**Addendum: Orchestrator Lifecycle & Resource Management**

This document provides a detailed specification for the Orchestrator's internal lifecycle, its communication patterns with the Resource Coordinator, and the data structures required for its operation. It synthesizes the discussions and decisions made regarding job execution, resource management, and system configuration.

**1. Core Principles**

* **Database as Source of Truth**: The central database is the authoritative source for all job and task states. The Orchestrator's in-memory structures are considered a non-authoritative cache to improve performance.
* **Resilience through Recovery**: The system is designed to recover from an Orchestrator crash. Upon restart, the Orchestrator will rebuild its in-memory state by reading the database, ensuring no work is lost.
* **Configurable Behavior**: The Orchestrator's operational logic, particularly resource management, is driven by system parameters, allowing it to adapt to different deployment models (EKS vs. SINGLE\_PROCESS).

**2. In-Memory Data Structures**

The Orchestrator holds a minimal set of data in memory for efficient operation.

**2.1 Orchestrator Job Cache**

The Orchestrator maintains a simple dictionary to cache high-level information about active jobs. This avoids repeated database queries for every decision.

* **Structure**: A dictionary where the key is the job.id (integer).
* # Example Orchestrator Job Cache
* job\_cache = {
* 12: { # job.id
* "execution\_id": "run-abc-123",
* "status": "DISCOVERY",
* "priority": "HIGH",
* "pending\_task\_cache": collections.deque(maxlen=100) # In-memory task queue
* },
* 15: {
* "execution\_id": "run-def-456",
* "status": "CLASSIFICATION",
* "priority": "NORMAL",
* "pending\_task\_cache": collections.deque(maxlen=100)
* }
* }
* **pending\_task\_cache**: This is a thread-safe queue (like collections.deque) that holds a batch of PENDING tasks fetched from the database. This is the cache we agreed upon to improve performance while maintaining resilience.

**2.2 Resource Coordinator State**

The Resource Coordinator maintains its own in-memory state to track resource allocation.

* **Structure**:
* # Example Resource Coordinator State
* resource\_state = {
* "deployment\_model": "EKS", # or "SINGLE\_PROCESS"
* "fair\_sharing\_tracker": {
* 12: 8, # job.id: number of assigned workers
* 15: 4
* },
* "datasource\_connection\_tracker": {
* "ds\_001": 5, # datasource\_id: active connections
* "ds\_002": 10
* },
* "process\_memory\_tracker": { # Only used in SINGLE\_PROCESS model
* "total\_allocated\_mb": 4096,
* "tasks\_in\_flight": {
* 101: 512, # task.id: estimated\_cost\_mb
* 102: 1024
* }
* }
* }

**3. System Parameters Influencing Behavior**

The following parameters, loaded from system.yaml or the SystemParameters table, directly control the Orchestrator's lifecycle.

* **orchestrator.deployment\_model**: Critical flag (EKS or SINGLE\_PROCESS) that dictates the resource management strategy.
* **orchestrator.task\_cache\_size**: The number of pending tasks to pre-fetch from the database for each active job (e.g., 100).
* **orchestrator.fair\_sharing\_policy**: The algorithm to use (e.g., ROUND\_ROBIN, PRIORITY\_BASED).
* **datasource.max\_connections**: A per-datasource limit on concurrent tasks.
* **system.total\_process\_memory\_limit\_mb**: (For SINGLE\_PROCESS model) The total memory the application can use before it stops assigning new tasks.
* **task\_cost\_estimation.\***: A set of parameters used to estimate the memory cost of different task types (e.g., classification\_mb\_per\_gb\_file).

**4. Detailed Flow of Actions: Task Assignment**

This is the step-by-step process for assigning a single task to a worker.

1. **Trigger**: An idle Worker calls the get\_task() API endpoint.
2. **Orchestrator Asks for Strategic Guidance**:
   * The Orchestrator's Task Assignment Engine calls its internal **Resource Coordinator**.
   * **Method Call**: resource\_coordinator.get\_next\_job\_for\_assignment()
   * **Input**: None.
   * Resource Coordinator Logic:

a. It inspects its fair\_sharing\_tracker.

b. It applies the fair\_sharing\_policy (e.g., round-robin) to select the job.id that is most deserving of the next resource.

* + **Output**: A job.id (e.g., 15).

1. **Orchestrator Prepares a Task**:
   * The Orchestrator checks its in-memory job\_cache for job 15.
   * It looks at the pending\_task\_cache for that job.
   * If the cache is empty, it runs a database query to fetch the next batch (orchestrator.task\_cache\_size) of PENDING tasks for job 15 and populates the cache.
   * It takes the next task from the front of the pending\_task\_cache.
2. **Orchestrator Performs Tactical Validation**:
   * The Orchestrator now has a specific task and its WorkPacket. It extracts the datasource\_id and estimates the task's resource cost.
   * It calls the **Resource Coordinator** again for final approval.
   * **Method Call**: resource\_coordinator.request\_resource\_approval(request)
   * **Input (ResourceRequest object)**:
   * {
   * "job\_id": 15,
   * "datasource\_id": "ds\_002",
   * "estimated\_cost\_mb": 512 # Will be 0 in EKS model
   * }
   * Resource Coordinator Logic:

a. It checks the datasource\_connection\_tracker. Is ds\_002's current count (10) less than its configured max\_connections?

b. If deployment\_model is SINGLE\_PROCESS, it checks the process\_memory\_tracker. Is (total\_allocated\_mb + estimated\_cost\_mb) within the system limit?

* + **Output (ResourceDecision object)**: {"is\_approved": True, "reason": "OK"}.

1. **Task Assignment and State Update**:
   * If the request was approved, the Orchestrator performs the critical database transaction to update the task's status from PENDING to ASSIGNED, setting the WorkerID and LeaseExpiry.
   * It updates its in-memory trackers (increments the counters in the Resource Coordinator).
   * It returns the WorkPacket to the waiting Worker.
   * If the request was denied, the task is put back at the front of the in-memory cache, and the Orchestrator can either try the next task or wait for the next cycle.

**5. Single-Process Deployment Model Management**

When orchestrator.deployment\_model is set to SINGLE\_PROCESS, the Orchestrator is responsible for managing the lifecycle and health of its internal worker threads.

**5.1 Worker Thread Pool Configuration**

The size of the worker pool is determined by a system parameter.

* **worker.in\_process\_thread\_count**: Defines the fixed number of worker threads the Orchestrator will create and maintain (e.g., 8). On startup, the Orchestrator reads this value and initializes its thread pool.

**5.2 Worker Thread Health Monitoring**

The Orchestrator directly monitors its threads using an in-memory tracker and a periodic health check, which is simpler and more direct than the network-based lease mechanism used for EKS.

* **In-Memory Health Tracker**: The Orchestrator maintains a thread-safe dictionary to track the status of each worker thread.
* # Example Worker Thread Health Tracker
* worker\_thread\_tracker = {
* "worker\_thread\_1": {
* "status": "PROCESSING\_TASK\_123",
* "last\_heartbeat": "2025-08-25T21:51:30Z"
* },
* "worker\_thread\_2": {
* "status": "IDLE",
* "last\_heartbeat": "2025-08-25T21:51:32Z"
* }
* }
* **Worker Responsibility**: Each worker thread is responsible for updating its own entry in this tracker. It updates its status when it starts/finishes a task and updates the last\_heartbeat timestamp every few seconds.
* **Orchestrator Health Check Loop**: A low-priority background thread within the Orchestrator runs periodically to scan the worker\_thread\_tracker.
  + It checks for any thread whose last\_heartbeat is older than the worker.in\_process\_health\_check\_timeout\_seconds parameter.
  + If a thread is deemed "stale," the Orchestrator will terminate and replace it to maintain the configured pool size. Any task the stale thread was processing will be marked as FAILED and re-queued in the database.