Notified Serializability: A Consistency Model for Concurrent LLM Agents

Hongtao Lyu*
Shanghai Jiao Tong University
China
hongtaolyu@sjtu.edu.cn

Dingyan Zhang*
Shanghai Jiao Tong University
China
healthcliff-ding@sjtu.edu.cn

Mingyu Wu
Shanghai Jiao Tong University
China
mingyuwu@sjtu.edu.cn

Xingda Wei
Shanghai Jiao Tong University
China
wxdwfc@sjtu.edu.cn

Haibo Chen
Shanghai Jiao Tong University
China
haibochen@sjtu.edu.cn

Abstract

Multi-agent LLM systems increasingly run agents in parallel against shared repositories, Kubernetes clusters, and documents. Once agents mutate common state, their interleaved reads and writes reproduce isolation anomalies. Classical remedies fit poorly: runs span minutes of inference, and agents gather broad context for task success; this read amplification turns irrelevant overlaps into long 2PL waits or OCC aborts that discard complete runs. Much state is live and admits neither forks nor buffered writes. We argue that concurrency control should be advisory: the runtime reports a conflict, and the affected agent decides whether it invalidates the plan and repairs only dependent operations. We formalize this guarantee as *notified serializability* and present MTPO, which fixes a serialization order at launch, serves order-filtered reads, applies writes in place, and reconciles misordered writes through registered inverses; at quiescence the run is serializable in the fixed order. On ten contended two-agent workloads, a prototype achieves a 1.4× speedup and 1.15× token cost with correctness within 5% of serial, while 2PL and OCC surrender nearly all concurrency gains.

1 Introduction

LLM agents modify repositories, databases, and clusters via tools. To cut latency, systems run them concurrently: Claude Code ships parallel teams [1], Codex and Cursor run background agents [2, 22], and Kimi coordinates up to a hundred agents [21]. Once agents share mutable state, classical concurrency failures follow; an audit of over 200 traces attributes more than a third of failures to inter-agent misalignment, including agents acting on the same resource unaware of each other [7].

A measured run makes the failure concrete. Two agents share a live Kubernetes cluster. In an AIOpsLab remediation task [9], agent A must repair every deployment left on a wrong image. Agent B independently reads one service's current image and creates a matching canary. A lists the cluster before the canary exists, so its sweep misses it; B reads before A's repair, so the canary mirrors the bad image. Both report success. Yet either serial order would leave the canary repaired: the observed outcome is nonserializable and stays silent until the canary takes traffic.

Serializability rules out such outcomes [3], but its classical enforcers, two-phase locking (2PL) [12] and optimistic concurrency control (OCC) [18], fit agent workloads poorly for three related reasons (§2). **(i) Long execution.** Agent runs spend most of their minutes in inference: 2PL stalls parallel work, while an OCC abort discards the whole run. **(ii) Read amplification.** Unlike transactions with explicit access paths, agents gather broad context for task success, often scanning repositories or namespaces. The inflated lock and validation sets turn irrelevant overlaps into waits or aborts. **(iii) Live state.** A cluster, production database, or external API generally admits neither a fork nor buffered writes; effects occur immediately, OCC has no commit-time install, and abort may not erase them.

The LLM supplies the missing semantic judgment. Given a conflict, it can decide whether a changed value invalidates a plan premise; most broad-read overlaps do not (e.g., a peer appending a log line to a scanned file disturbs nothing the agent relies on). If one does, the agent rewrites only dependent operations. Registered inverses provide saga-style compensation for an out-of-order write [14]; the affected agent then decides whether and how to issue a corrected write. Coordination can thus be *advisory*: the runtime reports, and the affected agent repairs.

We formalize *notified serializability* (§3), a correctness criterion for concurrent agents that incorporates conflict notification and local saga compensation. We then present MTPO (Monotonic Trajectory Pre-Order, §4), which fixes the serialization order and maintains its live state through three repair rules; at quiescence, every object equals what that serial execution would leave. Its rules separate mechanical filtering and undo from LLM revision of premise-dependent actions. §5 reports initial validation on ten contended workloads.

*Both authors contributed equally to this work.

2 Why Classical Mechanisms Fit Poorly

2.1 The Agent Execution Model

An LLM agent keeps a private, append-only context: each inference appends tool calls, results, and model output [32, 33]. A read remains a premise for later inference after its object changes. Because write parameters may derive from earlier reads in this context, we conservatively treat every write as *read-modify-write* (RMW), even when the tool call appears to be a direct assignment. We call these context-derived perceptions the agent's *view*, its basis for later action.

2.2 Two Gaps

Classical concurrency control assumes short transactions and cheap retries. Agent runs spend seconds to minutes in inference: 2PL holds locks throughout, while one OCC validation failure discards inference, tool calls, and artifacts. Unlike a programmed transaction's explicit access path, an agent may grep a repository or list a namespace before acting. This read amplification expands lock and validation sets, causing work to wait or abort on semantically irrelevant overlaps; §5 measures the effect.

Classical mechanisms also separate uncommitted writes from published state: they stage writes for commit (OCC, snapshot isolation, and Spanner [11]) or roll them back on abort (2PL). A live Kubernetes cluster, production database, or third-party API provides one copy; writes take effect immediately, and external effects may resist rollback.

These gaps yield two requirements. **R1:** coordination combines optimistic execution with localized recovery: agents avoid long-held locks, and recovery repairs only dependent work instead of aborting the task. **R2:** writes take effect in place because live targets provide no general fork, buffer, or rollback.

2.3 What Deployed Systems Do

Deployed systems serialize agents, partition writes, or fork state. Serialization forfeits concurrency [16, 26, 31]; static partitioning needs write sets before plans reveal them and misses the canary anomaly despite disjoint writes [1]. Coding agents enjoy a favorable special case: Git worktrees cheaply fork repository state, and tests check some post-merge invariants [2, 10]. This does not generalize: many targets cannot be forked, many consistency conditions lack a simple test oracle, and copies still admit read-committed-like anomalies [3].

3 Notified Serializability

3.1 Runs, Objects, and Orders

A *run* is a finite total order of tool invocations and returns, physical writes, repairs, notifications, and commits. Its external projection records writes reaching the target and agents' final reports; call metadata and repair bookkeeping are internal. Shared state partitions into *objects*. Each call τ declares its read/write *footprint* $R(\tau), W(\tau)$, and each reversible write carries an inverse. Because effects can exceed call arguments, footprints include indirect effects and let the runtime confine calls (§4.4).

The model adds two framework actions. *Notify* appends a message reporting another agent's applied write and resulting value; the receiver treats it as fact and decides whether and how to adapt. This short append is fast and token-light relative to rerunning an agent. *Saga undo* reverses an applied live-state write through its registered inverse. Undo is necessary to reconcile out-of-order write-write (ww) conflicts on a single live copy. Prior work has proposed saga-style compensation for multi-agent LLM planning [8].

The system has two orders. We assume a global *tool order* $\prec_t$ because tool-internal concurrency safety lies outside the agent framework: calls may overlap if their combined outcome equals execution in that order. The *serialization order* $\prec_\sigma$ is instead the agent-level serial reference the run should match. It is existential in the correctness criterion; §4.1 explains why MTPO fixes one before execution.

3.2 Agent Assumptions

The model assumes: **A1 (individual success):** run alone from a state satisfying its premises, the agent completes its task, adapting transaction theory's individually correct programs to a stochastic executor [4]. **A2 (well-formedness):** every shared-state access uses a registered tool within its declared footprint, and each delivered notification is consumed before the next action. This is the agent analogue of a *well-formed transaction*, locking theory's term for following its required access discipline [12]. **A3 (self-healing):** a notified agent recognizes whether the change invalidates a premise and revises exactly dependent work, possibly none; for an undone RMW, it also decides whether to reissue it and recomputes its parameters. A3 is a new capability claim about the LLM and is probabilistic in practice: §5 observes a 5% misjudgment rate on a budget model.

3.3 The Correctness Notion

Conflict serializability commutes adjacent nonconflicting operations to obtain a serial schedule. Agent runs add notifications, saga undo, and agent-issued replacements, which ordinary commutation cannot represent. *Notified equivalence* ($\cong_N$) therefore adds two cancellation laws. The read-side law treats a stale read followed by notification of the value exposed by the $\prec_\sigma$-serial execution and an A3-correct revision as equivalent to returning that value initially. The write-side law treats a rank-reversed RMW, framework undo, and notification of the refreshed serial value followed by A3-correct redo or omission as equivalent to executing only the corrected operation at its σ position. Undo is mechanical; both semantic revisions depend on A3 and isolate the LLM's uncertainty.

Applying these laws removes repair events and yields an ordinary interleaving *I. Serialization equivalence* ($\cong_S$) then

applies textbook commutation to decide whether I is conflict-equivalent to a serial schedule [4]. A run ρ is **notified-serializable** when, for some $\prec_\sigma$, $\rho \cong_N I \cong_S S_\sigma$, where S_σ is the corresponding serial schedule. The naive canary run lacks a correction, so no read-side cancellation yields either serial order. With A before B, notifying B of A's repair lets it reissue `set_image` from the corrected value; the stale premise and repair cancel, yielding the reference interleaving.

4 The MTPO Protocol

4.1 Fixing the Agent Order

Notify must be one-way: if agents may require each other to adapt, two rw antidependencies can form a nonserializable cycle and livelock. For $x = y = 1$, run $x \leftarrow y/2 \parallel y \leftarrow x/2$: each correction invalidates the other's read and triggers another, whereas either serial order ends after two writes. Classical protocols choose direction during execution: 2PL through lock waits and victim selection, OCC through validation or commit order and abort. Their blocking and whole-run retry violate R1, while private buffering and general rollback do not fit R2's single live copy. MTPO therefore fixes the agent order before execution.

At launch, MTPO assigns ranks σ_j and records their *pre-order* $\prec_\sigma$, following deterministic pre-ordering [29]. Repair moves only upward: notifications flow from lower to higher rank; for ww, the framework saga-undoes higher-ranked writes and notifies their agents to reconsider them. Dependencies thus form a σ-monotone DAG of depth at most $n-1$; in the example, the lower rank halves once and the higher consumes one correction to reach the serial outcome.

Fixed direction gives progress by rank induction. For finite tasks under A1 and A3, the lowest-ranked agent terminates without repair, and each later agent receives finitely many repairs from terminating predecessors. No predecessor waits for a successor, so idealized makespan excluding repair is no worse than serial execution; notification, inference, undo, and redo can exceed this bound in practice.

Pre-ordering has one scheduling cost: a short, late-ranked task may finish locally yet wait for predecessors to stabilize and absorb their repairs. This affects latency, not correctness when A1 holds for the chosen order; complexity-aware ranking is future work.

4.2 Repair by Conflict Type

MTPO has three repair rules with one serial reference. For object o, the *write trajectory* $T(o)$ lists uncancelled writes in σ order. Its materialization $M(o, \sigma_j)$ applies the prefix through σ_j to the initial state and is the $\prec_\sigma$-serial value exposed to j. Notification $v_{i\to j}(o)$ delivers that refreshed value into j's view.

Reads pull from the trajectory (wr). Before returning a read, the runtime produces $M(o, \sigma_j)$, screening out higher-ranked writes through a three-level fallback. If the object retains versions, it reads the corresponding one. For hard-to-version live targets such as a cluster or database, it instead caches historical results from common analytical read tools. If neither form exists, it saga-undoes higher-ranked writes and reads the restored rank-correct state.

Table 1. MTPO repair rules.

Dep.	Backward event	Repair
wr	Too-new read	Return $M(o, \sigma_j)$
rw	Read made stale	Notify reader; revise dependents
ww	Rank-reversed writes	Undo high; apply low; notify origin

Notifications push to readers (rw). The runtime tracks reads. When a lower-ranked write makes a higher-ranked agent's read stale, it delivers $v_{i\to j}(o)$ with refreshed $M(o, \sigma_j)$ before the receiver's next inference. The LLM keeps, rewrites, or retracts only dependent work (A3), reopening an agent that had finished locally if needed. Reverse notifications are forbidden by §4.1.

Writes apply speculatively (ww). A write arrives at its $\prec_t$ position, enters $T(o)$ at rank σ_j, and is *late* if a higher-ranked write already took effect. The framework automatically saga-undoes such writes in descending rank, cancels their old entries, then applies the lower-ranked write immediately. Because all writes are RMW, it notifies their originators with the refreshed rank-correct value; each decides whether to issue a replacement and recomputes its parameters (A3). A noninvertible tool waits until every lower rank commits, so it cannot arrive late.

4.3 Correctness

An agent is *quiescent* when it has no further write and has consumed every delivered notification; *GlobalQuiet* holds when all agents are quiescent and none is in flight. MTPO maintains one invariant: *at GlobalQuiet, every live object equals its trajectory's materialization.*

Proposition (notified serializability). *Assume A1–A3 and the three rules. At GlobalQuiet, the run is notified-serializable in $\prec_\sigma$, and every object's final state is what the $\prec_\sigma$-serial execution leaves.*

Proof sketch. Apply the cancellation laws to construct I, with each operation at its agent's rank and every read returning its GlobalQuiet $M(o, \sigma_j)$. Table 1 supplies every reduction required for $\rho \cong_N I$: wr screens out higher-ranked writes, rw invokes the read-side law through A3 revision, and ww invokes the write-side law through undo and A3 redo or omission. Every remaining conflict edge points forward in σ, so its precedence graph is acyclic and $I \cong_S S_\sigma$ [4]; the invariant gives the same final state. □

4.4 Realization via Tool Synthesis

Declared tools. A privileged, domain-independent *ToolSmith* agent realizes MTPO online. A worker submits a declarative capability request; ToolSmith synthesizes a concrete registered tool with its implementation and declared

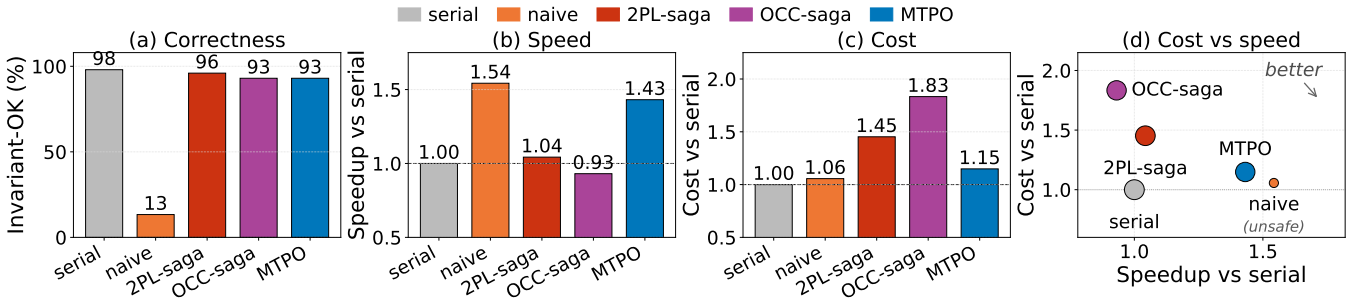

Figure 1. End-to-end results over ten contended two-agent workloads; speed and token cost are normalized to serial.

read/write footprints. Admitting calls only through these tools lets the runtime interpose and track dependencies without application-specific framework code.

History and undo. ToolSmith decides for each object or analytical read whether a historical version or result can be cached, then generates the cache-maintenance logic. To automate compensation, it emits three phases for each write: `prepare` saves the state needed for recovery, `exec` performs the call, and `reverse` restores it. For example, drop-database dumps before execution and restores that dump on reverse. ToolSmith writes all three phases so the runtime can invoke reverse automatically; a tool without one waits for lower ranks to commit (§4.2).

5 Initial Validation

Setup. We built ten contended two-agent cells from WorkBench (simulated office automation) [28] and AIOpsLab (incident remediation on a live Kubernetes cluster) [9]. Neither suite races by design, so each cell pairs one suite task with a hand-constructed second agent chosen to exhibit a textbook anomaly (dirty read, phantom, lost update, read skew, write skew). Hand-written invariants give the oracle: a run passes iff its outcome is equivalent to one of the two serial orders. Workers run `deepseek-v4-flash`; each cell runs ten trials per protocol, all on one middleware: **serial**, **naive** (no coordination), **2PL** (declared footprints as locks, deadlock detection, victim undo and restart), **OCC** (eager validation, abort and restart), and **MTPO**.

Results (Figure 1). Naive reaches ≈1.5× speedup but passes only 13% of trials. 2PL and OCC recover correctness but nearly no concurrency; OCC also costs 1.83× the tokens. MTPO passes every cell at a 1.4× speedup and 1.15× token cost, with correctness within 5% of serial. In the five residual failures, the notified agent misjudged relevance to its plan: the guarantee remains conditional on A3.

Case study (Figure 2). On the canary cell from §1, MTPO matches naive's wall time (33 versus 32 s) while preserving the serial outcome: A's repair triggers one notification, and B changes only the dependent canary image. 2PL takes 53 s after deadlock and lock waits; OCC takes 64 s after aborting and redoing B.

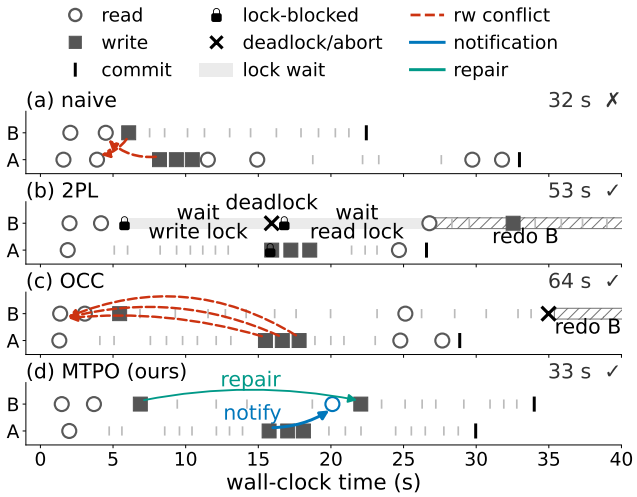

Figure 2. Execution timelines for the canary workload.

6 Related Work

Agent systems validate read sets [17, 19], port classical protocols [6, 23, 25, 34], or wrap tools transactionally [20]. They impose blocking, abort, or transaction boundaries and give no repair-based agent correctness model. SagaLLM proposes saga-style compensation for multi-agent planning, while GoEX studies undoable tool execution [8, 24]; MTPO decides when to compensate.

Federated databases, multidatabase middleware, and TP monitors coordinate fixed transactions across heterogeneous systems [5, 15, 27]. They rely on locks, validation, commit, or compensation. MTPO instead lets an executor revise dependent actions after notification. Deterministic databases pre-order transactions [29]; MTPO uses that order to direct repair on live state. Semantic concurrency control used static annotations [13, 30], whereas an LLM judges dependencies at runtime and its judgment must be checked.

References

[1] Anthropic. 2024. Claude Code: Sub-agents and Agent Teams. https://docs.anthropic.com/en/docs/claude-code/sub-agents.

[2] Anysphere. 2025. Cursor Docs: Worktrees—Run Parallel Agents in Isolated Copies of the Codebase. https://cursor.com/docs/configuration/worktrees.

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM Press, 1–10. doi:10.1145/223784.223785

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[5] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. 1992. Overview of Multidatabase Transaction Management. *The VLDB Journal* 1, 2 (1992), 181–239. doi:10.1007/BF01231700

[6] Umut Çalıkyılmaz, Nitin Nayak, Jinghua Groppe, and Sven Groppe. 2025. OptiMA: A Transaction-Based Framework with Throughput Optimization for Very Complex Multi-Agent Systems. arXiv:2511.03761 [cs.MA]

[7] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail?. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

[8] Edward Y. Chang and Longling Geng. 2025. SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4874–4886.

[9] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds. arXiv:2501.06706 [cs.SE]

[10] Cognition. 2024. Introducing Devin, the First AI Software Engineer. https://cognition.ai/blog/introducing-devin.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems* 31, 3 (2013), 8:1–8:22.

[12] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[13] Hector Garcia-Molina. 1983. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems* 8, 2 (1983), 186–213.

[14] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. 249–259.

[15] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA.

[16] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In *ICLR*.

[17] Sajjad Khan. 2026. S-Bus: Automatic Read-Set Reconstruction for Multi-Agent LLM State Coordination. arXiv:2605.17076 [cs.LG]

[18] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981).

[19] Mengyang Liu, Taozhi Chen, Zhenhua Xu, Xue Jiang, and Yihong Dong. 2026. Multi-agent Collaboration with State Management. arXiv:2605.20563 [cs.MA]

[20] Bardia Mohammadi, Nearchos Potamitis, Lars Klein, Akhil Arora, and Laurent Bindschaedler. 2026. Atomix: Timely, Transactional Tool Use for Reliable Agentic Workflows. arXiv:2602.14849 [cs.LG]

[21] Moonshot AI. 2026. Kimi Introduces Agent Swarm: Let 100 AI Agents Work for You. https://www.kimi.com/blog/agent-swarm.

[22] OpenAI. 2025. Introducing Codex: A Cloud-Based Software Engineering Agent That Can Work on Many Tasks in Parallel. https://openai.com/index/introducing-codex/.

[23] Vladyslav Parakhin. 2026. Token Coherence: Adapting MESI Cache Protocols to Minimize Synchronization Overhead in Multi-Agent LLM Systems. arXiv:2603.15183 [cs.DC]

[24] Shishir G. Patil, Tianjun Zhang, Vivian Fang, Noppapon C., Roy Huang, Aaron Hao, Martin Casado, Joseph E. Gonzalez, Raluca Ada Popa, and Ion Stoica. 2024. GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications. arXiv:2404.06921 [cs.SE]

[25] Sergey Pugachev. 2025. CodeCRDT: Observation-Driven Coordination for Multi-Agent LLM Code Generation. arXiv:2510.18893 [cs.DC]

[26] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *ACL*.

[27] Amit P. Sheth and James A. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *Comput. Surveys* 22, 3 (1990), 183–236. doi:10.1145/96602.96604

[28] Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. 2024. WorkBench: A Benchmark Dataset for Agents in a Realistic Workplace Setting. arXiv:2405.00823 [cs.AI]

[29] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[30] William E. Weihl. 1988. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.* 37, 12 (1988), 1488–1505.

[31] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI]

[32] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *NeurIPS*.

[33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[34] Weixing Zhou, Zhiyou Wang, Zeshun Peng, Hetian Chen, Yanfeng Zhang, and Ge Yu. 2026. ATCC: Adaptive Concurrency Control for Unforeseen Agentic Transactions. arXiv:2603.13906 [cs.DB]