AgentGuard: A Formally Verified System Guardrail for Multi-Tasking Agentic Workflow

Yi-De Wu
National Taiwan University
yiidtw@gmail.com

***Abstract*—When multiple LLM agent sessions operate concurrently—editing the same files, sharing rate-limited API credentials, invoking overlapping tools—resource conflicts arise that current agent runtimes neither detect nor prevent. Our analysis of 43 bug reports from two production-grade open-source runtimes (227k+ and 14k+ GitHub stars) reveals three systematic failure classes: duplicate execution from violated mutual exclusion, deadlocks from unrecovered stale locks, and lost writes from TOCTOU races. This paper presents AGENTGUARD, a formally verified runtime guardrail that eliminates all three classes. We specify a composable TLA+ hierarchy (937 LOC, three modules) whose safety invariants are exhaustively verified by TLC across >1M states with zero violations. A conforming Rust daemon (∼4,700 LOC, 146 tests, 7 Kani proof harnesses) intercepts tool calls via PreToolUse/PostToolUse hooks, coordinating resource access with sub-millisecond overhead and without modifying the agent runtime. End-to-end experiments with Claude Code and Ollama confirm that AGENTGUARD prevents all 43 identified failure patterns while sustaining ∼210 ops/s from 1 to 50 concurrent sessions. The specification hierarchy provides a reusable formal foundation for any runtime designer integrating verified resource coordination into multi-agent systems.**

I. INTRODUCTION

A single chat message triggers 44 duplicate containers and spikes CPU usage to 450%. Scheduled tasks deadlock behind idle containers for over 9 hours. Two concurrent agent sessions silently corrupt the same configuration file, each overwriting the other's changes. These are not hypothetical scenarios—they are production bugs reported by users of two of the most widely deployed open-source LLM agent runtimes: OpenClaw (227k+ GitHub stars) [1] and NanoClaw (14k+ stars) [2].

LLM-based agents—systems that autonomously read files, call APIs, and execute code [3], [4]—are rapidly moving from single-user prototypes to multi-session production deployments. Frameworks such as Claude Code [5], LangChain [6], AutoGen [7], and MetaGPT [8] now power team environments, CI/CD pipelines, and automated workflows where multiple agent sessions operate concurrently on shared resources. Yet the runtime infrastructure underneath these agents has not kept pace: the scheduling, resource-coordination, and failure-recovery logic is implemented ad hoc and validated only by testing.

Our analysis of 43 publicly filed bug reports across these two runtimes reveals that concurrency failures are *systematic*, not incidental. They recur across independent projects, affect production users, and resist ad hoc patches because the root cause is structural: *agentic workflow systems lack a principled resource-coordination layer*. We target single-host, multi-session coordination—the common deployment model for coding agents—where distributed consensus is unnecessary but resource serialization is critical.

A. Problem Statement

We aim to provide formally verified runtime guardrails for concurrent multi-session agent execution, without requiring modifications to the agent runtime or the underlying LLM. Concretely, this paper investigates: *how can we formally specify and enforce resource-access invariants for concurrent AI agent sessions at runtime?*

The bug reports cluster into three classes, each mapping to a distinct formal property:

a) Duplicate Execution (Bug Class A — Mutual-Exclusion Violation).: When two agent sessions invoke tools on the same resource simultaneously, both proceed without coordination. A scheduling timestamp updated only *after* task completion allows the poll loop to re-enqueue a running task, causing duplicate replies and runaway resource consumption (NanoClaw #89: 44 containers from one message; OpenClaw #20740: duplicate message replies). 8 of 43 reports fall in this class. The required property is *mutual exclusion*: at most one session may hold a given resource at a time.

b) Deadlock and Stale Locks (Bug Class B — Liveness Violation).: Agent sessions may crash, time out, or enter infinite loops while holding a resource lock. Without automatic recovery, all sessions waiting for that resource are blocked indefinitely. This is the dominant failure mode: 22 of 43 reports—more than half—fall in this class, including system hangs (OpenClaw #13508) and 9-hour container deadlocks (NanoClaw #293). The required properties are *liveness* (every submitted task eventually terminates) and *stale-lock recovery* (crashed sessions' resources are automatically reclaimed).

c) Lost Writes (Bug Class C — Atomicity Violation).: A resource-availability check and the subsequent resource claim are separate operations; between them, a concurrent session can claim the same resource (TOCTOU race). In OpenClaw #24237, concurrent agents silently corrupted shared configuration by interleaving writes; in OpenClaw #10330, a device-auth token was overwritten between check and storage. 13 reports fall in this class. The required property is *atomic scheduling*: resource checking and claiming must occur in a single indivisible step.

Each class was individually patched in the respective projects, but without a unifying specification, patches cannot be verified

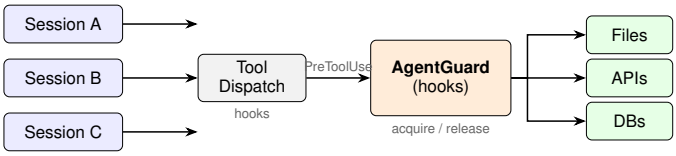

Fig. 1. Multi-session agent runtime with AGENTGUARD hook interception. AGENTGUARD serializes access to shared resources via Unix domain socket, without modifying the agent runtime.

to be complete or free from introducing new failures—precisely the scenario where formal methods provide the greatest value [9].

B. Background: Agent Runtime Architecture

Figure 1 illustrates the architecture of a typical multi-session agent runtime. Each session maintains an independent LLM conversation context and invokes tools (Read, Write, Edit, Bash, API calls) through a tool-dispatch layer. The tool-dispatch layer provides *hooks*—callback points before and after each tool invocation (PreToolUse, PostToolUse)—that external programs can intercept without modifying the runtime itself.

Current runtimes provide no coordination across sessions at this layer. A concrete interleaving illustrates the problem:

```
A: Read {count:0} -> Write {count:1}
B: Read {count:0} -> Write {count:1} // overwrites A
Result: count=1 (expected 2). A's write LOST.
```

This lost-write pattern occurs whenever two sessions read-modify-write the same resource without serialization. Existing mechanisms—file locks (`flock`), distributed locks (etcd, ZooKeeper)—are either too coarse-grained (locking entire files regardless of tool semantics) or designed for multi-host deployments with unnecessary network overhead. What is missing is a *tool-aware* coordination layer: one that groups Read, Write, and Edit into a single logical resource, supports session-level reentrant locks, and recovers automatically from agent crashes.

C. Technical Overview

AGENTGUARD addresses each bug class through a dedicated TLA+ specification module:

1) **Base mutual-exclusion scheduler** (415 LOC) → Bug Class A. A six-state task lifecycle with eight guarded transitions ensures *SkillExclusivity*: at most one task per resource in an active state. The Schedule action atomically checks availability and claims the resource in a single transition, eliminating TOCTOU races by construction (details in §III-C).
2) **Counted-semaphore rate limiting** (257 LOC) → Bug Class B. Generalizes mutual exclusion ($MaxConcurrent = 1$) to counted semaphores ($MaxConcurrent = k$) for rate-limited credentials, while preserving FIFO fairness and liveness. Preemption counts toward the retry limit—a design decision discovered by a TLC counterexample—preventing infinite rescheduling loops that cause the deadlocks in this class (details in §III-F).
3) **Multi-resource atomic transactions** (265 LOC) → Bug Class C. Tasks requiring multiple resources acquire all of them in a single atomic step (all-or-nothing), eliminating partial-hold deadlocks and interleaved writes. A **Deadlock-Freedom** invariant guarantees that schedulable tasks are never spuriously blocked (details in §III-G).

Each module strictly generalizes the previous while preserving all prior safety and liveness properties; TLC exhaustively verifies 8 safety invariants and 2 liveness properties across $>$1M states.

A conforming Rust daemon ($\sim$4,700 LOC) implements the specification as a single-threaded coordinator communicating via Unix domain socket. It intercepts tool calls through PreToolUse/PostToolUse hooks—a mechanism already supported by Claude Code and extensible to other runtimes. Fail-open semantics ensure that if the daemon is unreachable, agent execution proceeds normally—AGENTGUARD adds safety but never becomes a single point of failure.

D. Contributions

To the best of our knowledge, AGENTGUARD is the first formally verified resource-coordination layer for LLM agent runtimes. Our contributions are:

1) A **composable TLA+ specification hierarchy** (937 LOC, three modules) for resource coordination, with 8 safety invariants, 2 liveness properties, and a DeadlockFreedom guarantee exhaustively verified across $>$1M states (details in §III-C–§III-G).
2) A **4-layer verification chain** bridging specification and implementation: TLC → Kani/CBMC (7 proof harnesses) → proptest (500K transitions) → 146 targeted tests. Two TLC counterexamples directly shaped the design (details in §III-D–§III-E).
3) A **systematic analysis of 43 concurrency bugs** from two production agent runtimes, classifying them into three formally preventable categories (details in §V-C).
4) **Empirical validation** showing sub-millisecond coordination overhead and constant throughput ($\sim$210 ops/s) across 1–50 concurrent sessions, with Claude Code and Ollama (details in §V-B–§V-D).
5) All TLA+ specifications, Rust source, Kani harnesses, and experimental scripts are **publicly released**.[1]

II. RELATED WORK

a) TLA+ in industry.: Newcombe et al. [9] report that AWS applied TLA+ to 10+ production systems, finding subtle bugs in every one—with configurations of just 3–4 processes. MongoDB used TLA+ to catch linearizability violations invisible to years of testing [10]; Raft was co-designed with a TLA+ specification [11]; Azure Cosmos DB uses Dafny for verified consistency [12]. All of these target distributed infrastructure (consensus, replication, storage). Our work extends TLA+-based verification to a new domain—agent runtime scheduling—and contributes a specification

[1] https://github.com/yiidtw/agentguard

hierarchy rather than a single module, demonstrating systematic generalization while preserving verified properties.

b) Formal verification of concurrent systems.: Model checking has a rich history in verifying concurrent protocols, from hardware cache coherence to distributed consensus. The CHESS tool [13] systematically explores thread interleavings to find concurrency bugs; PCT [14] provides probabilistic guarantees. These tools operate at the implementation level, exploring actual code paths. In contrast, TLC model checking operates at the specification level, exploring *all* reachable states within bounds—providing exhaustive coverage that implementation-level tools cannot match (at the cost of requiring a separate specification). Our approach combines specification-level exhaustive verification (TLC) with implementation-level randomized testing (property-based testing [15], [16]), providing two complementary layers of assurance.

c) Agent frameworks and concurrency.: LangChain [6], AutoGen [7], MetaGPT [8], CrewAI [17], and OpenAI Swarm [18] provide multi-agent orchestration but lack formal scheduling guarantees. AIOS [19] introduces an LLM agent operating system with hashmap-based conflict detection for shared resources; however, it provides no formal specification or verified safety properties. AgentBench [20] evaluates task completion without addressing scheduler correctness. The three bug classes we identify (§I-A) are drawn from OpenClaw [1] and NanoClaw [2], demonstrating that ad-hoc concurrency control leads to recurring correctness issues even in production systems. AGENTGUARD differs from all of the above in providing *formally verified* resource coordination with exhaustively checked safety and liveness properties.

d) Distributed coordination services.: etcd, ZooKeeper (ZAB), and Consul provide distributed locks and leader election using Raft/ZAB consensus protocols. These target multi-host coordination with network round-trips ($\sim$1–10 ms per lock) and 3+ node clusters. Redis-based locks (Redlock [21]) offer lower latency but only probabilistic safety under clock skew. Advisory file locks (`flock`) lack FIFO fairness and formal specification. AGENTGUARD occupies a different design point: single-host, single-threaded coordination where distributed consensus is unnecessary. Its formal contribution is orthogonal—we specify and verify the *scheduling logic* itself, not the underlying consensus protocol.

III. TLA+ SPECIFICATION HIERARCHY

This section presents the TLA+ specification hierarchy that formalizes the three resource-coordination properties identified in §I-A: mutual exclusion (Bug Class A), liveness with stale-lock recovery (Bug Class B), and atomic multi-resource acquisition (Bug Class C). The hierarchy consists of three modules, each strictly generalizing the previous while preserving all prior safety and liveness properties.

A. System Model and State Space

AGENTGUARD models three entity types: *tasks* (units of work), *skills* (shared resources), and *nodes* (execution sites). The base specification uses 10 TLA+ variables organized into four groups:

- **Task state:** $taskState[t] \in \{\texttt{init}, \texttt{pending}, \texttt{scheduled}, \texttt{running}, \texttt{done}, \texttt{failed}\}$
- **Task metadata:** $taskSkill[t]$ (skill binding), $taskNode[t] \in NodeId \cup \{\texttt{NULL}\}$, $taskRetries[t] \in 0..MaxRetries$, $taskSubmitSeq[t]$ (FIFO counter)
- **Skill state:** $skillState[s] \in \{\texttt{available}, \texttt{busy}\}$, $skillHolder[s] \in TaskId \cup \{\texttt{NULL}\}$
- **Node state:** $nodeState[n] \in \{\texttt{online}, \texttt{offline}\}$
- **Global:** $submitted \subseteq Tasks$ (submitted task set), $nextSeq$ (monotonic counter)

The state space decomposes multiplicatively. For the default configuration of $|T|$ tasks, $|S|$ skills, and $|N|$ nodes:

- Task state contributes $6^{|T|}$ values (6 states per task).
- Task metadata contributes $(|N|+1)^{|T|} \times (MaxRetries+1)^{|T|} \times |T|!$ values (node assignment, retry count, and submission ordering).
- Skill state contributes $2^{|S|} \times (|T|+1)^{|S|}$ values (available/busy and holder).
- Node state contributes $2^{|N|}$ values.
- Global variables contribute $2^{|T|} \times (|T|+1)$ values (submitted set and sequence counter).

Not all combinations are reachable—invariants constrain the space significantly. Under our primary configuration (4 tasks, 2 skills, 2 nodes, $MaxRetries = 2$), TLC explores $>$1M reachable states. This configuration is sufficient to exhibit all three bug classes: Bug A requires $\geq$2 tasks sharing 1 skill, Bug B requires 1 task on a crashing node with retry mechanics, and Bug C requires 2 tasks racing for 1 skill. Following the *small-scope hypothesis* [9]—AWS reports that configurations of 3–4 processes found "subtle, serious bugs" in production systems operating at vastly larger scale—our 4/2/2 configuration provides combinatorial room for all relevant interleavings.

We additionally verify a larger configuration (6 tasks, 3 skills, 3 nodes) with safety-only checking (liveness omitted for tractability), confirming that no new invariant violations appear at increased bounds.

B. Task Lifecycle State Machine

The task lifecycle is a six-state machine with eight guarded transitions, shown in Figure 2.[2] States are partitioned into *active* (`scheduled`, `running`), *terminal* (`done`, `failed`), and *non-terminal* (`init`, `pending`). A critical structural property: a task holding a skill is always active, and releasing a skill always precedes leaving the active states.

Note on Init elision. The TLA+ specification retains `init` as a modeling artifact required by TLC for initial state exploration. The Rust implementation elides `init` entirely—`submit()` atomically creates tasks in `pending`—yielding a 5-state machine. This is safe because no external observer can witness the `init`→`pending` transition; the TLA+ specification verifies

[2]The lifecycle follows the classic 5-state process model from operating systems (new → ready → running → waiting → terminated) [22], with state names adapted from HPC batch scheduler conventions (SLURM [23], Kubernetes Pod Lifecycle [24]).

that the only outgoing edge from `init` is Submit, and no invariant references `init` as a reachable post-action state.

We describe all eight actions with their formal guards and effects.

a) Submit(t): transitions a task from `init` to `pending` and records a monotonic submission sequence number for FIFO ordering. The task enters the *submitted* set.

b) Schedule(t, s, n): is the critical action. It is **atomic**: claiming skill s (setting $skillState[s] \leftarrow$ `busy`, $skillHolder[s] \leftarrow t$) and assigning node n ($taskNode[t] \leftarrow n$) happen in a *single* transition. Five guards must hold simultaneously:

1) $taskState[t] =$ `pending`
2) $taskSkill[t] = s$ (task requires this skill)
3) $skillState[s] =$ `available` (mutual exclusion)
4) $nodeState[n] =$ `online` (target node alive)
5) **NoBypass:** $\neg\exists\, t_2 \in submitted : t_2 \neq t \wedge taskSkill[t_2] = s \wedge taskState[t_2] =$ `pending` $\wedge\ taskSubmitSeq[t_2] < taskSubmitSeq[t]$ (FIFO fairness)

This atomicity eliminates Bug C by construction—there is no window between the resource availability check and the resource claim. The TLA+ encoding is:

```
Schedule(t, s, n) ==
  /\ taskState[t] = "pending"
  /\ taskSkill[t] = s
  /\ skillState[s] = "available"
  /\ nodeState[n] = "online"
  /\ ~\E t2 \in submitted :  \* NoBypass (FIFO)
       /\ t2 /= t /\ taskSkill[t2] = s
       /\ taskState[t2] = "pending"
       /\ taskSubmitSeq[t2] < taskSubmitSeq[t]
  /\ taskState'  = [taskState  EXCEPT ![t] = "scheduled"]
  /\ taskNode'   = [taskNode   EXCEPT ![t] = n]
  /\ skillState' = [skillState EXCEPT ![s] = "busy"]
  /\ skillHolder'= [skillHolder EXCEPT ![s] = t]
  /\ UNCHANGED <<taskSkill, taskRetries, taskSubmitSeq,
                 nodeState, submitted, nextSeq>>
```

c) Start(t): transitions `scheduled` $\rightarrow$ `running`, modeling the two-phase dispatch where a node acknowledges task receipt. Guards require the assigned node to be online and the task to hold its skill.

d) Complete(t): transitions `running` $\rightarrow$ `done` and releases the skill ($skillState \leftarrow$ `available`, $skillHolder \leftarrow$ `NULL`) and clears the node assignment.

e) Fail(t): releases the skill *before* evaluating retry eligibility. If $taskRetries[t] < MaxRetries$, the task returns to `pending` with $retries$ += 1; otherwise it enters `failed`. Rescheduling requires a fresh Schedule with full guard checking, preventing Bug A.

f) Preempt(t): rescues tasks stranded on offline nodes ($nodeState[taskNode[t]] =$ `offline`). It applies to both `scheduled` and `running` tasks. A critical design decision, discovered by TLC (discussed in §III-E): preemption counts toward the retry limit.

g) NodeCrash(n) / NodeRecover(n): toggle $nodeState[n]$ between `online` and `offline`, modeling infrastructure failures. NodeCrash does *not* directly affect tasks—Preempt handles recovery, maintaining separation of concerns.

C. Verified Properties

TLC verifies eight safety invariants across all reachable states, grouped by category.

a) Safety (mutual exclusion and type correctness).:

1) **TypeOK**: All 10 variables are within their declared domains.
2) **SkillExclusivity**: No two tasks using the same skill are simultaneously in {`scheduled`, `running`}. Formally: $\forall s \in Skills : |\{t \in Tasks : taskSkill[t] = s \wedge taskState[t] \in \{\texttt{scheduled}, \texttt{running}\}\}| \leq 1$. Directly prevents Bug A.
3) **NoBypassSafety**: A later-submitted task for the same skill cannot be active while an earlier-submitted task is still pending. Ensures FIFO fairness is maintained at the state level, not just the transition level.
4) **NoDeadlockWithResources**: If a pending task exists whose skill is available and at least one node is online, then some task must be schedulable. This proves that FIFO ordering is well-founded and cannot create circular dependencies.

b) Consistency (bookkeeping correctness).:

5) **SkillConsistency**: $skillState[s] =$ `busy` $\iff skillHolder[s] \neq$ `NULL`, and any holder is in {`scheduled`, `running`} with matching skill.
6) **NoOrphanedAssignment**: $taskNode[t] \neq$ `NULL` $\Rightarrow taskState[t] \in$ {`scheduled`, `running`}.
7) **MaxUtilization**: The set of busy skills exactly equals the set of skills held by active tasks—zero bookkeeping overhead from the safety layer.
8) **OnlySkillConflictsBlock**: A pending task is blocked only by skill conflicts, absence of online nodes, or FIFO ordering—never by cross-skill interference. This proves the safety mechanism introduces no unnecessary blocking.

c) Liveness (progress guarantees).: Two temporal properties hold under fairness assumptions:

1) **Liveness**: $\forall t \in Tasks : t \in submitted \rightsquigarrow taskState[t] \in$ {`done`, `failed`}. Every submitted task eventually reaches a terminal state.
2) **SkillEventuallyFree**: $\forall s \in Skills : skillState[s] =$ `busy` $\rightsquigarrow skillState[s] =$ `available`. No skill remains busy forever.

D. Fairness Requirements: WF vs. SF

The distinction between weak and strong fairness is a key formal contribution of this work. In TLA+, weak fairness $\text{WF}_v(A)$ requires that if action A is *continuously* enabled (from some point onward, it remains enabled at every state), it must eventually be taken. Strong fairness $\text{SF}_v(A)$ requires that if A is *infinitely often* enabled (it keeps becoming enabled, though it may be disabled in between), it must eventually be taken.

Our specification requires:

- **Weak fairness (WF)** for Submit, Complete, Fail, and Preempt. These actions, once enabled, remain continuously enabled until taken. For example, once a running task's node goes offline, the Preempt action remains enabled until executed.
- **Strong fairness (SF)** for Schedule, Start, and NodeRecover. These actions may be repeatedly enabled and disabled.

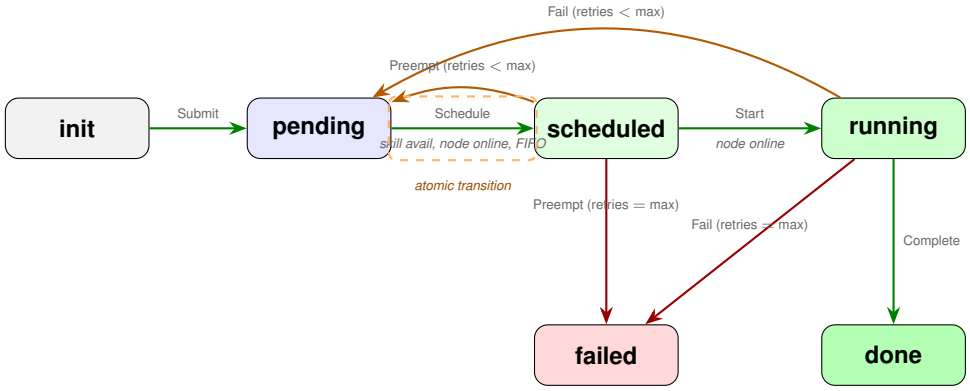

Fig. 2. Task lifecycle state machine. Green arrows show the happy path; orange arrows show retry paths (retries $<$ max); red arrows show terminal failure (retries $=$ max). The dashed box highlights the *atomic* Schedule transition: skill claim and node assignment occur in a single step, eliminating TOCTOU races (Bug C).

Schedule, for instance, is enabled when a skill becomes free but may be disabled when another task claims it first.

The formal specification of the next-state relation with fairness is:

$$Spec \equiv Init \wedge \Box[Next]_v \wedge \bigwedge_{A \in WF} \mathrm{WF}_v(A) \wedge \bigwedge_{A \in SF} \mathrm{SF}_v(A)$$

TLC found a concrete counterexample when only weak fairness is used for Schedule: a task is assigned to node n_1, the node crashes, the task is preempted back to pending, and the scheduler *always* picks n_1 again (which is valid under WF because the action is not *continuously* enabled for n_2—other tasks may intermittently claim the skill, disabling Schedule for n_2). Under strong fairness, if scheduling to n_2 is infinitely often possible, it must eventually happen. This is a reasonable assumption for any non-adversarial scheduler implementation.

The counterexample trace (simplified) is:

```
S1: t1 pending, n1 online, n2 online
S2: Schedule(t1,s1,n1) -> scheduled on n1
S3: NodeCrash(n1) -> n1 offline
S4: Preempt(t1) -> pending (retries=1)
S5: NodeRecover(n1) -> n1 online
S6: Schedule(t1,s1,n1) -> scheduled [AGAIN]
... (infinite loop, never tries n2)
```

This counterexample is valuable because it identifies a real implementation concern: a scheduler that preferentially routes to recently recovered nodes could exhibit this behavior.

E. The Preemption-Counts-as-Retry Design Decision

The most significant design decision discovered by model checking involves the interaction between preemption and retry limits. The initial specification treated preemption as a "free" rescheduling that did not increment the retry counter. TLC found a liveness violation:

Counterexample. Consider task t assigned to a node that repeatedly crashes and recovers. Without counting preemptions as retries, the following infinite behavior is valid:

$$\texttt{pending} \xrightarrow{\text{Sched}} \texttt{scheduled} \xrightarrow{\text{Preempt}} \texttt{pending} \xrightarrow{\text{Sched}} \texttt{scheduled} \xrightarrow{\text{Preempt}} \cdots$$

The task never reaches a terminal state (`done` or `failed`), violating the Liveness property. Moreover, during each `scheduled` phase, the task holds a skill, preventing other tasks from using it. This means a single task on a flaky node can starve all other tasks requiring the same skill, violating SkillEventuallyFree.

Fix. Preempt increments $taskRetries[t]$, ensuring the loop terminates after $MaxRetries$ iterations. After the fix, the task eventually enters `failed` (releasing its skill) if the node continues crashing. This design decision was not obvious from code-level reasoning—the intuition that "preemption is not the task's fault" argued against counting it. The TLC counterexample made the liveness implication immediate and unambiguous.

This is a concrete example of Newcombe et al.'s observation that "the value of formal specification is not just in finding bugs, but in the deeper understanding of the design it provides" [9].

F. Module 2: Counted Semaphores for Rate Limiting

The base specification enforces $|\text{holders}(s)| \leq 1$ per skill (mutual exclusion). A natural generalization is $|\text{holders}(s)| \leq k$ for $k \geq 1$—a counted semaphore. This models API credentials that permit at most k concurrent uses.

The 257-line extension module replaces the scalar $skillHolder[s] \in TaskId \cup \{\texttt{NULL}\}$ with a set $skillHolders[s] \subseteq Tasks$ and introduces a constant $MaxConcurrent[s] \in \mathbb{N}^+$. The central invariant is:

$$SkillConcurrencyBound \equiv \forall s \in Skills: \; |skillHolders(s)| \leq MaxConcurrent(s)$$

The original SkillExclusivity is recovered as a special case: $MaxConcurrent = [s \in Skills \mapsto 1]$ reduces SkillConcurrencyBound to $|skillHolders(s)| \leq 1$, which (given that holders are active tasks) is equivalent to the base module's at-most-one-holder constraint.

The Schedule action's guard changes from $skillState[s] =$ `available` to $|skillHolders[s]| < MaxConcurrent[s]$, and skill release removes the task from the holders set rather than setting a scalar to NULL.

TLC verifies 6 safety invariants and 2 liveness properties for this module under a configuration of 3 tasks, 3 skills, 2 nodes, with various $MaxConcurrent$ values. The key formal result is that FIFO fairness and eventual termination are preserved by the generalization—the NoBypass guard naturally extends to the multi-holder case.

G. Module 3: Multi-Resource Transactions with DeadlockFreedom

Real-world tasks may require *multiple* resources simultaneously. A "Deploy" operation might need both file system access and network access; acquiring them sequentially risks deadlock if another task holds one and waits for the other.

The 265-line extension module generalizes $taskSkill[t] \in Skills$ to $TaskSkillSet[t] \subseteq Skills$. The Schedule action acquires **all** required skills atomically:

```
Schedule(t, n) ==
  /\ taskState[t] = "pending"
  /\ nodeState[n] = "online"
  /\ \A s \in TaskSkillSet[t] :
       Cardinality(skillHolders[s])
         < MaxConcurrent[s]
  /\ \A s \in TaskSkillSet[t] :
       ~\E t2 \in submitted : ... \* NoBypass
  /\ skillHolders' = [s \in Skills |->
       IF s \in TaskSkillSet[t]
       THEN skillHolders[s] \cup {t}
       ELSE skillHolders[s]]
```

Because acquisition is all-or-nothing within a single atomic transition, there is no partial holding and thus no circular wait—the classical necessary condition for deadlock [25] is eliminated by construction. TLC verifies a **DeadlockFreedom** invariant:

$$\forall t\colon\ taskState[t] = \texttt{pending} \land \\ (\forall s \in TaskSkillSet[t]\colon\ |holders[s]| < MaxConc[s]) \\ \land\ (\exists n\colon\ nodeState[n] = \texttt{online}) \\ \Rightarrow \text{Schedulable}(t)$$

This invariant states that whenever a pending task's resource requirements can be satisfied (all skills have capacity, a node is available), the task can indeed be scheduled—no hidden blocking conditions exist.

H. Generalization Hierarchy Summary

Figure 3 illustrates the specification hierarchy. Each module strictly generalizes the previous:

- **Base → Rate Limiting**: $skillHolder : TaskId \cup \{\texttt{NULL}\}$ becomes $skillHolders : \mathcal{P}(Tasks)$; SkillExclusivity generalizes to SkillConcurrencyBound. Setting $MaxConcurrent = 1$ recovers the base case exactly.
- **Rate Limiting → Multi-Skill**: $taskSkill[t] \in Skills$ becomes $TaskSkillSet[t] \subseteq Skills$; the Schedule guard checks all skills atomically. DeadlockFreedom is verified as a new invariant. Setting $|TaskSkillSet[t]| = 1$ recovers the rate-limiting case.

This hierarchy demonstrates that mutual exclusion, counted semaphores, and multi-resource transactions are not separate mechanisms but points on a continuum. Each extension adds one dimension of generality (from scalar to set, from single to multi) while preserving the algebraic structure that makes the safety and liveness proofs work.

IV. Implementation and Conformance Testing

A. Rust Implementation

A Rust implementation (∼960 LOC kernel + ∼700 LOC daemon + ∼240 LOC invariant checker + ∼2,800 LOC supporting modules) faithfully translates the TLA+ specification with several engineering refinements:

a) Type-safe identifiers.: Rust newtype wrappers enforce compile-time distinction between entity types:

```
pub struct TaskId(pub String);
pub struct SkillId(pub String);
pub struct NodeId(pub String);
```

The compiler rejects mixing ID types at zero runtime cost. This catches argument-ordering bugs that the TLA+ specification cannot express.

b) Atomicity guarantee.: The daemon processes requests sequentially on a single-threaded Tokio runtime. Each action method runs to completion without preemption—the same atomicity assumed in the TLA+ specification is provided by the runtime model. No locks, mutexes, or compare-and-swap operations are needed.

c) SQLite persistence.: The production store uses SQLite with WAL (Write-Ahead Logging) mode. State mutations and transition log entries are wrapped in a single serializable transaction, ensuring the same atomicity guarantees hold across process restarts.

d) Compile-time invariant flag.: An `enable_invariant_checks` option controls whether invariants are verified after each action. In production, invariants are disabled for throughput; in CI and development, they provide continuous verification.

B. Conformance Testing

A common concern with model-checked specifications is the gap between the verified TLA+ model and the production implementation. While a formal refinement mapping would provide the strongest guarantee, we adopt a pragmatic approach—*conformance testing*—following the practice described by AWS [9].

The implementation is validated by 146 tests across eight categories:

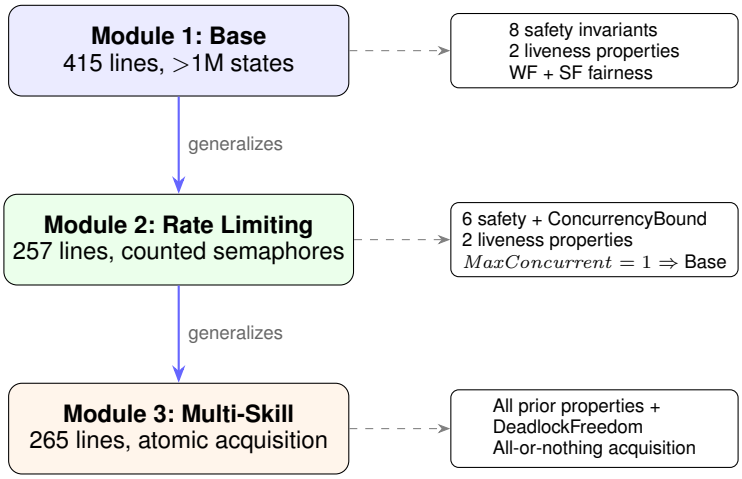

Fig. 3. TLA+ specification hierarchy. Each module strictly generalizes the previous while preserving all verified properties. Module 2 reduces to Module 1 when $MaxConcurrent = 1$. Module 3 adds multi-resource atomic acquisition with a DeadlockFreedom guarantee.

Category	Coverage	Tests
Kernel unit	All 8 actions individually	52
Invariants	8 safety invariants	29
Integrity	Path/cmd/tool envelope, RBAC	24
Scenarios	Multi-step lifecycle sequences (S1–S8)	17
Integration	Daemon-level bug reproductions	9
Bisimulation	TLA+ trace conformance	8
Store	State persistence	4
Random walks	Property-based invariant checks	3
Total		**146**

The eight integration scenarios (S1–S8) are designed to reproduce the exact failure modes that motivated the specification:

- **S1** (Duplicate execution): Two tasks needing the same skill; second `schedule()` correctly fails with `SKILL_BUSY`.
- **S2** (Deadlock prevention): Task on crashing node is preempted, skill released, rescheduled on a different node, completes.
- **S3** (TOCTOU prevention): Atomic schedule leaves no intermediate state; concurrent schedule for same skill correctly rejected.
- **S4** (Parallel skills): Two tasks with different skills run simultaneously; MaxUtilization holds.
- **S5** (FIFO ordering): Earlier-submitted task must be scheduled first; later task fails with `FIFO_BYPASS`.
- **S6** (Crash/preempt/retry): Node crash → preempt → retry on new node → completion.
- **S7** (Max-retry failure): Task exhausting retries enters `failed`; skill released.
- **S8** (Full lifecycle): All 8 invariants checked after every step in a complete multi-task lifecycle.

Property-based testing via `proptest` fires random transition sequences with all 8 invariants checked after each step—**zero violations**. Each walk randomly selects enabled actions, verifying that the implementation maintains all invariants regardless of action ordering.

C. Kani Bounded Model Checking

Because Kani/CBMC performs symbolic execution and cannot handle `BTreeMap<String, _>` symbolically, we create a *bounded array model* (∼520 LOC) that mirrors the kernel logic using fixed-size arrays with bounded indices (`MAX_TASKS`=2, `MAX_SKILLS`=2, `MAX_NODES`=2, `MAX_RETRIES`=2). All 8 actions and 8 invariant checkers are re-implemented against this model.

Seven Kani proof harnesses verify the model:

Harness	Steps	Property
`verify_invariants_5`	5	All 8 invariants
`verify_exclusivity_7`	7	SkillExclusivity
`verify_no_orphaned_7`	7	NoOrphanedAssign
`verify_no_bypass_7`	7	NoBypassSafety
`verify_submit_pending`	1	submit ⇒ Pending
`verify_complete_rel`	1	complete ⇒ released
`verify_integrity_5`	5	IntegrityEnvelope

Each harness uses `kani::any()` nondeterministic choices to explore all reachable action sequences within the bounded configuration. This establishes the following 4-layer verification chain:

Layer	Tool	Target	Guarantee	Scope
L1	TLC	TLA+ spec	Exhaustive	>1M states
L2	Kani	Rust model	Bounded	2T/2S/2N
L3	proptest	Rust kernel	Statistical	Random
L4	#[test]	Rust kernel	Targeted	146 tests

L1 verifies the abstract specification exhaustively. L2 closes the gap between specification and implementation by exhaustively verifying the Rust model within bounds. L3 provides statistical coverage of the full kernel with its dynamic data structures. L4 covers targeted bug-reproduction scenarios. Together, the four layers provide assurance that no single technique alone can offer.

V. Experiments

We evaluate AgentGuard along four dimensions: (1) exhaustive model checking of the TLA+ specification, (2) schedul-

TABLE I
TLC MODEL CHECKING RESULTS ACROSS THE SPECIFICATION HIERARCHY.

Module	Config	States	Properties	Viol.
Base	4T/2S/2N	>1M	8 safety + 2 liveness	0
Base (lg)	6T/3S/3N	>10M	8 safety only	0
Rate Lim.	3T/3S/2N	>500K	6 safety + 2 liveness	0
Multi-Skill	3T/3S/2N	>500K	7+DLFree + 2 liveness	0

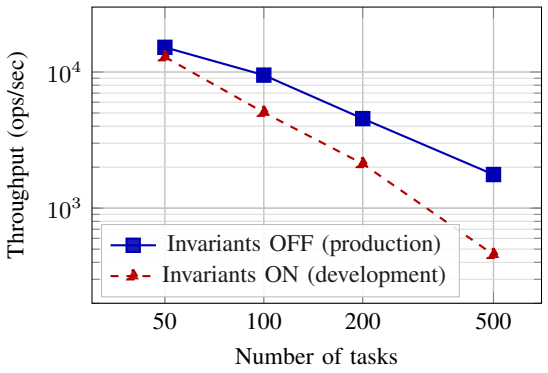

Fig. 4. Scheduling throughput vs. number of tasks. In production mode (invariants off), the coordinator sustains >15,000 ops/sec at 50 tasks and >1,700 ops/sec at 500 tasks. Invariant checking adds $O(n^2)$ overhead due to pairwise property verification.

ing throughput of the Rust implementation, (3) prevention coverage across the 43 bug reports from §I-A, and (4) end-to-end validation with production LLM agent runtimes.

A. Model Checking Results

Table I summarizes the TLC model checking results across all three specification modules.

The state space decomposes as follows: task state (6 values $\times$ $|T|$ tasks), skill state (2 values $\times$ $|S|$ skills), node state (2 values $\times$ $|N|$ nodes), plus auxiliary variables for retries, node assignments, and FIFO ordering. TLC uses breadth-first search to enumerate all reachable states, checking each invariant at every state and verifying liveness across all complete behaviors.

Under the base configuration (4 tasks, 2 skills, 2 nodes, $MaxRetries = 2$), the 4/2/2 parameters provide: 2 tasks per skill (sufficient for contention), 2 nodes (sufficient for crash/recovery scenarios), and 2 retries (sufficient for retry-limit exhaustion). All three bug classes are covered with combinatorial room for interleaving.

B. Scheduling Throughput

We benchmark full task lifecycles (submit→schedule→start→complete) on a single-threaded Rust process to validate that the formal model's atomicity assumptions are practical.

In production mode, the implementation sustains **15,184 ops/sec** at 50 tasks and **9,488 ops/sec** at 100 tasks. Invariant checking adds 1.2×–3.9× overhead ($O(n^2)$ from pairwise comparisons in SkillExclusivity), confirming the design decision to provide a compile-time toggle.

C. Bug Prevention Analysis

We analyzed 43 publicly filed bug reports from OpenClaw [1] (18 issues from 227k+ stars) and NanoClaw [2] (25 issues from 14k+ stars). Reports were selected by searching each project's issue tracker for keywords related to concurrency failures (race, deadlock, duplicate, timeout, stale, overwrite, leak) and manually verifying that the root cause involves concurrent resource access. Each report was classified by two of the authors into one of the three bug classes; disagreements were resolved by discussion.

Table II summarizes the classification and the verified property that prevents each class.

TABLE II
BUG CLASS DISTRIBUTION (43 REPORTS) AND VERIFIED PREVENTION.

Bug Class	OC	NC	Verified Prevention
A: Duplicate execution	4	4	SkillExclusivity
B: Deadlock / stale lock	12	10	Preempt + Liveness
C: Lost write / TOCTOU	2	11	Atomic Schedule
Total	18	25	

Bug Class B (liveness violations) dominates at 51% of all reports, consistent with the observation that failure recovery is the hardest property to get right without formal guarantees—stale-lock timeout, retry-limit interaction, and crash recovery each introduce subtle edge cases that resist ad hoc testing.

For each class, the prevention is not a patch but a *proven property*: SkillExclusivity is checked at every reachable state (not just test cases), Liveness holds across all possible behaviors (not just observed executions), and atomicity is inherent in the transition definition (not enforced by an external lock that could be forgotten).

D. Scalability

We measure coordination latency as concurrent sessions scale from 1 to 50 (Table III). Each session performs 20 acquire+work+release cycles (1ms simulated work), with 5 repetitions per configuration.

Throughput remains constant at ∼210 ops/s across all session counts, demonstrating that the formally verified coordinator introduces no scaling bottleneck. Each operation involves two Unix socket round-trips (acquire + release) plus 1ms simulated work; the ∼4ms per-acquire/release coordination overhead reflects Unix socket serialization on the single-threaded daemon.

E. Application-Domain Validation: LLM Agent Runtimes

To demonstrate that the formal guarantees translate to real-world correctness, we apply the verified coordinator to LLM-based agent runtimes. The coordinator intercepts tool calls through hooks, serializing access to shared resources.

a) Experiment setup.: Three concurrent agent sessions simultaneously read-modify-write a shared JSON file. Each session reads the current counter value, increments it, and appends its session ID to a log array. The expected result is `count=3` with all three session IDs in the log.

TABLE III
SCALABILITY: COORDINATION WITH 1–50 CONCURRENT SESSIONS (5 REPS, 20 OPS/SESSION, 1MS WORK).

Sess.	Wall (ms)	Tput (ops/s)	p50	p95
1	126±9	160±11	6.0	6.2
2	194±8	207±8	9.2	10.9
5	486±12	206±5	18.5	22.2
10	942±6	212±1	27.3	41.6
20	1842±15	217±2	32.0	221.2
50	4836±91	207±4	68.0	741.6

TABLE IV
END-TO-END RESULTS: 3 CONCURRENT LLM AGENT SESSIONS EDITING THE SAME FILE.

	Count	Wall (ms)	Cost	Verdict
Without	2/3	16,643	$0.18	Lost write
With	3/3	21,631	$0.15	Correct
Overhead		+30%	−18%	

b) Key design: resource grouping with non-releasing Read.: Per-tool locking is insufficient because the lock is released between Read and Write. The coordinator solves this with: (1) *resource grouping* maps Read, Write, and Edit to a single lock; (2) *session-level reentrant locks* allow the same session to acquire the resource for both Read and Write; (3) *non-releasing Read* holds the lock across the Read→Write gap. A stale-lock timeout (30s) handles read-only sessions.

c) Results.: Table IV shows the results. Without the coordinator, the race condition triggers reliably: one session's write is overwritten, yielding `count=2`. With the coordinator, all three writes are preserved. The 30% wall-clock increase reflects serialized execution; coordination overhead per operation is sub-millisecond.

d) Local LLM validation.: We additionally test with local models via Ollama [26]: Qwen 2.5 32B achieves 100% correctness at 3, 5, and 10 concurrent agents, while Llama 3.1 70B fails due to malformed tool-calling output despite correct coordination (locks acquired and released in proper order). This validates that the coordinator cleanly separates the coordination concern from the computation concern—a model with reliable tool calling benefits immediately, while coordination correctness is independent of LLM output quality.

e) Failure recovery.: Failure injection tests verify three recovery scenarios: (1) daemon crash—clean restart with implicit lock release; (2) session crash—stale lock timeout (30s) automatically releases held resources; (3) credential leak prevention—semaphore counts return to zero after session crash. These correspond to the Preempt and NodeCrash/NodeRecover actions in the TLA+ specification.

VI. DISCUSSION

a) Specification hierarchy as a design methodology.: The 3-module hierarchy demonstrates a systematic approach to formal specification: start with the simplest correct model (mutual exclusion), then generalize along one dimension at a time (scalar → set for counted semaphores, single → multi for resource sets), verifying at each step that all prior properties are preserved. This incremental approach makes the specification tractable—each module is small enough to understand in isolation—while the generalization proofs ensure global consistency. We believe this methodology is applicable beyond resource coordination to other domains where concurrency requirements evolve.

b) Counterexamples as design tools.: Two TLC counterexamples directly shaped the design: the preemption-retry interaction (§III-E) and the WF/SF distinction (§III-D). In both cases, the counterexamples identified real implementation concerns (infinite rescheduling on flaky nodes, preferential routing to recently recovered nodes) that were not apparent from informal reasoning. This confirms the observation by Newcombe et al. [9] that "the value of formal specification is not just in finding bugs, but in the deeper understanding of the design it provides."

c) Bounded verification and the small-scope hypothesis.: Our verification is bounded: 4/2/2 for the base module, 3/3/2 for rate limiting and multi-skill. While the small-scope hypothesis [9] and AWS's experience suggest that small configurations suffice to find design bugs (because they arise from logical errors in state transitions, not from scale), bounded model checking cannot guarantee the absence of bugs at larger scales. Unbounded proofs require inductive invariants—manually discovering invariants strong enough to be inductive is the main obstacle. Tools like Apalache (a symbolic model checker for TLA+) could help bridge this gap by checking bounded instances symbolically rather than enumeratively, potentially enabling larger bounds with the same computational resources.

d) Rate limiting as algebraic generalization.: The generalization from $\mathit{MaxConcurrent} = 1$ to $\mathit{MaxConcurrent} = k$ is not merely a parameter change but an algebraic generalization: the singleton holder becomes a bounded set, the availability check becomes a cardinality comparison, and release becomes set difference. That all safety and liveness properties are preserved by this generalization suggests a deeper algebraic structure (semiring-like) in the concurrency model. Exploring this structure could enable automated generalization of other TLA+ specifications.

e) Conformance testing as pragmatic refinement.: A formal refinement mapping (proving every implementation trace is a valid specification trace) would provide the strongest guarantee. We adopt the pragmatic approach described by AWS [9]: conformance testing through deterministic guard/effect tests (S1–S8) and randomized trace validation (500,000 property-based transitions). The two-layer approach—specification-level exhaustive verification plus implementation-level randomized testing—provides stronger guarantees than either technique alone.

f) Limitations.: AGENTGUARD targets single-host, multi-session coordination (§I-B); multi-host deployments would require distributed consensus, which is orthogonal to our contribution. The TLA+ model assumes instantaneous transitions;

timing properties (latency bounds, throughput) are validated empirically rather than formally. The specification prevents *resource-level* conflicts (two sessions writing the same file) but not *semantic-level* conflicts (two sessions editing different functions in the same file that interact logically). The bug classification (§V-C) is limited to publicly filed issues with sufficient detail; closed-source runtimes may exhibit additional failure modes.

VII. Conclusion

Concurrent LLM agent sessions present a new class of resource-coordination challenges that existing runtimes handle ad hoc. Our analysis of 43 production bug reports confirmed that the resulting failures—duplicate execution, deadlocks, and lost writes—are systematic and recur across independent projects.

AGENTGUARD addresses these failures through a composable TLA+ specification hierarchy (937 LOC) whose safety and liveness properties are exhaustively verified by TLC across $>$1M states. The hierarchy demonstrates that mutual exclusion, counted semaphores, and multi-resource transactions are points on a generalization continuum, each preserving all prior properties. Two TLC counterexamples directly shaped the design: preemption must count toward the retry limit to prevent liveness violations, and strong fairness is necessary for scheduling actions.

A conforming Rust daemon ($\sim$4,700 LOC) deploys the verified specification transparently via tool-call hooks, preventing all three bug classes with sub-millisecond overhead across 1–50 concurrent sessions. The 4-layer verification chain (TLC $\rightarrow$ Kani $\rightarrow$ proptest $\rightarrow$ 146 tests) bridges the gap from abstract specification to production implementation.

Future work includes: (1) *inductive invariants* for unbounded verification beyond the small-scope hypothesis; (2) *Apalache symbolic checking* for larger bounded configurations; (3) *formal refinement mapping* between the TLA+ specification and the Rust implementation; and (4) extension to *semantic-level conflicts* (e.g., detecting that two agents are editing the same function, not just the same file).

References

[1] OpenClaw Contributors, "OpenClaw: Open-source personal AI assistant runtime," GitHub repository, 2024, https://github.com/openclaw/openclaw. 227k+ stars. Issues #18760, #20740, #13508, #18470 document scheduling and concurrency bugs.

[2] NanoClaw Contributors, "NanoClaw: Lightweight AI agent runtime with container isolation," GitHub repository, 2024, https://github.com/qwibitai/nanoclaw. 14k+ stars. Issues #138, #300, #293, #358 document scheduling races and deadlocks.

[3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[4] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.

[5] Anthropic, "Claude computer use and Claude Code," Product documentation, 2024, https://docs.anthropic.com.

[6] H. Chase, "LangChain," 2022, https://github.com/langchain-ai/langchain.

[7] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[8] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *International Conference on Learning Representations (ICLR)*, 2024.

[9] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[10] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in MongoDB," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2071–2081, 2019.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference (ATC)*, 2014, pp. 305–319.

[12] Microsoft Research, "Formal verification of Azure Cosmos DB with Dafny," Technical report, 2024, https://github.com/dafny-lang/dafny.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *OSDI*, 2008, pp. 267–280.

[14] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 167–178.

[15] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," *ACM SIGPLAN Notices*, vol. 46, no. 4, pp. 53–64, 2011.

[16] D. R. MacIver and Z. Hatfield-Dodds, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[17] J. a. Moura, "CrewAI: Framework for orchestrating role-playing, autonomous AI agents," GitHub repository, 2024, https://github.com/crewAIInc/crewAI.

[18] OpenAI, "Swarm: Educational framework for lightweight multi-agent orchestration," GitHub repository, 2024, https://github.com/openai/swarm.

[19] K. Mei, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang, "AIOS: LLM agent operating system," in *Conference on Language Modeling (COLM)*, 2025.

[20] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang *et al.*, "AgentBench: Evaluating LLMs as agents," in *International Conference on Learning Representations (ICLR)*, 2024.

[21] M. Kleppmann, "How to do distributed locking," *martin.kleppmann.com*, 2016, https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html.

[22] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.

[23] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003, pp. 44–60.

[24] Kubernetes Authors, "Pod lifecycle," https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, 2024, accessed 2026-03-07.

[25] L. Lamport, "Specifying systems: The TLA+ language and tools for hardware and software engineers," in *Addison-Wesley Professional*, 2002.

[26] Ollama, "Ollama: Run large language models locally," Open-source project, 2024, https://ollama.com.