Worker System Rework

From imperative jobs to managed pipelines

Moving Responsibility to the Right Layer

Before: a single job function owned domain logic, retry policy, successor enqueueing, state updates, and Slack alerts.

After: jobs do domain work. Managers own coordination, state, retries, recovery, and auditability.

Dimension	Imperative	Declarative
Adding a step	One enqueue call inside the upstream job	One entry in the pipeline definition
Adding a parallel branch	Every upstream job must know its siblings and enqueue each one	Coordinator enqueues all ready jobs after each completion
Debugging a failure	Cross-reference Redis state, ARQ keys, and application columns	Query job run rows: status, failure category, retry history
Retry behavior	Each job writes its own retry logic; semantics drift	Central failure categories; one policy applies everywhere
Crash recovery	Application state and queue state diverge silently; manual repair	Scheduled cleanup detects stalls and re-enqueues or skips

Before: The Shape Of The System

The old worker package was tiny in file count, but dense in responsibility:

```
src/mavedb/worker/
  settings.py
  jobs.py        # 1,766 lines
```

ARQ knew the function list. The functions knew everything else.

```python
# old src/mavedb/worker/settings.py
BACKGROUND_FUNCTIONS = [
    create_variants_for_score_set,
    variant_mapper_manager,
    map_variants_for_score_set,
    submit_score_set_mappings_to_ldh,
    link_clingen_variants,
    link_gnomad_variants,
    submit_score_set_mappings_to_car,
]


async def on_job_start(ctx):
    ctx["db"] = SessionLocal()
    ctx["hdp"] = cdot_rest()
    ctx["state"] = {}


job_timeout = 5 * 60 * 60  # Keep jobs alive for a long while...
```

Before: State Was Not A System

The durable state model was mostly application tables plus Redis plus logs.

None of it could answer:

- Which jobs were created for this score set?
- Which ones ran, retried, skipped, or failed?
- Which dependency blocked the workflow?
- Which attempt produced this error?

We realized the importance of the answers to those questions only after we were unable to answer them.

After: State Is A First-Class Model

The new system stores workflow state in Postgres and uses ARQ as the execution transport.

```
Pipeline
  owns many JobRun rows
  has name, status, correlation_id, pipeline_params

JobRun
  has function, params, status, progress, retry_count,
  failure_category, error_message, metadata_["retry_history"]

JobDependency
  says job A depends on job B, with a dependency type

VariantAnnotationStatus
  says variant X had annotation Y succeed, fail, or skip
```

This gives us an audit log that is queryable even after Redis keys and logs expire.

We paid the infrastructure cost up front, instead of paying a smaller maintenance cost while writing each job.

Problem 1: Jobs Were Imperative Pipelines

Old jobs had to know what came next.

```
# old link_clingen_variants(...)
if job_succeeded:
    gnomad_linking_job_id = None
    try:
        new_job = await redis.enqueue_job(
            "link_gnomad_variants",
            correlation_id,
            score_set.id,
        )

        if new_job:
            gnomad_linking_job_id = new_job.job_id
            logging_context["link_gnomad_variants_job_id"] = gnomad_linking_job_id
            logger.info(msg="Queued a new gnomAD linking job.", extra=logging_context)
        else:
            raise LinkingEnqueueError()
```

Business and workflow logic were mixed together.

Why That Got Expensive

Imperative enqueueing made every job author answer pipeline questions:

- What is the next job?
- What args does it need?
- Should siblings continue?

That logic was repeated, and each repetition drifted.

After: Pipelines Are Data

The dependency graph now lives in one place.

```
# src/mavedb/lib/workflow/definitions.py
[
    ...,
    {
        "key": "link_gnomad_variants",
        "function": "link_gnomad_variants",
        "type": JobType.MAPPED_VARIANT_ANNOTATION,
        "params": {
            "correlation_id": None,
            "score_set_id": None,
        },
        "dependencies": [("warm_clingen_cache", DependencyType.SUCCESS_REQUIRED)],
    },
    ...,
]
```

The job no longer enqueues its successor. It returns an outcome, and coordination decides what is ready. Job dependencies make pipeline flow explicit in definitions.

After: Composition Became Possible

The same annotation block is reused across pipeline shapes.

```
def annotation_pipeline_job_definitions(upstream_mapping_key="map_variants_for_score_set"):
    mapping_dep = [(upstream_mapping_key, DependencyType.SUCCESS_REQUIRED)] if upstream_mapping_key else []
    return [
        {"key": "submit_score_set_mappings_to_car", "dependencies": mapping_dep, ...},
        {"key": "warm_clingen_cache", "dependencies": [("submit_score_set_mappings_to_car", SUCCESS_REQUIRED)], ...},
        {"key": "link_gnomad_variants", "dependencies": [("warm_clingen_cache", SUCCESS_REQUIRED)], ...},
        {"key": "populate_vep_for_score_set", "dependencies": [("submit_score_set_mappings_to_car", SUCCESS_REQUIRED)], ...},
    ]

PIPELINE_DEFINITIONS = {
    "validate_map_annotate_score_set": [create_variants, map_variants, *annotations],
    "map_annotate_score_set":      [map_variants, *annotations],
    "annotate_score_set":          annotation_pipeline_job_definitions(upstream_mapping_key=None),
}
```

One definition block replaced multiple places where jobs had to remember the workflow.

Problem 2: Jobs Mixed Concerns

The old mapping job was not just a mapping job.

```
# old map_variants_for_score_set(...)
score_set.mapping_state = MappingState.processing
db.commit()

mapping_results = await loop.run_in_executor(ctx["pool"], blocking)

except Exception as e:
    db.rollback()
    score_set.mapping_errors = {
        "error_message": f"Mapping will be automatically retried up to 5 times (attempt {attempt}/5)."
    }
    db.commit()

    send_slack_error(e)
    await redis.lpush(MAPPING_QUEUE_NAME, score_set.id)
    await enqueue_job_with_backoff(
        redis, "variant_mapper_manager", attempt, MAPPING_BACKOFF_IN_SECONDS,
        correlation_id, updater_id
    )
```

Domain logic, score set state, job state, retry policy, Slack, Redis queue repair, and pipeline scheduling all lived together.

After: Job Code Has A Smaller Contract

Current jobs still do real domain work, but they communicate through one outcome type.

```python
@with_pipeline_management
async def map_variants_for_score_set(ctx, job_id, job_manager):
    job = job_manager.get_job()
    validate_job_params(["score_set_id", "correlation_id", "updater_id"], job)

    score_set = job_manager.db.scalars(
        select(ScoreSet).where(ScoreSet.id == job.job_params["score_set_id"])
    ).one()

    # mapping domain logic...
    annotation_manager.flush()
    job_manager.db.flush()

    return JobExecutionOutcome.succeeded(
        data={"score_set_id": score_set.id, "mapped_count": successful_mapped_variants}
    )
```

The job still owns mapping decisions. It doesn't own anything else.

After: State Decisions Made In Decorators

The decorator owns the boring but critical outer shell — including: how do we prove this job already ran?

```
# src/mavedb/worker/lib/decorators/job_management.py
job_manager = JobManager(db_session, redis_pool, job_id)
kwargs["job_manager"] = job_manager

if job_manager.get_job_status() in TERMINAL_JOB_STATUSES:
    return JobExecutionOutcome.skipped(data={"reason": "already terminal"})

job_manager.start_job()
db_session.commit()

result = await func(*args, **kwargs)

if result.status == JobStatus.FAILED:
    job_manager.fail_job(result=result)
    if not job_manager.should_retry():
        send_slack_job_failure(...)
elif result.status == JobStatus.SKIPPED:
    job_manager.skip_job(result=result)
else:
    job_manager.succeed_job(result=result)
```

Every job now gets the same lifecycle behavior without rewriting it. One line decorating new job code gives it all our existing lifecycle rules.

After: Pipeline Coordination Moved To A Manager

`PipelineManager` runs after each job completes.

```
# src/mavedb/worker/lib/managers/pipeline_manager.py
for job in self.get_pending_jobs():
    job_manager = JobManager(self.db, self.redis, job.id)

    if self.can_enqueue_job(job):
        job_manager.prepare_queue()
        jobs_to_queue.append(job)
        continue

    should_skip, reason = self.should_skip_job_due_to_dependencies(job)
    if should_skip:
        job_manager.skip_job(
            result=JobExecutionOutcome.skipped(data={"reason": reason})
        )

self.db.commit()  # release row locks before async Redis enqueue

for job in jobs_to_queue:
    await self._enqueue_in_arq(job, is_retry=job.retry_count > 0)
```

The important part is not just "enqueue ready jobs." It is also "mark impossible jobs as intentionally skipped" and "defer coordination of things which are not ready."

Problem 3: ARQ Is A Queue, Not A Workflow Engine

ARQ gives us a function call, a Redis key, a status, and timeouts.

That is enough to execute jobs. It is not enough to answer workflow questions.

Question	ARQ alone	New worker state
Why did this job run?	function args	`JobDependency` + `Pipeline`
Why did it not run?	hard to tell	`SKIPPED` with reason
Should this retry?	coarse queue behavior	`failure_category` + `retry_count`
What happened last attempt?	maybe logs	`JobRun.metadata_["retry_history"]`
What is the score set workflow status?	not modeled	`Pipeline.status`

ARQ is still useful. It just stopped being the (extremely limited) source of truth.

Problem 4: Custom Queue Logic Leaked Everywhere

The old mapper had a separate Redis queue and a manager job to enforce serial execution.

```
# old variant_mapper_manager(...)
queue_length = await redis.llen(MAPPING_QUEUE_NAME)
queued_id = await redis.rpop(MAPPING_QUEUE_NAME)

mapping_job_id = await redis.get(MAPPING_CURRENT_ID_NAME)
mapping_job_status = (await Job(job_id=mapping_job_id, redis=redis).status()).value

if not mapping_job_id or mapping_job_status in (
    JobStatus.not_found,
    JobStatus.complete,
):
    new_job = await redis.enqueue_job(
        "map_variants_for_score_set",
        correlation_id,
        queued_score_set.id,
        updater_id,
        attempt,
    )
else:
    new_job = await redis.enqueue_job(
        "variant_mapper_manager",
        correlation_id,
        updater_id,
        attempt,
        _defer_by=timedelta(minutes=5),
    )
```

The workflow depended on Redis list discipline plus ARQ status interpretation. You can probably imagine this was super flaky!

After: Dependency Resolution Has Two Questions

For every pending job, the coordinator asks:

```
def can_enqueue_job(job):
    # all dependencies currently satisfied
    return all(
        job_dependency_is_met(dep.type, upstream.status) for dep, upstream in deps
    )


def should_skip_job_due_to_dependencies(job):
    # any dependency has become permanently impossible
    # SUCCESS_REQUIRED + upstream FAILED/ERRORED/SKIPPED/CANCELLED => skip
    return unreachable, reason
```

Upstream state	Coordinator decision
Still `PENDING` / `RUNNING`	Wait
`SUCCEEDED`	Enqueue
`FAILED` under `SUCCESS_REQUIRED`	Skip downstream without retry
`FAILED` under `COMPLETION_REQUIRED`	Enqueue anyway

It also answers: *should the job run after partial success?* The `COMPLETION_REQUIRED` type enqueues downstream even after an upstream failure; `SUCCESS_REQUIRED` skips it.

This is where imperative "what next?" logic became reusable infrastructure.

Problem 5: The Pipeline Was Mostly Synchronous

The old workflow could only move forward when the current job decided to enqueue the next one.

```
create variants
  -> mapping manager
     -> map variants
        -> submit to CAR
           -> submit to LDH
              -> link ClinGen
                 -> link gnomAD
```

Branches were possible only if a job manually enqueued each branch.

That made throughput a property of job bodies, not of the workflow graph.

After: The Graph Can Fan Out

Once mapping and CAR submission complete, independent annotation branches can be ready together.

```
create_variants
  -> map_variants
     -> submit_score_set_mappings_to_car
        -> warm_clingen_cache
           -> link_gnomad_variants
           -> refresh_clinvar_controls
           -> populate_hgvs_for_score_set
           -> populate_variant_translations_for_score_set
        -> populate_vep_for_score_set

     -> submit_uniprot_mapping_jobs_for_score_set
        -> poll_uniprot_mapping_jobs_for_score_set
```

The coordinator can enqueue every ready sibling. A job does not need to know its siblings exist.

Concurrency Still Has A Tradeoff

We did not simply turn every knob to "more concurrent."

```
# src/mavedb/worker/settings/worker.py
MAX_JOBS = 2
JOB_TIMEOUT_SECONDS = 2 * 60 * 60

# Limit concurrency to prevent event loop starvation from sync psycopg2 DB operations.
# TODO#715 Migrate to psycopg3 async driver to safely increase concurrency.
```

The system now has two layers of concurrency control:

- **Pipeline-level concurrency:** enqueue independent ready jobs.
- **Job-level concurrency:** use bounded async work inside I/O-heavy jobs.

`MAX_JOBS = 2` is a deliberate compromise until DB access is async-safe.

Problem 6: Long Jobs Needed Local Workarounds

The old worker used very long ARQ timeouts and job-local concurrency patches.

```
# old worker settings
ctx["pool"] = futures.ProcessPoolExecutor()
job_timeout = 5 * 60 * 60

# old mapping job
blocking = functools.partial(vrs.map_score_set, mapping_urn)
mapping_results = await loop.run_in_executor(ctx["pool"], blocking)
```

That was the right local fix for blocking work, but it did not solve pipeline throughput or observability.

It kept the worker alive. It did not make the workflow manageable.

After: Bounded Async Inside Annotation Jobs

VEP is the clearest example: batch the work, run independent batches concurrently, and checkpoint progress.

```
# src/mavedb/worker/jobs/external_services/vep.py
_VEP_BATCH_SIZE = 200
_RECODER_BATCH_SIZE = 25
_RECODER_CONCURRENCY = 5

semaphore = asyncio.Semaphore(_RECODER_CONCURRENCY)


async def _recoder_with_semaphore(batch, batch_idx, total):
    async with semaphore:
        result = await run_variant_recoder(batch)
        completed_recoder_batches += 1
        job_manager.update_progress(
            33 + int(completed_recoder_batches / total * 33),
            100,
            f"Completed Variant Recoder batch {completed_recoder_batches}/{total}",
        )
        return result


recoder_results = await asyncio.gather(*tasks, return_exceptions=True)
```

This makes timeout pressure visible and bounded instead of hidden inside one huge request.

Problem 7: CLI Scripts Duplicated Worker Logic

The old operational scripts were separate implementations of worker-shaped work.

```
scripts/
  clingen_car_submission.py
  clingen_ldh_submission.py
  link_clingen_variants.py
  link_gnomad_variants.py
  map_to_uniprot_id_from_mapped_metadata.py
  populate_mapped_variants.py
  populate_variant_translations.py
  refresh_clinvar_variant_data.py
  vep_functional_consequence.py
```

That meant two ways to annotate data, two places to fix bugs, and no shared job state.

Before: A Script Could Drift From The Worker

The old VEP script had its own blocking implementation.

```
# old scripts/vep_functional_consequence.py
def run_variant_recoder(missing_hgvs):
    recoder_response = requests.post(
        f"{ENSEMBL_API_URL}/variant_recoder/human",
        headers=headers,
        json={"ids": list(missing_hgvs)},
    )


def get_functional_consequence(hgvs_strings):
    response = requests.post(
        f"{ENSEMBL_API_URL}/vep/human/hgvs",
        headers=headers,
        json={"hgvs_notations": hgvs_strings},
    )
```

These scripts could re-annotated data, but it wasn't guaranteed they'd give you the same results as the job logic would.

After: Scripts Reuse The Worker Contract

`run_job.py` creates a `JobRun` from the same standalone job definitions.

```
# src/mavedb/scripts/run_job.py
lookup = {
    job_def["function"]: (func, job_def)
    for func, job_def in STANDALONE_JOB_DEFINITIONS.items()
}

job_run = job_factory.create_job_run(
    job_def=job_def,
    pipeline_id=None,
    correlation_id=correlation_id,
    pipeline_params=pipeline_params,
)

if enqueue:
    await redis.enqueue_job(
        job_run.job_function, job_run.id, _job_id=arq_job_id(job_run)
    )
else:
    await job_func(ctx, job_run.id)  # decorator injects JobManager
```

Operators can run the same job locally or enqueue it without a parallel implementation.

After: Pipelines Are Scriptable Too

`run_pipeline.py` creates the same `Pipeline`, `JobRun`, and `JobDependency` records the API creates.

```
# src/mavedb/scripts/run_pipeline.py
pipeline, entrypoint = pipeline_factory.create_pipeline(
    pipeline_name=pipeline_name,
    creating_user=user,
    pipeline_params={
        "correlation_id": correlation_id,
        "score_set_id": score_set.id,
        "updater_id": user.id,
    },
)

await redis.enqueue_job(
    entrypoint.job_function,
    entrypoint.id,
    _job_id=arq_job_id(entrypoint),
)
```

The CLI is now an entrypoint into the system, not a fork of the system.

Problem 8: Failure Recovery Was Ad Hoc

Every job also had to answer: *how should this retry?*

Old retry behavior lived inside individual jobs.

```
# old enqueue_job_with_backoff(...)
limit_reached = attempt > BACKOFF_LIMIT
if not limit_reached:
    backoff = backoff * (2**attempt)
    attempt = attempt + 1
    new_job = await redis.enqueue_job(
        job_name,
        *args,
        attempt,
        _defer_by=timedelta(seconds=backoff),
    )
```

That made retry semantics depend on which job you were reading.

Some errors retried. Some alerted and stopped. Some requeued a manager job. Some were manual. Some were invisible.

After: Failures Are Classified

`JobManager.should_retry()` centralizes the decision.

```
# conceptually
def should_retry(job):
    return (
        job.status in {FAILED, ERRORED}
        and job.retry_count < job.max_retries
        and job.failure_category in RETRYABLE_FAILURE_CATEGORIES
    )
```

Failure category	Retry?	Why
`NETWORK_ERROR`	yes	external system may recover
`TIMEOUT`	yes	transient slowness
`SERVICE_UNAVAILABLE`	yes	service-level transient
`DATA_ERROR`	no	rerun cannot fix input data
`VALIDATION_ERROR`	no	rerun cannot make invalid data valid

Retries are now a policy, not a custom branch in each job.

After: Retry Attempts Are Auditable

Retry does more than set `status = PENDING`.

```
# src/mavedb/worker/lib/managers/job_manager.py
job_run.status = JobStatus.PENDING
job_run.retry_count = (job_run.retry_count or 0) + 1
job_run.error_message = None
job_run.error_traceback = None
job_run.started_at = None
job_run.finished_at = None

job_run.metadata_.setdefault("retry_history", []).append({
    "attempt": job_run.retry_count,
    "timestamp": datetime.now().isoformat(),
    "status": previous_status,
    "error_message": previous_error_message,
    "reason": reason,
})
```

This is the audit log we did not have before: which attempts happened, why, and what changed.

Problem 9: Workers Could Get Stuck

And what happens if enqueueing itself fails? The old system had several limbo states:

- DB said a score set was processing, but ARQ had no runnable job.
- Redis queue said mapping was pending, but the manager job had died.
- A job timed out after hours, but no durable job row explained the attempt.
- A transient failure required manual rerun.

There was often no way to recover from this state without manual intervention. The new system turns recovery into a scheduled job.

After: Recovery Is Dependency-Aware

`cleanup_stalled_jobs` runs as a managed cron job and handles four stall classes.

Stuck item	Likely cause	New resolution
`QUEUED` job absent from ARQ	crashed after DB state, before Redis enqueue	fail + retry + enqueue
`RUNNING` job	worker died mid-execution	fail + retry if category allows
`PENDING` job	coordination crashed or dependency became unreachable	enqueue if ready, skip if impossible
non-terminal pipeline with no active jobs	final coordination did not run	rerun `coordinate_pipeline()`

The key improvement: a permanently unreachable job is marked `SKIPPED`, not left as mystery `PENDING`.

Problem 10: Repeated ClinGen Work Was Uncached

Multiple annotation jobs ask ClinGen for overlapping allele data.

Old behavior made repeated external calls across jobs and runs. This made jobs slower than necessary, and risked overwhelming ClinGen when jobs fanned out under the new workflow.

New behavior has two pieces:

- Cache the allele data lookup for 24 hours.
- Pre-warm the cache before downstream ClinGen-heavy annotation jobs run.

After: Cache Then Fan Out

```
# src/mavedb/lib/clingen/allele_registry.py
@cached(
    ttl=CACHE_TTL_SECONDS,
    key_builder=clingen_cache_key_builder,
    cache=CACHE_CLASS,
    **CACHE_CONFIG,
)
async def get_clingen_allele_data(clingen_allele_id: str):
    response = await loop.run_in_executor(
        None, requests.get, f"{CLINGEN_API_URL}/{clingen_allele_id}"
    )
    ...
```

```
# src/mavedb/worker/jobs/external_services/clingen_cache.py
semaphore = asyncio.Semaphore(CLINGEN_CACHE_WARMING_CONCURRENCY)


async def fetch_one(allele_id):
    async with semaphore:
        await get_clingen_allele_data(allele_id)
        return allele_id, True, None


for completed_task in asyncio.as_completed([fetch_one(a) for a in allele_ids]):
    allele_id, success, exc = await completed_task
    job_manager.update_progress(...)
```

The pipeline dependency makes the cache warm-up an explicit prerequisite.

Problem 11: "The Job Ran" Was Not Enough

A job-level success did not tell us why an annotation was missing.

For a variant without a VEP consequence, the old answers were often:

- maybe VEP never ran
- maybe VEP ran but skipped it
- maybe the HGVS string was missing
- maybe Variant Recoder failed
- maybe an external service rejected it

These answers aren't about job state, but **outcome** state.

After: Variant Annotation Status

Each annotation attempt writes an outcome per variant and annotation type.

```
# src/mavedb/lib/annotation_status_manager.py
annotation_manager.add_annotation(
    variant_id=mapped_variant.variant_id,
    annotation_type=AnnotationType.VEP_FUNCTIONAL_CONSEQUENCE,
    status=AnnotationStatus.FAILED,
    failure_category=AnnotationFailureCategory.EXTERNAL_REFERENCE_NOT_FOUND,
    annotation_data={
        "error_message": "VEP could not determine a functional consequence, even after Variant Recoder fallback.",
    },
)

annotation_manager.flush()
```

`JobRun` tells us the job ran. `VariantAnnotationStatus` tells us what the job did.

After: Current Status Plus History

`VariantAnnotationStatus` is append-friendly and version-aware, mirroring how our `MappedVariants` already work.

```python
class VariantAnnotationStatus(Base):
    variant_id: int
    annotation_type: str
    version: str | None
    status: AnnotationStatus  # success, failed, skipped
    failure_category: str | None
    error_message: str | None
    annotation_metadata: dict | None
    current: bool
    job_run_id: int | None
```

The manager retires old `current=True` rows and inserts new rows in batches. We have to be careful here; Sometimes we want to have many current annotations for different annotations versions (like ClinVar).

That gives support/debugging for both the current answer and a historical trail.

After: Queryable Annotation Outcomes

The view intentionally uses outer joins so missing data stays visible.

```
# src/mavedb/models/variant_annotation_view.py
definition = (
    select(
        Variant.urn,
        ScoreSet.urn,
        MappedVariant.vep_functional_consequence,
        VariantAnnotationStatus.annotation_type,
        VariantAnnotationStatus.status,
        VariantAnnotationStatus.error_message,
    )
    .select_from(Variant)
    .join(ScoreSet, ScoreSet.id == Variant.score_set_id)
    .outerjoin(MappedVariant, MappedVariant.variant_id == Variant.id)
    .outerjoin(
        VariantAnnotationStatus, VariantAnnotationStatus.variant_id == Variant.id
    )
)
```

Now "why is this annotation absent?" can be answered without reverse-engineering a job log.

Operator View

The new CLI summarizes current annotation status.

```
poetry run python -m mavedb.scripts.variant_annotations show-score-set urn:mavedb:00000097-0-2

Score set: urn:mavedb:00000097-0-2  (3893 variants)

------------------------------------------------
ANNOTATION TYPE                  STATUS     COUNT
------------------------------------------------
clingen_allele_id                success    2859
clinvar_control                  skipped    26220
clinvar_control                  success    20496
gnomad_allele_frequency          skipped    2818
gnomad_allele_frequency          success    43
mapped_hgvs                      skipped    1034
mapped_hgvs                      success    2859
variant_translation              skipped    101
variant_translation              success    2758
vep_functional_consequence       success    2859
vrs_mapping                      failed     1034
vrs_mapping                      success    2859
```

The important shift: we no longer need to infer annotation health from raw `mapped_variants` state.

Problem 12: Failure Noise Was Built In

In the old system, many catch blocks immediately alerted.

```
# old pattern repeated throughout jobs.py
except Exception as e:
    send_slack_error(e)
    send_slack_message(text=text % score_set.urn)
    logger.error(
        msg="This job will not be retried.",
        extra=logging_context,
    )
    return {"success": False, "retried": False, "enqueued_job": None}
```

Because ARQ saw failures independently, attempts and downstream failures could become alert noise.

After: Alert On Terminal Failure

The decorator can suppress noise because it knows retry state.

```
if result.status == JobStatus.FAILED:
    job_manager.fail_job(result=result)

    if not job_manager.should_retry():
        send_slack_job_failure(
            job_urn=job.urn,
            job_function=job.job_function,
            failure_category=str(result.failure_category or ""),
            retry_count=job.retry_count,
            max_retries=job.max_retries,
            will_retry=False,
        )
```

Transient attempt 1 can stay quiet. Terminal attempt 3 can send an alert with full context.

Edge-Case Infrastructure Details

Several details are not obvious from the high-level design.

Detail	Problem it solves
`ContextVar` task-local DB sessions	Concurrent ARQ tasks cannot corrupt each other's session state
Commit before async Redis enqueue loop	Releases Postgres row locks before `await` yields to jobs that may start immediately
`arq_job_id(job) = f"{job.urn}#{retry_count}"`	Each retry attempt gets a fresh ARQ key namespace
`STARTABLE_JOB_STATUSES` includes `RUNNING`	ARQ redelivery after worker crash can recover stale running rows

These small nuances and implementation details make the larger design reliable.

Testing: The Rewrite Changed The Unit Boundaries

Old jobs were hard to test because one function could touch:

- database state
- Redis queues
- ARQ job status
- Slack
- pipeline ordering
- retry policy
- external services

The new system lets us test those concerns independently.

Testing: What Became Testable

Layer	Tests can assert
`JobManager`	valid transitions, progress, retry history, failure categories
`PipelineManager`	dependency resolution, skipping, cancellation, fan-out, terminal status
Decorators	lifecycle shell, exception handling, retry-aware Slack, test-mode bypass
Jobs	domain outcome and database writes, with external services mocked at boundaries
Cleanup cron	queued/running/pending/stuck-pipeline recovery paths

Current worker tests cover managers, decorators, pipeline start, cleanup, variant processing, and every external-service job.

Tests are now easier to write, easier to maintain, and failures are easier to reason about.

Other Problems Worth Naming

Smaller old-system issues also shaped the rewrite.

Old problem	New answer
Shared mutable worker `ctx` made concurrent DB sessions risky	Task-local sessions via `ContextVar`
Retry attempts could reuse ARQ Redis keys	`arq_job_id(job)` includes `retry_count`
Jobs could be cancelled in DB but still sitting in Redis	Pre-execution terminal-state guard
Enqueueing while DB row locks were held could deadlock	Commit before async Redis enqueue loop
"Did not run" was indistinguishable from "stuck"	Explicit `SKIPPED` outcomes with reasons
The worker package was hard to navigate	Domain modules plus a central registry

These are not headline features, but they remove whole classes of weird behavior we often observed in production.

Current Gaps

The rewrite gives us a better control plane, but there are still unfinished edges.

Gap	Why it matters
Score sets still carry mapping/processing state	Some state is duplicated outside `JobRun` and annotation rows
`MAX_JOBS = 2` is still conservative	Sync DB calls can starve the event loop under heavier concurrency
`DependencyType` is binary	No first-class optional, best-effort, or degraded-mode dependency semantics
Flush-before-return is a convention	Tests rely on it, but the type system cannot enforce it
External submission idempotency is implicit	Retries assume downstream services tolerate duplicate submissions
Annotation queries use a non-materialized view	Fine now; likely expensive as annotation history grows

Under the rewrite, these gaps are much more tangible and addressable thatn they would have been under the old system.

The Big Before/After

	Old workflow	New workflow
Pipeline orchestration	Jobs enqueue next jobs	Pipelines declare dependencies
State model	`ctx["state"]`, Redis, logs	`Pipeline`, `JobRun`, `JobDependency`, annotation rows
Failure handling	Retry branches inside jobs	Central failure categories and retry policy
CLI tooling	Scripts duplicate worker logic	Scripts create real jobs and pipelines
Concurrency	Long synchronous chains	DAG coordination plus bounded job-level async
Alerting	Alerts per failure site	Alerts on terminal managed outcomes
Annotation observability	Raw SQL/debugging for missing annotations	Variant-level annotation status and view
Audit trail	No durable job audit trail	Every run and retry is a row

The Takeaway

The old system treated a job as the unit of work, the unit of orchestration, the unit of state, and the unit of observability.

That was fine while the workflow was simple.

The new system separates those units:

- **Jobs** do domain work.
- **Pipelines** define workflow shape.
- **Managers** own state transitions and recovery.
- **Annotation status** records what happened inside the job.
- **ARQ** moves execution from here to there.

Next Steps

- Expose pipeline and job-run status through API endpoints, then surface it in the UI.
- Expose annotation outcomes in the score set UI so missing annotations are explainable without SQL.
- Move score set processing and mapping state toward derived state from `JobRun` and annotation records.
- Materialize or otherwise optimize `v_variant_annotations` before annotation history gets large.
- Audit external-service jobs for explicit idempotency guarantees before increasing retry/fan-out pressure.
- Migrate worker DB access toward async-safe patterns so `MAX_JOBS` can rise safely.

The system is now structured enough that we can make these choices with thought when they make sense for the project and not as emergency repairs.