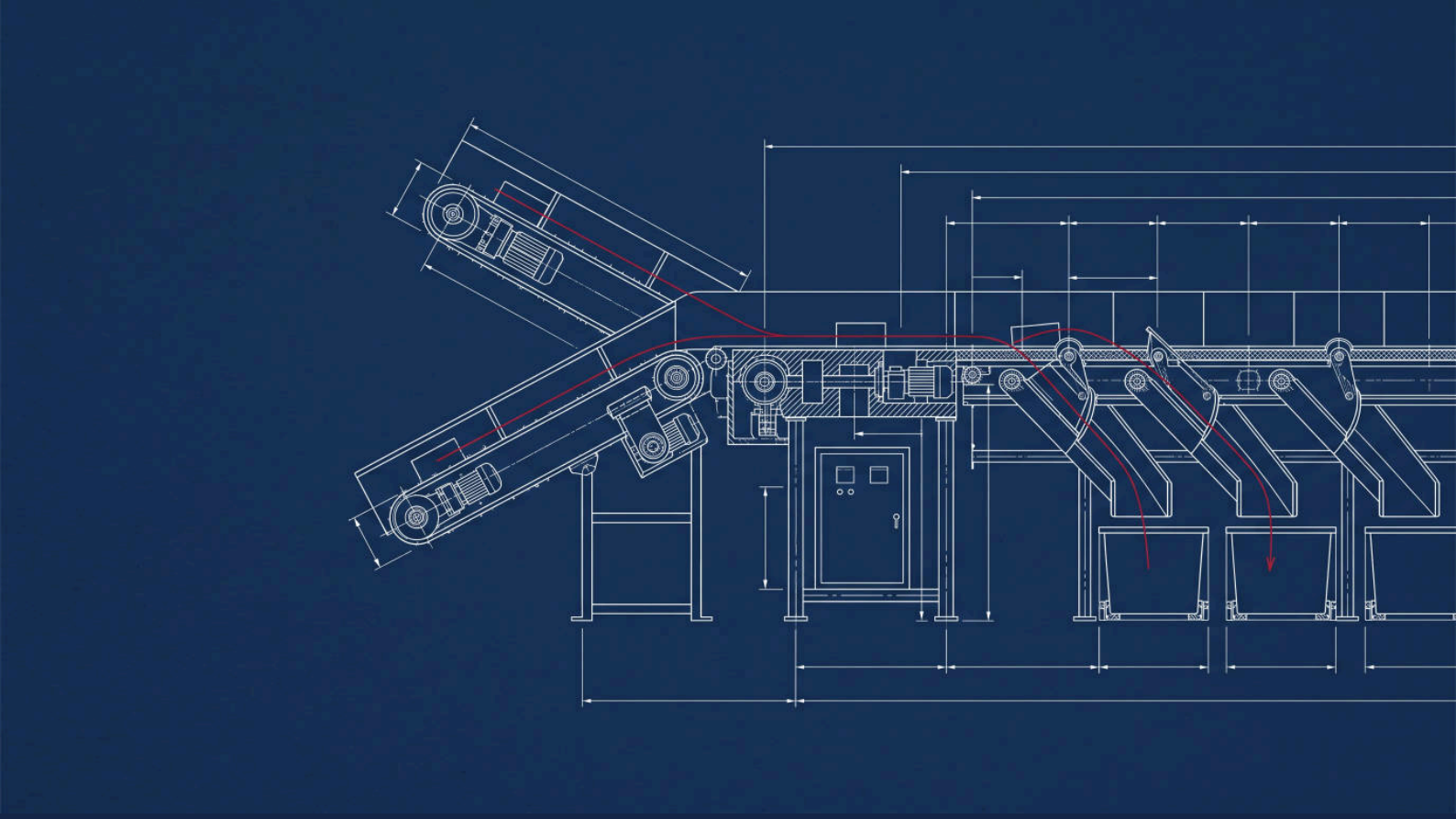

CHAPTER EIGHT

Scaling with Async Queues

SECTIONS 8.1 TO 8.11

Generation takes seconds to minutes. Web request handling is designed for milliseconds. Almost every scaling failure in an AI product is a consequence of ignoring that mismatch for one release too long.

CHAPTER EIGHT

Contents

PART II · RETRIEVAL

Generation takes seconds to minutes. Web request handling is designed for milliseconds. Almost every scaling failure in an AI product is a consequence of ignoring that mismatch for one release too long, and the arithmetic that forces the fix fits on a napkin.

8.1 Sync versus Async in RAG Architectures

A synchronous handler holds one worker for the entire duration of the work it is doing. In a conventional web application that is fine, because the work takes fifty milliseconds. Here it does not.

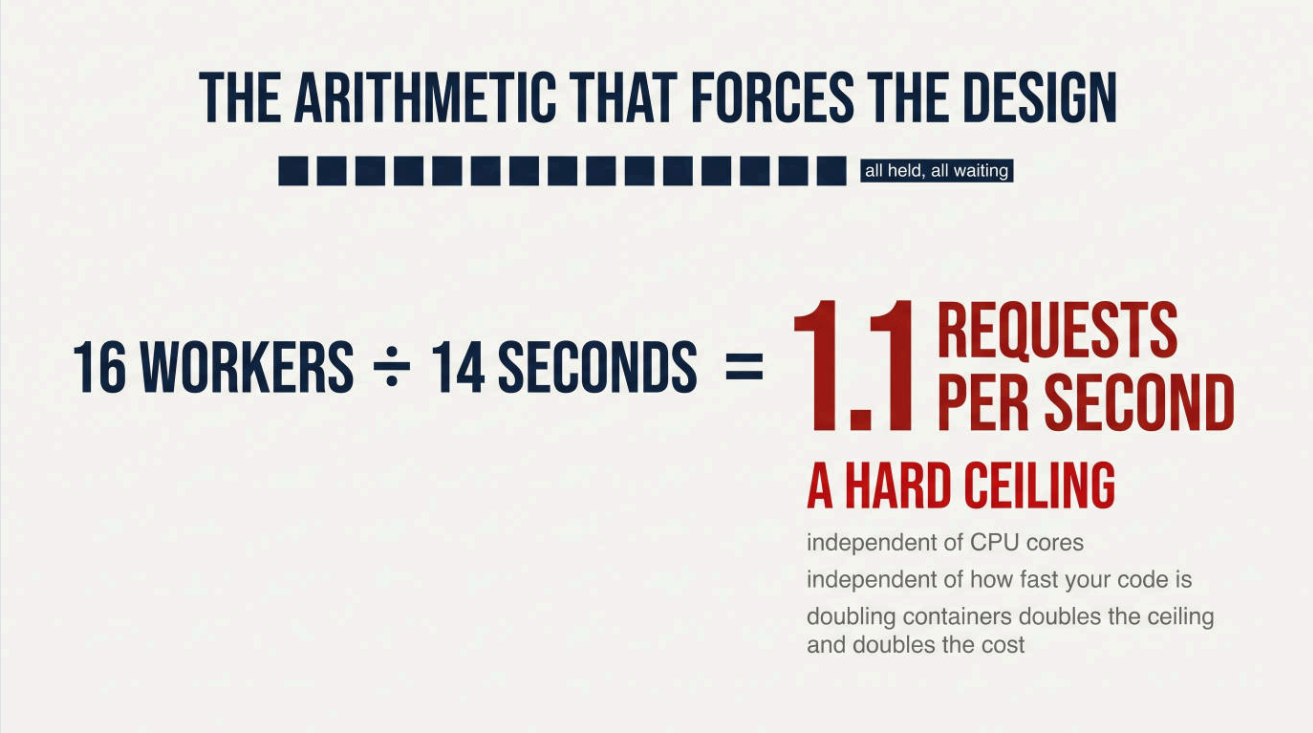

Figure 8.1 Capacity is workers divided by generation time, and generation time is the quantity you control least.

```
# src/docassist/queueing/capacity.py
@dataclass(frozen=True)
class SyncCeiling:
    workers: int
    generation_seconds: float

    @property
    def requests_per_second(self) -> float:
        return self.workers / self.generation_seconds

    def workers_needed_for(self, target_rps: float) -> int:
        return math.ceil(target_rps * self.generation_seconds)
```

```
SyncCeiling(16, 14).explain()
# 16 workers / 14s = 1.1 req/s hard ceiling (69 req/min), independent of CPU

SyncCeiling(32, 14).requests_per_second     # 2.29  -- still under three
SyncCeiling(16, 14).workers_needed_for(10)  #  140
SyncCeiling(16, 14).workers_needed_for(50)  #  700
```

Sixty nine requests per minute is not a load problem, it is a *shape* problem. Doubling the worker count doubles the ceiling and doubles the bill while leaving you under three requests per second. Reaching a modest fifty requests per second would take seven hundred workers.

This is the number that ends the "just add workers" conversation. Seven hundred workers to serve fifty requests per second is not a procurement decision, it is evidence that the architecture is wrong. The ceiling is independent of your CPU count and independent of how fast your own code is, because your code is not what is slow: it is waiting.

What actually breaks

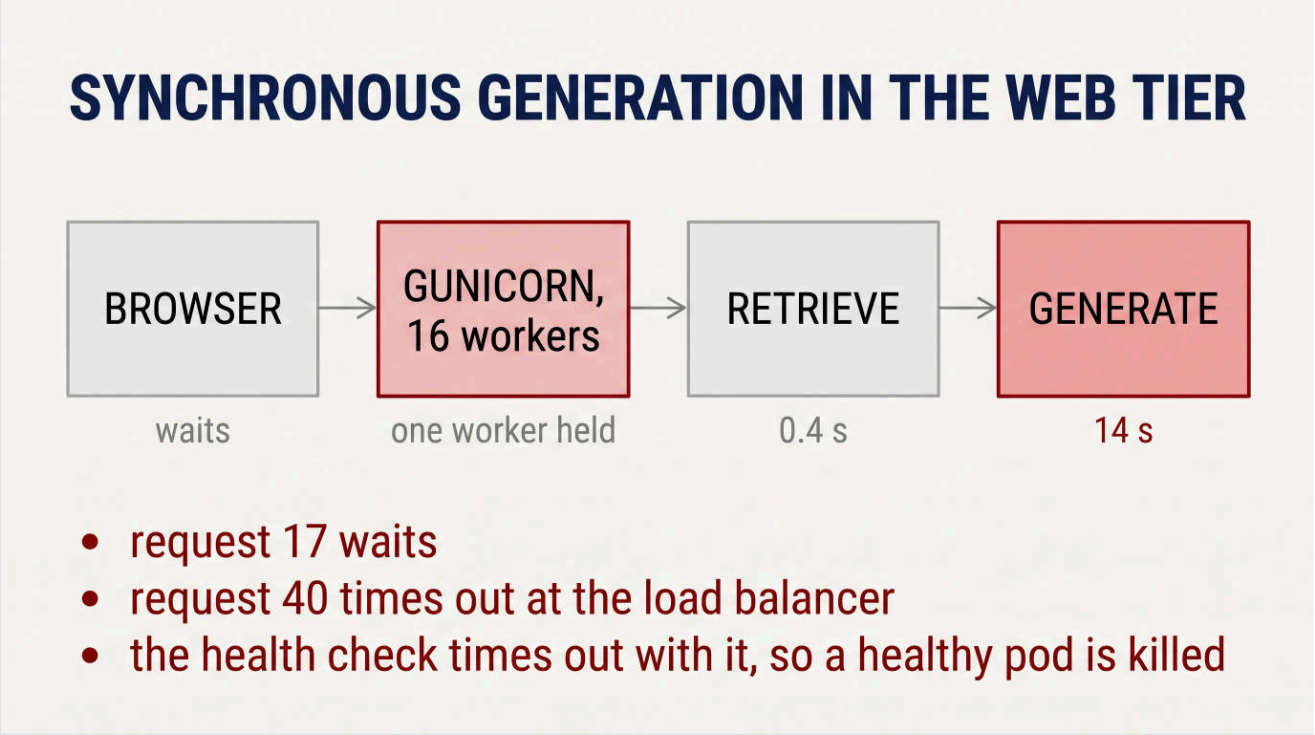

Figure 8.2 The cascade from section 3.6, arriving from your own traffic rather than a provider incident.

Section 3.6 traced this when a provider slowed down. The same cascade fires when nothing external has gone wrong at all and you simply have seventeen concurrent users. Workers saturate, requests queue at the load balancer, **the health check queues with them**, the orchestrator concludes the pod is unhealthy and kills it, and its traffic redistributes to pods that were already saturated.

The mitigations that look obvious and are not: raising the worker count multiplies memory and moves a ceiling that is still tiny; raising the health check timeout keeps unhealthy pods in rotation and makes the failure slower and more confusing. The real fixes are the separate `/alive` probe from section 4.4, and moving the work out of the request, which is the rest of this chapter.

8.2 Queue System Design for Async Setup

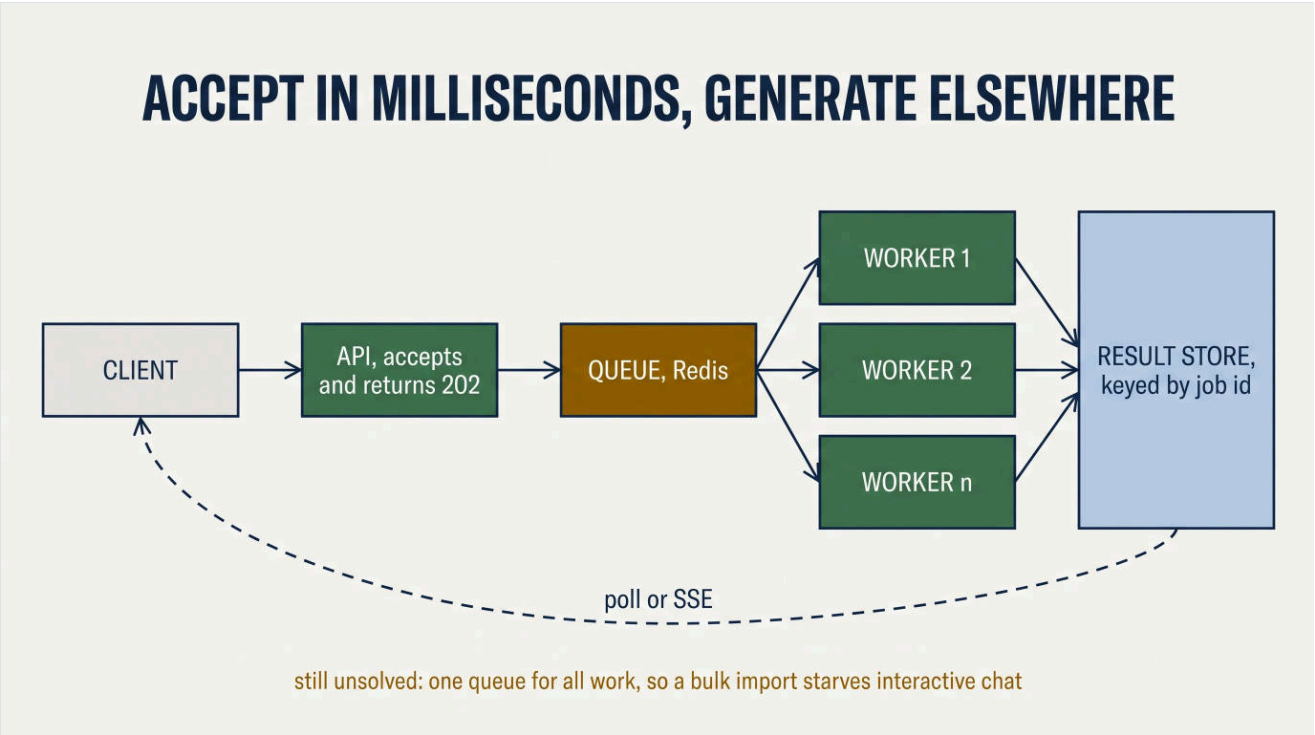

Figure 8.3 The API's job becomes accepting work, not doing it. Latency stops being a capacity constraint.

The reframing: **the API accepts work and returns an identifier; workers do the work; the client collects the result separately.** Request handling returns to milliseconds, and generation time stops determining how many users you can serve concurrently.

Four components, and each one is a decision.

Component	What it must provide
Broker	At-least-once delivery, visibility timeouts, and delayed messages for retries. Not "a list in Redis", which loses work on restart.
Result store	Keyed by job id with a TTL. Must be readable by the API tier, which is a different process from the worker that wrote it.
Worker pool	Separately deployable and separately scalable from the API. If they scale together you have kept the coupling and added a queue.
Delivery channel	How the client learns the answer: polling, server-sent events, or a callback. Section 8.8.

Be concrete about the result store, because the whole contract rests on it. In this build it is a Redis key per job id with a TTL, written by the section 8.10 handler and read by the API tier, on the append-only instance that section 8.3 configures; a Postgres row does the same job when results must outlive Redis. The 202 receipt of section 8.6 is a promise that the result will be collectable, and that promise rests on the result store's durability, not the broker's. The broker may redeliver or forget; the result store is where the answer lives.

> THE QUEUE DOES NOT REMOVE THE CEILING, IT MOVES IT
>
> Your workers still generate at fourteen seconds each. What changes is that the *waiting* is now visible, bounded and schedulable rather than consuming a web worker and taking the health check down with it. A queue converts an unbounded latency problem into a capacity problem, and capacity problems have arithmetic.

Choosing a broker: the distinction that decides it

Three options dominate, and teams usually compare them on throughput, which is the least relevant axis at this scale. The decisive distinction is older and simpler.

A queue delivers a message to one consumer and then forgets it. A log appends a message and remembers it, letting any number of consumers read at their own position. Task work wants the first. Event distribution wants the second. Choosing the wrong one is not a performance mistake, it is a semantic one, and it surfaces as a category of bug the tool cannot be configured out of.

Broker	Shape	Right when	Wrong when
Redis with ARQ	Queue	You already run Redis, jobs are hundreds per second rather than thousands, and one team owns the whole pipeline. The lowest-operational-cost option that is genuinely correct.	Queue depth can exceed memory, or you need routing beyond "which queue name".
RabbitMQ	Queue	You need real routing topology, per-message acknowledgement, dead-letter exchanges and priority as *broker* features rather than things you implement. The mature choice for task distribution.	You need to replay history, or fan the same message to many independent consumers who each track their own progress.
Kafka	Log	Many independent consumers read the same stream, you need replay and backfill, ordering within a key matters, and retention is measured in days.	You are distributing *variable-duration tasks*. See below, because this is the mismatch that catches people.

Why Kafka is usually the wrong shape for generation work

Kafka gives ordering within a partition, which means a partition is consumed *sequentially*. A single ninety second generation therefore blocks every message behind it in that partition, and section 8.1 established that generation duration is both long and highly variable. That is head-of-line blocking as a structural property rather than an incident.

The second mismatch is the redelivery model. Kafka has no per-message acknowledgement; a consumer commits an offset. If your handler takes longer than `max.poll.interval.ms`, commonly five minutes by default, the coordinator concludes the consumer is dead and rebalances the group, which redelivers the work and can cascade into repeated rebalances under load. It is the visibility-timeout problem of section 8.10 with a coarser instrument and a noisier failure.

THE ARRANGEMENT MOST MATURE SYSTEMS ARRIVE AT

Not one broker, but two, along the seam between tasks and events. **A queue for work**, Redis or RabbitMQ, carrying "generate this answer" and "ingest this document", where per-message acknowledgement, dead-lettering and independent job duration all matter. **A log for events**, Kafka, carrying the trace records from section 1.6, ingestion events and outcome signals, where several consumers read independently: the eval pipeline of section 7.10, the cost dashboards of section 16.2, and the warehouse.

Reach for that split when a second consumer wants the same stream, not before. One broker is a simpler system, and "we might want to replay this later" is not the same as wanting to.

The same concepts, three vocabularies

Every idea in sections 8.9 to 8.11 exists in all three, under different names. Knowing the mapping is what makes the rest of this chapter portable rather than ARQ-specific.

Concept	Redis + ARQ	RabbitMQ	Kafka
Concurrency limit *(8.5)*	`max_jobs`	`basic.qos` prefetch count	Partition count, plus `max.poll.records`
Lease before redelivery *(8.10)*	`job_timeout`	`consumer_timeout`, ack deadline	`max.poll.interval.ms`
Dead letters *(8.10)*	You implement it	Dead-letter exchange, built in	You implement it, usually a separate topic
Delayed retry *(8.10)*	`Retry(defer=...)`	TTL plus dead-letter exchange, or the delay plugin	You implement it, usually a retry topic per delay tier
Classes of service *(8.9)*	Separate queue names	Separate queues, or priority queues	Separate topics
Replay	None	None once acknowledged	Reset the consumer offset

Two entries in that table are worth reading twice. RabbitMQ gives you dead-lettering and delayed retry as broker features, which is a real reduction in code you own and the strongest practical argument for it over Redis. And Kafka's "you implement it" appears twice for exactly the reasons above: it is a log, and those are queue concerns.

8.3 Setting Up Redis and Valkey with Docker

The compose service exists from section 1.3. Three settings matter and are frequently wrong.

```
redis:
  image: valkey/valkey:8.0-alpine        # pinned, per section 1.2
  command: >
    valkey-server
    --appendonly yes                     # (1) durability
    --maxmemory 2gb
    --maxmemory-policy noeviction        # (2) NOT allkeys-lru
  healthcheck:
    test: ["CMD", "valkey-cli", "ping"]
```

(1) Append-only persistence. Without it, a restart loses every queued job. Users who submitted work receive nothing and no error, which is the worst available failure mode because nobody is notified.

(2) The eviction policy is a correctness setting, not a performance one. This is the one that bites. Under `allkeys-lru`, memory pressure causes Redis to silently discard keys, and some of those keys are queued jobs. Work vanishes with no error anywhere. `noeviction` makes the broker reject writes when full, which surfaces as a visible enqueue failure you can alert on and retry.

Use a separate Redis instance, or at minimum a separate logical database, for the queue and the cache. They have opposite requirements: a cache *should* evict under pressure, and a queue must never.

8.4 Python ARQ and Distributed Queues

ARQ is the right default for this stack, and the reason is structural rather than a matter of taste. Everything built so far is asyncio-native: the gateway holds one `httpx.AsyncClient`, retrieval awaits two searches concurrently, and the API is FastAPI. A thread-per-job queue would force a synchronous boundary at precisely the point where the work is I/O-bound waiting, which is the one thing asyncio handles well.

The incumbent you will be asked to compare against is Celery, and most Python teams already run it. The reason for ARQ here is fit rather than fault: ARQ's workers are async-native, which suits a workload that is almost entirely waiting on a model, while Celery's prefork model is shaped for CPU-bound tasks that a process can own. Everything this chapter builds on top of the queue applies to either.

```
# worker.py
from arq.connections import RedisSettings

async def doc_qa(ctx: dict, tenant_id: str, question: str,
                 doc_ids: list[str], prompt_version: int) -> dict:
    """A job is an async function taking a context dict.

    ctx carries what startup put there, plus arq's own bookkeeping:
    ctx["job_try"] is the attempt number, which is what section 8.10's
    retry classification reads.
    """
    gw = ctx["gateway"]                      # built once, in startup
    ...

class WorkerSettings:
    functions = [doc_qa, ingest_document]
    redis_settings = RedisSettings(host="redis")
    queue_name = "arq:interactive"           # per class, section 8.9
    max_jobs = 12                            # CONCURRENCY, see below
    job_timeout = 120                        # section 3.6's deadline, outward
    keep_result = 3600                       # result TTL
    max_tries = 3
    health_check_interval = 30
    on_startup = build_dependencies
    on_shutdown = close_dependencies
```

Four settings in that configuration matter more than the library choice, and you should demand their equivalents of whatever you use.

Setting	Why it is not optional
`job_timeout`	A hung generation must not hold a slot forever. This is the section 3.6 deadline one layer out, and it should exceed your gateway deadline rather than race it.
`keep_result`	Results are large. Without expiry your broker quietly becomes a database nobody meant to build.
`queue_name`	One worker deployment per queue is the mechanism for classes of service in section 8.9. Without separate names you have one pool wearing three labels.
`on_startup`	Build the gateway, the HTTP client and the prompt registry *once per worker*. Constructing them per job repeats the section 4.4 mistake in a new tier.

The deduplication that composes with section 8.10

```
await redis.enqueue_job(
    "doc_qa", tenant_id, question, doc_ids, prompt_version,
    _job_id=job.idempotency_key(),      # broker-level dedup
    _queue_name="arq:interactive",
)
```

Passing `_job_id` makes ARQ refuse to enqueue a second job with the same identifier while the first is queued or its result is still retained. Feeding it the idempotency key from section 8.10 collapses a double-clicked submit button before it ever reaches a worker.

This does not replace the handler-side check. Broker deduplication is bounded by result retention: once `keep_result` expires, the same key enqueues again. The handler's lookup against the result store is the durable guarantee; the broker's is an optimisation that saves a round trip. Keeping both, and understanding which is which, is the difference between a system that deduplicates and one that happens to.

Retries are raised, not returned

```
from arq.worker import Retry

async def doc_qa(ctx, ...):
    try:
        return await run_pipeline(...)
    except NonRetryable:
        raise            # arq will not retry; section 8.10 dead-letters it
    except TransientProviderError:
        # Defer THIS job with backoff, without occupying a slot meanwhile.
        raise Retry(defer=backoff_seconds(ctx["job_try"]))
```

`ctx["job_try"]` is what makes the retry classification of section 8.10 implementable: the handler can see which attempt it is on and decide whether a failure is worth another one. A queue that does not expose the attempt count forces you to track it in the payload, which then has to survive redelivery.

WHAT GOES IN THE MESSAGE

Enqueue **identifiers and parameters, never payloads**. A message carrying a whole document is slow to serialise, expensive to store, and impossible to inspect when it fails. Enqueue `{"doc_id": ..., "tenant_id": ..., "prompt_version": 4}` and let the worker fetch what it needs.

Include the **prompt version and index build id** in the message. A job queued before a deploy and executed after it should run against the versions it was queued with, or your canary from section 5.8 measures a population you did not define.

When you outgrow it

ARQ is the right default and it is not the right endpoint for every system. Three triggers, each of which points at RabbitMQ rather than at a bigger Redis:

- **Queue depth approaches memory.** Redis holds the queue in RAM, so a bulk import that enqueues two million jobs is a capacity event. RabbitMQ pages to disk.
- **You are reimplementing dead-lettering and delayed retry.** Section 8.10 builds both by hand, which is forty lines and correct. When you find yourself building routing on top of them, the broker already has it.
- **Routing beyond a queue name.** Fan a document event to ingestion, audit and notification without three enqueues from the producer, and you want an exchange.

Note what is absent from that list: raw throughput. Few document products are near the point where Redis is the bottleneck, and a team that migrates for throughput usually discovers the constraint was section 8.1's ceiling all along, which no broker changes.

PIN IT, LIKE EVERYTHING ELSE

ARQ is small and slow-moving, which is a virtue in a broker. It is still a dependency whose defaults determine behaviour, so section 1.2 applies unchanged: pin the version, and check `max_tries`, `keep_result` and `job_timeout` against your own numbers rather than inheriting whatever the current release ships.

8.5 Worker Orchestration

Workers are stateful in one specific way that matters: they hold a lease on a job, and killing one mid-flight discards work already paid for.

```
# The two settings that make deploys safe
terminationGracePeriodSeconds: 90        # longer than your longest job
lifecycle:
  preStop:
    exec:
      command: ["sh", "-c", "kill -TERM 1"]   # stop taking NEW work,
                                              # finish what is in hand
```

A worker receiving a shutdown signal should stop consuming from the queue and finish its current job. A grace period shorter than the longest permitted generation guarantees that every deploy discards in-flight work, which then gets redelivered and generated a second time. You pay twice and the user waits twice, on every release.

Concurrency lives in `max_jobs`, not in the process count. An ARQ worker runs many jobs concurrently inside one event loop, which is the right shape for I/O-bound generation and changes how you size the tier: one process per container, and `max_jobs` set from the capacity arithmetic rather than from intuition.

```
# The concurrency limit is the same number as section 4.5's admission control,
# for the same reason: (VRAM - weights - overhead) / KV-per-sequence.
max_jobs = 12          # NOT "however many the library defaults to"
```

Two failure modes sit either side of that number. Set it too high against a self-hosted model and you reproduce the out-of-memory restart loop from section 4.5 in the worker tier. Set it too high against a hosted provider and you become the thundering herd of section 3.6 with your own traffic. Set it too low and workers idle while the queue ages, which section 8.11 will show you and CPU will not.

8.6 FastAPI Endpoints for the Chat Queue

The endpoint's job changes from producing an answer to accepting a request. Three things must happen before it returns, and one must not.

```
@app.post("/ask", status_code=202)
async def ask(req: AskRequest, request: Request) -> AcceptedResponse:
    ctx = request.state.ctx

    # (1) Admission control BEFORE enqueue, per section 4.5. A queue you
    # cannot drain is a slower way to fail.
    if not admission.accepting(Priority.INTERACTIVE):
        raise HTTPException(429, headers={"Retry-After": "5"})

    # (2) The idempotency key comes from the WORK, so a double-submitted
    # form or a retried client request collapses into one job.
    job = Job(task="doc_qa", tenant_id=ctx.tenant_id,
              content_hash=content_hash(req.question, req.doc_ids))

    if (existing := await results.get(job.idempotency_key())) is not None:
        return AcceptedResponse(job_id=job.idempotency_key(), status="done")

    # (3) Enqueue with the versions in force AT ACCEPT TIME. queue.enqueue
    #     wraps arq's enqueue_job from 8.4: the class selects the queue name,
    #     and the idempotency key becomes _job_id.
    await queue.enqueue(Priority.INTERACTIVE, job,
                        prompt_version=registry.version("doc_qa"),
                        index_build_id=settings().index_build_id)

    return AcceptedResponse(job_id=job.idempotency_key(), status="queued")
```

The thing that must *not* happen is any model call. The moment an endpoint awaits a generation to decide what to enqueue, you have reintroduced the ceiling from section 8.1 in a less obvious place.

Note the status code. **202 Accepted, not 200 OK.** It is not pedantry: it tells every intermediary, client library and engineer reading the logs that the response is a receipt rather than a result.

8.7 Asynchronous Enqueueing and Streaming Back

Queueing and streaming appear to conflict. Section 3.5 established that streaming is the cheapest latency improvement available, and section 8.1 just moved generation into a worker the client is not connected to.

The resolution: **the worker publishes tokens to a channel keyed by job id, and the API subscribes.** The worker never holds a client connection, and the client never waits on a worker slot.

```
# Worker side: number every delta, accumulate durably, then publish.
seq = 0
async for delta in gateway.stream(messages=msgs, tenant_id=job.tenant_id):
    seq += 1
    await results.append(job_id, seq, delta)          # durable accumulation
    await bus.publish(f"job:{job_id}", (seq, delta))  # ephemeral fan-out
await bus.publish(f"job:{job_id}", DONE)

# API side: subscribe FIRST, then replay, and let the sequence numbers
# dedupe the overlap. Cheap: no worker slot, no model call.
@app.get("/ask/{job_id}/stream")
async def stream(job_id: str):
    async def events():
        sub = bus.subscribe(f"job:{job_id}")          # open the tap first
        last = 0
        for seq, delta in await results.deltas(job_id):
            last = seq
            yield f"data: {json.dumps({'delta': delta})}\n\n"
        async for msg in sub:
            if msg is DONE:
                break
            seq, delta = msg
            if seq > last:                            # replay already sent it
                last = seq
                yield f"data: {json.dumps({'delta': delta})}\n\n"
    return StreamingResponse(events(), media_type="text/event-stream")
```

Two consequences worth planning for. The channel is **ephemeral**, so a client that connects late misses the beginning, which is why the worker accumulates every delta into the result store as well as publishing it. The order on the API side is deliberate: replaying *before* subscribing leaves a race, because a delta published between the end of the replay and the start of the subscription is simply lost. Subscribe first, replay second, and the per-delta sequence numbers make the overlap between the two harmless. And a client that disconnects should not cancel the job, unlike the direct-streaming case in section 3.5, because the work is now shared: another tab, or the same user tomorrow, may collect the result.

8.8 Polling, SSE and Result Delivery

Mechanism	Right when	Costs
Polling	Jobs take minutes; a browser is not necessarily open; simplicity matters	Latency equal to the poll interval; wasted requests. Use exponential backoff, not a fixed one-second timer.
Server-sent events	Interactive chat where token-by-token delivery is the product	An open connection per active job. Cheap, but not free at the load balancer, which needs idle timeouts raised.
Webhook or callback	Machine-to-machine, bulk jobs, long pipelines	You now own delivery retries and the receiver's availability.

Whichever you choose, **the result store is the source of truth and the delivery channel is a convenience.** A design where the answer exists only in a stream is a design that loses answers to a dropped connection, and the user then re-asks, and you generate and pay again.

8.9 Classes of Service and Reserved Capacity

A single queue means a customer's ten-thousand-document import sits ahead of every interactive question asked while it drains. The import is not a bug and neither is the chat; the architecture simply has no way to express that they are different.

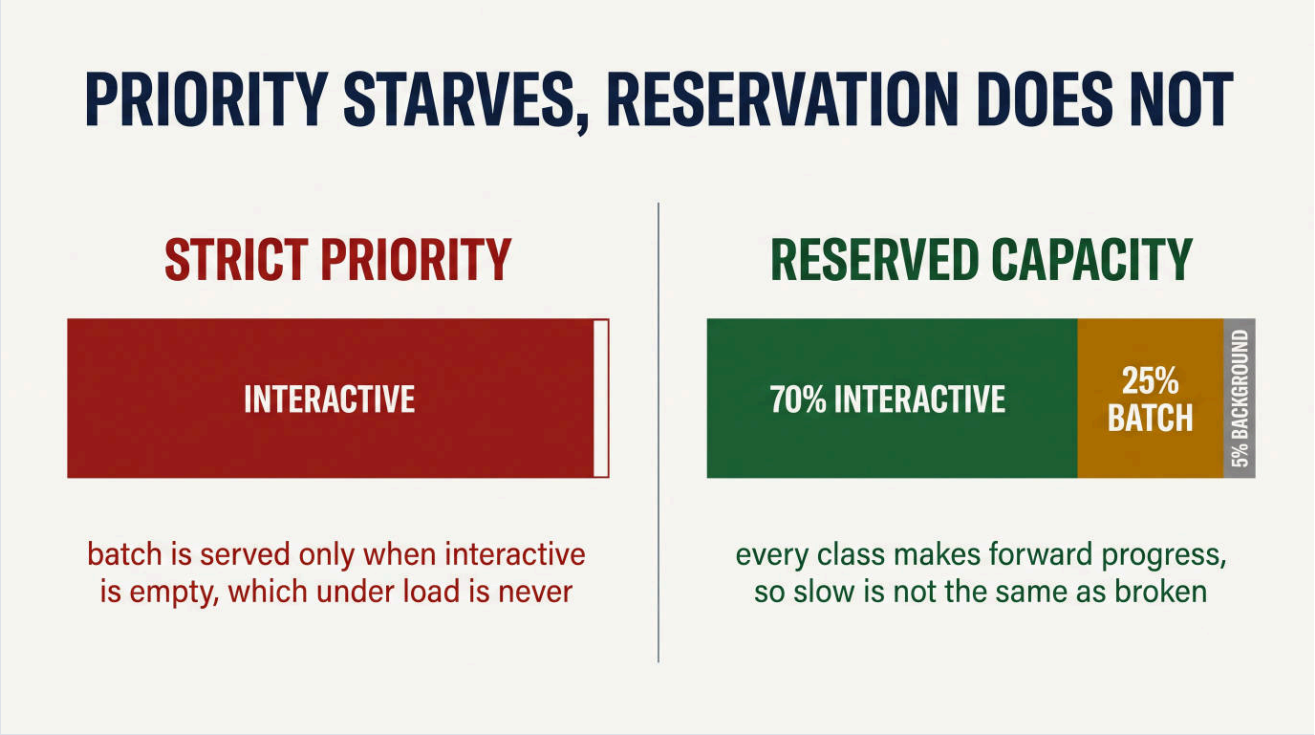

Figure 8.4 Strict priority is starvation with a scheduling diagram attached.

The instinct is priority: serve interactive first, batch when idle. **Under sustained load, "when idle" is never**, and the customer whose import has not moved in six hours cannot distinguish slow from broken. Reserved capacity guarantees forward progress for every class instead.

```
# src/docassist/queueing/classes.py
DEFAULT_ALLOCATION: dict[Priority, float] = {
    Priority.INTERACTIVE: 0.70,
    Priority.BATCH: 0.25,
    Priority.BACKGROUND: 0.05,
}

def plan_pools(total_workers: int, allocation=None) -> PoolPlan:
    """Allocate whole workers by fraction, guaranteeing at least one each.

    The floor of one is the entire point. A 5 percent share of 12 workers
    rounds to ZERO, and a class with zero workers is starved no matter what
    the configuration file claims.
    """
    if total_workers < len(allocation):
        raise ValueError(
            f"{total_workers} workers cannot guarantee forward progress for "
            f"{len(allocation)} classes")
    workers = {p: max(1, int(total_workers * f)) for p, f in allocation.items()}
```

```
plan_pools(12).workers    # {INTERACTIVE: 8, BATCH: 3, BACKGROUND: 1}
plan_pools(40).workers    # {INTERACTIVE: 28, BATCH: 10, BACKGROUND: 2}
plan_pools(3).workers     # {INTERACTIVE: 1, BATCH: 1, BACKGROUND: 1}
plan_pools(2)             # ValueError: cannot guarantee forward progress
```

Three design decisions in twenty lines, and each prevents a specific silent failure.

Fractions, not ranks. Ranks are priority wearing a different name, and priority starves.

A floor of one worker per class. Truncating `12 × 0.05` gives zero. A configuration file that reads "5 percent background" and a system that never runs background work is worse than one that admits it has no background tier.

Raising rather than silently starving. If you cannot honour the contract you declared, refuse to start. Section 4.1 made the same argument about the local runtime's context window: a service that will not boot is a far better outcome than one that quietly does two thirds of what you configured.

Reserved capacity stops one class starving another. It does nothing about one tenant starving the rest of its own class: a customer scripting a thousand questions occupies the interactive pool as thoroughly as any import. The standard fix is a per-tenant concurrency cap, or a weighted fair dequeue where caps are too blunt, and the admission controller of section 8.6 is where the cap lives. It already decides what enters the queue, and refusing one tenant's thirty-first concurrent job with a 429 is the same decision it makes for a saturated class, applied one level down.

> CLASSES ARE ALSO A COST CONTROL
>
> Because each class has its own pool, each can have its own model tier from section 3.6. Interactive work routes to the fast model; background summarisation routes to the cheap one; a bulk reindex can run entirely on a local model from Chapter 4 with no interactive latency requirement at all. The queue is where cost policy becomes operationally real rather than a configuration table.

For hosted models there is a further option the background class should usually take: every major provider sells an asynchronous batch tier, jobs submitted in bulk at roughly half the synchronous price, with a completion window measured in hours rather than a latency target. Where the model is hosted, route background work to that endpoint rather than to your own cheap-model queue. The provider's discount is comparable to what a smaller model saves, without giving up the model quality, and the provider does the queueing.

What it costs you is control and plumbing. The completion window is a latency ceiling you do not set, and the provider may use all of it. And results are collected from the provider rather than published by your worker, so the job that runs in your background pool becomes submit, poll and fan the outputs back into the result store, with the section 8.10 idempotency key still deciding what has already been processed.

It composes with classes of service cleanly, because the class now decides *which side of the wire the queue lives on*. Interactive and batch keep the queue on yours, where reserved workers and age thresholds apply. Background moves it to the provider's, and your own worker count for that class falls to whatever the submit-and-collect loop needs, which is close to zero.

8.10 Idempotency, Visibility Timeouts and Dead Letters

A distributed queue can promise at-least-once delivery. **It cannot promise exactly-once**, and any design that assumes otherwise will eventually process a job twice. Exactly-once *effect* is something your handler provides, and it costs one lookup.

Idempotency: derived from the work, not the attempt

```
def idempotency_key(self) -> str:
    """Derived from the WORK, not from the attempt.

    Keying on a job id would make a redelivery look like new work, which is
    exactly the case this exists to handle.
    """
    raw = f"{self.tenant_id}|{self.task}|{self.content_hash}"
    return hashlib.sha256(raw.encode()).hexdigest()[:32]
```

```
async def test_a_redelivered_job_replays_for_free():
    first  = await handle(job, ...)     # Outcome.COMPLETED
    second = await handle(job, ...)     # Outcome.REPLAYED, no work done
    assert second[1] == "answer"

async def test_the_idempotency_key_comes_from_the_work_not_the_attempt():
    a = Job("summarise", "acme", "hash-1", attempts=0)
    b = Job("summarise", "acme", "hash-1", attempts=2)
    assert a.idempotency_key() == b.idempotency_key()
```

The same key also collapses a double-clicked submit button and a client library's own retry into one job, which is a user-visible benefit rather than merely a correctness one.

Visibility timeouts and the heartbeat

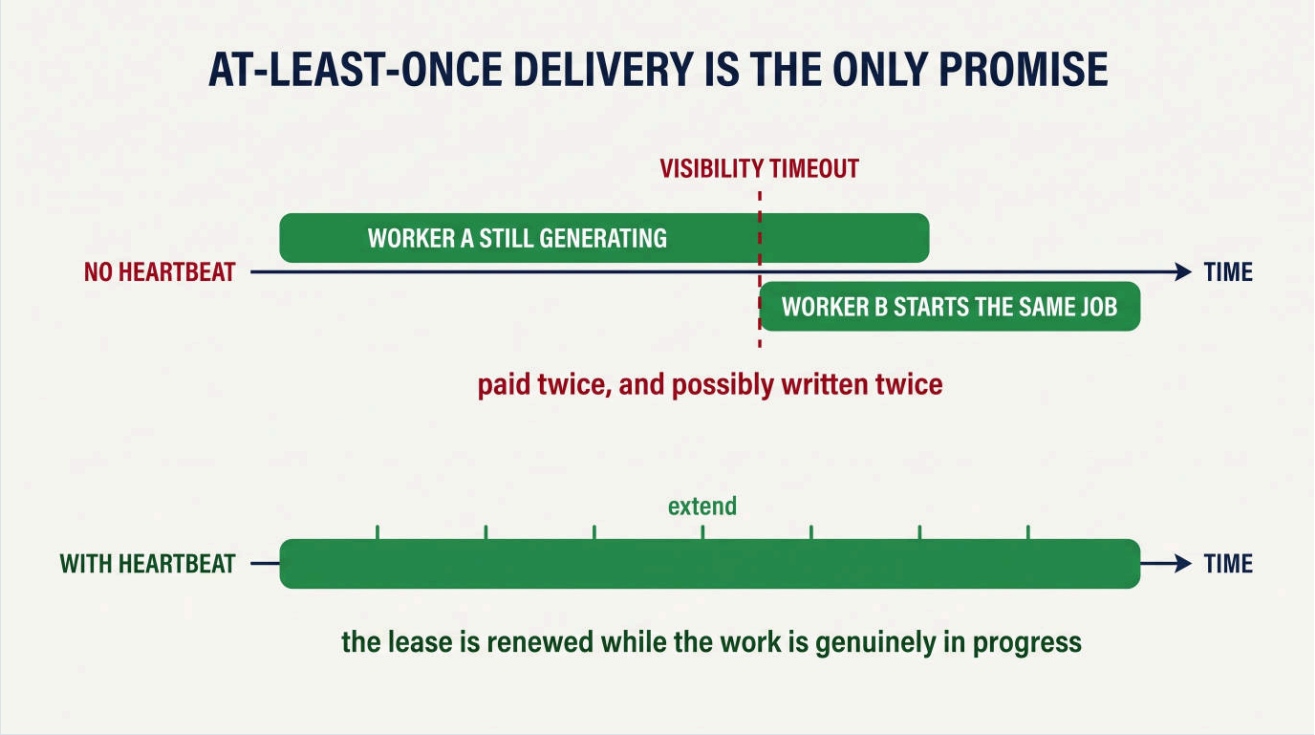

Figure 8.5 Without a heartbeat, generation outlives its lease and the job is delivered to a second worker.

A broker redelivers any message not acknowledged within its visibility window. Generation regularly exceeds any sensible default window. Without a heartbeat, the same job runs on two workers simultaneously: you pay twice, and if the handler has side effects you write twice.

```
class Lease:
    """Visibility timeout plus heartbeat. The lease is renewed while the work
    is genuinely in progress, which is different from extending the timeout
    to a number large enough to cover the worst case."""

    async def beat(self) -> None:
        await self._extend(self.window_s)
        self.extensions += 1
```

The distinction in that docstring matters. Setting the visibility window to ten minutes "to be safe" means a worker that genuinely dies holds its job hostage for ten minutes. A thirty second window with a heartbeat every ten seconds gives you both: fast recovery from real failures, and no spurious redelivery of work that is progressing.

Retry lanes and the dead letter queue

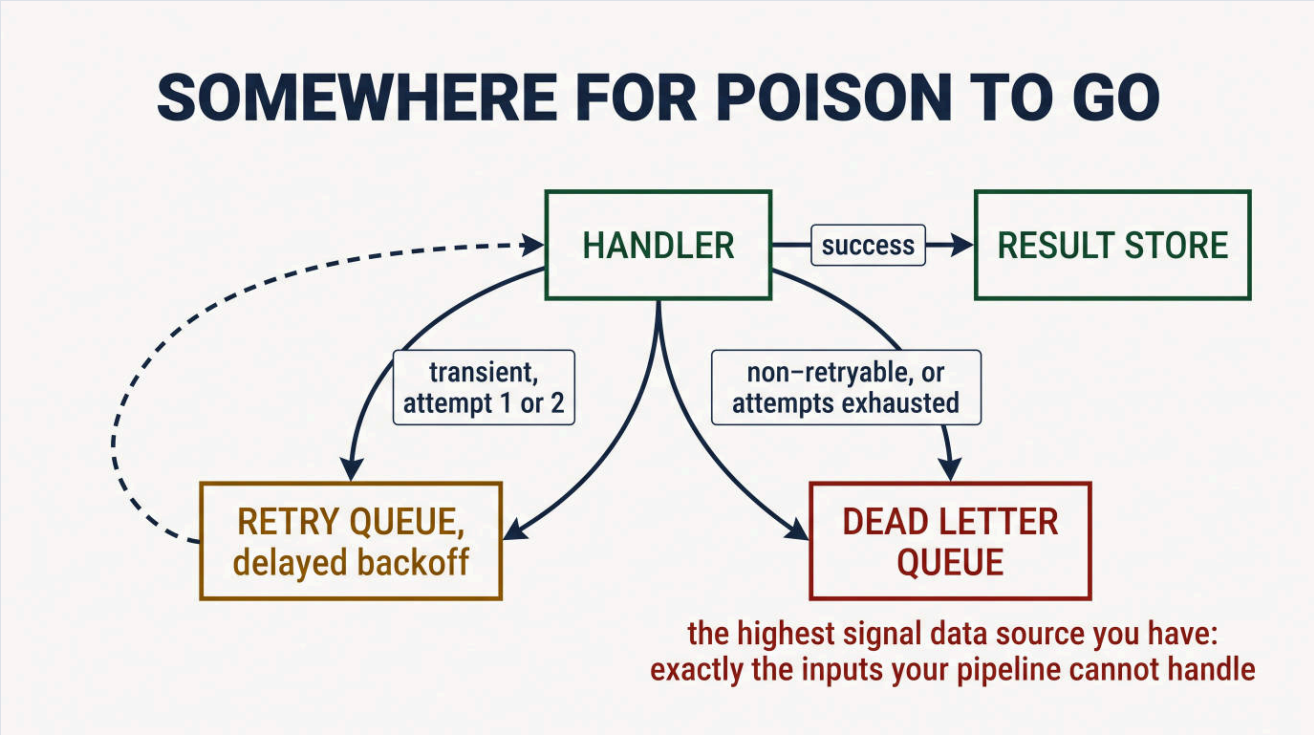

Figure 8.6 Two failure classes, two destinations. Conflating them means paying three times to learn the same thing.

```
except NonRetryable as exc:
    # First failure, straight to the dead letter queue. Retrying a
    # DETERMINISTIC failure burns money to learn nothing.
    await dead.put(job, f"non-retryable: {exc}")
    return Outcome.DEAD_LETTERED, None

except Exception as exc:
    if job.exhausted:
        await dead.put(job, f"exhausted after {job.attempts}: {exc!r}")
        return Outcome.DEAD_LETTERED, None
    await retry(job, backoff_seconds(job.attempts + 1))
    return Outcome.RETRIED, None
```

Classifying failures is the part teams skip, and it is a direct cost line. A document that exceeds the context window will exceed it on every attempt. Retrying it three times costs three full prefills to reach the identical conclusion, and at scale that is a meaningful fraction of an ingestion bill.

```
async def test_a_deterministic_failure_dead_letters_on_the_first_attempt():
    outcome, _ = await handle(job, do_work=raises_non_retryable, ...)
    assert outcome is Outcome.DEAD_LETTERED
    assert retried == []                      # never retried
```

The dead letter queue is the highest-signal data source in your system. It contains exactly the inputs your pipeline cannot handle, which makes every entry a candidate eval case for section 7.10. Alert on its *arrival rate*, not its depth, and review it weekly. A dead letter queue nobody reads is doing half its job: it has stopped the bleeding and thrown away the diagnosis.

8.11 Scaling Worker Nodes on the Right Signal

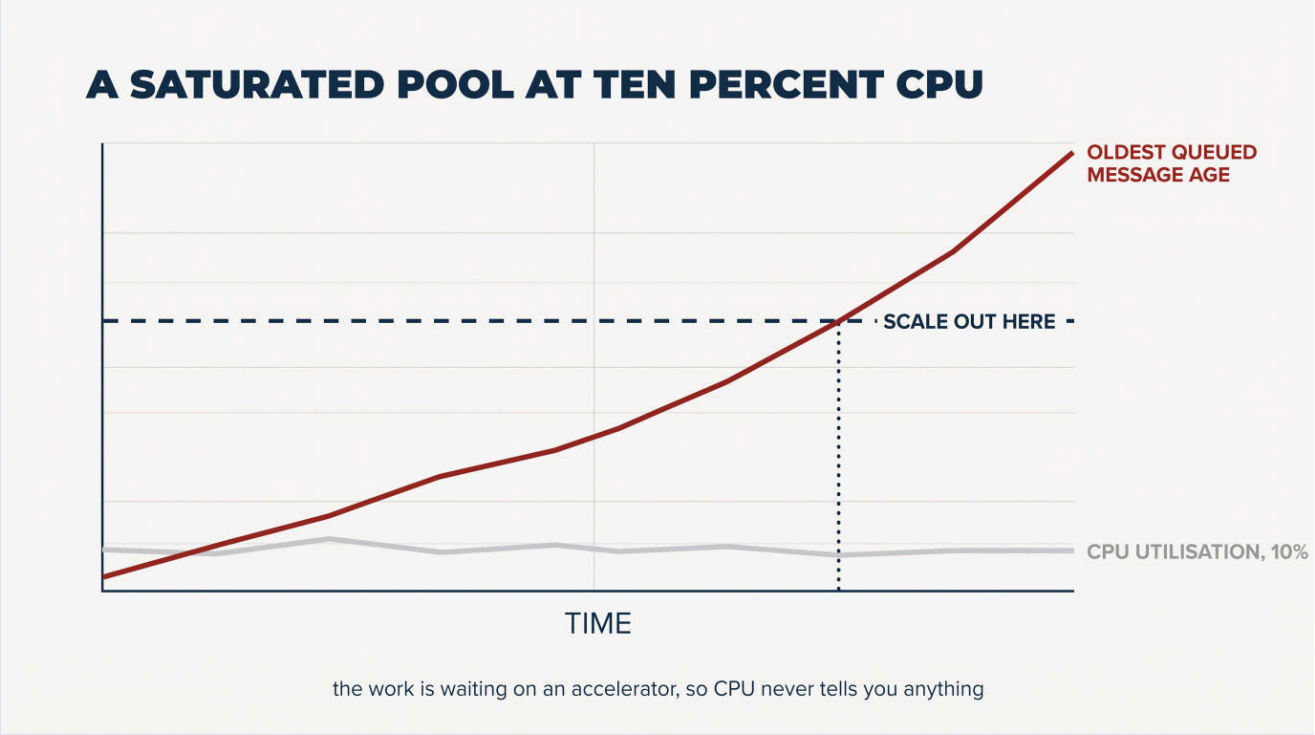

Figure 8.7 The work is waiting on an accelerator, so CPU never tells you anything.

A worker pool whose work is dominated by waiting on a model can be **fully saturated at ten percent CPU**. A conventional autoscaler watching CPU will do nothing while your queue grows without bound.

```
def test_a_saturated_pool_can_look_idle_to_a_cpu_autoscaler():
    s = ScalingSignal(oldest_age_s=42.0, depth=180, cpu_fraction=0.10)
    assert s.should_scale_out(max_age_s=30) is True
    assert s.cpu_would_say() is False          # CPU sees nothing wrong
```

Age, not depth

```
def breaches(self, thresholds: dict[Priority, float]) -> list[Priority]:
    """Section 8.11: AGE, not depth.

    A depth of 200 with fast jobs is healthy. A depth of 12 with stuck jobs
    is not. Age is what the user experiences.
    """
    return [p for p, limit in thresholds.items()
            if self.oldest_age_s.get(p, 0.0) > limit]
```

Depth is a proxy that breaks whenever job duration varies, which in this domain is always. The oldest queued item's age is the closest available measure of what a waiting user is experiencing, and it is the right thing to both alert and scale on.

Set a different threshold per class, because they mean different things: thirty seconds is an emergency for interactive and unremarkable for a bulk reindex.

Fast out, slow in

```
def scale_down_is_slower_than_scale_up(up_s: int, down_s: int) -> bool:
    """Removing a worker mid-generation discards work already paid for, so
    the scale-down window must exceed the longest generation you allow."""
    return down_s > up_s

scale_down_is_slower_than_scale_up(30, 300)     # True
```

Scale out on a thirty second window and scale in on five minutes. Aggressive scale-down is how a fleet ends up in a cycle of terminating workers mid-job, redelivering that work, and generating it a second time on the replacement, which costs more than the instance you saved.

What Chapter Eight Establishes

What exists now	What later chapters put through it
Accept-and-enqueue with 202 and a job id	Agent runs as durable jobs *(10.6)*
Result store as the source of truth	Checkpointing and resumable graphs *(10.5)*
Classes of service with reserved pools	Per-class model tiers and cost policy *(16.2)*
Idempotency keys derived from the work	Idempotent tool execution and side effects *(12.5)*
Leases with heartbeats	Long-running agent steps that outlive a request *(9.6)*
Dead letter queue	Failure triage and eval case mining *(7.10, 14.2)*
Queue-age scaling signal	Autoscaling policy and SLOs *(16.4)*

Chapter checklist

Can you...	Section
Compute your own synchronous ceiling, and the worker count for 50 req/s	8.1
Explain why raising the health check timeout makes the cascade worse	8.1
Say why a queue moves the ceiling rather than removing it	8.2
Explain why the eviction policy is a correctness setting	8.3
Say what belongs in a message and what does not	8.4
Give the grace period that makes deploys safe, and why	8.5
Name the one thing an accept endpoint must never do	8.6
Reconcile streaming with a worker the client is not connected to	8.7
Explain why strict priority starves and reservation does not	8.9
Say why the idempotency key derives from the work rather than the job id	8.10
Explain why a long visibility window is worse than a short one with a heartbeat	8.10
Justify scaling on queue age rather than depth or CPU	8.11

What Chapter 9 builds

Part II ends here. Chapters 6 to 8 built retrieval that is accurate, measurable and scalable, and every model call so far has been a single request with a single response.

Part III introduces the loop. An agent is a model whose output determines what happens next, which sounds like a small change and is not: it converts a bounded call into an unbounded process. Chapter 9 covers the ReAct loop and its three termination conditions, tool schemas as the contract that determines whether tools are called correctly, the step budget and the cost ceiling, and the permission model that section 5.9 argued for and section 1.5 made possible by keeping credentials away from anything the model can reach.

The framing that governs Part III: **an agent is a distributed system where one node is non-deterministic and bills you per message.** Everything in Chapters 1 to 8 was preparation for that sentence.