Position: Stop Hardcoding Multi-Agent Workflows That General Agents Will Outgrow

Xisen Wang [1]

Abstract

The agentic AI ecosystem is making a costly bet: that coordination logic written today will remain valid as base agents improve. We argue this bet is wrong. Static multi-agent workflows hardcode roles, routing, and decomposition against capability snapshots that shift with every model generation, producing coordination debt: systems that work for one model, one task, and one moment, but fail to adapt as agents improve or fail in unanticipated ways. This is not an argument against multi-agent systems. It is an argument that the field is doing multi-agent coordination at the wrong layer. LLM agents differ from traditional software components: their capability boundaries are fuzzy, overlapping, context-dependent, and often undiscoverable before execution, and their failures are semantic, plausible, and retry-resistant. We therefore argue that coordination should move from static application logic to runtime infrastructure: discovery, verification, trust, and re-routing as first-class primitives. We synthesize evidence from major agent frameworks, recent decentralized coordination work, and proof-of-concept experiments showing that retry and fixed fallback collapse under systematic and shifting capability failures, while runtime verification and re-routing recover. We call for multi-agent systems that scale with general agents, not workflows that general agents outgrow.

1. Introduction

In 2024, a startup builds a four-agent coding pipeline: Planner, Coder, Reviewer, Tester. Roles hardcoded, routing fixed. It works with GPT-4. Six months later, the team upgrades to a stronger model. The Planner's decompositions are too fine-grained because the new model handles larger chunks natively. The Reviewer catches errors the Coder no longer makes. The pipeline now performs worse than a single agent with the new model. The team rewrites everything. Six months later, they rewrite again.

This pattern is visible across the agentic AI ecosystem. Frameworks such as CrewAI, LangGraph, AutoGen, MetaGPT, and ChatDev expose roles, graphs, or workflows that fix at least part of decomposition or interaction topology at design time (CrewAI Inc., 2024; LangChain AI, 2024; Wu et al., 2024; Hong et al., 2024; Qian et al., 2024; Derouiche et al., 2025). At the product layer, low-code agent builders and vertical workflow products make the same bet: coordination logic is written first and expected to remain valid as models improve. The investment is enormous, and it is producing systems with a shelf life measured in months.

The problem is deeper than wasted tokens. Static workflows assume that agent capabilities are knowable before execution: Planner plans, Coder codes, Reviewer reviews. But LLM-agent capabilities are not stable API contracts. They are fuzzy, overlapping, and context-dependent. An agent may solve 90% of a task family and fail deterministically on the remaining 10%, while returning outputs that *look* correct. These failures are not caught by health checks and are not fixed by retry. They are *systematic semantic failures*: plausible-but-wrong outputs that reproduce identically on each call. They require semantic verification and, when failure is systematic, routing to a differently capable agent.

This paper is not an argument against multi-agent systems. Multi-agent coordination is necessary and valuable. The question is not *whether* to coordinate multiple agents, but *how*. We argue that the dominant approach, static coordination hardcoded against a specific model's limitations, is structurally wasteful. The field should instead invest in *runtime coordination infrastructure* that scales with general agents.

Position. Static workflows are useful data paths, but poor control planes. General agents need runtime coordination infrastructure that can discover, verify, trust, and re-route as capabilities change.

This paper makes four contributions: (1) identifying static

[1]University of Oxford, Oxford, United Kingdom. Correspondence to: Xisen Wang <xisen.wang@keble.ox.ac.uk>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

coordination as *coordination debt* that becomes stale as general agents improve, (2) explaining why LLM agents make this debt uniquely severe (fuzzy, non-stationary, undiscoverable capability boundaries and semantic, systematic, retry-resistant failures), (3) synthesizing evidence across frameworks, decentralized coordination research, and proof-of-concept experiments showing where retry, voting, and fixed fallback fail, and (4) proposing runtime coordination infrastructure (discovery, verification, trust, re-routing) as the layer that should replace task-specific workflow logic.

2. The Static Coordination Bet

We define *static coordination* as any system in which three properties are fixed before task execution. **P1**: task decomposition is specified by the designer. **P2**: agent interaction patterns (who talks to whom) are predetermined. **P3**: verification and recovery are deferred, centralized, or absent.

This applies far beyond academic frameworks. Visual workflow builders such as Dify, n8n, Flowise, and Coze let users draw or configure static workflows and DAGs (LangGenius, 2025; n8n GmbH, 2025; FlowiseAI, 2025; Coze, 2025); products that route queries through predetermined agent sequences instantiate the same pattern. Table 1 maps five major frameworks to these properties and their documented coordination constraints.

This pattern is unfolding amid a broader wave of AI investment: over $250B in total corporate AI investment in 2024 (Maslej et al., 2025) and a 6× surge in enterprise generative AI spending (Menlo Ventures, 2024). GitHub's Octoverse 2025 reports 1.1M+ public repositories using an LLM SDK with 178% year-over-year growth (GitHub, 2025); our own survey (Appendix A) finds 190,000+ repositories referencing "AI agent" or "LLM agent" and a 133× increase in repositories mentioning "agentic workflow." Large open-source agent systems have also scaled quickly: Dify exposes visual workflow construction (LangGenius, 2025), while MetaGPT and AgentScope expose role- and platform-level multi-agent orchestration (Hong et al., 2024; Gao et al., 2024). These are different coordination surfaces, not a single shared visual-DAG design, but they share the same design-time commitment to workflow structure.

Observed coordination churn. The hypothetical in our introduction is not hypothetical at the ecosystem level: public coordination abstractions are changing quickly. Microsoft has converged AutoGen and Semantic Kernel work into Microsoft Agent Framework, explicitly positioning the new framework as a path from experimental orchestration to enterprise-ready runtimes (Microsoft, 2025). OpenAI presents its Agents SDK as a production-ready upgrade from the earlier Swarm experiment (OpenAI, 2024; 2025). The original BabyAGI repository was archived and snapshotted in September 2024 (Nakajima, 2024). These lifecycle shifts are not direct causal proof that static coordination failed in each project. They are ecosystem-level warning signs: the abstractions used to coordinate agents are moving targets, and workflow logic written against one capability snapshot can become stale faster than the surrounding product cycle.

Critical work is emerging that challenges the paradigm from the other direction: single agents with appropriate skills can remain competitive with multi-agent systems (Li, 2026), coordination benefits diminish once single-agent baselines become strong (Kim et al., 2025), and on constraint satisfaction, implicit pressure-based coordination dramatically outperforms explicit dialogue-based multi-agent communication (Rodriguez, 2026).

Static coordination is popular for good reasons, and the practitioners building these systems are not naive. It is predictable: you can trace execution before running. It is debuggable: when something fails, you know which agent failed. It is cheap at design time: specifying roles is faster than building discovery protocols. It is fundable: investors understand "we do X for domain Y" better than "we build infrastructure for arbitrary agents." These are real advantages, *under the assumption that the coordination logic will remain valid*. Our argument is not that practitioners are wrong to start with static coordination — it is often the correct engineering choice under uncertainty. Our argument is that the assumption of stable capabilities is false for LLM-based systems, because the substrate (model capabilities, tools, and framework abstractions) is non-stationary, and the field lacks infrastructure to evolve coordination without rewriting it.

3. Why LLM Agents Break Static Coordination

The tension between static and dynamic coordination is not new (Horling & Lesser, 2004; Durfee, 2001; Wooldridge, 2009). Our contribution is identifying properties of LLM agents that make the case for runtime coordination qualitatively stronger than it was for classical agents, and that make *static* coordination uniquely fragile.

3.1. Fuzzy capability boundaries and systematic semantic failure

Microservices have crisp capability contracts: a service either supports a query type or returns an error code (Newman, 2015). You know at design time what each component can do. LLM agents are fundamentally different. Their capabilities are **context-dependent**, meaning an agent may solve 95% of puzzles in one formulation but fail on slight input variations that cross an invisible capability boundary (Sclar et al., 2024; McCoy et al., 2024). They are **overlapping**,

Table 1. Static coordination properties across major frameworks. All fix decomposition (P1) and most fix topology (P2). Documented constraints or failures are drawn from original publications; root-cause labels are our reading of the coordination design.

Framework	P1	P2	P3	Documented constraint/failure	Root cause
MetaGPT	✓	✓	✓	Cascading hallucinations from naively chaining LLMs (Hong et al., 2024)	SOP workflow encodes handoffs
CAMEL	✓	✓	✓	Role flipping, infinite loops (Li et al., 2023)	Prompted turn-taking needs termination constraints
ChatDev	✓	✓	✓	Communicative dehallucination for code hallucinations (Qian et al., 2024)	Sequential chat-chain phases
AutoGen	✓	✓		Cost-effective workflow design remains open (Wu et al., 2024)	Conversation behavior is developer-programmed
AgentVerse	✓		✓	Static roles "hinder adaptability" (Chen et al., 2024)	Dynamic recruitment avoids rigid roles

meaning multiple agents may have partial capability for the same task with different failure patterns on different input subclasses (Srivastava et al., 2023). They are **undiscoverable at design time**, meaning you cannot know which inputs an agent will fail on without running it on those inputs. And they are **non-stationary**, meaning model and tool improvements shift capability boundaries over time such that a pipeline designed for one capability snapshot can become a straitjacket for a newer one (Maslej et al., 2025).

Worse still, when LLM agents fail, they fail in ways that static coordination cannot detect. Classical distributed systems rarely produce *plausible but wrong* results that pass format checks; even Byzantine faults are normally modeled as protocol-level inconsistencies rather than semantically valid but incorrect content (Lamport et al., 1982). LLM agents do produce such content. Models have stable failure patterns on certain input types, consistently violating specific constraints while satisfying others (McCoy et al., 2024). The output is well-formatted, passes basic checks, and only full semantic verification detects the error. The agent responds promptly, in correct format, with high confidence, so no health check, circuit breaker, or timeout (Nygard, 2007) catches it. Retrying or self-consistency-style resampling (Wang et al., 2023) can fail when the error reflects a persistent bias rather than transient noise.

Figure 1 illustrates the consequence: when a component fails in a static pipeline, errors propagate to all downstream components because the route was fixed before execution. Runtime coordination selects routes *during* execution based on observed results, so failure of one candidate does not block the task. Table 2 makes the underlying distinction precise, and Table 3 formalizes the failure taxonomy. The bottom two rows of the taxonomy (systematic semantic failure and capability absence) are the regime where runtime coordination adds value beyond classical fault tolerance.

The documented failures in Table 1 are not random noise that retry would fix. They are instances of the bottom two rows of Table 3. MetaGPT's "cascading hallucinations" are systematic semantic failures: the upstream agent produces plausible-but-wrong output that passes format checks, the downstream agent builds on it, and retrying the upstream agent reproduces the same hallucination because it reflects a stable reasoning pattern, not transient sampling noise. ChatDev's communicative dehallucination mechanism is an explicit acknowledgement that code hallucinations must be detected and corrected during the pipeline. CAMEL's infinite loops are the degenerate case where systematic failure goes undetected: both agents produce plausible turn-level outputs while the conversation diverges, and no verification gate exists to break the cycle. AgentVerse explicitly reports that static roles "hinder adaptability" when the task requires capabilities that cut across predetermined assignments (capability absence). In each case, the failure exhibits the defining properties of the bottom two taxonomy rows: the output appears well-formed, retry does not resolve it, and recovery requires either re-routing to a differently-capable agent or restructuring the pipeline. These are not edge cases cherry-picked from failure logs. They are documented in the original publications as limitations or failure modes rather than rare logging artifacts.

3.2. From mechanism to consequence: why brittleness and fault tolerance are the same problem

Combining fuzzy boundaries with systematic failure creates a coordination problem that *cannot* be solved at design time. You cannot enumerate which inputs each agent will fail on because boundaries are undiscoverable. You cannot enumerate which agent to route to on each failure because capabilities overlap unpredictably. You cannot rely on retry because failures are systematic and deterministic. And you cannot rely on health checks because failures are semanti-

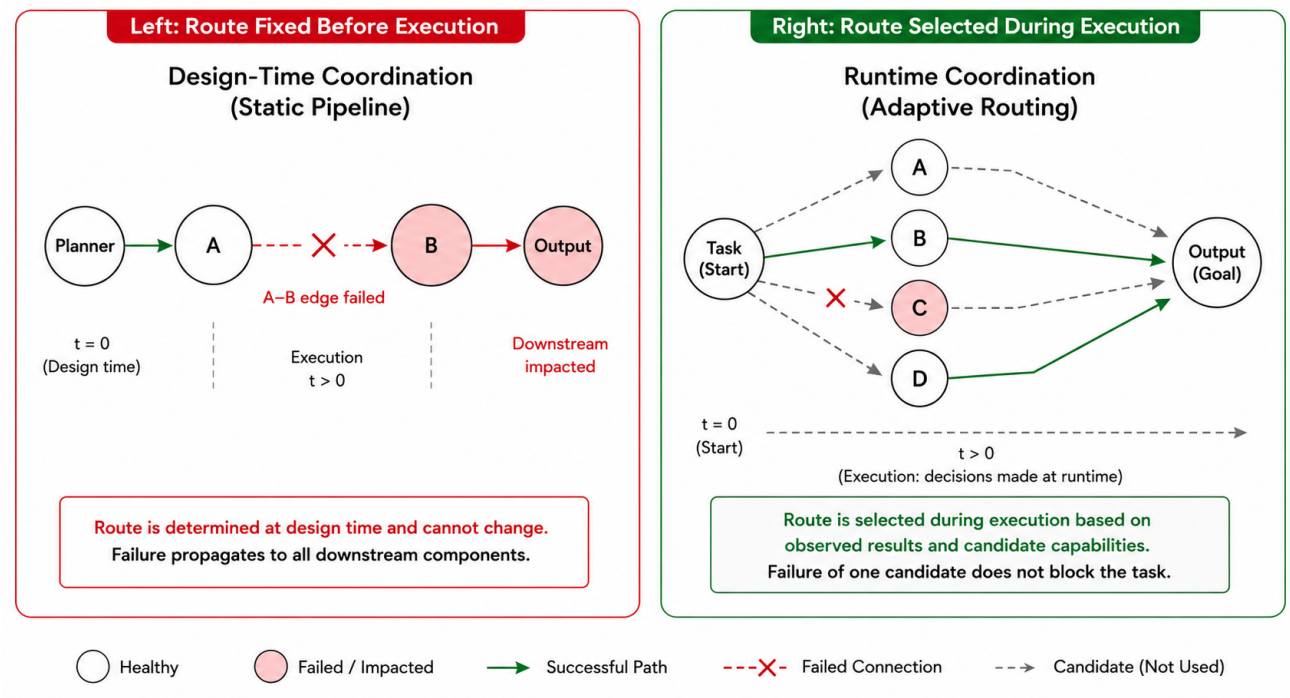

Figure 1. Design-time coordination (left) fixes the route before execution. When the A–B edge fails, errors propagate downstream with no recovery. Runtime coordination (right) selects routes during execution based on observed results and candidate capabilities. Failure of one candidate does not block the task.

cally invisible.

The only viable strategy is to run the agent, verify the output semantically, and if it fails systematically, route to a different agent at runtime. This is the causal chain from problem to solution: fuzzy boundaries plus systematic failure plus invisible errors necessitate runtime verification plus capability-aware re-routing.

Note that *temporal brittleness* (workflows breaking after model upgrades) and *within-generation fault tolerance* (handling systematic failures of today's model) are not separate problems requiring separate solutions. They are the same structural problem observed at different timescales. A model upgrade that shifts capability boundaries is, from the coordination layer's perspective, indistinguishable from discovering a new systematic failure on a new input class. In both cases, the previously-valid routing assumption is violated, the output looks correct but is wrong, and recovery requires verifying and re-routing. Runtime coordination infrastructure handles both because it makes no assumption about *when* or *why* a routing assumption became invalid — only that verification detected a failure and an alternative must be found.

Static pipelines are not useless. They are efficient *data paths* for predictable inputs. The problem is that they have no *control plane*: no mechanism to detect when assumptions fail, identify whether failure is systematic, and reconfigure routing in response.

3.3. Why rewriting workflows does not solve the problem

An immediate objection is that coordination debt can be serviced by periodic rewriting: when the model improves, update the workflow. This is true for any single team maintaining a single pipeline. But the argument for runtime coordination is not about any individual workflow. It is about the *ecosystem structure* that rewriting produces.

Consider what happens when many teams independently rewrite their workflows every model generation. Each rewrite is isolated. A legal-document-review team that discovers its Planner agent now over-decomposes tasks cannot share that insight with a medical-diagnosis team whose Planner has the same problem. A coding-agent startup that learns its Reviewer is now redundant cannot transfer that learning to a customer-service startup whose Verifier is equally redundant. Every rewrite is a local fix that produces local knowledge, trapped inside one organization's one pipeline. Millions of workflows cannot help each other because each encodes coordination against its own task, its own model version, and its own capability snapshot.

Runtime coordination infrastructure inverts this. When co-

Table 2. Why LLM agent coordination differs from microservice fault tolerance.

Property	Microservices	LLM Agents	Implication
Capability boundary	Fixed API contract	Fuzzy, overlapping, context-dependent	Cannot pre-determine who handles what
Failure signal	Crash, timeout, error code	Plausible-but-wrong output	Requires semantic verification
Failure pattern	Typically transient or total	Often systematic per input class	Retry reproduces the error
Recovery	Retry, failover to replica	Route to differently-capable agent	Replicas share the same bias
Routing decision	Health check (binary)	Quality judgment (continuous)	Requires output-level assessment

Table 3. Failure taxonomy for multi-agent LLM systems. Each type requires progressively more sophisticated coordination. The bottom two rows are the regime where runtime coordination is necessary.

Failure type	Example	Detected by	Retry fixes?	Required recovery
Crash fault	Timeout, garbage output	Format check	Yes	Simple retry
Random semantic	Wrong but different each call	Verification	Yes	Verification + retry
Systematic semantic	**Same wrong answer every call**	**Verification**	**No**	**Re-route to different agent**
Capability absence	Agent lacks skill entirely	Timeout or verif.	No	Discovery + re-route

ordination decisions are made at runtime based on observed verification outcomes and trust signals, the infrastructure itself accumulates knowledge that transfers across tasks. A trust signal that identifies an agent as unreliable on a certain input class applies to all future tasks in that class, not just the task that triggered the discovery. A verification primitive that detects constraint violations works for any domain where constraints can be checked, without requiring domain-specific workflow redesign. The difference between periodic rewriting and runtime infrastructure is the difference between isolated local optimizations and a shared adaptive layer whose value compounds across tasks, teams, and model generations.

3.4. Systematic failure need not be common to justify runtime coordination

A second objection is that systematic semantic failure may be rare in practice. Our own pilot study confirms this for Latin Square: GPT-5.5 solves all tested instances correctly. If frontier models rarely fail systematically, why invest in coordination infrastructure designed for this failure mode?

The answer is that the value of runtime coordination does not depend on failure being *frequent*. It depends on failure being *unpredictable*. After a model update, you do not know in advance which input classes will trigger systematic failure. The failure surface shifts with every capability change, and the shift is discontinuous: a model may gain strong multi-step reasoning while regressing on format adherence, or improve at code generation while degrading at code review. You cannot pre-design a fallback for a failure mode you have not yet observed.

This is the regime where runtime coordination is uniquely valuable: not the regime of frequent, known failures (where manual rewriting is tractable), but the regime of infrequent, unknown failures that emerge unpredictably after capability shifts. A workflow that handles 99% of inputs correctly will be rewritten for the 1% only after that 1% causes a production incident. Runtime coordination handles it at the moment of occurrence, before the incident propagates downstream, by verifying the output and routing to an alternative agent. The cost structure is asymmetric: the infrastructure is always on (adding modest overhead to the 99%), but it prevents catastrophic failures in the 1% that you could not have anticipated at design time.

Furthermore, the frequency argument has a selection bias. We observe few systematic failures in controlled settings precisely because production teams invest enormous engineering effort in prompt engineering, guardrails, and manual fallbacks to prevent them. This effort is itself coordination debt. It is the ongoing tax that static workflows impose. Runtime coordination replaces that tax with a general mechanism that does not require task-specific engineering.

Table 4. Two orthogonal axes of coordination design. Our argument is about *when* decisions are made (vertical), not *where* (horizontal).

	Centralized	Decentralized
Design-time	Fixed workflow with hardcoded fallbacks (MetaGPT, ChatDev)	Fixed local policies per agent (classical MAS)
Runtime	Adaptive orchestrator that verifies and re-routes on failure	Peer discovery + trust + capability-aware re-routing

The Coordination Lesson. Hand-engineered coordination (fixed roles, fixed routing, fixed decomposition) is the multi-agent equivalent of hand-engineered features (Sutton, 2019). It works for the current capability snapshot. It often fails to remain valid across model generations. The field should invest in coordination methods that *leverage* improving agents rather than methods that *assume* fixed agents.

The fine-tuning analogy clarifies this. Making something "vertical" by storing hundreds of workflows and proprietary routing logic is structurally equivalent to fine-tuning a foundation model. Both yield short-term gains on a narrow distribution but create frozen snapshots that diverge from the improving base. The analogy is not that fine-tuning or vertical workflows are inherently bad. Both are valuable when the target distribution is stable. The problem arises when the substrate changes faster than the coordination assumptions can be maintained.

4. Runtime Coordination Infrastructure

Our position is not that every decision should be dynamic. Static workflows remain valuable as efficient data paths when inputs are predictable and capabilities are stable. The missing layer is an adaptive *control plane*: a runtime coordination substrate that can verify outputs, update trust, discover alternatives, and re-route when static assumptions fail.

4.1. Axes of coordination and core primitives

Our argument operates on the *when* axis (when are coordination decisions made?), not the *where* axis (who makes them). Table 4 shows these are orthogonal. Each cell is a valid architecture. Our position is that the bottom row is necessary for robustness.

We identify four primitives that constitute minimal coordination infrastructure. **Discovery**: agents advertise capabilities in natural language and peers assess relevance without schema definitions, made newly cheap by LLMs as a zero-protocol-cost interface (Wooldridge & Jennings, 1995; Stone & Veloso, 2000). **Verification**: local, per-step output assessment that makes everything else safe (you cannot re-route intelligently without knowing whether the current result is acceptable). **Trust**: accumulated success and failure signals that convert reactive re-routing into predictive routing, updating automatically as agents improve or degrade. **Re-routing**: sending a task to a different agent when verification fails, combined with discovery and trust to find the best available agent at runtime.

Figure 2 shows how these primitives compose at the agent level. Each agent evaluates its own tools, skills, and memory against the incoming task. If confidence is high, it executes locally (path A). Otherwise, it discovers peer capabilities via metadata and delegates (path B). Outcomes update memory and trust for future routing decisions.

4.2. Design principles and supporting evidence

Coordination infrastructure should satisfy three properties. First, it must be **accessible by general agents**: any agent that can describe its capabilities in natural language, receive tasks, and return outputs should be able to participate, with no assumption about what any agent can do. Second, it must **scale with agent improvement**: better agents should produce better discovery, better verification, and better adaptation, and a coordination system that gets worse when you plug in a smarter agent has the wrong architecture. Third, it must **trust agent capabilities**: the coordination layer should set minimal structure and let agent capabilities determine the rest, rather than pre-decomposing tasks that a general agent can handle end-to-end (unless decomposition is needed for verification, parallelism, or safety).

Recent work suggests that runtime coordination is not merely theoretically appealing but technically plausible. Rodriguez (Rodriguez, 2026) demonstrates that agents operating locally on a shared artifact, guided only by pressure gradients derived from quality signals, dramatically outperform both hierarchical control and conversation-based multi-agent dialogue ($p < 0.001$). The approach maintains consistent performance as agent count scales without topology redesign. Chopra et al. (Chopra et al., 2025) show that lightweight "sensitivity signals" propagate through local networks to produce 41 to 100% improvement in coordination accuracy over standard agent-to-agent communication. Habiba and Khan (Habiba & Khan, 2025) argue that gossip protocols can provide adaptive coordination among agents that evolve beyond static roles, with a peer-to-peer structure that requires no central coordinator and naturally handles agents joining or leaving. The pattern across these results is that lightweight local coordination can be competitive with, and sometimes superior to, heavy explicit orchestration. Agents do not need to be told what to do. They need

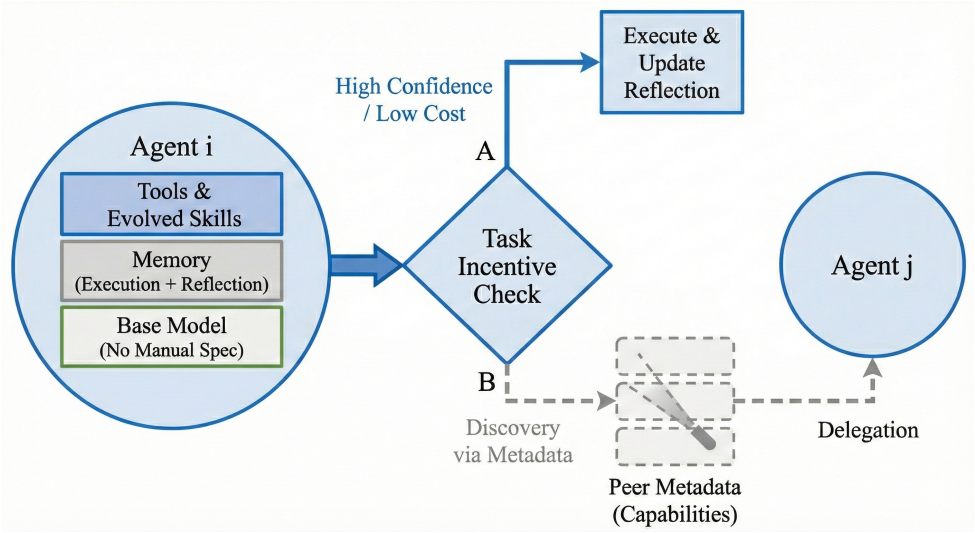

Figure 2. Local decision process. Agent i evaluates its tools, skills, and memory against the task. If confidence is high, it executes locally (path A). Otherwise, it discovers peer capabilities via metadata and delegates (path B). Outcomes update memory and trust for future routing decisions.

minimal signals about what matters, and the capability to act on those signals.

Classical multi-agent systems explored runtime coordination extensively (Horling & Lesser, 2004). The barrier was cost: discovery required shared ontologies and adaptation required multi-episode learning (Stone & Veloso, 2000). LLMs eliminate both barriers. Discovery through natural language is zero-cost. Adaptation within a single context window is instant. Test-time experience accumulation and tool-use guidance are feasible (Yang et al., 2025). The infrastructure that was impractical for classical agents is newly cheap for LLM agents.

4.3. Why this is not "just better error handling"

A natural objection is that runtime coordination reduces to well-engineered monitoring plus fallback logic — something any competent team already implements. The distinction is structural, not merely quantitative. Traditional error handling is *task-specific*: a team writes fallback logic for failure modes they have observed in their specific pipeline, against their specific model, for their specific inputs. When the model changes, the fallback assumptions may be wrong. When a new failure mode appears, no fallback exists until an engineer writes one.

Runtime coordination infrastructure is *task-general*: verification operates on output properties (constraint satisfaction, format adherence, semantic consistency), not on task-specific heuristics. Trust accumulates from observed outcomes regardless of task domain. Discovery finds agents by capability description, not by hardcoded lookup table. The coordination layer does not need to know what task it is coordinating — it needs to know whether the output passed verification and which agents are available.

The analogy to networking is precise. Every application could implement its own retry logic, its own connection pooling, its own load balancing. TCP/IP and DNS exist because doing this at the infrastructure layer produces compounding value: each application benefits from the same primitives without re-implementing them. The multi-agent ecosystem is currently at the stage where every team implements its own ad-hoc coordination. We argue for the infrastructure layer.

5. Proof-of-Concept Experiments

We provide controlled experiments isolating the mechanism our position depends on: static routing fails under systematic capability failures, while runtime verification and rerouting adapts. All experiments use Latin Square constraint satisfaction (4×4 through 6×6) with GPT-5.5 and GPT-5-mini, chosen because verification is exact and deterministic (eliminating evaluator noise). The decisive experiment uses N=100 puzzles replicated across three random seeds. We report bootstrap 95% CIs from 10,000 resamples. Full details in Appendix B.

5.1. The regime boundary: retry vs. re-routing

Under random faults (crash or different-each-call semantic errors), retry achieves **100%** recovery. Runtime coordination is unnecessary. This establishes the regime boundary: our position applies only when failure is *systematic*. We test this by marking 50% of puzzles as "poisoned": the Solver

deterministically produces a grid satisfying row constraints but violating column constraints. The output passes format checks and only full verification detects it. Table 5 shows the results.

Self-consistency ($3\times$ majority vote) produces identical outputs on 100% of poisoned puzzles, confirming systematic failure is a property of the agent's approach, not sampling noise. Re-routing achieves 100% recovery (Fisher's exact $p < 10^{-14}$).

5.2. Variable bias and majority voting

A natural objection: hardcode a fallback (Solver fails, route to Fixer). This works when the designer knows which agent is biased. When bias varies per input (as after a model update that shifts boundaries non-uniformly), no static fallback covers all cases. Table 6 tests this.

No static fallback covers all cases (54 to 58%), while runtime coordination achieves 100%. This also explains why majority voting fails under systematic bias: when $k > N/2$ agents share the same bias, majority vote selects the wrong answer with probability 1.0. The structural difference is fundamental: majority voting requires *most* agents to be correct, while verify-then-reroute requires only *one* correct agent plus a working verifier.

5.3. Robustness, trust convergence, and generalization

Runtime coordination depends on verification quality. We sweep verifier sensitivity s from 1.0 to 0.3 in simulation (1000 trials per setting). The system degrades gracefully: at s=0.5, recovery is still $8\times$ better than retry. The crossover where runtime equals retry occurs only below s=0.06. For constraint satisfaction where verification is exact, s is effectively 1.0. For softer tasks, even moderate verification ($s > 0.3$) yields substantial gains.

Over repeated tasks, trust weights converge to reflect true reliability. In simulation (5 agents, 1 persistently biased, 200 tasks), trust correctly identifies the biased agent within 10 tasks and routes directly to non-biased agents in 2 calls on average ($2\times$ fewer than exhaustive). When the biased agent is "fixed" (simulating a model update), trust re-converges within 8 tasks without manual intervention. Static coordination cannot learn this because it routes the same way on task 1 and task 1000.

To test generalizability, we replicate on HotpotQA (Yang et al., 2018) multi-hop QA (5 agents, N=30, GPT-5.5). Under node failure where the primary Searcher becomes unavailable, the static pipeline drops to **0%**. The runtime network discovers BackupSearcher and recovers to **77%** [60, 90] ($p < 10^{-9}$), demonstrating discovery-then-reroute for capability absence. Token overhead is $1.3\times$ on clean inputs, $1.8\times$ average at 50% fault rate. This is a per-query cost, whereas static pipelines impose a per-*generation* cost of months of engineering to redesign after each model update.

6. Discussion and Call to Action

We anticipate six objections and address each in turn.

(1) "Isn't this just classical MAS?" Classical MAS studied runtime coordination, but LLMs change the cost structure: natural language makes capability description cheap, context windows enable within-episode adaptation, and tool use allows dynamic specialization. The contribution is not the idea of runtime coordination but the argument that LLM-specific failure modes make it *necessary* rather than merely desirable.

(2) "Isn't this just centralized adaptive orchestration?" Centralized adaptive orchestrators are one valid form of runtime coordination. The broader position is that coordination decisions should be made at runtime. The centralized versus decentralized axis is orthogonal to the design-time versus runtime axis (Table 4).

(3) "Just try all agents exhaustively." An exhaustive static chain achieves equivalent accuracy in our 3-agent experiments, but scales as $O(N)$ per task versus $O(1)$ with trust-based routing, requires enumerating all agents at design time, and cannot adapt to agents added or removed without redesign.

(4) "Models won't keep improving this fast." Historically, betting on capability stagnation has been a losing bet (Sutton, 2019). Even moderate improvement rates (one significant jump per year) are enough to obsolete coordination designed for the previous generation.

(5) "What about safety and controllability?" Runtime coordination is not uncontrolled. Verification is a *gate*, not an afterthought. Every output is assessed before acceptance. Trust bounds limit delegation. Human oversight hooks can be placed at verification boundaries. Static coordination provides the illusion of control through predictability, but shatters under conditions its designer did not anticipate (Cemri et al., 2025; Li et al., 2023).

(6) "Don't routing benchmarks already solve this?" Multi-LLM routing systems (e.g., RouterBench (Hu et al., 2024)) optimize *which model* to call for cost-quality tradeoffs, typically via learned classifiers over input features. This is valuable but operates at a different level than what we propose. Routing benchmarks select models before execution based on input characteristics. Runtime coordination selects *agents* (model + prompt + tools + context) *during* execution based on observed output quality. The key difference: routing benchmarks assume you can predict performance from input features alone; our position is that

Table 5. Recovery from systematic faults (GPT-5.5, N=100, replicated 3×). Cross-model replication on GPT-5-mini confirms retry=0% and re-route=100%. Cohen's $h = 2.14$.

Condition	Recovery mechanism	Poisoned accuracy	Overall [95% CI]
Baseline	Retry same agent (3×)	4–6%	53% [43, 62]
Baseline	Self-consistency (3× majority)	0%	n/a
Reliable alternative	Re-route to Fixer	**100%**	**100%** [100, 100]
Degraded alternative	Re-route to Fixer (80% reliable)	**80%**	**90%** [84, 95]

Table 6. Variable poisoning (N=50, GPT-5.5). When the biased agent varies per input, no static fallback achieves full recovery. Runtime coordination discovers the working path dynamically.

	Known bias (controls)		Unknown bias
Topology	S+F biased	S+A biased	**Mixed (50/50)**
Static S → Fixer	12%	100%	54% [40, 68]
Static S → Alt	100%	12%	58% [44, 72]
Runtime Adaptive	**100%**	**100%**	**100%** [100, 100]

systematic semantic failures are often unpredictable from input features and only detectable post-execution via verification. The two are complementary: routing benchmarks can serve as the discovery heuristic that provides an initial routing guess, while runtime coordination handles the cases where that guess fails.

Deployment boundary. The intended deployment pattern is not to replace every static workflow with an expensive search over agents. Static pipelines should remain the fast path when the input distribution is stable, latency is tight, and failures are cheaply detectable. Runtime coordination is the control plane around that path: it activates at verification boundaries, after low-confidence outputs, under model or tool version changes, or for high-risk tasks where accepting a plausible wrong answer is more expensive than checking it. In small fixed agent pools, exhaustive fallback may be simpler than learning trust; the value of trust and discovery grows when agents, tools, models, or tasks change. This distinction matters because the cost of runtime coordination should be paid where uncertainty is high, not uniformly on every token.

Verification boundary. Verification also need not be a perfect semantic oracle. In realistic systems it will often be partial and typed: schema checks, unit tests, formal constraints for structured outputs, retrieval consistency checks, executable tool calls, decomposition into checkable subclaims, or human approval at high-stakes boundaries. The runtime layer should expose verifier uncertainty rather than hiding it. If all candidates fail verification, or if verifier confidence is itself low, the correct action is abstention, escalation, or human review, not blind re-routing. This is also where safety enters the design: discovery advertises capability but does not grant authority; trust updates should be bounded, auditable, and scoped to task classes so that compromised or adversarial agents cannot accumulate unbounded delegation power.

Limitations. Our experiments use synthetic fault injection to model the *observable behavior* of systematic bias, not naturally-occurring model biases. GPT-5.5 solves all tested Latin Squares correctly, which is precisely why controlled injection is necessary to study recovery mechanisms in isolation. The experiments estimate recovery conditional on systematic failure; they do not estimate the natural prevalence of such failures in deployed frontier systems. We test on constraint satisfaction and QA, not open-ended generation. Verification without ground truth remains the key bottleneck for broader deployment.

Research agenda. Two directions are critical. The first is benchmarks that inject systematic failures and measure graceful degradation. The second is verification without ground truth, including LLM-as-judge, formal verification for structured outputs, and partial verification. The coordination infrastructure we describe is a set of design principles rather than a complete system, but related primitives (discovery, verification, trust, re-routing) are being explored across recent work (Habiba & Khan, 2025; Rodriguez, 2026; Tarasova et al., 2025), and composing them into a first-class runtime layer is the natural next step.

Conclusion. The agentic AI ecosystem is investing heavily in coordination logic that often fails to remain valid across model generations. The alternative is coordination infrastructure that adapts at runtime: discover what agents can do, verify what they produce, learn whom to trust, and re-route when trust is violated. The coordination layer should be the part of the system that *improves* when you plug in a better model, not the part that *breaks*. Build coordination that works *with* general agents. Build coordination that scales *as* general agents improve. Stop hardcoding workflows that general agents will outgrow.

References

Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E., and Stoica, I. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Chan, C.-M., Yu, H., Lu, Y., Hung, Y.-H., Qian, C., Qin, Y., Cong, X., Xie, R., Liu, Z., Sun, M., and Zhou, J. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *ICLR*, 2024.

Chopra, A., Sharma, A., Ahmad, F., Muscariello, L., Pandey, V., and Raskar, R. Ripple effect protocol: Coordinating agent populations. *arXiv preprint arXiv:2510.16572*, 2025.

Coze. Workflow overview. Product documentation, 2025. URL https://www.coze.com/open/docs/guides/workflow.

CrewAI Inc. CrewAI: Fast and flexible multi-agent automation framework. GitHub repository crewAIInc/crewAI, 2024. URL https://github.com/crewAIInc/crewAI.

Derouiche, H., Brahmi, Z., and Mazeni, H. Agentic AI frameworks: Architectures, protocols, and design challenges. *arXiv preprint arXiv:2508.10146*, 2025.

Durfee, E. H. Distributed problem solving and planning. In *Multi-Agent Systems and Applications*, pp. 118–149. Springer, 2001. doi: 10.1007/3-540-47745-4_6.

FlowiseAI. Flowise: Drag & drop UI to build customized LLM flow. GitHub repository FlowiseAI/Flowise, 2025. URL https://github.com/FlowiseAI/Flowise.

Gao, D., Li, Z., Pan, X., Kuang, W., Ma, Z., Qian, B., Wei, F., Zhang, W., Xie, Y., Chen, D., Yao, L., Peng, H., Zhang, Z., Zhu, L., Cheng, C., Shi, H., Li, Y., Ding, B., and Zhou, J. AgentScope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*, 2024.

GitHub. Octoverse 2025: A new developer joins GitHub every second as AI leads TypeScript to #1. Technical report, GitHub Inc., 2025. URL https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/.

Habiba, M. and Khan, N. I. Revisiting gossip protocols: A vision for emergent coordination in agentic multi-agent systems. *arXiv preprint arXiv:2508.01531*, 2025.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024.

Horling, B. and Lesser, V. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2004. doi: 10.1017/S0269888905000317.

Hu, Q., Bieker, J., Li, X., Jiang, N., Keigwin, B., Ranganath, G., Keutzer, K., and Upadhyay, S. RouterBench: A benchmark for multi-LLM routing system. *arXiv preprint arXiv:2403.12031*, 2024.

Kim, Y., Gu, K., Park, C., Park, C., Schmidgall, S., Heydari, A. A., Yan, Y., Zhang, Z., Zhuang, Y., Liu, Y., Malhotra, M., Liang, P. P., Park, H. W., Yang, Y., Xu, X., Du, Y., Patel, S., Althoff, T., McDuff, D., and Liu, X. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025.

Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. doi: 10.1145/357172.357176.

LangChain AI. LangGraph: Low-level orchestration framework for stateful agents. GitHub repository langchain-ai/langgraph, 2024. URL https://github.com/langchain-ai/langgraph.

LangGenius. Dify: Open-source LLM app development platform. GitHub repository langgenius/dify, 2025. URL https://github.com/langgenius/dify.

Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., and Ghanem, B. CAMEL: Communicative agents for "mind" exploration of large language model society. In *NeurIPS*, 2023.

Li, X. When single-agent with skills replace multi-agent systems and when they fail. *arXiv preprint arXiv:2601.04748*, 2026.

Maslej, N., Fattorini, L., Perrault, R., Gil, Y., Parli, V., Kariuki, N., Capstick, E., Reuel, A., Brynjolfsson, E., Etchemendy, J., Ligett, K., Lyons, T., Manyika, J., Niebles, J. C., Shoham, Y., Wald, R., and Clark, J. The 2025 AI index report. Technical report, Stanford Institute for Human-Centered AI, 2025. URL https://hai.stanford.edu/ai-index/2025-ai-index-report.

McCoy, R. T., Yao, S., Friedman, D., Hardy, M. D., and Griffiths, T. L. Embers of autoregression show how large language models are shaped by the problem they are trained to solve. *Proceedings of the National Academy of Sciences*, 121(41):e2322420121, 2024. doi: 10.1073/pnas.2322420121.

Menlo Ventures. 2024: The state of generative AI in the enterprise. Technical report, Menlo Ventures, 2024. URL https://menlovc.com/2024-the-state-of-generative-ai-in-the-enterprise/.

Microsoft. Introducing microsoft agent framework: The open-source engine for agentic AI apps. Microsoft Foundry Blog, 2025. URL https://devblogs.microsoft.com/foundry/introducing-microsoft-agent-framework-the-open-source-engine-for-agentic-ai-apps/.

n8n GmbH. n8n: Workflow automation for technical teams. GitHub repository n8n-io/n8n, 2025. URL https://github.com/n8n-io/n8n.

Nakajima, Y. BabyAGI archive: Snapshot of the original BabyAGI repository as of september 2024. GitHub repository yoheinakajima/babyagi_archive, 2024. URL https://github.com/yoheinakajima/babyagi_archive.

Newman, S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

Nygard, M. T. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

OpenAI. Swarm: Educational framework for multi-agent orchestration. GitHub repository openai/swarm, 2024. URL https://github.com/openai/swarm.

OpenAI. OpenAI agents SDK. Software documentation, 2025. URL https://openai.github.io/openai-agents-python/.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., and Sun, M. ChatDev: Communicative agents for software development. In *ACL*, 2024.

Rodriguez, R. Emergent coordination in multi-agent systems via pressure fields and temporal decay. *arXiv preprint arXiv:2601.08129*, 2026.

Sclar, M., Choi, Y., Tsvetkov, Y., and Suhr, A. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In *ICLR*, 2024.

Srivastava, A. et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

Stone, P. and Veloso, M. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8 (3):345–383, 2000. doi: 10.1023/A:1008942012299.

Sutton, R. S. The bitter lesson. *Incomplete Ideas (blog)*, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Tarasova, E., Erofeeva, V., Granichin, O., and Chernikov, K. Decentralized adaptive task allocation for dynamic multi-agent systems. *Scientific Reports*, 15:39226, 2025. doi: 10.1038/s41598-025-21709-9.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.

Wooldridge, M. *An Introduction to MultiAgent Systems*. Wiley, 2nd edition, 2009.

Wooldridge, M. J. and Jennings, N. R. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *ICLR 2024 Workshop on LLM Agents*, 2024.

Yang, C., Yang, X., Wen, L., Fu, D., Mei, J., Wu, R., Cai, P., Shen, Y., Deng, N., Shi, B., Qiao, Y., and Li, H. Learning on the job: An experience-driven self-evolving agent for long-horizon tasks. *arXiv preprint arXiv:2510.08002*, 2025.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of EMNLP*, 2018. doi: 10.18653/v1/D18-1259.

A. Data Curation Methodology

This appendix details the data collection process behind the ecosystem statistics reported in Section 2.

Definition: What Counts as Static Coordination

A system qualifies as "static coordination" if it fixes at least two of: (P1) task decomposition, (P2) agent interaction topology, (P3) verification strategy, *before execution begins*. This includes fixed agent pipelines, predetermined routing, hardcoded role assignments, and visual DAG-based workflow builders.

A.1. Sources and search methodology

We collected data from four sources with date-restricted queries spanning January 2022 through March 2026.

Startup funding. We used public announcements and market reports from TechCrunch, Bloomberg, CB Insights, StartUs Insights, and Menlo Ventures. Vendor databases were used for discovery and qualitative sampling, but exact vendor database counts are not treated as independently reproducible primary evidence. We annotated a sample of 23 high-profile companies by examining publicly visible product descriptions, demos, documentation, and marketing materials. We did not audit internal architectures.

GitHub repositories. We queried the GitHub Search API with date filters, using the following search terms in both English and Chinese:

```
"multi-agent" in:readme language:python
created:2022..2026
"agent workflow" in:readme
language:python
"agentic workflow" in:readme
"ai agents" in:readme language:python
[zhinengti / intelligent agent]
in:readme (Chinese)
[duo-zhinengti / multi-agent] in:readme
(Chinese)
```

Repository counts were aggregated by quarter. We sampled the top 50 repositories per quarter and manually classified each as static coordination (yes/no/partial) based on README descriptions and architecture diagrams. The 190,000+ figure represents total matching repositories, of which our sampled classification found >85% expose static coordination patterns.

Academic papers. We queried arXiv API (categories cs.AI, cs.CL, cs.MA) and Google Scholar with date-restricted queries. The narrow query ("multi-agent" AND "large language model") yields conservative counts (78/468/1,540 papers in 2023/2024/2025). The broader query ("agentic" AND "language model") yields 7,134 in 2025.

Industry reports. We drew aggregate statistics (total corporate AI investment, enterprise generative AI spending, YC batch composition, and venture capital share) from published market reports by CB Insights, PitchBook, Menlo Ventures, GitHub Octoverse, and Stanford HAI's AI Index.

A.2. Annotation framework

For each data point (startup, repository, or paper), we applied the following annotation:

Q1. Static coordination? Fixed decomposition, routing, or roles hardcoded at design time. (Yes / No / Partial / Unclear)

Q2. Which properties fixed? P1 (decomposition), P2 (routing/topology), P3 (verification deferred). (Check all that apply)

Q3. Domain. Coding, customer service, sales, legal, medical, research, data analysis, general-purpose, other.

Q4. Scale indicator. Funding amount (startups), stars and forks (repos), citations (papers).

A.3. Limitations of the data

The $250B figure represents total corporate AI investment in 2024, not investment specifically in static coordination. We report it as ecosystem context, not as a direct measure. Startup and repository counts are ecosystem context rather than claims that every listed company or repository uses static coordination. The "100% static" annotation for our 23-company sample is based on externally visible product behavior and may not reflect internal architectures.

All collection scripts, raw data snapshots with timestamps, and annotation spreadsheets are available in our supplementary materials.

B. Full Experimental Details

This appendix provides complete experimental specifications for reproducing the proof-of-concept experiments in Section 5.

B.1. Task domain: Latin Square constraint satisfaction

Task Configuration

Puzzle sizes: 4×4, 5×5, 6×6
Removal rate: 50% of cells blanked
Generation: Random Latin Square via cyclic construction + row/column permutation
Verification: Deterministic constraint check (all rows contain $1..n$ exactly once AND all columns contain $1..n$ exactly once AND all given cells preserved)
Models: GPT-5.5 (frontier reasoning), GPT-5-mini (compact reasoning) via Azure OpenAI API
Temperature: Default (model-controlled)
Replication: N=100 puzzles $\times$ 3 random seeds (42, 123, 456) for decisive experiment. N=50 for controls.
Statistical methods: Bootstrap 95% CI (10,000 resamples), Fisher's exact test, Cohen's h

Why Latin Square. Latin Square was chosen because it provides three critical properties for studying coordination mechanisms in isolation. First, verification is *exact and deterministic*: a solution is correct if and only if every row and column contains each value exactly once and all given cells are preserved. This eliminates evaluator noise entirely. Second, the task is *decomposable*: you can check row constraints independently from column constraints, enabling granular diagnosis of failure types. Third, the task is *scalable in difficulty*: moving from 4×4 to 6×6 increases the search space from $4! \approx 24$ to $6! \approx 720$ arrangements per row while keeping verification complexity linear.

B.2. Agent architecture

The system uses three agent roles, each backed by the same underlying LLM but prompted with different strategies:

Solver receives the puzzle and generates a complete grid using a row-first strategy. The system prompt instructs: "You solve puzzles. Return ONLY a JSON array of arrays. No explanation." The user prompt provides the puzzle grid with blanks shown as zeros and requests a complete solution.

Fixer receives a broken grid along with specific constraint violations and attempts to repair it. The prompt includes the original puzzle, the incorrect attempt, and up to 8 violation descriptions (e.g., "Col 2: duplicates").

AlternativeSolver uses a column-first strategy ("First ensure all COLUMNS are valid, then verify rows") to provide a qualitatively different approach. This is the key to breaking systematic bias: because it approaches the puzzle from a different direction, it fails on different inputs than the row-first Solver.

B.3. Fault injection mechanism

Fault Injection Protocol

Determinism: Each puzzle is assigned a hash-based seed: `hash(str(puzzle)) + agent_seed_offset`. The fault is deterministic per (puzzle, agent) pair.

Mechanism: On "poisoned" puzzles, the agent's actual API call is intercepted and replaced with a pre-computed faulty grid that satisfies row constraints but violates column constraints. This ensures the fault is systematic (same output on retry) and semantically plausible (passes basic format checks).

Faulty grid generation: For each row, given values are preserved. Remaining values $1..n$ not already given in that row are shuffled (using the deterministic seed) and filled left-to-right. This guarantees valid rows but introduces column violations.

Control: Random faults use the same interception mechanism but generate a *different* faulty grid on each call (new random seed per invocation), confirming that the variable distinguishing systematic from random is *persistence*, not the replacement mechanism.

Why output replacement models systematic bias. We replace agent outputs rather than modifying internal model biases because this provides experimental control while modeling the *observable behavior* of persistent bias: same agent, same input, same wrong output across retries. The control condition (random faults with identical replacement mechanism but different output per call) confirms that the coordination failure is caused by persistence, not by the replacement technique itself.

B.4. Coordination topologies tested

Single Agent. One Solver call with no verification or recovery. Establishes baseline capability.

Static Pipeline (retry). Solver produces output, verification checks constraints. On failure, the same Solver is called again (up to 3 attempts). Models the standard engineering response to detected errors.

Static Pipeline (fixed fallback). Solver produces output, verification checks. On failure, routes to a pre-determined Fixer. The fallback route is hardcoded at design time. Models "Solver fails, route to Fixer" logic.

Runtime Adaptive. Solver produces output, verification checks. On failure, the system selects the next agent by trust weight (initially uniform). Failed agents receive a trust penalty ($\Delta = 0.15$), successful agents receive a trust bonus ($\delta = 0.15$). Violation context is passed to the next agent. Up to 3 rounds.

Exhaustive Chain. Tries all agents in fixed order (Solver, Fixer, AlternativeSolver) until one passes verification. Achieves equivalent accuracy to Runtime Adaptive in 3-agent settings but scales $O(N)$ and requires enumerating

agents at design time.

B.5. Variable poisoning protocol

In the variable-poisoning experiment, the bias assignment varies per puzzle. Specifically, for each puzzle: with probability 0.5, both Solver AND Fixer are biased (only AlternativeSolver produces correct output), and with probability 0.5, both Solver AND AlternativeSolver are biased (only Fixer produces correct output). The biased agents return deterministic faulty grids (different per agent, using `agent_seed_offset` of 0 vs. 1000). This ensures no single static fallback route covers all puzzles.

B.6. Verifier sensitivity simulation

The verifier sensitivity experiment does not require API calls. It is a Monte Carlo simulation over the coordination loop with parameterized detection probability:

Parameters: $N_{\text{puzzles}} = 100$, poison rate $= 0.5$, fixer reliability $= 1.0$, verifier specificity $= 1.0$ (no false positives), verifier sensitivity $s \in \{1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.3\}$.

Logic: For each poisoned puzzle, the verifier detects the fault with probability s. If detected, the system reroutes to Fixer (which succeeds with probability = fixer reliability). If not detected, the faulty output is accepted. For clean puzzles, the verifier correctly passes with probability = specificity.

Replication: 1000 independent trials per sensitivity level. Results averaged across trials.

Table 7. Verifier sensitivity sweep (Monte Carlo, 1000 trials per setting). Graceful degradation under imperfect verification. Even at s=0.5, improvement over retry exceeds $8\times$. The crossover where runtime equals retry occurs only below s=0.06.

Verifier sens. s	Recovery	Impr. over retry	Mean att.
1.0 (perfect)	100%	$16\times$	2.0
0.9	97%	$15\times$	2.1
0.7	88%	$12\times$	2.4
0.5	72%	$8\times$	2.9
0.3	48%	$4\times$	3.6

B.7. Trust convergence simulation

Setup: 5 agents, each biased on a different 20% subset of tasks. On any given task, exactly 1 agent is biased (produces wrong output detected by verification).

Trust dynamics: Initial trust $= 0.5$ for all agents. On success: trust $+= 0.15$ (capped at 1.0). On failure: trust $-= 0.40$ (floored at 0.0). Agents are tried in descending trust order.

Metrics: Calls per task (convergence = fewer calls over time), first-try success rate (should increase as trust learns), and re-adaptation speed after bias removal.

Replication: 200 tasks per run $\times$ 20 random seeds. Results averaged with standard deviation bands.

B.8. HotpotQA experiment

The HotpotQA experiment tests capability absence (a fundamentally different failure type from systematic semantic bias). Five heterogeneous agents are deployed: Searcher (primary retrieval), BackupSearcher (alternative retrieval via different search strategy), Reader (passage comprehension), Calculator (numerical reasoning), and Synthesizer (multi-hop answer composition). Under the node-failure condition, the Searcher agent is made unavailable (simulating a capability that has been removed or broken after a model update). The static pipeline has no alternative search path and drops to 0% on questions requiring retrieval. The runtime network discovers BackupSearcher via capability advertisement ("I can search and retrieve relevant passages") and routes retrieval tasks to it, recovering to 77%.

B.9. Compute and cost

All experiments use API calls to Azure OpenAI (GPT-5.5 and GPT-5-mini deployments). No local GPU compute is required. Total API calls across all conditions (including replications, controls, and pilot studies): approximately 2,500. Estimated cost: under $50 USD at standard API pricing. Wall-clock time for the full experimental suite: approximately 3 hours (dominated by sequential API calls with rate limiting). The verifier sensitivity and trust convergence simulations require zero API calls (pure computation, completes in under 1 second).