A-PXM: A Program Execution Model for Agentic AI

Anonymous Author(s)

Abstract

Agentic artificial intelligence (AI) systems increasingly run multi-step workflows that interleave large language model (LLM) calls, tool I/O, and memory operations, yet most frameworks execute these workflows as opaque, control-driven scripts. This hides data dependencies and state, preventing whole-workflow optimization and principled scheduling.

We present A-PXM (Agent Program Execution Model), a program execution model for agentic AI that provides a universal execution substrate between orchestration and LLM serving. A-PXM specifies (1) an Agent Abstract Machine (AAM), a formal state model capturing beliefs, goals, and capabilities with a tiered memory hierarchy; (2) a typed Agent Instruction Set (AIS), a portable intermediate representation (IR) implemented as an MLIR (Multi-Level Intermediate Representation) dialect spanning memory, latency-typed LLM calls (Ask/Think/Reason, annotated with expected latency classes), tools, control flow, synchronization, and native plan/reflect primitives; and (3) a dataflow execution model that schedules operations by data availability. A reference compiler/runtime executes AIS graphs and exports structured telemetry. Across ten workloads, A-PXM exposes parallelism opportunities (up to 10.37×, i.e., about one-tenth the time on multi-agent coordination), enables IR-level optimizations such as call fusion (1.29× lower latency by merging dependent call chains into fewer application programming interface (API) calls), and catches type errors before execution. Gains derive from critical-path compression and call-count reduction, not faster model inference.

CCS Concepts

• **Software and its engineering** → **Compilers**; **Runtime environments**; • **Computing methodologies** → *Concurrent computing structures.*

Keywords

agentic AI, execution substrate, program execution model, instruction set architecture, dataflow, compiler, runtime, LLMs

1 Introduction

Agentic AI systems are increasingly implemented as multi-step workflows that interleave large language model (LLM) calls, tool I/O, and memory operations (saving and retrieving state across steps). Recent agentic scientific workflows [36] and AI platforms [16, 22] show LLM agents coordinating instruments, simulation codes, and data services. Yet execution remains largely ad hoc: expensive, stochastic calls are chained without explicit state, a stable intermediate representation (IR), or principled scheduling.

These pressures are rising. Reasoning models internalize multi-step planning, agent workloads can consume millions of tokens, and failures compound across long call chains [5]. Multi-agent systems further stress coordination and resource management [2]. Together, these trends demand execution semantics and verification, not just better orchestration.

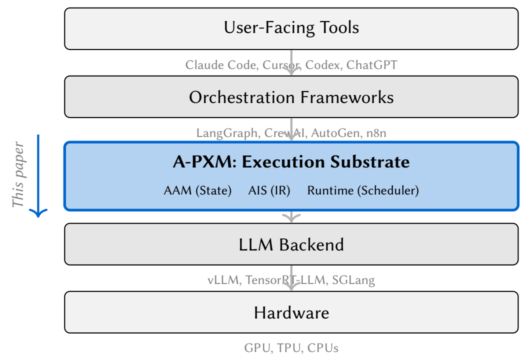


Figure 1: The agentic software stack. A-PXM provides a formal execution substrate between orchestration frameworks and LLM serving, analogous to LLVM IR enabling whole-program optimization across language frontends.

This pattern recreates a familiar architectural limitation, now at the workflow level. Popular frameworks such as LangChain and LangGraph [17] encourage developers to compose agents as linear sequences of LLM and tool invocations. Whether the task is travel planning or a scientific workflow on a high-performance computing (HPC) cluster, the control flow typically serializes reasoning and I/O: think, call a tool, wait, think again. This "call-at-a-time" script resembles the classic von Neumann bottleneck and Backus's critique of "word-at-a-time" programming [4] (Section 2). We refer to the agent-side version as the *Agentic von Neumann bottleneck*. It hides data dependencies, stacks latency across even mildly complex workflows, and compounds error probabilities as reasoning chains lengthen [5, 18].

Developers specify an *outer plan*: the workflow graph of operations they author. At runtime, the model constructs an *inner plan*: step-by-step reasoning used to choose tools, form arguments, and interpret results. Outer and inner plans must agree on state, dependencies, and effects, but today the inner plan is typically untyped natural language hidden inside prompts. Without a shared execution substrate, small prompt/tool changes can yield large global behavior changes, and opportunities for parallel execution and whole-workflow optimization are lost.

To address this gap, we propose A-PXM, a Program Execution Model (PXM), a formal contract between a program and its execution substrate that specifies state, operations, and scheduling [7], designed to make agent state and data dependencies explicit so that entire workflows can be verified, scheduled, and optimized. Figure 1 situates A-PXM in the agentic software stack and makes the missing contract explicit. Command-line interface (CLI) tools provide model access without formal execution semantics, while orchestration frameworks such as LangGraph [17] coordinate workflows but treat agent nodes as opaque units, precluding static analysis. Even LLMCompiler [14], which introduces directed acyclic graph (DAG) planning for parallel tool calls, lacks the explicit state model and typed IR needed for whole-program optimization. What is missing is a *universal execution substrate* that separates orchestration (*what* an agent does) from execution (*how* it runs), analogous to how LLVM IR provides a language-agnostic target for compiler optimization across diverse frontends [19]. The key takeaway is that, with an explicit IR and state model, we can verify and optimize the entire workflow rather than treating each agent node as a black box.

Addressing the agentic bottleneck requires treating agent behavior as a *program*. Execution-model papers are compelling not merely because they are "faster than X," but because they define a contract that exposes where a compiler/runtime may safely reorder work, overlap latency, fuse operations, or enforce policies across implementations and platforms [13, 24, 40]. In agentic AI, that contract must make state, dependencies, and effects explicit so the system can compress the critical path (by overlapping independent operations), reduce call count (via call fusion), fail early (via typed operations and capability signatures), and produce structured traces for auditability and budget- and policy-aware execution. These requirements echo dataflow execution and the codelet model [6, 41], along with reusable IRs such as MLIR [19]. These are properties of the execution model and IR; the compiler/runtime presented here are a reference implementation.

A-PXM is a *specification* of a universal execution substrate: it defines substrate semantics, not a particular implementation. It rests on three pillars. First, an *Agent Abstract Machine* (AAM) formalizes agent state as beliefs, goals, and capabilities, backed by a tiered memory hierarchy (short term, long term, and episodic memory for a structured trace of past steps) with explicit operations for queries, updates, and ordering. Second, an *Agent Instruction Set* (AIS) defines a portable IR implemented as an MLIR dialect with typed operations spanning memory, latency-typed LLM calls (Ask/Think/Reason), tools, control flow, synchronization, and native agent primitives (e.g., planning and reflection). AIS serves as a *lingua franca* between outer and inner plans: models can emit AIS subgraphs that are verified and linked into developer graphs, yielding one global IR amenable to whole-program analysis. Third, a *dataflow execution engine* schedules AIS instructions based on token availability (where "tokens" refer to data availability signals in the dataflow graph, distinct from LLM text tokens). We present a *reference implementation*, and other implementations could target distributed clusters or specialized hardware with different performance tradeoffs. Concretely, this paper contributes (1) the AAM state model (B, G, C) with tiered memory and explicit transitions; (2) AIS, a portable typed IR as an MLIR dialect with verifiers and plan/reflect primitives; (3) a dataflow runtime with token-based scheduling that overlaps LLM and tool calls and supports multi-agent composition; and (4) an evaluation across ten workloads that isolate substrate capabilities (parallelism, optimization, verification, memory, tools, reflection/planning, control flow, multi-agent coordination), validating that gains come from critical-path compression and call-count reduction rather than faster model inference.

The remainder of this paper is organized as follows. Section 2 revisits execution model foundations from von Neumann architectures through dataflow and codelets, and surveys agentic AI systems. Section 3 specifies the A-PXM design, including the AAM, AIS, and dataflow execution semantics. Section 4 details the compiler pipeline and runtime architecture. Section 5 presents our empirical evaluation. Section 6 positions A-PXM relative to existing frameworks and runtimes, and Section 7 discusses limitations and future directions.

2 Background

Section 1 motivated A-PXM by framing agent workflows as suffering a workflow-level von Neumann bottleneck. This section reviews the execution-model lineage behind that analogy and explains why dataflow and codelet concepts transfer to agentic AI.

The classic von Neumann architecture's single channel between a central processing unit (CPU) and memory fostered what Backus called "word-at-a-time" programming [4], an intellectual bottleneck that obscures parallelism even when hardware can exploit it. High-performance computing responded with dataflow: programs become directed graphs where operations *fire* when their input operands arrive, not when a program counter reaches them [6]. By making data dependencies explicit, dataflow exposes parallelism without requiring programmers to write threads and synchronization primitives.

Pure dataflow faced practical challenges, particularly token-management overhead in fine-grained implementations [34]. The Codelet Model [41] regained practicality by grouping work into schedulable units with explicit dependencies and an instruction set architecture (ISA)-like contract between program and runtime. This hybrid approach preserves dataflow's parallelism benefits while choosing a granularity that amortizes scheduling overhead. The codelet model instantiates what Dennis terms a *Program Execution Model* (PXM): a formal contract between a program and its execution substrate that specifies state, operations, and scheduling so implementations can vary without changing program semantics [7]. The design lesson is to make dependencies explicit, define a formal contract, and let execution order emerge from data availability rather than control flow.

Execution models for extreme-scale systems add two further themes. First, *hierarchy*: many-core chips and clusters behave as hierarchical and distributed systems, where memory, compute, and interconnect structure matter for both performance and programmability [24]. Second, *adaptivity*: runtimes must observe execution and adjust resource usage under contention (cores, bandwidth, power/energy), motivating self-aware execution models layered on event-driven dataflow [40]. ParalleX similarly frames scaling limits as an execution-model mismatch and proposes message-driven asynchronous execution in a global address space, as realized in HPX (High Performance ParalleX) [13]. Across these lines of work, the claim is not that a new runtime is inherently faster, but that a better execution model exposes optimization and scheduling opportunities that would otherwise be inaccessible.

This lineage transfers to agentic AI because the correspondence is structural, not merely analogical. A *threaded procedure* (the coarse-grained execution unit in the codelet model) maps naturally to an agent instance; the *frame* (local state carried across codelets) maps to agent context comprising conversation history, tool results, and beliefs; individual *codelets* map to tool invocations; the *Codelet Dependency Graph* (CDG) maps to the workflow graph; and *signals* map to tool results flowing to downstream operations. Additionally, *memory codelets* [8], specialized codelets that manage data movement and context, map to context-management operations such as history summarization, pruning, and selective filtering that maintain agent working memory within token budgets. Both domains coordinate high-latency, stateful operations while trying to expose parallelism. The mapping also clarifies what must become explicit to unlock optimization: in A-PXM, the "frame" becomes an explicit state model (AAM), the ISA-like contract becomes a typed intermediate representation (AIS), and signals become dataflow tokens that drive scheduling.

Modern frameworks recreate the bottleneck at a higher abstraction level. LangChain and LangGraph compose agents as "call-at-a-time" sequences of LLM and tool invocations [17], and the dominant ReAct (Reason+Act) prompting pattern serializes high-latency stochastic calls [37]. Latency stacks across chains, errors propagate through reasoning chains, and agent state remains implicit in Python closures rather than explicit data structures amenable to analysis; empirically, single-agent accuracy can collapse from 83% to below 20% as domains and tools grow [18]. Related work provides foundations but not a complete substrate: MLIR offers progressive lowering and dialect-specific verification [19], and LLM-Compiler validates DAG-based parallelism for tool calls [14], but it lacks formal state semantics and a typed IR for whole-workflow optimization. The gap remains an execution substrate that unifies developer-authored logic with model-generated plans and makes agent state, dependencies, and side effects explicit and optimizable. Next, we specify A-PXM's AAM, AIS, and dataflow execution semantics.

3 A PXM for Agentic AI

A Program Execution Model (PXM) defines the execution semantics of a computational system: its state representation, instruction set, and scheduling discipline [41]. A-PXM instantiates this concept for agentic AI, specifying three components: an *Agent Abstract Machine* (AAM) that makes agent state explicit as beliefs, goals, and capabilities; an *Agent Instruction Set* (AIS) that provides typed operations including latency-aware LLM calls; and a *dataflow execution* model that schedules operations by data availability. Together, these components turn opaque agent scripts into analyzable programs that can be verified and optimized. To the best of our knowledge, A-PXM is the first PXM for agentic AI that combines a formal state model, an MLIR-based typed IR, and a tiered memory hierarchy with compile-time verification.

The Agent Abstract Machine (AAM)

Drawing from BDI (Belief-Desire-Intention) agent architectures [30], the AAM captures the essential state of an autonomous agent as $\text{AAM} = (B, G, C)$: beliefs B (what the agent knows) are a typed key-value store, goals G (what it wants to achieve) are a priority queue of objectives, and capabilities C (what actions it can perform) map names to typed function signatures. A transition function $\delta(\text{AAM}, \mathit{Instr}) \rightarrow \text{AAM}$ specifies how each instruction transforms state.

The AAM embeds a three-tier memory hierarchy mirroring cache, dynamic random-access memory (DRAM), and storage: short-term memory (STM) provides fast access to recent context and tool output; long-term memory (LTM) stores persistent knowledge; episodic memory records execution traces for reflection [32]. Unlike opaque memory abstractions in MemGPT [27] and Generative Agents [28], A-PXM exposes memory tiers through typed instructions: `QMEM` for queries, `UMEM` for updates, and `FENCE` for ordering constraints. A fourth tier, analogous to I/O in the von Neumann model, encompasses the external world: RAG retrieval, tool-mediated data acquisition, and human-in-the-loop interactions, expressed through `INV` and `COMMUNICATE` instructions that generalize memory codelets [8] to event-driven I/O [9].

The Agent Instruction Set (AIS)

The AIS defines typed operations on the AAM state. Table 1 presents the core instructions organized by category: memory, LLM reasoning (including latency-typed calls and native plan/reflect/verify), tools, control flow, synchronization, and communication.

Each instruction category corresponds to an effect that A-PXM makes explicit so the compiler/runtime can analyze and schedule the workflow. Memory ops expose state reads/writes and ordering, LLM ops make prompting patterns typed, tool and communication ops model side effects and multi-agent coordination, and control/synchronization ops encode branching and joins in the dataflow graph. This small typed vocabulary enables dependency analysis, safe reordering, and diagnostics before any expensive LLM call.

Notable among these are *latency-typed LLM operations* (Table 1): Ask (~1s) for fast queries, Think (~3s) for medium reasoning, and Reason (~10s) for deep inference. These latency hints enable scheduling-aware optimization: short Ask calls can overlap with downstream tools without blocking, while long Reason calls inform critical-path analysis for makespan (end-to-end completion time) estimation. Latency is one of several annotatable dimensions: cost-per-token, reliability scores, and result-reuse counters are equally applicable, making the compiler a multi-objective optimizer that can balance speed, cost, and quality. Prompt-similarity metrics further enable memoization of semantically equivalent queries as a future optimization pass. The instruction set deliberately mixes coarse-grained, latency-dominated operations (LLM calls) with fine-grained state operations (e.g., `QMEM`). Instructions have orthogonal AAM effects, allowing safe reordering when dependencies permit. AIS uses future and handle types to represent asynchronous results: a handle names an in-flight result, and a future becomes concrete once its producer completes, enabling compile-time checks that consumers only read available data.

The tool invocation subsystem (`INV`) parallels asynchronous I/O in systems programming [10]: parameters are prepared, execution proceeds in the background, and completion signals are sent to downstream consumers via tokens. This design enables overlapping tool latency with ongoing reasoning.

Dataflow Execution

Agent plans are compiled into dataflow graphs where nodes represent AIS instructions and edges carry data tokens. The current compilation target is a DAG obtained by unrolling bounded loops, but the AIS includes iterative primitives (`LOOP_START`/`LOOP_END`) and conditional control flow (`BRANCH`, `TRY_CATCH`) that can represent cyclic patterns such as verify-fix-retry loops and recursive model

	Mnemonic	Instruction	Operands	Description
Memory	QMEM	QueryMemory	*q, sid, k*	Retrieve from memory (vector/kv); optionally stage into STM.
	UMEM	UpdateMemory	*data, sid*	Persist/update memory; update Beliefs.
LLM	ASK	Ask	*prompt, ctx*	Fast LLM query (~1s); classification, extraction, simple Q&A.
	THINK	Think	*prompt, ctx*	Medium LLM reasoning (~3s); analysis, planning, synthesis.
	REASON	Reason	*prompt, ctx*	Deep LLM reasoning (~10s); multi-step inference, complex decisions.
	PLAN	Plan	*goal, ctx*	Produce a structured plan/goal object for downstream execution.
	REFL	Reflect	*trace/content, ctx*	Structured self-critique over episodic trace or provided content.
	VERIFY	Verify	*claim, evidence*	Validate a claim against evidence; returns a typed verdict token.
Tool	INV	InvokeTool	*tool, params*	Call external tool with structured params; store result.
Control	BRANCH	BranchOnValue	*tok, val, lblT, lblF*	Conditional branch on token value.
	SWITCH	Switch	*disc, cases*	Multi-way branch on discriminant value.
Sync	MERGE	Merge	*tokens*	Sync parallel paths; aggregate tokens.
	WAIT_ALL	WaitAll	*tokens*	Block until all tokens available.
	FENCE	Fence	–	Memory barrier; order prior `QMEM`/`UMEM`.
Comm	TRY_CATCH	TryCatch	*trySubg, catchSubg*	Structured exception handling with recovery subgraph.
	COMM	Communicate	*rcpt, msg, prot*	Send message to recipient using protocol.
	FLOW	FlowCall	*agent, flow, args*	Invoke flow on another agent; enables cross-agent parallelism.

Table 1: Core Agent Instruction Set (AIS) operations organized by category: memory operations, reasoning, tool invocation, control flow, synchronization, error handling, and inter-agent communication.

invocations [29]; the DAG target is a pragmatic choice, not a fundamental expressiveness limitation. Unlike control-driven execution that follows a program counter, the scheduler fires instructions whenever their inputs become available, automatically exposing parallelism inherent in the dataflow structure. Small values pass by value between operations; large objects use references into shared memory, a hybrid strategy that balances token management overhead against memory pressure [41].

Contemporary reasoning models (e.g., OpenAI o1 [25]) internalize planning, creating what we term an "inner plan" that is opaque to the developer's "outer plan." A-PXM bridges this gap by prompting models to emit AIS subgraphs that undergo type-checking and linking into the developer-authored graph. The result is a single global IR where both human intent and model-generated logic become amenable to whole-program analysis and optimization.

More broadly, execution-model proposals are best understood as *opportunity enablers*: they define what can be optimized, measured, and controlled across implementations and platforms [13, 24, 40]. A-PXM exposes five opportunity classes that motivate both the design and our evaluation: (1) *critical-path compression* by overlapping independent LLM and tool operations derived from explicit data dependencies; (2) *whole-workflow optimization* via IR-level transforms across node boundaries (e.g., call fusion, dead-code elimination, and future caching/prompt-batching passes); (3) *safety and policy enforcement* through typed operations, explicit memory effects, and capability signatures that enable early failure and checks before costly execution; (4) *auditability and self-awareness* from inspectable beliefs/goals/capabilities and episodic traces with telemetry (latency, tokens, failures) for policy-aware scheduling; and (5) *portability* by separating workflow logic (*what*) from execution policy and backend selection (*how*).

4 Implementation

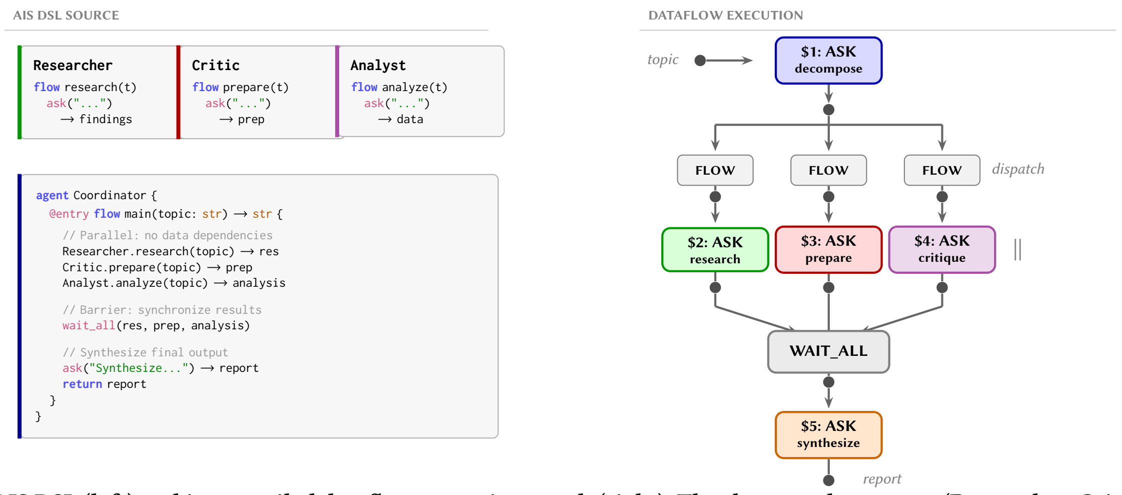
A-PXM materializes as a compiler-runtime system. The compiler transforms programs written in an AIS domain-specific language (DSL) through progressive lowering until reaching executable form; the runtime executes the resulting dataflow graphs. Figure 2 previews this end-to-end path from source-level agent flows to an executable dataflow graph.

Compilation Pipeline

Figure 2 (left) shows the AIS DSL: agents encapsulate capabilities as typed flows, and the `@entry` annotation marks the coordinator's entry point. Flow calls like `Researcher.research(topic)` express inter-agent invocations; the compiler infers that these three calls share no data dependencies and can execute in parallel. The `wait_all` primitive synchronizes results before the final `ask` synthesizes the report. In the compiled graph (right): operation \$1 (entry) decomposes the goal; \$2–\$4 are the three parallel Ask calls to worker agents; \$5 synthesizes results after the synchronization barrier.

The compiler transforms this source through four stages: parsing constructs an abstract syntax tree (AST), MLIRGen lowers to the AIS MLIR dialect with custom verifiers, optimization passes analyze and transform the IR, and the artifact emitter serializes to a binary format (`.apxmobj`). Figure 2 (right) shows the resulting dataflow graph where operations \$2–\$4 fire in parallel when their input tokens become ready. This is dataflow execution: parallelism emerges from the DAG structure, not from explicit async/await annotations.

Parsing constructs an AST from the AIS DSL, then MLIRGen lowers it to the AIS MLIR dialect, where custom verifiers catch type mismatches, undefined variables, and malformed dataflow before any LLM call. Optimization passes then transform the IR: *FuseAskOps* batches producer-consumer Ask chains to reduce API



```
Researcher
flow research(t)
  ask("...")
    → findings
```

```
Critic
flow prepare(t)
  ask("...")
    → prep
```

```
Analyst
flow analyze(t)
  ask("...")
    → data
```

```
agent Coordinator {
  @entry flow main(topic: str) → str {
    // Parallel: no data dependencies
    Researcher.research(topic) → res
    Critic.prepare(topic) → prep
    Analyst.analyze(topic) → analysis

    // Barrier: synchronize results
    wait_all(res, prep, analysis)

    // Synthesize final output
    ask("Synthesize...") → report
    return report
  }
}
```

Figure 2: AIS DSL (left) and its compiled dataflow execution graph (right). The three worker agents (Researcher, Critic, Analyst) map to parallel Ask operations ($2–$4). The Coordinator orchestrates execution: after $1 decomposes the goal, Flow operations dispatch work to agents. The Wait_All barrier synchronizes before $5 synthesizes the final report. Parallelism emerges from data dependencies without explicit async/await.

round-trips (bounded by fusion depth and token budgets), while canonicalization, common subexpression elimination (CSE), and dead-code elimination simplify the program. Finally, the artifact emitter serializes the typed execution DAG to a `.apxmobj` binary loaded directly by the runtime.

Dataflow Runtime

The runtime executes compiled programs using token-based dataflow scheduling, where data availability rather than control flow determines when operations execute. The scheduler tracks pending inputs per operation; when a token becomes ready, it decrements consumer counters, and operations whose counters reach zero are enqueued, providing O(1) readiness detection without graph traversal [3]. The Agent Abstract Machine maps directly from the formal model as a concurrent belief store, a priority goal queue, and a capability registry with schema validation, while state transitions are recorded in an append-only episodic log. Short-term memory (STM) is an in-memory key-value store, long-term memory (LTM) persists to SQLite, and episodic memory is an append-only log, with independent locks per tier for concurrent access. Multi-agent communication uses the `COMMUNICATE` instruction as a typed messaging primitive (currently recorded as intent, with inter-process communication (IPC) or network transport as future work). A unified backend loads model/provider parameters from a TOML (Tom's Obvious, Minimal Language) configuration file (`.apxm/config.toml`) and emits structured telemetry (per-operation latency, token usage, failures, scheduler statistics) to support post-hoc analysis and policy-aware scheduling.

5 Evaluation

We evaluate A-PXM around three questions: (i) does the substrate provide capabilities beyond ad hoc orchestration, (ii) is runtime overhead acceptable, and (iii) does the execution model expose optimization opportunities that reduce end-to-end latency?

Experimental Setup

All experiments run on a dedicated x86_64 Linux host (AMD EPYC 75F3 32-core, 755GB RAM, 4×A100-40GB, Ubuntu 24.04) with Ollama 0.14.2 serving gpt-oss:120b (120B parameters, 131K context window). We evaluate A-PXM through ten controlled workloads that isolate substrate capabilities: parallelism, optimization, verification, memory, tools, reflection/planning, control flow, and multi-agent coordination. Table 2 summarizes the workloads and their primary metrics. For framework comparison, A-PXM and LangGraph use identical LLM backends, model configurations, and system prompts (shared config). We report A-PXM internal `runtime_ms` (excluding subprocess spawn and compilation) alongside LangGraph's in-process `graph.invoke()` wall time; subprocess overhead (~50ms) is negligible for LLM-dominated workloads and most visible on lightweight fan-out (4a–c).

We organize evaluation around three pillars: **dataflow execution** (Workloads 1, 4, 9, 10), **typed IR** (Workloads 2, 3, 7, 8), and **AAM semantics** (Workloads 5, 6). We also implement equivalent workflows in LangGraph with Python; these comparisons measure workflow-level properties (LLM calls, lines of code, error detection timing) under the same LLM backend, not language runtime performance. Each benchmark runs 3 warmup iterations and 10 measured iterations. We report mean ± std and percentiles (p50, p95) for timing metrics, along with (i) *wall-clock time* (subprocess execution including compilation) and (ii) *internal runtime* (excluding subprocess spawn and compilation) for A-PXM vs LangGraph comparison. Using shared instrumentation, we decompose time into compilation, LLM latency, tool latency, and runtime overhead, and estimate framework overhead as `framework_overhead = total_time − llm_latency − compile_time`. Variance is dominated by stochastic LLM latency on LLM-heavy workloads, so we avoid significance claims when intervals overlap.

Table 2: Benchmark workloads and primary metrics, organized by A-PXM pillar. T_{sched} = scheduler overhead (μs); calls = LLM API call count; tokens = LLM input/output tokens; efficiency = speedup / ideal parallelism.

#	Workload	Pillar	Primary Metric(s)
1	Parallel Research	Dataflow	avg/max parallelism; T_{sched}
2	Chain Fusion	AIS	LLM calls; end-to-end latency; tokens
3	Type Verification	AIS	compile-time detection time; wasted calls avoided
4	Scalability	Dataflow	parallel efficiency vs. fan-out N
5	Memory Augmented	AAM	STM/LTM/episodic access latency
6	Tool Invocation	AAM	INV overhead; capability validation
7	Reflection	AIS	schema validity; prompt/-trace incorporation
8	Planning	AIS	goal schema validity; parallel step execution
9	Conditional Routing	Dataflow	call count under routing; unreachable paths
10	Multi-Agent	Dataflow	cross-agent parallelism; makespan

Dataflow Execution

In A-PXM, the scheduler fires operations when their input tokens become available, so independent operations execute in parallel without manual `async/await` coordination. Workload 1 isolates a simple fan-out/fan-in (three independent Ask calls followed by synthesis): A-PXM reaches 2.5 average parallelism and reduces mean end-to-end latency from 11,248ms (LangGraph) to 8,977ms (A-PXM), a 1.25× speedup (Table 4). To stress scalability, Workloads 4a–c increase fan-out to N=2/4/8; scheduler overhead stays 7.5μs per operation (Table 3), while fixed process overhead dominates the smallest graphs, yielding ratios from 0.83× at N=2 to 0.93× at N=8 (Table 4).

The largest gains appear when the workflow's dependency structure prevents wasted work or exposes parallelism across agents. In Workload 9, a classification step feeds a switch/case so only the selected branch executes; A-PXM reduces mean end-to-end time from 35,658ms (LangGraph) to 6,879ms (A-PXM), a 5.18× speedup (Table 4). Workload 10 composes three agents via cross-agent flow calls and reaches 3.5 average parallelism (max 5 concurrent ops), reducing mean latency from 177,436ms (LangGraph) to 17,112ms (A-PXM), a 10.37× speedup (Table 4). In both cases, A-PXM compresses the critical path by overlapping independent calls rather than accelerating any single LLM invocation.

Typed IR & Compiler

A-PXM's typed IR changes what can be verified and optimized before execution. In Workload 3, the AIS verifier catches semantic errors (undefined variables, type mismatches, malformed capabilities) at compile time: A-PXM detects 5 errors in 65ms with zero LLM calls, whereas LangGraph surfaces equivalent failures only at runtime after paying LLM latency. Workload 2 demonstrates whole-workflow optimization via FuseAskOps: fusing a producer-consumer chain reduces calls from 5 to 1 and lowers latency from 9,303ms (LangGraph) to 7,236ms (A-PXM), a 1.29× speedup (Table 4). For reflection and planning (Workloads 7–8), A-PXM benefits from schema-validated intermediate objects: Workload 7 reduces latency from 13,555ms (LangGraph) to 9,344ms (A-PXM), a 1.45× speedup, while Workload 8 is dominated by model latency (>95% of runtime) and shows a 0.89× ratio; here the value is typed plan objects and structured traces rather than speedup (Table 4).

Table 3: Runtime overhead breakdown per operation.

Phase	Mean Latency	% of Total
Ready set update	1.8μs	24.0%
Priority queue dequeue	0.9μs	12.0%
Dependency resolution	1.4μs	18.7%
Operation dispatch	2.4μs	32.0%
Token routing	1.0μs	13.3%
Total	**7.5μs**	100%

Agent Abstract Machine

The AAM makes beliefs, goals, and capabilities explicit and records transitions in episodic memory, but these state operations are inexpensive relative to LLM calls. Across STM/LTM/episodic tiers (hash-map reads, SQLite persistence, and append-only logging), overhead remains negligible: the memory workload (Workload 5) is near parity at 1.02× (1665ms vs 1695ms; Table 4). Tool invocation (Workload 6) exercises `INV` with capability validation and reduces mean end-to-end time from 6,181ms (LangGraph) to 3,823ms (A-PXM), a 1.62× speedup; the dispatch itself adds <1ms, and improvements come from workflow structure and explicit capability checks (Table 4). Table 3 reports scheduler overhead at 7.5μs per operation, so even a 1,000-operation workflow adds about 7.5ms of scheduling time.

Discussion

Table 4 and Figure 3 summarize end-to-end latency across all workloads (mean ± std over 10 runs). Ratios range from 0.83× (A-PXM slower) to 10.37× (A-PXM faster), and the spread is largely explained by whether the workflow exposes parallel work and whether optimization can reduce call count. Multi-agent coordination (Workload 10) and conditional routing (Workload 9) sit at the high end because A-PXM overlaps independent operations and avoids launching work on untaken branches, while chain fusion (Workload 2) shows a smaller but consistent benefit by collapsing serial LLM calls into fewer API round-trips. By contrast, planning (Workload 8) is dominated by a single long model call and sees no speedup, and small fan-out graphs are most sensitive to fixed process overhead (Workloads 4a–c). When A-PXM helps, it does so by shortening the critical path, reducing the number of calls, or avoiding wasted calls via compile-time verification, not by accelerating model inference.

Runtime overhead is negligible at 7.5μs per operation (Table 3), leaving headroom for structured tracing, policy-aware scheduling, and backend portability without materially changing latency. Beyond wall-clock time, the typed IR shifts developer experience and failure modes: Table 5 shows 7.3× fewer lines of workflow

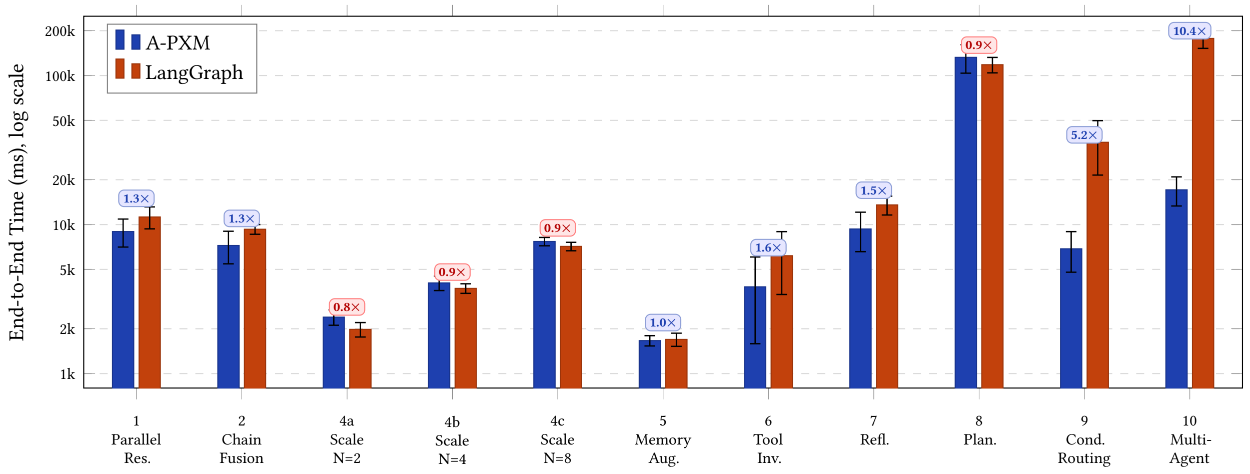


Figure 3: End-to-end execution time comparison (mean ± std, 10 iterations). Figure excludes Workload 3 (compile-time verification), which measures detection time rather than runtime latency. Left-to-right: Workloads 1, 2, 4a, 4b, 4c, 5, 6, 7, 8, 9, 10 (matching Table 4). Blue badges: A-PXM faster; red badges: LangGraph faster. Summary: Max 10.37×, Min 0.83×, Median 1.25×.

Table 4: Latency results (mean ± std, ms). Workload # refers to Table 2; 4a/b/c = Scalability with N=2/4/8 parallel operations. Categories: DF = Dataflow, AIS = Typed IR, AAM = Agent Abstract Machine. Ratio = (LangGraph latency)/(A-PXM latency), so Ratio >1 means A-PXM is faster (lower latency).

#	A-PXM (ms)	LangGraph (ms)	Ratio	Cat.
1	8,977 ± 1,912	11,248 ± 1,889	1.25×	DF
2	7,236 ± 1,783	9,303 ± 697	1.29×	AIS
4a	2,393 ± 283	1,980 ± 220	0.83×	DF
4b	4,060 ± 452	3,730 ± 276	0.92×	DF
4c	7,700 ± 495	7,137 ± 465	0.93×	DF
5	1,665 ± 132	1,695 ± 172	1.02×	AAM
6	3,823 ± 2,236	6,181 ± 2,786	1.62×	AAM
7	9,344 ± 2,763	13,555 ± 1,961	1.45×	AIS
8	132,426 ± 28,686	118,281 ± 14,021	0.89×	AIS
9	6,879 ± 2,090	35,658 ± 14,174	**5.18×**	DF
10	17,112 ± 3,785	177,436 ± 25,257	**10.37×**	DF
Summary			0.83–10.37×	

wiring, 49× faster error detection (compile time vs runtime), and 5× fewer LLM calls after fusion under the same backend. These benefits do not depend on Rust versus Python; they come from having an explicit, verifiable program representation. A-PXM does not claim universal speedups, and fusing operations can degrade response quality by collapsing intermediate steps and supervision, especially under long contexts [20]; our evaluation therefore focuses on substrate capabilities and latency rather than end-task accuracy.

6 Related Work

Having presented A-PXM's design and evaluation, we position it as an *execution substrate*: a layer between high-level orchestration frameworks and low-level LLM serving. This section contrasts A-PXM with CLI tools, orchestration frameworks, parallel function calling, memory systems, and dataflow runtimes. Table 6 summarizes key dimensions and highlights that A-PXM uniquely combines dataflow execution, explicit agent state (AAM), a statically typed IR (MLIR), and tiered memory, enabling compile-time errors, whole-workflow optimization, and auditability.

Table 5: A-PXM vs LangGraph on equivalent workflow.

Metric	A-PXM	LangGraph	Gain
Lines of code	12	88	7.3× fewer
Error detection	65ms	3.2s	49× faster
Parallelism	Automatic	Manual	Qualitative
State inspection	Always	Debugger	Qualitative
LLM calls (fused)	1	5	5× fewer
Compile checks	52+	0	–

Table 6: Comparison with LangGraph and LLMCompiler.

Dimension	LangGraph	LLMComp.	A-PXM
Execution	Control-flow	DAG planning	Dataflow
State	Implicit dict	None	AAM[†]
Types	Dynamic	None	Static (MLIR)
Memory	Closures	None	3-tier[‡]
Parallel	Manual	Auto (plan)	Auto (DAG)
Optimize	None	None	Fusion, CSE[*]
Auditability			
Diagnostics	–	–	✓
Metrics	–	–	✓
State query	Debugger	N/A	✓
Error time	Runtime	Runtime	Compile

[†]AAM = Agent Abstract Machine with Beliefs, Goals, Capabilities. [‡]3-tier = STM/LTM/Episodic. [*]CSE = Common Subexpression Elimination.

At the top of the stack, user-facing tools like Claude Code [1] and OpenAI Codex [26] provide model access and execution environments without a typed IR, explicit state model, or compile-time verification for whole-workflow analysis and optimization. While such tools can run parallel tool calls, parallelism is typically decided by prompting/model behavior rather than derived from an explicit program representation with verifiable dependencies and effects [35]. Orchestration frameworks such as LangChain and LangGraph [17] coordinate agent workflows through graph construction, but treat nodes as opaque units (Python callables, Hypertext Transfer Protocol (HTTP) calls), limiting static analysis and whole-program optimization across node boundaries. Execution-infrastructure proposals such as AIOS (LLM Agent Operating System) [22] and Runhouse [31] emphasize standardized interfaces and a separation between orchestration and execution, and difficulty-aware orchestration adapts strategies per query [33]. These efforts are complementary: A-PXM provides the portable, typed layer that can sit beneath such policy systems and make their decisions auditable and optimizable.

LLMCompiler [14] is the closest prior work, introducing DAG-based parallel function calling with reported 3.7× latency speedup and 6.7× cost savings over ReAct. However, it targets function calls without formal state semantics: it lacks an explicit agent state model, typed IR for whole-program optimization, and a memory hierarchy for persistent knowledge. A-PXM fills this gap by pairing a formal execution model (AAM) with typed instructions (AIS), tiered memory, and token-based dataflow scheduling that extracts parallelism automatically from data dependencies. Memory systems such as MemGPT [27] and Generative Agents [28] show the value of tiered context and reflection over episodic traces; A-PXM makes these mechanisms first-class through typed memory instructions (`QMEM`, `UMEM`, `FENCE`) and explicit ordering constraints, enabling compiler-level analysis. Finally, dataflow scheduling from high-performance computing, exemplified by the codelet model [41], motivates A-PXM's execution semantics: in our setting, the "tasks" are LLM/tool calls and the dependencies are token flows between reasoning steps.

7 Conclusion

This paper introduced A-PXM, a Program Execution Model (PXM) for agentic AI that defines a stable contract for representing workflow state, effects, and dependencies so that execution can be verified, scheduled, and optimized. A-PXM combines an explicit state model (AAM) with tiered memory, a typed MLIR-based IR (AIS), and token-driven dataflow scheduling.

Our reference compiler/runtime shows that making this contract explicit enables automatic overlap of independent LLM/tool calls, call-count reduction via fusion, and compile-time verification. Across ten workloads, A-PXM achieves up to 10.37× latency reduction (multi-agent coordination), 5.18× on conditional routing, and 1.29× from call fusion, with 7.5μs per-operation overhead. Gains stem from critical-path compression and call reduction, not faster model inference.

Limitations and Future Work. The current prototype is single-node and requires manual AIS authoring, and today it targets mostly DAG-shaped workflows (loops are unrolled) rather than dynamic, long-running services with evolving policies. Our evaluation focuses on substrate capabilities and latency under a controlled backend, not end-task accuracy, and quality-aware fusion and caching policies remain open problems. Future work includes higher-level frontends (e.g., transpilation from orchestration frameworks and LLMs fine-tuned to emit AIS subgraphs directly, closing the inner/outer plan gap), distributed execution behind `COMMUNICATE`, and policy-aware scheduling using exported telemetry (budgets, priorities, and provider selection). We also plan to evaluate on established agent benchmarks such as SWE-bench [12], AgentBench [21], GAIA [23], and WebArena [39] to quantify success-rate effects of explicit execution semantics.

Beyond these immediate steps, A-PXM suggests a broader compiler-toolchain agenda for agentic programs. A key direction is quality-aware optimization for LLM workflows: develop fusion heuristics that account for context length, intermediate supervision, and stochasticity, and couple them with regression-style evaluation so latency improvements do not silently degrade output quality. A second direction is integration: treat existing orchestration frameworks as frontends that lower into AIS, enabling whole-workflow analysis, debugging, and portability while preserving developer ergonomics.

The memory model is another research frontier because it makes agent state explicit and schedulable without prescribing a specific backend. We see opportunities in context compaction and token management (e.g., prompt compression and selective retrieval [11]), in memory tiering and lifecycle policies across STM/LTM/episodic traces [27, 28, 38], and in co-design with serving-level KV cache management (e.g., PagedAttention in vLLM [15]). Finally, explicit dependencies and effects enable heterogeneous placement as a formal optimization problem (across devices and agents), and, in the longer term, motivate exploring hardware support for readiness detection and memory fencing once software profiling identifies true bottlenecks.

References

[1] Anthropic. 2025. Claude Code: Best Practices for Agentic Coding. Engineering blog. https://www.anthropic.com/engineering/claude-code-best-practices

[2] Anthropic. 2025. How we built our multi-agent research system. Research blog. https://www.anthropic.com/engineering/multi-agent-research-system

[3] Arvind and Rishiyur S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318. doi:10.1109/12.48862

[4] John Backus. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (1978), 613–641. doi:10.1145/359576.359579

[5] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Ion Stoica, and Christopher Ré. 2025. Why Do Multi-Agent LLM Systems Fail? *arXiv preprint arXiv:2503.13657* (2025). doi:10.48550/arXiv.2503.13657

[6] Jack B. Dennis. 1974. *First Version of a Data Flow Procedure Language.* Technical Report. MIT Laboratory for Computer Science.

[7] Jack B. Dennis. 2012. Program Execution Models for Massively Parallel Computing. In *Applications, Tools and Techniques on the Road to Exascale Computing.* Advances in Parallel Computing, Vol. 22. IOS Press, 29–40. doi:10.3233/978-1-61499-041-3-29

[8] Dawson Fox, Jose Monsalve Diaz, and Xiaoming Li. 2023. On Memory Codelets: Prefetching, Recoding, Moving and Streaming Data. arXiv preprint arXiv:2302.00115. doi:10.48550/arXiv.2302.00115

[9] Guang R. Gao, Joshua Suetterlein, and Stéphane Zuckerman. 2011. *Toward an Execution Model for Extreme-Scale Systems: Runnemede and Beyond.* CAPSL Technical Memo 104. Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware.

[10] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

[11] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. *arXiv preprint arXiv:2310.05736* (2023). https://arxiv.org/abs/2310.05736

[12] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2310.06770

[13] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. 2009. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *2009 International Conference on Parallel Processing Workshops*. doi:10.1109/ICPPW.2009.14

[14] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM Compiler for Parallel Function Calling. In *Proceedings of the 41st International Conference on Machine Learning (ICML) (PMLR, Vol. 235)*. 24370–24391. https://arxiv.org/abs/2312.04511

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joaquin Gonzalez, Ion Stoica, and Hao Zhang. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. Preprint. https://arxiv.org/abs/2309.06180

[16] Pacific Northwest National Laboratory. 2024. Crete: An Active Memory Computer Purpose-built for AI Science Applications. News article. https://www.pnnl.gov/news-media/unique-active-memory-computer-purpose-built-ai-science-applications

[17] LangChain. 2024. *LangChain Documentation*. https://docs.langchain.com/

[18] LangChain. 2025. Benchmarking Single Agent Performance. Blog post. https://blog.langchain.com/react-agent-benchmarking/

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[20] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[21] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. AgentBench: Evaluating LLMs as Agents. *arXiv preprint arXiv:2308.03688* (2023). https://arxiv.org/abs/2308.03688

[22] Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. 2024. AIOS: LLM Agent Operating System. *arXiv preprint arXiv:2403.16971* (2024). https://arxiv.org/abs/2403.16971

[23] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raber, Paul Roit, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. 2023. GAIA: A Benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983* (2023). https://arxiv.org/abs/2311.12983

[24] Jose Monsalve, Kevin Harms, Kumaran Kalyan, and Guang Gao. 2019. Sequential Codelet Model of Program Execution - A Super-Codelet model based on the Hierarchical Turing Machine. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. 1–8. doi:10.1109/IPDRM49579.2019.00005

[25] OpenAI. 2024. *Learning to Reason with LLMs*. https://openai.com/index/learning-to-reason-with-llms/

[26] OpenAI. 2025. Introducing Codex. OpenAI Blog. https://openai.com/index/introducing-codex/

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

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

[29] Prime Intellect. 2025. Recursive Language Models: The Paradigm of 2026. Blog post. https://www.primeintellect.ai/blog/rlm

[30] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems*. AAAI Press, 312–319. https://cdn.aaai.org/ICMAS/1995/ICMAS95-042.pdf

[31] Runhouse. 2023. Please Separate Orchestration and Execution (Or: Packaging Hell is Dragging ML). Blog post. https://www.run.house/blog/orchestration-and-execution

[32] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint arXiv:2303.11366* (2023). https://arxiv.org/abs/2303.11366

[33] Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, and Lewei He. 2025. Difficulty-Aware Agentic Orchestration for Query-Specific Multi-Agent Workflows. arXiv preprint arXiv:2509.11079. https://arxiv.org/abs/2509.11079

[34] Arthur H. Veen. 1986. Dataflow Machine Architecture. *Comput. Surveys* 18, 4 (1986), 365–396. doi:10.1145/6462.6485

[35] Simon Willison. 2025. Embracing the Parallel Coding Agent Lifestyle. Blog post. https://simonwillison.net/2025/Oct/5/parallel-coding-agents/

[36] Lance Yao, Suman Samantray, Ayana Ghosh, Kevin Roccapriore, Libor Kovarik, Sarah Allec, and Maxim Ziatdinov. 2025. Operationalizing Serendipity: Multi-Agent AI Workflows for Enhanced Materials Characterization with Theory-in-the-Loop. arXiv preprint arXiv:2508.06569. https://arxiv.org/abs/2508.06569

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2023). https://arxiv.org/abs/2210.03629

[38] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2023. MemoryBank: Enhancing Large Language Models with Long-Term Memory. *arXiv preprint arXiv:2305.10250* (2023). https://arxiv.org/abs/2305.10250

[39] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2307.13854

[40] Stephane Zuckerman, Aaron Landwehr, Kelly Livingston, and Guang Gao. 2014. Toward a Self-Aware Codelet Execution Model. In *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*. 26–29. doi:10.1109/DFM.2014.12

[41] Stéphane Zuckerman, Joshua D. Suetterlein, Rob C. Knauerhase, and Guang R. Gao. 2011. Using a "codelet" program execution model for exascale machines. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. 64–69. doi:10.1145/2000417.2000424