ChatGPT

Apache Airflow: Architecture and Optimization

Apache Airflow is a workflow orchestration platform where **DAGs** (Directed Acyclic Graphs) define tasks and dependencies. Its core architecture is modular, with distinct components coordinating via a metadata database. In the simplest “one-machine” deployment (LocalExecutor), a single process runs the scheduler (with the executor) and webserver, sharing a local DAGs folder and SQLite/ Postgres DB. In larger deployments these components are separated: for example, a **Scheduler** process parses DAGs and triggers tasks; **Worker** processes (e.g. Celery workers or Kubernetes pods) execute tasks; a **Web Server** provides the UI; and a **Metadata Database** (Postgres/MySQL) stores all state. The scheduler uses an **Executor** (configurable) to submit tasks—e.g. LocalExecutor runs tasks in the scheduler process; CeleryExecutor pushes tasks to a queue; KubernetesExecutor launches each task in a pod. Optional components include a **Triggerer** (async event loop for deferrable tasks) and plugin folders for custom code.

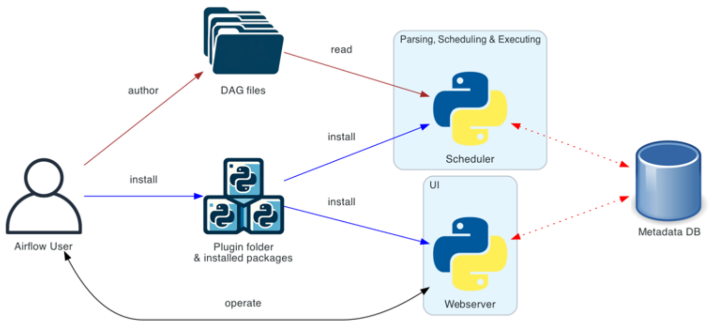

Figure: Basic Airflow architecture (single-machine LocalExecutor). The Scheduler reads DAG files, creates DAG runs and enqueues ready tasks via the Executor; Workers execute tasks and update the Metadata DB. The Webserver reads the DB (or serialized DAGs) to render the UI [1] [2] *.*

- **Scheduler:** Continuously monitors all DAGs and Tasks, creating new DAG runs and queuing ready tasks. It uses a configuration-defined executor to launch work [3] [4] . In each scheduling cycle it (1) reloads any changed DAG files, (2) checks if new DagRuns should be created (based on DAG schedule), and (3) finds *TaskInstances* whose dependencies are met and enqueues them to the executor [5] [4] . The scheduler’s main loop respects concurrency limits (`parallelism` , `max_active_tasks_per_dag` , pools, etc.) and leverages row-level DB locks to coordinate multiple schedulers if enabled [6] [4] .
- **Executor:** Logic built into the scheduler that actually runs tasks. In a LocalExecutor, tasks run in separate worker subprocesses on the same machine (fast, low latency) [7] . In a CeleryExecutor, tasks are serialized to a message queue (e.g. RabbitMQ/Redis) and pulled by external worker processes; this decouples workers from the scheduler and scales horizontally, but can introduce “noisy neighbor” contention on shared hosts [8] . In a KubernetesExecutor, each task runs in its

own Kubernetes Pod (full isolation and custom resources per task, but with pod startup latency) [9]. (Airflow 2.10+ even supports **multi-executor** mode, letting you combine executors per task type [10].)

- **Worker (optional):** When using a remote executor like Celery or Kubernetes, workers (or pods) pull and run tasks given by the scheduler. For example, Celery workers are long-running processes that can execute many tasks in parallel, while KubernetesExecutor spawns short-lived pods. In simple LocalExecutor setups, the worker code runs in the scheduler process. (If using CeleryExecutor, configure `worker_concurrency` or similar to tune how many parallel tasks each worker can handle [11].)
- **Metadata Database:** A relational database (Postgres, MySQL, etc.) stores the state of every DAG, DAG run, Task instance, sensor, etc. It is the single source of truth for Airflow's state. All components read/write this DB: the scheduler updates TaskInstance states (queued, running, success, etc.), the webserver reads it to show UI, and even DAG definitions can be serialized into it. Airflow *requires* a metadata database (SQLite is only for testing) [12].
- **Web Server/UI:** A Flask app that provides the user interface. It reads DAG and task status from the metadata DB. With **DAG Serialization** enabled (Airflow $\geq$1.10.7), the scheduler writes each parsed DAG's structure as JSON into the DB, and the webserver loads these serialized DAGs on demand (instead of parsing Python). This decouples the webserver from the DAG files and greatly improves UI performance at scale [13] [14].
- **Triggerer/Deferrable Tasks (2.x):** New in Airflow 2.2+, the triggerer is a special event loop process that handles *deferrable operators*. Standard Sensors or operators that wait (e.g. polling or long I/O) can be converted to deferrable form, letting them suspend without occupying a worker slot. The triggerer watches triggers and wakes operators when conditions are met, improving scalability of many long-running tasks.

Each component communicates via the database and queues. For example, the scheduler will write a TaskInstance as "queued" in the DB and push it to a Celery queue; a worker pulls it, executes the task code, and upon completion updates the DB to "success" (or another state). The UI simply displays what's in the DB.

Task Scheduling and Execution (System & Code)

On each scheduler cycle, Airflow executes the following high-level steps internally:

1. **Parse DAGs:** The scheduler (or a separate DagProcessor) refreshes the in-memory list of DAGs by scanning the DAG folder and parsing each file (see next section). This may spawn multiple subprocesses to parallelize parsing.
2. **Create DagRuns:** For each DAG whose schedule indicates a new run is due, the scheduler inserts a new DagRun entry (e.g. for each missed or periodic interval).
3. **Find Schedulable Tasks:** The scheduler loads active DagRuns and their TaskInstances from the DB. It then checks task dependencies (upstream tasks, `depends_on_past`, SLA misses, etc.). Any TaskInstance that is ready to run (all upstream dependencies are in a terminal state) is marked *schedulable*.
4. **Enqueue Tasks:** The scheduler respects concurrency limits (`max_active_tasks_per_dag`, pool slots, etc.) and then enqueues schedulable tasks. Depending on the executor: for CeleryExecutor it pushes them into a message queue; for KubernetesExecutor it creates a pod for each task; for LocalExecutor it forks a process to run the task's PythonOperator, BashOperator, etc. When enqueuing, it sets each TaskInstance state to *queued* in the metadata DB.
5. **Task Execution:** A Worker process picks up the queued task (or the LocalExecutor process begins it). The operator's code runs (e.g. a Python function or shell script). When finished, the

worker writes the result back to the DB (`success`, `failed`, etc.), and pushes any XCom values.

These steps repeat continuously. In code, the core scheduling loop is implemented in the `SchedulerJob` class. It uses a separate `DagFileProcessorManager`/`DagFileProcessorProcess` mechanism to handle parsing (next section). Then it follows roughly: *check for new DagRuns and create them; examine a batch of DagRuns for schedulable tasks; select tasks to schedule (honoring pools/concurrency); enqueue them* [5]. The scheduler always runs as a persistent service (`airflow scheduler`). Its heartbeat (by default once per minute) is how often it re-checks the DAG folder and enqueues tasks [4]. Internally, the scheduler uses the configured executor to actually launch tasks (no separate "Executor" daemon is needed – it's just logic inside the scheduler) [2].

DAG Parsing and Scheduler Performance

Parsing DAG files is one of the most CPU-intensive parts of scheduling. Airflow's scheduler employs a **DAG File Processor Manager** loop: it repeatedly looks for Python files in the DAGs folder that need refreshing, and for each it starts a `DagFileProcessorProcess` as a subprocess [15]. Each processor process then *executes the Python file*, instantiates DAG objects, and returns a *DagBag* (the collection of DAGs found in that file) back to the manager [16]. The manager may run several processes in parallel (controlled by `parsing_processes` config). A simplified view is shown below:

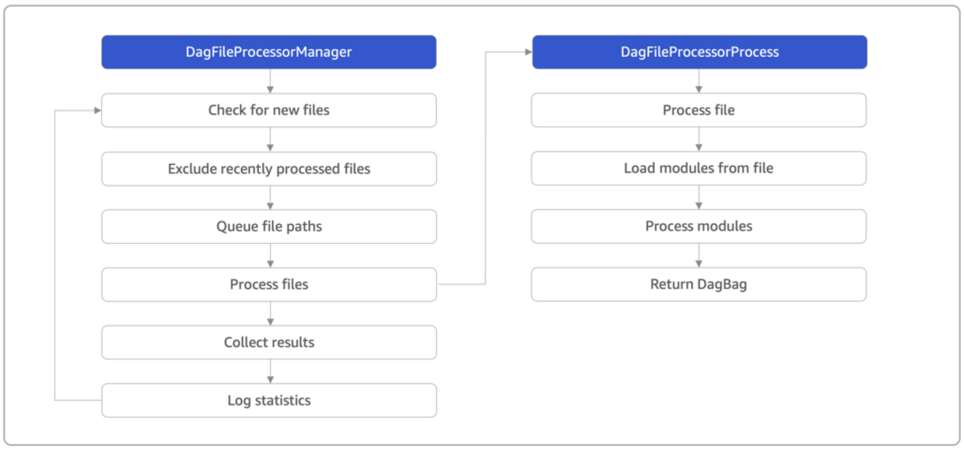

Figure: DAG file processing in Airflow. The `DagFileProcessorManager` *(left) loops over DAG Python files, queuing them for parsing. It spawns* `DagFileProcessorProcess` *workers (right) that load each file as a Python module, find all* `DAG` *objects, and return them. This allows parallel parsing of DAG files* [15] [16].

Because every DAG file is executed as Python, **top-level code in the file runs at parse time**. This means any global statements, API/database calls, or heavy imports in the DAG file will run every parse cycle (by default Airflow scans all files roughly every 30s, though this can overlap). This can severely slow down scheduling if the top-level logic is expensive [17]. To optimize: minimize work in global scope, avoid Airflow Variables or DB queries at parse time (use OS environment variables instead) [18], and cache any external results if needed. Tuning parameters is also key: for example, increasing `min_file_process_interval` (to e.g. 300+ seconds) reduces parsing frequency [19] [20] (at the cost of slower reaction to DAG changes). Similarly, increasing `parsing_processes` allows more files to be parsed in parallel, at the expense of CPU usage [20]. Monitoring DAG parsing time in the logs is a good practice.

In summary, the scheduler's parsing stage ensures all DAG definitions are loaded into the database (and, with DAG Serialization enabled, stored in JSON). After parsing, scheduling and enqueuing happen. When a task run completes, the worker updates the TaskInstance and XCom tables in the DB, which closes the loop.

Static vs Dynamic DAGs

In Airflow, **static DAGs** are those whose structure (tasks and dependencies) is fixed by the time the DAG file is parsed. A **dynamic DAG** is one whose Python file generates tasks (or even entire DAGs) programmatically, often in a loop or from external data. Internally, there is no special "dynamic DAG" object – Airflow simply executes the Python code and collects all `DAG(...)` instances it finds at top level of the module. For example:

```
from airflow import DAG
from airflow.operators.bash import BashOperator

with DAG('static_dag', ...) as dag:
    op1 = BashOperator(task_id='task1', ...)
    op2 = BashOperator(task_id='task2', ...)
    op1 >> op2
```

versus

```
from airflow import DAG
from airflow.operators.bash import BashOperator

# dynamic: create DAGs from a list
for cfg in [{"dag_id": "dag_A", "param": 1}, {"dag_id": "dag_B", "param": 2}]:
    dag = DAG(cfg["dag_id"], ...)
    BashOperator(task_id='task', bash_command=f"echo {cfg['param']}", dag=dag)
    # Each iteration defines a new DAG object in globals()
```

The **key rule** is that Airflow only **loads DAG objects found in the module's globals**. Objects created inside a function or not assigned at top level will *not* be seen by the scheduler [21]. In a static DAG file, typically a single `dag = DAG(...)` is in globals. In a dynamic file, the loop above creates two DAGs in globals, and Airflow will register both. The official docs note: "Airflow loads DAGs from Python files... execute it, and then load any DAG objects from that file" [22]. Note also that dynamic generation should produce a **consistent ordering** of tasks each parse; otherwise DAG views will shuffle on refresh. Use sorted lists or deterministic iteration when generating tasks or DAGs [23].

If your workflow truly needs a varying number of tasks per DAG run (decided at runtime), Airflow 2.x offers **Dynamic Task Mapping** (introduced in 2.3) as a better pattern than repeatedly re-parsing code. Dynamic Task Mapping creates tasks at *runtime* based on upstream data, rather than relying on the static DAG structure at parse time.

Best Practices for Scalable Dynamic DAGs

To write scalable, high-performance dynamic DAGs, follow these guidelines:

- **Minimize Top-Level Work:** Reduce CPU and I/O in the DAG file's top-level code. Avoid expensive imports, API calls, or database queries when the file is executed [17] [24]. For example, don't retrieve large lists of IDs from a database in global scope – instead, fetch what's needed inside an operator at task runtime. If you do need configuration data at parse time, cache it (in memory or on disk) or load from environment variables rather than calling Airflow Variables (which hit the DB) [18]. This greatly reduces DAG parsing time.
- **Raise `min_file_process_interval`:** If your DAG definitions seldom change, set a higher `scheduler.min_file_process_interval` (e.g. 300 seconds or more) to avoid constant re-parsing [25] [19]. This reduces scheduler load, though it delays recognition of updates.
- **Avoid Unnecessary DAG Instantiation:** Don't instantiate DAG objects you don't need. For example, wrapping DAG creation in a function or under `if __name__ == "__main__":` may prevent it from loading (Airflow won't see it), so instead use top-level code with caution. Only define as many DAGs or tasks as required, and consider combining very similar workflows into a loop (as in the example above) rather than dozens of almost-identical files [26]. Consolidating similar pipelines reduces the total number of files the scheduler must parse.
- **Efficient Jinja Templating:** Use Jinja templates in operator parameters for small substitutions (e.g. dates, paths, or simple XCom pulls). Remember that template rendering happens at task *render* time, not parse time, so it typically doesn't slow down the scheduler unless misused in loops. Access XCom values in templates via `{{ ti.xcom_pull(task_ids="...") }}` as needed [27]. However, do **not** generate tasks via Jinja (use Python logic instead), and avoid overly complex templated logic that the scheduler might evaluate.
- **TaskFlow API and XComs:** When possible, use the TaskFlow API (`@dag` and `@task` decorators) to simplify dependencies and data passing. Functions decorated with `@task` automatically push their return values as XComs. For example, using `@task(multiple_outputs=True)` lets a function return a dict, and Airflow will split it into individual XCom entries [28]. This makes it easier to build dynamic pipelines: downstream `@task` functions can accept the outputs of upstream tasks as arguments. In dynamic scenarios, you can loop over data and use TaskFlow to map variables to tasks, reducing manual XCom management.
- **Dynamic Task Mapping (Airflow 2.3+):** Instead of writing a Python loop in your DAG file to create many similar tasks, consider using dynamic task mapping with `expand()` on a TaskFlow task or operator. This defers creation of the actual task instances to runtime, which can improve scheduler responsiveness and is often more efficient.
- **Deferrable Operators for Sensors:** If your dynamic DAGs use many sensors or long-wait operators, use deferrable operators. For example, converting a traditional sensor into a deferrable one means it will release its worker slot while waiting. The task is suspended and a lightweight trigger handles the waiting. Halodoc observed that converting blocking sensors (e.g. EMR sensors) to deferrable freed up workers and prevented scheduler backlog [29]. This can dramatically improve scale when many dags contain long-running waits.

Following these patterns can greatly reduce parse times and resource usage. As one example, Halodoc reported that by applying best practices (raising `min_file_process_interval`, minimizing top-level code, combining similar DAGs, and using deferrable operators) they saw a complex DAG's parse time drop from ~5 seconds to ~1 second and overall scheduler CPU usage fall by ≈21% [25] [30].

Scaling Airflow: High-Scale Optimization

When running hundreds of DAGs and thousands of tasks, tuning both Airflow and infrastructure is essential. Key strategies include:

- **Parallelism and Concurrency Tuning:** At the *environment level*, set Airflow config such as `parallelism` (max concurrent tasks across the system) and `max_active_tasks_per_dag` to match your worker capacity [31]. For example, `core.parallelism=64` allows up to 64 concurrent tasks per scheduler; with multiple schedulers you multiply that. Also tune `max_active_runs_per_dag` to control how many DAG runs of a given DAG can run at once [32]. At the *DAG level*, parameters like `dag_concurrency` (or per-DAG `max_active_tasks`) can cap a busy DAG. For CeleryExecutor, configure `worker_concurrency` to match the CPU cores per worker node [11]. Monitor queued vs running tasks: if tasks stay queued, you may need to add workers or increase these limits.
- **Multiple Schedulers (HA mode):** For very high throughput, run *multiple schedulers* concurrently. Airflow supports HA schedulers out of the box by leveraging the metadata database rather than external locks [33]. All schedulers read serialized DAGs from the DB and compete to enqueue tasks. They coordinate via row-level locks on critical sections (e.g. locking the Pools table during scheduling) [6]. This avoids needing tools like Zookeeper. To use this safely, run on a supported DB (Postgres ≥10 or MySQL ≥8) so that `SKIP LOCKED`/`NOWAIT` queries work properly [6]. Multi-scheduler improves resilience and speed, since each can parse files and schedule in parallel.
- **DAG Serialization:** Enable DAG Serialization (config `store_serialized_dags=True`). This causes the scheduler to write each parsed DAG into the `serialized_dag` table as JSON [14]. The webserver then reads DAGs from the DB rather than importing code, which drastically cuts webserver memory and startup time with many DAGs [34]. The scheduler itself also uses the serialized form for scheduling decisions, meaning it doesn't need fresh Python parsing of DAG files each cycle [14]. Overall, serialization decouples components and reduces overhead in large environments.
- **Scheduler Tunables:** Beyond the basics above, tune scheduler performance settings. For example, `parsing_processes` (number of parallel parser subprocesses) and `num_runs` (how many scheduler loops to do) should be set based on available CPU [17] [20]. If the filesystem or network is slow, consider increasing `min_file_process_interval` and possibly using a **separate DAG processor** architecture (a dedicated service that parses DAGs, isolating the scheduler). Also monitor `dag_processing.total_parse_time` metrics to identify bottlenecks.
- **Infrastructure Scaling:** Ensure the metadata database and backing services can scale. Use a managed Postgres or a powerful RDS instance. Consider read-replicas for the webserver if query load is high. For CeleryExecutor, auto-scale worker nodes (or use KubernetesExecutor to leverage cluster autoscaling). For KubernetesExecutor, ensure the cluster has enough capacity and fast pod startup (e.g. with a local image cache).
- **Resource Pools and Priority:** Use Airflow Pools to limit how many concurrent tasks can hit a resource (like an external API or a cluster). Define pools for expensive downstream systems. Also use `priority_weight` or SLAs if some tasks should preempt others. These can prevent one DAG from starving others.

Optimizing Airflow at scale is an iterative process: measure parse times and task delays, then adjust configs or architecture. Astronomer's scaling guide notes that often increasing parallelism requires a matching increase in infrastructure (workers, CPU) [31] [11].

Executor Options: Trade-offs

Airflow offers several executors, each affecting architecture and performance:

- **LocalExecutor:** (Default for single-machine setups.) Tasks run as subprocesses on the scheduler host [7]. *Pros:* Easy to set up, very low task launch latency, no external queue needed. *Cons:* Shares CPU/memory with the scheduler; not suitable when tasks need isolation or when you outgrow one machine [7]. Use it only for small-to-medium loads.

- **CeleryExecutor:** (Distributed, queued.) Tasks are sent to a central broker; workers (on one or many nodes) pull tasks to execute. *Pros:* Decouples scheduler from workers, can leverage large, persistent worker fleet, horizontally scalable. *Cons:* Adds complexity (need a message broker and maybe a result backend), possible queue delays, and “noisy neighbor” issues on shared hosts [8]. It’s a battle-tested choice for many production deployments.

- **KubernetesExecutor:** (Containerized.) Each task is run in its own Kubernetes Pod, with whatever image and resources it needs [9]. *Pros:* Strong isolation (no resource contention between tasks), can customize environments per task, and autoscaling of pods. *Cons:* Pod startup adds latency (can be significant for short tasks), requires running a Kubernetes cluster and managing it, and may be cost-inefficient if tasks are very short-lived [9]. Use when isolation and dynamic scaling are priorities.

- **Other Executors:** Airflow also has a **DaskExecutor** (tasks as Dask futures in a Dask cluster), **CeleryKubernetesExecutor** (combining Celery with K8s), and **SequentialExecutor** (single-threaded, for testing only).

- **Multi-Executor (2.10+):** You can configure Airflow to use more than one executor type concurrently [10]. For instance, you could route certain DAGs to run on Kubernetes while others use Celery. The first executor listed behaves as before; subsequent executors can pick up tasks from different task queues. This allows leveraging multiple architectures in one Airflow instance.

When choosing, consider latency vs isolation vs cost. For example, CeleryExecutor gives relatively low latency and max throughput if you size your worker pool, whereas KubernetesExecutor is best when you need per-task customization or run in a Kubernetes-only environment. LocalExecutor is simplest but limited by a single host. In all cases, the Executor runs *within* the scheduler process (no separate “executor daemon”), even if tasks end up executed elsewhere [2].

Advanced Configuration and Trade-offs

- **DAG Serialization:** (Airflow ≥1.10.7) Storing parsed DAGs in the database as JSON (via `SerializedDagModel`) is strongly recommended in production. It makes the Webserver stateless and reduces load: the webserver loads only individual DAGs on demand from the DB [34]. It also enables multiple schedulers to share the same DAG definitions without needing filesystem access [35]. The trade-off is a bit more storage use in the metadata DB, and a slight delay (the time to serialize) after parsing. Overall it **improves UI and scheduling performance** in large environments.
- **Scheduler HA Mode:** As mentioned, you can run several schedulers for high-availability. This works only with serialized DAGs and a “modern” SQL database. Beware that if your DB does not support locking well (e.g. older MySQL or unsupported MariaDB), you might run into deadlocks. In HA mode, all schedulers share work via the DB: only one scheduler will hold the lock to

enqueue a given batch of tasks (so DAG limits are correctly enforced) [6]. The benefit is no single point of failure and higher throughput; the downside is more complexity (monitoring multiple schedulers) and the need to trust the DB as the coordinator.

- **Smart Sensors (Deprecated):** In Airflow 2.0/2.3, **Smart Sensors** were introduced as a way to consolidate many small sensor tasks. Instead of each sensor running in its own worker, the smart sensor service records "poke" requests in a `sensor_instance` table and uses a built-in DAG of `SmartSensorOperator` tasks to batch-process them [36]. This dramatically reduces resource use for fleets of sensors. However, Smart Sensors are an early-access feature and as of Airflow 2.4 have been superseded by Deferrable Operators and the Triggerer mechanism [37]. In modern Airflow, one would simply convert sensors (or other long-wait ops) to `DeferrableOperators`, which achieve a similar resource saving more flexibly.
- **Deferrable Operators & Triggers:** Introduced in Airflow 2.2+, a deferrable operator can suspend itself (freeing the worker) while waiting on an external event. When the event occurs, the Triggerer wakes the operator. This is superior to the old `poke`-loop style, and should be used for anything that waits (file arrival, EMR job, long polling). The trade-off is a bit more complexity (you must configure a trigger for each operator), but in return many more concurrent sensors or long waits can be handled with the same number of workers.
- **Smart Scheduling Patterns:** For example, **dynamic task mapping** (airflow 2.3+) avoids DAG-file looping altogether by expanding tasks at run time (unrelated to Smart Sensors). Another example is using **Task Groups** or sub-DAGs to break up huge DAGs for readability, but note that sub-DAGs have their own scheduling quirks and are generally superseded by TaskGroups in 2.x.

In all cases, the trade-offs boil down to **performance vs complexity**. Features like HA schedulers or serialization add overhead (DB writes, locking) but boost throughput. Deferrable operators and smart scheduling patterns may require re-writing tasks but save worker resources. DAG Serialization greatly improves UI and multi-scheduler behavior, so it is almost always worth enabling in production [14]. Smart Sensors are now obsolete, so prefer deferrable operators. Pools, priorities, and resource hints should be used to balance workloads when needed.

In practice, each environment requires tuning. For example, in one large deployment the team found that simply raising `scheduler.min_file_process_interval` and cleaning up top-level code in dozens of dynamic DAGs reduced the DAG processing CPU load by ~20%. They also switched their many EMR sensor tasks to deferrable EMR operators, freeing up workers for other tasks. In another case (on AWS MWAA), Airflow metrics showed that increasing `parsing_processes` from 2 to 8 and distributing DAG files over multiple processing nodes cut DAG parse queues significantly.

Ultimately, the best strategy is to measure (using Airflow's logs and monitoring metrics), identify bottlenecks (slow DAG parsing, queued tasks, high DB usage), and apply targeted optimizations from the above strategies.

Sources: Authoritative Airflow documentation and expert sources (Astronomer, community blogs, official best practices) were used to compile this overview [3] [4] [15] [17] [18] [22] [7] [13] [36] [28] [27] [31]. The architecture diagrams are from the official docs [1].

---

[1] [3] [12] **Architecture Overview — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html

[2] [7] [8] [9] [10] **Executor — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/executor/index.html

4 5 6 33 **Scheduler — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/2.5.1/administration-and-deployment/scheduler.html

11 31 32 **Scaling Airflow to optimize performance | Astronomer Documentation**
https://www.astronomer.io/docs/learn/airflow-scaling-workers/

13 14 34 35 **DAG Serialization — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/dag-serialization.html

15 16 17 20 **DAG File Processing — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/dagfile-processing.html

18 23 **Dynamic DAG Generation — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/howto/dynamic-dag-generation.html

19 **Dynamically generate DAGs in Airflow | Astronomer Documentation**
https://www.astronomer.io/docs/learn/dynamically-generating-dags/

21 22 **Dags — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

24 25 26 29 30 **Airflow Best Practices with Dynamic Dags**
https://blogs.halodoc.io/dynamic-dag-generation-in-airflow-best-practices-and-use-cases/

27 **Mastering Airflow XComs: Task Communication - A Comprehensive Guide**
https://www.sparkcodehub.com/airflow/variables-connections/xcoms

28 **Pythonic DAGs with the TaskFlow API — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/stable/tutorial/taskflow.html

36 37 **Smart Sensors — Airflow Documentation**
https://airflow.apache.org/docs/apache-airflow/2.3.0/concepts/smart-sensors.html