# Designing HA and Robust Scheduler

**✅ 1. Use a Persistent Job Store**

**🔧 Why:**

To survive crashes, you need a job store that persists job definitions and states (e.g., next run time, misfire policies).

**✅ Recommendation:**

* Use **MongoDB**, **PostgreSQL**, or **Redis** as the **job store**.
* Avoid MemoryJobStore or SQLite in production.

**🔍 APScheduler setup:**

jobstores = {

'default': MongoDBJobStore(

host='mongodb-cluster-url',

db='scheduler\_db',

collection='jobs',

username='user',

password='pass'

)

}

**✅ 2. Enable Leader Election**

**🔧 Why:**

In distributed environments (like Kubernetes), **multiple replicas** of the scheduler can run. Without coordination, tasks may run **multiple times**.

**✅ Recommendation:**

* Use **Redis-based leader election** (redlock-py)
* Alternatively, use **Kubernetes leader election** via sidecar (e.g., Kubernetes Lease API) or distributed locks via etcd/ZooKeeper.

**✅ Sample (with redlock-py):**

lock = dlm.lock("scheduler\_lock", 10000)

if lock:

start\_scheduler()

...

dlm.unlock(lock)

**✅ 3. Enable replace\_existing=True in add\_job()**

**🔧 Why:**

If your app restarts or redeploys, jobs should not duplicate. This setting ensures jobs are updated in-place.

scheduler.add\_job(..., replace\_existing=True)

**✅ 4. Use misfire\_grace\_time and coalesce wisely**

**🔧 Why:**

If the scheduler is down during a scheduled time, you need to decide whether to **catch up**, **skip**, or **merge** multiple missed runs.

**Recommended Settings:**

job\_defaults = {

'coalesce': False, # Run all missed executions individually

'misfire\_grace\_time': 3600 # Allow job to run if missed within 1 hour

}

**✅ 5. Set Health Checks and Alerting**

**🔧 Why:**

You must know if the scheduler is down or misbehaving.

**Recommendations:**

* Add /health endpoint to scheduler service
* Monitor the job run outcomes
* Use Prometheus, Grafana, or centralized logging for visibility

**✅ 6. Log Job Execution and Errors**

**🔧 Why:**

To audit task runs and debug failures

**Best Practice:**

* Use structured logging (timestamp, job\_id, status)
* Log to files or external logging systems (e.g., ELK stack)

**✅ 7. Graceful Shutdown**

**🔧 Why:**

To ensure in-progress jobs are not killed mid-way during pod restarts or SIGINT/SIGTERM signals

scheduler.shutdown(wait=True)

**✅ 8. Use Kubernetes Readiness/Liveness Probes**

**🔧 Why:**

Ensure Kubernetes restarts the pod if it becomes unresponsive.

livenessProbe:

httpGet:

path: /health

port: 8080

initialDelaySeconds: 10

periodSeconds: 30

**✅ 9. Retry Logic in Tasks**

**🔧 Why:**

If an API fails, retry a few times before marking the job as failed.

**Recommendation:**

Use tenacity, backoff, or manual retry logic:

@retry(stop=stop\_after\_attempt(3), wait=wait\_fixed(10))

def call\_api():

**✅ 10. Testing & Safe Deployments**

**🔧 Best Practices:**

* Use **dynamic trigger times** (e.g., 1-2 minutes from now) for test jobs
* Validate behavior in lower environments
* Use start\_date=datetime.utcnow() in cron jobs to avoid skipping

**✅ Summary Table**

| **Feature / Config** | **Description** |
| --- | --- |
| Persistent Job Store | Use MongoDB, Postgres, or Redis |
| Leader Election | Use redlock-py, Kubernetes Lease, or ZooKeeper |
| replace\_existing=True | Avoid job duplication on restarts |
| misfire\_grace\_time, coalesce | Control job behavior during downtime |
| Logging | Centralized logs with success/failure info |
| Health Checks | Monitor scheduler availability |
| Retry Logic in Jobs | Resilience for flaky downstream systems |
| Graceful Shutdown | Allow jobs to complete before shutdown |
| Probes in K8s | Liveness/readiness to auto-recover |
| Testing Strategy | Use dynamic trigger times and lower env validation |

## Redis based leader election system -

Using **Zookeeper**, **etcd**, or **Redis** for leader election is a more robust and production-grade solution, especially for distributed systems running in Kubernetes or across VMs. Let’s walk through how you can use **Redis-based leader election** (simplest to integrate with Python + APScheduler).

**✅ Option: Leader Election with Redis (Best for Python + Celery/APScheduler)**

You can use **Redis** with a distributed lock using the [redis.lock](https://redis.readthedocs.io/en/stable/#redis.lock.Lock) or higher-level tools like [redlock-py](https://github.com/SPSCommerce/redlock-py) or [fasteners](https://pypi.org/project/fasteners/).

**🔧 Setup**

**Install Redis client and redlock:**

pip install redis redlock-py

**✅ Code Example Using redlock-py (Redis-based Locking)**

from redlock import Redlock

import time

import logging

from scheduler\_core import SchedulerManager

# Connect to your Redis cluster/nodes

dlm = Redlock([

{"host": "your-redis-host", "port": 6379, "db": 0},

# Add more Redis nodes if using cluster mode

])

LOCK\_KEY = "scheduler-leader-lock"

LOCK\_TTL\_MS = 30000 # Lock expires in 30 seconds

def run\_scheduler\_as\_leader():

while True:

try:

lock = dlm.lock(LOCK\_KEY, LOCK\_TTL\_MS)

if lock:

logging.info("Acquired Redis lock. I am the leader.")

scheduler = SchedulerManager()

scheduler.schedule\_all\_jobs()

scheduler.start()

try:

while True:

# Keep lock alive by refreshing

if not dlm.lock(LOCK\_KEY, LOCK\_TTL\_MS):

logging.warning("Lost Redis lock. Shutting down scheduler...")

scheduler.shutdown()

break

time.sleep(10)

except KeyboardInterrupt:

scheduler.shutdown()

finally:

dlm.unlock(lock)

else:

logging.info("Another instance is the leader. Retrying...")

time.sleep(5)

except Exception as e:

logging.error(f"Error in acquiring Redis lock: {e}")

time.sleep(5)

**🔑 What This Does**

* Tries to acquire a **distributed lock** in Redis.
* If it gets the lock: it starts the scheduler.
* It keeps refreshing the lock every 10s.
* If it **loses** the lock (e.g., due to network issues), it **shuts down the scheduler** to avoid duplicate runs.

**🔄 Benefits of Redis-based Leader Election**

| **Feature** | **Description** |
| --- | --- |
| ✅ Fast | Redis is low-latency and ideal for distributed locking. |
| ✅ Simple | One small dependency (redlock-py). |
| ✅ Reliable | Works across multiple pods/hosts. |
| ✅ No K8s dependency | Doesn’t require ConfigMaps or K8s client logic. |

**🔒 Redis Cluster Considerations**

If you're using Redis Cluster:

* Ensure that all Redis nodes are listed in the Redlock constructor.
* All nodes should be reachable from your scheduler pods.

python

CopyEdit

Redlock([

{"host": "redis-node-1", "port": 6379, "db": 0},

{"host": "redis-node-2", "port": 6379, "db": 0},

{"host": "redis-node-3", "port": 6379, "db": 0}

])

**🚀 Summary**

| **Tool** | **Use Case** | **Good For** |
| --- | --- | --- |
| **Redis + redlock** | Lightweight, fast, easy | Most deployments |
| **Zookeeper** | Stateful apps with strict consistency | High reliability |
| **etcd** | Kubernetes-native or heavy infra | Advanced setups |

**⚠️ Common Pitfalls to Avoid**

* **Don’t use DEL directly** to release the lock — always check value first
* Use **short TTLs** to ensure locks don’t persist after crashes
* Avoid holding locks too long; keep the critical section short
* Ensure **clock drift** is minimal across nodes (important for time-based safety)