Scalable Job Scheduler (Airflow-Like) Design Documentation

1. Overview, Requirements, and Assumptions

A. Functional Requirements:

- Workflow Orchestration: Define and schedule workflows as Directed Acyclic Graphs (DAGs).
- Task Scheduling: Determine task execution order based on dependencies and schedules.
- Real-Time Execution: Dispatch tasks to executors and provide near real-time feedback.
- Retry and Failure Handling: Support task retries, error handling, and backfill operations.
- Extensibility: Allow custom operators, hooks, and plugins to integrate with external systems.
- User Monitoring: Provide a Web UI to visualize DAGs, monitor task status, and view logs.

B. Nonfunctional Requirements:

- Low Latency: Tasks are scheduled and dispatched within seconds.
- High Throughput: Support thousands of concurrent tasks and workflows per day.
- Scalability: Horizontally scale scheduler, executor, and worker nodes with cloud orchestration.
- Fault Tolerance: Use replication and failover strategies to prevent single points of failure.
- Global Distribution: Deploy services in multiple regions to reduce latency for distributed users.
- Security: Use TLS for communication; enforce strict authentication and role-based access control.

C. Assumptions:

- Millions of workflows and tasks are defined, but only a fraction run concurrently.
- The system is deployed as microservices on cloud infrastructure, using containers (Docker/Kubernetes).
- A persistent metadata database (e.g., PostgreSQL/MySQL) holds workflow definitions and state.
- The system supports multiple execution models (local, Celery, Kubernetes).

2. High-Level Architecture and Component Responsibilities

A. Client Tier:

- User Interfaces: Web UI and CLI enable users to create, edit, and monitor workflows.

B. API Gateway and Authentication:

- API Gateway: Authenticates requests, enforces rate limits, and routes API calls.
- Authentication Service: Issues tokens (e.g., JWT) and manages session integrity.

C. Scheduler Layer:

- Scheduler Service: Periodically polls the metadata database to identify due tasks.
- Task Prioritization: Applies policies to decide the order of task execution (considering dependencies, retries, and SLAs).

D. Executor and Worker Nodes:

- Executor: Dispatches scheduled tasks to Worker Nodes using a distributed message broker (e.g., Celery with RabbitMQ/Redis) or Kubernetes API.
- Worker Nodes: Execute tasks in isolated containers (or VMs) with strict resource limits and timeouts.

E. Metadata and Logging:

- Metadata Database: Stores DAGs, task instances, and workflow history with strong consistency and replication.
- Centralized Logging: Collects logs from all components for monitoring and debugging.

F. Monitoring and Analytics:

- Real-Time Dashboards: Tools like Grafana display system performance and operational metrics.
- Alerting: Automated alerts notify administrators of failures or performance issues.

3. Detailed Workflow

A. Workflow Deployment and DAG Definition:

1. Users define workflows as Python DAG files, specifying task dependencies and schedules.
2. DAG files are deployed to a central repository, and the Scheduler periodically scans for updates.

B. Task Scheduling and Dispatch:

1. The Scheduler queries the metadata database to determine which tasks are due to run based on schedules and dependencies.
2. Ready tasks are enqueued to a message broker.
3. The Executor pulls tasks from the broker and dispatches them to available Worker Nodes.

C. Task Execution and Feedback:

1. Worker Nodes execute tasks in sandboxed environments (containers) and capture logs and exit statuses.
2. Results are sent back to the Executor and recorded in the Metadata Database.
3. The Web UI updates in real time to show task progress, results, and logs.

D. Retry, Backfill, and Recovery:

1. Failed tasks trigger a retry mechanism as per defined policies.
2. Backfill operations re-run tasks for historical periods if necessary.
3. In case of node failure, tasks are re-assigned automatically to maintain continuity.

Scalable Job Scheduler (Airflow-Like) Design Documentation

4. Scalability, Fault Tolerance, and Global Distribution

A. Horizontal Scalability:

- The Scheduler, Executor, and Worker Nodes are stateless and can be scaled by adding more instances behind load balancers.
- The Metadata Database is sharded and replicated, handling high transaction rates efficiently.

B. Fault Tolerance:

- Critical services are deployed redundantly; if a Worker Node fails, tasks are re-queued and re-assigned.
- The Metadata Database is replicated (e.g., replication factor of 3) to ensure data durability.

C. Global Distribution:

- Regional deployments and global load balancing reduce latency and distribute user load.
- Multi-region clusters ensure that users receive responses from the nearest data center.

5. Protocols and Security

A. Communication Protocols:

- All client interactions occur over HTTPS; WebSocket (WSS) is used for real-time updates.
- Interservice communication uses gRPC or REST over secured TCP with mutual TLS.

B. Security Measures:

- TLS encrypts all communications; sensitive metadata is encrypted at rest.
- Authentication (JWT) and role-based access control (RBAC) ensure only authorized users can define or trigger workflows.
- Worker environments are isolated to prevent malicious code from affecting the entire system.
- Audit logs record all system activities for compliance and troubleshooting.

6. Final Thoughts

This design for an online job scheduler (similar to Apache Airflow) provides a robust, scalable, and fault-tolerant framework for orchestrating complex workflows. Key features include:

- DAG-based workflow definitions that clearly express task dependencies and schedules.
- A distributed Scheduler that continuously polls for due tasks and dispatches them efficiently.
- Scalable Executor and Worker Nodes that run tasks in isolated, secure environments with support for retries and backfills.
- Persistent metadata storage and detailed logging for tracking workflow history and debugging

failures.

- Global distribution, autoscaling, and high availability ensure that the system can operate at large scale with minimal latency.

This conceptual design lays the foundation for building an advanced job scheduler capable of handling millions of tasks per day, supporting a diverse array of workflows and integrations.