Morphz: Nondeterministic Cognitive Symbol Evaluation over Structured Context

Raymond Ren

Abstract

Language-model agents usually hide cognitive state inside append-only messages, prompts, and tool loops, leaving stable identity, revision relations, and result re-entry implicit. We present Morphz: programs, cognitive data, and evaluation results use the same recursively nested structured representation, while a large language model (LLM) performs **nondeterministic cognitive evaluation** over cognitive symbols in agent-owned Structured Context. Frames, relations, observations, bindings, provenance, and versions form addressable computational state across sessions and model instances. Model results may re-enter the current expression or become durable cognitive objects referenced by later tasks. A model-returned Yao program can also become a first-class Program Value after type, effective-effect, and capability admission, execute through a durable child boundary directed by its evaluation owner, and rejoin the parent result, while a deterministic Runtime retains authority over program validation, capabilities, versions, real observations, and committed effects.

Layered experiments show that this mechanism executes across three equal-information program representations and nine model configurations. In the nine-model matrix, 104/108 nondeterministic-evaluation responses returned a parseable selection permitted by the visible contract, and all eight authoritative-state commitment and fault-handling tests passed. In a three-system STATE-Bench comparison, Morphz, Letta, and the Mem0-backed reference all used GPT-5.6 Sol/max. Morphz completed 122/150 held-out tasks (81.33%), versus Letta at 93/150 (62.00%) and the Mem0-backed reference at 96/150 (64.00%). A trace audit verified that every Morphz held-out task used the corresponding domain's revision-100 Structured Context, each consolidated from 100 training trajectories (300 total). In complete Terminal-Bench 2.1 runs using the same model, host, and frozen protocol, the latest fixed Morphz Runtime and Codex CLI 0.149.1 each scored 74/89; the 95% task-bootstrap difference interval was [-7.87,+7.87] percentage points. These results support an executable combination of structured cognitive state, nondeterministic cognitive evaluation, learned-experience transfer, and deterministic Runtime control over authoritative state and real-world effects. S-expressions are the current representation, not the contribution itself.

Keywords: language-model agents; nondeterministic cognitive evaluation; cognitive symbols; Structured Context; S-expressions; persistent cognitive state; deterministic commitment

1. Introduction

Tool-using language-model agents have made a rapid transition from demonstrations to software systems. ReAct couples reasoning with environmental action [2], Toolformer studies self-supervised API use [3], and subsequent systems add planning, reflection, code execution, memory, and orchestration. Yet the

dominant execution model remains close to chat: serialize the past as messages, append a new observation, ask a model for the next message or tool call, and repeat.

This message-centric design is useful because it matches the interface of current language models. It does not, by itself, define a persistent agent. A message has no stable identity beyond its position in a transcript. A fact, a goal, a user constraint, a superseded belief, and an unprocessed tool result are all represented as text spans. Their provenance, scope, revision status, and lifetime are either implicit or reconstructed on every model call. External memory can retrieve older spans, but retrieval alone does not turn them into typed operands in an ongoing computation.

The limitation becomes visible when an agent must do more than complete one request. Consider an agent that works across several sessions, maintains multiple objectives, changes its model provider, survives a runtime restart, and revises a policy when new evidence conflicts with old experience. A transcript can describe each of these events, but it does not provide a uniform mechanism for naming the relevant cognitive objects, expressing their relations, or feeding a derived result into later evaluation without another round of unstructured interpretation.

Morphz starts from one primary premise: **an agent should own a Structured Context that serves as the state of cognitive evaluation**. Conversation is one input/output channel into that state, not the state itself. Frames, relations, observations, bindings, and version metadata provide stable cognitive objects. An LLM evaluates expressions over those objects, including cognitive symbols whose values depend on context, observations, and model knowledge. The evaluator is nondeterministic because several contextually admissible results may exist and because the symbol's semantics are not reduced to one fully specified deterministic state-transition function.

Using one representation for programs and cognitive data is equally important. Asking a model to emit JSON is not the same as constructing a persistent evaluative state. A returned value must retain identity and reference semantics, and a later expression must actually consume it. In the current Morphz implementation, a tool or semantic result may be bound within the current expression, committed as a durable Frame or Relation, or returned as a candidate Yao program. After type, effective-effect, and capability admission, that candidate becomes a first-class Program Value executed through an owner-directed durable child boundary whose result rejoins the parent plan.

The contrast is computational rather than merely probabilistic. *Nondeterministic* does not refer only to sampling noise: a context-dependent cognitive expression may admit several acceptable values, and its evaluation is conditioned on information that is not encoded as a fully specified deterministic transition function. The runtime, by contrast, determines which references resolve, which tools are permitted, which version may commit, and which real observation was produced. A plausible value generated by the LLM is only a candidate: it cannot commit state or trigger a real-world effect by itself.

Morphz makes the boundary between cognition and state-changing authority explicit. Runtime evaluation may suspend at an `infer` subtree and transfer that cognitive term to the LLM. The implementation accepts typed data results and inserts them into the suspended computation; when the declared result is `Program<T,E>`, the model instead returns one raw Yao candidate rooted at `(eval ...)` or `(infer ...)` that the Runtime admits, persists, and executes explicitly. These paths combine nondeterministic cognitive evaluation with deterministic operational evaluation while leaving state commitment and real-world effects under Runtime control.

This paper asks the following question:

> *Can a persistent agent use an LLM as a nondeterministic evaluator of cognitive symbols over structured, persistent context, with programs, cognitive data, and returned values sharing a recursive term representation, while state commitment and real-world effects remain under runtime control?*

We make four contributions:

1. We define the computational role of an LLM as a nondeterministic cognitive symbol evaluator whose values are conditioned on a Structured Context, observations, and an explicit operator kernel.
2. We define Structured Context as the persistent state of evaluation and use the same recursive structured representation for cognitive objects, candidate programs, and returned values, so that they can be addressed and consumed rather than merely appended as text; prior tasks can also be transactionally consolidated into newly formed or revised Mind Frames whose explicit experience participates as computational state in later Sessions.
3. We describe a working Morphz implementation with an S-expression term layer, typed result binding, first-class Program Value admission and durable child execution, versioned context transactions, deterministic plan validation and commitment, durable event records, and recoverable scheduling.
4. Through controlled mechanism experiments, tests across nine model configurations, and system-level comparisons on STATE-Bench and Terminal-Bench, we evaluate executability, cross-model applicability, experience transfer, and system-level task performance, while reporting observed failure modes and scope limitations.

The intended claim is deliberately narrower than "an LLM is a CPU" or "an agent is a virtual machine." Both descriptions already appear in prior systems. Morphz's central claim is instead the combination of **an LLM as a nondeterministic evaluator of context-dependent cognitive symbols; agent-owned Structured Context as persistent computational state; one recursive representation used by cognitive data and candidate programs; structured result re-entry; and deterministic Runtime control over state commitment and real-world effects**. Context-dependent admissible semantics explain why deterministic code does not exhaust these symbols; the primary computational identity remains nondeterministic cognitive evaluation.

2. From Message Loops to Cognitive State

2.1 Why append-only dialogue is not enough

Let a conventional agent construct a message history

$$H_t = [m_1, m_2, \ldots, m_t]$$

and sample the next response or action from a model conditioned on H_t. Context-window management may truncate, summarize, or retrieve messages, but the basic object remains a sequence. This representation has three useful properties: it is easy to serialize, it preserves conversational order, and it matches the training interface of current models. It has weak support for several properties required by persistent computation:

- stable identity for a cognitive object;
- explicit relations such as `derived-from`, `supersedes`, or `conflicts-with`;

- local and cross-task scope;
- versioned revision and concurrent update;
- a distinction between an unprocessed observation and a durable conclusion;
- direct reference to a prior result as an operand.

These properties can be simulated in prose, but simulation requires the model to reconstruct them on each call. The resulting semantics are implicit, expensive to inspect, and difficult to validate independently.

2.2 Structured context as the agent's state

Morphz models the active cognitive state at time t as

$$C_t = (F_t, R_t, O_t, B_t, V_t)$$

where:

- F_t is a set of frames with stable identifiers, content, type, provenance, and lifecycle metadata;
- R_t is a set of explicit relations among frames and observations;
- O_t is the set of active observations not yet retired from working context;
- B_t is the local binding environment of the expression under evaluation;
- V_t contains context-version and transaction metadata.

We use **Mind Frame** for a persistent Frame when emphasizing its role as a learned, revisable unit of the agent's cognitive state. It is not a hidden model activation: it is an explicit, versioned object with provenance and relation edges that can be projected into later evaluation.

This definition is intentionally implementation-neutral. A conforming system need not use the same schema, but its cognitive objects must be addressable, related, revisable, projectable to the evaluator, and recoverable independently of a single model call or conversation session.

Structured context is not synonymous with long-term memory. A memory store answers what information can be retrieved. A computational context additionally determines what values are active, which version is authoritative, how a value was derived, what can be modified, and how an expression may refer to it. In Morphz, memory operations are themselves context operations rather than a side channel attached to an otherwise message-based agent.

This distinction changes the meaning of learned experience. Transfer does not consist only of storing an earlier episode and retrieving a similar text span. An earlier task can cause the agent to form, revise, relate, protect, or retire Mind Frames; the resulting structure becomes the Context in which a later cognitive expression is evaluated. The learned object is therefore both durable data and part of the subsequent computational state.

We use *experience transfer* or *test-time agent learning* for this state-mediated process. It does not imply gradient updates, fine-tuning, or any change to the foundation model's weights.

Long-term memory benchmarks commonly evaluate the retention, retrieval, and synthesis of information from long histories; LongMemEval-V2 is a representative example [22]. STATE-Bench instead evaluates whether experience formed in earlier tasks transfers to new tool-using tasks [25]. This paper asks whether Mind Frames, once formed, participate as computational state in later action rather than merely supporting answers over long histories. ME-07 therefore uses STATE-Bench.

For a domain d, the ME-07 mechanism can be written schematically as

$$C_d^{(i+1)} = \mathrm{Commit}\Big(C_d^{(i)}, \Delta_\theta(\tau_i, C_d^{(i)})\Big)\,,$$
$$a_q \sim \mathrm{Eval}_\theta\Big(e_q; \Pi_q(C_d^{(100)})\Big)\,.$$

where τ_i is a completed training trajectory, Δ_θ is a model-proposed set of Frame and Relation changes, Commit is the Runtime-validated Context transaction, and Π_q is the task-conditioned projection supplied to the held-out cognitive evaluation. The empirical question is not merely whether τ_i remains stored, but whether its consolidated effects survive in $C_d^{(100)}$ and participate in the production of a_q.

ME-08 compares the task-completion capabilities of Morphz and a conventional agent on complex terminal tasks. We therefore use Terminal-Bench 2.1, which contains 89 difficult command-line tasks, with Codex CLI as the conventional-agent comparator under the same model and execution environment [23,26].

2.3 Conversation as I/O

A Morphz session routes input and output; it does not own the agent's mind. Several sessions may attach to one context, and the same context may continue after a model or runtime process changes. A user message becomes an observation associated with an activation. A reply is a delivery event derived from the resulting computation. This separation is the basis for multi-session continuity: sharing a context does not mean concatenating all session transcripts.

It also gives a precise meaning to agent identity. The persistent subject is not a model-weight snapshot and not a session token. It is the recoverable combination of Structured Context, durable events, commitments, and unfinished continuations. A model is an evaluator used by that subject.

2.4 Design objectives

The mechanism is designed for six properties:

1. **Structure.** Goals, facts, experience, plans, observations, and relations must not be distinguished only by paragraph position.
2. **Recursive composition.** Larger cognitive operations are composed from smaller terms whose intermediate values can be named and referenced.
3. **Nondeterministic cognitive evaluation.** The LLM evaluates cognitive expressions whose admissible results are conditioned by Context and observations and need not be unique, rather than simulating one fully specified deterministic state transition.
4. **Separated evaluation responsibilities.** The LLM evaluates context-dependent cognitive symbols; the Runtime evaluates predefined deterministic control logic and decides which state changes may be committed and which real-world effects may execute.
5. **Result re-entry.** A result can re-enter local bindings or durable Context instead of terminating as unstructured text; the model may also propose a new candidate program at an activation boundary.
6. **State-and-effect isolation.** Validation, permission, transaction, and causal boundaries separate candidate cognition from committed state and real-world effects.

3. Cognitive Symbol Evaluation

3.1 A recursive term domain

Let Σ be a set of cognitive operators and constructors, and let A contain atomic strings, numbers, booleans, identifiers, and references. Expressions belong to the free term algebra

$$e \in T_\Sigma(A) ::= a \mid (s\ e_1\ \ldots\ e_n)$$

for $a \in A$ and $s \in \Sigma$. Morphz currently serializes these terms as S-expressions. For example:

```
(resolve-conflict
  (experience (ref frame:old-policy))
  (evidence (ref observation:new-contract))
  (preference (ref frame:user-boundary)))
```

When we say that programs and data use the same recursive representation, we mean that executable structures, returned structures, and context objects share a recursive term representation and common constructors, references, traversals, and transformations. Lisp homoiconicity is well established and is not itself a contribution of this work [1].

The representation is valuable because it allows the main relation of an expression to be exposed at the root while preserving recursive composition. It can represent a fact, a request to infer a fact, a bound result, a conditional computation, a proposed tool plan, or a new expression using the same structural conventions.

3.2 Deterministic and nondeterministic cognitive symbols

Traditional processors are effective evaluators when semantics are fully specified. An expression such as `(+ 1 2)` has a fully specified operator, input domain, and transition rule. The expression

```
(resolve-conflict
  (experience old)
  (evidence new)
  (preference user))
```

does not. A conventional runtime can evaluate it only after a programmer specifies what counts as conflict, how to rank evidence, how preferences affect the decision, how exceptions are handled, and what output is admissible.

Language models change this constraint. They can interpret a symbol from its name, structural position, natural-language operator description, examples, current context, new observations, and parametric world knowledge. An operator may therefore be constrained without being exhaustively implemented as a deterministic function. We call such an operator a *nondeterministic cognitive symbol*.

Nondeterministic does not mean unconstrained. Each operator may define an input shape, an output contract, semantic instructions, admissibility conditions, and examples. The nondeterminism lies in the contextual semantic judgment required to obtain an admissible value, not in the absence of a protocol.

3.3 Context projection

The complete context may exceed a model's physical context window or contain information that the current evaluator is not permitted to observe. The runtime constructs a projection

$$\pi_t = \mathrm{project}(C_t, e_t, p_t)$$

where p_t contains relevance, permission, and budget policy. Unlike retrieval that returns anonymous text fragments, a projection preserves identifiers, selected relations, and reference paths. The evaluator may request additional frames or observations through controlled operations when the initial projection is insufficient.

3.4 The nondeterministic cognitive evaluator

For a fixed cognitive term, Context projection, observation, and evaluation kernel, first define the set of values admitted by the visible Context and output contract:

$$\mathcal{A}(e_t, \pi_t, o_t, k) = \{v \in V \mid \mathrm{accept}(v; e_t, \pi_t, o_t, k) = 1\}.$$

When $|\mathcal{A}(e_t, \pi_t, o_t, k)| > 1$, the expression has a one-to-many admissible evaluation relation at the specification level. Following the programming-semantics sense in which the same initial conditions need not uniquely determine the result, we call this nondeterministic cognitive evaluation [27]. The language model implements candidate selection through a conditional distribution:

$$q_\theta(v \mid e_t, \pi_t, o_t, k), \qquad v_t = \mathrm{decode}(q_\theta)\,.$$

Only a candidate satisfying $v_t \in \mathcal{A}(e_t, \pi_t, o_t, k)$ constitutes an accepted evaluation result. Nondeterminism is therefore primarily a one-to-many relation at the specification level; the probability distribution and sampling or greedy decoding are implementation mechanisms for selecting a candidate.

Here e_t is the current cognitive term, π_t is the projected Context, o_t is the current observation, and k is a kernel of operator descriptions, examples, and output contracts. The evaluator returns a cognitive value v_t; the admissible value domain may include text, structured data, Frame or Relation candidates, bindings, and candidate program terms. The current `infer` form accepts one complete Yao BODY rather than fixed task and evidence fields. That BODY uses the same parser, type checker, and effect analysis as `eval`; the outer operator selects whether the LLM or Runtime owns its evaluation. At a nested ownership boundary, only parent bindings explicitly named by the Yao source in `(captures NAME...)` may be sent with that `infer` request to the currently configured model provider. Unlisted bindings and the whole Runtime environment do not cross implicitly, and `captures` authorizes data disclosure rather than Tool or object capabilities. Each result is decoded under its declared Yao type, including `String`, `Json`, nominal or compound types, reference types, and `Program<T,E>`. When the declared result is `Program<T,E>`, the model may instead return one raw Yao candidate rooted at `(eval ...)` or `(infer ...)`; it becomes executable only after Runtime parsing, type checking, preservation of evaluation ownership, effective-effect/capability bounding, canonicalization, provenance binding, and persistence. For an `infer` root, entering a new formal child Evaluation is itself included in the effective `infer` effect.

In the proposed computation model, the LLM serves as a **nondeterministic cognitive evaluator**. Such an evaluator is needed because the result of a cognitive symbol is not uniquely determined by the symbol's name and arguments; it also depends on the current Structured Context and observations. A

cognitive operator can still be constrained by input structure, natural-language instructions, output contracts, and acceptance conditions without requiring a programmer to exhaustively implement all of its judgment rules in deterministic code.

3.5 The deterministic runtime evaluator

Let P_t be a validated and lowered plan, S_t the authoritative Runtime state, and x_t an external input already observed by the Runtime, such as a tool result, clock event, or scheduling event. Runtime evaluation is a constrained state transition:

$$\mathrm{eval}_R : (P, S, X) \to (S', O, \kappa)$$

or, for one step:

$$\mathrm{eval}_R(P_t, S_t, x_t) \to (S_{t+1}, o_{t+1}, \kappa_t)$$

The Runtime evaluates operators whose semantics are fully specified, resolves references, checks permissions and versions, dispatches effects, records the resulting observation, and determines which state transition may commit. κ_t indicates completion, waiting, failure, or a continuation. Here, *deterministic* qualifies the Runtime's validation, evaluation, and commitment decision given a plan, committed state, and explicit external input; it does not claim that tool results, concurrent scheduling, networks, or the external environment are themselves deterministic.

3.6 Composing cognitive and runtime evaluation

Morphz currently connects nondeterministic cognitive evaluation and deterministic runtime evaluation through the following implemented paths:

```
model activation
  └── proposes an eval program
        └── parse + validate + lower
              └── eval_R executes predefined deterministic control and effects

eval_R executing a typed plan
  └── suspends at an infer node and hands the complete Yao BODY to infer_θ
        ├── only explicitly captured parent bindings cross with the request
        └── infer_θ evaluates that BODY and returns the declared result type
              ├── data value: Runtime decodes, validates, binds, and resumes
              └── Program<T,E>: raw Yao (eval ...) or (infer ...) candidate
                    └── Runtime admission + owner-directed dispatch
                          ├── eval root: durable child Plan
                          ├── infer root: formal child Evaluation
                          └── declared result rejoins and resumes parent Plan
```

This is more structured than treating arbitrary model tool calls as approved state changes or real-world effects. The model proposes a recursive term; the Runtime owns validation, references, capabilities, scheduling, and commitment. Within a validated plan, an `infer` node hands the same complete Yao BODY to the model-owned evaluator, creates a durable child Evaluation, and returns its typed result as deterministic plan data.

A model-returned program executes only after the enclosing program declares a `Program<T,E>` result, Runtime admits it, and `(run PROGRAM)` invokes it explicitly. Runtime revalidates current authority, prevents capture of caller-local bindings, and persists the admitted program behind a causally linked child-execution boundary. An `eval` root advances as a Runtime-owned child Plan; an `infer` root immediately creates a formal child Evaluation whose typed terminal value resumes the parent. Either form may contain or return further Program Values; Program nesting, inference, tool use, and aggregate child work share bounded budgets. Deterministic tests cover admission of both `eval`-rooted and `infer`-rooted Program Values, effective type/effect bounds, durable child execution, owner dispatch, and restart recovery. Dynamic-adaptation effects on real tasks remain to be evaluated separately.

3.7 Structured result re-entry

The central mechanism is not structured generation in isolation. A generated value can enter local bindings or, after transactional commitment, become a Context object referenced by later expressions. Let normalization produce candidate state changes and local bindings:

$$n(v_t) \to (\Delta C_t, b_t)$$

A permitted context transition is then committed against a version:

$$C_{t+1} = \mathrm{commit}(C_t, \Delta C_t, V_t)$$

and a later expression operates on the updated context:

$$\mathrm{infer}_\theta(e_{t+1}, \mathrm{project}(C_{t+1}), o_{t+1}, k)$$

The current system supports three result-re-entry paths:

1. **Local re-entry.** A tool or semantic result is bound to a name and referenced by a later subtree in the same expression.
2. **Context re-entry.** A result becomes a versioned frame or relation that can be referenced, revised, or superseded in a future activation or session.
3. **Program-valued re-entry.** A nested `infer` may return a raw Yao candidate rooted at `eval` or `infer`. After admission, explicit `run` creates the owner-appropriate durable child execution and rejoins its terminal value into the parent.

This criterion distinguishes the proposed loop from common structured-output patterns. Saving a model response as an untyped JSON blob is useful persistence, but it does not provide re-entry unless later computation can address and consume its internal structure with defined identity and reference semantics.

3.8 The boundary for state commitment and real-world effects

Not every term must cross the runtime boundary. Pure cognitive evaluation may remain with infer_θ; a subtree whose semantics are fully specified may remain with eval_R. The invariant is that an effectful or authoritative state transition crosses the deterministic runtime boundary. A representative cross-activation cycle is:

$$
\begin{aligned}
(e_t, C_t, S_t) &\xrightarrow{\text{infer}_\theta} (v_t, \Delta C_t^{\text{cand}}, P_t^{\text{cand}}) \\
&\xrightarrow{\text{validate+lower}} (\Delta C_t', P_t') \\
&\xrightarrow{\text{eval}_R} (S_{t+1}, o_{t+1}, \kappa_t) \\
&\xrightarrow{\text{project}} C_{t+1}.
\end{aligned}
$$

This decomposition protects both sides of the design. Moving predefined deterministic control and operational effects to eval_R does not weaken nondeterministic cognitive evaluation; it preserves the LLM's freedom within its semantic domain. Conversely, infer_θ may propose a state change but cannot commit it by itself. The architecture would be weakened if context-dependent cognitive operators were prematurely reduced to fixed runtime rules, or if an unverified cognitive candidate were committed as authoritative state.

4. System Architecture

4.1 Overview

The current Morphz prototype has four logical layers:

1. **Structured Context.** Stores frames, relations, observations, lifecycle state, and context versions, and projects relevant cognitive state to an evaluator.
2. **Cognitive Term Layer.** Represents cognitive data, control structure, and candidate programs as recursive S-expression/Yao terms.
3. **Plan and Validation Layer.** Checks effectful candidates and lowers them into a typed plan intermediate representation.
4. **Runtime Layer.** Executes plans through a scheduler, records authoritative events durably, delivers outputs, and recovers unfinished work.

These layers are separated by authority rather than by deployment. The nondeterministic cognitive evaluator spans Context projection, cognitive-symbol evaluation, and structured candidate generation. The deterministic Runtime spans validation, evaluation of subtrees whose semantics are fully specified, scheduling, and commitment. They may run in one process in the current prototype. Their rights remain distinct: the LLM may construct values and candidate programs, while only the Runtime can authorize and record operational transitions. `eval` and `infer` share the same parsed and typed Yao BODY; changing the outer operator changes its evaluation owner. A nested `infer` does not inherit the whole parent scope: only bindings explicitly listed by source-level `captures` may enter its model request.

4.2 The Yao syntax tree and shared Language Card

Yao source uses an S-expression surface syntax whose parsed abstract syntax tree contains only atoms and recursively nested lists. Semantics do not come from this minimal tree alone. A single Language Card describes types, value constructors, control forms, and effectful operators; semantic analysis then lowers source into typed intermediate representation. The model and Runtime consume the same Language Card and type contracts rather than maintaining separate operator descriptions that merely share names.

An expression can combine deterministic dataflow with a nondeterministic cognitive step:

```
(eval
  (requires (tools read))
  (seq
    (bind source
      (call read (path "docs/design.md")))
    (infer
      (captures source)
      (returns String)
      (seq
        (bind criterion "retain claims verifiable against the source")
        "use criterion to extract the central hypotheses and evidence from source"))))
```

Operator semantics are supplied by the Language Card, type system, and Runtime specification rather than relying entirely on a model's pretrained interpretation of names such as `seq` or `bind`. The specification defines evaluation order for `seq`, prohibits effects from an unselected `if` branch, fixes lexical binding scope and parallel-branch isolation, and permits `run` to execute only an admitted Program Value.

In recorded Morphz runs, models have also constructed expressions of the form `(eval (requires ...) (seq ... (infer ...)))` without being given the complete instance in advance. In this form, `eval` submits a recursive program to deterministic Runtime evaluation, while `infer` gives its complete BODY to the nondeterministic cognitive evaluator. The parent binding `source` crosses that boundary only because the source lists it in `captures`. The typed result then returns to the surrounding program. Autonomous program construction is an execution-trace observation; ME-02 and ME-05 evaluate a fixed suite of recursive programs.

4.3 Validation and typed plan lowering

An S-expression that contains side effects is not executed directly. The validator checks structural depth, call counts, tool registration, capability declarations, references, and output contracts. It then lowers the candidate into a typed plan intermediate representation (IR).

The current typed representation covers value constructors, `Seq`, `Bind`, `If`, `Match`, `Fallback`, `Map`, `Infer`, `Call`, `Par`, `Run`, and `Host`. An `Infer` node preserves a complete typed Yao BODY, an explicit capture set, and a result type; that type may be `String`, `Json`, a nominal or compound type, a reference type, or `Program<T,E>`. The implementation places explicit limits on expression depth, map width, tool calls, inference nodes, parallel branches, Program Value nesting, and aggregate child work.

Typed plans are bridged to the scheduler kernel. This boundary provides three properties:

- the model proposes a plan but does not directly exercise effect authority;
- references, tool access, and resource constraints can be inspected before dispatch;
- real tool outputs and completion events are recorded as durable events rather than existing only as transcript text.

4.4 Context transactions

Context mutation uses an explicit S-expression transaction protocol. A transaction carries a base context version and may derive, revise, protect, unprotect, or retire frames; add relations; and retire observations that have been incorporated into durable cognition. The runtime automatically rebases disjoint updates

and rejects conflicting or stale updates.

This protocol turns “what the model remembers” into an auditable state transition. A frame has an identity and provenance. A relation can record derivation or supersession. An observation may remain as evidence, be retired from the active projection, or be summarized into another frame. Multi-version concurrency control (MVCC) prevents sibling objectives from silently overwriting the same cognitive object.

The current runtime also fences the root observation of an active user request: Context maintenance cannot retire the request before the associated delivery has completed. This preserves the request as an active operand throughout its reply lifecycle.

4.5 Durable execution, scheduling, and continuation

The runtime persists objectives, evaluations, activations, tool jobs, observations, and deliveries. A tool result is recorded as an observation and may awaken a continuation. Leases and fencing allow unfinished objectives to be resumed after a process restart without granting two evaluations simultaneous authority over the same work.

This machinery makes re-entry durable. A structured result need not remain in one model response or in process memory; it can be committed, projected into a later evaluation, and consumed by a different model instance.

5. Evaluation

5.1 Research questions and evidence boundary

The evaluation comprises mechanism, system, and external-validity studies that ask six questions:

- **RQ1 — representation:** can the same recursive structure and evaluation semantics be rendered as an S-expression, a JSON abstract syntax tree (AST), or a Markdown program without a representation-specific loss of task correctness?
- **RQ2 — cognitive state:** can structured Context preserve and revise current state across sessions, concurrent updates, process restarts, and delayed use, without degrading the final decision relative to message or compaction baselines?
- **RQ3 — nondeterministic evaluation:** can a cognitive symbol admit several contract-valid values, remain constrained by the visible Context, and retain this property across model families?
- **RQ4 — authority:** can the deterministic runtime prevent a model-generated candidate, stale transaction, replay, or untrusted observation from silently becoming authoritative state or a real side effect?
- **RQ5 — learned experience:** after observing the same completed training trajectories, how reliably do Morphz, the complete open-source Letta agent runtime, and a Memo-backed vector-reference agent transfer procedural experience to held-out enterprise-tool tasks; and, within Morphz, do transactionally formed Mind Frames actually constitute the Context snapshot used by held-out evaluation?
- **RQ6 — external validity:** does the complete Morphz agent retain general command-line task capability on a public task suite under the same model and environment as a reference agent?

We distinguish formal-batch raw scores from post-hoc diagnostics. A post-hoc semantic rescore never changes a stored model response or replaces the prespecified exact-schema score; it answers a different question when a schema-shape error is separable from the semantic value. If the official verifier times out or returns no result, we rerun it sequentially only against the frozen final state and unchanged official tests, without another model call, while retaining the formal raw score. Provider refusals and tasks that do not pass the official verifier remain failures; a run is excluded only when prespecified integrity checks establish that the apparatus invalidated the run. Architectural features absent from a baseline are recorded as not applicable.

ME-01 through ME-03 are small controlled mechanism studies. ME-04 tests authoritative-state commitment and fault handling. ME-05 tests nine model configurations. ME-06 is a three-fixture paired long-horizon study using the Morphz state path. ME-07 compares learned-experience transfer across three systems. ME-08 adds a public terminal-task suite. Each study addresses a distinct research question and is reported separately.

5.2 Common controls

Model-based mechanism studies used GPT-5.6 Sol through CLIProxyAPI at maximum reasoning effort, with one candidate and no fallback to another model. Each arm used an isolated state directory, database, Structured Context, and model-service configuration. We recorded the executed binary or runner commit, exact model identifier, prompts, responses, usage, and scorer artifacts. Reported results include only runs that passed the prespecified integrity checks.

ME-07 uses STATE-Bench v0.8.1 [25]. All three systems receive the same 100 canonical training trajectories per domain, 50 held-out tasks per domain, and domain tools. The tested agents, user simulator, and language-model judge all use GPT-5.6 Sol/max, with the same simulator prompt, judge prompt, and one-attempt policy. Each held-out trial starts from an isolated copy of the corresponding post-training domain state. The internal prompt, state representation, memory management, tool loop, and scheduler remain part of the system under test.

STATE-Bench Agent Learning evaluates memory through transfer to held-out tool tasks rather than through literal recall questions. The agent interacts with a simulated shopping, travel, or customer-support environment and may execute domain tools that mutate an isolated task database. The primary metric, `task_completion_pass@1`, is one only when two binary surfaces both pass: deterministic `state_requirements_met` verifies the final environment state (including the absence of a mutation when none is authorized), while judged `task_requirements_met` verifies non-state conversational and procedural requirements such as disclosure, confirmation, policy accuracy, and prohibited claims. The component averages and the 1--5 user experience (UX) score are reported separately as diagnostic secondary metrics.

ME-08 evaluates the complete 89-task Terminal-Bench 2.1 set in both arms [23,26]. Morphz and Codex CLI 0.149.1 each attempt every task once using GPT-5.6 Sol at maximum reasoning effort through the same CLIProxyAPI service, on the same Linux host and task-registry digest, with arm-internal concurrency of eight and zero retries. The paper uses the latest fixed Morphz full run and the frozen Codex reference run; they follow the same protocol but started at different times and used different model-service accounts.

ME-08 uses the official verifier's binary task judgment. When a formal run lacks a verifier result, the frozen final state is rechecked without a model call under the rule above. We apply an exact two-sided binomial/McNemar-equivalent test to Morphz-only versus Codex-only passes and report a fixed-seed task-bootstrap interval for the task-aligned difference.

5.3 Core results through ME-06

Study	Design	Result	Supported interpretation
ME-01	5 task families × append-only, structured read-only, full Morphz	15/15 strict success	Structured Context and result re-entry did not degrade tasks without capacity pressure; full Morphz also exercised SQLite persistence and transactions
ME-02	6 tasks × S-expression AST, JSON AST, Markdown program	18/18 strict success; each arm 6/6	All three renderings completed every task, supporting the feasibility of representing programs and cognitive data with the same recursive structure
ME-03	24 base/intervention episodes	nondeterministic conditions 12/12 strict; Context shifts 6/6; deterministic controls 11/12 strict and 12/12 semantically correct	Admissible cognitive values are Context constrained and need not be reducible to one fully specified deterministic transition; nondeterminism does not require sampling diversity
ME-04	8 state-commitment and fault-handling cells	8/8 passed; 989 library tests passed	Generating candidate cognition and committing authoritative state are separate Runtime responsibilities
ME-05	9 models × ME-02/03 core subsets	144/144 complete; 98/144 strict; 32/36 program final values; 104/108 semantic nondeterministic selections, and 104/104 among parseable responses	The semantic mechanism reproduced across nine model families; exact structural compliance varied by model
ME-06	3 long-horizon fixtures × controlled compaction, full Morphz	both arms 3/3 semantic success, 24/24 final fields, 3/3 unique actions	Both arms preserved final state and actions; Morphz additionally exercised transaction, recovery, and audit paths

5.4 Structured Context and result re-entry

ME-01 compared an append-only history, the Morphz structured read-only Context path with direct Context transactions disabled, and full Morphz. Five task families covered delayed reference, source authority, explicit supersession, cross-session continuity, and Context isolation. All 15 cells returned the strictly correct final action. Without capacity pressure, complete message history was sufficient for these tasks; ME-01 therefore establishes non-degradation of the structured paths rather than superiority over messages. Full Morphz additionally exercised SQLite persistence, Context projection, and transactions.

Full Morphz committed Frames through the Context transaction tool, projected them into later activations, resumed from SQLite after a process restart, shared one Context across sessions, and excluded a neighboring Context. Context transactions are recorded as not applicable for the two comparison arms because those architectures do not implement that operation.

ME-06 moved the same question to three 120-event, 12-checkpoint long-horizon fixtures. A controlled-compaction reference performed one compaction before checkpoint 6; Morphz used the configured 262,144-token input capacity without artificial pressure. Both arms achieved 3/3 semantic fixture success, all 24 final state fields, and all three unique final actions. Morphz additionally executed 40 successful Frame transactions and raw traces recorded cross-session continuity, version-conflict rereading, process restart recovery, Context isolation, and complete causal audit.

5.5 Equal-information representation

ME-02 generated three renderings from one canonical typed program IR. Six tasks covered nested sequencing, branch selection, bindings, shared references, fallback, and alternating branches. S-expression AST, JSON AST, and Markdown Program each passed 6/6 with identical mean model-request

and tool-call counts. Because all three scored 6/6, this task set has a ceiling effect: the result establishes representational feasibility rather than a ranking among formats.

Together, the three renderings show that the contribution lies in representing programs, cognitive data, and returned values through an addressable and evaluable recursive structure, rather than in one surface syntax. S-expressions are the current compact, homoiconic, and readily validated implementation.

5.6 Context-constrained nondeterministic evaluation and cross-model generality

ME-03 defined three task families in which a cognitive expression had three to six contract-valid values. The visible Context selected a valid subset; an intervention changed the relevant Frames so that the before/after valid sets were disjoint. All 12 nondeterministic episodes satisfied the strict schema and visible contract, and all six paired interventions changed the selected value into the intervention-valid set. Two repetitions often selected the same value. This is consistent with the definition: nondeterminism is a one-to-many admissible evaluation relation, not a requirement that repeated sampling exhibit high entropy.

A deterministic control specified one unique value. It achieved 11/12 strict success; the remaining response selected the correct value but serialized an array field as a string. ME-05 reports exact-schema compliance and semantic correctness separately.

ME-05 ran the same ME-02/03 core across GPT-5.6 Sol, Claude Opus 5, Grok 4.6, Gemini 3.7 Flash High, DeepSeek V4 Pro and Flash, Kimi K3-256K, GLM-5.3, and Qwen 3.8 Max Preview. All 144 cells completed. The strict aggregate was 98/144. Program final values were correct in 32/36 cells; the four missing values were Claude provider safety refusals. For nondeterministic evaluation, the prespecified exact-schema score was 72/108. A clearly labeled post-hoc diagnostic ignored basis-array/string and extra-field shape differences and rechecked the selected value against the original visible contract: 104/108 overall, and 104/104 among responses with a parseable selection. The result supports cross-family semantic feasibility while showing why deterministic parsing, validation, lowering, and failure handling remain necessary.

5.7 Runtime control of authoritative state and real-world effects

ME-04 invoked no model. Eight cells used Morphz runtime components and deterministic fault injection to test tool admission, forged or malformed program values, tampered replay state, Frame MVCC, disjoint and conflicting concurrent updates, duplicate event and transaction identities, idempotent versus non-idempotent effect recovery, continuation after a committed Context transaction, adversarial observations, and Principal/Session/Context isolation. All eight tests passed; the full library suite reported 989 passing tests.

The tests show that visible text cannot expand the runtime's tool boundary and that stale writes, replay attempts, and tested crash windows are rejected or recovered under explicit semantics.

5.8 Learned-agent comparison: STATE-Bench

ME-07 compares three end-to-end system boundaries: Morphz with Structured Context and Context transactions; Letta 0.16.8 as a complete open-source agent runtime using its native core, recall, and archival memory (an architecture originating in MemGPT [15]); and a reference agent backed by Mem0 2.0.19 add-time learning and `search(top_k=3)` [24]. The Mem0 arm uses local Qdrant, the `nomic-embed-text` embedding model, and base vector retrieval; we refer to it as the Mem0-backed vector reference.

ME-07 records two evidence layers. The primary task score compares the three complete system boundaries end to end. A read-only Morphz trace audit checks the mechanism across the training/test boundary: each domain-training Session commits all 100 training episodes through Context transactions; the post-training projection contains training-attributed active Mind Frames and explicit Relations; and every model call in the isolated held-out Session binds to that domain's final training revision. Together, these layers establish both end-to-end performance and the participation of learned structured state in held-out evaluation.

The experiment includes all three domains and all 50 held-out tasks per domain, and runs each task once across the three systems: 150 runs per arm and 450 runs overall. A paired cell is one `(domain, task)` tuple. Up to four cells may run concurrently, with their three arms executed in parallel; every run uses an isolated copy of the post-training state. Every terminal outcome is written atomically, scored failures remain zero, and no failed run is silently retried. Primary comparisons are Morphz minus Letta and Morphz minus Mem0, with task-clustered intervals, sign-flip tests, and Holm correction. Secondary measures include pass@1, requirement satisfaction, UX, tokens, wall time, training cost, and failure class.

All 450 runs reached a terminal artifact and passed the prespecified integrity checks. Morphz completed 122/150 tasks (81.33%), Letta 93/150 (62.00%), and the Mem0-backed reference 96/150 (64.00%). The diagnostic state-requirement rates were 92.67%, 79.33%, and 82.67%; task-requirement rates were 85.33%, 68.00%, and 70.00%; mean UX scores were 4.410, 3.625, and 3.833. One Morphz, seven Letta, and four Mem0 terminal failures were retained as zero.

System	Completion	State req.	Task req.	Mean UX	Terminal failures
Morphz	122/150 (81.33%)	92.67%	85.33%	4.410	1
Letta	93/150 (62.00%)	79.33%	68.00%	3.625	7
Mem0-backed reference	96/150 (64.00%)	82.67%	70.00%	3.833	4

Morphz exceeded Letta by 19.33 percentage points (task-clustered bootstrap 95% confidence interval [+10.67, +28.00]) and Mem0 by 17.33 points ([+10.00, +24.67]). Raw paired sign-flip p-values were 0.000040 and 0.000030; both Holm-adjusted values were 0.000060. These estimates concern the complete end-to-end system boundaries. The protocol holds training trajectories, held-out tasks, foundation model, domain tools, evaluator, and one-attempt policy fixed, but it intentionally leaves each system's internal prompt, memory representation, scheduler, and tool loop intact.

The read-only Morphz audit passed all 150 held-out traces. Training in each domain ended at revision 100 after 100 committed Context transactions. The three post-training projections contained 144 active Mind Frames, 395 active Relations, and 795 retired objects in total. Every held-out task's first model call used the corresponding revision-100 Context; later versions and transaction hashes formed nondecreasing, contiguous chains. Three shopping-domain held-out tasks then performed six additional Context transactions. These traces establish that consolidated training experience was active computational state for every held-out Morphz evaluation.

5.9 External terminal-task evaluation: Terminal-Bench 2.1

ME-08 uses the Terminal-Bench official verifier's binary task judgment. The paper compares the latest fixed Morphz Runtime with the frozen Codex reference run. Both attempted all 89 tasks once using GPT-5.6 Sol at maximum reasoning effort through the same model-service route, on the same Linux/amd64 host and task version, with full access, arm-internal concurrency of eight, and zero retries. The two complete runs followed the same frozen protocol but did not start concurrently.

Agent	Passes	Accuracy	Wilson 95% interval
Morphz	74/89	83.15%	[74.04%, 89.51%]
Codex CLI 0.149.1	74/89	83.15%	[74.04%, 89.51%]

The task-aligned table contains seven Morphz-only passes, seven Codex-only passes, 67 joint passes, and eight joint failures. The difference, Morphz minus Codex, is 0 percentage points; a fixed-seed task bootstrap gives a 95% interval of [-7.87,+7.87] percentage points, and the exact two-sided paired p-value is 1.0.

The two complete runs also provide a system-level engineering comparison. Provider-reported prompt-plus-completion volume was 67,387,261 tokens for Morphz and 83,361,987 for Codex, or 757,160 versus 936,652 tokens per attempted task. Morphz used 19.2% fewer reported total tokens over the identical 89-task workload. End-to-end wall time was 6,349.93 seconds (1.76 hours) for Morphz and 7,065.16 seconds (1.96 hours) for Codex, making the Morphz run 10.1% shorter. Their input-cache rates were 29.88% and 93.12%, respectively. The Morphz run executed alongside the cache-transport treatment, so wall time is reported only as a descriptive engineering observation.

Request traces localized the cache-rate difference primarily to context transport. The default path in this Morphz version encoded the complete Structured Context as one message and regenerated it on every request. The OpenAI Codex subscription route used here did not stably reuse the unchanged internal prefix of that changing message. Codex instead appended a multi-message history, leaving previously transmitted messages unchanged. We therefore compiled a separate experimental strategy, not included in the default Runtime, that divided the Structured Context into stable content and append-only deltas at the model-service boundary. It changes request encoding rather than the state model and is unnecessary when the model service can reuse an intra-message prefix or expose a suitable cache breakpoint.

To test this strategy, we ran a contemporaneous A/B over all 89 tasks. The arms started together and used the same Morphz Runtime version, GPT-5.6 Sol at maximum reasoning effort, model-service account, and full access; each task was attempted once without retries, with arm-internal concurrency of eight. The control retransmitted the full Structured Context on each request, while the treatment used append-only deltas. The raw batch scores were 72/89 and 71/89. Four tasks produced no verifier result within the 900-second limit, which included test-environment startup. A sequential recheck retained the final files and official tests and made no model calls; three tasks passed and one failed. In one additional treatment task, the final correct file was written about 57 seconds before the deadline, but the terminal reply and verifier subsequently timed out. The unchanged file passed rechecking, so it is included in task accuracy while the execution timeout remains recorded. The rechecked scores were therefore 74/89 and 73/89. The arms jointly passed 68 tasks and jointly failed 10; six were control-only passes and five treatment-only passes. The treatment-minus-control difference was -1.12 percentage points, with an exact two-sided McNemar `p=1.0`. All 178 task runs passed the prespecified persistence, execution-trajectory, and result-integrity checks, and neither arm used additional evaluation-specific prompting.

Cache transport	Passes after verifier recheck	Input-cache rate	Uncached input tokens	Input + output tokens	Wall time
Full-context reserialization	74/89	29.88%	46,285,487	67,387,261	6,349.93 s
Experimental append-only context deltas	73/89	86.27%	8,669,562	64,979,430	6,765.76 s

The experimental append-only encoding increased the input-cache rate by 56.39 percentage points and reduced uncached input by 81.27%. Provider-reported input plus output fell by only 3.57%, while treatment output tokens increased by 32.67%, reasoning tokens by 44.09%, and wall time by 6.55%. The transport experiment therefore substantially improved cache reuse but did not reduce end-to-end time in this run.

The contemporaneous A/B ran two arms at concurrency eight. On the host with 16 logical CPUs and 61.52 GiB of memory, whole-host memory use averaged 5.14 GiB and peaked at 9.62 GiB; the one-minute load average was 5.439 and peaked at 25.648. These samples cover the whole host and both arms and do not estimate the resource use of either agent alone.

The two same-protocol full runs each scored 74/89, but the difference interval was wide and crossed zero. They therefore establish neither superiority nor equivalence or non-inferiority; statistical-power boundaries are reported in Section 8.4.

6. What the Results Establish

6.1 The implemented mechanism is empirically reachable

The combined evidence supports the paper's primary mechanism claim. Models can evaluate a recursive structured term representation; a cognitive expression can admit several Context-valid values and change under a relevant Context intervention; returned tool and cognitive values can become bindings or persistent Frames; and a deterministic runtime can keep candidate cognition separate from authoritative state and effects. These paths are not limited to one model family, although exact schema and trace compliance vary.

Structured Context also enables capabilities that an append-only message list does not directly provide as runtime primitives: stable cognitive-object identity, explicit provenance and supersession, versioned transaction, cross-session state, process-independent recovery, Context isolation, and causal audit. ME-01 and ME-06 show that obtaining these capabilities did not reduce final-task correctness in the paired tasks. ME-07 supplies broader held-out evidence that this state can carry learned experience: Morphz outperformed both public comparison systems on 150 matched tasks, while the trace audit verified that every Morphz task began from the trained Mind Frame projection.

6.2 Mechanism evidence and system-level evidence

ME-01--ME-06 isolate representation, state re-entry, nondeterministic evaluation, authority, cross-model execution, and long-horizon state handling. ME-07 compares learned-experience transfer across three systems and pairs the task score with a trace audit of the Morphz Mind Frame path. ME-08 compares Morphz and Codex CLI on terminal tasks. This layered design connects the computational mechanism to end-to-end behavior without collapsing distinct research questions into one score.

The current `infer` interface returns ordinary typed values or a first-class `Program<T,E>` to a deterministic plan. Program admission, durable child execution, and recovery are implemented; what remains open is whether this path improves dynamic real-world tasks relative to fixed scripts or ordinary replanning.

6.3 How the public-task results should be used

ME-07 and ME-08 answer complementary external-validity questions. ME-07 measures whether learned experience transfers to held-out stateful enterprise-tool tasks, and its trace audit connects the end-to-end score to an actually trained Structured Context. Terminal-Bench compares the terminal-task capabilities of Morphz and Codex CLI under a shared model and environment. ME-01--ME-06 supply the controlled mechanism evidence that relates these system outcomes to the proposed computation model.

Across the completed studies, Morphz implements persistent and transactional cognitive state and demonstrates Context-constrained nondeterministic evaluation across model families. In ME-07, Morphz completed 81.33% of held-out tasks, versus 62.00% for Letta and 64.00% for the Mem0-backed reference. The 150/150 trace audit confirms that the trained Mind Frame state participated in every Morphz held-out task. In Terminal-Bench full runs using the same model, host, and frozen protocol, the latest fixed Morphz Runtime and Codex CLI 0.149.1 each scored 74/89; this one-attempt design did not resolve a stable system difference.

7. Related Work

7.1 Programs as model scaffolding

Program-Aided Language Models and Program of Thoughts ask a model to generate a conventional program that delegates exact computation to an interpreter [4,5]. LMQL, SGLang, and DSPy make model calls programmable through constrained queries, structured execution, or declarative compilation [7–9]. These systems primarily program the process around a model call. Morphz instead treats the evolving cognitive state and its operations as values in an agent-owned term domain.

BINDER is the closest conceptual precedent [6]. It extends SQL or Python with calls to language-model capabilities, uses a model both to parse a task into a program and to answer semantic subproblems during execution, and stores returned values in variables compatible with the surrounding symbolic grammar. Morphz adopts the importance of continued symbolic execution after an LM call, but extends the unit of computation from a task program to a persistent cognitive subject. Frames, relations, observations, candidate programs, and returned values share the same stateful loop across activations and sessions. The completed controlled studies show that this persistent loop is executable and adds transactional state semantics on the tested tasks.

PLSemanticsBench evaluates whether a language model can act as the interpreter of a conventional programming language from formal operational semantics [10]. It reports weaker behavior under nonstandard semantics and difficulty with exact execution traces. Morphz addresses a different division of labor. Operational semantics that are fully specified need not be simulated by the model; the model is used where the symbol's meaning is intentionally incomplete and context dependent.

7.2 Tool-using agents and verified effects

ReAct interleaves reasoning and actions [2], while Toolformer studies learned API-use decisions [3]. Reflexion persists verbal feedback, and Voyager builds a reusable code-skill library [11,12]. These approaches demonstrate environmental feedback and result reuse, but their persistent cognitive values are primarily text or code artifacts rather than members of a shared context term algebra.

ToolGate maintains explicit symbolic world state and uses Hoare-style preconditions and postconditions to gate and verify tool calls [13]. Its runtime commitment boundary is closely aligned with Morphz's separation between a semantic candidate and a committed state change or real-world effect. ToolGate

focuses on the logical safety of tool execution. Morphz additionally studies how cognitive state is represented, how nondeterministic cognitive symbols are evaluated, and how returned cognitive values re-enter later computation.

7.3 Memory and cognitive architectures

Generative Agents use observation, reflection, and retrieval to support persistent social behavior [14]. MemGPT virtualizes the physical context window through hierarchical memory [15]. CoALA provides a conceptual organization of language agents around working memory, long-term memory, decision procedures, and learning [16]. AIOS applies operating-system abstractions to agent scheduling, context, memory, and tools [17].

Agentic Memory exposes storage, retrieval, update, summarization, and forgetting as actions learned by the agent [18]. StructMem preserves event-level bindings and constructs cross-event relations for long-horizon conversational reasoning [19]. These systems support the broader move from passive retrieval to active memory management. Morphz's narrower distinction is that context is not an auxiliary memory subsystem attached to a message loop. It is the state against which cognitive programs are evaluated, and its objects can be directly referenced as operands.

7.4 LLM computers and virtual machines

L2MAC describes a stored-program automatic computer built from language-model agents, an instruction registry, a file store, and a control unit [20]. Playbooks AI describes an assembly language, compiler, runtime, and common-language-runtime analogy for AI programs [21]. Together, these projects define the broader “LLM as computer/VM” design space.

Morphz focuses on an LLM evaluating recursive cognitive symbols against agent-owned structured state, with cognitive data and candidate programs sharing a term domain, returned values re-entering later computation, and a deterministic Runtime controlling state commitment and real-world effects. Context dependence explains why some cognitive terms require the LLM and identifies its computational role as a nondeterministic cognitive evaluator. Morphz also implements bounded Program-valued re-entry for Runtime-validated programs rooted at either `eval` or `infer`; its effects on real dynamic tasks remain an empirical question.

7.5 Symbolic expressions

McCarthy's original Lisp work established recursive symbolic expressions as a representation for programs and data [1]. Morphz neither revives Lisp as a general-purpose language nor claims S-expression syntax as an invention. The historical idea becomes newly useful because a language model can assign contextual meaning to cognitive symbols that a conventional evaluator cannot execute without exhaustive formalization.

8. Limitations and Threats to Validity

8.1 Construct validity

ME-01 through ME-03 isolate narrow mechanism properties whose scale and difficulty are insufficient for estimating a stable effect size. Their results establish executability without an observed regression. ME-04 tests deterministic boundaries rather than model intelligence. ME-06 combines long-term state with

concurrency and recovery, but only three project-shaped fixtures exist. No single score measures "persistent cognition," so the paper reports state accuracy, action accuracy, contract validity, runtime events, and external task rewards separately.

A structured representation can improve auditability without improving answer accuracy. Conversely, a system-level benchmark can change because of prompting, tool policy, convergence behavior, or provider reliability rather than Context. ME-07 reduces this ambiguity by pairing the end-to-end score with an execution-trace audit, but it still does not randomize the internal memory mechanism while holding all other agent internals constant. We therefore describe its advantage as a system-level learned-transfer result, and avoid attributing either ME-07 or Terminal-Bench differences exclusively to Structured Context.

8.2 Internal validity

Every reported result is bound to an exact protocol, binary hash, runner commit, and scorer version. A run is excluded only when prespecified integrity checks establish that the evaluation apparatus invalidated it; its raw artifacts remain in the audit record.

Two post-hoc semantic diagnostics are reported. ME-05 separates schema-shape errors from valid cognitive selections; ME-06 accepts semantically equivalent decision rules and a concrete evidence-event identifier equivalent to its source alias. Neither diagnostic changes a response or replaces the exact-schema score defined before execution. Their rules are deterministic and applied to every arm, but their evidence is weaker than a semantic scorer defined before execution.

ME-08 separately applies a zero-model sequential recheck when the official verifier returned no result. The recheck preserves the frozen final files, task digest, and official tests, while the formal-batch raw scores remain separately available in the audit record. Codex timeout tasks were reviewed under the same rule.

8.3 External validity

ME-01 through ME-03 use synthetic tasks, and ME-05 expands to nine model configurations while retaining the same small task core. ME-06 contains only three paired long-horizon fixtures and therefore cannot establish broad memory generalization. ME-07 covers three simulated enterprise domains from STATE-Bench, 150 held-out tasks, and two public comparison systems; all three systems use GPT-5.6 Sol/max and run each task once. ME-08 uses the complete public task set and Codex CLI as the comparator, also with one attempt per task; both groups share the same CLIProxyAPI service configuration and policy constraints.

The experiments do not cover real multi-month organizational use, robotics, cross-user privacy, or accumulated false beliefs. They also do not establish that the same state model is optimal for every agent category.

8.4 Statistical power

ME-01 has five paired task families, ME-02 six tasks per representation, ME-03 six paired interventions, and ME-06 three paired fixtures. These small controlled studies are insufficient to estimate statistically significant system differences. ME-05 reports cross-model counts but has no repeated-sampling estimate for each model. ME-07 provides 150 paired held-out tasks and reports task-clustered bootstrap intervals plus paired sign-flip tests with Holm correction; its intervals exclude zero, but one attempt per task cannot estimate within-task sampling variance, and its judge-based task-requirement and UX measures have not been independently calibrated by human raters. ME-08 provides 89 task-aligned samples: its exact two-

sided p-value is 1.0 and its task-bootstrap difference interval is [-7.87,+7.87] percentage points. One attempt per task still cannot estimate within-task sampling variance, and the interval is too wide for an equivalence or non-inferiority conclusion. The subsequent experimental cache-transport A/B also contains 89 paired tasks; after verifier rechecking, the treatment-minus-control difference was -1.12 percentage points with an exact two-sided McNemar `p=1.0`. This single paired run likewise does not establish equivalence between the transport methods.

8.5 Efficiency and measurement validity

Provider-reported token fields and reasoning-token semantics differ across protocols, so the complete-agent efficiency comparison uses ME-08 full runs with the same model, task set, host, and frozen protocol. Morphz used 19.2% fewer prompt-plus-completion tokens and 10.1% less end-to-end wall time. Because the Morphz run executed alongside the cache-transport treatment, wall time is descriptive only.

The subsequent 89-task contemporaneous A/B enabled only the experimental cache transport. It raised the input-cache rate from 29.88% to 86.27% and reduced uncached input by 81.27%; verifier-rechecked passes were 74 and 73 with paired `p=1.0`. The treatment made fewer requests but generated 32.67% more output tokens and 44.09% more reasoning tokens, while wall time increased by 6.55%. Cache reuse, generation volume, and end-to-end time are distinct measurements.

8.6 System maturity and terminology

Morphz remains an evolving Runtime. Broad Context projections, repeated maintenance, Frame growth, incorrect consolidation, ownership conflicts, and recovery of partially delivered dialogue still require engineering work. Deterministic Runtime control over state commitment and real-world effects reduces some failure modes but does not make model-generated cognition correct or safe by construction.

The phrase “nondeterministic cognitive evaluation” denotes a context-conditioned one-to-many admissible relation.

9. Remaining Work

9.1 Scaling the mechanism studies

Larger mechanism studies should use unseen fixtures in which old evidence, source authority, delayed action, competing updates, and sustained Context pressure interact, and compare Structured Context with a strong modern compaction baseline. A direct ablation inside Morphz could vary Mind Frame projection or consolidation policy while holding the remaining Runtime fixed, thereby testing the causal contribution of these mechanisms. ME-07's judge-based measures also require calibration against a blinded human sample; repeated attempts would estimate within-task variance.

9.2 Dedicated efficiency evaluation and Runtime optimization

A dedicated efficiency study should repeat matched terminal-task workloads while varying projection granularity, Frame decomposition, tool-schema size, maintenance trigger policy, and unchanged-state suppression. It should report total and uncached-equivalent input, cached input, output, verified success, maintenance calls, wall time, and—when real API billing is used—cost. The objective is not compression for its own sake; it is preserving stable identity, provenance, and transaction semantics while improving the measured end-to-end tradeoff.

9.3 Evaluating Program-valued runtime adaptation

The implementation already allows an `infer` to return one raw Yao candidate program after observing the environment; Runtime then performs parsing, type/effect/capability admission, canonicalization, provenance binding, durable child execution, and recovery join. Future experiments should compare Program-valued re-entry with fixed scripts, ordinary replanning, and one-way plan generation, measuring program-synthesis reliability and the effect of re-entry on subsequent behavior. Existing tests establish the interface, isolation, and recovery properties; dynamic-adaptation effects on real tasks remain to be measured.

9.4 Learning the evaluator and Context policy

ME-05 shows that semantic feasibility is broad while exact contract compliance varies. Future work should separate in-context operator learning, supervised structure generation, preference optimization for transaction timing, and policies for derive/revise/protect/consolidate/retire. Training should preserve the Runtime boundary for state commitment and real-world effects rather than teaching the model to describe a candidate as though it had already been committed.

9.5 Longitudinal deployment and release

Longer studies must observe false-belief accumulation, provenance decay, cross-user information flow, model migration, multi-session coordination, and partial-effect recovery over weeks rather than checkpoints. A public artifact should include the exact prompts, fixtures, model and provider configurations without credentials, scorer source, raw immutable results, checksums, statistical notebooks, and exact runtime binaries or reproducible build identities.

10. Discussion

10.1 A persistent agent as a computational subject

The architecture suggests a definition of persistence that does not depend on uninterrupted inference. A Morphz agent remains the same subject because its cognitive objects, authoritative events, and continuations survive the evaluator. This permits a smaller or differently trained model to take over without treating the transition as a new agent, provided that both models can consume the same context and term contract.

This view also explains why Structured Context underlies several product features. Concurrent threads need addressable shared state and conflict semantics. Multiple sessions need a context independent of any one transcript. Context self-maintenance requires that frames and observations be first-class mutable objects. Model portability requires an external cognitive representation rather than state that exists only in one provider's hidden conversation.

These are architectural implications, not yet proof that Morphz implements each feature optimally.

10.2 A learnable cognitive symbol evaluator

A conventional virtual machine fixes its instruction semantics in a specification and implementation. Morphz retains that deterministic mode for operations whose semantics are fully specified, while using an LLM for cognitive symbols whose values are distributed across the operator kernel, context, observations, examples, and model parameters. The cognitive evaluator can improve through prompting, examples, fine-tuning, or preference optimization, while state changes and real-world effects remain controlled by explicit rules.

One useful design taxonomy is:

- **deterministic operators with fully specified semantics**, which must be executed exactly by the Runtime;
- **bounded nondeterministic operators**, which require model judgment but return a structure with testable acceptance conditions;
- **context-dependent cognitive operators**, whose value depends substantially on current Context and world knowledge.

This taxonomy avoids two unproductive extremes: asking the model to simulate every deterministic transition, and attempting to encode all cognitive semantics in conventional code. The current typed `eval` / `infer` connection makes part of the boundary executable, but the paper's primary object is the nondeterministic evaluation of cognitive terms against structured state.

10.3 Bounded Program-valued re-entry and dynamic Runtime adaptation

The current Runtime supports first-class `Program<T,E>` results. After observing the environment, an `infer` may return one raw Yao program rooted at `(eval ...)` or `(infer ...)`. Runtime does not execute that source directly: it parses, type-checks, preserves evaluation ownership, bounds effective effects and capabilities, canonicalizes, hashes, binds provenance, and persists the candidate. Explicit `run` then creates a durable child execution: an `eval` root advances as a Runtime-owned child Plan, whereas an `infer` root immediately dispatches a formal child Evaluation. Both rejoin their declared terminal result into the parent.

This mechanism permits bounded dynamic transfers between the two evaluators while Runtime budgets constrain capabilities, Program nesting, inference, tool use, and aggregate child work. It provides an executable basis for robotics and long-running automation in which useful behavior cannot always be enumerated in advance. Current evidence verifies implementation, isolation, and recovery boundaries; real-task synthesis reliability, benefit relative to fixed scripts or ordinary replanning, and end-to-end behavior around non-idempotent external effects remain empirical questions.

10.4 Why S-expressions, and why they may not win

S-expressions currently provide a small grammar, explicit trees, direct recursive composition, and a natural representation of code as data. They make the root operator easy to identify and can express binding, scope, sequence, branch, and frame structure without a separate parser generator.

JSON has stronger ecosystem support and familiar schema tooling. Prose matches pretraining. A typed AST may provide better validation. The Morphz computation model does not depend on S-expression surface syntax; if another representation is more reliable or efficient in equal-information experiments, the system can retain nondeterministic cognitive evaluation, Structured Context, result re-entry, and deterministic commitment while replacing the surface representation.

10.5 Safety and auditability

Structure does not make an agent safe by itself. It does make boundaries explicit enough to inspect. A candidate term can be validated, a frame revision can carry provenance and version, a tool call can be checked against capability policy, and an observation can be tied to a real execution event. Future work must add information-flow labels, source trust, taint propagation, cross-user isolation, and human approval policies for high-impact operations.

The relevant safety property is not deterministic cognition. It is that a deterministic Runtime controls which state changes are committed and which real-world effects are executed.

11. Conclusion

Morphz explores nondeterministic cognitive symbol evaluation as a foundation for persistent language-model agents. An LLM evaluates recursive cognitive terms against Structured Context whose frames, relations, observations, bindings, provenance, and versions survive any one conversation or model call. Programs and cognitive data share a recursive term domain; returned structures can become local bindings or durable frames and relations. A deterministic Runtime validates candidate programs and Context transitions and controls state commitment and real-world effects.

The implementation and ME-01--ME-06 results establish the proposed computation as an executable system. Models consumed structured programs in three surface representations, real observations re-entered local dataflow, Context interventions changed admissible cognitive values, and nine model configurations completed the same mechanism matrix. The deterministic Runtime passed eight authority and fault tests. In the long-horizon controlled study, Morphz matched a controlled-compaction reference on every final state field and action while exercising transactions, sessions, version-conflict recovery, restart, isolation, and audit. ME-07 extends the evidence to learned-experience transfer: Morphz completed 81.33% of 150 held-out stateful tasks, versus 62.00% for Letta and 64.00% for a Mem0-backed reference, and a 150/150 trace audit verified that every Morphz task began from its domain's trained revision-100 Structured Context.

The completed studies show that Morphz implements persistent, addressable, transactional cognitive state. In ME-07, Morphz completed 81.33% of held-out tasks, versus 62.00% for Letta and 64.00% for the Mem0-backed reference. In Terminal-Bench full runs using the same model, host, and frozen protocol, the latest fixed Morphz Runtime and Codex CLI 0.149.1 each scored 74/89 (83.15%); the task-aligned difference was 0 percentage points with a 95% bootstrap interval of [-7.87,+7.87]. Morphz used 19.2% fewer provider-reported prompt-plus-completion tokens, and its observed end-to-end wall time was 10.1% shorter. A subsequent full-workload A/B showed that the experimental transport strategy could raise the input-cache rate from 29.88% to 86.27%; after verifier rechecking, the two transport methods scored 74/89 and 73/89. These results connect the cognitive-state mechanism to controlled behavior, experience transfer, and terminal-task execution; their validity boundaries are summarized in Section 8.

Bounded Program-valued re-entry is implemented and provides an executable path toward dynamic Runtime adaptation; its reliability and benefit in real dynamic tasks remain to be evaluated. The paper's central contribution is the combination of nondeterministic cognitive-symbol evaluation, agent-owned structured computational state, structured result re-entry, and deterministic Runtime control over state commitment and real-world effects.

Appendix A. End-to-End Example

The following schematic expression reads external evidence, asks the model to evaluate a context-dependent conflict, commits the returned structure as a revision, and references the committed frame in the final explanation:

```
(eval
  (seq
    (bind contract
      (call read_file (path "contract-v2.md")))

    (bind decision
      (infer
        (captures contract)
        (returns String)
        (seq
          (bind criterion
            "compare the new contract evidence, the existing delivery policy, and the user's
risk boundary")
          "use criterion and contract to return a revised delivery policy with reasons")))

    (call context_tx
      (base-version (ref context.version))
      (revise
        (frame-id "delivery-policy")
        (content decision)
        (derived-from contract.observation-id)))

    (reply
      (explain (ref frame:delivery-policy)))))
```

The `decision` value is not consumed only by the immediate reply. It is committed under a stable frame identifier with provenance and version metadata. A later session, objective, or evaluator can use `frame:delivery-policy` as a structured operand.

Appendix B. Claim-Evidence Matrix

Claim	Evidence status	Current evidence	Missing evidence
Structured Context and durable result re-entry introduce no visible regression on simple tasks without capacity pressure	Controlled mechanism study	ME-01: five task families across three arms, 15/15 strict success; the full Morphz arm exercised SQLite, Context projection, `context_tx`, cross-session access, process restart, and Context isolation	Larger held-out tasks and a design with a prespecified non-inferiority margin
Programs and cognitive data can use the same recursive representation without depending on S-expression surface syntax	Controlled mechanism study	ME-02: S-expression AST, JSON AST, and Markdown Program each scored 6/6 from the same canonical intermediate representation	More complex programs and representation-specific reliability and efficiency measurements
The admissible values of a nondeterministic cognitive symbol are Context-constrained and may include multiple valid results	Controlled mechanism study	ME-03: 12/12 nondeterministic-condition contracts; 6/6 paired Context interventions changed the admissible result; deterministic controls were 11/12 strict and 12/12 semantically correct	Larger unseen operator families and semantic scoring defined before execution

Claim	Evidence status	Current evidence	Missing evidence
State commitment and real-world effects remain Runtime-controlled while candidate cognition is model-generated	Deterministically verified	ME-04: 8/8 commitment and fault-handling cells passed; capability, version, replay, effect, and recovery boundaries have dedicated tests	External adversarial safety evaluation and formalized end-to-end properties
The mechanism executes across model families, while exact structural-contract reliability varies by model	Cross-model controlled study	ME-05: 144/144 cells completed across nine models; 98/144 strict; 32/36 Program final values; 104/108 nondeterministic-evaluation semantic diagnostics; four additional provider refusals	Repeated samples, a larger task core, and provider-independent deployments
Long-horizon structured state can preserve final outcomes while adding transaction and recovery semantics	Long-horizon controlled study	ME-06: controlled compaction and full Morphz each achieved 3/3 fixtures, 24/24 final fields, and 3/3 actions; Morphz executed 40 real Context transactions plus conflict reread, restart, isolation, and audit paths	More fixtures, sustained Context pressure, and a stronger statistical design
Trained structured cognitive state can transfer experience to held-out stateful tasks	System-level external comparison plus mechanism trace	ME-07: Morphz 122/150 (81.33%), Letta 93/150 (62.00%), and the Mem0-backed reference 96/150 (64.00%); all 150 Morphz tasks began from their audited domain-specific revision-100 trained Context, with 144 active Mind Frames and 395 Relations in total across the three post-training domain Contexts	Blinded human calibration, within-task repeated samples, and a within-Morphz causal ablation of consolidation/projection
In same-model, same-task, same-host, same-frozen-protocol Terminal-Bench full runs, the single-run Morphz-Codex score difference was not resolved as a stable system difference	Single-run external comparison	ME-08: the latest fixed Morphz Runtime and Codex CLI 0.149.1 both scored 74/89; difference 0 pp, task-bootstrap 95% interval [-7.87,+7.87], exact paired `p=1.0`	Repeated attempts for within-task variance; a prespecified margin if equivalence or non-inferiority is claimed
In same-protocol Terminal-Bench full runs, Morphz used fewer reported total logical tokens and less end-to-end time	Observed descriptor	ME-08: 67.39M tokens for Morphz versus 83.36M for Codex, 19.2% fewer; end-to-end wall time was 6,349.93 versus 7,065.16 seconds, 10.1% shorter	Repeated matched trials and mechanism-specific ablations
Experimental append-only transport can increase input-cache reuse on a complete task workload	Contemporaneous full-workload A/B plus verifier recheck	The 89-task control and treatment scored 74/89 and 73/89 after rechecking, with paired `p=1.0`; input-cache rate rose from 29.88% to 86.27%, uncached input fell by 81.27%, and wall time increased by 6.55%	Repeated attempts, additional service routes, and assessment of whether this adaptation should enter the default path
The same complete typed Yao BODY can be evaluated under `eval` or `infer` and rejoin its parent Plan	Implemented and observed	Owner-symmetric BODY parsing and type checking, the source-level `captures` disclosure boundary, durable child Evaluation, uniform declared-type decoding, binding tests, and live-model smokes	Broader cognitive workflows and fault-injection repetitions

Claim	Evidence status	Current evidence	Missing evidence
An `infer` result can become a new executable program and rejoin its parent without redeployment	Deterministically verified	Raw-Yao Program results, admission of `Program<T,E>` values rooted at either `eval` or `infer`, type/effective-effect/capability bounds, owner-directed dispatch, content hashing, caller-local isolation, durable child execution, shared nesting budgets, and restart-recovery tests	Real-model dynamic-adaptation tasks, repeated samples, and end-to-end external-effect studies

Appendix C. Reproduction Entry Points

The following are source-level entry points. ME-07 and ME-08 correspond to exact commits of the evaluation repository. Full reproduction also requires the protocol, model and provider configuration, Runtime binary, raw outputs, and scorer version recorded in the paper and artifact manifests.

```
# ME-01, ME-02, ME-03, and ME-06 mechanism studies
cargo run -q -p morphz-evals --bin me01_context_reentry_eval -- --help
cargo run -q -p morphz-evals --bin me02_representation_eval -- --help
cargo run -q -p morphz-evals --bin me03_bounded_open_eval -- --help
cargo run -q -p morphz-evals --bin me06_long_horizon_eval -- --help

# ME-04 authoritative-state commitment and fault-handling tests; ME-05 cross-model
diagnostic
cargo test -p morphz --lib
python3 morphz-evals/tools/analyze_me05_results.py --help

# ME-07 at evaluation-source commit e32d3e165d35a69451d73845470641dfd02ba156
python3 -m benchmarks.state_bench.v2.run_public_systems_formal --help
python3 -m benchmarks.state_bench.v2.summarize_public_systems_formal --help
python3 -m benchmarks.state_bench.v2.audit_morphz_mind_frame_transfer --help

# ME-08: Morphz Runtime commit 89adf739454da52bce2b35b00fb9e8fa050c5557;
# Morphz control infrastructure f8ba6bbdfacdf59dad8e226c48e26909a6237a9b;
# Codex comparison infrastructure a226bfef1b555e2d83fa4b3ce6d90790bc522705
# latest fixed Morphz versus frozen Codex derived evidence commit 63a437e;
# symmetric Codex timeout recheck evidence commit 4bcc79f;
python3 -m benchmarks.harbor.run_benchmark --help
python3 -m benchmarks.harbor.run_codex_comparison --help
python3 -m benchmarks.harbor.run_me08_postfix_all89_morphz --help
python3 -m benchmarks.harbor.summarize_me08_full89_pair --help

# ME-08 cache-transport A/B: Runtime commit 89adf739454da52bce2b35b00fb9e8fa050c5557;
# frozen-evidence commit 011cec8; verifier-recheck evidence commit
b38a3a70359374e260308d2e6466ed61f66b0251;
# arm strategies, binary hashes, and recheck results are pinned in the manifest
```

A reproduction should pin the repository commit, model identifier, provider configuration, seeds or the absence of seed support, raw JSON outputs, scorer version, and binary hash. A release artifact should neither depend on local temporary paths nor contain credentials.

References

[1] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960. https://doi.org/10.1145/367177.367199

[2] Shunyu Yao, Jeffrey Zhao, Dian Yu, et al. ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*, 2023. https://arxiv.org/abs/2210.03629

[3] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, et al. Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*, 2023. https://arxiv.org/abs/2302.04761

[4] Luyu Gao, Aman Madaan, Shuyan Zhou, et al. PAL: Program-Aided Language Models. *ICML*, 2023. https://arxiv.org/abs/2211.10435

[5] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *TMLR*, 2023. https://arxiv.org/abs/2211.12588

[6] Zhoujun Cheng, Tianbao Xie, Peng Shi, et al. Binding Language Models in Symbolic Languages. *ICLR*, 2023. https://arxiv.org/abs/2210.02875

[7] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting Is Programming: A Query Language for Large Language Models. *PLDI*, 2023. https://arxiv.org/abs/2212.06094

[8] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, et al. SGLang: Efficient Execution of Structured Language Model Programs. *NeurIPS*, 2024. https://arxiv.org/abs/2312.07104

[9] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *ICLR*, 2024. https://arxiv.org/abs/2310.03714

[10] Aditya Thimmaiah, Jiyang Zhang, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. LLMs Lean on Priors, Not Programming Language Semantics. *ICML*, 2026. https://arxiv.org/abs/2510.03415

[11] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS*, 2023. https://arxiv.org/abs/2303.11366

[12] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, et al. Voyager: An Open-Ended Embodied Agent with Large Language Models. *TMLR*, 2024. https://arxiv.org/abs/2305.16291

[13] Yanming Liu, Xinyue Peng, Jiannan Cao, et al. ToolGate: Contract-Grounded and Verified Tool Execution for LLMs. *Findings of ACL*, 2026. https://doi.org/10.18653/v1/2026.findings-acl.470

[14] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. *UIST*, 2023. https://arxiv.org/abs/2304.03442

[15] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. 2023. https://arxiv.org/abs/2310.08560

[16] Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive Architectures for Language Agents. *TMLR*, 2024. https://arxiv.org/abs/2309.02427

[17] Kai Mei et al. AIOS: LLM Agent Operating System. *COLM*, 2025. https://arxiv.org/abs/2403.16971

[18] Yi Yu, Liuyi Yao, Yuexiang Xie, et al. Agentic Memory: Learning Unified Long-Term and Short-Term Memory Management for Large Language Model Agents. *ACL*, 2026. https://doi.org/10.18653/v1/2026.acl-long.981

[19] Buqiang Xu, Yijun Chen, Jizhan Fang, et al. StructMem: Structured Memory for Long-Horizon Behavior in LLMs. *ACL*, 2026. https://doi.org/10.18653/v1/2026.acl-short.12

[20] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. L2MAC: Large Language Model Automatic Computer for Extensive Code Generation. *ICLR*, 2024. https://arxiv.org/abs/2310.02003

[21] Playbooks AI. Language Runtime, PBAsm, Compiler, and Runtime Documentation. Accessed August 4, 2026. https://docs.runplaybooks.ai/

[22] Di Wu, Zixiang Ji, Asmi Kawatkar, Bryan Kwan, Jia-Chen Gu, Nanyun Peng, and Kai-Wei Chang. LongMemEval-V2: Evaluating Long-Term Agent Memory Toward Experienced Colleagues. 2026. https://arxiv.org/abs/2605.12493

[23] Mike A. Merrill et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *ICLR*, 2026. https://arxiv.org/abs/2601.11868

[24] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. 2025. https://arxiv.org/abs/2504.19413

[25] Microsoft. STATE-Bench: Stateful Agent Task Evaluation Benchmark, version 0.8.1. 2026. Accessed August 26, 2026. https://github.com/microsoft/STATE-Bench

[26] The Terminal-Bench Team. Terminal-Bench 2.1: A Revision of Terminal-Bench 2.0. 2026. https://www.tbench.ai/news/terminal-bench-2-1

[27] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975. https://doi.org/10.1145/360933.360975