024). HEXIS instead turns traces into local instructions, variable bindings, and guarded transitions of a reusable machine, whose runtime applies them instead of re-inferring the same control decisions.

**Workflow search.** Workflow methods move control from the model into explicit programs, as panel (c) of Figure 2 illustrates. StateFlow (Wu et al., 2024) models task solving as a state machine whose states and transitions are designed for a task domain, and AFlow (Zhang et al., 2025) searches over code-represented workflows of model calls with Monte Carlo tree search and execution feedback. Other compilers start from different sources: TraceCompiler (Yadouni & Li, 2026) uses a compiler skill to mine agent traces into mostly deterministic workflows, and Compile Then Page (Yu et al., 2026) compiles machine-readable SOP constraints into executable programs with a capability-gated runtime. None of these methods takes an existing skill document as the object to compile, and a searched workflow can behave very differently across executors, as AFlow does in Section 5.3. HEXIS compiles the skill itself, keeping its knowledge in state prompts for model reasoning while the runtime executes its control flow.

## 3 Motivation and Problem Formulation

### 3.1 Motivation

**Sources of deviation in native skill execution.** A skill document $D$ conveys knowledge $\mathcal{K}_D$ about how to perform operations and control requirements $\mathcal{R}_D$ about their order, dependencies, branching, repetition, and termination. These components can be interleaved in natural language. Native execution supplies both as context, together with task input $x$ and history $h_t$, and the model selects an operation $a_t$ and its inputs through

$$a_t \sim \pi_\theta(\cdot \mid D, x, h_t), \tag{1}$$

where $\pi_\theta$ includes the executor's decoding rule. As panel (a) of Figure 1 illustrates, each decision couples applying task knowledge with recovering progress, identifying the applicable control requirements, and choosing the next operation. Supplying the requirements as context influences this choice but does not enforce it. Let $\mathcal{A}_t = \mathcal{A}_D(x, h_t)$ denote the operations and inputs permitted by the skill's applicable requirements. This is a semantic reference, not an action filter assumed available to the executor. The local deviation probability is

$$\epsilon_t(h_t) = \Pr(a_t \notin \mathcal{A}_t \mid D, x, h_t)\,. \tag{2}$$

For example, after correctly identifying a failed check, the model may report the failure instead of carrying out the required revision. Correct evidence alone does not ensure $\epsilon_t(h_t) = 0$; applying the control requirement remains a model decision, even with deterministic decoding. For a fixed horizon $K$, let $S_t$ denote the event that the first $t$ decisions respect the applicable requirements, with $S_0$ certain and permitted termination treated as an absorbing compliant outcome; its complement $S_t^c$ is the event that at least one deviation occurs within the first $t$ steps. Define $\bar{\epsilon}_t = \Pr(S_t^c \mid S_{t-1}, D, x)$ over histories that remain compliant. The chain rule gives the cumulative deviation probability

$$F_K = \Pr(S_K^c \mid D, x) = 1 - \prod_{t=1}^{K}(1 - \bar{\epsilon}_t). \tag{3}$$

**Separating knowledge execution from control.** Panel (b) of Figure 1 illustrates how HEXIS retains the relevant $\mathcal{K}_D$ in state-specific prompts and resource references and compiles $\mathcal{R}_D$ into states, guarded transitions, and termination rules. Models and tools perform local operations; typed variables store their inputs and results. Let $q_t$ be the current state, $a_{q_t}$ its assigned operation, and $\nu_t^+$ the values after that operation. Following a successful nonterminal operation, the runtime selects the first enabled outgoing edge

$$a_t = a_{q_t}, \qquad j^* = \min\{j : g_j(\nu_t^+) = \text{true}\}, \qquad q_{t+1} = q'_{j^*}, \tag{4}$$

where $g_j$ and $q'_j$ are the condition and destination of edge $j$. A model may use the skill's checking criteria to write a verdict, while the runtime routes a failed verdict to revision. When the requirement is faithfully encoded and its condition correctly established, the prescribed continuation no longer requires another model decision. This removes an opportunity for control deviation at that boundary, targeting the repeated risks in Eq. (3).

### 3.2 Problem Formulation

Given a skill document $D$, tool specifications, a task input schema, and development traces, skill compilation constructs an executable extended finite state machine $M$. Skill knowledge guides operations within states, and control relations are expressed as conditional transitions. We focus on whether the machine provides sufficient information to determine how the skill permits execution to continue. For task input $x$ and prior execution history $h$, let $\mathcal{B}_D(x, h)$ denote the set of continuations permitted by the skill. Each continuation includes operations, their inputs and outputs, and how execution terminates. The machine configuration is denoted by $Z_M = (q, \nu)$, where $q$ is the current state and $\nu$ contains the variable values. For each candidate machine, the mapping from task input and history to configuration is fixed before evaluation.

For a fixed skill $D$ and a common distribution $\mu$ over contexts, let $B = \mathcal{B}_D(x, h)$ and assume that $B$ is a discrete random variable with finite entropy. Let $\mathfrak{M}$ denote the family of candidate machines satisfying structural and interface requirements and predefined context storage specifications. The representational objective of compilation is to find a machine in this family that minimizes information loss:

$$\min_{M \in \mathfrak{M}} \mathcal{L}_{\text{info}}(M), \qquad \mathcal{L}_{\text{info}}(M) = H_\mu(B \mid Z_M), \tag{5}$$

where $\mathcal{L}_{\text{info}}$ is the information loss of the representation. The conditional entropy $H_\mu(B \mid Z_M)$ measures the remaining uncertainty about the continuations permitted by the skill, given configuration $Z_M$ under distribution $\mu$. A smaller loss indicates that the configuration better distinguishes these execution semantics. This equation specifies the representational objective; the compilation procedure uses static checks and trace replay to decide whether to accept an update, and Appendix A relates this acceptance rule to the objective.

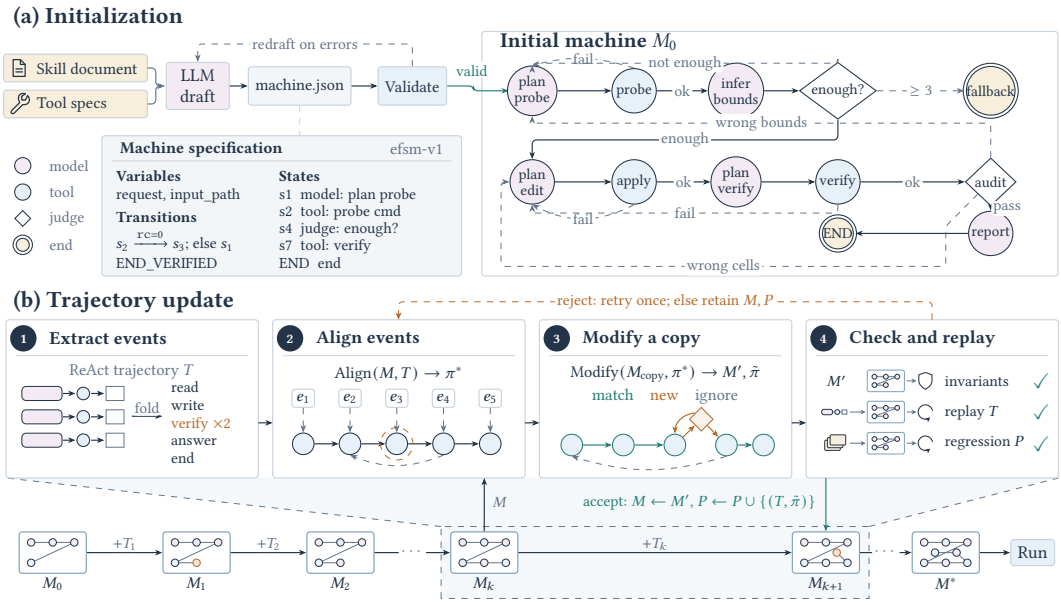


Figure 3: Skill compilation consists of initialization and trajectory updates. Initialization generates and validates an initial machine from the skill document and tool specifications; updates extract events, align states, modify a copy, and check and replay it to obtain the final machine for execution.

# 4 METHOD

To reduce the representation loss defined in the preceding section, HEXIS uses states to record execution stages and variables to store data needed by subsequent operations. As Figure 3 shows, the compiler constructs an initial machine from the skill document, then uses development traces to add missing operations and dependencies. The model applies skill knowledge through local instructions within states, while the runtime selects transitions using variable values.

## 4.1 STATE MACHINE REPRESENTATION

An extended finite state machine is

$$M = (Q,\ q_0,\ V,\ \mathrm{op},\ E,\ F,\ \tau), \tag{6}$$

where $Q$ is the finite state set, $q_0$ the initial state, $V$ the typed variables, op the operations, $E$ the ordered outgoing edges, $F \subseteq Q$ the terminal states, one of which is the fallback state $q_{\mathrm{fb}}$, and $\tau$ their outcome categories. A non-terminal state is a model, judge, or tool state with a read set $R_q$, a write set $W_q$, and local instructions $p_q$ or a tool call $\mathrm{tool}_q$. Execution starts from $z_0 = (q_0, \nu_0)$, with $\nu_0$ assigning task inputs and defaults to $V_0 \subseteq V$.

With $\nu$ the current variable values, $\nu_{R_q}$ its values on $R_q$, and $L_\theta$ the execution model, state $q$ produces

$$y_q = \begin{cases} L_\theta(p_q,\ \nu_{R_q}) & \text{model and judge states,} \\ \mathrm{tool}_q(\nu_{R_q}) & \text{tool states,} \end{cases} \qquad \nu^+ = \nu[W_q \leftarrow y_q], \tag{7}$$

where $\nu[W_q \leftarrow y_q]$ sets the variables in $W_q$ to the fields of $y_q$. Model and judge states receive only $p_q$ and $\nu_{R_q}$, so a state cannot depend on anything the compiler did not declare. A judge state writes one label from a fixed set $\Lambda_q$.

Each outgoing edge $(g_j, q'_j)$ of $q$ has a guard $g_j$, a Boolean test on the variables. After the operation completes, the runtime takes the first edge whose guard is true,

$$j^* = \min\{\, j :\ g_j(\nu^+) \text{ is true} \,\}, \qquad (q, \nu)\ \Rightarrow\ (q'_{j^*},\ \nu^+), \tag{8}$$

or moves to $q_{\mathrm{fb}}$ if the operation fails or its output does not parse. A run stops at a terminal state or at a fixed step limit and returns the outcome category of that state. Every non-terminal state ends with a guard-true edge, so $j^*$ exists, and every loop counter has a limit and an exit, so every cycle is bounded. The pair $z_t = (q_t, \nu_t)$ is the configuration $Z_M$ in Eq. (5).

### 4.2 Initial Compilation

Let $\mathcal{T}$ list the tools, task input fields, and labels. The construction model $L_\phi$ extracts rules from $D$ on required operations, ordering, prohibitions, termination, and event labels, each with a quotation, and a rule is kept only if its quotation occurs in $D$ and its tools and labels are in $\mathcal{T}$; the kept rules form $\mathcal{R}_D$. The construction model then drafts a machine from $D$, $\mathcal{R}_D$, and $\mathcal{T}$, redrafting from the checker's error list for a bounded number of rounds, with every clause of $D$ realized as a state or inside a local instruction.

$\mathrm{Check}(M)$ holds when five conditions pass: the syntax and graph conditions require well-formed fields and guards, every state reachable from $q_0$ and reaching $F$, and a default edge on every non-terminal state; the variable condition is $R_q \subseteq \mathrm{Def}(q)$ for every $q$, where $\mathrm{Def}(q)$ is the set of variables assigned on every path to $q$, the largest solution of

$$\mathrm{Def}(q_0) = V_0, \qquad \mathrm{Def}(q) = \bigcap_{\text{edges } p \to q} \big(\mathrm{Def}(p) \cup W_p\big). \tag{9}$$

Because $\mathrm{Def}$ is taken over all paths, a variable read after a branch must be assigned on every side of the branch. The terminal condition requires the evidence variables of each terminal state $f$ to be in $\mathrm{Def}(f)$. The rule condition checks $\mathcal{R}_D$ on the graph: with $Q_o$ the states performing $o$, $F_{\mathrm{ver}}$ the verified terminal states, and $\mathrm{Reach}(M - Q_o)$ the states reachable from $q_0$ without $Q_o$, a required $o$, an ordering $o_1$ before $o_2$, and a prohibited $o$ hold if and only if

$$F_{\mathrm{ver}} \cap \mathrm{Reach}(M - Q_o) = \varnothing, \qquad Q_{o_2} \cap \mathrm{Reach}(M - Q_{o_1}) = \varnothing, \qquad Q_o = \varnothing, \tag{10}$$

respectively: no verified end is reachable without $o$, and $o_2$ is not reachable without $o_1$. The first draft that passes the syntax and graph conditions becomes $M_0$ if it also satisfies $\mathrm{Check}$; otherwise, or if no draft passes within the round limit, initialization fails.

### 4.3 Trajectory Updates

Round $k$ extracts from one development trace the events $T_k = (e_1, \ldots, e_n;\ \omega)$, each a tool call or a model output with its inputs, outputs, and reasoning text, and the recorded outcome $\omega$. With $L_\phi(e_i, q) = 1$ meaning that the construction model judges state $q$ of $M_k$ able to perform the operation of $e_i$, events are aligned in order:

$$\pi(i) = \begin{cases} q & \text{a state of the same type as } e_i \text{ with } L_\phi(e_i, q) = 1, \\ \star & \text{no such state; a new state is created,} \\ \varnothing & e_i \text{ realizes no skill operation and is ignored.} \end{cases} \tag{11}$$

Tool events also require the same tool, reachable states are preferred, and a new edge to a reused state is rejected if it bypasses a required state.

A copy of $M_k$ becomes $M'_k$: each $\pi(i) = \star$ becomes a new state $q_i$ written from the reasoning text of $e_i$ and the related clauses, $\pi(i)$ then refers to $q_i$, and consecutive aligned events become edges,

$$Q' = Q \cup \{\, q_i :\ \pi(i) = \star \,\}, \qquad E' = E \cup \{\, (\pi(i),\ g_i,\ \pi(i')) :\ e_{i'} \text{ follows } e_i \,\}, \tag{12}$$

where $e_{i'}$ is the next aligned event, $g_i$ the guard read from the outcome of $e_i$, and passed values become variable bindings. If two edges leave $p$ under the same guard $g$ toward different $q_1$ and $q_2$, a new judge state $j$ describing the two events replaces them,

$$\{(p, g, q_1),\ (p, g, q_2)\} \ \to\ \{(p, g, j),\ (j,\ y_j = \lambda_1,\ q_1),\ (j,\ y_j = \lambda_2,\ q_2)\}, \tag{13}$$

so that the label $y_j$ decides between $q_1$ and $q_2$; each new cycle receives a counter.

Let $r_k = (T_k, \pi)$ record the trace and $\mathcal{P}_k$ be the archive of accepted records, $\mathcal{P}_0 = \varnothing$. Replaying $r$ on $M$ runs $M$ from $z_0$ with every state returning its recorded output instead of calling a model or tool, visiting $s_{1:T}$, and

$$\mathrm{Replay}(M, r) = \mathbf{1}\big[\ \pi \preceq s_{1:T},\ \ q_{\mathrm{fb}} \notin s_{1:T},\ \ \tau(s_T) = \omega\ \big], \tag{14}$$

where $\pi \preceq s_{1:T}$ means that the aligned states occur in order in $s_{1:T}$. Because replay uses recorded outputs, a previously accepted trace can fail only if the modification changed a guard or a transition on its path. The candidate is accepted if and only if

$$(M_{k+1}, \mathcal{P}_{k+1}) = \begin{cases} (M'_k, \mathcal{P}_k \cup \{r_k\}) & \mathrm{Check}(M'_k) \wedge \mathrm{Replay}(M'_k, r)\ \ \forall r \in \mathcal{P}_k \cup \{r_k\}, \\ (M_k, \mathcal{P}_k) & \text{otherwise.} \end{cases} \tag{15}$$

A failed candidate is realigned once over reachable states; a second failure discards the trace.

# 5 EXPERIMENTS

This section describes the benchmarks, baselines, and compilation protocol, then reports success rates, request compliance, execution cost, and the combination with skill optimization.

## 5.1 BENCHMARKS AND SKILLS

We evaluate HEXIS on four benchmarks, each paired with one skill document. Every task is scored as a success or a failure, and we report the success rate on held-out test tasks. The last three benchmarks are split approximately 80/20 with random seed 2026. Appendix C lists the sources of all benchmarks and skills.

**Spreadsheet manipulation.** The 107 verified cell-manipulation tasks of SpreadsheetBench (Ma et al., 2024) ask the agent to produce an edited workbook from a natural-language instruction. We use the SigLeak spreadsheet skill (Geng et al., 2026) and split the tasks 50/57 into development and test.

**Mathematical reasoning.** LiveMathematicianBench (LiveMath; He et al., 2026) poses mathematician-level problems with answer options, which we shuffle with a fixed seed. Its SigLeak skill is compiled on 487 development tasks and tested on 121, stratified by month.

**Data analysis.** The 257-question InfiAgent-DABench release (Hu et al., 2024) asks questions about data files that are answered by writing and running code. We use the Pandas Pro skill (Jeffallan, 2026) and a 206/51 split stratified by difficulty.

**Long-context question answering.** LongSeal from SealQA (Pham et al., 2026) asks questions whose evidence is spread over long collections of webpages. We run it offline with only the supplied webpages, use its SigLeak skill, and split the questions 203/51.

## 5.2 BASELINES

We compare against five methods that share the executor and tools with HEXIS but supply the skill differently: as the document itself, induced experience, revised text, or a searched workflow.

**Skill + ReAct** (Yao et al., 2023) supplies the skill document as context, and the model selects every tool call during interleaved reasoning and acting. It is the native execution that HEXIS replaces.

**AWM** (Wang et al., 2025) induces reusable workflows from successful development trajectories and adds them to the context of later runs.

**ReasoningBank** (Ouyang et al., 2026) distills lessons from both successful and failed trajectories and retrieves the relevant ones for each new task.

**SkillOpt** (Yang et al., 2026) revises the skill document under execution feedback and selects the revision to execute natively, testing whether better skill text alone closes the gap.

**AFlow** (Zhang et al., 2025) searches for a workflow of model calls using execution feedback and executes the searched workflow in place of the document.

**Settings.** Execution models are hosted `qwen3.6-flash` (Alibaba Cloud, 2026) and GLM-4.7-FlashX, together with Qwen3.5-9B and Qwen3.8-27B served locally with vLLM (Kwon et al., 2023) in FP8. Within each comparison, methods share the backbone and OpenCode shell/file tools (Anomaly, 2026). Claude Fable 5.1 (Anthropic, 2026a) performs compilation, skill optimization, and memory induction. Machines are compiled and refined solely from the skill document and all `qwen3.6-flash` development trajectories, successful or failed, including observed intermediate results. One machine per skill is then transferred directly to GLM-4.7-FlashX, Qwen3.5-9B, and Qwen3.8-27B, retaining its states, prompts, data bindings, and transitions without target-model recompilation or adaptation. HEXIS is shown in bold in every table.

**Metric.** We measure request-level full compliance, the fraction of executions satisfying every applicable, mechanically verifiable request requirement. Let $A$ be an evaluated method, $\mathcal{X}$ its test set, and $r_A(x)$ the execution trace and artifacts for task $x$. The nonempty set $C_A(x)$ contains the checks applicable to the request and interface supplied to $A$; each check $c$ returns 1 if satisfied and 0 otherwise. The metric is

$$\mathrm{RFC}(A) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \prod_{c \in C_A(x)} c\big(r_A(x)\big). \tag{16}$$

Here, $|\mathcal{X}|$ is the number of tasks, and the product is one only when all applicable checks pass. Figure 4 reports percentages. Checks cover file delivery, output format, and tool-use requirements

independently of answer correctness. For local models on LiveMath, escape sequences in outputs are normalized identically for all methods before checking.

### 5.3 Main Results

**Compiled execution improves success across tasks and executors.** Table 1 shows that HEXIS improves on Skill + ReAct in 15 of 16 settings and is best or tied-best in 11. Its gains are largest on LiveMath, at 31.4–38.0 percentage points, and smallest on DABench, at 1.9–4.0 percentage points, where native success is already 78.4–86.3%. Adding experience or revising skill text helps inconsistently: AWM, ReasoningBank, and SkillOpt each fall below Skill + ReAct on LiveMath with at least two executors. AFlow's searched workflows are competitive on LiveMath but drop to 51.0% and 15.7% on DABench with the local executors, whereas machines compiled only from `qwen3.6-flash` traces improve 11 of 12 settings on the other executors without recompilation. Appendix B.2 shows that the gains persist with Sonnet 5 as the compiler.

Table 1: Success rates (%) on four benchmarks using four execution models. Bold and underline mark the best and second-best result for each benchmark and model.

| Benchmark | Method | Qwen3.6-flash | GLM-4.7-FlashX | Qwen3.5-9B | Qwen3.8-27B |
|---|---|---|---|---|---|
| **Spreadsheet Bench** | Skill + ReAct | 45.6% | 22.8% | 33.3% | 56.1% |
| | AWM | <u>61.4%</u> | 31.6% | 35.1% | 59.6% |
| | ReasoningBank | 43.9% | 15.8% | 33.3% | 63.2% |
| | SkillOpt | 59.6% | <u>40.4%</u> | **47.4%** | 66.7% |
| | AFlow | 36.8% | 22.8% | <u>38.6%</u> | **73.7%** |
| | **HEXIS** | **75.4%** | **47.4%** | <u>38.6%</u> | <u>71.9%</u> |
| **LiveMath** | Skill + ReAct | 44.6% | 12.4% | 40.5% | 33.9% |
| | AWM | 32.2% | 14.9% | 38.0% | 13.2% |
| | ReasoningBank | 37.2% | 8.3% | 39.7% | 41.3% |
| | SkillOpt | 38.0% | 12.4% | 37.2% | 37.2% |
| | AFlow | <u>58.7%</u> | <u>34.7%</u> | <u>41.3%</u> | <u>43.8%</u> |
| | **HEXIS** | **76.9%** | **48.8%** | **71.9%** | **71.9%** |
| **DABench** | Skill + ReAct | 78.4% | <u>78.4%</u> | 82.4% | <u>86.3%</u> |
| | AWM | 74.5% | <u>78.4%</u> | 82.4% | <u>86.3%</u> |
| | ReasoningBank | 80.4% | 74.5% | 78.4% | 82.4% |
| | SkillOpt | **88.2%** | **82.4%** | <u>84.3%</u> | <u>86.3%</u> |
| | AFlow | 80.4% | <u>78.4%</u> | 51.0% | 15.7% |
| | **HEXIS** | <u>82.4%</u> | **82.4%** | **86.3%** | **88.2%** |
| **LongSeal** | Skill + ReAct | 7.8% | 7.8% | <u>23.5%</u> | 17.6% |
| | AWM | 11.8% | <u>9.8%</u> | 19.6% | 23.5% |
| | ReasoningBank | 7.8% | 7.8% | <u>23.5%</u> | 23.5% |
| | SkillOpt | <u>13.7%</u> | **19.6%** | **27.5%** | <u>25.5%</u> |
| | AFlow | <u>13.7%</u> | <u>9.8%</u> | 19.6% | **33.3%** |
| | **HEXIS** | **21.6%** | **19.6%** | <u>23.5%</u> | 23.5% |

**Runtime control raises request compliance.** Figure 4 shows that LiveMath, where the largest success gains occur, is also where native execution complies least: Skill + ReAct fully complies on only 23.1–79.3% of tasks, and HEXIS raises this to 91.7–100%. On DABench, native compliance is already 88.2–98.0%, and the success gains are small. The improvement comes from runtime control rather than from the compiled text, since executing the same machine as a prompt, as Appendix B.1 does, lowers success by 16.8 and compliance by 28.0 percentage points on average. Compliance is necessary but not sufficient: with `qwen3.6-flash` on SpreadsheetBench, SkillOpt and HEXIS both reach 100% compliance yet succeed on 59.6% and 75.4% of tasks, so how states apply skill knowledge also matters.

**Token cost depends on the executor.** Figure 5 shows that HEXIS saves the most tokens on long-context LongSeal, 90.0% relative to Skill + ReAct with `qwen3.6-flash` and 80.8% even with Qwen3.5-9B, consistent with each state reading only its declared inputs rather than the accumulated history. Costs rise slightly with `qwen3.6-flash` on SpreadsheetBench and LiveMath, by 4.9% and 8.9%, where the machines add explicit verification steps. The balance otherwise depends on the executor: with Qwen3.8-27B, HEXIS is the cheapest method on every benchmark,

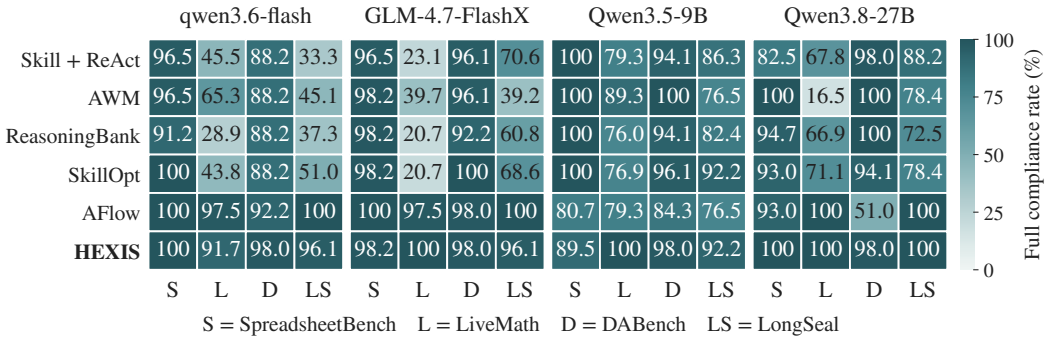


Figure 4: Full compliance rates (%) across four models and benchmarks.

using 38.4–88.9% fewer tokens than Skill + ReAct, and with GLM-4.7-FlashX it cuts costs by 28.4–94.5%, whereas Qwen3.5-9B uses more tokens on the other three benchmarks. Appendix B.3 reports machine sizes and execution diagnostics.

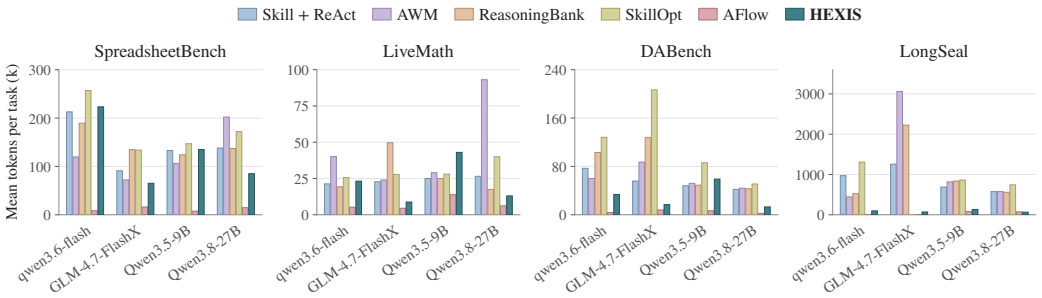


Figure 5: Mean input and output tokens per task (k); each benchmark uses its own axis scale.

**Content optimization and execution structure are complementary.** On SpreadsheetBench, we compile the original and the SkillOpt-optimized documents and compare each with native Skill + ReAct execution. Table 2 shows that SkillOpt with HEXIS reaches 84.2% success at 69k tokens per task, a gain of 24.6 percentage points and a 73.2% token reduction over native execution of the optimized skill. SkillOpt refines the skill's guidance, while HEXIS organizes its application through explicit operations, data dependencies, and transitions. The further gain of 8.8 percentage points over HEXIS on the original skill suggests that better skill content still helps after compilation.

Table 2: Combining skill optimization with state machine execution on SpreadsheetBench. Tokens are mean total execution tokens per task.

| Skill document | Execution | Success (%) | Tokens (k) |
|---|---|---|---|
| Original | Skill + ReAct | 45.6% | 213 |
| Original | **HEXIS** | 75.4% | 223 |
| SkillOpt | Skill + ReAct | 59.6% | 257 |
| SkillOpt | **HEXIS** | **84.2%** | **69** |

## 6 Conclusion

**Limitations.** HEXIS depends on the quality and coverage of skill documents and development traces, so requirements or branches that they never express may be missing from the machine. Static checks and trace replay validate recorded execution paths but do not guarantee correct in-state reasoning or coverage of unseen situations. Targeted trace collection and stronger in-state verification are promising directions for future work.

**Conclusion.** HEXIS compiles skills into extended FSMs with explicit control and data dependencies, retaining in-state reasoning and refining machines through trace alignment, validation, and replay. Across four benchmarks and four executors, it improves native execution in 15 of 16 settings and leads or ties in 11. Source-model compilation supports cross-model reuse without target-model recompilation, while retaining the selected executor's reasoning within each state.

### AI USE STATEMENT

We used generative AI tools to assist with research execution, including writing analysis scripts and interpreting experimental results. We also used these tools to draft and revise parts of the manuscript, improve readability, and assist with translation. The authors take full responsibility for the accuracy, originality, and integrity of the final content.

### ETHICS STATEMENT

This work aims to improve the reliability and transparency of LLM-based skill execution, with evaluations conducted on existing benchmarks. Since the framework can invoke tools and modify artifacts, practical deployment should respect user authorization, apply appropriate access controls, and retain human oversight for consequential actions. Benchmark data, third-party skills, and model services should be used according to their applicable licenses and terms.

### REPRODUCIBILITY STATEMENT

The method section describes state-machine construction, trace-guided refinement, validation, and execution, and Appendix A states the assumptions underlying the theoretical analysis. The experimental setup specifies the benchmarks, baselines, execution models, compilation and transfer protocol, and evaluation metrics. The appendices also provide execution statistics and a concrete state-machine example. Code and supporting experimental artifacts will be released at `https://anonymous.4open.science/r/HEXIS-575F/`.

## REFERENCES

Agent Skills. Agent Skills Specification. Online specification, 2026. URL `https://agentskills.io/specification`. Version accessed September 9, 2026.

Alibaba Cloud. qwen3.6-flash. Alibaba Cloud Model Studio documentation, 2026. URL `https://www.alibabacloud.com/help/en/model-studio/qwen3-6-flash`. Accessed September 10, 2026.

Anomaly. OpenCode: The Open Source AI Coding Agent. Software repository, 2026. URL `https://github.com/anomalyco/opencode`. Accessed September 10, 2026.

Anthropic. Claude Fable 5.1 and Mythos 5.1, September 2026a. URL `https://www.anthropic.com/claude-fable-and-mythos-5-1`.

Anthropic. Claude Sonnet 5, 2026b. URL `https://www.anthropic.com/claude/sonnet`.

Jianing Geng, Ruiqi He, Zekun Fei, Biao Yi, Xuansheng Wu, Ruijie Wang, Zheli Liu, Xia Hu, and Qingkai Zeng. Agent Skills Matter: Inferring Proprietary Skills from Execution Trajectories. *arXiv preprint arXiv:2607.25560*, 2026. doi: 10.48550/arXiv.2607.25560. URL `https://arxiv.org/abs/2607.25560v2`.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. LiveMathematicianBench: A Live Benchmark for Mathematician-Level Reasoning with Proof Sketches. *arXiv preprint arXiv:2604.01754*, 2026. doi: 10.48550/arXiv.2604.01754. URL `https://arxiv.org/abs/2604.01754`.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. InfiAgent-DABench: Evaluating Agents on Data Analysis Tasks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 19544–19572. PMLR, 2024. URL `https://proceedings.mlr.press/v235/hu24s.html`.

Jeffallan. Pandas Pro. Claude Skills repository, 2026. URL `https://github.com/jeffallan/claude-skills/tree/882ef55e377dbf9a4dbe496bb41ac6ccd0e555cf/skills/pandas-pro`. Version 1.1.0.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han-chung Lee, and Dawn Song. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. *arXiv preprint arXiv:2602.12670*, 2026. doi: 10.48550/arXiv.2602.12670. URL https://arxiv.org/abs/2602.12670v4.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, and Xifeng Yan. SOPBench: Evaluating Language Agents at Following Standard Operating Procedures and Constraints. *arXiv preprint arXiv:2503.08669*, 2025. doi: 10.48550/arXiv.2503.08669. URL https://arxiv.org/abs/2503.08669v2.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. SpreadsheetBench: Towards Challenging Real World Spreadsheet Manipulation. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-3007. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/ac840df270ac537dd74530a15c332684-Abstract-Datasets_and_Benchmarks_Track.html.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Thinh Pham, Nguyen Nguyen, Pratibha Zunjare, Weiyuan Chen, Yu-Min Tseng, and Tu Vu. SealQA: Raising the Bar for Reasoning in Search-Augmented Language Models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a0e0f9c39b7f6d07a266a3326216ec40-Abstract-Conference.html.

Yunjia Qi, Hao Peng, Xiaozhi Wang, Amy Xin, Youfeng Liu, Bin Xu, Lei Hou, and Juanzi Li. AGENTIF: Benchmarking large language models instruction following ability in agentic scenarios. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-1892. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/51bb3a8a33610a25aae074bfc51b1b1f-Abstract-Datasets_and_Benchmarks_Track.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 36, pp.

8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. StateFlow: Enhancing LLM Task-Solving through State-Driven Workflows. *arXiv preprint arXiv:2403.11322*, 2024. doi: 10.48550/arXiv.2403.11322. URL https://arxiv.org/abs/2403.11322v5.

Salma El Yadouni and Guanyi Li. TraceCompiler: Skill-Guided Mining and Compilation of LLM Agent Traces into Mostly Deterministic Workflows. *arXiv preprint arXiv:2608.02680*, 2026. doi: 10.48550/arXiv.2608.02680. URL https://arxiv.org/abs/2608.02680v1.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *arXiv preprint arXiv:2605.23904*, 2026. doi: 10.48550/arXiv.2605.23904. URL https://arxiv.org/abs/2605.23904v2.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Chenglin Yu, Li Yin, Qingxin Fan, Ying Yu, RunyangRay Zhong, and Ming Li. Compile, Then Page: Executable SOP Programs and a Capability-Gated Runtime for Procedural LLM Agents. *arXiv preprint arXiv:2607.11346*, 2026. doi: 10.48550/arXiv.2607.11346. URL https://arxiv.org/abs/2607.11346v3.

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. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=z5uVAKwmjf.

Xi Zhang, Meijun Gao, Yuntian Zhao, Xinyu Tan, Yilun Yao, Feiyu Wang, Yanshu Wang, Dingsiyi, and Tong Yang. Formal Skill: Programmable Runtime Skills for Efficient and Accurate LLM Agents. *arXiv preprint arXiv:2605.19604*, 2026. doi: 10.48550/arXiv.2605.19604. URL https://arxiv.org/abs/2605.19604v1.

# A Proof That the Compilation Procedure Satisfies the Problem Formulation

Section 3.2 asks for a machine whose configuration carries the information needed to continue execution as the skill permits, and measures the shortfall by the representation loss of Eq. (5). This appendix proves that the compilation procedure of Section 4 meets this objective as traces accumulate: after $n$ accepted updates the loss of the compiled machine is at most $\varepsilon_n H(B) + h_2(\varepsilon_n)$ with $\varepsilon_n$ of order $|M_n|/n$ given in Eq. (A.1), so the loss vanishes as the number of traces grows, and the machine identifies the full set of permitted continuations once the traces rule out every wrong reading of the skill. The argument rests on one property of the procedure: the acceptance rule of Section 4.3 keeps a candidate machine only if it reproduces every trace accepted before, so the compiled machine is always consistent with all the traces it has learned from. From this property it follows that a machine consistent with $n$ independent traces reproduces a new trace with high probability, and that while a machine reproduces a trace its configuration determines the next operation at every step.

We say that a machine $M$ reproduces a trace $r$ if $\mathrm{Replay}(M, r) = 1$ in Eq. (14): run on the task input and the recorded outputs of $r$, the machine performs the recorded operations in order and ends in the recorded outcome. Let $\nu$ be the distribution of the traces used for updates, and assume that the traces of different rounds are independent draws from $\nu$. The execution model is deterministic in this appendix, so the operation a machine performs next is a function of its configuration. Machines are stored as text over a vocabulary of size $C$, and $|M|$ is the length of the description of $M$.

**Lemma A.1.** *For every $n$, $\mathrm{Check}(M_n)$ holds and $M_n$ reproduces every trace in $\mathcal{P}_n$.*

*Proof.* $M_0$ passes the checks and $\mathcal{P}_0 = \varnothing$. If the claim holds for $n$, then by Eq. (15) either $M_{n+1} = M_n$ and $\mathcal{P}_{n+1} = \mathcal{P}_n$, or $M_{n+1} = M'_n$ passes the checks and reproduces every trace in $\mathcal{P}_n \cup \{r_n\} = \mathcal{P}_{n+1}$. □

The lemma is the only property of the compiler used in Propositions A.2 and A.3. For a machine $M$, let $\mathrm{err}(M)$ denote $\Pr_{r\sim\nu}[\mathrm{Replay}(M, r) = 0]$, the probability that $M$ fails to reproduce a new trace.

**Proposition A.2.** *With probability at least $1 - \delta$ over the traces, the machine after $n$ accepted updates satisfies*

$$\mathrm{err}(M_n) \le \varepsilon_n, \qquad \varepsilon_n = \frac{2|M_n| \ln(2C) + \ln(1/\delta)}{n}. \tag{A.1}$$

*Proof.* Fix $K \ge 1$, let $\mathcal{H}_K$ be the set of machines whose description has length at most $K$, so that $|\mathcal{H}_K| \le C^{K+1}$, and let $\varepsilon(K) = \big(2K \ln(2C) + \ln(1/\delta)\big)/n$. A fixed machine $M$ with $\mathrm{err}(M) > \varepsilon(K)$ reproduces $n$ independent traces with probability at most $(1 - \varepsilon(K))^n \le e^{-\varepsilon(K)n}$. By the union bound, the probability that some $M \in \mathcal{H}_K$ with $\mathrm{err}(M) > \varepsilon(K)$ reproduces all $n$ traces is at most $C^{K+1} e^{-\varepsilon(K)n} \le \delta\, 2^{-K}$. Summing over $K \ge 1$, with probability at least $1 - \delta$ every machine that reproduces all $n$ traces has $\mathrm{err}(M) \le \varepsilon(|M|)$. By Lemma A.1, $M_n$ reproduces all $n$ traces, and $\varepsilon(|M_n|) = \varepsilon_n$. □

The bound holds for whichever machine the compiler returns, because it holds uniformly over all machines that reproduce the accepted traces. Consistency with past traces is therefore what makes the compiled machine transfer to new traces, and the description length $|M_n|$ is the price of the transfer.

We now relate reproduction to the representation loss of Eq. (5). Let $\mu$ be the distribution of contexts obtained by drawing $r \sim \nu$ and then a prefix $h$ of $r$ uniformly at random, let $B$ be the operation recorded in $r$ immediately after $h$, where termination counts as an operation, and let $Z_M(x, h)$ be the configuration of $M$ after running on $h$. Let $h_2$ denote the binary entropy function.

**Proposition A.3.** *For every machine $M$,*

$$H_\mu(B \mid Z_M) \le \mathrm{err}(M)\, H(B) + h_2\big(\mathrm{err}(M)\big). \tag{A.2}$$

*Consequently, with probability at least $1 - \delta$ over the traces, $H_\mu(B \mid Z_{M_n}) \le \varepsilon_n H(B) + h_2(\varepsilon_n)$ whenever $\varepsilon_n \le 1/2$.*

*Proof.* Let $F$ indicate that $M$ fails to reproduce the trace from which the context was drawn, so $\Pr[F = 1] = \mathrm{err}(M)$. Then $H(B \mid Z_M) \le H(B \mid Z_M, F) + H(F)$. If $F = 0$, the machine follows the trace, its configuration after $h$ is $Z_M(x, h)$, and the operation it performs next is a function of that configuration and equals the recorded operation $B$, so $H(B \mid Z_M, F = 0) = 0$. If $F = 1$, $H(B \mid Z_M, F = 1) \le H(B)$. Hence $H(B \mid Z_M, F) \le \mathrm{err}(M) H(B)$, and $H(F) = h_2(\mathrm{err}(M))$. The second claim follows from Proposition A.2 because $h_2$ is increasing on $[0, 1/2]$. □

The loss in Proposition A.3 is measured against the operations that traces record, since a trace reveals the permitted continuations one at a time. The permitted set $\mathcal{B}_D$ itself is identified when the traces rule out every wrong reading of the skill. For a machine $M$ and a context $(x, h)$, let $s_M(x, h)$ be the set of continuations that $M$ can execute from $Z_M(x, h)$; it is a function of $Z_M(x, h)$. A machine reproduces a trace exactly when every continuation recorded in the trace lies in $s_M$ at the corresponding prefix, so by Lemma A.1 the set $s_{M_n}$ contains every continuation recorded in $\mathcal{P}_n$. A map $s$ from contexts to sets of continuations is called incorrect if $\Pr_\mu[s(x, h) \neq \mathcal{B}_D(x, h)] > 0$, and a trace rules out $s$ if it records a continuation outside $s$.

Let $\mathcal{S}$ be a finite set of $N \ge 2$ maps from contexts to sets of continuations that contains $s^\star = \mathcal{B}_D$.

**Assumption 1.** *Every recorded continuation lies in $s^\star$.*

**Assumption 2.** *In every round, each incorrect $s \in \mathcal{S}$ that no earlier trace has ruled out is ruled out by the new trace with probability at least $\gamma \in (0, 1]$, conditionally on the earlier traces.*

**Assumption 3.** *$s_{M_n} \in \mathcal{S}$ for every $n$.*

**Proposition A.4.** *Under Assumptions 1 to 3, the probability $p_n$ that some incorrect member of $\mathcal{S}$ is not ruled out by the first $n$ traces satisfies $p_n \le (N - 1)(1 - \gamma)^n$, and*

$$\mathbb{E}\big[H_\mu(\mathcal{B}_D \mid Z_{M_n})\big] \le p_n \, H(\mathcal{B}_D) \le (N - 1) e^{-\gamma n} H(\mathcal{B}_D). \tag{A.3}$$

*Proof.* By Assumption 2, a fixed incorrect member of $\mathcal{S}$ survives $n$ rounds with probability at most $(1 - \gamma)^n$, and the union bound over the at most $N - 1$ incorrect members gives the bound on $p_n$. If every incorrect member has been ruled out, then each of them excludes some recorded continuation, while $s_{M_n}$ contains all recorded continuations by Lemma A.1 and belongs to $\mathcal{S}$ by Assumption 3; hence $s_{M_n} = s^\star$, and $\mathcal{B}_D(x, h) = s_{M_n}(x, h)$ is a function of $Z_{M_n}(x, h)$, so $H_\mu(\mathcal{B}_D \mid Z_{M_n}) = 0$. Otherwise $H_\mu(\mathcal{B}_D \mid Z_{M_n}) \le H(\mathcal{B}_D)$. Taking expectations gives Eq. (A.3), and $1 - \gamma \le e^{-\gamma}$ gives the last inequality. □

Assumption 1 holds when traces are executions permitted by the skill. Assumption 2 concerns the traces alone: a trace rules out only maps that exclude a recorded continuation, so $\gamma$ is the smallest probability, over the incorrect members of $\mathcal{S}$, that a trace records a continuation the member excludes, and maps that permit more than the skill are never ruled out by traces. Assumption 3 is an idealization of the compiler. Lemma A.1 shows that $M_n$ never returns to a reading ruled out by an accepted trace, but the checks do not force $s_{M_n}$ into a prescribed finite set.

# B Supplementary Results

## B.1 Executing Compiled Skills as Prompts or State Machines

To assess the contribution of program-controlled execution, we compare Skill + ReAct, Prompt-only, and HEXIS with `qwen3.6-flash`. Skill + ReAct uses the original skill document. Prompt-only mechanically renders the compiled machine as a skill document containing local instructions, tool templates, variable declarations and bindings, ordered transition conditions, and termination rules. The model follows these descriptions to track variables and select subsequent operations. HEXIS executes the machine through its runtime. Table 3 reports success and request-level full compliance.

Prompt-only explicitly describes the control flow, but leaves transition selection and progress tracking to the model. It therefore provides no programmatic enforcement of the prescribed procedure. Its success rate is lower than HEXIS on all four benchmarks and even falls below native Skill + ReAct on LiveMath, at 33.9% versus 44.6%. Executing the machine with HEXIS improves success and full compliance over Prompt-only by 16.8 and 28.0 percentage points on average. These results show that explicit state machine execution contributes to the gains, which cannot be explained by textual optimization alone.

Table 3: Success and request-level full compliance with `qwen3.6-flash`. Both metrics are reported as percentages.

| Benchmark | Success | | | Full compliance | | |
|---|---|---|---|---|---|---|
| | Skill + ReAct | Prompt-only | **HEXIS** | Skill + ReAct | Prompt-only | **HEXIS** |
| SpreadsheetBench | 45.6% | 63.2% | **75.4%** | 96.5% | 75.4% | **100.0%** |
| LiveMath | 44.6% | 33.9% | **76.9%** | 45.5% | 82.6% | **91.7%** |
| DABench | 78.4% | 80.4% | **82.4%** | 88.2% | 96.1% | **98.0%** |
| LongSeal | 7.8% | 11.8% | **21.6%** | 33.3% | 19.6% | **96.1%** |

### B.2 Effect of the Compilation Model

To examine whether the gains depend on Fable 5.1 as the compiler, we compile state machines with Claude Sonnet 5 (Anthropic, 2026b) at its low effort setting and execute them with `qwen3.6-flash`. Table 4 compares their success rates with Skill + ReAct and the Fable-compiled machines reported in Table 1.

Table 4: Success rates with `qwen3.6-flash` execution. State machines are compiled by Fable 5.1 or by Sonnet 5 at the low effort setting.

| **Benchmark** | **Skill + ReAct** | **HEXIS** Fable 5.1 | **HEXIS** Sonnet 5 low |
|---|---|---|---|
| SpreadsheetBench | 45.6% | 75.4% | 71.9% |
| LiveMath | 44.6% | 76.9% | 72.7% |
| DABench | 78.4% | 82.4% | 80.4% |
| LongSeal | 7.8% | 21.6% | 19.6% |

Sonnet-compiled machines outperform Skill + ReAct on all four benchmarks and score only 2.0–4.2 percentage points below the Fable-compiled machines. The gains persist across compilers, consistent with the benefit of separating control flow from model reasoning. Explicit state transitions carry out prescribed operations and reduce repeated inference of the next step, without requiring Fable 5.1 as the compiler.

### B.3 Machine Size and Execution Diagnostics

We analyze the size and execution behavior of the compiled state machines using `qwen3.6-flash` as the executor. The statistics are computed from 280 task trajectories: 57 from SpreadsheetBench, 121 from LiveMath, 51 from DABench, and 51 from LongSeal. Each task contributes one execution trajectory. Table 5 summarizes the results.

**Definitions.** We count all serialized states, including terminal states and the fallback placeholder, and every ordered transition once. Entries into the fallback state are intercepted by the *recovery hub* of the execution harness, which either retries from a machine state or hands execution to interpretive fallback. A *recovery task* enters the recovery hub at least once. *Interpretive fallback* starts only when execution actually continues by interpreting the full skill. A *runtime retry* is an explicitly logged `retry` action at the hub; it differs from an ordinary transition that revisits a state for refinement or verification. Machine steps exclude retry records and interpretive fallback steps. A state revisit is a non-fallback state visit after its first occurrence; it can reflect either a compiled loop or work repeated after a runtime retry. All rates below use the number of retained task trajectories as the denominator.

Table 5: Sizes and execution diagnostics of compiled state machines using `qwen3.6-flash`. Retry events count runtime recovery actions, while revisits count repeated visits to non-fallback machine states.

| Benchmark | $N$ | States | Edges | Recovery (tasks) | Fallback (tasks) | Retry events (total) | Steps (mean) | Revisits (mean) |
|---|---|---|---|---|---|---|---|---|
| SpreadsheetBench | 57 | 17 | 35 | 0 (0.0%) | 0 (0.0%) | 0 | 24.56 | 10.60 |
| LiveMath | 121 | 12 | 19 | 3 (2.5%) | 1 (0.8%) | 3 | 9.04 | 0.11 |
| DABench | 51 | 15 | 29 | 0 (0.0%) | 0 (0.0%) | 0 | 13.57 | 1.57 |
| LongSeal | 51 | 16 | 30 | 1 (2.0%) | 1 (2.0%) | 1 | 30.20 | 17.31 |

**Graph size and repeated execution.** The machines contain 12–17 states and 19–35 transitions. Compact graphs can nevertheless support substantial repeated execution: SpreadsheetBench revisits a state in 27/57 trajectories, with a mean of 10.60 repeated visits and up to 67 machine steps. The corresponding revisit frequencies are 12/121 for LiveMath, 22/51 for DABench, and 44/51 for LongSeal. These repetitions include re-analysis, tool execution, and output checking; they should not all be labeled error retries. In particular, SpreadsheetBench has no logged hub retry despite its frequent state revisits.

**Fallback and recovery.** LiveMath records 3 runtime retries across 3 tasks, and LongSeal records 1 across 1 task. Only 1 LiveMath task and 1 LongSeal task enter interpretive fallback; SpreadsheetBench and DABench record none. Thus entry into the recovery hub does not imply that the full skill is reinterpreted. Most trajectories remain within compiled execution; one SpreadsheetBench run reaches its step limit. The retry counts also exclude uninstrumented transport retries and repetitions inside generated tool code. All three LiveMath recovery tasks enter from `s1`. Two resume compiled execution after one retry; the third, task `lm_202512_013`, uses one retry followed by three interpretive steps. In LongSeal task `sq_013`, the calculation counter reaches its limit at `s5`; the hub resets that counter and retries from `s6`. A second counter-limit event leads to two interpretive steps. Both benchmarks use at most one runtime retry per task, with a mean of 0.025 retries per task on LiveMath and 0.020 on LongSeal.

### B.4 An Executed State-Machine Example

We illustrate HEXIS with the frozen LiveMathematicianBench machine used in the main experiments, `machine_v11.json`. It contains 12 stored states, 19 explicit transitions, and 18 typed variables. The states comprise five model actions, two judge actions, two tool actions, two ordinary terminal states, and the designated fallback state. The initial state is `s1`; the artifact declares a 40-step execution limit. Table 6 summarizes the state operations and reproduces the complete ordered transition structure. Model-prompt descriptions are condensed for readability; no state or explicit edge is omitted.

The integer counters $m$, $w$, and $r$ denote `meta_count`, `s3_count`, and `repair_count`, respectively, and all start at zero. The variable $c$ is the most recent tool's `returncode`; $n$ and $v$ are `verify_note` and `verify_verdict`. We write $\perp$ for the stored abstention label. The two task inputs are `request` and `output_path`. Other string variables hold `analysis`, `answer_letter`, `justification`, `write_cmd`, `edit_log`, `file_content`, `result`, `stdout`, and `stderr`; `ok` is Boolean. Together with $n$, $v$, $c$, and the three counters, these are all 18 declared variables. The initial values of `edit_log`, `file_content`, and $n$ are the string `none`.

**How the guards organize execution.** The selection stage can be revisited once through the explicit `s2m`→`s2` edge. Writing is followed by reading and a consistency check before reaching $V$. Here, `verified` means that the delivered file agrees with the machine's selected letter; it does not certify that the mathematical answer is correct. The two judges also have different abstention behavior: `s2m` continues to `s3` on $\perp$, whereas `s6` requests another read while $r < 2$ and otherwise routes to the unverified-report state. These are the actual artifact's rules. Fallback handoff and runtime recovery attempts are implemented by the execution harness, separately from the 19 stored edges and the three machine counters.

**A recorded execution.** For held-out task `lm_202511_026`, the `qwen3.6-flash` trace records the following path:

$$\texttt{s1} \to \texttt{s2} \to \texttt{s2m} \xrightarrow{n=\texttt{incomplete},\, m\leftarrow 1} \texttt{s2} \to \texttt{s3}$$
$$\to \texttt{s4} \xrightarrow{c=0} \texttt{s5} \xrightarrow{c=0} \texttt{s6} \xrightarrow{v=\texttt{pass}} \texttt{s7} \to V.$$

The second visit to `s2` receives the stored `incomplete` feedback and proceeds directly to `s3` because $m = 1$. Both tool calls return zero, and the read-back judge returns `pass`. The trace contains 10 state records, two tool calls, and seven model calls, two of which are made by the judges; $w = r = 0$ throughout. The result log records 25,894 total tokens, 89.41 seconds, zero runtime retries, and no interpretive fallback. Thus, the repeated selection step is a programmed semantic revision, not a fallback recovery attempt. The task's answer and generated reasoning are omitted here because the control path, labels, and counters suffice to reproduce this execution account.

Table 6: The actual LiveMath v11 machine. Outgoing rules are evaluated from top to bottom after the state's operation; "else" is the stored unconditional edge. Counter increments occur only when the corresponding edge is taken. $V$, $U$, and $F$ abbreviate `END_VERIFIED`, `END_UNVERIFIED`, and `FALLBACK`.

| State | Assigned operation | Ordered outgoing rules |
|---|---|---|
| `s1` | Model: analyze the question, hypotheses, option support, and relative strength; write `analysis`. | Always $\rightarrow$ `s2` |
| `s2` | Model: select an option using `request`, `analysis`, and $n$; write `answer_letter` and `justification`. | $m \geq 1 \rightarrow$ `s3`;<br>else $\rightarrow$ `s2m` |
| `s2m` | Judge: check whether the selected option covers the stated conclusion; write $n \in \{\texttt{complete}, \texttt{incomplete}, \bot\}$. | $n = \texttt{incomplete} \wedge m < 1$<br>$\rightarrow$ `s2`, $m \leftarrow m + 1$;<br>else $\rightarrow$ `s3` |
| `s3` | Model: generate `write_cmd` to write the selected letter in the required boxed format to `output_path`, using `edit_log` when available. | $w \geq 4 \rightarrow F$;<br>$w < 4 \wedge r \geq 2 \rightarrow$ `s8`;<br>else $\rightarrow$ `s4` |
| `s4` | Tool: execute `bash` with `command=write_cmd`; bind the returned `stdout` to `edit_log`. | $c = 0 \rightarrow$ `s5`;<br>else $\rightarrow$ `s3`, $w \leftarrow w + 1$ |
| `s5` | Tool: execute `read` with `filePath=output_path`; bind the returned `stdout` to `file_content`. | $r \geq 2 \rightarrow$ `s8`;<br>$r < 2 \wedge c = 0 \rightarrow$ `s6`;<br>else $\rightarrow$ `s3`, $w \leftarrow w + 1$ |
| `s6` | Judge: compare `file_content` with `answer_letter`; write $v \in \{\texttt{pass}, \texttt{wrong_content}, \bot\}$. | $v = \texttt{pass} \rightarrow$ `s7`;<br>$v = \texttt{wrong_content} \wedge r < 2$<br>$\rightarrow$ `s3`, $r \leftarrow r + 1$;<br>$v = \bot \wedge r < 2$<br>$\rightarrow$ `s5`, $r \leftarrow r + 1$;<br>else $\rightarrow$ `s8` |
| `s7` | Model: report the file's content, selected letter, and justification in `result`. | Always $\rightarrow V$ |
| `s8` | Model: report the intended answer and observed file status in `result`, explicitly marking it unverified. | Always $\rightarrow U$ |
| $V$ | End action with terminal kind `verified`; return `result`. | No stored outgoing edges |
| $U$ | End action with terminal kind `unverified`; return `result`. | No stored outgoing edges |
| $F$ | Designated fallback state, stored as an end placeholder with terminal kind `fallback`; intercepted by the execution harness. | No stored outgoing edges |

## C LICENSES

Table 7 lists the licenses and URLs of the benchmarks, skills, baselines, models, and serving software used in this work. SigLeak releases its skills through an anonymous repository, and the hosted models are used under their providers' terms of service.

Table 7: Resource licenses and URLs.

| Resource | License | URL |
|---|---|---|
| SpreadsheetBench (Ma et al., 2024) | CC BY-SA 4.0 | https://huggingface.co/datasets/KAKA22/SpreadsheetBench |
| LiveMathematicianBench (He et al., 2026) | Available online | https://huggingface.co/datasets/LiveMathematicianBench/ |
| InfiAgent-DABench (Hu et al., 2024) | Apache 2.0 | https://github.com/InfiAgent/InfiAgent |
| SealQA (Pham et al., 2026) | Apache 2.0 | https://huggingface.co/datasets/vtllms/sealqa |
| SigLeak skills (Geng et al., 2026) | Available online | https://anonymous.4open.science/r/SigLeak-D1DB |
| Pandas Pro (Jeffallan, 2026) | MIT | https://github.com/jeffallan/claude-skills |
| AWM (Wang et al., 2025) | Apache 2.0 | https://github.com/zorazrw/agent-workflow-memory |
| ReasoningBank (Ouyang et al., 2026) | Apache 2.0 | https://github.com/google-research/reasoning-bank |
| SkillOpt (Yang et al., 2026) | MIT | https://github.com/microsoft/SkillOpt |
| AFlow (Zhang et al., 2025) | MIT | https://github.com/FoundationAgents/AFlow |
| Qwen3.5-9B | Apache 2.0 | https://huggingface.co/Qwen/Qwen3.5-9B |
| Qwen3.8-27B (FP8) | Apache 2.0 | https://huggingface.co/Qwen/Qwen3.8-27B-FP8 |
| qwen3.6-flash (Alibaba Cloud, 2026) | Terms of service | https://www.alibabacloud.com/help/en/model-studio/ |
| Claude Fable 5.1 (Anthropic, 2026a) | Terms of service | https://www.anthropic.com/claude-fable-and-mythos-5-1 |
| OpenCode (Anomaly, 2026) | MIT | https://github.com/anomalyco/opencode |
| vLLM (Kwon et al., 2023) | Apache 2.0 | https://github.com/vllm-project/vllm |