ChatGPT

LLMC Multi-Agent Coordination Protocol Design

Executive Summary

LLMC's development environment currently relies on ad-hoc, manual coordination to prevent agents from **"stomping"** on each other's work. With 6–8 parallel tmux sessions running agents (Beatrice on OpenAI Codex, Otto on Anthropic Claude, Rem on Google Gemini), collisions occur in ~5% of parallel operations, leading to lost work and inconsistent state. This document formalizes an **Anti-Stomp Ticket Protocol** – a file-based locking and messaging system – to **eliminate file conflicts, detect/resolve deadlocks**, and **dynamically route tasks** to the best-suited agent. The design borrows concepts from distributed lock services (e.g. Google's Chubby uses whole-file locks [1]) but operates entirely on the local filesystem (no external servers). Key improvements include:

- **Ticket-based locking** with **TTL (time-to-live)** and ownership metadata, preventing simultaneous writes to the same file.
- **Deadlock detection** via wait-for graphs (cycle detection) and **priority-based preemption** to break circular waits.
- **Agent specialization & routing**, assigning tasks to agents based on their strengths (e.g. Beatrice for new code, Otto for analysis) and current workload.
- **Message passing and heartbeats**, giving visibility into each agent's status and enabling safe task hand-off or failover if an agent is busy or unresponsive.
- **Failure recovery** mechanisms such as stale lock reaping, task timeouts with automatic rollback, and retries or rerouting on crashes.

By implementing this protocol, LLMC can **scale to 8+ concurrent agents** with **zero file conflicts**, automated deadlock resolution within seconds, and significantly reduced manual intervention. Agents will coordinate through a shared `.contract/` directory, transparently acquire and release file locks via wrappers, and report status to a central queue and dashboard. This document outlines the full specification, algorithms, and integration guide for deploying the coordination system within LLMC's existing `codex_wrap.sh` workflow. Success will be measured by near-elimination of collisions (<0.1% tasks conflict), timely deadlock recovery, and improved parallel throughput without human orchestration.

Ticket Protocol Specification

Ticket Files are the core of the locking mechanism. A ticket is a small JSON file stored under `.contract/tickets/` that represents a lock (or "work ticket") on a resource (usually a file). The ticket records which agent holds the lock, the type of operation, and timing information including expiration. This acts similar to an advisory lock file seen in distributed systems [1], but enriched with metadata to support coordination. Each ticket file is named by a unique ID (e.g. `uuid.lock` or `ticket_{id}.json`).

Ticket File Format: Every ticket contains fields as follows:

```
{
  "ticket_id": "uuid-v4",
  "resource": "/path/to/file.py",
  "operation": "read|write|exclusive",
  "owner": "beatrice|otto|rem",
  "pid": 12345,
  "acquired_at": "2025-11-03T10:30:00Z",
  "expires_at": "2025-11-03T10:35:00Z",
  "priority": 5,
  "metadata": {
    "task": "enrich span sha256:abc...",
    "session": "tmux-session-name"
  }
}
```

- `resource`: The file or resource being locked. For write or exclusive operations, only one agent can hold a ticket on that resource at a time. Read operations may allow multiple concurrent tickets if only reading.
- `owner`: Which agent holds the ticket (Beatrice, Otto, or Rem). The `pid` helps detect if the process is still alive.
- `operation`: Lock type: `"read"` (shared lock), `"write"` (exclusive for writing), or `"exclusive"` (for actions like renaming, where no other access is allowed). By default, write locks are exclusive.
- `priority`: An integer indicating task priority or agent priority. Higher priority tasks can preempt lower ones if necessary (used in deadlock resolution).

Ticket Lifecycle: Locks transition through defined states with automatic expiration to avoid orphaned locks:

1. **Acquire:** When an agent needs to modify or read a file, it first requests a ticket. This is done by creating the ticket file in an **atomic operation** (to avoid race conditions, e.g., using `O_CREAT|O_EXCL` or a temporary file rename). If a conflicting ticket already exists, the agent will wait or be queued.
2. **Hold:** Once acquired, the agent proceeds with its operation. The ticket serves as a claim on the resource. The agent should periodically refresh or "heartbeat" the ticket (updating a timestamp or extending `expires_at`) if the operation is long-running.
3. **Renew:** If the task is not finished before the initial `expires_at`, the agent can extend the lock by updating the ticket's `expires_at` (assuming no higher-priority preemption has been signaled). This prevents false expiration while still enabling a safety net for crashes.
4. **Release:** When the operation completes, the agent **deletes its ticket file**, freeing the resource for others. This is done in a finally/cleanup block even if the task fails, to ensure locks don't linger.
5. **Expire:** If an agent dies or fails to release a lock, the ticket's TTL comes into play. A background **reaper** process will check for tickets past `expires_at` and remove them if the owning process `pid` is no longer running. This **auto-releases abandoned locks** after a timeout.
6. **Preempt:** In rare cases (deadlocks or high-priority tasks), an agent can forcibly take over a resource. The protocol allows an agent with a higher `priority` to preempt a lower-priority ticket: the higher-priority agent creates a preemption request (or simply acquires a new ticket with the same

resource, which signals the conflict). The lower-priority agent, upon detecting this (via a signal or by the ticket manager), must abort or pause, release its lock, and let the higher-priority task proceed. Preemption is a last resort to break deadlocks or handle urgent tasks.

Lock Compatibility: The protocol defines which ticket types can coexist:

- **Read–Read:** Allowed concurrently. Multiple agents can hold read locks on the same file (the ticket manager will grant multiple read tickets simultaneously).
- **Read–Write:** Not allowed concurrently. A write ticket request on a file will wait until all read tickets are released. Conversely, new read requests will wait if a write lock is active.
- **Write–Write:** Strictly exclusive. Only one write ticket can exist for a given resource. A second writer must wait or trigger conflict resolution.
- **Exclusive:** If a resource is locked in exclusive mode (for example, during a refactor moving or renaming files), no other tickets of any kind are allowed on that resource.

Conflict Scenarios: The ticket manager (a Python module) handles collision logic. If Agent A holds a write lock on `foo.py` and Agent B requests a write on the same file, B's request is queued or blocked until A releases. If a deadlock arises (A waiting on a resource locked by B, and B simultaneously waiting on something locked by A), the system detects the cycle and invokes resolution (see next section). The goal is to **prevent file edit collisions entirely** – similar multi-agent frameworks implement such lock files to avoid parallel edits.

Each ticket includes **ownership metadata** for traceability. For instance, `metadata.task` might contain a human-friendly description or a hash of the task request, and `metadata.session` records which tmux session (or agent instance) created the lock. This makes it easier to debug who is doing what by simply inspecting the ticket files. The `.contract/tickets` directory effectively becomes an **active work registry**, listing all resources currently in use by agents – a design seen in other agent coordination systems. Tools or dashboards can read this directory to display current locks and activities.

Deadlock Prevention & Detection

Even with proper locking, **deadlocks** can occur if agents wait on each other in a circular chain. For example, Beatrice holds a lock on `fileA` and needs `fileB`, while Otto holds `fileB` and needs `fileA`. To address this, the protocol incorporates both prevention and detection mechanisms:

- **Ordering & Prevention:** Where possible, agents should acquire multiple needed resources in a consistent global order to avoid circular waits (similar to how databases avoid deadlocks by ordering lock acquisition). In practice, tasks often involve one file at a time, but if an agent needs to lock multiple files, the ticket manager can enforce an ordering (e.g. alphabetical by filename or some hash order) to minimize risk of cycles.
- **Wait-For Graph & Detection:** The coordination system maintains a **wait-for graph** of agents and the locks they're waiting on. Each agent is a node; an edge from Agent A to Agent B means *A is blocked waiting for a resource currently held by B*. A cycle in this directed graph indicates a deadlock condition. The `deadlock_detector.py` component will periodically (e.g. every few seconds) analyze active tickets and any pending lock requests to construct this graph. If a cycle is found, it signals a deadlock.

- **Deadlock Resolution:** Upon detecting a deadlock cycle, the system automatically resolves it using **priority-based preemption** or timeouts. The simplest strategy is to choose the **lowest-priority** agent in the cycle and force it to relinquish its lock (preempt its ticket). For example, if Rem (priority 3) and Beatrice (priority 5) are deadlocked, the system would instruct Rem to abort its task and release locks so Beatrice can proceed. In the absence of priority differences, a simple heuristic is to pick the agent that has done the least work or can most easily restart (perhaps the one with the shortest runtime so far or the one designated as preemptible).
- **TTL Timeout as Fallback:** The TTL on tickets provides a safety net: if a deadlock is not resolved by preemption (e.g. because all tasks are equal priority and nothing happens), eventually one or more tickets will **expire** and be cleaned up by the reaper. In effect, this breaks the deadlock, albeit in a less controlled way. The TTLs should be set such that deadlocks don't persist for long (e.g. a lock times out after a minute of no progress unless renewed). The goal is to detect and resolve deadlocks within 60 seconds or less, to keep agents moving.

Deadlock Detection Algorithm: Pseudocode for cycle detection using a wait-for graph might look like:

```
# Pseudocode for deadlock detection
graph = {}
# e.g., {'beatrice': ['otto'], 'otto': ['beatrice']} means Bea waits for Otto,
Otto waits for Bea
for ticket in active_tickets:
    if ticket.requesting_agent is blocked:
        blocker = ticket.owner  # who holds the resource
        waiter = ticket.requesting_agent
        graph[waiter].append(blocker)

cycle = find_cycle(graph)
if cycle:
    victim = choose_lowest_priority(cycle)
    preempt(victim)
```

This algorithm would be part of `deadlock_detector.py`. It scans all current lock requests (agents that are waiting) and builds edges from waiting agent -> holding agent. A depth-first search (DFS) or union-find can detect cycles. When a cycle is found, the system logs it (for debugging) and **initiates resolution**: marking one ticket as preempted. Preemption could be implemented by adding a field in the ticket (e.g. `"preempted_by": "<agent>"`) or by simply deleting the low-priority agent's ticket and putting its task back in the queue for later retry. The affected agent should be notified (via its status file or a signal) that it lost its lock due to deadlock, so it can gracefully back out (e.g. revert partial changes) and retry later.

Priority and Aging: The protocol allows each task or agent to have a priority. This not only helps with preemption decisions but also with scheduling (urgent tasks first). However, to prevent starvation (a low-priority agent never getting a chance because higher ones always preempt it), the system can implement **priority aging** – if a task has waited for a long time, effectively bump its priority gradually. This ensures fairness over time.

In summary, deadlocks are handled by *avoidance where possible, detection when needed, and automatic resolution*. The design aims for a transparent recovery: ideally agents won't even need human intervention to recover from a deadlock; the system will sort it out within a minute by aborting one of the tasks, logging the event, and continuing.

Agent Specialization and Task Routing

LLMC's three agents each have unique strengths, so the coordination system will intelligently route tasks to the most appropriate agent, improving efficiency and outcomes. We formalize an **Agent Specialization Matrix** and a routing algorithm to dispatch tasks based on task type, complexity, and agent availability.

Agent Roles:

- **Beatrice (OpenAI Codex):** Excels at generating new code and writing tests. Fast and creative in code synthesis, making her the primary choice for net-new code generation tasks.
- **Otto (Anthropic Claude via Desktop Commander):** Excels at understanding context, performing code review, refactoring, and writing documentation. Claude's larger context window and analytical ability make Otto ideal for analyzing or improving existing code (reviews, refactors, documentation).
- **Rem (Google Gemini API):** Specialized in API integration tasks and data retrieval. Rem can call external APIs or handle tasks that involve web queries and integration, and is used when online knowledge or Google's capabilities are needed. Rem is also effective for supplementary coding tasks when others are busy.

Using these characteristics, we define a specialization matrix for common task types:

Task Type	Primary Agent	Secondary	Tertiary	Rationale
Code Generation (new feature or file)	Beatrice	Otto	Rem	Beatrice's code synthesis is strongest (OpenAI Codex), producing high-quality new code quickly. Otto can assist if Beatrice is overloaded, and Rem can handle simpler generation if needed.
Code Review / Refactor (improve existing code)	Otto	Beatrice	Rem	Otto (Claude) is best at analyzing and explaining code and suggesting improvements. Beatrice can handle straightforward refactors, and Rem can do minor tweaks if others are busy.
API Integration (calling external APIs or handling web data)	Rem	Otto	Beatrice	Rem (Gemini) is designed for external integration tasks (especially Google-related APIs). Otto can serve as backup given Claude's general abilities, and Beatrice can implement if it's mainly coding wrappers.

Task Type	Primary Agent	Secondary	Tertiary	Rationale
Documentation & Comments (writing explanations, README updates)	Otto	Beatrice	Rem	Otto's strength in natural language makes him ideal for documentation. Beatrice can generate technical content as well, and Rem can supply info if needed.
Testing (writing tests or verifying behavior)	Beatrice	Rem	Otto	Beatrice (Codex) is very good at generating unit tests and test code. Rem can generate tests (especially if they involve external data or integration tests), and Otto can assist especially in understanding test requirements.
Bash/Automation Scripts (shell scripts, CI tasks)	Otto	Beatrice	Rem	Otto (Claude via Desktop Commander) has direct shell execution and can write bash scripts effectively. Beatrice can code scripts too, and Rem can fill in for simpler scripts or when leveraging online info.

Dynamic Task Routing: The system will include an **Agent Router** (`agent_router.py`) that uses the above matrix along with real-time agent status to assign tasks from the central queue to agents. The routing logic works as follows:

1. **Task Categorization:** Each incoming task (from a user or generated by a pipeline) is categorized by type and complexity. This could be done via simple pattern matching or by an AI-based classifier. For example, tasks containing keywords like "build" or "create" likely involve **Code Generation**, whereas "refactor" or "optimize" indicate **Refactor/Review**.
2. **Primary Agent Selection:** Based on the category, the router picks the primary agent (from the matrix). E.g., for a "generate new module" task, choose Beatrice first. It then checks the agent's **status** to see if they are available (idle) or lightly loaded. If the primary agent is free or has capacity, the task is assigned to them.
3. **Load Balancing & Secondary Agents:** If the primary agent is **busy or at capacity**, the router considers the secondary agent. Workload is determined from the agent's status file (which includes a `queue_depth` and `avg_task_duration_sec`). For instance, if Beatrice currently has a queue of several tasks, and Otto is idle, the router might assign a new code generation task to Otto (even though he's secondary) to avoid delays. This dynamic failover ensures no agent becomes a bottleneck – similar to how a multi-threaded system might offload work if one thread is saturated.
4. **Tertiary/Fallback:** In rare cases (primary and secondary both overloaded or unavailable), the tertiary agent can take the task. This ensures **no task is dropped**; all tasks will eventually be picked up by someone, albeit potentially less optimally.
5. **Queuing if All Busy:** If all suitable agents are busy and the task is not urgent, it remains in the **task queue** until an agent frees up. The queue is prioritized by task priority and timestamp. The routing loop will periodically re-check if an agent can take the next high-priority task.

Under the hood, the **task queue** is implemented as a set of JSON files in `.contract/queue/`, each representing a pending task (with details like task type, description, priority, etc.). Agents (or the central router) perform an atomic claim on a task file when they begin working on it – e.g., renaming it to move into a `.contract/tickets` or `.contract/active` directory, or updating its status to "in-progress" with an assigned agent. This ensures that two agents don't accidentally grab the same task. Because tasks have priorities, the claiming mechanism should always pick the highest priority task that an agent is eligible for.

Capability Declarations: Each agent can declare its capabilities and preferences, possibly in a config file (or as part of its status file). For example, Beatrice's config might list `["code_gen", "testing", "doc"]` as strengths with a certain efficiency rating. The router can use this information to make more nuanced decisions – e.g., even if Beatrice is slightly busy, a high-priority code-gen task might still go to her because she's much faster at it than Otto. These declarations could be static (based on our matrix) or dynamic (based on past performance metrics).

In summary, task routing ensures that **"the right tool for the job"** is chosen, improving quality and speed. It also provides **load balancing**: if one agent is unavailable (e.g., Rem's API key is failing or Beatrice hit a rate limit), the system automatically routes tasks to others. This increases the robustness of the multi-agent system and reduces idle time.

Message Passing & Coordination

For effective coordination, agents need to communicate their status and receive signals about global events (like "stop, you've been preempted" or "new task available"). Instead of direct inter-process communication (which could be complex across tmux sessions), LLMC will use a **file-based message passing** approach. Each agent writes to a **status file** and monitors a central **task queue**, as well as its own message inbox for any direct signals.

Agent Status Files: Located in `.contract/status/{agent}.json`, these files provide a live snapshot of each agent's state. The format is:

```
{
  "agent": "beatrice",
  "state": "idle" | "working" | "blocked" | "crashed",
  "current_task": {
    "ticket_id": "uuid-1234",
    "started_at": "2025-11-03T10:30:00Z",
    "estimated_completion": "2025-11-03T10:35:00Z"
  },
  "workload": {
    "queue_depth": 3,
    "avg_task_duration_sec": 45
  },
  "last_heartbeat": "2025-11-03T10:31:00Z"
}
```

Fields:
- `state`: High-level status of the agent. `idle` means ready for work; `working` means it's actively performing a task; `blocked` means it's waiting on a lock or resource; `crashed` indicates the agent has unexpectedly stopped or is unresponsive.
- `current_task`: Info about the task in progress (if any). It references the `ticket_id` (so one can cross-reference which ticket file corresponds to this work) and a start time. The `estimated_completion` may be set by the agent or router to help timeouts (e.g., based on typical durations or explicit deadlines).
- `workload`: A summary of the agent's queued work. `queue_depth` is how many tasks are waiting in its personal queue (if the router assigns multiple tasks or if the agent pulled several from the central queue). `avg_task_duration_sec` could be a rolling average of how long tasks take – useful for the router to estimate load.
- `last_heartbeat`: A timestamp of the last heartbeat update from this agent. Agents should update their status file periodically (e.g. every 30 seconds or after finishing each step) to indicate liveness. If this timestamp is stale (no update for a while), the agent might be hung or crashed, and recovery can begin.

These status files serve as a **lightweight bulletin board** for agent health. The coordination system or a monitoring tool can read them to display a dashboard of all agents. They also allow agents to know about each other indirectly. For example, if Beatrice is about to take on a big task, the router or Beatrice's process can check if Otto is idle and perhaps offload something to him.

Task Queue: All new tasks (from user requests or system triggers) go into `.contract/queue/` as individual JSON files (e.g., `task_123.json`). A task file contains details like:

```
{
  "task_id": "task-123",
  "description": "Add email validation to signup form",
  "type": "code_gen",
  "priority": 5,
  "created_at": "...",
  "requirements": {...}
}
```

Additional fields can include hints or requirements (like "needs internet" or "affects files: X, Y" which the router can use to decide on the agent). Agents (or the central `queue_manager.py`) watch this directory for tasks.

Task Claiming Protocol: When an agent is free, it will attempt to **claim** a task from the queue:
- The agent (or router) picks the highest-priority task file that matches an agent's capabilities.
- To claim, it performs an atomic move or rename of the task file, e.g., rename `queue/task_123.json` to `queue/claiming/task_123.{agent}.json` or simply deletes it from the queue and notes the assignment in the agent's status. This should be done in a way that if two agents race to claim, only one succeeds (file system atomic operations can guarantee this if done carefully). Alternatively, the `queue_manager.py` could mediate claims using a lock.
- Once claimed, the agent updates its `status` file to `working` on that task and creates the necessary ticket locks for any target files the task will modify.

Inter-Agent Signaling: While the design avoids direct sockets or network messaging, we can emulate signaling via files:
- If an agent needs to notify another (e.g., preemption or asking to release a lock), it can write a small **signal file** under `.contract/signals/`. For example, if Otto needs Beatrice to release a lock (in a deadlock), the system might create `.contract/signals/beatrice_release_{ticket_id}`. Beatrice's process (or the ticket manager) would periodically check for any signals addressed to it. Alternatively, the deadlock resolver could directly modify Beatrice's ticket file (marking it preempted) which Beatrice will notice on next action.
- Agents can also communicate completion or failure of tasks via the **completed work log** or by updating their status. For instance, when an agent finishes a task, it could append an entry to `.contract/logs/completed_tasks.log` or set `state` to idle and clear `current_task`.

Heartbeat and Health Checks: Each agent is responsible for writing to its status file at a regular interval (heartbeat). A small watchdog could be implemented in `ticket_manager.py` or a separate daemon to monitor these heartbeats. If an agent's heartbeat timestamp goes stale beyond a threshold (e.g., no update for 2 minutes), the coordinator flags that agent as possibly crashed (`state: crashed`). This can trigger **failure recovery** (the reaper will release that agent's locks, and tasks in progress will be re-queued).

By using the filesystem for coordination, we ensure that even if the higher-level logic fails, a human can inspect these files to understand the system state, and no complex networking is needed. This approach is similar to other multi-agent systems where a directory of JSON files acts as the coordination hub. The downside (slight latency in polling) is acceptable given our environment (a local machine with likely sub-second file I/O). The benefit is simplicity and transparency: every significant event (ticket acquire, release, new task, agent heartbeat) is recorded as a file update, creating an audit trail in `.contract/logs/` for debugging.

Failure Recovery

A robust coordination protocol must gracefully handle failures and unexpected events: agent crashes, network outages for API calls, partial results, etc. LLMC's anti-stomp system includes several **failure recovery** strategies to ensure the system can self-heal or at least fail safe:

1. Agent Crash or Freeze: If an agent process crashes (e.g., segmentation fault, out-of-memory, or the tmux session is killed) while holding tickets, those locks risk remaining indefinitely (blocking other agents). To handle this:
- A **Reaper** process (`reaper.py`) runs periodically to scan all active tickets. For each ticket, it checks if the `pid` recorded is still an active process on the system. If not, it means the agent died without releasing the lock. The reaper will log a warning and delete that ticket file, effectively freeing the resource. This is analogous to how distributed systems use ephemeral locks that disappear when a client disconnects, or how stale lock files are cleaned on reboot.
- Additionally, if an agent hasn't updated its heartbeat (`last_heartbeat`) in a while, the reaper may consider it deadlocked or hung. In that case, it can mark the agent as `crashed` and proceed to release *all* locks owned by that agent (since we assume it's not actually working on them anymore). This is somewhat drastic but prevents a wedged agent from halting the entire system.
- Example: Suppose Beatrice's process was terminated mid-task. She might have a ticket on `CONTRACTS.md` that others are waiting for. The reaper sees `beatrice` ticket with `pid` that no longer exists; it removes the ticket and writes an event to the log. Now Otto and others can proceed to lock

`CONTRACTS.md` as needed. A **stale lock detection** like this (here TTL + pid check) was also identified as crucial in similar multi-agent setups.

2. Task Timeout: Sometimes an agent might hang or take far too long on a task (maybe the LLM got stuck or is in a loop). Each ticket has an `expires_at` and each task an `estimated_completion`. If those times are exceeded:
- The ticket automatically expires (any other agent waiting can proceed as if the lock is free – though they should ideally confirm via the reaper or a double-check that the original process is indeed not actively writing).
- The task can be returned to the queue. For instance, if Rem was calling an external API and it hasn't responded for 5 minutes, the system can assume failure: Rem's wrapper will abort the API call, release any partial locks, and put the task back into `.contract/queue/` (perhaps with a note that it failed once). Then maybe another agent or a retry later can attempt it.
- The coordination system could also increment a retry count in the task file. If a task fails due to timeout too many times, it might be marked as errored and require human attention. But for transient issues, automatic retry is valuable.

3. Partial Work and Rollback: A tricky scenario is when an agent partially completes a task – e.g., it wrote half of a file or made some changes and then crashed or was preempted. This could leave the codebase in a broken state if not handled. Solutions:
- Use **atomic file operations** for writes. As much as possible, agents should **write to a temp file and then rename** to the final name only when complete. This way, other agents never see a half-written file. For example, Rem's wrapper can write the API response to `temp.json` then move to `output.json` once fully retrieved, to avoid Otto reading an incomplete file (a scenario that has caused failures before). This is already a known workaround being used manually in some cases.
- Implement a **git-based checkpoint/rollback** for larger changes. Before an agent starts a major task, the system can create a git worktree or a new branch as a safety checkpoint. If the agent crashes mid-change or the task is preempted, we can easily revert the repository to the last good commit (or have the next agent restart from that point). This ensures partial edits do not corrupt the main branch. The design can integrate with version control – e.g., every task runs on a "ticket branch", and only when finished and verified, it's merged to main. If aborted, the branch is discarded. This way, the main codebase remains stable.
- Alternatively, maintain a **shadow copy** of any file an agent is editing (like a .bak file). If the agent doesn't finish, the shadow can be used to restore the original content.

4. Network or External Failures (specific to Rem/Gemini): Rem relies on an API (Google's Gemini). If the network is down or the API fails (timeouts, rate limits), Rem's tasks could fail frequently. The coordination system should detect this and **reroute tasks** if needed:
- If Rem reports via its status or an error file that it cannot reach the API, the router can stop assigning new API tasks to Rem and instead give them to another agent if possible (e.g., perhaps Beatrice or Otto can attempt it, or queue them until the network is back).
- Implement **exponential backoff retries** for transient errors. For instance, if an API call fails, Rem's wrapper script (`gemini_wrap.sh`) can wait a few seconds and try again a couple times. During this time, it should keep its lock (or extend the TTL) if it still intends to write output. If ultimately unsuccessful, it should clean up as described (release locks, mark task for retry later).
- If an agent like Rem becomes completely unavailable (say API key expired), this should be flagged in its status (state = crashed or error). The system operator (or an automated script) can disable Rem from the pool until fixed, and the router will not assign tasks to it.

5. Log and Continue: In all failure cases, the philosophy is to **fail gracefully**: log what happened, release resources, and continue operating. The `.contract/logs/` directory will contain logs like `coordination_events.log` where events such as "Deadlock detected between Otto and Rem, preempting Rem's lock on X" or "Beatrice heartbeat missed, assuming crash – releasing 2 locks" are recorded with timestamps. These logs are invaluable for debugging and post-mortem analysis.

6. Orphaned Session Cleanup: Since tmux sessions can be ephemeral, the system should clean up after tasks that spawn their own sessions. If an ephemeral tmux session (like a one-off invocation of `codex_wrap.sh`) finishes normally, it should exit and release locks. If it hangs around (perhaps waiting on input or stuck), a timeout can kill it. The reaper or a specialized script can list tmux sessions and their idle time – if a session named e.g. `dc-beatrice-1234` has been idle for over 2 hours with no heartbeat, it's safe to terminate that session. This prevents accumulation of ghost processes.

By covering these scenarios, the coordination protocol ensures that even when things go wrong, they don't spiral into persistent issues. **Stale locks** are cleared, **tasks are not lost** (they either complete or go back to queue), and the system can continue running close to 24/7. The manual burden on the developer (previously needed to check for collisions or hung agents) will drop dramatically – perhaps by 50% or more – as the system now handles these routine failure modes automatically.

Implementation Guide & Integration

Integrating the new coordination protocol into LLMC's existing workflow involves augmenting the wrapper scripts and providing the coordination tools in the `scripts/coordination/` module. The solution is designed to be **backwards-compatible** and incrementally adoptable. Below is the proposed code structure and the key integration points in the current system:

```
scripts/coordination/
├── __init__.py
├── ticket_manager.py       # Core ticket acquire/renew/release logic
├── agent_router.py         # Task routing logic (assign tasks to agents)
├── deadlock_detector.py    # Cycle detection and resolution
├── reaper.py               # Cleanup stale tickets, monitor heartbeats
├── queue_manager.py        # Task queue operations (enqueue, claim tasks)
└── cli.py                  # Command-line interface for manual ops (optional)

.contract/
├── tickets/                # Active ticket files (one per lock)
├── status/                 # Agent status files (one per agent)
├── queue/                  # Incoming task files
├── signals/                # Signal files for inter-agent messaging (if used)
└── logs/                   # Coordination event logs and audit trail
```

Wrapper Script Modifications: The existing `codex_wrap.sh` and similar scripts (`gemini_wrap.sh`, Desktop Commander calls for Otto) will be modified to use the ticket system transparently. For example, in `codex_wrap.sh`, before the script invokes the Codex API or writes files, it should:

```
# Pseudocode injection at start of codex_wrap.sh
resource="$TARGET_FILE"  # the main file the agent will edit, determined from
input or context
python3 scripts/coordination/ticket_manager.py acquire --resource "$resource" --
agent "beatrice" --operation write --priority 5
# If acquire fails (blocked), the script can either wait or exit; for
simplicity, maybe wait with timeout
...
# [Proceed with usual Codex operations]
...
# Upon completion (or error):
python3 scripts/coordination/ticket_manager.py release --resource "$resource" --
agent "beatrice"
```

We can wrap these calls in shell functions for convenience, as hinted in the design notes. For instance:

```
acquire_ticket() {
    python3 scripts/coordination/ticket_manager.py acquire "$@"
}
release_ticket() {
    python3 scripts/coordination/ticket_manager.py release "$@"
}
# Usage:
acquire_ticket --resource "foo.py" --agent "beatrice" --operation write --
priority 5
# ... do work ...
release_ticket --resource "foo.py" --agent "beatrice"
```

The `ticket_manager.py` when invoked with "acquire" will attempt to create a ticket file in `.contract/tickets`. It will either succeed immediately (and perhaps print or return a ticket ID) or block/retry until the lock is acquired (or a timeout triggers). In blocking mode, the shell script can just wait; alternatively, we could implement it to return an error code if it couldn't get the lock in a reasonable time. For the first version, a simple approach is fine: just wait. This ensures the agent doesn't start editing until it has the lock. Similarly, on release, the script will delete the corresponding ticket file and possibly signal any waiting processes (e.g., via a notify file or simply others will notice the file gone).

Gemini (Rem) Wrapper: `gemini_wrap.sh` should also call `ticket_manager.py acquire` for any file it will output to. For example, if Rem will write to `output.json`, acquire a write lock on that file. Additionally, because Rem's outputs might be read by others (scenario where Otto reads a JSON that Rem produced), Rem could use a temp file as described (write to temp, then acquire lock on final file, move temp to final, release lock). The coordination overhead for Rem also includes marking when it's making an API call (perhaps setting its status to `blocked` because it's waiting on external input).

Desktop Commander (Otto) Integration: Otto operates by receiving commands (like "open file X and insert Y"). We need to ensure that when Otto tries to modify a file through the MCP (Managed Command

Proxy), it respects locks. Ideally, the Desktop Commander layer can be made aware: before performing a file write, it should check `.contract/tickets` for an existing lock on that file. This could be done by calling a small check in `ticket_manager.py` (like a `can_write(path)` function). If locked, Otto's agent should refrain from writing (perhaps queue that operation or wait). However, since Otto's actions are driven by Claude's reasoning, a simpler approach is to **have the codex/Claude orchestrator also use the same `codex_wrap.sh` pipeline**. In the architecture described in **AGENTS.md**, Claude (Otto) actually calls `codex_wrap.sh --local` or similar to perform tasks [2]. If we keep that design, then Otto is indirectly going through the same path as Beatrice, meaning the lock acquisition happens in the shared wrapper. Essentially, unify how all agents invoke the edit operations via a common path that includes locking.

Tmux Session Management: We aim for **long-lived agent sessions** (to maintain state and continuously update heartbeat). For example, keep a persistent `tmux` window for Beatrice that constantly runs a loop: check for tasks, do tasks, update status. This way, the coordination is easier (the agent process is always there to receive signals and maintain heartbeats). Ephemeral sessions (which start for one task and terminate) are still supported, but they should be treated as short-lived "agent instances" that create a status file, do work, and then set state to idle or remove their status on exit. We will implement a way to remove an agent's status file when it fully exits to avoid confusion (or mark it as terminated).

For long-lived sessions, the `codex_wrap.sh` can be run in a loop mode. Alternatively, we write a small supervisor script that runs inside tmux for each agent: `agent_main.py` that polls the queue and executes tasks by invoking the underlying commands. That might be future enhancement; initially, even if each task launches a new process, our locking will still work – just we rely on TTL to clean up if something dies unexpectedly.

Integration with Version Control and Logs: We recommend integrating with `git` to assist rollback. For example, `ticket_manager.py acquire` could create a lightweight git branch named after the ticket (e.g. `ticket-<id>`). The agent will work on that branch. On release, if everything is good, changes can be merged to main (or main is fast-forwarded). If a crash or preemption occurs, the branch can be left for analysis or deleted. This is an optional integration but adds an extra safety layer. At minimum, instruct agents to commit their changes when a task is done (perhaps to a local dev branch) so nothing is lost and conflicts can be resolved via git merge if two tasks did end up overlapping.

Backward Compatibility: The new system can run side-by-side with manual methods at first. For example, if `ticket_manager.py` fails or isn't used, things operate as before (with risk of conflict, but nothing breaks). We can start by enabling it for the most conflict-prone operations (like editing the shared `CONTRACTS.md` file or the RAG index) to prove its value. Over time, all agent actions will funnel through the ticketing. The performance overhead is minimal – writing a small JSON file and deleting it – likely on the order of milliseconds to tens of milliseconds, which is negligible compared to LLM execution times.

Example Workflow (with integration):
1. Developer or Claude issues a command that triggers a coding task (e.g., "Implement feature X").
2. The task is added to `.contract/queue/task123.json` with type "code_gen".
3. Beatrice's session (idle) sees a new task. The router (could be part of Beatrice's loop or a central process) assigns it to Beatrice (primary for code_gen).
4. Beatrice's `ticket_manager.py` is called to acquire locks on files she's going to modify (say `foo.py`). A ticket `ticket_ABC.json` is created in `.contract/tickets` with owner=Beatrice, resource=`foo.py`.

5. Beatrice proceeds to generate code and write to `foo.py`. Suppose while she's working, Otto gets a task to refactor `foo.py` (bad luck, overlapping). Otto's attempt to lock `foo.py` via the same process will see an existing ticket by Beatrice and will block. Otto's status becomes `blocked` on that resource. The deadlock detector sees no cycle (just one waiting), so it does nothing.
6. Beatrice finishes in 30 seconds and releases the lock (`ticket_manager.py release` deletes `ticket_ABC.json`). It updates her status to idle or ready for next task.
7. The moment the lock file is gone, Otto's waiting process acquires the lock (unblocked), and Otto proceeds with the refactor on `foo.py`. No conflict occurred – Otto is working with Beatrice's final code. If Beatrice had partially edited the file when Otto started, those edits were either complete (thanks to atomic save) or if preempted, would have been reverted, ensuring Otto sees a consistent file.
8. Throughout, each agent wrote heartbeats. If any had crashed, the reaper would have kicked in accordingly.

Thus, the integration touches the entry points of each agent's workflow (the shell scripts or commands that start tasks) and ensures they all respect the centralized `.contract` coordination directory.

Monitoring & Observability

To confidently run multiple agents in parallel, we need good visibility into what each agent is doing and how the system is performing. The coordination protocol provides a wealth of data (status files, logs, queue info) that can be harnessed for monitoring.

Agent Dashboard: A simple dashboard (could be a CLI script or a small web page served locally) can be built to aggregate `.contract/status` and `.contract/queue` information. For example, a Python script can read all `status/*.json` files and print a table:

```
Agent     State    Current Task        QueueDepth   LastHeartbeat
Beatrice  working  Add feature X (id123)   2          10:51:00Z
Otto      blocked  Waiting for lock on Y   1          10:51:05Z
Rem       idle     (none)                 0          10:50:59Z
```

It can also list active tickets (which files are locked by whom) and queued tasks with their priorities. This gives a real-time snapshot, so the developer (or even the agents themselves) can see if someone is stuck or if work is piling up.

Logging and Audit Trails: Every significant event in the coordination system should be logged to `.contract/logs/coordination.log` (or split into multiple logs for different concerns):
- Ticket events: lock acquired, lock released, lock expired, preemption occurred.
- Deadlock events: detected cycle between these agents, resolved by killing X's lock.
- Task events: task queued, task assigned to agent, task completed, task failed/retried.
- Errors: e.g., "Failed to acquire lock for 60s, giving up", or "Agent Rem heartbeat missed".

These logs not only help in debugging but can also feed metrics.

Metrics to Track: We define success metrics to evaluate the system's effectiveness, and these can be gathered from the logs and status:
- **Collision Rate:** Ideally zero. This measures how often two agents attempted to write to the same file without coordination. With the protocol in place, this should drop to 0% (compared to the ~5% collisions before). Any time a collision is only avoided by the lock (which is fine) or if the lock fails and a collision happens (should not happen), we note it.
- **Deadlocks and Recovery Time:** How many deadlocks were detected, and how quickly resolved. The target is that deadlocks, if they occur at all, are resolved within 60 seconds. We can measure the time from detection to resolution. Possibly, with ordering and proper design, actual deadlocks will be rare; but if tasks are complex, it might happen occasionally.
- **Throughput and Latency:** Number of tasks completed per hour and average task latency (time from entering queue to completion). With parallel agents and smart routing, we expect throughput to increase and latency to decrease (especially for tasks that can now run concurrently rather than sequentially).
- **Agent Utilization:** How often each agent is busy vs idle. This helps identify bottlenecks or if one agent is overloaded. The specialization/routing should yield a more balanced utilization. If, for example, Beatrice is 100% busy while others idle, the router might need adjustment.
- **Manual Intervention Count:** How often did a human have to step in (e.g., fix a conflict or restart an agent)? We aim to reduce this by at least 50%. Ideally, once the system is stable, the human mostly monitors and rarely intervenes.

These metrics can be derived from log analysis scripts. For example, each time a deadlock is resolved, log an entry with timing; each time an agent times out, log it. A simple Python script can parse these and output a summary at the end of the day.

Monitoring Tools: We can extend the `cli.py` to have commands like `./scripts/coordination/cli.py status` (to print the dashboard), `cli.py locks` (to list current locks), `cli.py metrics` (to output some basic counts from logs). This isn't strictly required but can greatly enhance usability. Over time, one could even integrate this with a small web UI or use an existing monitoring tool (like feeding events to Grafana or a terminal UI) – but since we want zero external dependencies, sticking to console outputs and maybe a minimal Flask app for a web view (if one is willing to run it) could suffice.

Observability during Development: During the rollout of this system, it will be important to observe that everything works as expected. For that, we can run in a **verbose/debug mode** initially: every ticket operation prints to console or log, so we see the sequence of events. We might simulate some conflicts to test deadlock resolution (e.g., have two dummy agents deliberately try to lock each other and see the system break the tie). Ensuring that log messages are clear ("[DeadlockDetector] Cycle detected: Beatrice -> Otto -> Beatrice. Preempting Otto's lock.") will make it easier to trust the automation.

Finally, **scalability & dynamic adjustments**: The system is built to handle adding or removing agents easily. If tomorrow a new agent "Mallory" is introduced (say using a new model), we just give Mallory a status file and perhaps update the specialization matrix (maybe Mallory specializes in UI design tasks, for example). The coordination logic (tickets, deadlock detection) doesn't need to change – it's generic. Also, if we temporarily want to scale down (say run only 2 agents instead of 3 due to resource limits), we can do so by not launching the others; the queue will simply be processed by whoever is available. The monitoring dashboard would show fewer agents, but tasks still get done (maybe slower). This flexibility ensures the protocol is future-proof for LLMC's evolving needs.

Conclusion

By introducing a formal ticket-based coordination protocol, LLMC can transform its multi-agent workflow from a fragile, manual process into a robust, self-managed system. The design outlined ensures that **no two agents will edit the same file concurrently** (eliminating lost work), that any waiting or blocking is detected and resolved intelligently (via deadlock breaking and task rerouting), and that agents can **collaboratively work** on complex projects without stepping on each other's toes. The `.contract/` directory becomes the single source of truth for coordination state – much like a mini operating system managing processes and resources – but implemented with simple file primitives for transparency and ease of debugging.

Moving forward, the implementation can be done in phases: start with basic locking and status reporting, then layer on deadlock detection and advanced routing. Testing with simulated agents will be crucial to fine-tune TTLs and priorities. Once in production, the expected outcome is a dramatic reduction in conflicts (target ~0% from ~5%) and smoother parallel task execution. This will enable the team to confidently run 6–8 or even more agent sessions in parallel, scaling up development throughput. Moreover, the principles here align with known best practices in distributed systems and multi-process OS design, adapted to LLMC's local environment – ensuring the solution is not only practical but built on a sound theoretical foundation (e.g., wait-for graphs for deadlocks and TTL locks).

With this coordination protocol in place, LLMC's Beatrice, Otto, Rem (and future agents) will truly function as a **cohesive, cooperative unit** rather than isolated sessions. Developers can focus on high-level guidance and let the agents handle the low-level dance of not colliding with each other. The end result: faster development cycles, fewer errors, and a scalable path to adding more agents or more complex workflows in the future.

Sources:

- LLMC internal design notes and collision scenarios (provided in prompt).
- Dicklesworthstone, *Claude Code Agent Farm* – example of multi-agent lock coordination (GitHub).
- Wikipedia – Wait-for graph concept for deadlock detection.
- Google Chubby lock service paper – inspiration for file-based locking approach [1].
- Varun Kruthiventi, *Distributed Locking with Redis* – TTL lock expiration pattern.

---

[1] systems.cs.columbia.edu
https://systems.cs.columbia.edu/ds1-class/lectures/09-chubby.pdf

[2] AGENTS.md
https://github.com/vmlinuzx/glideclubs/blob/192e899e53e6dc63cf3660b86021b03227bf03d9/AGENTS.md