Airflow Cluster Sizing Guide — Load-Driven Reverse Engineering

Workload profile: dashboard-triggered, one-time, script-generated DAGs. Every task is a `BashOperator` shell-out. DAGs use native scheduling (`schedule=<fixed datetime>`, `is_paused_upon_creation=False`) — no external polling/trigger step. DAGs are archived after their single run.

How to use this guide: Section 1 defines the load model as a small set of named variables. Every config/resource decision downstream is a formula against those variables. To rescale for a different user count, change the variables in Section 1 and recompute Sections 2–7 using the formulas given — don't just copy the N=25 numbers.

1. Load Model — Define the Variables First

Variable	Meaning	Value for this guide	How to set it for your case
`N`	Peak concurrent users submitting jobs in a short burst window	**25**	Your actual expected peak, not average
`F`	Average concurrent *active* tasks per DAG at any instant (accounts for fan-out; 1 = strictly linear chain)	**2**	Look at your real DAG shapes — widest realistic parallel branch count, averaged
`T`	Peak concurrent active tasks across the cluster = `N × F`	**50**	Derived
`AVG_PARSE_S`	Approx. seconds to parse one generated DAG file	**~1–3s**	These are simple templated files — measure once with `time airflow dags reserialize` on a sample if you want precision
`CONN_MB`	Planning heuristic: memory per idle Postgres connection	**8MB**	Conservative; raise if `work_mem` is large or queries are heavy

Everything below is `T`, `N`, or a component's allocated cores/memory as the input — not a guess.

2. Celery Workers — the actual execution capacity

Formula:

```
worker_concurrency = cores_per_worker × 4     (I/O-bound: subprocess launches + waits, not CPU-bound)
worker_replicas    = ceil(T / worker_concurrency)
parallelism (core)  = worker_replicas × worker_concurrency   (with ~15–20% headroom)
```

Worked for N=25 (T=50), `AirflowWorker` = 2048MB / 2 cores per replica:

```
worker_concurrency = 2 × 4 = 8       ← matches your existing compose value, good sanity check
worker_replicas    = ceil(50 / 8) = 7
parallelism        = 7 × 8 = 56 → round to 64 for headroom
```

Config:

```
[celery]
worker_concurrency = 8

[core]
parallelism = 64
```

Resource change: keep `AirflowWorker` at 2048:2 per replica, but run **7 replicas**, not 4. Replica count is the scaling lever here, not per-container size — 2048MB/2cores comfortably handles 8 concurrent bash-wrapper subprocesses as long as the underlying scripts themselves aren't individually memory-hungry. If any of your real scripts are heavy (large data processing, not thin wrappers), this changes — check actual script RSS under load.

3. Scheduler — absorbing the submission burst

Since DAGs now use native scheduling, the scheduler (not an external script) is what creates and fires all `T` peak task instances and up to `N` new DAG runs arriving close together.

Param	Default	Formula / Recommendation for N=25	Why
`max_dagruns_to_create_per_loop`	10	`≥ N` → **30**	Must create all N new dag_runs without queuing across multiple loop iterations
`max_dagruns_per_loop_to_schedule`	20	`≥ N × 1.2` → **35**	Same idea, one stage downstream
`max_tis_per_query`	16	`≥ T` → **64**	Must pull all T peak task instances per loop pass
`max_active_tasks_per_dag`	16	Keep unless your widest real DAG fan-out exceeds this	Per-DAG concurrency ceiling, independent of N
`max_active_runs_per_dag`	16	Lower to **1–2**	Each `dag_id` is one-time; no legitimate case for multiple concurrent runs of the same dag_id
`use_row_level_locking`	True	Keep True	Critical now — many DAG runs created/scheduled per loop increases DB row contention

Resource: `AirflowScheduler` at 2048:2 is fine at N=25 — this is a config-tuning problem at this scale, not a memory/CPU one. Watch scheduler loop duration under real burst load before assuming this holds at higher N.

4. DAG Processor — quick pickup is the whole point of `setu-dags`

Formula:

```
parsing_processes ≈ cores_allocated - 1  (min 2, since a burst of N files needs real parallelism)
```

At N=25, a burst of 25 new files landing together needs meaningful parallel parsing capacity. The current mapping (`AirflowDAGProcessor = 1024:1`) only supports `parsing_processes=1` by the formula — **that's a bottleneck at this load**, not a safe default.

Recommended resource change: bump `AirflowDAGProcessor` to **2048:2**, giving:

```
[dag_processor]
parsing_processes = 3
bundle_refresh_check_interval = 5
min_file_process_interval = 10
file_parsing_sort_mode = modified_time
```

And keep your bundle-specific `refresh_interval: 5` on `setu-dags`.

Expected discovery latency at N=25 burst: roughly `bundle_refresh_check_interval (5s) + queue_wait(parsing_processes=3, T_files=25, ~2s avg parse) + min_serialized_dag_fetch_interval (5s)`. With 3 parallel parsers and ~2s/file, 25 files clear in ~17s of pure parse time, plus polling overhead → expect DAGs visible/schedulable within **~25–35s** of a synchronized 25-user burst. This is the number to actually measure once running, not trust blindly.

`[core]` (serialization → API visibility, still relevant for dashboard status views even without the trigger polling loop):

```
min_serialized_dag_update_interval = 10
min_serialized_dag_fetch_interval = 5
```

5. API Server — lower stakes now, still needs real workers

With native scheduling, the api-server is **out of the critical path for job submission** — the dashboard returns after the file write, not after an Airflow round-trip. This removes your race-condition risk entirely for the submit flow. The api-server now only serves: dashboard status polling/UI, Flower-adjacent queries, admin actions — real but lower-intensity load.

Formula:

```
workers = (2 × cores) + 1      (safe default)
```

At `AirflowWebserver = 2048:2`:

```
[api]
workers = 5
```

Set via `AIRFLOW__API__WORKERS`, since it's not exposed as a compose env var by default in your current file — add it explicitly.

6. Database Connection Budget — this is the number that gates everything

Sum realistic pool usage, **not defaults** — at N=25 scale, default `pool_size=5 / max_overflow=10` (15/instance) is oversized for what you need and eats connection budget fast across 7 worker replicas.

Component	Instances	Recommended pool_size + overflow	Connections
api-server	5 workers	3 + 2 = 5	25
celery workers	7 replicas	3 + 2 = 5	35
scheduler	1	5 + 5 = 10	10
dag-processor	1	5 + 5 = 10	10
triggerer	1	2 + 2 = 4	4
Total			**~84**

Add ~30–40% headroom for admin CLI sessions, `airflow db check`, ad-hoc queries → **`max_connections = 150`**.

Config for each component's `sql_alchemy_conn` pool (set per-process, they don't have to match):

```
[database]
sql_alchemy_pool_size = 3
sql_alchemy_max_overflow = 2
```

(Scheduler/dag-processor can keep the larger 5+5 shown in the table — they're single instances, so the larger pool doesn't multiply the way the per-replica worker/api-server pools do.)

Memory check at `max_connections=150`:

```
150 × 8MB (CONN_MB)     = 1200MB
shared_buffers (~25%)   = 512MB   (at 2048MB container)
Postgres base overhead  ≈ 200MB
                          -------
                          1912MB → 93% of 2048MB. Too tight for comfort.
```

Recommended resource change: bump `AirflowDB` from `2048:2` to **`3072:2`** (3GB). This drops utilization to a safe ~62% and gives headroom for connection spikes, autovacuum, and query working memory without touching `max_connections` again as N grows moderately.

How to actually set `max_connections` — not via the multi-db init SQL script (that's per-database DDL; `max_connections` is cluster-wide in `postgresql.conf`). Add it as a startup command override in the compose block:

```yaml
command: ["postgres", "-c", "max_connections=150"]
```

Requires a full Postgres restart to take effect (not hot-reloadable).

7. Redis & Triggerer — not your bottleneck at this scale

Redis (`AirflowRedis`, 1024:1): used only as the Celery broker (result backend is Postgres per your compose). Broker load is proportional to queued-task message volume, not task runtime — at `T=50` peak in-flight tasks, message volume is trivial. **No change needed** at N=25. Revisit only if N grows by an order of magnitude.
Triggerer (`AirflowTriggerer`, 512:0.5): only matters for deferrable operators. `BashOperator` is not deferrable, so triggerer is essentially idle in this workload. **No change needed** — keep it minimal unless you introduce sensors/deferrable operators later.
Flower (`AirflowFlower`, 1024:1): monitoring only, not in any execution or submission path. **No change needed.**

8. Final Proposed Resource Table (N=25)

Component	Default mapping	Proposed	Changed?	Reason
AirflowWorker (per replica)	2048:2	2048:2 × **7 replicas**	Replica count↑	`T=50` peak tasks / `concurrency=8`
AirflowScheduler	2048:2	2048:2 (unchanged)	Config only	Burst absorbed via loop-size configs
AirflowWebserver / api-server	2048:2	2048:2 (unchanged)	Config only	`workers=5`, lower stakes post-native-scheduling
AirflowTriggerer	512:0.5	512:0.5 (unchanged)	—	Not used by this workload
AirflowDAGProcessor	1024:1	**2048:2**	Memory+cores↑	Needs `parsing_processes≥2` for burst pickup
AirflowFlower	1024:1	1024:1 (unchanged)	—	Monitoring only
AirflowRedis	1024:1	1024:1 (unchanged)	—	Broker load trivial at this scale
AirflowDB	2048:2	**3072:2**	Memory↑	Fit `max_connections=150` safely

9. Rescaling for a Different N

Quantity	Formula
`T` (peak active tasks)	`N × F`
`worker_replicas`	`ceil(T / (cores_per_worker × 4))`
`parallelism`	`worker_replicas × worker_concurrency × 1.15` (headroom), rounded
`max_dagruns_to_create_per_loop`	`≥ N`
`max_dagruns_per_loop_to_schedule`	`≥ N × 1.2`
`max_tis_per_query`	`≥ T`
`parsing_processes`	`cores_allocated_to_dag_processor - 1`, min 2
`api workers`	`(2 × cores_allocated_to_webserver) + 1`
Total DB connections	`(api_workers × api_pool) + (worker_replicas × worker_pool) + scheduler_pool + dagproc_pool + triggerer_pool`
`max_connections`	`total_connections × 1.3–1.4` headroom
`AirflowDB` memory	`(max_connections × CONN_MB) + shared_buffers + ~200MB base`, then round up to next comfortable tier

Before trusting any of this at a new N: re-measure `AVG_PARSE_S` and real per-task script resource usage at that scale — the formulas assume the same lightweight-bash-wrapper workload profile. If future DAGs run heavier in-process work (not just `bash_command` shell-outs), `worker_concurrency`'s `×4` I/O-bound multiplier and the worker memory sizing both need re-validation.

10. What to Actually Load-Test Before Trusting This

Synchronized burst of N real dashboard submissions → measure actual time from file-write to DAG schedulable/visible (validates Section 4's ~25–35s estimate).

Postgres active connection count under that same burst (validates Section 6's budget holds, not just the theoretical sum).

Scheduler loop duration (`scheduler_heartbeat_sec` cadence vs. actual loop time) during the burst — if loops start taking longer than `scheduler_heartbeat_sec=5`, that's your signal to raise scheduler CPU before raising N further.

Worker replica memory under real script execution, not synthetic bash — thin wrapper assumption needs confirming against your actual heaviest scripts.