LLM Learning State Engines Improve Reasoning with Self-Supervised Operation Discovery

Anonymous authors
Paper under double-blind review

Abstract

We want LLMs to make explicit logical choices and execute them consistently, providing a basis for stable extrapolation to longer or unfamiliar problems. Checkable reasoning and computation matter for scientific discovery and software security (Romera-Paredes et al., 2024; Klein et al., 2009). Chain-of-thought (CoT) exposes intermediate steps (Wei et al., 2022), but choosing an operation and carrying it out share the same generation process. Rule-based systems obtain explicit control through hand-designed operations. In this paper, we introduce LLM Learning State Engines that learn this interface from transition examples using self-supervised targets. For Qwen3-1.7B, we cluster hidden-state changes between adjacent proof states into anonymous codes and train their use with evidence and next-line targets. The classes align with human proof roles; fixing or permuting their codes eliminates successful proofs. Separate evidence generation and selection support one-line updates. On PrOntoQA (Saparov & He, 2023), the engine improves answer accuracy from 48.33% to 81.67% and exact-process accuracy from 23.33% to 57.78% over Token CoT. Tape experiments test learned execution, new operation sequences, and encoder transfer. With matched Transformer architectures, delegating execution reduces prediction to action selection and improves complete-trajectory accuracy beyond training horizons.

1 Introduction

Autoregressive language models learn next-token prediction; large-scale pretraining lets them use instructions and examples to perform new tasks (Brown et al., 2020). Scratchpad training introduced intermediate calculations (Nye et al., 2021), and chain-of-thought (CoT) prompting elicits natural-language reasoning steps (Wei et al., 2022; Kojima et al., 2022). Our concern is whether the logic behind these steps is clear and controllable, and whether a chosen operation is executed consistently. Mathematics builds new results on explicit assumptions and deductions; physical theory also depends on consistent derivations from stated laws. For LLMs to support such work, their reasoning procedures must remain stable as problems change or computations grow longer. Yet a generated proof can cite the right rule and write a consequence that does not follow. CoT leaves selection and execution in the same generation process. Process supervision provides feedback on intermediate steps (Uesato et al., 2022; Lightman et al., 2024); Coconut and Abstract CoT carry computation in hidden states or reserved tokens (Hao et al., 2025; Ramji et al., 2026). Faithfulness studies also show that readable rationales need not reflect the computation behind an answer (Turpin et al., 2023; Lanham et al., 2023; Cox et al., 2026; Jia et al., 2026). We focus on proofs, derivations, and program traces, where dependable logical steps matter for scientific discovery (Romera-Paredes et al., 2024) and software security (Klein et al., 2009).

Traditional rule-based systems obtain explicit control by specifying each rule's conditions and effects. Once selected, a rule is applied by a fixed execution procedure. The designer supplies the rule vocabulary, representations, and execution machinery. Toolformer learns when and how to call supplied APIs through self-supervision (Schick et al., 2023); Logic-LM translates problems for a symbolic solver (Pan et al., 2023); adaptive solver composition selects and combines symbolic solvers (Xu et al., 2026). These systems depend on the operations already provided. Extending

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.

their control to new reasoning procedures still requires deciding what the operations should be and implementing them.

In this paper, we introduce the *LLM Learning State Engine* to make operation selection explicit and learn its connection to execution from transition examples. Self-supervised targets group recurring state changes into anonymous codes; their examples teach the model to select and use them. This supplies an operation interface without naming and implementing each operation separately. Learning representations for execution connects the approach to world models and neural algorithmic reasoning (Ha & Schmidhuber, 2018; Veličković & Blundell, 2021). CTRLS clusters reasoning-prefix representations to condition later segments (Wu et al., 2026), while RA3 learns short rationales that guide expert code generation (Zhang et al., 2026). We cluster changes between adjacent states. For Qwen, these are hidden-state changes between proof prefixes. Separate modules choose a code, gather evidence, select what is needed next, and write a line. The resulting proof becomes the input to the next decision.

Section 2 defines the learned engines and matched comparison. Section 3 first connects Qwen's codes to human proof roles and tests their control over generation. Tape experiments inspect execution, composition, and transfer. Matched Token and State models then show how assigning execution to a separate component reduces the network's choices and stabilizes prediction under the same architecture. Longer rollouts examine extrapolation beyond training horizons. Section 4 discusses the remaining obstacles and the use of explicit, stable reasoning in science.

2 METHOD

We make the chosen operation an explicit input to each workspace update. The Qwen and tape engines learn these operations from examples; matched Token and State models use supplied actions to compare prediction under the same architecture.

2.1 QWEN STATE ENGINE

For Qwen, we use the proof prefix as the workspace. Let x be the fixed problem, including its source statements and query, and let $w_t = (u_0, \ldots, u_{t-1})$ contain the proof lines already written. One update chooses an anonymous operation code a_t, selects evidence e_t, and writes the next line u_t, giving $w_{t+1} = w_t \parallel u_t$. The workspace contains proof text. Hidden-state vectors are used to discover the code vocabulary; the modules exchange code IDs and text during execution.

We first group changes between adjacent training proof states to obtain code labels. We then train four modules on these labels: Policy chooses a code, Grounder collects source evidence, Route Value selects an item, and Executor writes one line. Figure 1 shows the loop. A code identifies a recurring kind of update; the problem and evidence supply the content for a particular instance.

2.1.1 DISCOVERING PROCEDURAL CATEGORIES FROM PROOF TRANSITIONS

For a demonstrated transition from w_t to w_{t+1}, let h_t and h_{t+1} denote Qwen's extracted hidden-state representations. We cluster their differences, so the input to category discovery describes the change made by the step:

$$\Delta h_t = h_{t+1} - h_t, \qquad v_t = \mathrm{PCA}_{48}(\mathrm{normalize}(\Delta h_t)), \qquad a_t^* = C(v_t). \tag{1}$$

PCA (principal component analysis) projects the normalized differences onto directions of greatest variance across training transitions. We retain 48 components, so $v_t \in \mathbb{R}^{48}$ is a 48-dimensional vector describing the change from w_t to w_{t+1}. These vectors are the inputs to clustering.

K is the number of clusters, each supplying one anonymous operation code. We fit MiniBatch k-means separately for $K = 2, \ldots, 8$. The silhouette score measures how closely vectors group within a cluster and how well clusters separate. We select the partition maximizing this score minus $0.005K$, which adds a small cost for more categories. In Eq. 1, $C(v_t)$ returns the assigned cluster ID, and the star in a_t^* marks a training target.

Transitions involving different entities and predicates can receive the same code. These assignments supply the self-supervised category targets, and the demonstrated next lines supply the execution

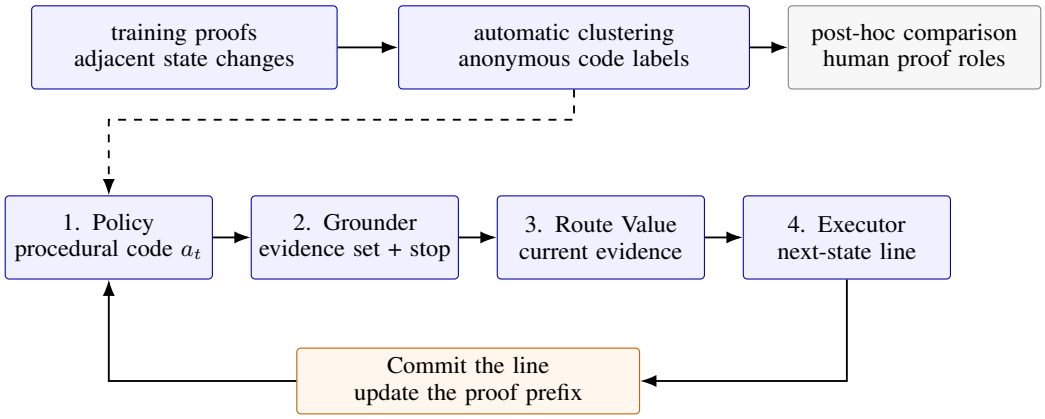


Figure 1: Learning and using the Qwen codes. Training proof transitions supply anonymous cluster labels for Policy; the downstream modules train on those IDs with evidence and next-line targets. Human proof roles are compared with the partition after induction. At inference, Policy selects a code, Grounder proposes evidence, Route Value selects an item, and Executor writes the next line.

targets. We use human proof-role names afterward to examine the partition. At inference, Policy predicts the code from x and w_t.

2.1.2 Learning what to do with a code

We connect a discovered category to execution by using the same ID in the training examples for all four modules. We implement them as separate LoRA adapters on Qwen. Policy receives the problem and proof prefix and learns to predict a_t^*. Grounder receives these inputs together with the code and learns which source sentences will be needed. Route Value scores a proposed sentence using the prefix and code. Executor receives the problem, code, and selected evidence, with the demonstrated next proof line as its target.

We construct Grounder's target from the source sentences used in the remaining demonstrated proof. They are extracted from the training traces, serialized in their original source order, and followed by `<STOP>`; a state requiring no source evidence uses `<NONE>`. For Route Value, the evidence used by the current transition is positive, while other proposals that do not advance that prefix are negative. We train it to score the needed item above those alternatives.

The adapters learn an ID's use from its evidence and next-line examples, without a written operation definition. Adapter sizes and optimization settings appear in Appendix C.

2.1.3 Selecting evidence and committing the next step

Grounder can propose evidence before the proof is ready to use it. Route Value selects what is needed next, regardless of source order. The learned `<STOP>` lets the set grow or shrink with the context.

Illustrative example. Suppose the problem contains "All warm things are round," "Nia is red," and "All red things are warm," in that order, and asks whether Nia is round. The proof has already established that Nia is red. Grounder's target includes both remaining rules in source order, ending in `<STOP>`. Route Value must select the second item, "All red things are warm." Executor writes that source-rule citation. A subsequent update derives "Nia is warm," after which the other rule can be used. The citation and its consequence are separate proof lines, each produced in its own pass through the loop.

Writing π, G, R, and E for Policy, Grounder, Route Value, and Executor, respectively, the inference update for a step with source evidence is

$$\begin{aligned} a_t &= \pi(x, w_t), & \mathcal{E}_t &= \mathrm{dedup}\big(G(x, w_t, a_t)\big), \\ e_t &= \arg\max_{e \in \mathcal{E}_t} R(w_t, a_t, e), & u_t &= E(x, a_t, e_t), \qquad w_{t+1} = w_t \parallel u_t. \end{aligned} \tag{2}$$

One deterministic Grounder call produces $\mathcal{E}_t$. We remove duplicate outputs, and Route Value scores each member using its YES minus NO logit. The selected evidence is passed to Executor as text. We append its output before making the next choice, then run the modules again on the updated prefix. After "Nia is warm" is added, Route Value can select the other rule from Grounder's new proposal set. Generation ends at an answer or at the gold proof length plus two updates; the official parser scores the saved proof afterward.

2.2 Tape State Engine

We also build a State Engine for tape programs, where an update has a directly observable effect on a tape, program counter, and head position. We learn an operator reader, a workspace encoder, and a neural executor. The reader identifies an operation from its demonstrations; the encoder supplies the information needed to apply it to the query workspace.

2.2.1 Discovering tape operations

For an operator instance o, let $D_o = \{(p_j, q_{o,j})\}_{j=1}^{16}$ contain its before–after demonstrations on shared probe workspaces p_j. We form the successor signature $\sigma(o) = (q_{o,1}, \ldots, q_{o,16})$ and group instances by

$$o \sim o' \quad \Longleftrightarrow \quad \sigma(o) = \sigma(o').$$

Each observed group supplies an anonymous target for the reader; the number of groups supplies the vocabulary size. Visible instruction symbols are reassigned across environments, so the demonstrations establish their meaning. A single probe can leave two operations indistinguishable. The signature collects their effects across the probe bank.

2.2.2 Encoding and updating the workspace

For the active instruction $o(w_t)$, the learned update is

$$\begin{aligned} a_t &= \mathrm{Reader}(D_{o(w_t)}), \qquad z_t = \mathrm{Encoder}(w_t), \\ d_t &= \mathrm{Executor}(a_t, z_t), \quad w_{t+1} = \mathrm{write}(w_t, d_t). \end{aligned} \tag{3}$$

The encoder produces four categorical variables through straight-through assignments to learned codebooks. These variables are learned from successor supervision. The executor receives the operator code and encoded workspace and predicts four edit fields: position, write value, counter displacement, and head displacement. The writer applies the fields by assignment and cyclic movement, preserving the program and other read-only fields.

For a concrete example, write a tape workspace as $w = (P, b, c, h)$, where P is the program, b the eight-bit tape, c the program counter, and h the head position. Flipping the current bit requires the tape edit

$$b' = b[h \leftarrow 1 - b_h].$$

The executor must predict the write position h and new bit $1 - b_h$, together with the counter and head displacements, from its two learned inputs. A head movement, in contrast, does not need the tape content. The encoder learns which workspace information to retain through these different update requirements. Its categorical output carries all workspace information that reaches the executor; the writer then applies the predicted edit, including an incorrect one. It does not choose the operation.

After the edit, the next call encodes the resulting workspace. Appendix C gives the architectures and optimization settings.

2.3 Matched Token and State interfaces

In the matched comparison, we keep the encoder–decoder Transformer architecture the same while dividing prediction and execution between the model and executor. Token writes the complete next workspace; State predicts an action label and a fixed executor applies it. For current workspace w_t, with the program or rule set included in the input, we write

$$\hat{w}_{t+1}^{\mathrm{Token}} = D_\theta(w_t), \qquad \hat{a}_t = \pi_\phi(w_t), \qquad \hat{w}_{t+1}^{\mathrm{State}} = F_{\hat{a}_t}(w_t). \tag{4}$$

Here D_θ generates and parses a full successor, π_ϕ selects the action, and F_a applies its specified effect. On tape, State uses eight context-conditioned action labels. In controlled English, it selects an index into the shuffled rules presented with the workspace. The applicator carries out that rule's replacements even when its conditions are false. Delegating these field updates reduces the model's output choices: in English, Token predicts four fields with six values each, giving $6^4 = 1296$ possible field assignments, while State selects one of three or four presented rules. The executor handles the coupled changes, leaving fewer degrees of freedom in the neural prediction.

We train the models on the same reference workspaces. Token learns the successor tokens under teacher forcing; State's target is the action label. Paired runs share initialization, input order, optimizer settings, and training budget. At one-step evaluation both receive a reference input; during rollout each receives the workspace produced by its previous update. Predictions are parsed without repair. Appendix C.1 gives the tape architecture and training settings, and Appendix C.4 gives the English configuration.

3 Experiments

We test whether the codes identify usable reasoning operations, control their execution, and remain useful as a procedure is extended or recombined. The tasks constrain individual updates, so we can follow both the logical choices and their effects. Full protocols and training settings appear in Appendices A and C.

3.1 Qwen on PrOntoQA

We generate deductive problems with the original PrOntoQA generator (Saparov & He, 2023). Each contains facts, rules, and a Boolean query. Its reference proof alternates source citations with derived consequences, then answers the query. We use 600 training, 180 validation, and 180 independent test problems, with 60 test problems at each depth from two through four hops.

Both systems start from Qwen3-1.7B-4bit. Token CoT learns to generate the whole proof with a rank-8 LoRA adapter; our engine trains the four adapters described in Section 2.1 and makes additional inference calls to construct the proof line by line. Answer accuracy checks the Boolean response, and exact-process accuracy requires the reference chain. The official semantic evaluator also accepts alternative valid proofs: step legality checks each inference against the actual preceding proof, and every-step legality requires all of them to pass. Legal completion further requires the target conclusion and correct answer. Appendix A.5 gives the stopping and scoring protocol, and Appendix C.5 gives adapter sizes and optimization settings.

3.1.1 Discovered classes correspond to human proof roles

The selection procedure returns five occupied classes, with silhouette 0.3614. To understand them, we compare the resulting partition with the existing human annotations of proof roles, such as citing a source or deriving a consequence. The annotations are introduced only for this comparison, after induction; the modules train on anonymous IDs.

The learned groups correspond closely to these proof roles: adjusted Rand index is 0.846 (Table 1). This score compares which transitions are grouped together, regardless of the names assigned to the clusters.

3.1.2 Code identity controls proof generation

Policy predicts all 1,440 held-out reference-transition codes correctly with the five-code vocabulary. With eight codes, its accuracy is 85.63%. To test the use of the selected code, we freeze the trained modules and change only the ID sent to Grounder, Route Value, and Executor. The permutation is fixed and maps every ID to a different one.

Permuting the identities eliminates every correct answer and exact process on the 180 test problems. Fixing the code to `S0` also gives no correct answers, although 40.28% of individual steps remain legal. Omitting the code line leaves 6.11% answer accuracy and no exact processes. Appendix B.12 gives the paired significance tests.

Table 1: Discovery and use of Qwen's anonymous codes. Top: the selected partition, post-hoc agreement with proof roles, and Policy accuracy on 1,440 held-out reference transitions. Bottom: frozen-engine interventions on all 180 test problems (percent); only the code sent to Grounder, Route Value, and Executor changes.

Discovery and code prediction	Result
Automatically selected occupied codes	5
Agreement with proof roles (adjusted Rand index)	0.846
Policy accuracy, five-code vocabulary (%)	100.00
Policy accuracy, eight-code vocabulary (%)	85.63

Code channel	Answer	Exact proc.	Step legal	Every step	No answer ↓
Learned code	81.67	57.78	81.80	63.33	12.22
Removed	6.11	0.00	9.70	0.00	91.11
Fixed to `S0`	0.00	0.00	40.28	0.00	100.00
Permuted identities	0.00	0.00	20.42	0.00	99.44

3.1.3 Using evidence from the current problem

The code identifies a kind of proof step; the engine must still find evidence for this particular problem. We audit that choice from unseen reference proof prefixes, where the next line and required evidence are known. Taking Grounder's first item would supply the evidence needed next in 44.17% of the 240 audited proof prefixes. Route Value selects the needed item in 92.92%. Grounder returns 2.91 items on average and includes the required evidence in 93.33% of its sets. Every set ends with `<STOP>`.

As the proof advances, Grounder removes the evidence just used in all 120 prefix pairs we checked and retains 81.06% of the evidence still needed later. Replacing the selected evidence changes Executor's output in 99.17% of 120 transitions, with only 0.83% still producing the correct next line.

Table 2: How Qwen's learned interface controls the next step (percent). Set and route audits use 240 unseen gold prefixes; prefix updates use 120 pairs, and evidence substitutions use 120 transitions.

Stage	Measurement	Result
Grounder	Current evidence included	93.33
	Complete target set matched	66.25
Route Value	First generated item is current	44.17
	Selected item is current	92.92
	Correct selection when current item is present	99.55
Prefix update	Consumed evidence removed	100.00
	Still-needed evidence retained	81.06
Executor	Output changed by evidence replacement	99.17
	Wrong evidence still yields the correct next state	0.83

3.1.4 Complete proof generation

After the reference-prefix audits, we let both systems generate complete proofs from the problem. On the 180 test problems, the full engine reaches 81.67% answer accuracy and 57.78% exact-process accuracy, compared with 48.33% and 23.33% for Token CoT. Paired McNemar tests give $p = 4.45 \times 10^{-10}$ for answers and 9.94×10^{-11} for processes.

State reaches 81.67% answer accuracy at each depth (Table 3). The semantic evaluator marks 81.80% of State's steps legal, versus 44.63% for Token, and accepts 57.78% versus 25.56% of complete proofs. Appendix B.12 gives the full process diagnostics.

Table 3: Qwen3-1.7B on PrOntoQA: answer and exact-process accuracy (percent) on all 180 problems and on 60 problems at each depth.

	Answer		Exact process	
Test split	Token	State	Token	State
All 180	48.33	81.67	23.33	57.78
2 hops	60.00	81.67	38.33	71.67
3 hops	46.67	81.67	21.67	63.33
4 hops	38.33	81.67	10.00	38.33

3.2 Learned tape operations

The Qwen experiments follow generated proof lines. Tape programs let us inspect the execution more directly: we can see where an operation writes, what it writes, and how it moves the pointers. We also want to reuse what the engine has learned, so we compose its operations in new sequences and transfer the workspace encoder to a new operation family.

For this experiment, we randomly assign five visible symbols to five hidden operations: flipping a bit, copying it to the next cell, moving the head left or right, and conditionally advancing the program counter. The workspace contains an eight-cell program, an eight-bit tape, a counter, and a head position. The same symbol can mean different things in different environments. Its 16 before–after demonstrations on shared probe workspaces tell the reader which operation to apply to a new query workspace.

Each run uses 80 training and 20 unseen symbol assignments, with 4,800 training and 1,600 held-out query transitions. We run three independent splits and initializations, giving 4,800 held-out transitions in total. Success requires the entire predicted successor to match the target. The reader, encoder, and executor are those in Section 2.2; their training and intervention protocols appear in Appendix C.3.

3.2.1 The discovered codes determine which edit is executed

Only 11 of these successors are wrong, giving $99.77 \pm 0.13\%$ exact accuracy. In each error, the edit position is wrong; the write value and both pointer displacements are right. The reader itself identifies every held-out operation correctly. All three splits recover five anonymous classes, with 100% purity against the hidden operation identities.

Changing a procedural code after training changes every output, and 93.94% match the replacement operation. Fixing or shuffling the execution state leaves left and right movement correct but reduces flip accuracy to about 6% (Table 4). Flip needs the bit under the head to work out its update; the two head movements do not.

Table 4: Discovery and causal control in the tape engine (percent). Full-engine results are mean $\pm$ sample s.d. over three runs; operator and state interventions are separate audits with their own intact-code baselines.

Measurement	Correct	Fixed/shuffled	Result
Operator discovery	100.00 ± 0.00	–	100.00 purity
Complete successor	99.77 ± 0.13	–	4,800 tests
Operator intervention	93.88	20.00/22.25	93.94 substituted match
State, all operators	99.94	61.19/56.31	–
State, flip only	99.69	5.63/5.94	L/R remain 100

3.2.2 Learned operations can be composed and their states transferred

Although training uses individual transitions, we can run the learned operations in new sequences by applying each predicted edit before encoding the next workspace. Across 2,500 test cases, exact dynamic-state accuracy is 99.56% after eight operations; all 25 ordered operation pairs score between 99% and 100%. The state-capacity and sequence results appear in Appendices B.3 and B.4.

We also reuse the workspace encoder with a new five-operation family. With 320 target examples, freezing it gives 51.25% exact accuracy, compared with 36.75% from scratch. With 3,000 examples, fine-tuning reaches 94.44%, while freezing levels off at 57.69%. Appendix B.6 gives all three data sizes and the transfer protocol. Appendix B.5 reports what happens when demonstrations are missing, including runs trained with missing probes.

3.3 Matched Token and State comparisons

The learned engines discover operations and learn to execute them. We next focus on why separating the two jobs helps prediction, using Token and State models with the same architecture and training conditions (Section 2.3). In State, a fixed executor handles the concrete updates, so the network only has to choose an action. We use this division to reduce the network's choices and degrees of freedom, making prediction simpler and more stable over successive steps.

3.3.1 State preserves more correct trajectories over 30 steps

We test extrapolation in computation length on an eight-cell program and eight-bit tape (Appendix A.2). Training trajectories have 3–7 steps and supply 16,000 balanced transition rows. We evaluate the same 300 unseen machines at every horizon from 3 through 30, feeding each predicted successor into the next update. For reference workspaces w_t and generated workspaces $\hat{w}_t$, the two trajectory scores at horizon H are

$$Y_{\text{final}} = \mathbf{1}[\hat{w}_H = w_H], \qquad Y_{\text{process}} = \prod_{t=1}^{H} \mathbf{1}[\hat{w}_t = w_t]. \tag{5}$$

Their test-set means measure final-workspace and exact-process accuracy. A trajectory that leaves the reference path and later returns can pass the first measure but fails the second. The local-update score instead starts from a reference workspace and checks the executed action: Token's successor is parsed to identify its update, and State's selected action is executed.

At seven updates, the longest training horizon, 220 of the 300 Token trajectories and 291 State trajectories are exact. At 30 updates, 148 and 279 remain exact (Table 5). Of each model's trajectories still correct at step 7, Token therefore preserves 67.27% through the next 23 updates, and State preserves 95.88%. Appendix B derives these rates from the complete-process counts.

We also check whether each update obeys the machine's transition function F from the model's actual predecessor: $\hat{w}_{t+1} = F(\hat{w}_t)$. The audit finds 350 violations for Token and 33 for State across 300 rollouts of 30 steps (9,000 updates). Most Token violations occur at conditional jumps. An earlier wrong jump can lead to a different workspace from which later execution is valid; this audit counts those later steps as legal even though they disagree with the reference. Appendix B.10 separates these cases, and Appendix B.7 gives the full horizon curves.

Table 5: Matched tape interfaces on 300 test machines. Scores are percentages except the invalid-update count. One-step code accuracy evaluates the executed update from a reference input.

Measurement	Token	State
One-step code accuracy, pooled	96.23	99.64
30-step final workspace	50.33	94.33
30-step exact process	49.33	93.00
Invalid updates / 9,000	350	33

3.3.2 Choosing a rule keeps its field updates together

Even when the selected instruction is wrong, we want its required field changes to stay together. To look at this, we construct a controlled-English task in which shuffled rules act on four named fields, each with six values. A rule tests three or four fields and replaces one or two. For example, "if badge is amber and seal is open and route is north, then set badge to blue and route to east" changes two fields together and leaves the others unchanged. Each program has two or three reference updates and one distractor, with exactly one licensed rule at every reference workspace. State predicts a rule

index; its applicator carries out that rule's replacements even when its conditions are false. Token writes the four successor fields. Both models train on 14,704 transitions. Three paired seeds use the same 600 test programs and 1,495 reference-input transitions; full details are in Appendix A.4.

We score the final workspace and complete trajectory as in Eq. 5, then examine the wrong one-step predictions. An error is action-contained if it applies some presented rule's replacements together, including a rule whose conditions were false. The one-variable oracle repair replaces State's rule index, or one of Token's four predicted fields, with the gold value and checks the whole successor. This is an analysis of saved errors, described in Appendix B.9.

State reaches $96.00 \pm 5.77\%$ exact final-workspace accuracy and $94.72 \pm 7.56\%$ exact-process accuracy (Table 6). Across all three runs, State makes 104 wrong predictions and Token makes 2,552. Every parseable State error applies one alternative rule in full.

Table 6: Matched controlled-English interfaces. Results are mean $\pm$ sample s.d. over three paired seeds, in percent except field distance. Action containment conditions on parseable errors.

Measurement	Token	State
One-step exact	43.10 ± 2.74	97.68 ± 3.38
Final workspace	34.22 ± 3.83	96.00 ± 5.77
Exact process	17.67 ± 1.89	94.72 ± 7.56
Action-contained errors	27.57 ± 3.16	100.00 ± 0.00
One-variable oracle repair	65.69 ± 1.66	100.00 ± 0.00
Wrong fields per error	1.398 ± 0.016	2.378 ± 0.399

A wrong rule can change several fields at once, explaining why State's errors differ from gold in more fields despite being less frequent. Additional answer-readout probes are reported in Appendix B.11.

4 Discussion and conclusion

We built the State Engine so that a model chooses an operation before updating its workspace. In Qwen, the learned categories correspond to human proof roles, and changing their codes changes the generated proofs. The categories and execution targets come from the transition examples themselves. A designer need not name and implement every operation in advance.

With execution handled separately, the network has less to predict. It chooses an action label, and the executor carries out that action's field changes together. Under the same architecture, State preserves more correct trajectories beyond the training horizons. The learned tape operations also compose in new sequences.

Route Value cannot recover evidence that Grounder omitted, and the loop has no backtracking step. A failed update should be able to request more evidence or revise an earlier choice. The learned state representation also reflects its training operation family: a frozen encoder helps with few target examples but levels off as more data become available, whereas fine-tuning continues to improve. Extending the operation vocabulary may require adapting what the workspace representation retains.

We want to extend this work to mathematical derivations and physical models. In mathematics, new results build on established deductions; in physics, a derivation must keep track of the assumptions under which each law applies. An LLM helping with scientific work needs to carry these rules through a longer argument without changing how it applies them. For these tasks, we want to learn operations whose selection is explicit and whose execution stays consistent as the argument grows or is applied to a new problem.

AI USE STATEMENT

Generative AI tools provided limited assistance with generating synthetic tape-machine data under author-defined rules and with line-level language and LaTeX editing of author-written drafts. The author developed the research question and State Engine framework, designed and implemented the experiments and evaluation criteria, performed the substantive manuscript revisions, interpreted the results, and determined the paper's claims. All AI-assisted work was reviewed and revised by the author, who takes responsibility for the final paper, code, and artifacts.

REPRODUCIBILITY STATEMENT

Appendices A–C describe data generation and splits, architectures, optimization, metrics, interventions, and probe protocols. The complete result tables and trajectory audits appear in Appendix B; the formal arguments and their assumptions appear in Appendix D. Each experiment reports its saved seeds and sample sizes. Supplementary material will include an anonymized archive of training and evaluation scripts, generated datasets, and saved aggregate outputs.

REFERENCES

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Kyle Cox, Darius Kianersi, and Adrià Garriga-Alonso. Post-hoc reasoning in chain of thought: Decoding and steering pre-committed answers. *arXiv preprint arXiv:2603.01437*, 2026. doi: 10.48550/arXiv.2603.01437.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018. doi: 10.48550/arXiv.1803.10122.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. In *Second Conference on Language Modeling*, 2025.

Jinghan Jia, Joe Benton, and Eric Easley. Faithfulness as information flow: Evaluating and training faithful chain-of-thought reasoning. *arXiv preprint arXiv:2605.24286*, 2026. doi: 10.48550/arXiv.2605.24286.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pp. 207–220. ACM, 2009.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pp. 22199–22213, 2022. doi: 10.52202/068431-1613.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, Karina Nguyen, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Robin Larson, Sam McCandlish, Sandipan Kundu, Saurav Kadavath, Shannon Yang, Thomas Henighan, Timothy Maxwell, Timothy Telleen-Lawton, Tristan Hume, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*, 2023. doi: 10.48550/arXiv.2307.13702.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.248.

Keshav Ramji, Tahira Naseem, and Ramón Fernandez Astudillo. Thinking without words: Efficient latent reasoning with abstract chain-of-thought. *arXiv preprint arXiv:2604.22709*, 2026. doi: 10.48550/arXiv.2604.22709.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023. doi: 10.52202/075280-2997.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems*, volume 36, pp. 74952–74965, 2023. doi: 10.52202/075280-3275.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022. doi: 10.48550/arXiv.2211.14275.

Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021. doi: 10.1016/j.patter.2021.100273.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022. doi: 10.52202/068431-1800.

Junda Wu, Yuxin Xiong, Xintong Li, Sheldon Yu, Zhengmian Hu, Tong Yu, Rui Wang, Xiang Chen, Jingbo Shang, and Julian McAuley. CTRLS: Chain-of-thought reasoning via latent state transition. In *Proceedings of the 29th International Conference on Artificial Intelligence and Statistics*, volume 300 of *Proceedings of Machine Learning Research*, pp. 2728–2736. PMLR, 2026.

Lei Xu, Pierre Beckmann, Marco Valentino, and André Freitas. Adaptive LLM-symbolic reasoning via dynamic logical solver composition. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1187–1208, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.54.

Shenao Zhang, Donghan Yu, Yihao Feng, Bowen Jin, Zhaoran Wang, John Peebles, and Zirui Wang. Learning to reason as action abstractions with scalable mid-training RL. In *The Fourteenth International Conference on Learning Representations*, 2026.

A Experimental setup

A.1 The control loop at a glance

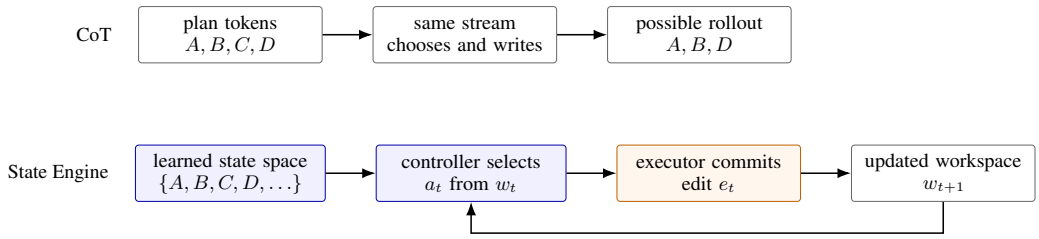


Figure 2: Conceptual comparison of a plan carried by token generation and a State Engine that commits each edit before making its next choice. The main text shows the two implemented engines.

A.2 A deterministic tape machine

We construct the tape task around an eight-cell program acting on an eight-bit tape. The workspace also records a program counter and a head position. Each program cell independently draws one of five instructions: `F` flips the bit under the head, `C` copies that bit into the next cell, `L` and `R` move the head, and `J` moves the program counter forward by one or two positions according to the current bit. The initial tape, counter, and head are sampled independently. Together they determine a unique next workspace.

We run every sampled trajectory for L updates and score the complete final workspace. Neither model receives L, and the machine has no halt instruction. Longer horizons repeat the same transition rule without increasing the input size.

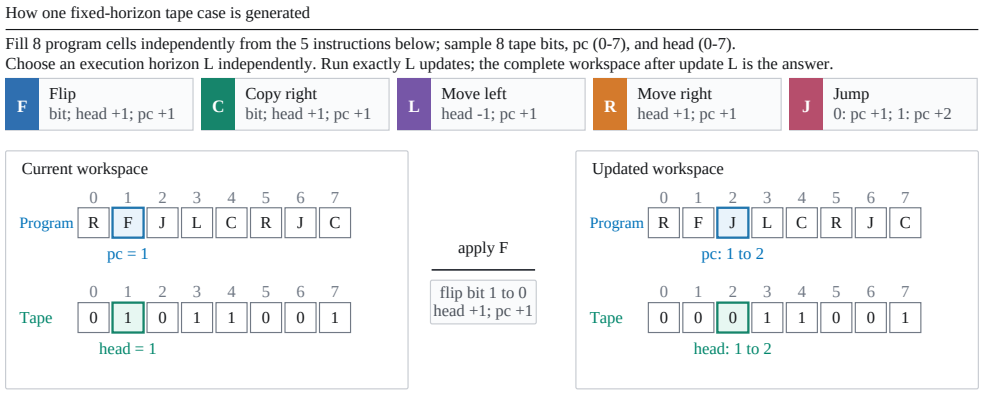


Figure 3: One deterministic tape transition. The program, tape, counter, head, and evaluation horizon are sampled independently. Evaluation compares the full workspace after the requested number of updates.

A.3 Paired data and trajectory metrics

We generate 300 training cases at each horizon from 3 through 7 for the matched study. Resampling their transitions with replacement balances the eight context-conditioned update types and produces 16,000 training rows. Token and State receive identical current workspaces in identical shuffled order and undergo the same number of optimizer updates. Testing begins from a separate set of 300 machines. Each machine is evaluated at every integer horizon from 3 through 30, giving paired measurements of the same underlying cases across horizons.

One-step Code Accuracy starts from a reference workspace and scores the executed update: we parse Token's successor to identify its update, and execute State's predicted code. For rollouts, **Final Workspace Accuracy** checks the complete endpoint after the prescribed number of steps; **Exact Process Accuracy** requires agreement at every earlier position as well. **All Steps Legal** instead evaluates each transition from the model's own preceding workspace.

A model can correctly advance its altered workspace and still disagree with the reference, or jump to a reference state that is unreachable from its actual predecessor. From the correct start, a fully legal deterministic rollout must follow the reference.

We classify updates by their starting position and their effect. An *unlicensed departure* is an invalid move away from the reference. A *lawful off-reference continuation* correctly advances a workspace produced by an earlier error. Return to the reference is a *dynamics-mediated re-entry* when the move is legal and an *unlicensed snap-back* when it is invalid. Malformed and out-of-bounds predictions are violations. This accounting assigns one violation to an invalid departure followed by 20 lawful off-reference steps, even when all 21 positions disagree with the reference.

A.4 Rules whose order matters in controlled English

We construct the controlled-English task by replacing tape instructions with conditional rules. Four named fields—badge, seal, route, and tier—each hold one of six word-valued states. A shuffled program contains rules such as "`q2 if badge is amber and seal is open and route is north, then set badge to blue and route to east`." Each rule checks three or four fields and replaces one or two. Every reference workspace licenses exactly one rule.

A training program contains two or three successive updates and one distractor. We present its unchanged rule set alongside every workspace in the reference trajectory. As the workspace changes, the licensed rule changes with it. Independent shuffling across cases prevents a rule's position from becoming a stable operation label.

The applicator carries out the replacements specified by any well-formed State prediction, even when the selected rule's conditions are false. Each prompt also contains a balanced binary query about one terminal field. Our primary endpoint checks all four final fields; the binary query is a secondary readout because it can survive an error elsewhere in the workspace. During rollout, updates are classified as licensed, unlicensed but action-conforming, malformed, or off-action.

To study the structure of failure, we separately audit predictions already known to be wrong. For a current workspace w, define $\mathcal{R}(w)$ as the set of successors obtained by applying each presented rule's replacements without checking its conditions. A wrong successor is *action-contained* if it belongs to $\mathcal{R}(w)$; its *action distance* is the minimum four-field Hamming distance to that set. A one-variable oracle repair then replaces either State's predicted rule index or one of Token's four predicted fields with the corresponding gold value and tests whether the complete successor becomes correct. State has three or four rule choices, and each Token field has six possible values.

A.5 PrOntoQA with Qwen3-1.7B

We use synthetic problems from the original PrOntoQA generator and its official semantic parser. The data comprise 600 training, 180 validation, and 180 test problems, including 60 test problems at each depth from two through four hops. An independent test seed is reserved throughout training and checkpoint selection. Proofs alternate a cited fact or rule with the consequence derived from it, followed by a Boolean answer.

We initialize both methods from Qwen3-1.7B-4bit. For Token CoT, we train one rank-8 LoRA adapter with 2.49M parameters to generate the complete proof under teacher forcing. The State Engine builds the proof a line at a time. Policy first predicts an anonymous procedural code; Grounder uses that code, the raw problem, query, and proof so far to generate an evidence set ending in `<STOP>`. Route Value scores its members against the prefix. Executor then writes the next proof state from the selected evidence, and the new line is added to the history.

We induce the codebook from training transitions. Adjacent proof states yield Qwen hidden-state differences, which are normalized and reduced to 48 principal components. MiniBatch k-means is fitted for $K \in \{2, \ldots, 8\}$, and silhouette score minus $0.005K$ selects five occupied codes with

silhouette 0.3614. Proof-role names play no role in clustering or code supervision. Their post-hoc agreement with the induced classes has adjusted Rand index 0.846. Grounder targets contain the source sentences used by the remaining demonstrated proof, serialized in source order and terminated by <STOP>; states requiring no source evidence use <NONE>. These targets are extracted from training traces without a logic parser. For Route Value, the current transition's observed evidence is positive, and other proposals that fail to advance that prefix are negative.

At inference, one deterministic Grounder call produces the candidate set and its stop token. We remove duplicate outputs; any omitted source sentence remains absent. Route Value chooses from the resulting set, and Executor uses the selected text to generate the next state. Evidence is passed as text, without rule numbers or source indices. Grounder determines the number and contents of the proposals, with neither an enumerated source list nor a symbolic permission signal supplied to the loop. Generation ends at an answer or at the gold proof length plus two updates. The official parser scores saved trajectories afterward.

The official process measures are primary. One-step legality counts parsed lines justified by a single inference; "Every step legal" requires this throughout the generated proof. "Legal complete" further requires the target conclusion and correct answer. We record parser failures and trajectories containing illegal lines separately. Exact reference-chain and positional scores provide a stricter secondary view, since legal alternative proofs can fail those comparisons. Paired McNemar tests compare outcomes on the same test problems.

B Additional experimental results

B.1 Recovering operations from transition traces

Grouping demonstrations by their successor signatures yields five classes in every split. The reader identifies held-out classes with 100% accuracy in all three runs. Comparing these classes with hidden operation identities after training gives 100% purity and NMI 1.0. We separately train small diagnostic executors on this partition; they score 82.88%, 87.25%, and 92.81%.

Connecting each reader to a newly trained categorical encoder and executor gives the complete engines in Table 7. Each run uses an independent rule split and initialization and 1,600 held-out transitions. Across the three runs, all 11 errors select the wrong edit position. Every write value, program-counter displacement, and head displacement is correct.

Table 7: End-to-end performance with learned procedural codes and execution states on unseen symbol assignments. Training uses anonymous transition classes and successor supervision, with no human operator or state labels. The summary reports mean and sample standard deviation across three independent runs.

Run	Operator class	Operator purity	Exact successor
1	100.00	100.00	99.8125
2	100.00	100.00	99.6250
3	100.00	100.00	99.8750
Mean $\pm$ s.d.	100.00 ± 0.00	100.00 ± 0.00	99.7708 ± 0.1301

B.2 Interventions on operation codes and execution states

For Table 8, we hold the trained engine fixed and replace one input to its executor. Under an operator-code substitution, every output changes and 93.94% match the newly requested operation.

Left and right movement remain at 100% after execution states are fixed or shuffled. These operations need no tape content. Flip needs the bit under the head; its accuracy falls from 99.69% to 5.63% and 5.94%.

Table 8: Effects of changing procedural and execution-state codes after training (percent exact). The substituted-code condition is evaluated against the newly requested operation; the state audit also isolates flip, which requires tape information.

Interface	Score	Correct	Fixed	Shuffled
Operator	Original successor	93.88	20.00	22.25
Operator	Substituted action	93.94 under substitution		
State	All operations	99.94	61.19	56.31
State	Flip only	99.69	5.63	5.94

B.3 Execution-state capacity and irrelevant variation

The second eight-valued variable accounts for most of the improvement in Table 9.

Table 9: Accuracy as the categorical execution bottleneck expands. The executor receives the learned state and operation code; the raw workspace remains inaccessible.

Discrete state	Nominal states	Exact successor (%)
1×8	8	77.4375
2×8	64	98.5625
4×8	4096	99.9375

Changing the program counter in 1,600 paired workspaces leaves every target edit unchanged. With two state slots, 93.00% of pairs retain the complete code tuple and 97.56% retain every predicted output field. Exact accuracy changes from 98.56% to 98.50%.

B.4 Composing primitive updates

We feed each predicted successor into the next operation of an unseen ordered composition. The engine was trained only on single transitions. Across 2,500 cases, exact dynamic-state accuracy is 100% after one operation, 99.88% after two, 99.80% after four, and 99.56% after eight. All 25 ordered pairs score between 99% and 100%.

B.5 Training with incomplete probe sets

The probe workspaces must distinguish the hidden operations. Keeping only the first eight probes merges two classes: purity falls to 80%, and exact successor accuracy of that chain falls from 99.8125% to 88.25%.

Retaining eight probes at random gives mean class accuracies of 86.50%, 85.34%, and 96.56% for three readers trained on complete 16-probe sets. Two pilot readers trained on random subsets of 8–16 probes reach 98.00% and 99.40%, using the same architecture. Both retain 100% with all 16 probes. At 12 probes, the standard readers average 95.93% across three seeds and the dropout readers average 99.90% across two (Figure 4).

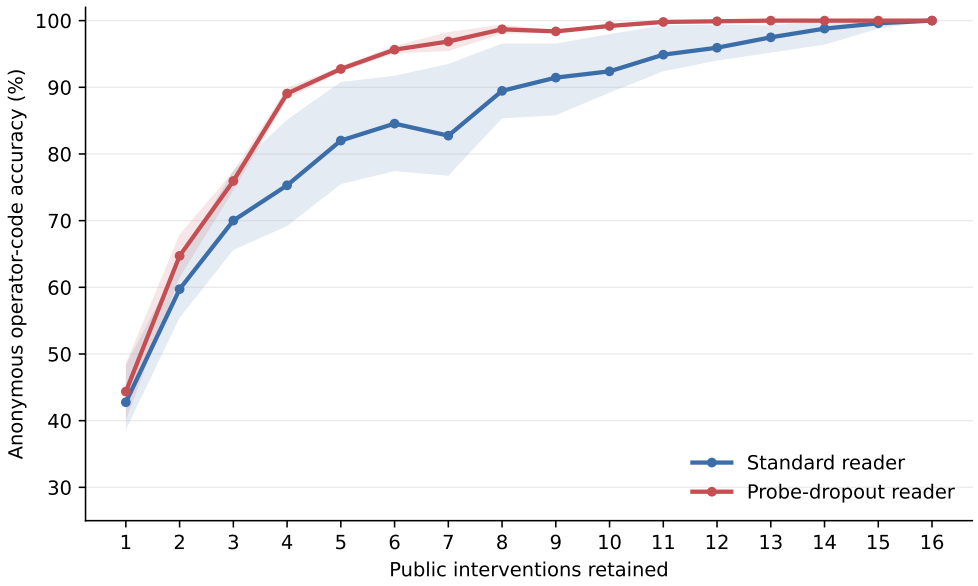


Figure 4: Identifying anonymous operations from progressively smaller probe sets. Curves show the mean across independently trained readers and bands show the full range. Probe-dropout varies the demonstrations during training while retaining the architecture and targets.

B.6 Reusing an execution state for new operations

The target family replaces flip, copy-next, left, right, and conditional jump with set-one-and-move-right, set-zero-and-move-left, copy-to-previous, invert-next, and reversed conditional jump. The workspace layout stays the same, and every target system receives the target operation index.

All conditions use four slots, the same target examples and optimizer, and 300 epochs. Training from scratch initializes the entire system anew. Transfer retains the source encoder and codebooks and initializes new target-operation embeddings and an executor. We either freeze the source components or fine-tune the full system. Table 10 uses three independent target-data and initialization seeds.

Table 10: Exact successor accuracy after changing the operator family (percent, mean $\pm$ sample standard deviation over three target seeds). Architecture, target supervision, and training budget are shared across conditions.

Target rows	Scratch	State frozen	State fine-tuned
320	36.75 ± 2.54	51.25 ± 1.39	43.03 ± 3.73
1,000	36.89 ± 1.75	54.08 ± 1.06	74.21 ± 7.45
3,000	87.43 ± 3.14	57.69 ± 0.12	94.44 ± 0.74

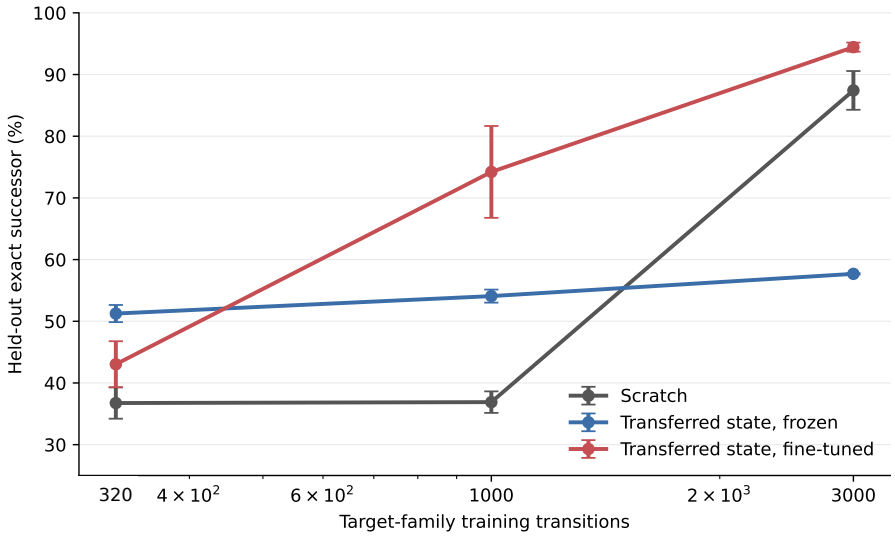


Figure 5: Learning a new operator family from increasing amounts of target data. Frozen source states give the largest gain at 320 rows; with more data, adapting the transferred state gives the best accuracy. Error bars show sample standard deviations across three seeds.

B.7 Matched-capacity prediction and rollout

Across 9,000 reference-input positions, local-update accuracy is 99.64% for State and 96.23% for Token. Table 11 reports local and rollout scores at each horizon. By step 30, the exact-process gap has grown to 43.67 points.

The same 300 machines are followed through all horizons in Figure 6. These matched models use the supplied State executor described in Appendix C; the fully learned engine is evaluated separately in Section B.1.

Let S_h be the fraction of these machines whose first h updates all match the reference. Exact trajectories at a later horizon are a subset of those at an earlier one, so S_b/S_a is the fraction that remains exact from a through b, conditional on an exact prefix at a. Table 11 gives 220 and 148 exact Token trajectories at steps 7 and 30, and 291 and 279 for State. Hence the retained fractions are $148/220 = 67.27\%$ and $279/291 = 95.88\%$, each using its own model's step-7 survivors.

These ratios are products of conditional step-success probabilities, as in Appendix D. The pooled one-step score averages reference-input predictions across all cases, including ones whose earlier rollout has failed. It does not supply those conditional probabilities. If the conditional probability of a correct update is the same p at every step, then $S_H = p^H$ and $S_b/S_a = p^{b-a}$.

Table 11: Local prediction and repeated execution with matched 527,064-parameter Transformers (percent). Both models share initialization, source workspaces, row order, and optimization. Training uses horizons 3–7, and evaluation reuses 300 machines at every horizon. One-step scores use reference inputs; final and exact-process scores use each model's own rollout.

Executed steps	Token one step	State one step	Token final	State final	Token exact	State exact
3	95.00	99.33	89.33	98.67	89.33	98.67
7	94.67	100.00	74.00	97.00	73.33	97.00
10	98.00	100.00	68.00	96.33	67.33	96.33
15	95.33	99.67	62.67	96.33	62.00	96.00
20	94.33	99.67	55.33	96.00	54.67	95.33
25	97.33	99.33	51.33	95.33	50.67	94.33
30	96.67	99.67	50.33	94.33	49.33	93.00

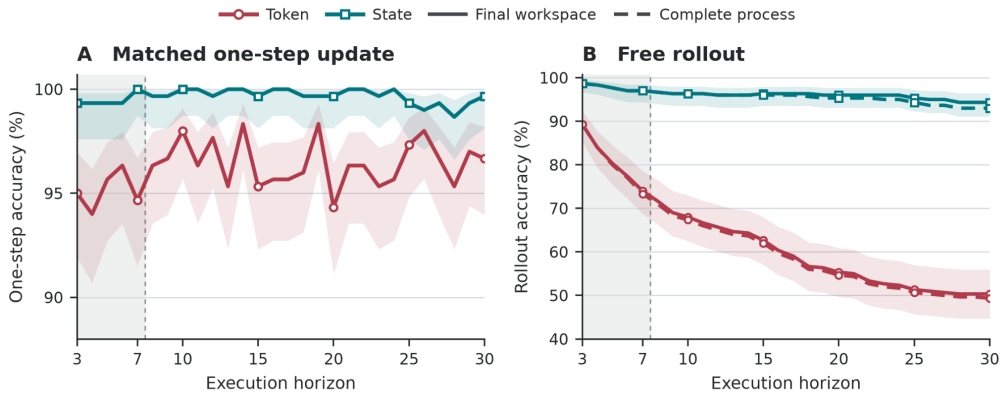


Figure 6: Matched Token and State accuracy across horizons 3–30. Every horizon uses the same 300 test machines, and both models share architecture, initialization, training rows, and optimization. In the rollout panel, solid curves score the endpoint and dashed curves require the complete trajectory. Bands are Wilson 95% intervals.

B.8 Controlled-English results

Table 12 uses 600 unseen programs and 1,495 reference-input transitions. The programs are fixed across three paired runs; initialization and minibatch order vary by seed.

Table 12: Controlled-English results across three paired model seeds (percent, mean ± sample standard deviation). The two systems share a 1,404,477-parameter Transformer, initialization within each seed, examples, minibatch order, and optimization budget. Exact agreement on all four terminal fields is the primary endpoint.

Measurement	Token	State
One-step update accuracy	43.10 ± 2.74	97.68 ± 3.38
Exact final workspace	34.22 ± 3.83	96.00 ± 5.77
Binary terminal query	84.61 ± 1.67	98.50 ± 1.88
Exact complete process	17.67 ± 1.89	94.72 ± 7.56
Action-conforming update	58.45 ± 1.19	99.93 ± 0.12
Licensed from actual workspace	30.38 ± 2.41	95.90 ± 5.66
Off-action update	41.12 ± 1.40	0.00 ± 0.00
Premature future state	9.54 ± 0.72	0.67 ± 0.93

Malformed outputs stay below 0.5% in every run. Most Token failures can therefore be parsed: 41.12% of updates combine fields in ways that no presented rule produces, and 9.54% reach a later reference state prematurely.

B.9 Error structure and single-variable repair

The saved models produce 2,552 Token errors and 104 State errors across three seeds and 4,485 reference-input transitions. Table 13 pools the counts and summarizes conditional rates and distances across seeds.

Table 13: Structure and repairability of errors on controlled-English reference inputs. Means and sample standard deviations use three model seeds. Action containment and distance are evaluated on parseable errors; oracle repair covers all errors. Smaller values are preferred in the two distance rows.

Measurement conditioned on error	**Token**	**State**
Pooled wrong predictions	2,552	104
Action-contained (%)	27.57 ± 3.16	100.00 ± 0.00
Distance to nearest action	0.763 ± 0.023	0.000 ± 0.000
Exact repair from one output variable (%)	65.69 ± 1.66	100.00 ± 0.00
Workspace fields differing from gold	1.398 ± 0.016	2.378 ± 0.399

Every parseable State error applies one alternative rule in full. One run also produces a malformed code. Replacing the code with its gold value repairs all 104 errors, including that case; the corresponding one-variable replacement repairs $65.69 \pm 1.66\%$ of Token errors.

A wrong rule can change several fields together. State errors consequently differ from gold in 2.378 ± 0.399 fields on average, more than Token's 1.398 ± 0.016, even though one code replacement reverses all those changes.

B.10 Errors and lawful continuations in tape rollouts

Table 14 checks each of the 9,000 rollout updates against its actual predecessor. Violations fall from 350 to 33, a 10.6-fold reduction. An update can be lawful even after the trajectory has left the reference.

Table 14: Executed-transition audit over $300 \times 30 = 9{,}000$ updates per model. Each successor is compared with both its actual predecessor and the reference trajectory. These are the matched models with a fixed State executor; Table 7 evaluates the independently trained learned engine.

Update category	Token	Matched State
On reference, correct update	5,794	8,670
On reference, unlicensed departure	149	21
Off reference, lawful continuation	2,854	293
Off reference, unlicensed continuation	183	12
Dynamics-mediated re-entry	2	4
Unlicensed snap-back	1	0
Out-of-bounds output	4	0
Unparseable output	13	0
State-relative violations	350 (3.89%)	33 (0.37%)
Trajectories with any violation	152/300	21/300

The conditional jump accounts for 335 of Token's 350 invalid updates; 311 take the branch for the opposite tape bit. Of its 152 first departures, 144 change only the program counter. State makes 18 wrong jump-branch choices and 15 wrong bit-conditioned flip or copy choices.

Token's 2,854 lawful off-reference updates and State's 293 carry an earlier error forward without introducing another violation.

Four State trajectories return to the reference after an error, all through lawful dynamics. Token has two lawful re-entries and one unlicensed snap-back. These seven cases account for the small gaps between endpoint and exact-process accuracy.

B.11 Distant-answer readouts during correct execution

We fit linear probes to predict the step-30 tape bits, counter, and head from earlier representations of the matched models. Appendix C gives their fitting procedure.

A field readout is confident when its maximum predicted probability exceeds τ. A confident prediction is correct (C) or wrong (W); the remaining predictions are unresolved (U).

Windows pair the same case, field, and starting step, with both models executing exact reference successors throughout. Both probes start in W for the persistence and W→U→C analyses, or in C for the corruption analysis, which records any later W. The five probe fits use fixed model checkpoints, the same 300 test machines, and 8,638 jointly exact one-step transitions.

Table 15: Changes in paired answer readouts at $\tau = 0.8$. Means and sample standard deviations summarize five probe fits on fixed checkpoints and trajectories. Remain W requires W at every observation; WUC records a W→U→C path; Corrupt records any W after a paired C start. W starts are counts; all other entries are percentages.

	Paired	Remain W		WUC		Corrupt	
Steps	W starts	Token	State	Token	State	Token	State
1	1288 ± 197	71.73 ± 1.78	65.40 ± 1.93	0.00	0.00	0.006 ± 0.013	0.005 ± 0.006
2	1242 ± 187	54.65 ± 1.81	44.68 ± 3.39	0.08 ± 0.18	0.37 ± 0.29	0.025 ± 0.015	0.036 ± 0.015
3	1183 ± 179	42.88 ± 1.89	31.03 ± 3.88	0.71 ± 0.44	1.01 ± 0.83	0.069 ± 0.039	0.084 ± 0.051
5	1077 ± 162	26.04 ± 2.93	15.07 ± 3.71	2.14 ± 0.72	3.10 ± 1.16	0.20 ± 0.08	0.22 ± 0.11

At $\tau = 0.8$, all five fits give State fewer readouts that remain in W through five updates. The mean gap is 10.97 points (Table 15). Leaving W can mean becoming unresolved; complete W→U→C paths are rare.

At $\tau = 0.7$, the five-step persistent-W rates are 26.64±3.35% for State and 35.25±1.91% for Token. W→U→C paths occur on 11.52±1.34% and 9.22±0.69% of windows. Corruption also increases, to 2.28±0.41% for State and 1.49±0.09% for Token.

Table 16: Future-answer readouts during the same correct computation (percent; mean ± sample s.d. across five probe fits). Remain W requires W throughout a paired W-start window. WUC records W→U→C; corruption is any W following a paired C start.

Threshold	Readout event	Token	State
$\tau = 0.8$	Remain W	26.04 ± 2.93	15.07 ± 3.71
	WUC	2.14 ± 0.72	3.10 ± 1.16
	Corruption	0.20 ± 0.08	0.22 ± 0.11
$\tau = 0.7$	Remain W	35.25 ± 1.91	26.64 ± 3.35
	WUC	9.22 ± 0.69	11.52 ± 1.34
	Corruption	1.49 ± 0.09	2.28 ± 0.41

B.12 Complete LLM rollouts and route selection

Removing the code channel leaves 6.11% answer accuracy and no exact processes; paired exact McNemar tests against the intact engine give $p = 7.98 \times 10^{-40}$ for answers and 9.86×10^{-32} for processes. The permutation intervention uses $\{0{\to}3, 1{\to}4, 2{\to}0, 3{\to}1, 4{\to}2\}$, keeping the supplied code tokens valid while changing their learned identities.

Table 17: Answer and exact-process accuracy for 60 held-out PrOntoQA problems at each proof depth (percent).

	Token CoT		State, evidence set	
Depth	Answer	Exact	Answer	Exact
2 hops	60.00	38.33	81.67	71.67
3 hops	46.67	21.67	81.67	63.33
4 hops	38.33	10.00	81.67	38.33

Table 18: Paired outcomes on the same 180 PrOntoQA problems. Two-sided exact McNemar tests use the Token-only and State-only counts.

Metric	Both	Token only	State only	Neither	p
Answer	69	18	78	15	4.45×10^{-10}
Exact process	25	17	79	59	9.94×10^{-11}

Table 19: PrOntoQA evaluation of generated proof paths (percent). The downward arrows mark error or failure rates.

Metric	Token CoT	State, evidence set
Step legal	44.63	81.80
Every step legal	26.11	63.33
Legal complete	25.56	57.78
Wrong branch ↓	51.67	12.22
Skip ↓	12.78	3.33
Invalid ↓	67.78	8.33
Parse fail ↓	40.00	0.00
No answer ↓	32.22	12.22
Answer accuracy	48.33	81.67

The evidence audits use gold prefixes: 240 unseen transitions for set generation and selection, 120 adjacent-prefix pairs, and 120 selected-evidence replacements (Table 20).

Table 20: Learned evidence and execution audits. The evidence-set and route-selection audit uses 240 unseen gold-prefix transitions; prefix changes and evidence replacement use 120 held-out pairs and 120 transitions, respectively.

Audit	Value
Mean predicted set size	2.91
Learned <STOP> produced	100.00%
Current evidence present	93.33%
Exact evidence set	66.25%
First item is current evidence	44.17%
Route-selected current evidence	92.92%
Selection given current evidence present	99.55%
Consumed evidence removed at next prefix	100.00%
Still-needed evidence retained	81.06%
Executor output changed by evidence replacement	99.17%
Wrong evidence still produced correct next state	0.83%

C Implementation details

C.1 Matched Token and State models

For the matched tape comparison, we implement Token and State with the same two-layer encoder–decoder Transformer: width 96, four attention heads, a 384-unit feed-forward block, fixed sinusoidal positions, and 527,064 parameters. They share their vocabulary, initial weights, optimizer, 30-epoch budget, and shuffled sequence of 16,000 source workspaces. Sampling with replacement supplies 2,000 rows for each of the eight context-conditioned update labels.

Token learns to serialize the complete successor workspace through teacher-forced cross-entropy. State predicts one action label, whose effect is applied by the fixed executor used in this comparison. At one-step evaluation, Token's output is parsed without repair and scored according to its executed update; State's label is scored according to the effect produced by the executor.

Both systems begin from the same 300 disjoint test machines at every integer horizon from 3 through 30. The horizon is available only to the evaluator.

C.2 The learned tape interface

Let w_t be the workspace after the previous update has been committed, a_t the procedural state, z_t the execution state, and e_t the edit. For active tape instruction $u(w_t)$ and its transition demonstrations $D_{u(w_t)}$, the engine computes

$$a_t = G_\eta(D_{u(w_t)}), \qquad z_t = E_\theta(w_t), \qquad e_t = T_\phi(z_t, a_t), \qquad w_{t+1} = \mathrm{write}(w_t, e_t).$$

Here G_η reads the demonstrations, E_θ encodes the current workspace, and T_ϕ predicts an edit from their outputs. The writer applies this edit, and the resulting workspace is encoded at the next step.

Discovering operations and encoding the workspace. To learn the operation vocabulary, we randomly assign five visible symbols to five hidden tape operations in each environment. A symbol is accompanied by 16 unlabeled before-and-after probes. Symbols with identical successor signatures are assigned the same anonymous target. The targets and class count are obtained from these signatures automatically. Visible symbols change meaning across environments, so the reader learns their effects from the demonstrations.

We implement the workspace encoder as a separate Transformer and train it through straight-through categorical assignments and successor supervision, without human state-field targets. The executor receives only its output z_t and the reader's procedural code a_t (Figure 7). It predicts four edit fields: position, write value, program-counter displacement, and head displacement. The writer applies them by assignment and cyclic movement.

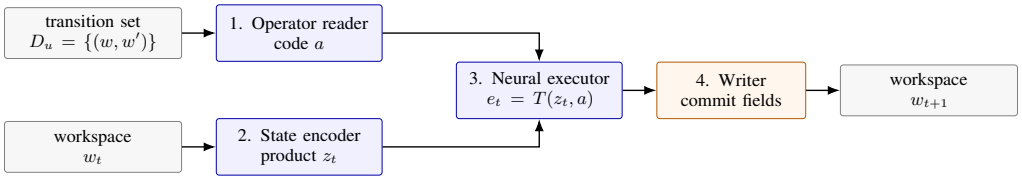


Figure 7: The learned tape engine. Demonstrations determine procedural code a_t; the workspace determines execution state z_t. The executor sees only these two codes, and the writer commits the predicted edit.

C.3 Learning operator codes and execution state

Each environment randomly assigns five visible symbols to five hidden operators. We train on 80 assignments and test on 20 disjoint assignments. For each symbol, a shared bank of 16 public probe workspaces supplies raw before–after transitions. We group symbols by equality of their complete successor-token sequences across this bank. These groups define anonymous self-supervised targets, and their observed number determines the class count. The reader receives no operator names or supplied operator count. Each split contains 4,800 training query transitions and 1,600 held-out query transitions.

We build the operator reader around a width-96 Transformer that encodes each before–after pair. It represents the probe set by concatenating the coordinatewise mean and maximum of these encodings, then predicts an anonymous class with a linear head. Training uses AdamW, batch size 128, and 140 epochs with cosine learning-rate decay from 5×10^{-4}. The probe-dropout condition keeps this architecture and optimizer and, for each minibatch, samples a common subset size uniformly from 8 through 16.

We give the execution-state encoder two width-128 self-attention layers with four heads. A linear selector produces four categorical variables, each with eight possible values. Straight-through hard assignments select entries from four learned 12-dimensional codebooks. The executor concatenates these four embeddings with a 16-dimensional operator embedding and passes the result through a two-layer width-128 MLP. Its four outputs specify edit position, write value, cyclic program-counter

displacement, and cyclic head displacement. All workspace information reaches this executor through the categorical state. Training runs for 180 epochs with AdamW, batch size 128, cosine decay from 5×10^{-4}, code-usage weight 0.05, and slot-independence weight 0.02.

The writer applies the four predicted fields by assignment and cyclic pointer arithmetic, preserving read-only fields. It applies even incorrect edits directly, without searching or checking their validity. The routine contains no individual operator implementations or calls to the reference transition function.

For the post-training audits, hidden operator identities supply the labels for purity and normalized mutual information. These labels never enter optimization. The nuisance audit changes the program counter between paired workspaces while keeping the required edit fixed. The composition audit begins from unseen workspaces and feeds each predicted successor into the next primitive operation. Operator interventions replace the action code at the executor input. Execution-state interventions fix, shuffle, or substitute categorical variables while keeping the requested operator constant.

C.4 Controlled-English rules

Each program acts on four categorical fields with six values per field. A reference rule tests three or four field values and replaces one or two. Programs contain two or three reference updates and one distractor rule. We retain a generated program only when exactly one rule is licensed at every reference workspace, and shuffle rule order independently for each case. Each prompt includes a balanced question about one terminal field. The primary endpoint is the complete final workspace, which detects an error in any of the four fields.

The applicator carries out its replacements even if the preconditions are false, passing any resulting error to the next step.

The fixed data seed produces 6,000 training programs and 600 disjoint test programs, containing 14,704 and 1,495 transitions, respectively. Both systems use three-layer encoder–decoder Transformers with width 128, four heads, a feed-forward width of 512, sinusoidal positions, and 1,404,477 parameters. Within each run, they share initialization, vocabulary, source rows, minibatch order, AdamW settings, and two consecutive 50-epoch cosine schedules. State predicts the shuffled rule index; Token teacher-forces the complete successor workspace. Outputs are parsed without repair. Seeds 260908–260910 vary initialization and minibatch order while keeping the programs fixed.

The free-rollout audit scores each update from the model's actual current workspace. An update is action-conforming if at least one presented rule produces it through unconditional field replacement. It is licensed when that rule's preconditions also hold in the same workspace. A parseable Token successor that matches no rule effect is off-action. A premature-future update reaches a later reference workspace before executing the intervening licensed updates.

The conditional controllability audit uses all 1,495 gold-input test transitions for each seed. It uses the frozen saved models with no further training. We compute Hamming distances over the four workspace fields and score one-variable oracle repair from each saved prediction and its gold target.

C.5 Qwen Policy, Grounder, Route Value, and Executor

We implement the four modules with separate LoRA adapters, exclude prompt tokens from every adapter loss, and disable Qwen thinking. Token uses LoRA rank 8. Policy uses rank 4 on eight Transformer layers and trains for 2,400 optimizer steps. With the selected five-code interface, Policy reaches 100% accuracy on 1,440 held-out transitions. The eight-code alternative reaches 85.63% and fails the prespecified validation audit.

Grounder uses rank 8 on all 16 layers. Its 4,800 training contexts are paired with variable-length evidence sets ending in <STOP>. Training uses a maximum sequence length of 640 and two consecutive 600-step schedules, starting at learning rates 8×10^{-5} and 4×10^{-5}. On 240 held-out gold-prefix transitions, the generated sets achieve 87.26% recall, 84.17% precision, and 66.25% exact-set accuracy; every generation produces the stop token. Executor uses rank 4 on eight layers and trains for 600 steps. Given the problem, code, and selected evidence, it emits one next-state line.

Route Value uses rank 4 on eight layers and trains for 2,400 steps on 19,200 positive and negative proposal rows. It scores evidence with the logit difference between `YES` and `NO`. At inference, its choices are restricted to the set produced by one deterministic Grounder call. On the 240 held-out gold-prefix transitions, the first generated item is the current evidence in 44.17% of cases. Route Value selects the current evidence in 92.92% overall and in 99.55% of cases where Grounder includes it. A separate pairwise audit places correct evidence above wrong evidence in 99.51% of comparisons.

We also audit changes to the prefix and to the selected evidence. Across 120 held-out adjacent-prefix pairs, Grounder removes consumed evidence in 100% of pairs and retains 81.06% of evidence needed later. In 120 held-out transition interventions, replacing the correct evidence changes 99.17% of Executor outputs; only 0.83% retain the correct next line, despite Executor's access to the full problem.

We test code-channel interventions on the same 180 problems, keeping Policy, Grounder, Route Value, and Executor frozen. The *removed* condition omits the `ACTION CODE` line from all three downstream prompts. The *fixed* condition supplies `S0` at every step. The *permuted* condition first runs Policy normally, then applies a fixed permutation with no identity mappings before sending the resulting valid code to Grounder, Route Value, and Executor. Decoding, stopping, evidence-set parsing, route scoring, and evaluation remain identical across conditions.

C.6 Three-way probe protocol

For each matched model, we collect encoder representations at all 31 gold workspace positions in 1,500 independent training trajectories. We mean-pool and standardize the representations, then fit ten linear heads to predict the terminal eight tape bits, program counter, and head. Each head is shared across every trajectory position.

Probe training uses AdamW for 25 epochs with learning rate 0.01, weight decay 10^{-3}, and batch size 256. Seeds 9137–9141 change only the probe fit. The model checkpoints, probe-training data, and 300 test trajectories are fixed, and reported spreads are sample standard deviations across the five fits.

At thresholds 0.7 and 0.8, each field's maximum softmax probability and predicted label assign its readout to confident-correct (C), confident-wrong (W), or unresolved (U). Comparison windows begin with both models in W, or both in C for the corruption analysis. Both models must then produce exact reference successors at every intervening update, ensuring that each pair traverses the same workspaces.

"Remain W" requires a W readout at every observed position, allowing the particular wrong label to change. WUC records a W→U→C path. Corruption records any W readout after a paired C start. All probes remain frozen during evaluation and their predictions are used only for this analysis. Confidence refers to the probe output, without assigning belief status to the underlying model.

D Theoretical analysis

We formalize the workspace information needed for execution, the conditions for identifying operations, and how component errors affect composition and transfer.

Minimal sufficient state. Let $e(w, o)$ be the edit required by workspace w and operator o, and define $w \sim w'$ when $e(w, o) = e(w', o)$ for every operator. For an equivalence class $[w]$, set

$$T_z([w], o) = e(w, o).$$

The equivalence relation guarantees that this definition gives the same edit for every representative of $[w]$. The quotient is therefore sufficient for exact execution. Now let h be any other sufficient deterministic representation. If $h(w) = h(w')$, its decoder receives identical inputs for the two workspaces under each operator and must return identical edits. Hence $w \sim w'$. Every sufficient representation thus refines the quotient. The quotient preserves precisely the workspace distinctions required for execution and is unique up to relabeling.

Operator identification. Assume that the shared probe bank separates the operators and that an exact deterministic executor obtains operator identity only through code a. Suppose two distinct

operators receive the same code. For any fixed probe workspace, the executor then receives identical inputs for both and returns the same output. Because the bank separates the operators, at least one probe requires different outputs, contradicting exactness. An exact interface therefore requires at least $|\mathcal{O}|$ codes. A minimum exact representation assigns one code to each operator and identifies the operator family up to a permutation of code identities.

Modular error and finite composition. Condition on a correct prefix. If the execution state preserves the required information, the procedural code is correct, and the executor is correct on those inputs, the next workspace is correct. Successor failure must therefore fall within the union of the three corresponding error events. Writing their probabilities as $\delta_z(t)$, $\delta_a(t)$, and $\delta_T(t)$ gives

$$\epsilon_t = \Pr(\hat{w}_{t+1} \neq w_{t+1} \mid \text{correct prefix}) \leq \delta_z(t) + \delta_a(t) + \delta_T(t).$$

The chain rule over correct prefixes gives

$$\Pr(\text{no failure by } H) = \prod_{t=0}^{H-1} (1 - \epsilon_t), \qquad \Pr(\text{failure by } H) = 1 - \prod_{t=0}^{H-1} (1 - \epsilon_t) \leq \sum_{t=0}^{H-1} \epsilon_t.$$

These relations require no independence assumption between steps. When every primitive update has zero error, induction on the number of updates establishes exact execution of every finite composition.

Transfer. Suppose z^* is sufficient for both the source and target operator families. Let δ be the probability that the transferred encoder fails to supply the required z^* class, and let ϵ be the target executor's error probability given a correct class. A target failure belongs to at least one of these events, so the union bound gives error at most $\delta + \epsilon$.

This result also identifies when the transferred partition needs refinement. If a target operator requires different edits for two workspaces merged by the source representation, a frozen decoder must give both the same prediction. Let that pair have target probability mass ρ, with conditional masses p and $1 - p$ for its two members. Its contribution to total error is at least $\rho \min\{p, 1 - p\}$. Resolving such representation errors requires fine-tuning to split source classes or to correct encoder errors within classes that are already sufficient for the target family.