what idf we were to make task agentic, actual sub agent capable of performing their own actions based on inputs and events? i woudnt need the pipeline agfent , the orchestrator agent coulod just spawn tasjk agents who move through the process... what do yoou think?

Thought for ...

Yeah, this is a really interesting pivot — and it actually fits your whole "agentic ops" story *better* than a big monolithic Pipeline Orchestrator.

Let me break it down in plain terms.

✅ What you're proposing

Instead of:

- **One Pipeline Orchestrator Agent** that:
  - watches all tasks
  - moves them through stages
  - assigns people
  - enforces SLAs

You'd have:

- **One Orchestrator Agent** that:
  - classifies intake
  - builds the `TaskDefinition`
  - decides initial pipeline/stage/assignee
  - then **spawns a Task Agent** for that task
- **Many Task Agents (one per task)** that:
  - listen to events for *their* task
  - move the task through stages
  - call tools / APIs / send emails
  - decide when to wait for human vs continue

    - stop when the task is done or overridden

Pipelines become **configuration + views**, not the “thing doing the work.”

Conceptually this is 🔥 .

👍 Why this is attractive

1. Encapsulation: each task owns its own logic

- The “logic for how a billing question behaves” lives with the **BillingQuestionTaskAgent**, not some giant switch inside a pipeline agent.
- You can specialize behavior heavily:
    - `BookingTaskAgent` , `SupportTicketAgent` , `BillingDisputeAgent` , etc.
- You can version them:
    - `SupportTicketAgent:v2` handles escalations differently; existing tasks can stay on v1 if you want.

2. Orchestrator becomes simpler and more composable

The Orchestrator does basically:

1. Classify intake → build `TaskDefinition` + initial `PipelineAssignment`
2. Spawn / invoke the right Task Agent template
3. Monitor / supervise at a high level (health, stuck tasks, anomalies)

It becomes **“router + supervisor”**, not “brain of everything.”

3. Fits the mental model: “This task is working on itself”

For users (and marketing), this is easy to sell:

> “Every task in AstralisOps is an intelligent agent trying to complete itself, and only bothers humans when necessary.”

That’s way more on-brand for you than “we have a pipeline processor.”

👎 The catches / things you have to get right

1. You *don’t* want a literal always-on agent process per task

If you naïvely do:

- one long-lived LLM loop per task
- constantly polling / thinking

…you'll nuke cost, complexity, and observability.

Better pattern:

> **Event-driven, stateless-ish Task Agents.**

- Task Agent = *function/class/handler* that:
  - is invoked when a relevant event happens (`task.created` , `task.status_changed` , `task.sla_breached` , etc.)
  - loads the current `TaskDefinition`
  - decides "what's the next action?"
  - performs that action (call n8n, send email, update stage, etc.)
  - exits

So "agentic" = *behavior + policy per task*, not *daemon per task*.

2. You still need some global view

Some things are inherently global:

- Load balancing across staff
- Org-level priorities ("all billing tasks > low-priority support")
- Rate limiting, quotas, scheduling windows

You can still do this with agentic tasks, but you need:

- a **Global Supervisor / Scheduler** (could be the same Orchestrator), that:
  - looks across tasks
  - decides which ones are allowed to act now
  - triggers their "next step" events

So: **task agents decide what they want**, supervisor decides ***when*** they get CPU.

3. Debuggability and replay

Many tiny agents makes debugging harder unless you're strict about:

- event logs (every decision is an event)
- deterministic or at least re-playable flows

- strong correlation IDs per task

You're already event-based, so you're in a good place for this—just lean in.

🧠 How I'd structure it (concrete)

Keep what we already defined (TaskDefinition, events, default pipelines), but change *who* owns what:

Orchestrator Agent

Responsibilities:

1. On `intake:created` :
    - build `TaskDefinition` from intake + template
    - choose initial pipeline/stage/assignee
    - emit `task.created + pipeline.item_added`
    - register which Task Agent to use (e.g. `type = "SUPPORT_REQUEST" → SupportTaskAgent`)
2. On `task.reprocess_requested` :
    - re-run classification
    - possibly switch task type + task agent

It **does not** move tasks across stages day-to-day.

Task Agent (per type)

Example: `SupportTaskAgent`

You can implement this as a simple module:

```
// pseudo export const SupportTaskAgent = { onTaskCreated(event) { ... }, onStatusChanged(event) { ... }, onSlaBreached(event) { ... }, onCustomerReplied(event) { ... }, };
```

It reacts to:

- `task.created` (for its type)
- `task.status_changed`

- `task.sla_breached`
- domain events like `message.inbound` , `doc.processed` , etc.

Typical behaviors:

- Move from `new` → `in_progress` when first picked up
- Move to `awaiting_customer` when response sent
- If `sla_breached` while `awaiting_customer` , maybe ping client again or flag to human
- Close ticket automatically if unanswered for X days

Each agent:

- knows required steps (from `TaskDefinition.steps`)
- knows timeline expectations (`typicalMinutes`)
- decides when to call tools, change stages, or notify humans

Pipelines in this model

Pipelines become:

- **configuration**:
  - which stages exist
  - which statuses map to which column
  - stage-specific rules (e.g. “only ADMIN can move into `escalated` ”)
- **projections / views**:
  - queries over tasks to show columns in the UI

They are no longer “brains.”
The *Task Agent* is the brain, and the pipeline is the *board*.

That aligns with your earlier instinct that the kanban view should be a **control tower**, not the engine.

🔚 My honest recommendation

If we summarize:

- **Yes**, agentic tasks is a **better conceptual model** for AstralisOps than a single pipeline agent.
- But implement it as:
  - **event-driven Task Agents** (modules/handlers per task type),
  - not long-lived, free-roaming LLM loops.
- Keep:
  - Orchestrator: intake → task creation → agent selection → global policies
  - Pipelines: configuration + UI view
- Let:
  - Each Task Agent own the logic for "how this task advances itself."

If you want, next step I can:

- sketch a `TaskAgentRegistry` + interface, e.g.:

```
interface TaskAgent { supports(task: TaskDefinition): boolean;
handleEvent(event: TaskEvent | DomainEvent): Promise<void>; }
```

and show how the Orchestrator would route events into the right agent(s) with minimal ceremony.

no, let re generate the whole plan from top to bottom, i still want the task defined by json structures, my thinking is we have a base agent that is able to handle any of the tasks we have defined, tghe json struture provides its system prompt etc.. i also wonder if there is a way to cache the agents or their actions using an audit log? if ever decision has an id, if we push those id into an array it becomes replayable for audits. bu if we also cache them then they become reusable too, no?

Thought for ... >

Got it, let's wipe the slate and design this clean from the ground up with:

- **JSON-defined tasks**

- **One Base Agent** that can handle *any* task type
- **Decision log** for **audit + replay**
- **Caching / reuse** of agent decisions where it makes sense

I'll keep this tight but end-to-end.

1. High-Level Architecture

Flow:

1. **Intake** (form/email/chat/api/call) → `IntakeEvent`
2. **Orchestrator**:
   - Classifies intake (AI)
   - Chooses a **TaskTemplate JSON**
   - Instantiates a **TaskInstance JSON**
   - Emits `task.created`
3. **Base Task Agent**:
   - Triggered by events (`task.created`, `task.status_changed`, `task.sla_breached`, etc.)
   - Reads TaskInstance JSON + AgentConfig from template
   - Decides the **next action**
   - Calls tools (send email, update pipeline, assign staff, etc.)
   - Writes to **DecisionLog** (for audit/replay)
4. **Pipelines UI**:
   - Just a **projection** over tasks (stages/status/assignee/SLA)
   - Not a brain, just the board

So: **Orchestrator + Base Agent + JSON templates + event bus**.

2. Core JSON Structures

2.1 TaskTemplate (what you define once)

This is your "playbook" per task type.

```
{ "id": "BOOKING_REQUEST_V1", "label": "Booking Request",
"applicableSources": ["FORM", "EMAIL", "CHAT", "CALL"], "category":
"SALES_INQUIRY", "department": "Sales", "staffRole": "AGENT",
"typicalMinutes": 240, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "sales-tasks", "defaultStageKey": "new_intake"
}, "steps": [ { "id": "capture-details", "label": "Verify contact &
request", "order": 1 }, { "id": "propose-slots", "label": "Send
available time slots", "order": 2 }, { "id": "confirm-booking", "label":
"Confirm appointment", "order": 3 } ], "agentConfig": { "systemPrompt":
"You are a task agent handling booking requests for a small
business...", "allowedActions": [ "SET_STATUS", "SET_STAGE",
"ASSIGN_STAFF", "PING_CUSTOMER", "ADD_INTERNAL_NOTE" ],
"completionCriteria": { "status": "done", "requiredStepsCompleted":
["capture-details", "confirm-booking"] } } }
```

2.2 TaskInstance (what's created per intake)

```
{ "id": "task-123", "templateId": "BOOKING_REQUEST_V1", "orgId": "org-
123", "source": "FORM", "sourceId": "form-submission-999", "title":
"Need to book a consultation", "description": "Looking for a consult
next week...", "category": "SALES_INQUIRY", "department": "Sales",
"staffRole": "AGENT", "priority": 3, "status": "new", "stageKey":
"new_intake", "pipelineKey": "sales-tasks", "timeline": {
"typicalMinutes": 240, "startedAt": null, "dueAt": null, "breachedAt":
null }, "steps": [ { "id": "capture-details", "status": "new" }, { "id":
"propose-slots", "status": "new" }, { "id": "confirm-booking", "status":
"new" } ], "attachments": [], "data": { "customerName": "Jane Doe",
"preferredDays": ["Monday", "Wednesday"] }, "tags": ["booking", "form"],
"assignedToUserId": null, "override": { "overridden": false, "reason":
null, "byUserId": null, "at": null }, "agentState": { "lastDecisionId":
null, "decisionIds": [] }, "createdAt": "2025-11-24T20:00:00Z",
"updatedAt": "2025-11-24T20:00:00Z" }
```

> 🔑 Note: `agentState.decisionIds` is your pointer into the **DecisionLog** for replay/audit.

2.3 DecisionLogEntry (for audit + replay)

Each time the Base Agent thinks and acts, it appends one of these:

```
{ "id": "dec-001", "taskId": "task-123", "orgId": "org-123",
"templateId": "BOOKING_REQUEST_V1", "eventName": "task.created",
"eventId": "evt-abc", "inputSnapshot": { "status": "new", "stageKey":
"new_intake", "steps": [ { "id": "capture-details", "status": "new" }, {
"id": "propose-slots", "status": "new" }, { "id": "confirm-booking",
"status": "new" } ] }, "agentConfigHash": "hash-of-template-and-config",
"llmCall": { "model": "gpt-4.1-mini", "promptType":
"DECIDE_NEXT_ACTION", "tokensIn": 512, "tokensOut": 96 }, "decision": {
"actions": [ { "type": "SET_STATUS", "toStatus": "in_progress" }, {
"type": "ASSIGN_STAFF", "strategy": "LEAST_BUSY_IN_ROLE", "role":
"AGENT" } ], "notes": "Start processing and assign to a sales agent." },
"appliedAt": "2025-11-24T20:01:00Z" }
```

- **Audit:** you can show this entire chain in the UI: "here's why this moved."
- **Replay:** reapply all `decision.actions` in order to reconstruct state.

You don't cache "agents" as objects; you cache **their decisions** and potentially **their reusable plans**.

3. Base Task Agent Design

3.1 Core behavior (pseudocode)

```
async function handleEventForTask(event: TaskEvent | DomainEvent) {
const task = await loadTaskInstance(event.taskId); const template =
await loadTaskTemplate(task.templateId); if (task.override.overridden)
return; // human owns it now const agentConfig = template.agentConfig;
const relevantDecisions = await loadRecentDecisions(task.id); // e.g.
last 5–10 const context = { task, template, agentConfig, event,
recentDecisions: relevantDecisions }; const decision = await
callLLMToDecideNextAction(context); const decisionId = await
appendDecisionLog(task.id, decision, context); const updatedTask = await
applyDecisionToTask(task, decision, decisionId); await
emitEventsFromDecision(task, updatedTask, decision); }
```

- **Single Base Agent**

  Same code path for all tasks.

  Behavior differences come from:

  - `template.agentConfig.systemPrompt`
  - `allowedActions`
  - `completionCriteria`

- domain-specific fields in `TaskInstance.data`

4. Replay, Audit, and Caching

4.1 Replay

To replay a task:

1. Load its **initial snapshot** (or reconstruct from `task.created`).
2. Load all `DecisionLogEntry` sorted by `appliedAt` .
3. For each entry, apply `decision.actions` in order (idempotent transforms).
4. You get final state **without** re-calling the LLM.

This is:

- Compliance/audit friendly
- Debuggable: “why is this in `needs_review` ?” → inspect decisions.

4.2 Caching / Reuse

You don’t really cache “agent instances”; you cache **derived artifacts**:

a) Action plans per template

First time the agent sees a new template/event combo, it might generate a **plan** like:

> “For BOOKING_REQUEST on `task.created` , these are the default 3 actions.”

You can store this as:

```
{ "cacheKey": "BOOKING_REQUEST_V1|task.created|plan", "actions": [ {
"type": "SET_STATUS", "toStatus": "in_progress" }, { "type":
"ASSIGN_STAFF", "strategy": "LEAST_BUSY_IN_ROLE", "role": "AGENT" } ] }
```

Then:

- For future tasks of `BOOKING_REQUEST_V1` on `task.created` , you can:
  - **skip the LLM**, apply cached actions directly, or
  - call LLM with the cached plan as a suggestion and ask: “confirm or modify?”

b) Decision pattern caching

You can also summarize decisions into "patterns":

- "If `SUPPORT_REQUEST` and priority >= 4 and category = BUG → do X, Y, Z"

This effectively becomes **automated rule extraction** from repeated LLM decisions.

c) Input hashing

Each `DecisionLogEntry` already has `agentConfigHash`.
You can also hash:

- task template
- event type
- coarse classification of input

Then you can create a key like:

```
support_request_v2|task.created|hash(important-fields)
```

to look up **prior similar decisions** and either:

- replay them exactly, or
- feed them as exemplars in the next LLM call.

5. Where Pipelines Fit in This Model

Pipelines are **still valuable**, but now:

- They are **configuration + UI**:
  - pipeline definitions (stages, labels, colors, terminal flags)
  - mapping of `stageKey` → column
- They are updated by **decisions**:
  - `SET_STAGE` actions → update `task.stageKey` and emit `pipeline.item_moved`
- They surface:
  - SLA state (`timeline` vs `typicalMinutes`)
  - `overridden` status (human vs agent-controlled)
  - decision history link

So your kanban board:

- Visualizes **TaskInstance state**
- Uses `agentState.decisionIds` to let you open the **audit / decision timeline**
- Shows visual cues from decisions (e.g. "auto-assigned by agent", "escalated by agent").

6. Answering Your Core Question Directly

> *"If every decision has an id and we push those into an array, it becomes replayable for audits. But if we also cache them then they become reusable too, no?"*

Yes – exactly.

- **Array of decisionIds per task** →
  replayable, fully auditable history.
- **DecisionLog as a global store** →
  you can mine it for reusable plans / patterns:
  - by template
  - by event type
  - by input characteristics

Just be intentional:

- Use caching for **repeatable patterns** (e.g. standard booking behavior)
- Still allow the LLM to adapt for:
  - edge cases
  - high priority / escalated items
  - novel types

If you want, next I can:

- Define a **TaskTemplate JSON library** (5–10 templates ready to drop in), or
- Draft the **BaseAgent LLM prompt** that reads `TaskInstance + template + decisions` and outputs a structured `decision` object.

yes both then whats missing?

Alright, let's lock this in.

You asked for:

1. A **TaskTemplate JSON library** (5–10 templates)
2. A **Base Task Agent LLM prompt**
3. Then: **"what's missing?"** from the whole design

I'll do all three.

1️⃣ TaskTemplate JSON Library (drop-in examples)

Shape recap (for consistency):

```
interface TaskTemplate { id: string; // e.g. "BOOKING_REQUEST_V1"
label: string; applicableSources: string[]; // ["FORM","EMAIL",...]
category: string; // SALES_INQUIRY, SUPPORT_REQUEST... department:
string; staffRole: string; typicalMinutes: number; defaultPriority:
number; // 1–5 pipeline: { preferredPipelineKey: string; // "sales-
tasks" defaultStageKey: string; // "new_intake" }; steps: { id: string;
label: string; order: number }[]; agentConfig: { systemPrompt: string;
allowedActions: string[]; completionCriteria: { status: string;
requiredStepsCompleted?: string[]; }; }; }
```

1) Booking Request (Sales)

```
{ "id": "BOOKING_REQUEST_V1", "label": "Booking Request",
"applicableSources": ["FORM", "EMAIL", "CHAT", "CALL"], "category":
"SALES_INQUIRY", "department": "Sales", "staffRole": "AGENT",
"typicalMinutes": 240, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "sales-tasks", "defaultStageKey": "new_intake"
}, "steps": [ { "id": "capture-details", "label": "Verify contact &
request", "order": 1 }, { "id": "propose-slots", "label": "Send
available time slots", "order": 2 }, { "id": "confirm-booking", "label":
"Confirm appointment & send confirmation", "order": 3 } ],
"agentConfig": { "systemPrompt": "You are a task agent handling booking
requests for a small business. Your job is to make sure the client is
contacted, suitable time slots are proposed, and an appointment is
confirmed and recorded. You coordinate between the automation system and
human agents, and only escalate when needed.", "allowedActions": [
"SET_STATUS", "SET_STAGE", "ASSIGN_STAFF", "PING_CUSTOMER",
"ADD_INTERNAL_NOTE" ], "completionCriteria": { "status": "done",
"requiredStepsCompleted": ["capture-details", "confirm-booking"] } } }
```

2) Support Request (Support)

```
{ "id": "SUPPORT_REQUEST_V1", "label": "Support Request",
"applicableSources": ["FORM", "EMAIL", "CHAT", "API"], "category":
"SUPPORT_REQUEST", "department": "Support", "staffRole": "AGENT",
"typicalMinutes": 120, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "support-tasks", "defaultStageKey": "new_intake"
}, "steps": [ { "id": "triage", "label": "Triage & categorize issue",
"order": 1 }, { "id": "investigate", "label": "Investigate / reproduce",
"order": 2 }, { "id": "respond", "label": "Respond with resolution or
workaround", "order": 3 } ], "agentConfig": { "systemPrompt": "You are a
task agent handling support requests. You triage issues, help route them
to the correct person, track investigation progress, and ensure the
customer receives a clear response. Escalate when the problem is severe
or cannot be resolved quickly.", "allowedActions": [ "SET_STATUS",
"SET_STAGE", "ASSIGN_STAFF", "TAG_TASK", "PING_CUSTOMER", "ESCALATE" ],
"completionCriteria": { "status": "done", "requiredStepsCompleted":
["triage", "respond"] } } }
```

3) Billing Question (Billing)

```
{ "id": "BILLING_QUESTION_V1", "label": "Billing Question",
"applicableSources": ["FORM", "EMAIL"], "category": "BILLING_QUESTION",
"department": "Billing", "staffRole": "BILLING_SPECIALIST",
"typicalMinutes": 180, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "billing-tasks", "defaultStageKey": "new_intake"
}, "steps": [ { "id": "verify-account", "label": "Verify customer
account & invoice", "order": 1 }, { "id": "resolve-issue", "label":
"Resolve billing discrepancy or explain", "order": 2 }, { "id":
"confirm-customer", "label": "Confirm with customer", "order": 3 } ],
"agentConfig": { "systemPrompt": "You are a task agent for billing
questions. You make sure the issue is matched to the right billing
specialist, that account and invoice details are verified, and that the
customer receives a clear explanation or correction.", "allowedActions":
[ "SET_STATUS", "SET_STAGE", "ASSIGN_STAFF", "TAG_TASK",
"ADD_INTERNAL_NOTE", "PING_CUSTOMER" ], "completionCriteria": {
```

```
"status": "done", "requiredStepsCompleted": ["verify-account", "resolve-
issue"] } } }
```

4) Document Review / Approval (Ops / Internal)

```
{ "id": "DOCUMENT_REVIEW_V1", "label": "Document Review",
"applicableSources": ["FORM", "EMAIL", "API"], "category":
"SUPPORT_REQUEST", "department": "Operations", "staffRole": "ADMIN",
"typicalMinutes": 720, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "internal-tasks", "defaultStageKey": "backlog"
}, "steps": [ { "id": "check-completeness", "label": "Check document
completeness", "order": 1 }, { "id": "review-content", "label": "Review
and annotate content", "order": 2 }, { "id": "approve-or-change",
"label": "Approve or request changes", "order": 3 } ], "agentConfig": {
"systemPrompt": "You are a task agent managing document review tasks.
You ensure documents are complete, assigned to the correct internal
owner, and either approved or sent back with requested changes. You
track progress and deadlines.", "allowedActions": [ "SET_STATUS",
"SET_STAGE", "ASSIGN_STAFF", "ADD_INTERNAL_NOTE", "TAG_TASK" ],
"completionCriteria": { "status": "done", "requiredStepsCompleted":
["check-completeness", "approve-or-change"] } } }
```

5) Request More Info (Sales / General)

```
{ "id": "REQUEST_MORE_INFO_V1", "label": "Request More Information",
"applicableSources": ["FORM", "EMAIL", "CHAT"], "category": "GENERAL",
"department": "Sales", "staffRole": "AGENT", "typicalMinutes": 1440,
"defaultPriority": 2, "pipeline": { "preferredPipelineKey": "sales-
tasks", "defaultStageKey": "new_intake" }, "steps": [ { "id": "clarify-
needs", "label": "Clarify client needs", "order": 1 }, { "id": "send-
materials", "label": "Send materials or links", "order": 2 } ],
"agentConfig": { "systemPrompt": "You are a task agent handling
information requests. Your job is to clarify what the client needs, make
sure the right information or links are sent, and ensure someone follows
up if necessary.", "allowedActions": [ "SET_STATUS", "SET_STAGE",
"ASSIGN_STAFF", "PING_CUSTOMER", "ADD_INTERNAL_NOTE" ],
"completionCriteria": { "status": "done", "requiredStepsCompleted":
["clarify-needs", "send-materials"] } } }
```

6) Internal Follow-up (Internal)

```
{ "id": "INTERNAL_FOLLOWUP_V1", "label": "Internal Follow-up",
"applicableSources": ["FORM", "API", "EMAIL"], "category": "INTERNAL",
"department": "Operations", "staffRole": "AGENT", "typicalMinutes":
1440, "defaultPriority": 2, "pipeline": { "preferredPipelineKey":
"internal-tasks", "defaultStageKey": "backlog" }, "steps": [ { "id":
"clarify-task", "label": "Clarify requested internal work", "order": 1
}, { "id": "do-work", "label": "Complete internal work", "order": 2 }, {
"id": "report-back", "label": "Report back or record outcome", "order":
3 } ], "agentConfig": { "systemPrompt": "You are a task agent for
internal follow-up items. You ensure the task is clearly defined,
assigned to the right internal person, and completed within a reasonable
timeframe.", "allowedActions": [ "SET_STATUS", "SET_STAGE",
"ASSIGN_STAFF", "ADD_INTERNAL_NOTE", "TAG_TASK" ], "completionCriteria":
{ "status": "done", "requiredStepsCompleted": ["do-work"] } } }
```

7) Generic Task (Fallback)

```
{ "id": "GENERIC_TASK_V1", "label": "Generic Task",
"applicableSources": ["FORM", "EMAIL", "CHAT", "API", "CALL"],
"category": "GENERAL", "department": "General", "staffRole": "AGENT",
"typicalMinutes": 720, "defaultPriority": 3, "pipeline": {
"preferredPipelineKey": "generic-tasks", "defaultStageKey": "new_intake"
}, "steps": [ { "id": "understand", "label": "Understand what is
needed", "order": 1 }, { "id": "act", "label": "Take appropriate action
or assign", "order": 2 }, { "id": "confirm", "label": "Confirm with
requester if needed", "order": 3 } ], "agentConfig": { "systemPrompt":
"You are a general-purpose task agent. When a task does not match a more
specific template, you help clarify what is needed, route it
appropriately, and ensure progress is made.", "allowedActions": [
"SET_STATUS", "SET_STAGE", "ASSIGN_STAFF", "TAG_TASK", "PING_CUSTOMER",
"ADD_INTERNAL_NOTE" ], "completionCriteria": { "status": "done" } } }
```

This is enough to cover your 5 pipelines + fallback.

2️⃣ Base Task Agent LLM Prompt

2.1 Input context to the LLM

You'd pass something like:

```
{ "task": { /* TaskInstance JSON */ }, "template": { /* TaskTemplate
JSON */ }, "event": { /* triggering event (task.created,
task.status_changed, etc.) */ }, "recentDecisions": [ /* 0..N
DecisionLogEntry summaries */ ] }
```

2.2 Output schema (Decision)

Have the LLM always respond in this shape:

```
interface AgentDecision { reasoning: string; // short natural language
trace actions: AgentAction[]; } type AgentAction = | { type:
"SET_STATUS"; toStatus: TaskStatus; // "new" | "in_progress" | ... } | {
type: "SET_STAGE"; toStageKey: string; // e.g. "qualification" } | {
type: "ASSIGN_STAFF"; strategy: "LEAST_BUSY_IN_ROLE" | "KEEP_EXISTING" |
"UNASSIGN"; role?: string; } | { type: "TAG_TASK"; add: string[];
remove?: string[]; } | { type: "PING_CUSTOMER"; channel: "EMAIL" | "SMS"
| "CHAT"; templateHint: string; // short hint for downstream template
system } | { type: "ADD_INTERNAL_NOTE"; note: string; } | { type:
"ESCALATE"; reason: string; targetRole?: string; } | { type: "NO_OP";
reason: string; };
```

2.3 System prompt (Base Agent)

You can tune this, but here's a solid starting point:

```
You are the AstralisOps Base Task Agent. You manage BUSINESS TASKS
represented as JSON. Each task is created from a Task Template and
contains: - metadata (type, category, pipeline, stage, priority,
department, staff role), - a list of steps, - a timeline with an
expected duration, - status and stage, - assignment info, - override
flags. You DO NOT chat with end users. You only decide what the
automation system should do next for a task. You must follow these
rules: 1. Think in small, safe steps. Prefer incremental progress over
large changes. 2. Respect the task template: - Only perform actions
listed in template.agentConfig.allowedActions. - Try to drive the task
towards template.agentConfig.completionCriteria. 3. Respect human
overrides: - If the task.override.overridden is true, DO NOTHING and
return a NO_OP action explaining why. 4. Respect status and stage
semantics: - status "done" or a terminal stage usually means the task is
complete. Prefer NO_OP unless something is clearly wrong. 5. Use the
event: - React only to what changed: e.g. task.created,
task.status_changed, task.sla_breached, message.inbound, etc. 6. Be
deterministic and auditable: - Use the existing task data and recent
decisions to be consistent. - Avoid random behavior. Make decisions that
can be explained. You must ALWAYS respond with a single JSON object of
```

```
the form: { "reasoning": "...short explanation...", "actions": [ {
"type": "...", ... } ] } No extra keys. No prose outside JSON. If you
want to do nothing, return a single NO_OP action with a clear reason.
```

2.4 User message template

For each call:

```
Here is the current TASK INSTANCE: ```json {taskJson}
```

Here is its TASK TEMPLATE:

```
{templateJson}
```

Here is the triggering EVENT:

```
{eventJson}
```

Here is a summary of recent DECISIONS for this task:

```
{recentDecisionsSummary}
```

Decide what should happen NEXT for this task, following the system instructions.

Return ONLY a valid AgentDecision JSON object.

```
That’s enough to plug directly into your LLM layer.

---

## 3️⃣ So… what’s still missing?

From a *product + architecture* perspective, here’s what’s left to make th

### A. **Concrete storage & access layer for templates + tasks + decisions
- We sketched schemas conceptually, but you still need:
  - DB tables/Prisma models for:
    - `TaskTemplate`
    - `TaskInstance` (or `Task`)
    - `DecisionLogEntry`
  - DAO methods:
    - `loadTaskInstance(taskId)`
```

~~~
    - `loadTaskTemplate(templateId)`
    - `appendDecisionLog(...)`
    - `applyDecisionToTask(...)`

(You already have a chunk of this from the previous Prisma zip — it just n

---

### B. **Event bus wiring**

We've defined events, but you still need:

- A **concrete dispatcher**:
  - e.g. `emitEvent(name, payload)` → writes to DB/Redis/n8n webhook queue
- A **Task Agent runner**:
  - subscribes to `task.*` events
  - fans them into the Base Agent
- Whether you're using:
  - in-process event emitter, or
  - Redis / DO managed MQ / n8n as the bus.

---

### C. **Action executor layer**

Right now, `AgentDecision.actions` is just data.

You still need an **executor** that:

```ts
async function applyDecisionToTask(task, decision) {
  for (const action of decision.actions) {
    switch (action.type) {
      case "SET_STATUS": ...
      case "SET_STAGE": ...
      case "ASSIGN_STAFF": ...
      case "PING_CUSTOMER": ...
      // etc
    }
~~~

```
    }
  }
```

And each case needs:

- DB updates
- Event emission (`task.status_changed` , `pipeline.item_moved` , etc.)
- Side-effect tools (email/SMS/n8n webhooks)

D. Override + reprocess UX

You've described the behavior; you still need:

- Actual UI pieces:
  - "Override automation" toggle on card
  - "Reprocess with AI" button
  - Visual indicator on cards: *agent-driven* vs *human-driven*
- API endpoints:
  - `POST /api/tasks/[id]/override`
  - `POST /api/tasks/[id]/reprocess`

E. Template configuration UI

Right now templates are JSON files.

To really ship this as a product:

- You will eventually want:
  - UI to manage Task Templates:
    - add/edit steps
    - change typicalMinutes
    - change allowedActions
  - Maybe versioning: `BOOKING_REQUEST_V1` , `V2` , etc.

Not required for v1, but it's on the horizon.

F. LLM infra decisions

You still need to decide:

- Which model(s) for:
  - classification (intake → template selection)
  - base task agent decisions
- When to:
  - use cached decisions / plans vs call LLM live
- Rate limits / cost protection:
  - maximum calls per task
  - backoff/retries
- Where to plug in:
  - OpenAI vs local vs pluggable provider

G. Instrumentation & metrics

For this to be defensible and improvable:

- Track:
  - number of tasks by type
  - average time to completion vs `typicalMinutes`
  - how often humans override
  - how many actions are NO_OP vs meaningful
  - cost per task (LLM usage)
- Use that to:
  - refine templates
  - add rules to bypass LLM where patterns are stable
  - tune your product story (“we reduce average response time by X%”)

H. Migration from your current pipeline UI

You’ll need:

- A mapping from your existing `pipelineItem` model → `TaskInstance` with `templateId`.
- A soft rollout:
  - Phase 1: agent only suggests actions (no auto move)
  - Phase 2: agent moves low-risk tasks automatically
  - Phase 3: full automation with overrides.

If you want, next I can:

- Adapt the **Prisma schema** from the last zip to this exact `TaskTemplate` / `TaskInstance` / `DecisionLog` model, and bundle it as another zip you can drop into the repo.