DEFENSIVE PUBLICATION — TECHNICAL DISCLOSURE (TDCOMMONS DEPOSIT COPY) · 2026-06-25

Keywords: defensive-publication, prior-art, ai-agents, multi-agent-coordination, workflow-orchestration, concurrency-control, human-in-the-loop, audit-trail

Task-as-Hub Coordination for a Mixed Human/AI-Agent Workforce

A Technical Defensive Publication

Subtitle: A single database-backed Task record that owns a unit of work, arbitrates concurrent human/AI-agent writes through its state machine in lieu of git-branch ownership, and enforces an id-reference-only no-double-entry invariant across calendar, notes, timesheet, and activity ledger.

Field	Value
Publisher / copyright holder	Gus IT LLC (Florida, USA)
Author	Gustavo Assuncao, PhD
Publication date	2026-06-25
Version	1.0
Document type	Technical Defensive Publication (public prior art)
Classification	Public
License	AGPL-3.0-or-later (copyleft; commercial license available)
Deposit channel	To be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record.
Citation	Assuncao, G. (2026). *Task-as-Hub Coordination for a Mixed Human/AI-Agent Workforce*, v1.0. Gus IT LLC.

Abstract

A modern software organization increasingly runs a **mixed workforce**: human employees alongside a fleet of autonomous AI agents ("personas") that are asked to do work, do it, and report findings — concurrently, around the clock. Two structural failures recur. First, teams adopt the **version-control branch** as the implicit unit of work ownership. A branch, however, models *file isolation*, not *work ownership*: it does not represent assignment, lifecycle, blockage, priority, or effort, and when many agents operate on one shared repository, branch identity provides no mechanism to arbitrate *who holds* a unit of work, producing collisions, lost commits, and stomped working trees. Second, the same work data — the request, the hours, the progress narrative, the outcome, the billable effort — is **re-entered** across a ticketing tool, a calendar, a notes app, and a timesheet, producing drift and destroying auditability.

This publication describes a coordination architecture that resolves both with one move: make a single mutable **Task** row in a relational database the *authoritative owner* of a unit of work for the whole mixed workforce. Every intake channel resolves to a single idempotent entry point that creates exactly one Task,

carrying an `origin_ask_ref` to the channel's own system-of-record rather than copying it. Work sessions, progress notes, the resolution note, the timesheet rollup, and the activity ledger each reference the Task **by foreign key** — an **id-reference-only, no-double-entry** invariant — and the timesheet total is *mechanically aggregated* from referenced work-session durations rather than hand-keyed. Concurrent agents acquire and arbitrate write ownership at **task-record write time** through the Task's finite-state lifecycle (`open → in_progress → blocked/complete`), so the **Task — not the branch — owns the work**; a conflicting concurrent claim is serialized by the record's state machine before any repository write. The effect: every unit of work has one auditable owner, an AI persona's workday is as legible and accountable as a human employee's, and branch-ownership conflicts among concurrent agents are removed structurally. This document is published as prior art to keep the technique freely practiceable.

---

1. Executive Summary

1.1 Thesis

The right primitive for coordinating a mixed human/AI workforce is **not** a version-control branch and **not** a per-tool record duplicated across apps. It is a **single, mutable, database-backed Task record** that (a) is the sole owner of a unit of work, (b) arbitrates concurrent writes through its own finite-state lifecycle *before* any side-effecting repository action, and (c) acts as the join key for an id-reference-only data model so that no work datum is ever stored twice.

1.2 Contributions

#	Contribution	Where
C1	**Task-as-hub for a mixed workforce.** One mutable Task row owns a unit of work performed by humans, AI personas, groups, or a mix; `actor_type` partitions the populations without forking the model.	§6, §8
C2	**Single idempotent intake.** Every channel (UI, chat, mailbox, issue tracker, code-review, agent-to-agent dispatch) resolves to one `createTaskFromAsk(ask)` that writes exactly one Task and stores an `origin_ask_ref` instead of re-copying the request.	§7.2
C3	**Id-reference-only / no-double-entry invariant.** Calendar work-sessions, notes, resolution note, timesheet, and activity ledger reference the Task by id; each entity has exactly one owning table.	§5, §8
C4	**Branch-free write-time concurrency arbitration.** Concurrent agents acquire/arbitrate write ownership via the Task FSM at record-write time, *replacing* git-branch ownership as the coordination primitive.	§7.3, §7.4
C5	**Mechanical effort aggregation.** The timesheet `total_hours` is computed as the sum of referenced work-session durations — never hand-keyed — making effort tamper-evident and reconcilable.	§7.6
C6	**Work-discipline RBAC.** Capability gates (`{app}:work-discipline:{view	write}`) govern persona access to the hub and segregate a cross-persona "AI view" from human-private data.

1.3 Headline claim (plain language)

> *A mixed human/AI workforce is coordinated by making one database Task record the owner of each unit of work; concurrent agents arbitrate write ownership through that record's state machine instead of through git branches, and all downstream artifacts (calendar, notes, timesheet, ledger) reference the Task by id so nothing is entered twice and effort is summed mechanically.*

1.4 Scope

This is: a coordination and accountability architecture; a data model; an intake-to-rollup discipline; a write-time arbitration mechanism that uses a DB record rather than a branch.

This is NOT: a replacement for version control (the repository still holds code; the Task governs *who may write*); a new database engine; a scheduler or autoscaler; a claim of compliance certification with any named standard (we assert *semantic alignment* only, §12); a product manual.

---

2. Introduction & Motivation

2.1 The concrete problem

Consider an organization that has stood up a fleet of autonomous AI coding agents. Each agent is "asked" to do work through some channel — a person types a request in a chat box, an email lands in a shared mailbox, a code-review bot files a finding, or one agent dispatches a sub-task to another. The agent then edits a shared code repository, runs tests, and reports back.

Two questions immediately become hard to answer:

1. **Who owns this unit of work, right now, and what state is it in?** Teams answer this *implicitly* by convention — "the agent on branch `agent/x/feature-y` owns feature-y." But a branch is just a named pointer into a commit graph. It carries no assignment, no lifecycle, no blockage, no priority, no effort, and — crucially — **no arbitration**: if two agents both decide to work the same feature, nothing about branch identity stops them, and they collide in the shared tree.
2. **How did the work go, and how much did it cost?** The request lives in the chat log; the hours live (maybe) in a calendar; the progress narrative lives in scattered comments; the outcome lives in a closing message; the billable effort lives (maybe) in a timesheet someone typed by hand. The same facts are re-entered, drift apart, and no single place answers "what was agent X asked to do, how did the work go hour by hour, what did it produce, and how long did it take?"

2.2 The "tax"

These two failures impose a continuous tax:

- **Concurrency tax.** Lost commits, stomped working trees, and rebase churn when agents collide; engineer time spent untangling who-was-doing-what. Because the branch never *arbitrated* ownership, the only conflict detector is the merge — far too late.

- **Double-entry tax.** Every datum entered N times across N tools costs N× the entry effort and, worse, produces N− 1 stale copies. A timesheet typed by hand diverges from the calendar it was supposed to summarize, and the audit trail becomes untrustworthy.
- **Legibility tax.** Without one owner per unit of work, an AI persona's day is a black box. Leadership cannot review it the way they review a human's day, so the AI workforce is neither auditable nor accountable.

2.3 Why existing approaches fall short

- **Work-item trackers (Jira, Azure DevOps).** They make a *ticket* the hub and even link branches/PRs to it — but the ticket does not *arbitrate concurrent write ownership* of a shared repository, and effort fields are typically hand-entered or imported, not mechanically derived from a referenced calendar.
- **Branch-per-agent orchestration.** Treats the branch as the unit; inherits exactly the concurrency tax above.
- **Omnichannel intake (ServiceNow).** Funnels many channels into tickets, but for a human service-desk model, not a mixed human/AI work-execution model with write-time arbitration.
- **Calendar→timesheet automation.** Exists for humans (e.g., deriving timesheets from calendar events), but is not wired to a Task hub that also owns concurrent agent writes.

None of these *combine* a single owning record, a no-double-entry id-reference model, **and** write-time concurrency arbitration that displaces the branch — for a workforce that is explicitly part-human, part-AI.

2.4 Why now

Autonomous AI coding agents reached the point in 2024–2026 where many of them run concurrently against shared repositories. The branch-as-ownership convention, inherited from human git workflows, breaks under fleets of agents. The mixed workforce also created a new accountability demand: an AI persona that does real work must be as auditable as an employee. Both pressures make the Task-as-hub discipline newly necessary — and worth publishing so it stays open.

---

3. Problem Statement

3.1 Formal framing

Let a **unit of work** w be a request that must be assigned, performed over time, documented, concluded, and accounted for. Let the workforce be a set of **actors** $A = H \cup P$, where H is humans and P is autonomous AI personas. Let R be a shared code repository under version control. We require a coordination object $T(w)$ such that:

1. $T(w)$ is the **sole authoritative owner** of w for any actor in A.
2. The artifacts of w — request, work-sessions, notes, resolution, effort, ledger — are each owned by exactly one table and **reference** $T(w)$ by id; no datum is stored twice.
3. When two actors $a_1, a_2 \in A$ attempt to take w concurrently, the conflict is **detected and serialized at the moment a claim is written to $T(w)$**, *before* either actor writes to R.
4. The **effort** of w is derived mechanically from artifacts referencing $T(w)$, never hand-keyed.
5. Access to $T(w)$ and its cross-actor views is governed by capability checks.

3.2 Derived requirements

Req	Requirement	Satisfied in
R1	Exactly one Task record is created per ask, regardless of channel; the request body is stored once; an `origin_ask_ref` links to any pre-existing system-of-record row.	§7.2, §8.1
R2	A Task owns a unit of work for humans, personas, groups, or a mix; populations are partitioned by `actor_type`, not forked into separate models.	§6, §8.1
R3	Calendar work-sessions, notes (progress + resolution), timesheet, and ledger reference the Task by foreign key; each entity has one owning table (no double entry).	§5, §8
R4	A Task moves through a finite-state lifecycle (`open → in_progress → blocked → complete/cancelled`) that rejects illegal transitions.	§7.3
R5	Concurrent claims to a Task are arbitrated at task-record write time and serialized by the state machine *before* any repository write.	§7.4
R6	Timesheet `total_hours` equals the summed durations of referenced work-session events; it is never hand-keyed for discipline-linked entries.	§7.6
R7	Every interaction is recorded in an activity ledger row that carries the Task reference.	§7.7
R8	Capability gates govern persona read/write of the hub and segregate cross-persona "AI view" from human-private rows.	§11, §12
R9	Every behavior-controlling value (enabled flag, default view, eligible channels, group-completion rule, rollup cadence) is externally configurable.	§7.8
R10	The model is additive over a pre-existing single-actor (human-only) system; v1 rows and workflows are unchanged.	§8.6

4. Related Work & Prior Art

This architecture is assembled deliberately on top of well-understood building blocks. We name the relevant prior art and state what we adopt from each — the contribution is the *combination and the write-time arbitration*, not the invention of any single block ex nihilo.

Prior art	What we adopt / acknowledge
Jira / Azure DevOps work-item-as-hub + branch/PR linking	The idea that a tracked item is the hub and that branches/PRs link *to* it. We extend the hub from a passive tracker to an *active owner that arbitrates concurrent writes*.
ServiceNow omnichannel intake	Funneling many intake channels into a single record. We generalize it to a mixed human/AI work-execution flow with a single idempotent `createTaskFromAsk`.
Calendar-to-timesheet automation (e.g., TimeCamp / Toggl; US 9,659,260 B2)	Deriving effort from calendar events instead of hand entry. We bind it to the Task hub and the id-reference model.
Workday Agent System of Record (ASOR)	The concept of a system of record spanning a mixed human/agent workforce. We make the *unit-of-work record* the hub and add write-time repository arbitration.
STORM: Multi-agent Collaboration with State Management; ESAA (event-sourced autonomous agents)	Shared mutable state coordinating multiple agents; event-sourced agent histories. We acknowledge these for shared-state arbitration and use a DB record's FSM as the arbitration point in lieu of a branch.
Postgres `SELECT … FOR UPDATE [SKIP LOCKED]`; advisory locks	Row-level write serialization primitives used to implement the claim-at-write-time guarantee.
Database normalization (3NF) / single-source-of-truth modeling	The id-reference-only invariant is classic normalization; we apply it across a multi-app work-discipline boundary.
US 9,020,885 / US 8,375,086 (shared-state managers)	General shared-state management for concurrent processes; acknowledged as adjacent to write-time arbitration.
Google "Rosie" / LLM code-migration at scale (Ziftci et al., FSE 2025)	Fleets of automated change agents operating on shared code; motivates the concurrency problem this addresses.

Honest posture. As recorded in this portfolio's own screening, every *element* here is a commodity, and the strongest candidate-technical effect (write-time concurrency arbitration on shared state) is adjacent to STORM/ESAA and shared-state-manager patents. We therefore publish defensively rather than claim a strong patent. The novelty we assert is the *specific combination*: a Task hub that simultaneously (a) owns a mixed human/AI unit of work, (b) enforces id-reference-only no-double-entry across calendar/notes/timesheet/ledger, and (c) arbitrates concurrent agent writes at record-write time *in lieu of branch ownership*, with effort summed mechanically. See §5 and Appendix A.

---

5. Prior-Art Delta

Per novel feature, what the closest prior source *has*, what it *lacks*, and what this disclosure *adds*.

Prior source	What it has	What it LACKS	What THIS adds
Jira / Azure DevOps work-item-as-hub	A tracked item; branch/PR links; manual effort fields	The item does not **arbitrate concurrent write ownership** of a shared repo; effort is hand-entered/imported	The hub *arbitrates writes at record-write time* and **mechanically** sums effort from referenced calendar events
Branch-per-agent orchestration	File isolation per agent	No assignment/lifecycle/effort; **no arbitration** — conflicts surface only at merge	Replaces the branch with a **DB Task FSM** that serializes claims *before* any repo write
ServiceNow omnichannel intake	Many channels → one ticket	Human service-desk model; no mixed human/AI **work-execution** with write arbitration	One idempotent `createTaskFromAsk` for a **mixed** workforce + `origin_ask_ref` (no re-copy)
Calendar→timesheet automation (US 9,659,260 B2)	Effort derived from calendar	Not bound to a Task hub that also **owns concurrent agent writes**	`total_hours` summed from work-sessions **that reference the same Task** the agents arbitrate on
Workday ASOR (mixed workforce SoR)	Mixed human/agent system of record	Not a **unit-of-work record** that arbitrates **repository** writes	The **Task** (unit of work) is the hub *and* the write-arbitration point
STORM / ESAA / US 9,020,885 (shared-state)	Shared mutable/event-sourced state for agents	Not framed as a **task record replacing a git branch** with an id-reference no-double-entry model around it	A normalized **Task-as-hub** model + branch-displacing arbitration + no-double-entry invariant as one discipline

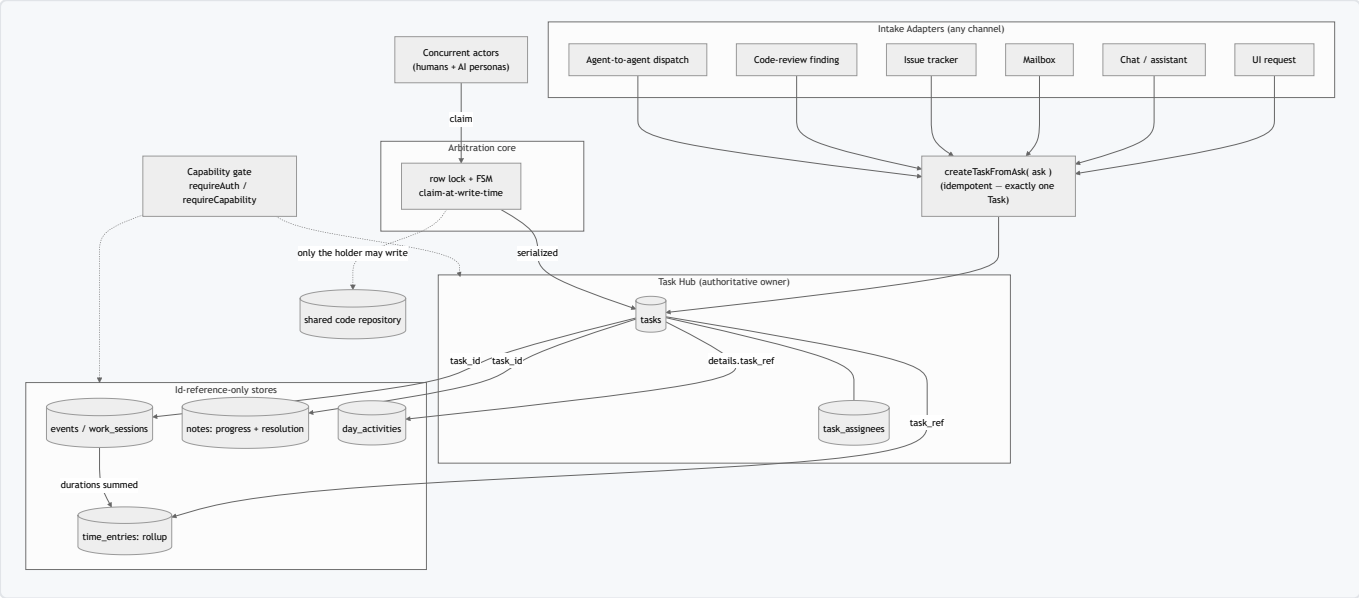
6. System Architecture

6.1 Components

- **Intake Adapters** — one per channel (UI, chat, mailbox, issue tracker, code-review, agent-to-agent dispatch). Each normalizes an *ask* and calls the single entry point.
- **Task Hub (`tasks`)** — the authoritative record of a unit of work; carries subject, body, `actor_type`, assignee shape, lifecycle `status`, `origin_ask_ref`, and optional `project_ref` / `process_ref`.
- **Assignee set (`task_assignees`)** — the persona(s), human(s), or group bound to a Task, with per-assignee `picked_up_at` / `resolved_at`.
- **Work-session store (`events`)** — calendar events with `task_id` set (`event_kind = work_session`); the worked hour ranges.
- **Notes store (`notes`)** — progress notes (many) and the resolution note (one per assignee), each with `task_id`.
- **Timesheet rollup (`time_entries`)** — one rollup per complete Task; `total_hours` summed from referenced work-sessions.
- **Activity ledger (`day_activities`)** — interaction ledger; each row carries the Task reference in its details payload.
- **Arbitration core** — the write-time claim logic that uses row locking and the Task FSM to serialize concurrent claims.
- **Capability gate** — `requireAuth` on data routes, `requireCapability` on mutations; segregates AI view from human-private rows.

6.2 Architecture diagram

> **Figure 1 — System architecture.** Intake adapters fan into one idempotent entry point that writes a single Task; all downstream stores reference the Task by id; the arbitration core gates concurrent agent writes.



6.3 Cross-cutting properties

- **Single source of truth per entity.** Each store owns exactly one entity type; cross-store relationships are foreign keys, never copies (§5, §8).
- **Additivity.** The discipline is layered onto a pre-existing human-only system by adding NULLable/defaulted columns; legacy rows backfill to safe defaults and legacy human workflows are byte-for-byte unchanged (R10).
- **Fail-safe arbitration.** The arbitration core's posture and failure modes are specified in §11.
- **Configurability.** Every behavior-controlling value is external (R9, §7.8).

---

7. Detailed Mechanics

7.1 The canonical three-stage flow

A unit of work moves through **Ask → Task → Work Loop → Reporting**:

1. **Ask → Task (Stage 1).** An ask on any channel auto-creates exactly one Task.
2. **Pickup → Work Loop (Stage 2).** An assignee picks up the Task (→ `in_progress`), blocks calendar work-sessions, writes progress notes, and finally a resolution note.
3. **Reporting (Stage 3).** A complete Task rolls up to a timesheet entry (hours summed from work-sessions) and the activity ledger.

7.2 Stage 1 — idempotent intake (R1, R2)

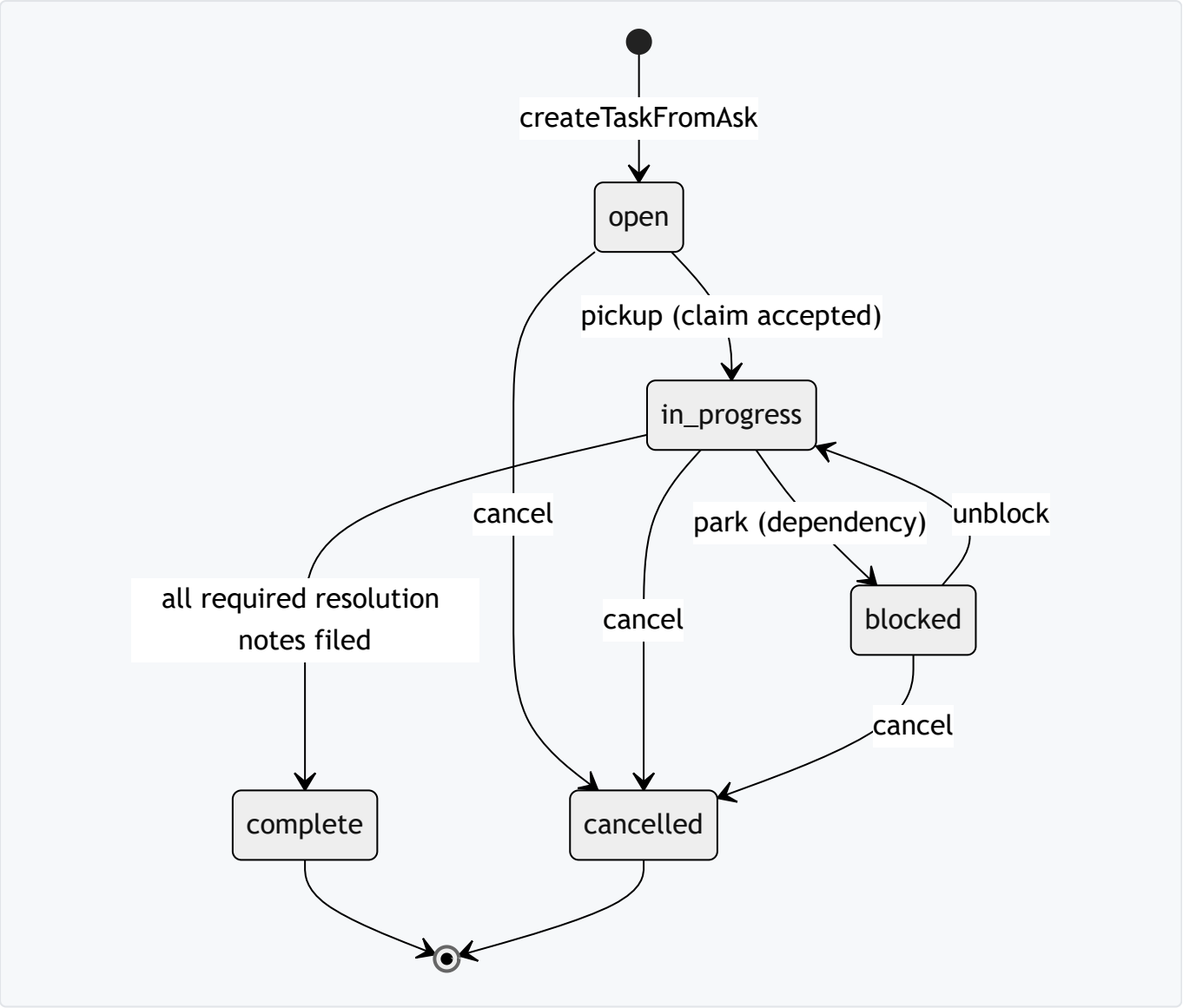
Every channel resolves to one internal call:

```
function createTaskFromAsk(ask):
    # ask: { channel, origin_ref?, subject, body, assignees[], assoc? }
    origin = ask.origin_ref            # e.g. mailbox id / issue url / CR path
    if origin and taskExistsForOrigin(origin):
        return existingTaskFor(origin) # idempotent: never two Tasks per ask
    task = insert into tasks {
        subject       : shorten(ask.subject),
        body          : ask.body,        # stored ONCE here
        actor_type    : ask.assignees.anyPersona ? 'persona' : 'human',
        assignee_kind : shape(ask.assignees),   # single | group | mixed
        status        : 'open',
        origin_ask_ref: origin,          # link, do NOT re-copy origin body
        project_ref   : ask.assoc.project ?? NULL,
        process_ref   : ask.assoc.process ?? NULL
    }
    for a in ask.assignees:
        insert into task_assignees { task_id: task.id, assignee_type: a.type,
                                     assignee_id: a.id }
    return task
```

Key invariants. (a) Exactly one Task per ask (idempotency keyed on `origin_ask_ref`). (b) The request body is stored once on the Task; if the ask already has a system-of-record row, the Task links to it via `origin_ask_ref` rather than copying it. (c) `actor_type` partitions human vs. persona without a separate schema.

7.3 Stage 2 — the Task finite-state machine (R4)

> **Figure 2 — Task lifecycle state machine.** Legal transitions only; the FSM is the arbitration surface.



State	Meaning	Entry condition
`open`	Generated from an ask; unclaimed	`createTaskFromAsk`
`in_progress`	≥1 assignee has an accepted claim; work loop active	a `pickup` claim won arbitration
`blocked`	Parked pending a dependency	explicit `park`
`complete`	Required resolution note(s) filed (per group-completion rule)	resolution-note write satisfies the rule
`cancelled`	Abandoned before completion	explicit `cancel`

Illegal transitions (e.g., `open → complete`, `complete → in_progress`) are rejected by the FSM guard.

7.4 Branch-free write-time concurrency arbitration (R5) — the core mechanism

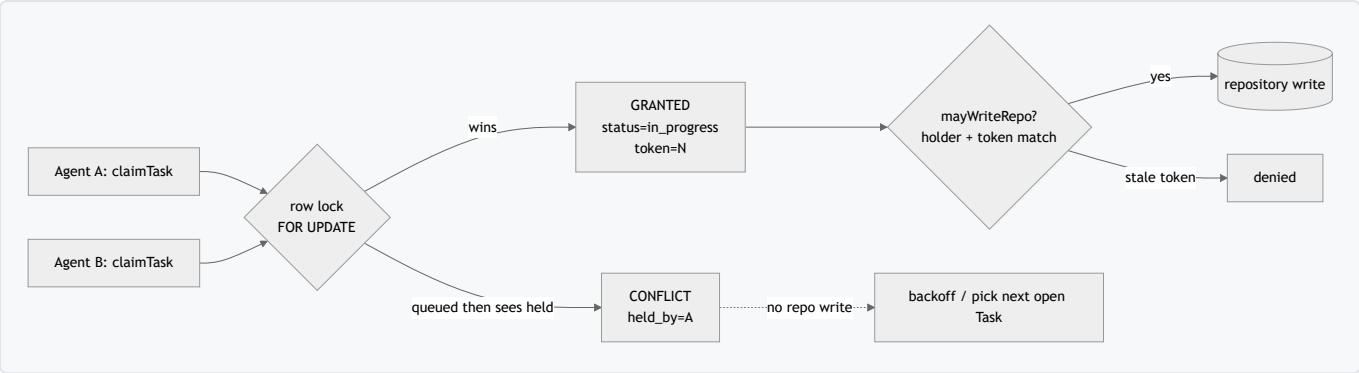
The central mechanism replaces "the branch owns the work" with "the **Task record** owns the work," and detects/serializes conflicts at the moment a claim is written to the Task — *before* any repository write.

```
function claimTask(taskId, claimant):
    BEGIN TRANSACTION
        # 1. Serialize on the row itself.
        row = SELECT status, claimed_by, claim_token
              FROM tasks WHERE id = taskId
              FOR UPDATE                      # row lock: concurrent claims queue here

        # 2. FSM guard: only an open (or self-held) Task is claimable.
        if row.status NOT IN ('open') and row.claimed_by != claimant:
            COMMIT                            # release lock
            return CONFLICT(row.claimed_by)   # serialized: loser told who holds it

        # 3. Accept the claim atomically with the state transition.
        token = newFencingToken()             # monotonically increasing
        UPDATE tasks
           SET status = 'in_progress',
               claimed_by = claimant,
               claim_token = token,
               claimed_at = now()
         WHERE id = taskId
    COMMIT
    return GRANTED(token)

# A repository write is permitted ONLY by the current holder + token.
function mayWriteRepo(taskId, claimant, token):
    cur = SELECT claimed_by, claim_token FROM tasks WHERE id = taskId
    return cur.claimed_by == claimant AND cur.claim_token == token
```

Why this works:

- **The conflict is detected at claim time, not merge time.** Two agents racing to take the same Task queue on the `FOR UPDATE` row lock; exactly one wins, the other is told who holds it. The shared repository is never touched by the loser.
- **The branch is demoted to an implementation detail.** A holder may still *use* a branch to stage code, but ownership and arbitration live in the Task record, so branch naming collisions cannot cause ownership ambiguity.
- **Fencing token** prevents a stale holder (e.g., a paused agent that resumes after its claim was reassigned) from writing: `mayWriteRepo` checks the live token.
- **Row locking + optional `SKIP LOCKED`** lets a dispatcher hand out the *next* open Task to an idle agent without two agents grabbing the same row.

> **Figure 3 — Write-time arbitration data-flow.** Two agents race; the row lock
>
> - FSM serialize the claim; only the holder + fencing token may write the repo.



7.5 Stage 2 — work-sessions and notes (R3)

While the work loop runs, the holder:

- Creates **work-sessions**: calendar `events` rows with `task_id` set and `event_kind = work_session`; `start_at`/`end_at` are the worked hour range. Many per Task.
- Writes **progress notes**: `notes` rows with `task_id` set and `note_kind = progress`. Many per Task.
- On completion writes one **resolution note**: a `notes` row with `task_id` set and `note_kind = resolution`. One per assignee.

The resolution-note write **signals the FSM**: when every required assignee has filed a resolution note (per the configured group-completion rule), the Task transitions to `complete`.

7.6 Stage 3 — mechanical effort rollup (R6)

On completion, one timesheet `time_entries` row is produced:

```
function rollupTimesheet(taskId):
    sessions = SELECT id, start_at, end_at FROM events
               WHERE task_id = taskId AND event_kind = 'work_session'
    noteIds  = SELECT id FROM notes WHERE task_id = taskId
    hours    = sum( (end_at - start_at) for s in sessions )   # MECHANICAL
    insert into time_entries {
        task_ref         : taskId,
        related_event_ids: ids(sessions),   # references, not copies
        related_note_ids : noteIds,         # references, not copies
        total_hours      : hours            # never hand-keyed
    }
```

`total_hours` is **derived**, not entered. Because it is the sum of the very work-session rows it references, the rollup is reconcilable and tamper-evident: an auditor can recompute it from the referenced ids.

7.7 Stage 3 — activity ledger (R7)

Each interaction during the work loop writes a `day_activities` row whose details payload carries `task_ref = taskId`. No schema change to the ledger is needed; the discipline only guarantees the Task reference is present so the ledger joins back to the hub.

7.8 Configuration (R9)

Key	Purpose	Default
`WORK_DISCIPLINE_ENABLED`	Master feature flag; the surface stays dark until enabled	off
`WORK_DISCIPLINE_DEFAULT_VIEW`	Default per-app view (`human` / `ai`)	`human`
`WORK_DISCIPLINE_AUTO_TASK_CHANNELS`	Which channels auto-create a Task	configurable list
`WORK_DISCIPLINE_GROUP_COMPLETION_RULE`	`all-assignees` vs `any-assignee` for group completion	`all-assignees`
`WORK_DISCIPLINE_TIMESHEET_ROLLUP_CADENCE`	When Stage-3 rollup runs (`on-complete` / `scheduled`)	`on-complete`

7.9 Error handling

- **Illegal FSM transition** → rejected by guard; caller receives the current state and the set of legal transitions.
- **Lost claim race** → `CONFLICT(holder)`; loser backs off and may request the *next* open Task.
- **Stale holder write** → `mayWriteRepo` denies on token mismatch (fencing).
- **Partial rollup failure** → the rollup is idempotent on `task_ref`; re-running recomputes from referenced ids without creating duplicates.
- **Orphan reference** → referential integrity constraints (FK) prevent a work-session/note/timesheet row pointing at a non-existent Task.

---

8. Data Model

8.1 Tables and fields

Names below are illustrative and engine-neutral; the disclosure does not depend on any vendor schema. All discipline columns are additive over a pre-existing human-only system (R10): NULLable or defaulted, with legacy rows backfilling to `human` / `general` / `event`.

`tasks` — the hub

Column	Type	Null	Default	Purpose
`id`	integer (PK)	no	—	Task identity / join key
`subject`	text	no	—	Short title from the ask
`body`	text	no	—	Full request text — **stored once**
`actor_type`	varchar(15)	no	`human`	`human` / `persona` partition
`assignee_kind`	varchar(15)	no	`single`	`single` / `group` / `mixed`
`status`	varchar(15)	no	`open`	FSM state
`claimed_by`	varchar(120)	yes	NULL	Current holder (arbitration)
`claim_token`	bigint	yes	NULL	Fencing token
`origin_ask_ref`	varchar(400)	yes	NULL	Link to the ask's system-of-record row
`project_ref`	varchar(200)	yes	NULL	Optional project association
`process_ref`	varchar(200)	yes	NULL	Optional process association (mutually exclusive with `project_ref`)
`priority`	varchar(10)	no	`medium`	`high` / `medium` / `low`
`due_date`	varchar(50)	yes	NULL	Optional target
`created_at`	timestamp	no	now	Creation time

`task_assignees` — the assignee mix

Column	Type	Null	Notes
`id`	integer (PK)	no	—
`task_id`	integer (FK→tasks.id)	no	The owning Task
`assignee_type`	varchar(15)	no	`persona` / `human` / `group`
`assignee_id`	varchar(120)	no	persona id, username, or group label
`picked_up_at`	varchar(50)	yes	set on pickup
`resolved_at`	varchar(50)	yes	set on resolution-note filing

`events` — work-sessions (calendar)

Column	Type	Null	Default	Purpose
`task_id`	integer (FK)	yes	NULL	References the Task
`event_kind`	varchar(20)	no	`event`	`event` / `work_session`
`start_at` / `end_at`	timestamp/iso	—	—	Worked hour range
`actor_type`	varchar(15)	no	`human`	partition
`persona_id`	varchar(100)	yes	NULL	Persona that worked the session

`notes` — progress + resolution

Column	Type	Null	Default	Purpose
`task_id`	integer (FK)	yes	NULL	References the Task
`note_kind`	varchar(15)	no	`general`	`general` / `progress` / `resolution`
`actor_type`	varchar(15)	no	`human`	partition
`persona_id`	varchar(100)	yes	NULL	Authoring persona

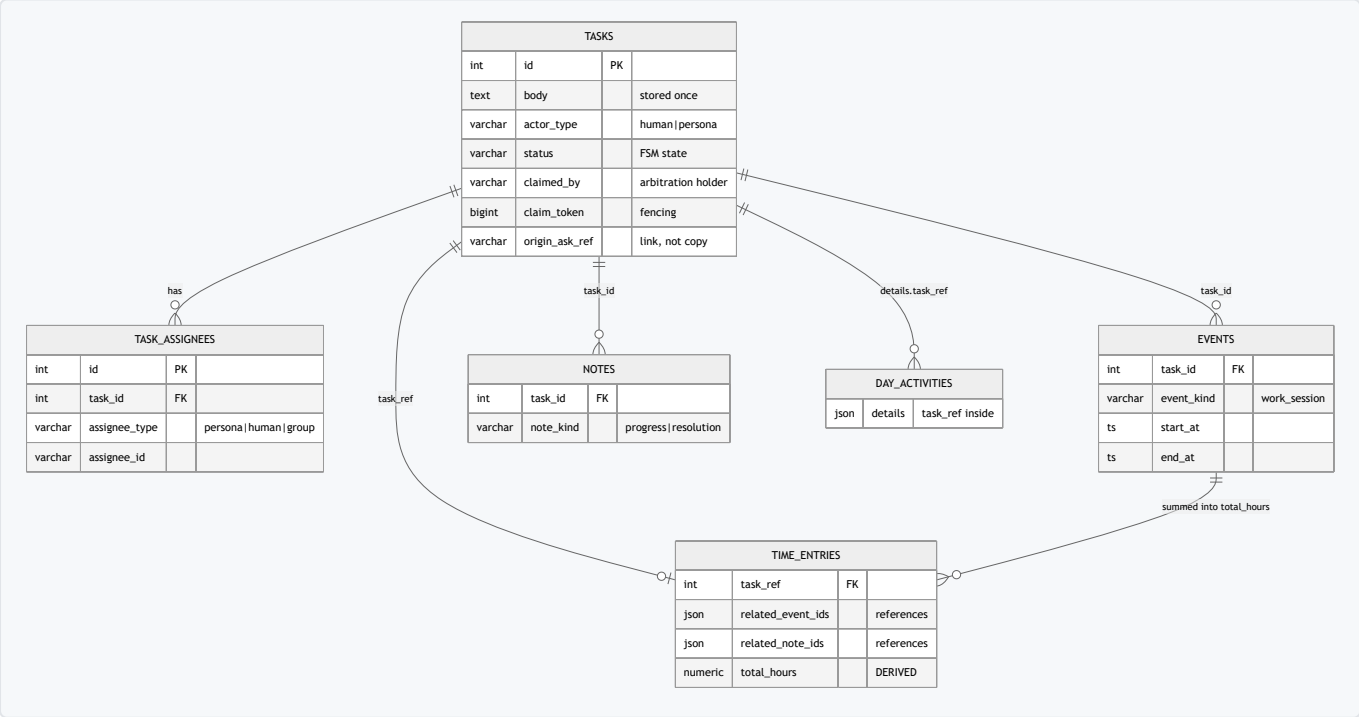
`time_entries` — the rollup

Column	Type	Null	Default	Purpose
`task_ref`	integer (FK)	yes	NULL	References the Task
`related_note_ids`	json/text	yes	NULL	Array of note ids — references
`related_event_ids`	json/text	yes	NULL	Array of work-session ids — references
`total_hours`	numeric	no	—	**Summed from `related_event_ids`**
`actor_type`	varchar(15)	no	`human`	partition
`persona_id`	varchar(100)	yes	NULL	Persona this entry is for

`day_activities` — **the ledger** (no schema change): carries `task_ref` inside its existing details payload.

8.2 Entity-relationship diagram

> **Figure 4 — Data model (ER).** The Task is the hub; every other entity references it by id. No body is duplicated.



8.3 The no-double-entry invariant (R3)

Entity	Single owning table	References (by id)
The work request	`tasks`	`origin_ask_ref` → the ask's existing row, if any
What was produced / how it went	`notes`	`note.task_id` → Task
When the work happened	`events`	`event.task_id` → Task
The final outcome	the one `resolution` note	`note.task_id` → Task
Effort / time	`time_entries`	`task_ref` → Task; `related_*_ids`; `total_hours` summed
Interaction ledger	`day_activities`	`details.task_ref` → Task

8.4 Recommended indexes

`tasks(status)`, `tasks(actor_type)`, `tasks(project_ref)`, `tasks(claimed_by)`; `task_assignees(task_id)`, `task_assignees(assignee_type, assignee_id)`; `events(task_id)`, `events(event_kind)`; `notes(task_id)`, `notes(note_kind)`; `time_entries(task_ref)`.

8.5 Referential integrity

Foreign keys from `events.task_id`, `notes.task_id`, `time_entries.task_ref`, and `task_assignees.task_id` to `tasks.id` make orphaned references impossible (R3). `project_ref` and `process_ref` are mutually exclusive (a CHECK or app-level guard); both NULL ⇒ standalone.

8.6 Additivity / migration posture (R10)

Every discipline column is NULLable or defaulted, so the migration is a non-blocking `ALTER TABLE ADD`. Legacy human-only rows backfill to `human` / `general` / `event`, preserving v1 behavior exactly. The discipline can therefore be retrofitted onto an existing single-actor system without downtime or data loss.

---

9. Reference Implementation & Enablement

9.1 What `src/` demonstrates

The `src/` directory contains an **original, clean-room** Node.js reference that reduces the core mechanism to practice with **zero external dependencies** (standard library only, an in-memory store):

- `task-hub.js` — the Task hub with the FSM, the write-time `claimTask` arbitration (row-lock semantics emulated with a per-row mutex), fencing tokens, the id-reference-only stores, and the mechanical `rollupTimesheet`.
- `demo.js` — a runnable scenario: two concurrent agents race for one Task (one wins, one is told who holds it), the holder logs two work-sessions and three progress notes, files a resolution note (auto-completing the Task), and the timesheet rolls up with `total_hours` equal to the summed session durations.
- `self-check.js` — assertions verifying R1, R3, R4, R5, R6 (exactly one Task per ask; no double entry; illegal transitions rejected; concurrent claim serialized; `total_hours` equals summed durations).

9.2 How it reduces the invention to practice

The reference shows, in executable form, the load-bearing claim elements: (1) a single mutable record owning a unit of work; (2) claim arbitration at record-write time via lock + FSM, *before* any "repository write"; (3) fencing-token gating of the side-effecting write; (4) effort derived by summation over referenced events, not entry. It is illustrative — not production software — and deliberately avoids any vendor, infra, or proprietary detail.

9.3 Configuration points

The reference exposes the §7.8 knobs as plain constructor options (`groupCompletionRule`, `rollupCadence`, `autoTaskChannels`), demonstrating R9 without any environment-specific binding.

---

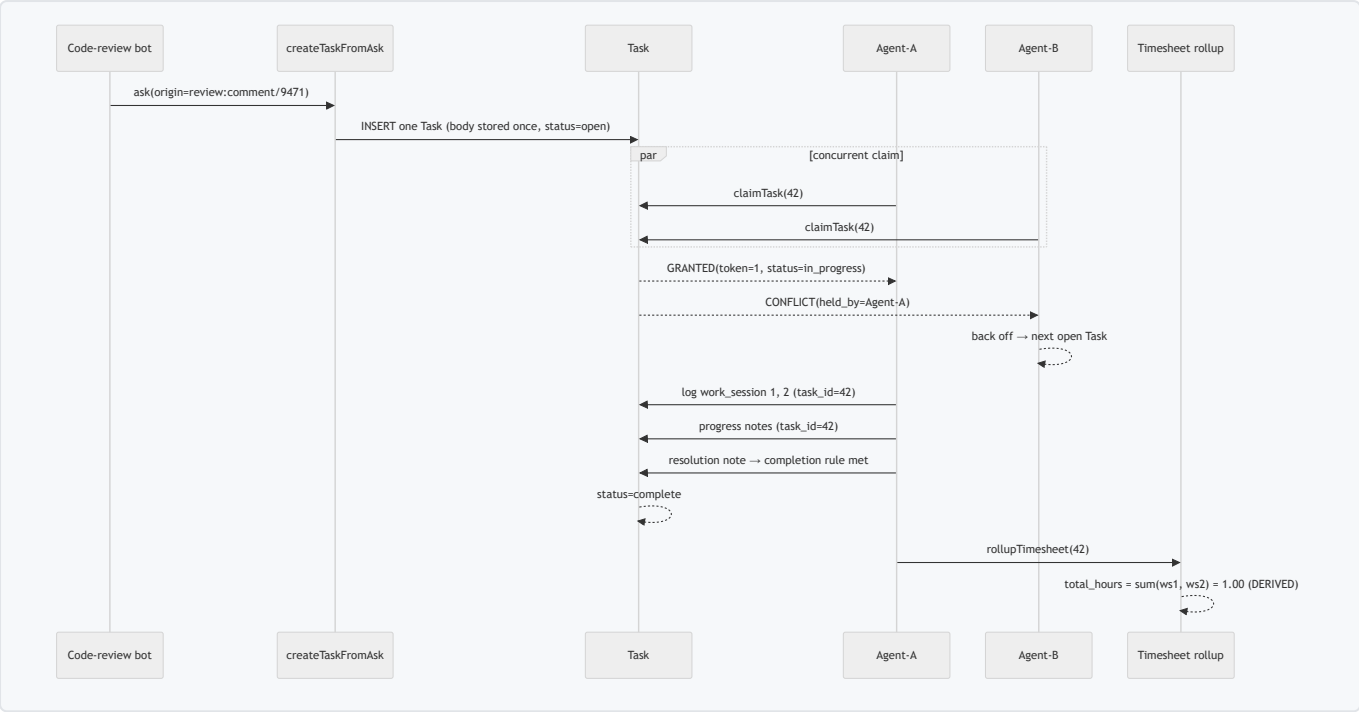
10. Worked Example / Scenario

Setup. Two AI personas, *Agent-A* and *Agent-B*, are both idle. A code-review bot files a finding: *"Fix the null-deref in the parser."* The finding already has a system-of-record row (a review comment id).

1. The code-review intake adapter calls `createTaskFromAsk` with `origin_ask_ref = review:comment/9471`. Exactly one Task `#42` is created with `body` = the finding text (stored once), `status = open`, `actor_type = persona`.
2. *Agent-A* and *Agent-B* both call `claimTask(42, …)` within milliseconds. Both queue on the Task row lock. *Agent-A* wins: Task `#42 → in_progress`, `claimed_by = Agent-A`, `claim_token = 1`. *Agent-B* receives `CONFLICT(held_by = Agent-A)` and backs off to the next open Task. **No repository write happened for the loser.**
3. *Agent-A* checks `mayWriteRepo(42, Agent-A, 1)` → true, edits the parser, logs **work-session 1** (`events`, `task_id = 42`, 09:10–09:40) and **work-session 2** (10:00–10:30), and writes **progress notes** at each step (`notes`, `task_id = 42`, `note_kind = progress`).
4. *Agent-A* files the **resolution note** (`notes`, `task_id = 42`, `note_kind = resolution`). The single-assignee completion rule is satisfied → Task `#42 → complete`.
5. `rollupTimesheet(42)` runs: `total_hours = 1.00` (30 min + 30 min), with `related_event_ids = [ws1, ws2]` and `related_note_ids = [progress×N, resolution]`. A `day_activities` row records the interaction with `details.task_ref = 42`.

Everything about Task `#42` — request, hours, narrative, outcome, effort — is now answerable from one place, the body is stored exactly once, and the hours are recomputable from the referenced sessions.

> **Figure 5 — Worked-example sequence.** Two agents race; the Task arbitrates; the holder works and reports; the rollup is mechanical.



11. Security, Safety & Failure Modes

11.1 Fail-open vs fail-closed posture

Surface	Posture	Rationale
Claim arbitration	**Fail-closed**	If the Task row cannot be locked/read, the claim is *denied* — never grant ownership on uncertainty (prevents two holders).
Repository write gate	**Fail-closed**	`mayWriteRepo` denies on any token/holder mismatch or read failure; a stale agent cannot write.
AI-view read	**Fail-closed**	Without the view capability, the user only ever sees Human view; AI-view endpoints 403.
Timesheet rollup	**Idempotent, recompute**	A failed rollup re-runs from referenced ids; never produces a second entry for the same `task_ref`.
Intake adapter	**Idempotent**	Re-delivery of the same ask returns the existing Task (keyed on `origin_ask_ref`), never a duplicate.

11.2 Failure-mode / threat table (STRIDE-flavored)

Threat / failure	Vector	Mitigation
Spoofing a claim as another actor	Forged claimant id	`requireAuth` on the claim route; `claimed_by` is the authenticated principal, not a body field
Tampering with `total_hours`	Hand-editing the rollup	`total_hours` is derived and recomputable from `related_event_ids`; divergence is detectable
Repudiation of who did the work	"I never claimed it"	Claim writes `claimed_by` + `claim_token` + `claimed_at`; ledger row carries `task_ref`
Information disclosure of human-private rows via AI view	Cross-actor query leak	AI-view queries filter `actor_type = persona`; human rows are never in the result set; view is capability-gated
Denial of service by claim storms	Many agents hammering one Task	Row lock + `SKIP LOCKED` queueing; losers back off to next open Task
Elevation of privilege	Persona writing without capability	`requireCapability('{app}:work-discipline:write')` on every mutation
Lost-update / double-holder	Two agents both "own" a Task	`SELECT … FOR UPDATE` serializes; FSM guard rejects the second; fencing token blocks stale writes
Stale-holder write	Paused agent resumes after reassignment	`mayWriteRepo` checks the live `claim_token`; mismatch ⇒ denied
Orphaned reference	Note/event pointing at a deleted Task	FK constraints prevent orphan rows

11.3 Safety note on the mixed workforce

Because AI personas act as first-class principals, every persona mutation is capability-gated exactly like a human's, and the AI view is an *accountability* surface, not a privilege escalation: it is read-only and cross-persona by design, and it can never surface a human's private rows.

---

12. Standards & Framework Mapping

We assert **semantic alignment** only — not certified compliance with any named standard.

Framework / standard	Relevant aspect	How this aligns (semantic)
Database normalization (3NF / single-source-of-truth)	One fact, one place	The id-reference-only invariant is a direct application across a multi-app boundary (R3)
ACID transactions (ISO/IEC 9075 SQL)	Atomicity + isolation of the claim	`claimTask` runs in a transaction with row locking; the claim+transition is atomic (R5)
Fencing tokens (distributed-locking literature)	Preventing stale lock holders from acting	`claim_token` gates `mayWriteRepo` (R5)
NIST SP 800-53 — AU (Audit & Accountability)	Traceable record of who did what, when	One auditable owner per unit of work; ledger carries `task_ref` (R7)
NIST SP 800-53 — AC (Access Control)	Least-privilege gating	`requireAuth` + `requireCapability` on every data route/mutation (R8)
SOX-style segregation of duties / tamper-evidence	Effort not unilaterally editable	`total_hours` derived and recomputable from referenced sessions (R6)
Event sourcing / append-only ledgers	Reconstructable history	The activity ledger plus the Task FSM transitions provide a reconstructable trail

13. Evaluation Methodology

The numbers below are **illustrative**; the contribution is the *methodology* for evaluating a Task-as-hub discipline, which any adopter can run on their own data.

Dimension	Metric	How to measure	Interpretation
Concurrency safety	Double-holder rate	Count Tasks ever simultaneously `claimed_by` two principals (should be 0)	0 ⇒ arbitration sound
Conflict cost	Repo-write collisions per 1k tasks	Compare branch-ownership baseline vs Task-arbitration	Lower ⇒ branch displacement working
No-double-entry	Duplicate-body count	Count request bodies stored in >1 table	0 ⇒ invariant holds
Effort fidelity	|`total_hours` − Σ(session durations)|	Recompute from `related_event_ids`	0 ⇒ rollup tamper-free
Idempotency	Tasks per ask	Group by `origin_ask_ref`	1 ⇒ intake idempotent
Legibility	"Time to answer" for *what/how/output/effort*	Stopwatch a reviewer answering all four from the hub	Lower ⇒ legibility gain
Throughput	Claims/sec under contention	Load-test concurrent `claimTask`	Bounded by row-lock; should degrade gracefully

Illustrative result (synthetic, *not* a measurement): in the bundled `self-check.js`, double-holder rate = 0, duplicate-body count = 0, and |`total_hours` − Σ| = 0 across the scenario — exactly the values the methodology expects when the invariants hold. Adopters should replace these with real production figures.

14. Novelty & Inventive Claims

The following are stated in prose as a public, enabling disclosure — **not** as asserted patent claims. They define the contributed technique precisely so it is established as prior art.

Claim 1 (independent). A computer-implemented method for coordinating a workforce comprising both human actors and a plurality of autonomous software agents operating on a shared code repository, the method comprising: maintaining, in a relational database, a single mutable task record as the authoritative owner of a unit of work for both human and agent actors, in lieu of per-agent version-control branch isolation; creating exactly one said task record per intake ask via a single idempotent entry point that stores a reference to the ask's originating system-of-record rather than copying its body; binding each actor's work session, progress note, resolution note, effort-rollup record, and activity-ledger entry to said task record by foreign-key reference rather than by duplication, such that each said artifact entity has exactly one owning table; arbitrating concurrent write ownership among the actors at task-record write time by acquiring a row-level lock on the task record and applying a finite-state lifecycle guard, whereby a conflicting concurrent claim is detected and serialized by the task record before any write to the shared code repository occurs; gating each subsequent repository write on a fencing token issued with the granted claim; and deriving an effort total for the unit of work by mechanically aggregating the durations of the foreign-key-referenced work-session calendar events, without manual entry.

Claim 2. The method of claim 1, wherein the single idempotent entry point is keyed on an origin-ask reference such that re-delivery of the same ask returns the existing task record rather than creating a second one.

Claim 3. The method of claim 1, wherein the task record carries an `actor_type` attribute partitioning human and agent populations within one schema without forking the data model.

Claim 4. The method of claim 1, wherein the finite-state lifecycle comprises states open, in_progress, blocked, complete, and cancelled, and a transition guard rejects any transition not in the legal transition set.

Claim 5. The method of claim 1, wherein arbitrating concurrent write ownership comprises executing the claim and the state transition atomically within a single database transaction holding a row-level lock on the task record.

Claim 6. The method of claim 1, wherein the fencing token is monotonically increasing and a repository-write gate denies any write whose presented token does not equal the task record's current claim token, thereby preventing a stale claim holder from writing.

Claim 7. The method of claim 1, wherein a dispatcher assigns a next available unit of work by selecting an unclaimed task record using a lock that skips already-locked rows, preventing two agents from claiming the same record.

Claim 8. The method of claim 1, wherein the effort-rollup record stores arrays of references to the work-session event ids and note ids it aggregates, and the effort total is recomputable from those referenced ids for audit.

Claim 9. The method of claim 1, wherein the unit of work is owned by a set of assignees comprising any of one or more agents, one or more humans, a group, or a mix thereof, and the task record transitions to complete when a configured group-completion rule over the assignees' resolution notes is satisfied.

Claim 10. The method of claim 1, further comprising recording each interaction in an append-only activity ledger row whose payload carries a reference to the task record, such that the ledger joins back to the hub.

Claim 11. The method of claim 1, further comprising gating each read of a cross-agent accountability view and each write to the hub on a capability check, wherein the cross-agent view filters to agent-authored rows and never returns human-private rows.

Claim 12. The method of claim 1, wherein the binding columns are added additively to a pre-existing human-only system as nullable or defaulted columns, and legacy rows backfill to default values such that legacy human workflows are unchanged.

Claim 13. The method of claim 1, wherein the task record optionally references exactly one of a project association or a process association, both references being null for a standalone unit of work.

Claim 14. The method of claim 1, wherein the effort-rollup is idempotent on the task reference such that re-execution recomputes the effort total from the referenced events without creating a duplicate rollup record.

Claim 15. The method of claim 1, wherein each behavior-controlling parameter — including a master enablement flag, a default view, an eligible-intake-channel set, the group-completion rule, and an effort-rollup cadence — is supplied by external configuration rather than hard-coded.

Claim 16. A system comprising a processor and a database configured to perform the method of any of claims 1–15, the database storing the task record, an assignee set, work-session events, progress and resolution notes, an effort-rollup record, and an activity ledger, each non-task entity referencing the task record by foreign key.

---

15. Limitations & Threats to Validity

- **Element-level commodity.** Each building block (work-item-as-hub, omnichannel intake, calendar→timesheet, row-locking, normalization) is well known. The contribution is the *specific combination* plus write-time arbitration that displaces the branch; we publish defensively rather than claim a strong patent.
- **Closest prior art.** STORM/ESAA and shared-state-manager patents (US 9,020,885 / US 8,375,086) are adjacent to write-time arbitration on shared state; the delta is the *Task-as-hub-not-branch* framing with the no-double-entry model around it (§5, Appendix A).
- **Database-centric assumption.** The arbitration guarantees assume a relational store with row-level locking and transactions; a different substrate would need an equivalent serialization primitive.
- **Scope boundary.** Project/process internals are intentionally out of scope; the Task only carries a reference. Version control still holds the code — the Task governs *who may write*, not the code itself.
- **Illustrative numbers.** All metrics in §13 are illustrative; real evaluation requires production data.
- **Intentionally withheld.** No internal infrastructure, deployment topology, or proprietary code is disclosed; the reference in `src/` is clean-room.

16. Future Work & Open-Source Reference App

Planned: a small **open-source reference application** ("task-hub") implementing the discipline end to end — intake adapters, the FSM + write-time arbitration, the id-reference-only schema, the mechanical rollup, and the dual human/AI accountability views — deployable to a generic Kubernetes/AKS cluster. See docs/OPEN-SOURCE-APP.md. Roadmap:

1. Reference schema + migrations (additive ALTER posture).
2. Arbitration core with Postgres `FOR UPDATE [SKIP LOCKED]` + fencing tokens.
3. Intake adapters (webhook, mailbox poll, dispatch).
4. Rollup job (`on-complete` and `scheduled` cadences).
5. Read-only human/AI accountability views.
6. Helm chart + generic `kubectl` deployment sketch (no cluster identifiers).

17. Conclusion

For a mixed human/AI-agent workforce, the version-control branch is the wrong ownership primitive and per-tool duplication is the wrong data model. Making a single database-backed **Task** record the authoritative owner of a unit of work — one that arbitrates concurrent writes through its own state machine *before* any repository write, and around which every artifact is an id-reference rather than a copy — yields one auditable owner per unit of work, eliminates branch-ownership conflicts structurally, and makes an AI persona's workday as legible as a human employee's, with effort summed mechanically rather than hand-keyed. This document is published as dated prior art so the technique remains free for all to practice.

Appendix A — Prior-Art Landscape

A.1 Well-trodden (commodity) ground

- Work-item-as-hub with branch/PR linking (Jira, Azure DevOps).
- Omnichannel intake into a single record (ServiceNow).
- Calendar-derived timesheets (TimeCamp, Toggl; US 9,659,260 B2).
- Mixed human/agent system of record (Workday ASOR).
- Row-level write serialization (`SELECT … FOR UPDATE`, `SKIP LOCKED`, advisory locks); fencing tokens for stale-holder prevention.
- Relational normalization / single-source-of-truth modeling (3NF).
- Shared-state managers for concurrent processes (US 9,020,885 / US 8,375,086).
- Multi-agent shared-state coordination and event sourcing (STORM, ESAA).
- Fleets of automated code-change agents (Google Rosie; Ziftci et al., FSE 2025).

A.2 Candidate-novel (the asserted delta)

The *combination*: a single Task record that simultaneously (a) owns a **mixed** human/AI unit of work, (b) enforces an **id-reference-only no-double-entry** invariant across calendar, notes, timesheet, and

ledger, and (c) **arbitrates concurrent agent writes at record-write time in lieu of git-branch ownership**, with a fencing-token-gated repository write and **mechanically derived** effort.

A.3 Honesty attestation

The prior-art search supporting this document is **directional, not exhaustive**. It reflects a good-faith review of named products, papers, and patents as of the publication date. No formal USPTO/EPO full-text claim-chart search was performed. This disclosure makes no assertion that any element is individually novel; it is published precisely because the realistic value is **freedom to operate**, not a patent grant. Readers should perform their own searches before relying on patentability conclusions.

Appendix B — Glossary

Term	Definition
Ask	The entry event — a request to do work, on any channel. Not itself a stored entity in this discipline; it triggers Task creation.
Task	The hub record that authoritatively owns a unit of work.
Actor	A human or an autonomous AI persona that performs work.
Persona	A named autonomous AI agent treated as a first-class principal.
Assignee	The actor(s)/group bound to a Task.
Work session	A time-blocked calendar event with `task_id` set; the worked hour range.
Progress / resolution note	A note referencing the Task; many progress, one resolution per assignee.
Claim / arbitration	The write-time act of acquiring Task ownership, serialized by row lock + FSM.
Fencing token	A monotonically increasing token issued with a granted claim; gates repository writes against stale holders.
No-double-entry invariant	Each entity has one owning table; cross-entity relationships are id references, never copies.
Rollup	The mechanical aggregation of work-session durations into a timesheet total.
AI view	A read-only, capability-gated, cross-persona accountability surface.

Appendix C — Reference-Implementation Index

Path	Purpose
`src/README.md`	What the reference shows, how to run it, clean-room disclaimer
`src/task-hub.js`	Task hub: FSM, write-time `claimTask` arbitration, fencing tokens, id-reference stores, mechanical `rollupTimesheet`
`src/demo.js`	Runnable end-to-end scenario (two agents race; holder works; rollup)
`src/self-check.js`	Assertions verifying R1, R3, R4, R5, R6

Appendix D — Defensive-Publication Deposit & Timestamp

- **Publication date:** 2026-06-25.
- **Publisher:** Gus IT LLC (Florida, USA).
- **Author:** Gustavo Assuncao, PhD.
- **Deposit channel:** to be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is asserted here.
- **Intent:** this disclosure is intentionally public to **bar later patenting of the described techniques by others** and to keep them freely practiceable. The public Git commit history of this repository provides an additional independent timestamp.