

Robust DAG-Scoped Git Worktree Orchestration: State-of-the-Art Techniques

Overview: Ensuring robustness and reliability in a DAG-scoped Git worktree orchestration system requires drawing on several research areas. We review four key areas – DAG scheduling theory, version-control merge models, distributed agent coordination, and fault-tolerance frameworks – highlighting state-of-the-art methods and formal models. Each area's insights are then related to Git-based DAG orchestration, with recommendations on integration.

1. DAG Modeling and Scheduling Theory

Structured DAG Models (TTSP Graphs): Directed acyclic graphs that are *two-terminal series-parallel* (TTSP) have a single source and sink and are composed only by series or parallel combinations ¹ ². Such graphs belong to the class of *series-parallel partial orders*, which are **N-free** posets with order dimension ≤ 2 ³. This structured nature simplifies analysis – e.g. series-parallel tasks appear in job-shop scheduling and dataflow systems ⁴. In particular, scheduling algorithms can exploit the decomposition tree of a TTSP DAG to achieve more deterministic and efficient schedules. For example, *Hans Stadtherr (2000)* showed that if precedence constraints form a series-parallel graph, an **optimal two-processor schedule** can be computed in *logarithmic time* with only linear operations ⁵. (By contrast, optimal scheduling for arbitrary DAGs is NP-hard, and even the two-processor case is unsolved in general ⁶ ⁷.) Such results indicate the value of structuring an orchestration's dependency graph as series-parallel, enabling polynomial-time scheduling decisions.

Deterministic Scheduling & Feasibility: A core challenge in parallel DAG execution is deciding *which ready task to run when*, in order to meet some optimality or feasibility criteria. *List scheduling* heuristics assign each task a priority (e.g. based on critical-path length or topological level) and then iteratively schedule the highest-priority ready task on an available processor ⁸. Common priorities include: (a) **Top-level / bottom-level** (t-level/b-level) which estimate how critical a task is by longest path remaining ⁹ ¹⁰, and (b) **Critical Path First** strategies that explicitly prioritize tasks on the longest path through the DAG. These heuristics aim to minimize the total makespan (finish time) and avoid idle resources. They come with informal guarantees – e.g. Graham's classic result that simple list scheduling yields a finish time within $(2 - \frac{1}{m})$ of optimal on m processors (for independent tasks). Deterministic scheduling policies (always breaking ties in a fixed way) help ensure *reproducibility* – i.e. the DAG executes in a predictable order run-to-run. Additionally, formal methods like *timed automata* or *model-checking* can verify **deadlock absence** and **feasibility** in task schedules (though a well-formed DAG by definition has no circular waits). In TTSP DAGs, any schedule respecting the partial order will avoid deadlock; further, specialized algorithms exist for *deterministic parallel scheduling* in such graphs that guarantee all tasks complete if each is allotted sufficient resources ⁵ ¹¹.

Priority Scheduling and Critical-Path Heuristics: Modern DAG schedulers often use variants of **critical-path heuristics**. One example is the *HEFT (Heterogeneous Earliest Finish Time)* algorithm, which computes an “upward rank” (essentially critical-path length from each task to the sink) and schedules tasks in descending order of this rank on the fastest available resource. Such priority-driven methods have been very effective in practice ¹⁰. They do not guarantee absolute optimality (except in special cases), but they do ensure *schedule feasibility*: if a schedule exists that meets all dependencies, a well-designed heuristic will find a valid schedule (though not necessarily minimal length). Additionally, **work-**

efficient parallel execution models provide theoretical guarantees. For instance, *work-stealing schedulers* (as used in Cilk and similar frameworks) achieve expected execution time $T_{\text{exec}} = T_1/P + O(T_{\infty})$ on P processors¹². Here T_1 is the total work (serial time) and T_{∞} is the critical path length; thus the scheduler runs within constant-factor of the ideal span and perfectly balances load¹². This result, proved by Blumofe et al. (1999), formally guarantees near-optimal makespan and no deadlocks for *fully strict* (well-structured) DAGs under a randomized work-stealing policy. Such guarantees give confidence that parallel execution can be both **work-efficient** (no blow-up in total work done) and **time-efficient**, which is crucial for an orchestrator coordinating many tasks.

Recommendations for Integration: In a Git worktree orchestrator, representing the pipeline as a TTSP DAG can simplify reasoning. The DAG's **decomposition tree** could be used to derive deterministic schedules (e.g. always expand parallel branches in a fixed order). Employing a *critical-path priority* scheduler ensures that time-critical tasks (like merging a long chain of commits) run early, reducing overall completion time. Meanwhile, adopting a **work-stealing** execution engine under the hood would provide resilience against load imbalance – if one agent becomes free, it can “steal” a pending task from a busier agent, with provable near-optimal efficiency¹². Finally, modeling the scheduling and dependencies in a formal way (e.g. using TLA+ or Petri nets) could prove absence of deadlocks and confirm that every task will eventually run to completion if agents do not fail.

2. Git Branching and Merge Conflict Resolution

Formal Models of Version Control: Git's branching and merging semantics can be described in terms of **graph theory and patch algebra**. Each commit is a node in a repository DAG, and a *non-fast-forward merge* creates a new commit with two parent pointers (the merged branches). The merge operation can be viewed as computing a **least common successor** (LCS) of two commits given their common ancestor, but only if changes do not conflict. Research in *patch theory* formalizes this: for example, *Darcs* (a DVCS like Git) is built on an abstract *theory of patches* that defines how patches commute and merge in a principled way¹³. In Darcs' patch algebra, merging is the act of finding a sequence of patches that represents the combination of two histories, using **inverse patches** to undo conflicts if needed¹⁴¹³. This formalization (using structures like *inverse semigroups* or even category theory) provides *provably correct merging algorithms* and insights into conflict handling, making the system more robust¹³. Adopting a patch-theoretic view in a Git worktree orchestrator could mean treating each task's changes as algebraic operations and ensuring that applying them in different orders yields equivalent outcomes when possible.

Conflict Detection and Semantic Merge: Standard Git uses line-based **three-way merge**, which detects conflicts when the same lines are edited divergently or when one side deletes a file the other modified. However, many conflicts are *semantic* – code merges that succeed syntactically may still break behavior. State-of-the-art methods aim to catch or resolve such issues. For instance, **semantic merge tools** incorporate program analysis or tests: the recent *SAM (SemAntic Merge)* approach generates unit tests for each change and runs them on the merged version to detect unintended behavior interactions¹⁵. This allows detection of *semantic conflicts* that don't show up as text conflicts (e.g. two changes compile but produce a runtime error together)¹⁵. Other research uses static analysis or symbolic execution to find semantic conflicts by checking if the merged code violates certain specifications or tests¹⁶¹⁷. On the resolution side, **structured merge** (parsing code into ASTs and merging the trees) can automatically handle many non-textual reorders or insertions. More formally, merges can be treated as computing the **least upper bound** in a lattice of program states – when that lattice has additional structure (like a knowledge of program variables or operations), more conflicts can be resolved automatically. For example, *operation-based* version control and **CRDT** approaches (see below) can merge certain changes without conflict by design (commutative operations).

AI/ML for Merge Automation: A cutting-edge avenue is using machine learning to assist or automate merges. Large datasets of open-source merges have enabled training models to predict resolutions. For example, **MergeBERT** and similar models treat the merge conflict as an input (with left, base, right versions) and output a resolved file, essentially learning from historical resolutions. AI-based developer tools (JetBrains' AI Assistant, CodeGPT, etc.) now offer *contextual merge suggestions*, analyzing code intent and proposing how to reconcile differences ¹⁸ ¹⁹. These tools leverage language models that have "seen" many code edits, thereby providing likely resolutions or at least pinpointing the contentious parts. While not *formally proven*, such AI assistants can significantly reduce manual effort by handling routine merges or by explaining conflicts. In practice, ML-based merge tools might be integrated to attempt an automatic resolution first; if the model is confident (e.g. passes compilation and tests), it can apply the fix, otherwise fall back to a human or a safer strategy. It's worth noting that any AI-driven merge should be combined with verification steps (compile, test suite) to ensure the suggested resolution is correct.

Proven Merge Strategies and Fallbacks: Regardless of AI, robust systems incorporate *safe fallback mechanisms*. **Semantic conflict avoidance** can be done by design – e.g. requiring that all tasks in the DAG frequently merge/rebase from a mainline to minimize divergent changes, or using *feature flags* to decouple deployment. When a merge conflict is detected, one proven strategy is **automated bisection/rollback**: attempt the merge, run tests; if it fails or tests fail, automatically revert the merge commit and mark the task as needing manual intervention. Some systems use *merger bots* that try a variety of strategies: e.g. first try a recursive merge, if conflict then try a *content-aware merge tool* (that might understand syntax), and if still failing, as a last resort, produce a diff with conflict markers for a human. Another approach is **two-phase merges**: attempt to merge in a dry-run, if conflicts exist, notify or apply heuristics to resolve (such as keeping both changes in separate sections of code). The key is to incorporate *idempotency*: if a merge fails and is rolled back, the system should be able to retry (perhaps after other tasks complete) without side effects. Some research prototypes even treat merging as a *search problem*, exploring possible merges or orderings of patches to find a conflict-free sequence ¹⁴ ¹³.

Recommendations for Integration: For a DAG-scoped Git orchestrator, we recommend incorporating **formal version-control logic** to manage branches. Each task could operate on an isolated Git *worktree* (a separate working copy tied to a new branch), so its changes are isolated. The orchestrator can use a **patch algebra** internally: treat each task's diff as a patch and define commutativity rules – for example, edits to disjoint files commute and can be merged in any order automatically. This could allow parallel tasks to apply without conflict. In case of conflicts, the system should leverage a *semantic merge check*: e.g. automatically run a relevant test suite on the merged result (if available) to detect subtle issues ¹⁵. We also advise integrating an **automated merge resolver**: a pipeline where a lightweight ML model or rule-based engine first attempts to resolve trivial conflicts (like whitespace, imports, or function reordering). Proven techniques like *ours* (3-way merge) remain the baseline, but augmented with semantic awareness. If the orchestrator detects a merge conflict it cannot resolve, it should support a *rollback* or *retry* mechanism: for example, it might postpone that merge and allow other independent tasks to proceed, then retry merging later (perhaps the conflict is resolved by another task's completion). This ties into agent coordination discussed next. Finally, design all merge-related tasks to be **idempotent** – e.g. merging the same branch twice should have no additional effect – and use "scratch" branches for merges so that if a merge fails, the main branches remain intact. Adopting these strategies will greatly enhance the reliability of an automated Git workflow.

3. Agent Coordination and Retry Semantics

Coordinating Parallel Agents: In a distributed orchestration, multiple agents (workers) execute tasks concurrently. Classic coordination models from distributed systems can be applied to ensure these

agents work in unison without stepping on each other's toes. One elegant paradigm is *observation-driven coordination* via a shared dataspace (in contrast to direct message passing) ²⁰ ²¹. This is inspired by Linda tuple spaces and blackboard systems where agents communicate by reading/writing to a common store. Modern implementations use **Conflict-Free Replicated Data Types (CRDTs)** as the shared state substrate. CRDTs provide *strong eventual consistency* – all agents can update a replicated state (like a set of completed tasks or a queue of pending tasks) independently, and despite out-of-order operations, all replicas converge deterministically to the same state ²⁰ ²¹. In other words, CRDT updates are *commutative* and no central lock or merge resolution is needed; the algorithm ensures **deterministic state convergence** and **monotonic progress** (no state changes are lost or undone) ²² ²¹. For an agent orchestration, this means we can have, for example, a CRDT-based work queue: agents pull tasks from the queue and mark them done in a way that even if two agents briefly take the same task, the CRDT state will reconcile (perhaps by one agent's completion overwriting the duplicate, etc.) with no conflict. A recent study on multi-agent LLM systems demonstrated this approach, showing that CRDT-coordinated agents had *zero merge conflicts* at the data structure level and guaranteed that all agents eventually saw a consistent world-state ²³ ²⁴. Applying these principles yields **lock-free coordination** – no need for traditional locking or leader election to assign tasks – which improves fault tolerance (since any agent can pick up work) and avoids deadlocks.

Retry Models and Backoff Policies: When tasks or agents contend for resources, or when tasks can fail transiently, a robust retry strategy is critical. *Exponential backoff* is the de facto standard for managing retries in distributed systems ²⁵. Instead of retrying immediately and repeatedly (which can overload the system), an agent waits an increasing amount of time after each failure before retrying. For example, after the first failure wait 1s, after the second wait 2s, then 4s, etc. This **exponential delay** drastically reduces collision probability (the chance that two agents keep failing and retrying in sync). In practice, backoffs are **capped and bounded** – one sets a maximum wait (to avoid unbounded delays) and a maximum number of retries ²⁵ ²⁶. Amazon describes using *capped exponential backoff with jitter*, meaning each client picks a random delay in the exponential range to further desynchronize retries ²⁷ ²⁸. Jitter prevents a thundering herd when many agents retry at once by adding a random perturbation ²⁹. In formal terms, if each retry time is randomized, one can prove with high probability that at least one agent will succeed after a bounded number of attempts (assuming an underlying issue gets resolved). **Deterministic retry policies** (without randomness) are sometimes used in controlled environments – e.g. a fixed schedule of retries – but they risk synchronized collisions unless the system ensures only one agent is acting at a time (which reduces parallelism). A compromise is to assign each agent a distinct retry offset or priority so that one agent effectively “wins” if a conflict occurs while others back off deterministically.

Agent Coordination Techniques: Beyond shared-state approaches, distributed algorithms like leader election, quorum consensus, or token-ring scheduling can coordinate agents. However, these introduce more complexity than needed for DAG orchestration (which doesn't typically need consensus beyond agreeing on task completion). A simpler approach is **distributed queues** or pub-sub: tasks are pushed to a queue (Kafka, etc.) and agents consume them, ensuring no two agents do the same work. This is a form of coordination via *work distribution*. Another advanced technique is using *vector clocks* or *version vectors* to track the state of each agent's knowledge of task completions, which can help in reconciling state (though CRDT sets often incorporate vector-clock-like mechanisms under the hood). Importantly, if multiple agents might attempt operations on the same Git repo (like pushing to a branch), coordination is needed to avoid race conditions. One could implement a **distributed lock** (e.g. using etcd or Zookeeper) such that at most one agent can perform a push at a time. But a lock can become a bottleneck or single point of failure. Instead, orchestrators often rely on *atomic Git operations* (Git itself can handle concurrent fetches, and a push will fail if the remote changed, prompting a retry with backoff).

Task State Propagation and CRDTs: Keeping all agents in sync about which tasks are done, which failed, which are ready, is itself a state management problem. CRDTs shine here: for example, a **G-counter** (Grow-only counter) CRDT could track how many tasks have completed; more usefully, a **PN-set** CRDT can track the set of completed task IDs. Each agent updates its local copy when it finishes a task, and these sets merge without conflict, yielding a globally consistent view of completed tasks ²⁰ ²¹. If an agent crashes, its updates might be delayed, but as soon as it recovers or others receive its state, the overall system state becomes consistent. This approach tolerates network partitions and reconnection – no single agent’s failure will confuse the others about what’s done. In essence, using CRDTs or event sourcing logs ensures that agent coordination is **eventually consistent** and *monotonic* (we never “lose” knowledge of a completed task, as even if an update is late, it will be incorporated eventually ²⁰ ²¹). Another relevant CRDT is a *LWW-register* (last-writer-wins): for tracking a single authoritative value like “current HEAD of branch X”, LWW registers let whichever update has the latest timestamp win, preventing endless conflicts. These could be used to allow multiple agents to propose new HEADs (from merges), with a deterministic rule picking one.

Recommendations for Integration: A DAG-scoped Git orchestrator should leverage **shared state with eventual consistency** for agent coordination. Concretely, the system can maintain a **central task graph state** (or replicate it among agents) as a CRDT. Agents then mark tasks as running or done by updating this state, and all agents will see the update and adjust accordingly ²² ²¹. This avoids complex locking: two agents will never permanently think they own the same task, because the CRDT merge will clarify the single owner (e.g., perhaps using a *tiebreaker* like agent ID to decide if both try to pick up the same task). For retries, implement **exponential backoff with jitter** for any operation that can fail due to contention – e.g., if two agents try to push to the same branch, one’s push will fail; the failing agent should wait a random exponential interval before trying again ²⁵ ²⁶. This greatly increases the chance the push will succeed on the next attempt without manual intervention. Over the orchestration network, **bounded retries** should be combined with **failure escalation**: after X retries, a task is marked as error to be handled by a higher level (perhaps logged for human or retried later). Additionally, ensure that **task operations are idempotent**: an agent might attempt the same Git operation multiple times due to retries, so operations like “apply patch” or “create branch” should either check if the action was already done or be designed such that doing it twice has no side effect. Finally, use the CRDT state to facilitate *agent failover*: if agent A dies while holding a task, another agent can detect via the state (e.g., a heartbeat mechanism or simply a timeout on task start) and safely retry that task. The use of CRDTs and careful backoff means the system can achieve **high throughput (parallelism)** without sacrificing correctness or liveness (progress).

4. Fault Tolerance and Rollback Models

Error Propagation in DAGs: In a long-running DAG with many tasks, some tasks may fail. Understanding how failures propagate is key to designing resilience. One model is to attach a failure probability to each task and then compute the probability the entire DAG completes successfully. This can be done via a **probabilistic failure graph** analysis: essentially, the DAG becomes a Bayesian network of successes/failures. Recent research has introduced reliability models to measure fault tolerance of DAG executions, including *probabilistic execution path models* that calculate the success probability of a task sequence given node and link failure rates ³⁰ ³¹. For example, Cai *et al.* (2021) present a reliability model for edge computing DAG scheduling and an extended probabilistic path analysis to evaluate the impact of node/link failures on overall task completion ³⁰ ³¹. Such models help identify critical points in the graph where a failure would be most detrimental (perhaps a single node whose failure aborts many downstream tasks). Additionally, *resilient task graphs* may include **redundant paths or tasks** to handle failures: e.g., a DAG might have an alternate task B that can run if task A fails (forming a “fault-tolerant DAG” where at least one of {A, B} must succeed for the workflow to

continue). This starts to resemble *probabilistic workflows* or *dynamic DAGs* that change structure on failure.

Rollback-Safe Workflow Techniques: To recover from failures, systems use **checkpointing, lineage, and restart** techniques. **Checkpointing** means saving the state of a task or data at certain milestones so that if a failure occurs, you don't have to redo everything from scratch. In the context of Git worktrees, a checkpoint might be a commit or tag recorded after a complex series of changes, so that if something fails further on, the orchestrator can reset the worktree to that commit and retry from there. In distributed data processing, Spark's RDD model is a prime example: it forgoes replicating data in favor of *lineage-based recomputation*. A Resilient Distributed Dataset logs the *transformations* used to derive a dataset (its lineage); if a partition is lost, the system **recomputes it from the original data using the lineage log** ³². This is effectively a form of backward recovery using provenance. Spark will occasionally write out RDDs to disk ("checkpointing") to truncate very long lineage chains and reduce re-computation cost ³². For our orchestrator, lineage tracking would mean every output (e.g. a merged branch) "knows" which inputs (commits/branches) and tasks led to it. If the output is corrupted or lost, the orchestrator can retrace those dependencies and redo the necessary operations (for instance, re-merge the base branches into a new worktree).

Idempotence and Commutativity in Recovery: A fundamental principle for fault-tolerance is: *make operations idempotent and make state updates commutative whenever possible*. In distributed systems, it's said that *"if you're distributed, forget about ordering and start thinking about commutativity. Forget about guaranteed exactly-once delivery and start thinking about idempotence."* ³³ ³⁴. This insight reflects that enforcing a strict sequential order or exactly-once execution in a distributed setting is costly or impossible; instead, design operations that can be repeated or applied out-of-order without harming correctness. For the orchestrator, this means each task (like applying a commit, running a test, merging a branch) should either produce the same result if executed multiple times or have a way to detect it's already done. Git itself is largely idempotent in that applying the same patch twice results in no net change the second time (it either applies once or says "already applied"). If tasks are idempotent, **retries and rollbacks** are much simpler: a crash midway through a task is not catastrophic since the task can be safely re-run after resetting to a known state. **Commutativity** ensures that even if tasks happen slightly out of the intended order (due to delays or failures), the end state is consistent. For example, if two parallel tasks create two branches in a repo, the order in which they finish shouldn't matter – they produce two branches regardless. If there were a conflict (say both want to use the same branch name), that's a design issue; giving them distinct names makes their operations commutative.

Fault-Tolerant Scheduling and Replication: To achieve higher reliability, especially under hardware faults, orchestrators can replicate tasks. *Active replication* means running two or more instances of the same task in parallel (potentially on different agents or machines) and using the first successful result, ignoring the rest ³⁵. This is expensive resource-wise but can greatly improve success probability for flaky tasks. Some DAG scheduling algorithms incorporate **task duplication** strategically – for instance, replicating only tasks on the critical path or those feeding many downstream tasks ³⁶. The cited edge computing solution (RATM) duplicates certain tasks while managing CPU frequency (DVFS) to trade off energy vs reliability ³⁶. Another technique is **task retry with alternate strategies**: if task fails on Branch X, try it on Branch Y or with a smaller load, etc. The *sagas* pattern from databases is relevant: a **saga** breaks a transaction into a sequence of tasks, each with a compensating undo task. If any step fails, the compensating tasks run to roll back the prior steps, ensuring the system returns to a consistent state. In a Git workflow context, compensation might mean if a merge failed and partially introduced changes, one would perform a reverse patch to undo any partial application (though Git merges are atomic – they either succeed or not – but if a test after merge fails, "undo merge" by resetting HEAD is the compensating action).

Resilient Branching and Recovery: A *restart-safe branching* model implies that starting a workflow or branch can be done repeatedly without side effects. For example, if an orchestration workflow creates a temporary feature branch for testing, and the process crashes, restarting it should ideally not create a duplicate branch or leave the repo in a weird state. Techniques to ensure this include using unique branch names (like including a run ID or timestamp) so a restart makes a new branch rather than conflicting with the old one, or cleaning up (deleting) any existing temp branch at start. Another aspect is **atomicity** in branch updates: using Git’s reference locking, one can attempt to update a branch pointer and if it fails (someone else updated in the meantime), simply retry – ensuring that the branch is either updated to the new commit or left untouched, never in a half-baked state.

Recommendations for Integration: Our Git DAG orchestrator should embrace **lineage-based recovery**: log every Git operation (commits applied, merges done, etc.) in an append-only journal. This serves as both an audit trail and a lineage record, so if a node fails, a new agent can replay the journal (or pick up from a checkpoint) to reconstruct the state. Implement periodic **checkpoints** – e.g., after a major milestone like merging 50 branches, push the repository state to a remote backup or tag it – so that recovery can start from that point rather than from scratch. Use **idempotent actions** and explicitly check for prior completion: for instance, before creating a branch or pushing a commit, have the agent verify if that branch or commit already exists (perhaps from a previous run) and skip or update accordingly. To handle partial failures, design *compensating actions*: if a task times out or fails, ensure any side effect is either negligible or can be undone. For example, if a merge commit was made but tests failed, the orchestrator can automatically **git reset** the main branch to undo the merge commit (rollback) and mark that task as failed. By doing so, the system returns to a clean state as if the task never ran, allowing other tasks or retries to proceed without pollution.

To guard against multiple failures, incorporate *redundancy*: critical path tasks (like final deployment merges) could be run on two agents in parallel – the chances of both failing are low, and the faster one wins, the slower can be canceled. Use careful *timeout and heartbeat mechanisms* to detect hung tasks and trigger their retries or failovers. And when designing the DAG, consider adding **alternative paths** for important outcomes. For example, if “Merge feature to main” fails, perhaps the DAG could try an alternative strategy like “Rebase feature onto main then merge” as a different node. This makes the workflow more resilient by not giving up on the first failure.

In summary, adapt the mantra of commutativity and idempotence: *make each operation reversible or repeatable*. Combined with formal scheduling and coordination from prior sections, these fault-tolerance measures ensure the orchestrator can withstand agent crashes, merge conflicts, test failures, and other issues while still guaranteeing eventual completion of the DAG (or a graceful halt with a clear error state). By using these state-of-the-art techniques – from CRDT-coordinated task management ²⁰ ²¹ to exponential backoff ²⁵ ²⁶, from semantic merge checks ¹⁵ to lineage-driven rollback – a DAG-scoped Git worktree orchestration system can achieve a high degree of robustness with formal guarantees on correctness and reliability.

Sources: The insights and models above draw from a range of academic and industry sources, including scheduling theory textbooks ⁵ ¹¹, distributed systems literature on CRDTs and coordination ²² ²¹, software engineering research on merging and conflict resolution ¹⁵ ¹³, and practical guidelines from large-scale system builders (e.g. AWS’s backoff recommendations ²⁵ ²⁶). By integrating these proven techniques, the orchestration system can operate with formal underpinnings (for determinism and safety) while effectively handling the messy realities of parallel task execution, version control conflicts, and failures.

1 2 5 6 7 11 **mediatum.ub.tum.de**

<https://mediatum.ub.tum.de/doc/601655/601655.pdf>

3 4 **Series-parallel partial order - Wikipedia**

https://en.wikipedia.org/wiki/Series-parallel_partial_order

8 9 10 **DAG-based Task Planner Overview**

<https://www.emergentmind.com/topics/dag-based-task-planner>

12 **drum.lib.umd.edu**

<https://drum.lib.umd.edu/bitstreams/d4e5941c-79a9-48a2-9c8e-a8468d1f95eb/download>

13 14 **ww3.math.ucla.edu**

<https://ww3.math.ucla.edu/camreport/cam09-83.pdf>

15 16 17 **Detecting semantic conflicts with unit tests - ScienceDirect**

<https://www.sciencedirect.com/science/article/pii/S0164121224001158>

18 19 **The role of AI in merge conflict resolution**

<https://graphite.com/guides/ai-code-merge-conflict-resolution>

20 21 22 23 24 **CodeCRDT: Observation-Driven Coordination for Multi-Agent LLM Code Generation**

<https://arxiv.org/pdf/2510.18893>

25 26 27 28 29 **Timeouts, retries and backoff with jitter**

<https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>

30 31 36 **Failure-resilient DAG task scheduling in edge computing | Request PDF**

https://www.researchgate.net/publication/353661598_Failure-resilient_DAG_task_scheduling_in_edge_computing

32 **usenix.org**

<https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>

33 34 **THEORY: Distributed Transactions and why you should avoid them (2 Phase Commit , Saga Pattern, TCC, Idempotency etc) · GitHub**

https://gist.github.com/rponte/9477858e619d8b986e17771c8be7827f?permalink_comment_id=2874268

35 **(PDF) A Fault Tolerant Scheduling Algorithm for DAG Applications in ...**

https://www.researchgate.net/publication/267369702_A_Fault_Tolerant_Scheduling_Algorithm_for_DAG_Applications_in_Cluster_Environments