

CONTROL-PLANE PLACEMENT AND MEMORY SUBSTRATES IN MODERN AGENTIC AI: A COMPARATIVE METHOD FOR PROMPT-NATIVE SYSTEMS, AGENT SHELLS, AND DURABLE RUNTIMES

A. Bondarenko*, N. Zamorskaia,
D. Djakons, A. Djakona, R. Djakons

Riga Nordic University,
1-5 Valerijas Seiles Str., Riga, LV-1019, LATVIA
*e-mail: andrejs.bondarenko@gmail.com

AI agents that autonomously write, test, and modify code are now deployed across architectures that share similar surface features but differ substantially in where and how operating control is exercised. Contemporary systems span general-purpose SDKs, prompt-native workflow packs, interactive coding shells, hosted agents, memory services, and durable workflow runtimes, yet the existing language of framework comparison often treats them as interchangeable. The remaining problem is comparative precision: these systems still look artificially similar if operational control and durable experience are assumed to live in the same architectural layer. This paper proposes control-plane placement, complemented by memory substrate analysis, as a comparative method for modern agentic systems. The organising principle is the enforcement set – the collection of architectural layers that can independently constrain agent behaviour, and the five system families compared here represent its five stable configurations. The method asks where workflow policy, tool exposure, permissioning, routing, verification, and recovery are specified and enforced, and separately how durable experience is stored, searched, updated, and reused across sessions. Using official documentation and open-source implementation evidence, the paper compares prompt-native systems, prompt-modular shells, runtime-governed shells, host-mediated or coordinator forms, and durable workflow runtimes across a twelve-system ecosystem. It contributes a qualitative coding matrix for control-plane salience and a companion memory-oriented comparison covering artefact memory, procedural memory, write-policy visibility, and traceability. The paper also argues that coding-agent

benchmarks are best read as evaluations of *model × reasoning policy × harness/runtime × memory substrate × environment* under explicit tool, budget, and infrastructure conditions. The result is a more discriminating vocabulary for agentic AI and a clearer method for comparing rapidly converging systems.

Keywords: *Agent harnesses, agentic AI, benchmarks, coding agents, control-plane placement, episodic memory, memory substrate, procedural memory, semantic memory, workflow runtimes.*

1. INTRODUCTION

Discussion of agentic AI increasingly suffers from a mismatch between language and reality. The language still often assumes that framework brand is the main comparative unit, while the reality is a heterogeneous ecosystem that includes SDKs, prompt-native workflow packs, interactive coding shells, hosted agents, coordinator layers, and durable workflow runtimes. Even framework-adjacent systems increasingly present themselves in broader agentic terms; LlamaIndex, for example, explicitly frames its role as agentic document processing rather than as a narrowly defined retrieval layer [1]. These objects share some components, but they do not place operational control in the same part of the system. Treating them as one generic category obscures the most important design differences.

The literature has already made several essential moves. Recent work has established that harnesses and scaffolds matter, that workflow graphs are analytically important, and that agent performance is often a property of a larger system rather than of the model alone [2]–[6]. The contribution of this paper is the comparative question that follows from that: *Where is the control plane actually placed, and how strongly is it enforced?*

In this paper, control-plane placement means the primary architectural layer that owns the system’s working policy: how tasks are staged, which tools are exposed, where permissions are checked, how delegation occurs, how failures are repaired, and how long-running work is resumed. In some systems, that policy is mainly implemented through prompts, skills, and artefacts. In others, it sits in a shell runtime, a host environment, a coordinator layer, or an explicit workflow graph.

The organising principle that gives the spectrum its analytical purchase is what the paper terms the enforcement set: the collection of architectural layers that can independently constrain or veto what an agent does, regardless of what the model outputs or what its prompt instructs. At the prompt-native end, the enforcement set contains only the document layer; policy holds only if the model chooses to follow it. Each step along the spectrum adds an independent enforcing layer: a session runtime, a host platform, a workflow graph. The five families are not arbitrary taxonomy entries but the five stable configurations of this enforcement set. Every finding in this paper – the flexibility – governance trade-off, the benchmark formula, the safety implications, the cost–quality relationship – is a

consequence of how large and independent that set is.

This reframing is useful for three reasons. First, it provides a more discriminating vocabulary for systems that now converge on similar visible primitives: instructions, skills, tools, hooks, subagents, approvals, sessions, and checkpoints, without implementing them in the same way [7]–[13]. Second, it helps position current work relative to overlapping scholarship on harnesses, externalization, workflow graphs, coding scaffolds, and external-memory evaluation. Third, it changes how benchmarks should be interpreted, because many measured differences arise from harness design, memory substrate, tool policy, budget, and infrastructure rather than from model capability alone.

The approach is complementary to, but distinct from, reasoning-topology analysis [14]–[22]. Reasoning topology describes the shape of inference; control-plane placement describes the enforcement substrate that constrains, stages, verifies, and recovers that inference – the two dimensions are orthogonal, and production coding agents typically exhibit a ReAct-like outer loop

regardless of which enforcement family they belong to.

An earlier shorter overview by the present authors introduced the vocabulary of control surfaces and identified four recurrent system forms visible in current practice [23]. The present paper extends that foundation with a formal comparative method, adds memory substrate as an independently analysed comparison axis, introduces host-mediated coordinator forms as the fifth system family, and operationalizes the comparison through a qualitative coding matrix and a benchmark interpretation formula.

The paper makes three contributions. It proposes a comparative method centred on control-plane placement while treating memory substrate as a separate comparison axis. It operationalizes that method through a qualitative coding matrix with heatmap-style salience coding across several architectural dimensions and through a companion memory-oriented comparison. In addition, it argues that agentic coding benchmarks should be read as measurements of *model* $\times$ *reasoning policy* $\times$ *harness/runtime* $\times$ *memory substrate* $\times$ *environment* under stated tool, budget, and infrastructure conditions.

2. RELATED WORK AND CONTRIBUTION BOUNDARY

Prior work on harnesses and externalized infrastructure [2], [3] establishes that memory, skills, protocols, and harnesses are explicit system layers rather than invisible scaffolding; workflow-graph research [4] makes graph structure and runtime adaptation first-class analytical objects; and coding-scaffold analysis [5] demonstrates that agent behaviour is scaffold-dependent across open-source implementations. Evaluation guidance and benchmark design [6], [24], [25] establishes that the evaluated object is

a configured system, not a bare model. Governance and design-pattern literature [26], [27] treats roles, protocols, and oversight as architecture. The present paper contributes the comparative question that cuts across all of these: *Where is the enforcement set located, and what are the consequences?* It applies that question to a current cross-section of unlike product forms: prompt packs, coding shells, IDE-hosted agents, and durable runtimes, using official documentation and open-source evidence.

3. COMPARATIVE METHOD AND ANALYTIC VOCABULARY

The method used here combines three evidence classes. Level 1 evidence consists of official documentation, protocol specifications, benchmark papers, and official engineering notes. Level 2 evidence consists of open-source repositories, runtime code, prompt packs, skill bundles, and archived public artefacts. Level 3 evidence

would include third-party prompt extractions or observational commentary; such material is not used as foundational support for the paper’s main claims. This discipline matters because agentic systems differ not only in behaviour but also in observability. The comparison tracks five main variables summarised in Table 1.

Table 1. Analytic Variables Used in the Comparison

Variable	Guiding question	Typical evidence
Memory substrate	What durable experience can be stored, searched, updated, and reused across turns, tasks, or resumptions?	Sessions, files, reports, memory notes, skill stores, branches, threads, checkpoint stores, codebase index graphs
Context policy	What is surfaced to the model at a given step?	Retrieval rules, summaries, prompt assembly, skill loading, compaction
Workflow topology	How does work move through the system?	Tool loops, delegation patterns, checkpoints, artefact pipelines, graph nodes
Enforcement mode	How strongly are permissions, stop rules, and repairs imposed?	Approval modes, runtime gates, callbacks, review steps, retry logic
Control-plane placement	Which layer primarily specifies and enforces operating policy?	Prompts and artefacts, shell runtime, host roles, coordinator layer, graph runtime

These variables are operationalized in the coding matrix through six heatmap dimensions. Control-plane placement decomposes into three positional columns (Prompt, Runtime, Host/graph locus) mapping the enforcement-set configuration. Enforcement mode maps to Governance visibility; Memory substrate maps to Durable state; Workflow topology and context policy jointly produce the Optimization leverage column. The key practical distinction is not whether a system has a feature such as subagents or approvals, but where that feature is encoded and whether it is enforced independently of model output.

Durable experience should not be collapsed into context engineering: context policy determines what the model sees now; memory substrate determines what can per-

sist long enough to shape later action. This distinction also reveals a progression along the spectrum: at the prompt-native end, context is assembled once per step from whatever the prompt template selects; at the orchestrator and durable-runtime end, the enforcement set decomposes context across parallel sub-agents, each receiving a focused, scoped window, and explicitly governs the information passed between them. Context management thereby shifts from a prompt-engineering concern to a first-class architectural property.

The spectrum in Fig. 1 is an enforcement-set map. Moving from left to right marks the addition of independent enforcement layers; the same agentic components recur across families, but their primary locus shifts.

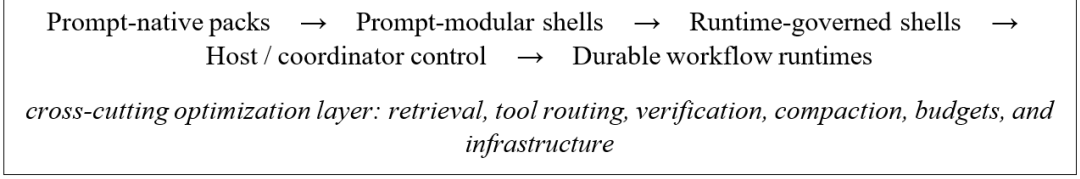


Fig. 1. Control-plane placement spectrum used in the comparison.

3.1. Memory as a Separate Comparison Axis

Although current agent writing often mentions memory, comparison still tends to fold it into context engineering or generic harness quality. Recent benchmark families argue against that simplification by isolating memory-specific performance or by testing whether remembered experience actually changes later action [28]–[34]. The key comparative lesson is that control-plane placement and memory substrate must be reported together but not collapsed: Pi externalizes planning state into artefact files; Hermes treats memory, search, and skills as runtime primitives; Honcho shows that memory services can be layered across shells rather than bound to one host [35]–[37].

For the purposes of this paper, memory substrate means the combination of content type and storage mechanism through which an agent system preserves and reuses experience. The field has converged on four content types grounded in cognitive science and formalized for language agents by

Sumers et al. (2023) [38]: working memory (active reasoning context), episodic memory (records of past events), semantic memory (general facts and preferences), and procedural memory (skills, workflows, action patterns). These are orthogonal to the storage mechanism, in-context (prompt-resident), in-cache (KV activations), in-weights (parametric), or external (retrieval stores, files, graphs), and to temporal scope (within-session versus cross-session) [39], [40]. In practice, agentic systems combine multiple substrate forms: bounded note stores, transcript search, knowledge-graph entity memory, procedural skill stores, repository-visible artefacts (PLAN.md, AGENTS.md), and structural/indexical memory in the form of AST-parsed codebase knowledge graphs [35], [41]–[46]. Distinguishing these from context policy clarifies a basic question: not only what is surfaced now, but also what content types and storage mechanisms can persist and later shape action.

4. COMPARATIVE CASES AND THE CODING MATRICES

The five representative families discussed in depth below are not exhaustive. Table 4 maps twelve actively maintained coding agents to their control-plane families, showing that the framework discriminates

meaningfully within what the ecosystem collectively calls “AI coding IDEs.” The host-mediated family is the most populated; Cursor, Windsurf, Kiro, and GitHub Copilot all fall there, reflecting the structural fact

that IDE (Integrated Development Environment) products naturally elevate control to the platform layer. The runtime-governed shell family has expanded to include Gemini CLI and Warp alongside Claude Code,

OpenCode, and Qwen Code. The notable ecosystem observation is that the five families still accommodate all current tools without residual; the taxonomy has not been broken by the 2025–2026 product wave.

4.1. Prompt-Native Systems

Prompt-native systems place all workflow logic in reusable skills, markdown procedures, and methodology artefacts that run inside whatever generic agent the user already has. The enforcement set contains only the document layer: governance holds only if the model follows instructions. Superpowers exemplify this family [47], [48], as do GitHub Copilot SKILL.md bundles and large curated prompt col-

lections [49], [50]. A point worth clarifying: CLAUDE.md and AGENTS.md are *instruction files loaded by agents in other families*, not prompt-native packs – they are the payload a runtime-governed shell reads at session start, not the shell itself. The characteristic limit of the prompt-native form is that documentation drift becomes behavioural drift when no independent enforcement layer is present.

4.2. Prompt-Modular Shell Control

Cline makes prompt composition itself visible as system architecture. Registry entries, prompt builders, role-dependent prompt sections, and Plan/Act mode semantics show that a full-capability coding agent can place much of its operating policy in prompt modules rather than in compiled runtime code [11]. Cline v3 adds meaningful governance structure on top: cline rules files specify per-project permissions (which directories the agent may modify, which tool categories require approval), and an SDK hook system enables lifecycle-level policy enforcement for auditing and

compliance. Critically, however, these rules are read and applied by the model at instruction time rather than enforced by a runtime kernel independently of model output – the structural distinction that keeps Cline in the prompt-modular family rather than in the runtime-governed family alongside Claude Code. This also illustrates a characteristic limit of the prompt-modular approach: governance strength depends on the model correctly following its instruction context, so when hard enforcement boundaries must hold robustly, runtime-layer restrictions are still needed alongside prompt-level rules.

4.3. Runtime-Governed Shells

OpenCode, Claude Code, and Qwen Code represent a stronger runtime-centred family. In these systems, the agent loop is visibly shaped by session processors, permissions, approval modes, compaction, subagents, and instruction-loading rules [7], [12], [13]. Prompts still matter,

but operational discipline is imposed more directly by the session runtime. This shifts analytical attention toward event handling, per-mission policy, anti-loop safeguards, context management, and instruction provenance rather than toward prompt wording alone.

4.4. Host-Mediated and Hierarchical Control

Some systems push control upward into the host or coordinator, adding a third independent enforcement layer. GitHub Copilot provides the clearest example: built-in roles, repository instructions, workspace summaries, approval-gated execution, and branch-and-pull-request workflows are all part of the effective control plane [8]. Cursor is the most governance-intensive instance: Cursor Hooks intercept execution

at the MCP call layer, an enterprise administration layer controls model access and spend across parallel agent VMs, and Mission Control manages concurrent workflows as a first-class surface. In all these cases, the decisive behaviour cannot be reduced to a single prompt or a single session runtime; the enforcement set spans prompt, runtime, and host simultaneously.

4.5. Durable Workflow Runtimes

Durable runtimes remain essential when resumability, inspectability, and governed continuation are themselves the core requirements. LangGraph, ADK workflow agents, and OpenAI workflow tooling expose that pole of the design space through graph structure, callbacks, persisted state, approval checkpoints, and resumable execution [9], [10], [51]. OpenAI Codex represents a fully cloud-hosted instance of this family: each task is assigned an isolated cloud sandbox (providing infrastructure-level enforcement independent of model output), a manager agent coordinates parallel worker

agents executing code across multiple files, and the structured workflow terminates in a GitHub pull request as the durable, reviewable output. Their comparative advantage is not replacing every agent shell, but making continuity and control explicit for long-running or approval-heavy work. Graph-structured execution also enables per-node context scoping: each workflow node or sub-agent receives precisely the state its step requires rather than the full accumulated conversation, making context management an explicit graph-design decision.

4.6. How the Saliency Coding is Assigned

The coding matrix marks how visibly central a given dimension is to each system family. A dimension is coded high when it is a primary site of operating policy or a strong enforcement mechanism. It is coded

medium when it materially shapes behaviour but is shared with other layers. It is coded low when it is present as support rather than as the main driver of system behaviour.

Table 2. Interpretive Rubric for Saliency Coding in the Comparison Matrix

Code	Interpretation	Practical reading
L	Present but secondary	The dimension exists, but another layer carries most operating control.
M	Important but shared	The dimension materially shapes behaviour, though not as the sole dominant locus.
H	Primary and strongly visible	The dimension is a major site of workflow policy, enforcement, or differentiation.

Table 2 operationalizes the comparison through a qualitative coding matrix. Its three purposes are pattern recognition across families (scanning a column reveals which families share a governance or mem-

ory profile), benchmark interpretation (two systems in different rows with similar leaderboard scores are likely measuring different things), and architecture selection.

Table 3. Qualitative Coding Matrix for Representative System Families

Representative family	Prompt control locus	Runtime control locus	Host/graph locus	Durable state	Governance visibility level	Optimisation leverage level
Prompt-native packs (superpowers, skill packs, prompt methodology collections)	H	L	L	M	M	M
Prompt-modular shells (Cline, KiloCode)	H	M	L	M	M	H
Runtime-governed shells (Claude Code, OpenCode, Qwen Code, Gemini CLI, Warp)	M	H	L	M	H	H
Host-mediated / coordinator forms (GitHub Copilot, Cursor, Windsurf, Kiro [†])	M	M	H	H	H	H
Durable workflow runtimes (LangGraph, Vertex AI ADK, OpenAI Codex)	L	H	H	H	H	M

Three patterns are immediately visible. First, similar primitives recur across system families, but their primary locus changes; the enforcement set grows rather than the primitives themselves changing. Second, stronger governance always accompanies stronger runtime or host involvement; prompt-native systems can implement meaningful gates through artefacts, but those gates depend on model compliance rather than independent enforcement. Third, the optimisation levers most likely to affect benchmark scores, retrieval policy, tool routing, verification, compaction, and budget control, which are most accessible when the runtime or host exposes them as

explicit, configurable surfaces.

The taxonomy handles hybrid configurations by identifying the dominant control locus, the enforcement layer carrying operating policy for the majority of agent decisions, and noting secondary layers where present. Warp exemplifies this: its terminal runtime enforces a command denylist with absolute precedence (runtime-governed primary) while its Oz cloud orchestrator adds coordinator-level back-ground management (host-mediated secondary). Kiro (AWS) presents the most structurally distinctive profile: specification artefacts serve as source-of-truth control documents (elevating Prompt locus unusually high for

an infrastructure-backed tool), while AWS IAM enforces permissions at the cloud layer (Host locus), two reinforcing loci rather than a single dominant one.

Table 4 applies the classification to the broader current ecosystem of actively maintained coding agents, providing a reference for readers placing unfamiliar tools within the framework. Warp: the primary locus is runtime-governed (denylist); the secondary locus is host-mediated (Oz orchestrator). Kiro: unique dual-high profile: Prompt locus (spec artefacts) and Host locus (AWS IAM) both elevated simultaneously.

Table 5 adds the companion memory-oriented view. It makes the second dimension of divergence visible: some systems rely on transparent repository artefacts, some on runtime-native memory and search, some on platform-scale prompt and session surfaces, and some on persisted workflow state.

The memory-native provider space illustrates that memory substrate choice is an infrastructure and sovereignty decision as much as a recall-quality one. Providers in

the Hermes ecosystem span cloud-hosted vector stores with automated fact extraction (Mem0, RetainDB), knowledge-graph synthesis (Hindsight), dialectic developer-intent modelling (Honcho), git-diffable local Mark-down (ByteRover), Holographic Reduced Representation algebra for air-gapped deployments (Holographic), Ollama-native tiered file hierarchies (OpenViking), and context-fenced ingest (Supermemory) [36], each carrying different cost, auditability, and latency profiles.

A distinct substrate relevant to coding agents is structural/indexical memory: persistent, queryable representations of the codebase itself rather than of past interactions. Tools such as codebase-memory-mcp [52], graphify [53], and cocoindex-code [54] build AST-parsed knowledge graphs of a repository, enabling architecture-aware edits and cross-file impact analysis at low token cost. This form is orthogonal to episodic recall; it indexes what the codebase *is* rather than what the agent has *done*.

4.7. Limitations and Threats to Validity

The method is qualitative and interpretive. Four threats should be noted. (1) Documentation gap: official sources do not always reflect full deployment state; this is mitigated by restricting foundational claims to Level 1 and 2 evidence. (2) System evolution: the matrix reflects evidence as of early 2026; the rubric (Table 2) provides the update procedure, and a worked example is available in the companion

repository. (3) Unvalidated coding: L/M/H ratings are the authors' considered judgment; a different analyst could reasonably code some entries differently, particularly at the M/H boundary. (4) Scope: cases are drawn from coding-agent contexts; applicability to research agents, enterprise workflows, or multi-modal settings has not yet been tested.

5. BENCHMARKS AS SYSTEM MEASUREMENT

The control-plane view changes how benchmark results should be interpreted. In

coding tasks, the measured object is usually not the model in isolation. Once durable

experience becomes a design variable in its own right, the measured object is better described as: model $\times$ reasoning policy

$\times$ harness/runtime $\times$ memory substrate $\times$ environment, under explicit tool, budget, and infrastructure conditions.

Table 4. Extended Ecosystem Classification: Active Coding Agents Mapped to Control-Plane Families (as of mid-2026). RooCode is omitted; it was shut down in May 2026.

System	Control-plane family	Product type	Key architectural property
Claude Code	Runtime-governed shell	CLI / terminal	Permission modes enforced by session runtime, not model instruction; hook system for lifecycle events
OpenCode	Runtime-governed shell	CLI / terminal	Open-source, session-runtime-governed; architecture mirrors Claude Code
Qwen Code	Runtime-governed shell	CLI / terminal	Qwen-model CLI; session-runtime governance derived from OpenCode base
Gemini CLI	Runtime-governed shell	CLI / terminal	GEMINI.md context files; trusted-folders execution policy; sandbox mode; lighter runtime enforcement than Claude Code
Warp	Runtime-governed shell ⁺	Terminal / cloud	Denylist with absolute enforcement precedence (runtime); Oz cloud orchestrator for background agents (secondary host layer)
Cline	Prompt-modular shell	VS Code ext.	Plan/Act semantics and governance in structured prompt composition; cline rules are model-read instruction files
KiloCode	Prompt-modular shell	VS Code ext.	Superset of Cline; 500+ models, five agent modes, budget tracking; same instruction-level governance pattern
GitHub Copilot	Host-mediated	IDE / cloud	Repository rules, workspace summaries, branch and review workflows; platform-managed session lifecycle
Cursor	Host-mediated	IDE fork	Cursor Hooks intercept at execution layer; Enterprise admin, parallel Ubuntu VMs, Mission Control; strongest governance pro-file in the IDE family
Windsurf / Cascade	Host-mediated	IDE fork	Cascade persistent agent with temporal context tracking; IDE-level permission settings; acquired by Cognition (Devin team)
Kiro	Host-mediated [†]	IDE (AWS)	AWS Kiro; specs are source-of-truth control artefacts (high Prompt locus); AWS IAM permission enforcement; event-driven hooks on file-save / PR-open; Bedrock model routing
OpenAI Codex	Durable workflow runtime	Cloud agent	Each task in isolated cloud sandbox (infrastructure enforcement); manager-worker orchestration; structured clone $\rightarrow$ code $\rightarrow$ test $\rightarrow$ PR workflow; GitHub as host state

That formula is consistent with official evaluation guidance and with the structure

of current agent benchmarks [6], [24], [25], [55]. It is also consistent with vendor report-

ing that shows measurable effects from retrieval design, toolset trimming, reviewer topology, and infrastructure setup [56]–[58]. Recent memory-oriented benchmarks strengthen the same point from another direction by isolating long-horizon recall, action-conditioned memory use, procedural reuse, and coding-experience selection rather than treating them as invisible background variables [28]–[34], [59].

To make the formula concrete: two agent systems reporting comparable task

resolution rates on SWE-bench Verified, say, Claude Code (runtime-governed shell) and GitHub Copilot autonomous mode (host-mediated), differ on at least three of the five formula dimensions before any model difference is considered. Their enforcement sets are structurally distinct: the shell enforces governance through session-local permission modes and hook-based auditing; the host-mediated system enforces it through repository-level.

Table 5. Memory-Oriented Comparison across Representative System Forms

Representative form	Dominant memory substrate	Artefact memory	Procedural memory	Main trade-off
Prompt-native packs / artefact workflows	Repository files, reports, staged artefacts, skill instructions	High	Medium	Highly transparent and share-able, but discipline-sensitive and vulnerable to schema drift
Runtime-governed / prompt-modular shells (artifact-heavy configuration)	Session trees plus explicit repo artefacts such as PLAN.md and TODO.md	High	Low–Medium	Low context tax and strong observability, but depends on deliberate file maintenance
Memory-native runtimes (Hermes-style)	Bounded notes, transcript search, provider-backed memory, reusable skills	Medium	High	Richer recall and reuse, but higher write/select complexity and maintenance burden
Platform / host surfaces (OpenClaw / Honcho-style layering)	Injected prompt files, session tools, shared memory services, skills	High	Medium	Cross-surface flexibility, but heterogeneous enforcement and provenance challenges
Durable workflow runtimes	Persisted workflow state, checkpoint stores, resumable threads, approval nodes	Medium	Medium	Strongest continuity and governed resumption, but higher schema and runtime overhead
Structural/indexical memory services (codebase-memory-mcp [52], graphify [53], cocoindex-code [54])	AST-parsed knowledge graph of the codebase, call chains, type graphs, symbol indices	Low	High	Enables architecture-aware edits and cross-file impact analysis at low token cost; orthogonal to episodic/fact memory

A leaderboard score that reports only the resolution rate conflates these differences; the appropriate response is to publish enforcement-set variables alongside the score.

Public evidence supports this reading: benchmark outcomes can shift measurably when retrieval improves, tool exposure is narrowed, reviewer loops are added, or infrastructure conditions change [24], [25], [55]–[58],

[60]. Memory benchmarks add a parallel caution: MemoryArena shows that conversational-memory scores do not automatically transfer to multi-session agency, while SWE-Context-Bench shows that unfiltered prior experience can hurt as much as help [31], [34]. Benchmarks measure what they measure reliably; enforcement-set disclosure is what makes those measurements comparable across settings.

Table 6. Benchmark Interpretation Matrix

Benchmark style	Representative examples	Mostly fixed	Mostly variable	Main misreading if ignored
Shared-scaffold comparison	HumanEval, MBPP, LiveCodeBench	Test harness, language runtime, grading protocol	Underlying model or provider	Treating a harness-local Pass@k as a universal capability ranking
Common-environment agent comparison	SWE-bench, Terminal-Bench, RepoMod-Bench	Sandbox environment and grading protocol	Full agent-system design (harness, memory substrate, retry policy)	Mistaking a configured-system score for a model-only capability measurement
Vendor component study	GitHub Copilot productivity reports, Cursor retrieval studies, Anthropic engineering notes	Broader product family and infrastructure	One harness variable at a time (retrieval, tool exposure, reviewer loop)	Overgeneralizing a narrow gain to the full product or to other architectures

Minimum benchmark disclosure checklist for agentic coding evaluations	
Derived from the formula: $model \times reasoning\ policy \times harness/runtime \times memory\ substrate \times environment$	
☐	Model identifier and version
☐	Reasoning policy (e.g., ReAct loop, multi-agent, structured planning)
☐	Harness/runtime family and version
☐	Memory substrate configuration (artefact-only / runtime store / checkpoint)
☐	Tool exposure set and routing policy
☐	Token and context budget per task
☐	Retry policy and consistency budget
☐	Infrastructure conditions (timeout, compute, environment isolation)

Fig. 2. Recommended benchmark disclosure items. Analogous to model cards and dataset documentation standards, this checklist makes agentic benchmark scores substantially more comparable across evaluation settings.

Table 7. Memory-Sensitive Benchmark Anchors Relevant to Coding Agents

Benchmark family	What it isolates	Why it matters for this paper
LongMemEval / LoCoMo	Long-horizon recall, updating, and multi-session synthesis	Shows that longer context alone is not a sufficient proxy for durable memory quality
MemGym /Memor-yArena/ Mem2ActBench	Memory-isolated and action-conditioned memory use	Separates remembered experience from general tool-loop quality and tests whether memory changes later behaviour
SkillsBench/SWE-ContextBench	Procedural reuse and repository-experience selection	Connects memory directly to coding workflows, skill libraries, and experience reuse
LongMemEval-V2	Environment-experience reuse through coding-agent-style retrieval	Shows that file-based memory and coding agents can themselves become memory engines
SWE-ContextBench /cocoindex-code evaluations	Structural/indexical memory: AST-aware retrieval of code-base content	Isolates whether <i>codebase structure</i> rather than past-interaction recall enables correct cross-file edits; distinct from episodic memory quality

6. GOVERNANCE, INTEROPERABILITY, AND DESIGN IMPLICATIONS

If the enforcement set is central, interoperability and governance become technical layers rather than afterthoughts. MCP lowers the cost of exposing tools through a common protocol [61], but once that is easy the hard problem moves upward: which tools are exposed, which actions need approval, what instruction sources are trusted, and how failures are audited. These questions resolve differently across families: prompt-native systems encode governance as phase gates and checkpoints; runtime-governed shells make permissions and approval modes into visible policy surfaces; host-managed systems add repository rules and review boundaries; durable runtimes make governance explicit through callback nodes and resumable state. The enforcement-set principle makes this visible: a larger set means governance is structurally enforced rather than merely requested.

This point has direct empirical support. Research on AI-assisted code review has shown that when a model from the same family reviews its own generated output, correlated failure modes cause errors to echo rather than cancel [62]. A prompt-native system that relies on the same model both to generate and to verify its output cannot, by design, provide independent verification; the governance layer and the generating layer are architecturally the same layer. A runtime-governed shell or durable workflow runtime with a dedicated verification step referencing an external specification introduces a structural boundary that breaks this circularity. Control-plane placement therefore has direct implications for the structural strength of governance a system can offer, not only for its operational convenience.

One practical lesson follows from the comparison: complexity should be introduced on demand rather than by default. Strong models with clean tool exposure and disciplined verification can already support substantial automation. Extra topology: subagents, host orchestration, coordinator layers, or durable graphs, should be added when the task structure genuinely requires more control, persistence, or auditability.

The enforcement-set size also encodes a safety property. At the prompt-native end, the governing and generating layers are architecturally the same layer, so prompt injection and instruction-drift risks are highest. At the runtime-governed and durable-runtime ends, the permission system, hook pipeline, or checkpoint graph constitutes a structural boundary independent of model output. Teams with adversarial-input exposure or regulated-output obligations should weight enforcement-set depth explicitly when selecting an architecture.

The same progression applies to context management. Runtime shells apply compaction budgets and summarisation policies that bound context growth explicitly. Host-mediated systems inject structured workspace and repository summaries in place of raw conversation history. Orchestrators decompose context further: each sub-agent receives a scoped, task-relevant window, and the orchestrator governs what information passes between agents. This decomposition enables coherent operation over longer task horizons and across more parallel agents than any single accumulated context window can support – and it is why the Optimisation leverage column rises as the enforcement set grows.

Table 8 maps five task types to the

enforcement-set depth that best fits them; the rationale in each row follows directly

from the family’s position on the placement spectrum.

Table 8. Control-Plane Placement Selection Guide (see Table 3 for evidence basis)

Task type	Recommended family	Rationale
Short exploratory or high-tempo coding	Runtime-governed shell	Interactive permission management and compaction support fast iteration; governance is visible but lightweight
Method-heavy, artefact-driven workflows	Prompt-native/skill centric	Reusable skill packs and procedure artefacts maximise portability; governance encoded in explicit check-points
Repository, branch, or pull-request workflows	Host-mediated/coordinator	Built-in review gates, branch policies, and session hand-offs are native to the host layer
Long-running or approval-required processes	Durable workflow runtime	Resumability, checkpoint-based continuity, and explicit callback nodes handle interruptions structurally
Multi-agent, cost-sensitive tasks	Hierarchical orchestrator design	Orchestrator capacity drives quality [63]; hierarchical designs reach near-equivalent accuracy at substantially lower cost [64]

These heuristics have empirical grounding. Orchestrator capacity drives system quality more than per-subagent scale [63], and hierarchical supervisor-worker architectures achieve approximately 97.8 % of reflexive-architecture accuracy at 60.9 % of the cost [64], a concrete illustration that enforcement-set placement has measurable

performance and budget consequences. Moving toward the prompt-native end reduces infrastructure cost but raises token cost; moving toward durable runtimes reverses the balance. Teams should therefore choose placement with an explicit budget model alongside capability and governance considerations.

7. CONCLUSION

All four findings below follow from a single architectural principle: the enforcement set – the collection of layers that can independently constrain what an agent does. The five families in this paper represent the five stable configurations of that set, from a single document layer at the prompt-native end to a four-layer stack of prompt, runtime, host, and graph at the durable-workflow end. Once that principle is stated, the findings are its consequences rather than independent observations.

Finding 1: Control-plane placement and memory substrate are correlated but separable. Prompt-native systems tend toward artefact memory, durable runtimes toward checkpoint memory, and runtime-governed shells toward mixed substrates,

but the axes can vary independently. A runtime-governed shell can layer a memory-native service; a durable runtime can rely on PLAN.md-style artefact conventions. Treating them as a single “harness quality” score loses this distinction.

Finding 2: The placement spectrum encodes a flexibility–governance trade-off. Moving from prompt-native toward durable workflow runtimes reliably increases the strength of governance enforcement while reducing operational flexibility. This trade-off is structural, not incidental: at the prompt-native end the system trusts the model to self-govern; at the runtime-enforced end the substrate constrains behaviour independently of model output. Teams with auditability or adversarial-input

requirements should weight this trade-off explicitly when selecting an architecture.

Finding 3: Benchmark scores conflate multiple system dimensions. The formula $model \times reasoning\ policy \times harness/runtime \times memory\ substrate \times environment$ shows that leaderboard differences may reflect harness strength, verifier design, or infrastructure headroom as much as model capability. The actionable corollary is a community practice: requiring benchmark submissions to disclose the five formula dimensions would make scores substantially more comparable. This is analogous to model cards and dataset documentation standards already established in the field.

Finding 4: Orchestrator placement, not per-agent scale, drives system quality, and enables fine-grained context management. Empirical evidence [63], [64] confirms that the governing architectural layer sets the quality ceiling. A well-placed orchestrator with specialised sub-agents can match or exceed a single larger model, and hierarchical designs can achieve near-equivalent accuracy to more expensive reflexive architectures at substantially lower cost. An underappreciated corollary is that orchestrators make context management architectural: each sub-agent operates in

a focused, scoped window rather than an accumulating shared conversation, and the orchestrator governs inter-agent information flow explicitly. This decomposition is what allows multi-agent systems to operate coherently over task horizons and codebase scales that overwhelm single-agent context windows. Control-plane placement is therefore not only a descriptive category but a design lever with measurable consequences for performance, cost, and context quality.

The qualitative matrices demonstrate how control-plane placement and memory substrate function as usable comparative instruments, with a rubric and evidence basis that support systematic updates and extensions to finer-grained rubrics, inter-rater validation, or non-coding agentic domains. Future work can extend the approach in three directions: longitudinal replication at regular intervals to track ecosystem evolution, domain extension beyond coding agents to research and enterprise workflow settings, and formalization of the five-variable formula as a structured benchmark reporting standard analogous to model cards, the highest-leverage next step for addressing the benchmark interpretation problem identified in Section 4.

REFERENCES

1. Liu, J. (2026, March 3). *LlamaIndex is more than a RAG framework: It is agentic document processing*. LlamaIndex. <https://www.llamaindex.ai/blog/llamaindex-is-more-than-a-rag-framework>.
2. Pan, L., Zou, L., Guo, S., Ni, J., & Zheng, H.-T. (2026). *Natural-language agent harnesses* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2603.25723>
3. Zhou, C., Chai, H., Chen, W., Guo, Z., Shan, R., Song, Y., . . . & Zhang, W. (2026). *Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2604.08224>
4. Yue, L., Bhandari, K. R., Ko, C.-Y., Patel, D., Lin, S., Zhou, N., ... & Pan, S. (2026). *From static templates to dynamic runtime graphs: A survey of workflow optimization for LLM agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2603.22386>

5. Rombaut, B. (2026). *Inside the scaffold: A source-code taxonomy of coding agent architectures*. [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2604.03515>
6. Anthropic. (2026, January 9). *Demystifying evals for AI agents*. <https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents>.
7. Anthropic. (2025–2026). *Claude Code materials, including How Claude Code works, Create custom subagents, Agent SDK overview, and Modifying system prompts*. <https://code.claude.com/docs/en/how-claude-code-works> (documentation set).
8. GitHub / Microsoft / VS Code team. (2025–2026). *Copilot agent-mode and coding-agent materials, including Introducing GitHub Copilot agent mode (preview), Local agents in Visual Studio Code, and About GitHub Copilot cloud agent*. <https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode> (documentation set).
9. OpenAI. (2025–2026). *Agents SDK and workflow tooling materials, including the Building agents track, Agents SDK guide, Agent Builder guide, and Results and state guide*. <http://developers.openai.com/tracks/building-agents> (documentation set).
10. Google. (2025–2026). *Agent Development Kit (ADK) materials, including workflow agents, callbacks, and skills documentation*. <https://google.github.io/adk-docs/agents/workflow-agents/> (documentation set).
11. Cline. (2025–2026). *Documentation and repository materials, including Plan & Act docs and prompt architecture*. <https://docs.cline.bot/> (see also the project repository at <https://github.com/cline/cline/>)
12. Anomalyco. (2026). *OpenCode repository and session-runtime sources*. <https://github.com/anomalyco/opencode>
13. QwenLM. (2025–2026). *Qwen Code materials, including Approval Mode, Subagents, Settings, and repository docs*. <https://qwenlm.github.io/qwen-code-docs/en/users/features/approval-mode/> (documentation set).
14. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., & Zhou, D. (2022). *Chain-of-thought prompting elicits reasoning in large language models* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2201.11903>
15. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). *ReAct: Synergizing reasoning and acting in language models* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2210.03629>
16. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). *Tree of thoughts: Deliberate problem solving with large language models* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2305.10601>
17. Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., ... & Hoefler, T. (2023). *Graph of thoughts: Solving elaborate problems with large language models* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2308.09687>
18. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegrefe, S., ... & Clark, P. (2023). *Self-refine: Iterative refinement with self-feedback* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2303.17651>
19. Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). *Reflexion: Language agents with verbal reinforcement learning* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2303.11366>
20. Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., & Weston, J. (2023). *Chain-of-verification reduces hallucination in large language models* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2309.11495>
21. Fricke, F., Malberg, S., & Groh, G. (2026). Framework of thoughts: A foundation framework for dynamic and optimized reasoning based on chains, trees, and graphs. In *Proceedings of the First Workshop on Structured Understanding, Retrieval, and Generation in the LLM Era (SURGeLLM 2026)* (pp. 132–151). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2026.surgellm-1.8>

22. Plaat, A., Wong, A., Verberne, S., Broekens, J., van Stein, N., & Bäck, T. (2025). Multi-step reasoning with large language models: A survey. *ACM Computing Surveys*, 58(6), 1–35. <https://doi.org/10.1145/3774896>
23. Bondarenko, A., Zamorskaja, N., Djakons, D., Djakona, A., & Djakons, R. (2026). *Rapid evolution of agentic AI: From framework labels to control surfaces* [Short overview]. Riga Nordic University.
24. SWE-bench project. (2024–2026). *Official benchmark materials, including the SWE-bench leaderboard and mini-SWE-agent*. <https://www.swebench.com/> (benchmark site and mini-SWE-agent materials).
25. Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., . . . & Schmidt, L. (2026). *Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.11868>
26. Dao, M.-D., Le, Q. M., Lam, H. T., Le, D.-T., Pham, Q.-V., O’Sullivan, B., & Nguyen, H. D. (2026). *Agentic design patterns: A system-theoretic framework* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.19752>
27. Milosevic, Z., & Rabhi, F. (2026). *Architecting agentic communities using design patterns* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.03624>
28. Xu, W., Wang, Y., Mei, K., Liang, K., Wang, Z., Jin, M., . . . & Metaxas, D. N. (2026). *MemGym: A long-horizon memory environment for LLM agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2605.20833>
29. Wu, D., Wang, H., Yu, W., Zhang, Y., Chang, K.-W., & Yu, D. (2024). *LongMemEval: Benchmarking chat assistants on long-term interactive memory* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2410.10813>
30. Maharana, A., Lee, D.-H., Tulyakov, S., Bansal, M., Barbieri, F., & Fang, Y. (2024). *Evaluating very long-term conversational memory of LLM agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2402.17753>
31. He, Z., Wang, Y., Zhi, C., Hu, Y., Chen, T.-P., Yin, L., . . . & Pentland, A. (2026). *MemoryArena: Benchmarking agent memory in interdependent multi-session agentic tasks* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2602.16313>
32. Shen, Y., Li, K., Zhou, W., & Hu, S. (2026). *Mem2ActBench: A benchmark for evaluating long-term memory utilization in task-oriented autonomous agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.19935>
33. Li, X., Liu, Y., Chen, W., You, B., Di, Z., He, Y., . . . & Song, D. (2026). *SkillsBench: Benchmarking how well agent skills work across diverse tasks* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2602.12670>
34. Zhu, J., Wu, J., Hu, M., Zhu, S., Pan, J., Shen, W., . . . & Xu, M. (2026). *SWE Context Bench: A benchmark for context learning in coding* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2602.08316>
35. pi.dev/Zechner, M. (2025–2026). *Pi coding agent materials, including the official site and architectural write-up on explicit artifact memory*. <https://pi.dev/> (see also <https://mariozechner.at/posts/2025-11-30-pi-coding-agent/>)
36. Nous Research. (2026). *Hermes Agent official materials*. <https://hermes-agent.nousresearch.com/>
37. Honcho. (2026). *Honcho design patterns and memory-service architecture materials*. <https://honcho.dev/docs/v3/documentation/core-concepts/design-patterns>
38. Sumers, T. R., Yao, S., Narasimhan, K., & Griffiths, T. L. (2023). *Cognitive architectures for language agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2309.02427>
39. Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2023). *MemGPT: Towards LLMs as operating systems* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2310.08560>
40. Du, P. (2026). *Memory for Autonomous LLM Agents: Mechanisms, evaluation, and emerging frontiers*. [Preprint]. arXiv. <https://arxiv.org/abs/2603.07670>
41. Nous Research. (2026). *Hermes Agent memory documentation*. <https://hermes-agent.nousresearch.com/docs/user-guide/features/memory>

42. Fang, J., Xu, B., Wang, Z., Cao, H., Deng, X., Dong, B., ... & Zhang, N. (2026). *Rethinking memory as continuously evolving connectivity* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2605.28773>
43. Chen, H., Zhang, Z., Pei, W., He, B., Wu, M., Zeng, J., ... & Zhang, H. (2026). *MemForest: An efficient agent memory system with hierarchical temporal indexing* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2605.23986>
44. Latimer, C., Boschi, N., Neeser, A., Bartholomew, C., Srivastava, G., Wang, X., & Ramakrishnan, N. (2025). *Hindsight is 20/20: Building agent memory that retains, recalls, and reflects* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2512.12818>
45. Yu, Y., Yao, L., Xie, Y., Tan, Q., Feng, J., Li, Y., & Wu, L. (2026). *Agentic memory: Learning unified long-term and short-term memory management for large language model agents* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.01885>
46. OpenClaw project. (2026). *OpenClaw repository and platform materials*. <https://github.com/openclaw/openclaw>
47. obra. (2025–2026). *superpowers repository*. <https://github.com/obra/superpowers>
48. glebis. (2025–2026). *claude-skills repository*. <https://github.com/glebis/claude-skills>
49. Anthropic. (2025–2026). *skills repository and marketplace materials*. <https://github.com/anthropics/skills>
50. GitHub. (2025–2026). *awesome-copilot repository*. <https://github.com/github/awesome-copilot>
51. LangChain. (2025–2026). *LangGraph materials, including the LangGraph overview, persistence docs, and workflows-and-agents documentation*. <https://docs.langchain.com/oss/python/langgraph/overview> (documentation set).
52. Deus Data. (2026). *codebase-memory-mcp: High-performance code intelligence MCP server with persistent knowledge graph*. <https://github.com/DeusData/codebase-memory-mcp> (see also the preprint at <https://arxiv.org/abs/2603.27277>).
53. Shamsi, S. (2026). *graphify: AI coding assistant skill for building queryable knowledge graphs from any codebase*. [Computer software]. GitHub. <https://github.com/safishamsi/graphify>.
54. CocoIndex. (2026). *cocoindex-code: Lightweight embedded AST-based code search engine and MCP server for coding agents*. [Computer software]. GitHub. <https://github.com/cocoindex-io/cocoindex-code>.
55. Li, X., Ben-Israel, N., Raz, Y., Ahmed, B., Serebro, D., & Raux, A. (2026). *RepoMod-Bench: A benchmark for code repository modernization via implementation-agnostic testing* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2602.22518>
56. Heule, S., Jia, E., & Jain, N. (2025, November 6). *Improving agent with semantic search*. Cursor. <https://cursor.com/blog/semsearch>
57. Agarwal, A., & Peet, C. (2025, November 19). *How we're making GitHub Copilot smarter with fewer tools*. The GitHub Blog. <https://github.blog/ai-and-ml/github-copilot/how-were-making-github-copilot-smarter-with-fewer-tools/>
58. Anthropic. (2026, February 5). *Quantifying infrastructure noise in agentic coding evals*. <https://www.anthropic.com/engineering/infrastructure-noise>
59. Wu, D., Ji, Z., Kawatkar, A., Kwan, B., Gu, J.-C., Peng, N., & Chang, K.-W. (2026). *LongMemEval-V2: Evaluating long-term agent memory toward experienced colleagues* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2605.12493>
60. Wang, H., Mang, Q., Cheung, A., Sen, K., & Song, D. (2026, April 8). *How we broke top AI agent benchmarks: And what comes next*. Berkeley RDI. <https://rdi.berkeley.edu/blog/trustworthy-benchmarks-cont/>
61. Anthropic. (2024, November 25). *Introducing the Model Context Protocol*. <https://www.anthropic.com/news/model-context-protocol>
62. Zietsman, C. (2026). *The specification as quality gate: Three hypotheses on AI-assisted code review* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2603.25773>

63. Żywot, A., Chen, X., Yuan, Y., Søgaard, A., & de Rijke, M. (2026). *Can small agents collaborate to beat a single large language model?* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2601.11327>
64. Kulkarni, S., & Kulkarni, Y. (2026). *Benchmarking multi-agent LLM architectures for financial document processing: A comparative study of orchestration patterns, cost-accuracy tradeoffs and production scaling strategies* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2603.22651>