ChatGPT

Task Manager Specification for n8n Async Workflows

Introduction

Managing long-running or asynchronous tasks in n8n requires a coordinated approach to track task state and results outside the normal linear workflow execution. In scenarios like external API jobs (e.g. data processing, report generation) that don't return immediate results, a *task manager* can ensure these tasks are created, monitored, and updated independently. This specification outlines a **generic Task Manager system in n8n** that meets the requirements for task creation, status tracking (pending/in-progress/completed/failed), asynchronous polling of external APIs, retry and timeout handling, and status querying by task ID. We also include a Phase 2 plan for a dashboard/UI to visualize tasks, and we compare similar existing solutions (both within and outside n8n).

Requirements and Goals

The Task Manager system must fulfill the following key requirements derived from the prompt:

- **Task Creation with Unique ID:** Allow any workflow to create a task entry (with a globally unique Task ID) at the point where an asynchronous operation is initiated. The Task ID will be used to track progress.
- **Central Task Registry:** Maintain a centralized store (e.g. a database table or data store) of all tasks with their metadata and current status. This allows decoupling task state from the original workflow execution.
- **Task Status Lifecycle:** Support statuses including **Pending** (or queued), **In Progress**, **Completed**, and **Failed**. Transitions are driven either by external events or polling results.
- **Polling & Updates:** Provide a mechanism to poll external APIs or endpoints for each task (especially when no callback is available) and update the task's status and results accordingly.
- **Retry & Timeout Handling:** Implement retry logic for transient errors or failures when polling, with a finite number of attempts. Define timeouts or maximum wait durations for tasks – if a task exceeds a deadline or maximum retries without completion, mark it as **Failed** and trigger an alert.
- **Alerting:** Generate alerts or notifications for failed or timed-out tasks (e.g. via email, Slack, etc.) so that issues can be addressed promptly.
- **Status Query by ID:** Enable other workflows (or external clients) to easily check the status of a given task by its Task ID, without duplicating logic. This could be a sub-workflow or an API endpoint that returns the current status and any result available.
- **Generic & Extensible Design:** The system should be agnostic to specific APIs or task types – no hardcoded logic for particular services. Instead, it should store the necessary info to track different types of tasks (e.g. endpoints or instructions to poll) and handle them in a configurable way. This ensures new async tasks can be integrated without redesign.

- **Scalability & Performance:** The solution should handle multiple concurrent tasks and potentially many polling operations efficiently (e.g. batching or rate-limiting polls if needed). It should avoid blocking n8n's main workflow execution for long periods.
- **Phase 2 – Dashboard Integration:** Anticipate the addition of a user interface or dashboard that visualizes all tasks and their states. The design should expose necessary data (via database or API) to feed a UI, and possibly allow filtering or manual intervention if needed.

System Architecture & Workflow Design

To achieve the above, the Task Manager will consist of multiple pieces working together within n8n:

- **Task Data Store:** A persistent data store accessible to n8n (such as a database table, Airtable/Baserow, or even a custom key-value store) holds all task records. Each record includes a unique Task ID and fields like status, timestamps, external reference (job ID from API), and any info needed to poll for status. Using an external database is recommended since n8n has no built-in global datastore that can be updated by workflows (n8n provides "variables" but they are read-only in workflows, so an external DB is needed for cross-workflow state [1]). This central table acts as the **source of truth** for task states.

- **Task Creator (in workflows):** When a workflow needs to start a long-running job, it will: (1) call the external API to initiate the job, (2) generate a new Task ID (e.g. a UUID or unique string), and (3) create a new task entry in the Task Data Store with status "Pending" or "In Progress". This Task ID can then be returned or logged. For example, one could have an HTTP Webhook workflow that immediately responds with a generated job ID while the heavy process runs asynchronously – a pattern suggested by community users [2]. The task entry persists the job ID and initial status so it can be monitored independently. (Some implementations even use separate tables for status and results [2], but a single table can suffice with fields for result data or location.)

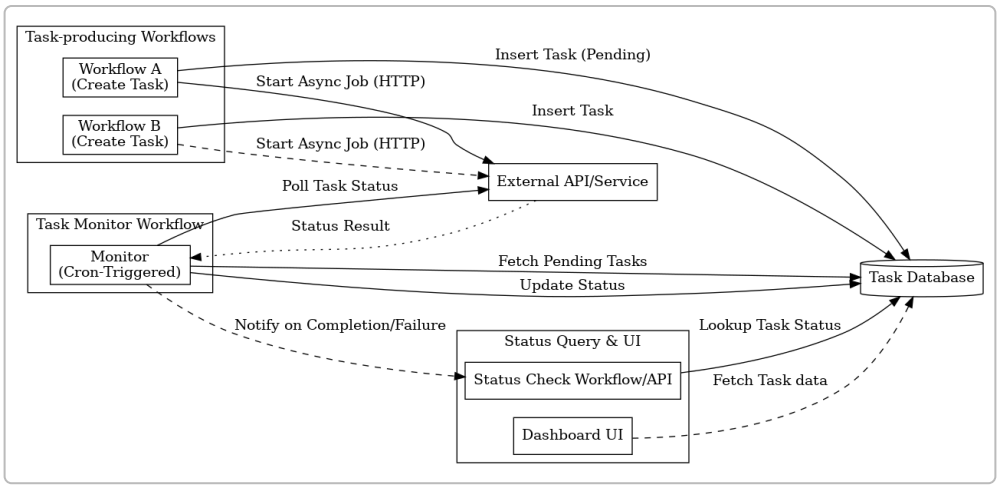
- **Central Monitoring Workflow:** A dedicated n8n workflow ("Task Monitor") runs on a schedule (using Cron or Interval trigger) to periodically check all outstanding tasks. On each run, it queries the Task Data Store for tasks that are not yet completed (status pending/in progress). It then polls the respective external APIs for updates on each task. This workflow encapsulates all logic for checking task statuses and updating the Task Data Store. Because it is separate, it runs in the background and does not block any user-facing workflows. The polling interval can be adjusted (e.g. every minute or every 5 minutes) depending on how responsive the external tasks need to be and API rate limits. We will include error handling here to implement retries and mark timeouts.

- **Status Query Mechanism:** We will provide a simple way for other workflows (or external clients) to query a task's status by ID. This could be done via an "Execute Workflow" node that calls a small sub-workflow (which reads the DB and returns status), or via a Webhook-based endpoint that returns the status in JSON. This allows any workflow to get the latest status of a task (for example, to decide whether to proceed or to inform a user). The key is that they only need the Task ID and don't need to know how the monitoring is done internally.

- **Notification/Alert Workflow:** Optionally, a separate workflow can be triggered for alerting on failures or timeouts. The monitor workflow could send events (via an n8n internal trigger or simply by updating a flag in the DB that an alert is needed) to another workflow that handles notifying the

relevant channel (email, chat, etc.). This keeps alert logic isolated. Alternatively, the monitor workflow itself can directly include nodes for sending alerts when it marks a task as failed.

The figure below illustrates how these components interact in n8n:



High-level architecture for the proposed n8n Task Manager. Workflows that initiate async jobs will insert a new task entry (with a unique ID and initial status) into a shared Task Database after triggering the external API. A dedicated Task Monitor workflow (running on a timer) periodically fetches pending tasks from the database and polls the external API/service for each task's status, then updates the task record (e.g. marking it Completed or Failed). Other workflows or a dashboard UI can query the Task Database (directly or via a small API workflow) to retrieve the current status of any given Task ID.

Task Data Storage & Unique IDs

Data Store: Use a reliable external storage for tasks. A SQL database (MySQL, PostgreSQL, etc.), a no-code database like Baserow/Airtable, or even a key-value store could work. The choice may depend on your environment – for self-hosted n8n, a SQL table is straightforward; if you prefer not to manage a DB, services like Airtable or n8n's built-in credentials datastore (via a custom node) might be used. The community consensus is that an external DB is the cleanest solution for sharing state between workflows [3] (many have used Supabase or similar for this [2]).

Task Table Schema: Design the table with columns to capture all relevant info for each task. For example:

- **task_id** (primary key): string or GUID, uniquely identifies the task (could be auto-generated by the DB or generated in n8n). This is the ID returned to clients and used for lookup.
- **status**: current status (e.g. "pending", "in_progress", "completed", "failed"). Define your status enums clearly. Initially it might be "pending" right after creation (or "in_progress" if the external job started immediately).
- **external_id**: identifier returned by the external API for the job (if applicable). For example, if you call an API that returns `jobId` or `taskId`, store it here – you will need it to poll status.
- **created_at** and **updated_at** timestamps.
- **last_checked**: timestamp of the last poll attempt (optional, for logging or adaptive polling).

- **poll_endpoint** (or **type** and other fields): information on how to check status. This could be a full URL to poll for status, or you might store a task "type" and have the monitor workflow know how to handle each type. A generic approach is to store the exact endpoint and method needed for status check. For example, store `poll_url`, `poll_method` (GET/POST), and maybe `poll_headers` or payload template if required. Then the monitor workflow can use these to make the HTTP call. This keeps the system generic – each task, when created, carries the instructions for how to poll its status.
- **result**: this could either be a JSON field or a separate table to store output data when completed. If the final result is small (like a URL or status message), it can be stored in the same record. If it's large (like a file or report content), you might store a link (e.g. an S3 URL) or use a separate "results" table linked by Task ID.
- **error_message**: if status is failed, optional field to record error details or reason for failure.
- **attempt_count**: integer count of how many times we've tried to poll (for retry logic).
- **max_attempts** or **timeout**: if you want per-task custom timeout or max retry, these fields could store those values at creation (or use defaults globally in the monitor logic).

With this schema, the Task Data Store centralizes all info. All workflows will *write* new tasks or *read* task status from this store, but only the monitor workflow will routinely *update* statuses. This ensures a clear single-writer principle to avoid race conditions.

Unique ID Generation: When creating a task, we need a unique Task ID. Possible strategies in n8n: use a **Function** node with JavaScript to generate a UUID (for example, using `crypto.randomUUID()` if available, or a short random string). Alternatively, if using a SQL database with an auto-increment primary key, you could insert the task and get back the generated ID. However, generating in n8n allows you to return it immediately (especially useful for a webhook that responds with the task ID). The ID should be unique across all tasks (to prevent collisions) and ideally non-guessable if exposed to users. A UUID or a combination of timestamp and random component works well. This Task ID is what clients or other workflows will use to reference the task.

Concurrency Consideration: Since multiple workflows could create tasks concurrently, ensure the ID generation method avoids collisions (again, GUIDs are safest). If using a DB with auto-increment, the DB will handle uniqueness. If generating in code, consider prefixing with something if needed to avoid any chance of overlap.

Task Creation in Workflows

To integrate task management, any workflow that initiates a long-running external process should incorporate a "task creation" step. Here's how a typical sequence would look in an n8n workflow for an async API call:

1. **Trigger External Task:** Use an **HTTP Request** node (or an app-specific node if available) to start the external process. For example, call an API endpoint `/start-long-job` – this might return a response immediately with a job reference (like `jobId`). Capture that response.
2. **Generate Task ID:** Immediately after, use a **Function** node (or Set node with expression) to create a unique Task ID. If the external API provided its own ID, you may combine them (for instance, your Task ID could include the external ID or you store both separately). Often it's simplest to use your own Task ID as the primary key and keep the external job ID in a field.

3. **Create Task Entry:** Add a node to insert a new record into the Task Data Store:
4. If using a SQL database: use the appropriate **Database node** (e.g. PostgreSQL node) with an INSERT query or configured fields. Map the Task ID, initial status, external job ID, and poll info into the query.
5. If using Airtable/Baserow: use their node to create a row with those fields.
6. If using a custom key-value (like Redis or AnyState), use an HTTP node or the specific integration to save the data. *(For example, the AnyState tool allows making a simple HTTP request to store a JSON value by key [4] – one could use Task ID as the key and store a JSON of the task info.)*
7. The initial status will likely be "pending" or "in_progress". If the external call *guarantees* the task has started, you might set "in_progress" immediately; otherwise "pending" until first poll confirms it.
8. Include the `poll_endpoint` (if applicable) or any metadata the monitor needs.
9. If implementing custom timeout per task, you could also set a `expires_at` (e.g. now + X hours) field here for the monitor to use.
10. **Return Task ID (if needed):** If this workflow was triggered by an API request (webhook from a user or system expecting a response), you can now respond with a payload containing the Task ID and perhaps a message like "Task scheduled" or "Processing started". This immediate acknowledgment decouples the client from the long processing. As one user described, their webhook responds with a job ID right away, and the actual processing then continues asynchronously [2]. This is crucial for not tying up HTTP connections or UI waiting times.

After these steps, the original workflow can end (or continue with other independent actions). The heavy lifting is now delegated to the Task Monitor workflow. Each created task is an entry the monitor will pick up on its next cycle.

Reusability: To avoid duplicating the task-creation logic in every workflow, consider abstracting it: - One approach is to create a **sub-workflow** (a workflow that is designed to be called by others via the **Execute Workflow** node) that handles the insertion of a task. You could pass it parameters like external ID, initial status, poll URL, etc., and it inserts the DB record and returns the new Task ID. Workflows would then just call this sub-workflow after starting an async job. However, note that n8n's Execute Workflow node by default waits for the sub-workflow to finish (synchronous call) [5], which is fine in this case since it's just a quick DB write. (If, on the other hand, one wanted truly asynchronous fire-and-forget sub-workflows, the trick is to call a webhook trigger of another workflow, which returns immediately [6] – but that's not necessary here.) - Alternatively, use n8n's **HTTP Request** node to call n8n's own **REST API** (if enabled) to create a task. For instance, if you built a custom REST endpoint (via an Express or cloud function) to insert tasks, you could call that. This is more complex and generally not needed if you can write to the DB directly in n8n. - For simplicity, including a "DB Insert" node in each workflow is okay, but make sure any change in task schema is updated in all workflows. A sub-workflow reduces maintenance overhead for changes.

By following this pattern, any number of workflows can create tasks and immediately free themselves from waiting. Each task record is effectively a placeholder that something is in progress externally. The unique Task ID ties together the request and its eventual result, even though they are handled in different workflows and timeframes.

Task Monitoring Workflow (Polling & Updates)

This is the core of the Task Manager's runtime. The **Task Monitor** workflow runs periodically to update task statuses. Let's break down its design and steps:

Trigger: Use an **Interval** or **Cron Trigger** node to schedule this workflow. The frequency depends on how quickly you want to detect task completions versus how many API calls you can afford. For example, a cron schedule of every 1 minute or every 5 minutes is common. *(For near-real-time updates, 30 seconds or less could be used, but be mindful of rate limits and n8n's capacity. If tasks typically take hours, a longer interval might suffice.)*

Fetch Pending Tasks: The first step inside is to retrieve all tasks that need monitoring. Use a **Database** node (or appropriate node for your data store) to query tasks where status is "pending" or "in_progress" (and optionally tasks that are not already marked failed). You might also filter out tasks that have exceeded a max retry or time (if you mark them failed separately). The result will be a list of task items to process.

- If no tasks match (result is empty), the workflow can simply end or do nothing until next trigger. This avoids unnecessary API calls.

Loop Through Tasks: We need to poll each task's external status. n8n doesn't have an explicit foreach loop node, but it automatically processes multiple items sequentially through nodes. Alternatively, you can use the **Split In Batches** node if you want to control how many tasks to handle at once (e.g. process 10 at a time, then loop, to avoid a flood of simultaneous requests). For simplicity, assume we let n8n iterate item by item.

Inside the loop for each task item: 1. **Determine Poll Endpoint:** If the task record contains the `poll_url` and details, use an **HTTP Request** node configured to use those fields. You can set the URL field to an expression like `{{$json["poll_url"]}}` which grabs the URL from the current item. Likewise set method (GET/POST) from a field if stored. If different tasks require different auth (e.g. API keys), you may need to store credentials or tokens per task or per type; the HTTP node could use them via expression as well. (An alternative is to use Switch nodes on a task "type" field: e.g. if type == "ServiceA", call ServiceA status endpoint; if type == "ServiceB", call ServiceB's node, etc. This works but adding new types requires editing the monitor workflow. A fully data-driven approach via a generic HTTP node is more scalable, at the cost of storing more info in the task record.) 2. **Call External API for Status:** The HTTP node calls the external service's status endpoint. For example, GET `/job-status/{external_id}`. The response is typically a JSON containing something like `status` ("completed" or "still_running" etc.), and possibly result data or a result URL. The HTTP node will output this data for the current item. 3. **Evaluate Response:** Use an **IF** node or a **Switch** node after the HTTP call to decide what to do based on the response: - If the status indicates the task is still not finished (e.g. "pending", "running", etc.), then we leave the task as is. We might update a "last_checked" timestamp in the DB, but status remains in progress. (Optionally, increment an attempt count.) - If the status indicates **Completed**, then proceed to mark the task as **completed** in our DB: - Prepare any result data. If the API returned the final result or a link to it, you might store it. (e.g. if it returned a URL to a file, save that in the `result` field; if it returned a short result message or some data, save it or even attach it as separate output table.) - Use a **Database** Update node to update this task's record: set status = "completed", set `updated_at`, store result fields, etc. - Optionally, you might remove or clear the `poll_url` if it's no longer needed, or set a `completed_at` timestamp. - If the status indicates **Failed** (the external service reports a failure), or the HTTP request itself returned an error status code, then mark the task as **failed**: - Update the task record's status to "failed". You can also record an `error_message` if the API provided a reason, or use the HTTP error message/response. - Note: If an HTTP node fails (e.g. a network error or non-2xx status and you have "Continue on fail" disabled), n8n might stop the workflow unless handled. To handle this gracefully, consider enabling *Continue On Fail* for the HTTP

node, or use a **Try/Catch** node around it. In a Try/Catch, the error path can catch exceptions and then you can update the task as failed due to an error. This way one task's API failure doesn't halt the entire monitor run. - If no clear status is given (maybe the API returns some progress percentage?), then the logic might consider certain fields or continue polling until some condition. But typically, we look for a "done" flag or a known end state. 4. **Retry Logic:** If the HTTP call fails due to a transient issue (e.g. network timeout or rate limit), you might not want to immediately mark as failed. Implement a simple retry mechanism: - If using Try/Catch, in the catch path, check the `attempt_count` for the task (which you would have fetched). If attempts so far < max_retries, you can decide *not* to mark failed yet. Instead, increment the attempt_count and update that in the DB (or you can simply leave it for now and let the next scheduled run try again, essentially achieving retry by doing nothing now). - If attempts >= max, then update status to failed. - You could also implement an exponential backoff by not checking a task every cycle after failures. For example, store next_attempt_time in the DB and have the query skip tasks until that time. - For many use cases, a simpler approach is: try on every schedule tick, and if a certain number of consecutive failures occur (or after X minutes/hours of no success), mark it failed. This can be done by storing a fail count or by comparing current time with created_at. 5. **Timeout Handling:** Similar to retries, decide on a global or per-task timeout. For example, if a task has been in progress for more than 2 hours (based on created_at or a stored `deadline`), you may assume it hung or lost and mark it failed. In the monitor loop, for each task you can check `if (now - created_at) > maxDuration` and if so, update status to failed ("timeout") and perhaps skip the API call (no need to poll if we consider it timed out). - You might send a different alert for timeouts vs explicit failures. - The timeout duration could be constant or set per task type. (If tasks from ServiceA usually finish in 5 min but ServiceB can take 2 days, you'd have different expectations. Such info could be encoded in the task metadata or type.)

After processing each task item in this manner, the Task Monitor workflow ends and will run again on the next interval. Over successive runs, it will keep picking up unfinished tasks until they reach a terminal state (completed/failed).

Updating Task States: All updates happen via DB update queries. For consistency, ensure that only the monitor workflow updates the statuses (the creating workflows only insert new tasks). This avoids multiple sources of truth. If manual intervention is needed (say an admin wants to cancel a task), they could manually update the DB or you could build a small "cancel task" workflow that sets status to failed/cancelled – but that's an extension beyond current scope.

Example: Suppose a task was created and the external ID is `abc123`. The monitor runs, fetches it, calls `GET https://api.service.com/job/abc123/status`. The response JSON might be `{"id":"abc123", "status":"done", "result":"https://files.service.com/output.csv"}`. The monitor sees status "done", so it updates the task row: status = "completed", result_url = (that CSV link). On next run, that task will either be filtered out (since status is no longer pending) or even if fetched, the logic would skip since already completed.

Parallelization Consideration: By default, n8n will handle the tasks one by one in a single workflow execution. If you expect a huge number of tasks and slow responses, this could become a long loop. If needed, you could run multiple monitors in parallel or use a queue system (discussed later) – but for most cases, a single monitor with sequential polling is sufficient (especially if each API call is quick). You can also adjust the batch size: for instance, use Split In Batches node to take 5 tasks at a time, call their APIs (n8n will still do them sequentially in one workflow execution, but it prevents one run from trying to do 1000 tasks and timing out; you could then loop the Split node to continue in the same execution). Alternatively,

schedule the monitor more frequently but limit each run to processing X tasks, and the rest in next run (this is an advanced optimization).

Resource Cleanup: Once tasks are completed or failed, they remain in the database. You may want a retention policy. For example, you might later archive or delete tasks older than N days to keep the table from growing indefinitely. This could be done with another maintenance workflow or a periodic query.

Task Status Query (Lookup by ID)

One of the requirements is that other workflows (and presumably the Phase 2 UI or even external clients) can check the status of a task by providing its ID. There are multiple ways to implement this in n8n:

- **Sub-workflow approach:** Create a workflow called "Get Task Status" that accepts an input (Task ID) and simply queries the Task Data Store for that ID, then returns the data. For instance, it could be triggered by an **Webhook** (with the Task ID as a query parameter or part of URL) or by an **Execute Workflow** node call from within n8n. Within this workflow, use a DB node to select the task where id = input, then perhaps format the output (e.g. return only the status and maybe result or a message). This is straightforward. Other workflows can call this using an **Execute Workflow** node (if internal usage) or via HTTP if you set it up as a webhook. The benefit is encapsulation: if the storage or schema changes, you update this one workflow. The downside is a slight overhead of an extra workflow call, but that's usually fine for quick queries.

- **Direct DB query:** In any workflow, one can directly use a DB node to fetch the task by ID (just as the sub-workflow would). This avoids an extra call. However, you'd duplicate the query logic in multiple workflows. Still, for something as simple as "SELECT status FROM tasks WHERE id = ?" that's not a big deal. If you have only a couple of places that need it, direct query is fine.

- **HTTP API endpoint:** If you want external systems (like a frontend web app) to fetch status directly from n8n, you can create a workflow with a **Webhook** trigger (method GET) at an endpoint like `/check-task`. It expects a query param or route param for the Task ID. The workflow then does the DB lookup and responds with a JSON containing the status (and perhaps other info). For example, response: `{ "taskId": "...", "status": "in_progress", "result": null }`. This essentially turns n8n into a simple REST API for task status. This is useful for building a UI or for clients that cannot directly access the database.

- Make sure to secure this if needed (n8n can require API key or you implement auth in the workflow).

- The response can include as much info as appropriate – e.g. you might include a percentage if the external API gave progress info, etc.

- **n8n UI usage:** Note that n8n's own UI (the editor) is not intended to query user-defined data at runtime. So you wouldn't use n8n's interface to check tasks except by looking at the database or building the aforementioned endpoint.

Given these options, a recommended implementation is the Webhook workflow for external and perhaps the sub-workflow for internal. For example, other workflows could simply call the Webhook via HTTP node

as well if that's easier than using Execute Workflow (especially if the checker is designed as a Webhook anyway). This way everything uses one method.

Usage Example: Suppose you have a user-facing workflow where the user can request the result of their task. That workflow might accept a Task ID (maybe the user enters it or it was stored in your app). That workflow could call the "Get Task Status" sub-workflow and get the status. If the status is completed, it could even fetch the result (maybe the result data or file link) and return it to the user. If still pending, it might return a message "still in progress" or perhaps trigger another wait. (However, usually it's the client's responsibility to poll the status periodically, not n8n pushing it – unless you implement Server-Sent Events or such).

Thus, the Task Manager supports checking status on-demand via the ID, decoupling it from the monitor's schedule. This is important: even if the monitor runs every 5 minutes, a client could query in between – they'd just get the last known status from the DB (which might still be "pending" if not yet updated). This is fine and expected in a polling system.

Error Handling, Retries, and Alerts

Robustness in the Task Manager comes from handling failures gracefully and notifying when something goes wrong beyond recoverable.

Retries in Detail: As mentioned, when a poll fails (due to network or a 500 error, etc.), the monitor workflow should ideally try again a few times before giving up: - The simplest approach is inherently built-in: since the monitor workflow triggers periodically, a transient failure now might succeed on the next run. You might not even need to code explicit "retry immediately" logic; just avoid marking the task failed on the first error. You can log the error and leave the status as in_progress, so that on the next scheduled run, it tries again. This is a form of retry with a delay equal to the monitor interval. - If the API is particularly flaky and you want to retry sooner than the next cron, you could implement a loop or a short Wait in the monitor workflow. For example, if a poll returns a network error, use a Wait node to pause 1 minute and then try the same task again within the same execution (with a max of N attempts). However, this can complicate the flow, and keeping the workflow execution open is usually not necessary since we have the periodic trigger. Leveraging the periodic nature is simpler. - Keep track of how many times we've tried or how much time has passed. A field like `attempt_count` can be incremented each time a task is polled. If a task is polled and still not finished, that could increment a counter or just be ignored; if an error occurs, increment a separate error counter perhaps. When either counter exceeds a threshold, mark fail. - Alternatively, use time: if `now - created_at > X` or `now - last_successful_check > Y`, then decide it's abnormal. For example, if no update for 24 hours, likely stuck.

Timeouts: Decide on a timeout threshold appropriate for your use case: - Could be a fixed value, e.g. 1 hour or 1 day. If tasks commonly finish well before that, then any exceeding it are considered failed. - Could be dynamic: maybe when creating a task, if you know from context how long it should take (some APIs might give an ETA or you know small tasks vs big tasks). - The monitor can compare current timestamp to the task's `created_at`. If beyond threshold and status still not completed, it updates status to "failed" (or maybe a specific "timeout" status) and possibly adds a message "Timed out after X hours". - This ensures tasks don't languish forever in "in progress".

Alerting: When a task fails (either due to explicit failure from API or due to timeout/retries exceeded), it's often important to alert someone: - The monitor workflow, right after marking a task as failed, can branch to an **Email node** or **Slack node** (or whatever notification) to send an alert. For example: "Task {{task_id}} has failed. Reason: ...". This could go to an admin or to the user who initiated it if you have their contact (that would require storing e.g. a user ID or email with the task). - If there are many tasks, you might want to throttle alerts or group them (perhaps an hourly summary of failures). But for critical tasks, immediate alerts are preferable. - Another approach: have a separate workflow that triggers on a "failed task" event. Since n8n doesn't have an internal event bus, you can simulate this by using the database as trigger (some DB nodes can trigger on new rows, but that's not quite applicable here). Instead, maybe the monitor, after updating to failed, could make a simple HTTP call to a **webhook trigger** of an "Alert workflow" sending the task details. This decouples the alert logic (and doesn't slow down the monitor if sending email takes time). - Or simpler, the monitor does the email itself (within a batch or separate branch).

Logging and Visibility: It could be useful to log every status change. With a DB, you can always see the history via timestamps. If needed, a separate "task_history" table could record each change event. This is advanced and may not be needed initially.

Edge Cases: Consider what happens if n8n or the monitor workflow goes down for a while. Tasks might not be polled during that downtime. When it comes back up, the tasks are still in the DB as pending; the monitor will resume and pick them up. They might be overdue (some could have completed meanwhile externally). On the next poll, you may find some tasks complete (that's fine, you mark them accordingly albeit later than ideal). If a task had a timeout that passed during downtime, you might mark it as failed only when the monitor resumes. That's acceptable. For critical scenarios, ensure n8n (or at least the monitor workflow) is running reliably; you might even run multiple monitors or use n8n's own queue mode for high reliability. But these are more about deployment than design.

Ensuring a Generic & Extensible System

The above design avoids hardcoding for any specific external API. It uses data-driven polling where possible. To solidify the generic nature:

- The **Task Data model** should accommodate different task types. This usually means storing a reference to *how* to poll the task. We chose to store `poll_url` /method, etc. If some tasks don't require polling (maybe they trigger a callback instead), we could store that info and the monitor can see that and skip polling (or a Wait mechanism might handle those – see below for callback alternative). In short, design the schema to handle multiple kinds of tasks.
- The **Monitor logic** can be written to handle different cases:
- If using a single HTTP node with dynamic fields, it already can call any endpoint given by the item. Just ensure the JSON from each service can be interpreted. You might have to do some parsing in a Code node if responses vary widely. For example, ServiceA might return `{status: "done"}` vs ServiceB returns `{state: "finished"}`. You could standardize by mapping these in the task record (like store an expected success status value or a JMESPath query to evaluate). For initial simplicity, you might implement support for a handful of known services with if/switch, and later generalize.
- Another approach: store a "service type" and have a mapping in the monitor (like a Switch node) that directs each task to a different sub-stream where you use the appropriate node for that service. For instance, if one task is a long Google Cloud operation, you might use the Google API node; if another

is an internal microservice with a REST API, use HTTP node. This is extensible but does require editing the monitor workflow to add new services.

- **Any asynchronous process** can plug into this: whether it's an external HTTP API, an internal script, or even a human approval step. As long as you can represent its status in the table and query it (either by polling or receiving an update), you can use the task manager to track it.
- For tasks that have **callback/webhook support**: Some external systems allow registering a callback URL so they will hit your webhook when the task is done. If you have such tasks, you can actually handle them differently:
- When creating the task, instead of expecting to poll, you could register a unique webhook URL (perhaps using n8n's **Wait** node feature or a dedicated Webhook workflow) with the external service. The external service will call that URL upon completion. The n8n workflow listening at that endpoint can then directly update the task status to completed.
- n8n's **Wait node** is helpful here: it can generate a one-time webhook URL to resume the workflow when called [7]. For example, your main workflow could start a task, then use a Wait node configured to "Resume on Webhook" (the node provides `$execution.resumeUrl`). You could pass that URL to the external API as the callback. Then the workflow would pause. When the external service calls that URL, n8n will wake up and resume the workflow at that point, where you can then mark the task completed. This avoids polling entirely and is efficient. n8n supports very long waits (it persists the state to the database and can resume even days or months later without holding memory) [8] [9].
- However, using Wait nodes ties the task's lifecycle back to the original workflow execution. In our design, we separated them. It may be simpler to handle callbacks with a separate Webhook workflow (that receives a job ID from the external call, then updates the Task DB accordingly, and maybe triggers any follow-up logic needed). The Wait node approach is more if you want to keep the original workflow open. For a generic task manager, it's often better to close the initial workflow and have the callback be handled by a dedicated listener that knows how to update the task status.
- Regardless, our system can accommodate both polling and callbacks: tasks that expect a callback might have no poll_url and instead have a field like `callback_expected=true`. The monitor can ignore those (or mark them "waiting for callback"). If a certain time passes without callback, that could also timeout.
- **Scaling Out:** If at some point there are too many tasks for a single monitor workflow (say hundreds of tasks per minute), n8n might struggle if each poll is slow. In such cases, one could run multiple instances of the monitor (sharded by task type or ID range) to distribute load. Or incorporate the BullMQ queue system (discussed below) to manage polling in parallel jobs. But until that complexity is needed, the single monitor design is easier.

By fulfilling these design considerations, the Task Manager remains largely configuration-based for different tasks. To add a new async service integration, you would: have the workflow that calls it insert the appropriate poll info, and update the monitor workflow if needed to parse its responses. No major refactoring is required to support new cases.

Phase 2: Dashboard and UI Integration

In Phase 2, we plan to introduce a user interface for visualizing and potentially managing the tasks. Because we have structured the data in a central store and provided access mechanisms, integrating a UI is relatively straightforward. Here are some approaches and considerations for the dashboard:

- **Using the Task Database Directly:** If the Task Data Store is something like a database or Airtable, we could leverage its UI or API. For example, **Baserow/Airtable** have their own grid UI where you can see rows; one could create a shareable view for tasks. However, that might be too low-level for end users (and you might not want them to see all raw data).
- **n8n as an API for the UI:** As mentioned, setting up a Webhook endpoint to fetch status allows external clients to query. For a full dashboard, you might create additional endpoints:
- One endpoint to list all tasks (optionally with filtering/pagination). This would do a DB query for tasks and return JSON.
- One endpoint to get details of a single task (similar to the above "status by ID" endpoint).

- If needed, endpoints to trigger certain actions (like retry a task or cancel a task, if you implement those). These endpoints essentially turn n8n into a backend for your dashboard application. The dashboard UI (could be a simple web app) would make AJAX calls to these to display data.

- **Dashboard Implementation Options:**

- You could build a custom web front-end (using React, Vue, etc.) that hits the n8n endpoints. This gives full flexibility in design (tables, charts showing number of tasks in each status, etc.).
- Alternatively, use a tool like **Retool**, **Metabase**, or **Grafana** to create a quick dashboard connecting to the database. For instance, Grafana could connect to a Postgres DB and you can write a query to show number of pending vs completed tasks over time. This is more for internal monitoring than user-facing.
- If the tasks correspond to user actions, you might integrate the status into your existing app UI (just calling the API).

- Ensure that if multiple users are creating tasks, you have some way to filter tasks by owner in the UI (which means you'd store a user identifier in the task data).

- **Visualization:** At minimum, the UI can show a table of tasks with columns like Task ID, Status, Created Time, Last Updated, perhaps a short description of what the task is (you might include a "task_name" or "type" field for display). You can highlight failed tasks in red, completed in green, etc. For long-running tasks, a progress bar could be shown if the external API provides progress percentage (if so, store that in the task record and update it in monitor).

- **Interactive Control:** Phase 2 might also consider allowing some control via UI:

- **Retry/Restart:** If a task failed, an option to retry it. This would require triggering a workflow to perhaps create a new task (if the external system supports restarting) or resetting its status and re-polling. This gets complex and might not be implemented unless needed.

- **Cancel:** If the external API supports cancellation of a running task, you could add a feature where the UI (or an n8n endpoint) triggers a cancel API call and updates status to "cancelled". That would require storing something about cancel endpoint or similar in the task data.

- These are additional features beyond the core requirements, but mentioning them for completeness in design planning.

- **Real-Time Updates:** A sophisticated dashboard could use WebSocket or Server-Sent Events to get live updates when a task changes state. Since n8n primarily operates in batch mode, achieving real-time push would involve some effort:

- One could integrate something like **Supabase Realtime** if using a Supabase DB (it can stream table changes to clients).
- Or the UI can poll the "list tasks" API every few seconds – which is simplest though not the most efficient. Given moderate scale, polling the UI every 5-10 seconds might be acceptable to update the view.
- Or when the monitor updates a task, it could also send a Webhook to a service like Pusher or Firebase to notify the UI. This might be overkill unless real-time is critical.
- Since the question specifically says "provisions for integrating a UI", we just ensure our system exposes data in a convenient way (which it does via the DB and n8n endpoints).

Integration Example: Let's say we use a simple approach – the UI will periodically call `GET /tasks` (an n8n webhook workflow) to get all tasks. That workflow does `SELECT * FROM tasks` and returns JSON. The UI displays it. For status filtering, we could support a query param like `?status=pending` to only get those. Similarly, `GET /tasks/{id}` for a single task detail. This essentially is creating a mini REST API using n8n, which is feasible. We'd just need to secure it (maybe require an API key or basic auth on the webhook, or run it internally).

One could also skip n8n for the read side and query the database directly from the UI backend (if you have another backend). But using n8n as the backend is convenient if you don't want to set up another service.

UI Mockup: The UI could show something like:

Task ID (clickable)	Created At	Status	Last Update	Details/Result
20231101-abc123	2023-11-01 10:00	in_progress	2023-11-01 10:05	(external job id, etc.)
20231101-xyz789	2023-11-01 09:30	completed	2023-11-01 09:45	result: file.csv (link)
20231101-lmn456	2023-11-01 08:20	failed	2023-11-01 08:45	error: API error 500

The dashboard can have a filter by status or a search by Task ID. This is all enabled by the data we store.

In summary, **Phase 2** leverages the existing data and workflows: - We ensure task records include any info needed for display (e.g. user-friendly name or type). - We expose the data via n8n (or allow direct DB reads). - We handle security if exposing externally. - The UI itself can be built with any tool since it will communicate via standard protocols (HTTP/JSON).

By planning these provisions early, we made sure the system isn't a black box: it's observable and queryable, which is crucial for a reliable long-running task system.

Comparison to Existing Solutions

There are both community-driven solutions within the n8n ecosystem and external platforms that address asynchronous task orchestration and monitoring. Below is a comparison of our proposed **n8n Task Manager** with similar solutions:

Solution	Type & Deployment	Key Features	Notes
Custom n8n Task Manager (our design)	Implemented as workflows within n8n, using an external DB for state.	Tight integration with n8n workflows; flexible task tracking tailored to specific needs; no additional infrastructure (besides a DB) for basic operation. Leverages n8n nodes (HTTP, Function, DB) for polling and updates. Supports custom logic, and immediate response with task IDs for long jobs [2].	Requires manual setup and maintenance of workflows and database. Not a plug-and-play feature – essentially building a mini orchestrator. However, it aligns with n8n's low-code philosophy and uses external storage since n8n lacks global state storage [1]. No out-of-the-box UI, but can be extended with endpoints/UI as described.
n8n + BullMQ (Queue Mode)	Community-contributed nodes + external Redis (queue); n8n Queue Mode.	BullMQ is a Node.js job queue library for background jobs. There's a community node package to integrate BullMQ into n8n [10] [11]. Key features include job queuing, concurrency control, retries, scheduling delayed jobs, and progress events. Comes with **Bull Board** UI for monitoring queued jobs [12]. Essentially, tasks can be pushed to a queue and processed by separate worker processes (could even be n8n itself or custom processors).	Using BullMQ introduces an external Redis service and a different paradigm (queue workers) which might be more complex if you're staying within n8n. However, it's very powerful for rate-limited or CPU-heavy tasks. The integration nodes allow n8n to add jobs and react to their completion. This approach might be overkill for simple external API polling, but it's great for distributing work and ensuring reliability (jobs persisted in Redis) [13]. Consider this if you need fine-grained control of retries, priorities, and a robust queue system built in.

Solution	Type & Deployment	Key Features	Notes
Netflix Conductor	Open-source **workflow orchestration engine**; runs as a server (Java-based) with persistence (MySQL/Redis) and a UI.	Designed for orchestrating complex microservice workflows, especially long-running processes. Defines workflows and tasks (which can be synchronous or async) via JSON or using a UI. Handles task scheduling, retries, timeouts, and has a rich UI to monitor workflow state. Particularly *"perfect for managing asynchronous tasks in complex microservices architectures"* [14] .	Conductor is a separate platform – using it means running the Conductor server and possibly writing workers or using HTTP tasks to interface with your services. It's heavy-weight compared to n8n, and not low-code. It's suited for large-scale enterprise scenarios (originally by Netflix). Integration with n8n would be indirect (perhaps n8n triggers a Conductor workflow or vice versa). For purely n8n users, Conductor likely isn't necessary unless you outgrow n8n's capabilities.
Temporal (by Temporal.io)	Open-source **distributed workflow engine** (programmatic, code-driven). Deployable service with client SDKs (Java, Go, Python, etc.).	Temporal provides *"a robust workflow management system that excels in reliability and scalability, ideal for long-running processes"* [15] . Workflows are written in code (using Temporal's libraries) and the engine ensures state persistence, timeouts, and retries. It has the concept of long-running workflows that survive process restarts, and offers built-in guarantees (e.g. "exactly once" task execution). Great for handling complex retry logic and audit trails automatically.	Temporal is very powerful, but requires software development – it's not a no-code tool. It's more comparable to AWS Step Functions or Durable Functions but self-hosted. One would typically use Temporal when building a large system in code where workflow reliability is paramount. It's probably beyond the needs of an n8n user unless you have in-house developers to integrate. If we compare, our n8n solution is simpler to set up via UI, whereas Temporal requires writing workflows in code and operating a cluster.

Solution	Type & Deployment	Key Features	Notes
AWS Step Functions	**Cloud service (AWS)** – managed orchestration with a visual state machine designer.	**Serverless orchestration**: You define workflows using states (tasks, wait, choice, parallel, etc.) in a JSON (Amazon States Language) or via a visual workflow studio. It can coordinate AWS services and custom tasks. It natively supports long waits and even human approval steps. Step Functions can poll AWS services or integrate callbacks (it has an async callback pattern). It's described as *"a serverless orchestration service that makes it easy to coordinate components of distributed applications and microservices using visual workflows"* [16] . It has built-in error handling and retry capabilities for each step, and you pay per state transition.	As an external service, using Step Functions would mean possibly migrating your automation logic to AWS rather than n8n. However, one could use n8n to trigger Step Function executions and wait for results. Step Functions are robust and managed (no server to maintain), but come with AWS lock-in and costs. They shine if you're already in AWS ecosystem. Compared to n8n's approach, Step Functions provides a polished visual flow specifically for orchestration, including a console to track execution progress step by step. It could be overkill if you just need to poll a couple of APIs, but for complex multi-step async flows it's a strong option.

Solution	Type & Deployment	Key Features	Notes
Camunda Platform 8 (Zeebe)	**Open-source BPMN workflow engine** (with enterprise and SaaS options). Deploys as a cluster of brokers and workers.	Camunda 8 (Zeebe engine) is a cloud-native workflow engine for orchestrating long-running business processes using BPMN 2.0 diagrams. It supports asynchronous tasks, human tasks, and has horizontal scalability for high throughput. Notably, *"Zeebe supports asynchronous process execution and horizontal scaling, enabling the simultaneous handling of millions of processes."* [17] . It provides an advanced web Modeler for designing processes and Operate UI for monitoring instances, with features like retry policies and SLA timers built-in.	Camunda is a full-fledged process orchestration platform often used in enterprise scenarios (financial transactions, order processing, etc.). It requires understanding BPMN notation and deploying the engine (or using Camunda's cloud service). Compared to our n8n solution, Camunda offers a more standardized modeling approach and can handle very complex workflows with sub-processes, etc. It might be considered if your workflows involve many steps and integrations and you need robust transaction handling. However, it is significantly more complex to set up than n8n, and might be "too heavy" if you just need async API polling. Camunda would usually replace a tool like n8n rather than complement it, except you could have n8n trigger Camunda workflows.

Table: Comparison of our n8n-based Task Manager with other orchestration/async task solutions. Each of these solutions provides a way to manage long-running tasks, but they target different user bases and complexity levels.

As seen, our custom n8n solution is characterized by flexibility and easy integration into existing n8n workflows, at the cost of some manual configuration. In contrast, dedicated orchestration systems (Temporal, Conductor, Camunda, Step Functions) provide out-of-the-box frameworks for async processes (with features like visual tracking, inbuilt retries, etc.), but they require adopting a new platform or writing code, which may be overkill for many automation needs.

Within n8n's ecosystem, because there isn't a native module for long-running task management, users have improvised with databases and clever use of webhooks [2] or community packages (like the BullMQ nodes) to achieve similar goals. The solution we propose essentially formalizes those best practices (database for state + scheduled checks) into a reusable pattern.

Conclusion

By implementing this Task Manager in n8n, we achieve a robust way to orchestrate asynchronous and long-running processes without blocking workflow execution. The design meets all specified requirements: workflows can create tasks with unique IDs and immediate acknowledgment, a centralized monitor workflow reliably updates task statuses via API polling (with retries and timeouts), and any workflow or client can query the status by ID at any time. The system is generic – it can handle any task type as long as we provide the polling instructions – and can be extended to new integrations easily.

Phase 2 planning ensures that the task data is readily accessible for visualization, paving the way for a dashboard that could be used by end-users or administrators to track progress in real time. With the data and mechanisms in place, adding a UI is straightforward, whether via n8n's own endpoints or external tooling.

While n8n doesn't natively include a full-featured async job manager, this specification leverages n8n's flexibility to create one. We also reviewed existing tools and platforms: for many n8n-centric use cases, our solution is the most direct and low-code. For extremely large-scale or complex orchestrations, one might evaluate external workflow engines or queue systems, but those come with additional complexity and cost.

In summary, the proposed Task Manager offers a comprehensive solution for long-running task orchestration in n8n, integrating seamlessly with workflows and providing a clear path to monitoring and user visibility – a solution built with current best practices from the community and analogous systems [2] [18]. By implementing this, teams can trust that even their asynchronous processes are being tracked and managed reliably, all within the familiar n8n environment.

---

[1] [3] How to store data as global variable? - Questions - n8n Community
https://community.n8n.io/t/how-to-store-data-as-global-variable/27125

[2] how do you handle long-running processes in n8n with status updates? : r/n8n
https://www.reddit.com/r/n8n/comments/1kq5l00/how_do_you_handle_longrunning_processes_in_n8n/

[4] GitHub - interactafraz/anystate: AnyState is a lightweight tool to easily store and retrieve values (like texts, numbers or arrays) in a JSON file using your own web server.
https://github.com/interactafraz/anystate

[5] [6] Async Workflow Nodes - Questions - n8n Community
https://community.n8n.io/t/async-workflow-nodes/12809

[7] Wait | n8n Docs
https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.wait/

[8] [9] [18] Evaluating n8n for very long workflows - Questions - n8n Community
https://community.n8n.io/t/evaluating-n8n-for-very-long-workflows/30069

[10] [11] [12] [13] GitHub - minhlucvan/n8n-nodes-bullmq: N8n node enable queue operators using bullmq interfaces
https://github.com/minhlucvan/n8n-nodes-bullmq

14 15 ☆ Top 10 Open-source Workflows Projects with the Most GitHub Stars | by NocoBase | Medium
https://medium.com/@nocobase/%EF%B8%8F-top-10-open-source-workflows-projects-with-the-most-github-stars-nocobase-28e216e09cc8

16 Introduction to AWS Step Functions: Workflow Orchestration Made Easy | by Christopher Adamson | Medium
https://medium.com/@christopheradamson253/introduction-to-aws-step-functions-workflow-orchestration-made-easy-5af653fc9d29

17 Time to Migrate: Why Businesses Opt for Next-Level Process Orchestration with Camunda 8 — IBA Group
https://ibagroupit.com/insights/time-to-migrate-why-businesses-opt-for-next-level-process-orchestration-with-camunda-8/