Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Claim Check + Semaphore-Based Queuing: A Fault-Tolerant Pattern for Distributed OCR/LLM Inference without Managed Infrastructure

Alejandro Jaime

alejandro.jaime@gmail.com

National University of La Plata

Research Article

Keywords: asyncio race condition, GPU inference, temporal decoupling, actor model, claim check pattern, fault-tolerant pipeline, distributed systems, fault tolerance

Posted Date: March 3rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8960646/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Claim Check + Semaphore-Based Queuing: A Fault-Tolerant Pattern for Distributed OCR/LLM Inference without Managed Infrastructure

Alejandro Jaime

Universidad Nacional de La Plata (UNLP), Facultad de Informática, La Plata, Argentina
alejandro.jaime@gmail.com

Abstract. We present the *Claim Check + Semaphore-Queued Inference* (CCSQI) pattern: a formally specified, production-validated architecture for fault-tolerant GPU inference that replaces $1,480–$2,100/month of managed cloud infrastructure with five open-source components on commodity hardware. The pattern targets *temporally decoupled* workloads — systems where producers and consumers operate in separate time windows and the SLA is completeness before a deadline, not real-time latency. For this workload class, reactive autoscaling solves a problem the architecture does not generate.

The principal contribution is the formal characterisation of an `asyncio` race condition specific to GPU-bound inference pipelines. While the interaction between cancellation and semaphore acquisition is a known difficulty in the Python community [1,2], its manifestation as silent slot leakage under sustained batch load has not, to the best of our knowledge, been formally characterised. Both `wait_for(semaphore.acquire())` and `asyncio.shield` produce deadlock under 30+ concurrent tasks with no exception and no log entry. The resolution follows from the actor model [11]: liveness belongs to the broker, exclusion belongs to the semaphore.

We additionally prove that a six-operation pipeline has exactly four structurally reachable failure modes, three automatically repairable and one preventable by operation ordering. The pattern is validated over 90 days, 4,000 documents, 24 classification types, with zero document loss and $0 recurring infrastructure cost.

Keywords: asyncio race condition · GPU inference · temporal decoupling · actor model · claim check pattern · fault-tolerant pipeline · distributed systems · fault tolerance

1 Introduction

1.1 The Knowledge Gap That Costs Billions

Modern commodity hardware is extraordinary. A single server with 32 cores, 128 GB RAM, dual high-end GPUs, and NVMe storage contains more raw parallelism than an entire data centre of a decade ago. The bottleneck in most AI

inference deployments is not hardware. It is the knowledge required to use it correctly.

The AI inference industry has responded to this knowledge gap not by closing it but by selling around it. The dominant market answer to "how do I run inference reliably at scale" is Kubernetes with HPA autoscaling, managed GPU queues, multi-region cloud deployments, and vendor-provided MLOps platforms. This stack costs between $1,500 and $2,100 per month for workloads that a single well-configured server can handle — and it requires specialised operational knowledge that most organisations cannot maintain internally.

This is not a technical solution. It is the industrialisation of architectural ignorance.

The knowledge that would eliminate most of this complexity is not new. The actor model, which enables correct concurrent computation without shared mutable state, was formalised by Hewitt, Bishop, and Steiger in 1973 [11]. Work-stealing schedulers, which optimally utilise multi-core hardware without kernel intervention, were proven optimal by Blumofe and Leiserson in 1999 [5]. Lock-free concurrent data structures with formal linearizability guarantees were codified by Herlihy and Shavit [10]. At-least-once messaging with manual offset commit — the backbone of fault tolerance in this paper — was established by the log-based messaging literature formalised in [13].

These results are not obscure. They form the theoretical foundation of Apache Spark [23], Apache Kafka [14], and Ray [16] — systems the industry adopted precisely because someone translated formal concurrency knowledge into usable implementations with honest benchmarks. The pattern is consistent: the systems that achieved lasting industrial impact are those that packaged existing formal knowledge correctly, not those that invented new primitives.

What the industry systematically fails to do is transmit this knowledge to the practitioners who build the majority of production systems. The result is a market where organisations pay for infrastructure complexity to compensate for the absence of concurrency knowledge — a dynamic that Hellerstein et al. [9] identify specifically in the serverless context as trading programmability for operational convenience at structural cost. We observe the same pattern in GPU inference deployment.

1.2 The Concrete Problem and Why Naive Solutions Fail

Consider the scenario representative of thousands of real enterprise deployments: a mid-size organisation must process PDF documents arriving from two asynchronous sources — an RPA bot polling a corporate mailbox overnight, and internal operators submitting documents via a web interface during business hours. Each document requires GPU-accelerated OCR followed by classification into one of 24 document types, using compiled regex patterns for known types and LLM-based inference for previously unseen document structures.

The requirements are non-negotiable: no document may be lost under any failure condition; the system must self-recover from crashes without human intervention; classification results must be available before the start of the business

day regardless of overnight arrival volume; the entire infrastructure must run on existing on-premises hardware with near-zero recurring cost.

This is a *temporally decoupled* workload. Producers (the RPA bot, the web interface) and consumers (the GPU inference pipeline) operate in different time windows. The relevant guarantee is completeness before a future deadline — not real-time latency. This distinction is architecturally fundamental: it means that reactive autoscaling (Kubernetes HPA, KEDA) solves a problem this system does not have, while creating operational complexity it cannot afford.

Naive solutions fail in predictable ways that expose the underlying knowledge gap. Synchronous REST APIs block the RPA for 3–5 seconds per document — the model load time alone — and collapse entirely under concurrent submissions (a fundamental limitation of synchronous RPC architectures [18]). In-memory queues lose all pending work on any process crash. Unguarded thread pools cause GPU out-of-memory errors when two inference calls overlap: PaddleOCR PP-OCRv4 requires ≈2.1 GB VRAM per instance, and without explicit concurrency control, threads compete for the same resource and crash the serving process. Distributed transactions (2PC, Saga) impose coordination overhead incompatible with the simplicity constraint and require operational expertise the organisation does not have.

No managed queue. No Kubernetes. No GPU cluster. No cloud budget. No SRE team.

There is no established architectural pattern in the published literature that directly addresses this class of deployment. This paper provides one — derived from first principles, validated in production, and formally proved.

1.3 Contributions

1. ***[C1] — Formal characterisation of an asyncio race condition in GPU-bound inference pipelines.*** The interaction between `asyncio` cancellation and semaphore acquisition is a known source of difficulty in the Python community [1,2]. What has not, to the best of our knowledge, been formally characterised is its specific manifestation in GPU-bound pipelines under sustained load: the standard idiom `wait_for(semaphore.acquire())` and the widely-recommended `asyncio.shield` workaround both produce silent GPU slot leakage under 30 or more concurrent tasks — no exception, no log, no observable error. The pipeline deadlocks. We provide, to our knowledge, the first formal proof of why both v1 and v2 fail in this deployment context, and derive the resolution from the actor model: separate liveness (Kafka) from exclusion (semaphore). We provide a complete formal proof of mutual exclusion, slot integrity, and liveness for the correct implementation.
2. ***[C2] — Exact failure mode completeness for a six-operation pipeline.*** We prove that a pipeline of six atomic operations has *exactly four* structurally reachable inconsistency states — not three, not five, exactly four. Three are automatically repairable. One is permanently prevented by a single deterministic ordering rule: update the record store before deleting transient storage.

The wrong order creates an irrecoverable state no checker can repair. The correct order makes every failure recoverable by construction. This result eliminates distributed transactions for this workload class entirely, by proof — not by assumption.

3. ***[C3] — Production evidence against architectural overkill.*** 90 days. 4,000 documents. 24 classification types. Zero document loss. $0 recurring infrastructure cost versus $1,480–$2,100/month for equivalent managed cloud. The system runs on existing commodity hardware at under 52% mean GPU utilisation. We demonstrate, with production data and formal proofs, that for temporally decoupled workloads, reactive autoscaling solves a problem that a correct architecture does not generate.
4. ***[C4] — CCSQI pattern specification.*** We formally specify the pattern as a six-component tuple $\Pi = (G, O, B, W, C, R)$ with a document state machine, a two-tier classifier (compiled regex for known types, LLM inference for unknown document structures with automatic promotion to the pattern catalogue), and a proven at-least-once processing guarantee. Implementable with five open-source tools on any commodity server.
5. ***[C5] — An architectural thesis.*** The limiting factor in most enterprise AI deployments is not hardware, not budget, not tools. It is the knowledge of concurrency and distributed systems formalised over five decades that does not reliably reach the practitioners who build production systems. CCSQI is a case study in what changes when it does.

2 Background and Related Work

2.1 The Formal Foundations the Industry Ignores

The concurrency knowledge required to build CCSQI is not new. Hewitt, Bishop, and Steiger [11] formalised the actor model in 1973: concurrent computation through message passing, with no shared mutable state, and each actor managing its own resources exclusively. Every major language runtime that achieves reliable concurrency at scale — Erlang/OTP, Akka, Tokio in Rust, Ray in Python — implements this model. The asyncio semaphore in CCSQI is an actor-model primitive: it owns the GPU slot count and responds to acquire/release messages without exposing shared state.

Blumofe and Leiserson [5] proved in 1999 that work-stealing schedulers achieve within a constant factor of the optimal parallel execution time with high probability. Ray's distributed scheduler implements this result. The consequence for CCSQI is that 32-core PrepWorker parallelism is not a heuristic — it is a formally bounded approach to CPU utilisation.

Herlihy and Shavit [10] codified the theory of concurrent data structures: linearizability, wait-freedom, lock-freedom. These guarantees are what separate correct concurrent systems from systems that appear to work under low load and fail silently under high load — exactly the failure mode of v1 and v2 in Sect. 5.

This body of knowledge is not obscure. It is taught in graduate systems courses. It forms the foundation of every major distributed systems tool the industry uses. The gap is not that the knowledge does not exist; it is that it is not transmitted to the practitioners who build production systems. The result is that organisations pay for infrastructure to compensate for knowledge they do not have.

2.2 Systems That Closed the Gap

The systems with the largest industrial impact share a pattern: they translated formal concurrency knowledge into usable implementations with honest benchmarks.

Zaharia et al. [23] applied lineage-based fault tolerance from database theory to in-memory parallel computation, producing Apache Spark — a system that replaced Hadoop MapReduce (which required deep operational expertise and was slow for iterative workloads) with a simpler abstraction that was 10–100× faster. The insight was not new algorithms but correct application of known theory. Kreps, Narkhede, and Rao [14] applied the append-only log as coordination primitive — a concept from Lamport's distributed systems work — to produce Apache Kafka, replacing complex message brokers with a system that was operationally simpler and orders of magnitude more durable. Moritz et al. [16] combined the actor model with work-stealing and lineage-based fault tolerance to produce Ray, enabling distributed ML workloads that no prior framework could express cleanly.

CCSQI follows this tradition: it applies the actor model, at-least-once messaging, and semaphore-based resource protection — all formally established — to a class of deployment that published frameworks do not address.

2.3 AI Inference Serving Frameworks

Clipper [7] introduced adaptive batching for latency-throughput tradeoffs. Orca [22] proposed iteration-level scheduling for transformer models. Triton Inference Server [20] provides GPU-efficient batching with multiple backend support. vLLM [15] introduced PagedAttention for memory-efficient LLM serving, achieving throughput improvements of up to 24× over naive baselines. General-purpose model serving frameworks such as BentoML [4] and Seldon Core [19] abstract deployment concerns but assume horizontally scalable, stateless consumers — an assumption that does not hold for semaphore-bounded GPU workers. All of these systems are architecturally excellent and represent the state of the art for their target use case. All of them assume managed infrastructure that the majority of real-world deployments cannot operate. *The gap is not a deficiency of these systems — it is an unaddressed class of deployment they were not designed for.*

2.4 The Claim Check Pattern

The Claim Check (CC) pattern [12] separates payload storage from message routing: large artifacts are stored durably; only lightweight reference tokens tra-

verse the broker. Classical CC consumers are stateless and horizontally scalable; GPU-bound consumers are neither. This asymmetry introduces the concurrency control requirement formalised in Sect. 5.

2.5 Architectural Overkill and its Costs

Hellerstein et al. [9] argue that serverless computing, despite its operational convenience, trades away programmability in ways that impose structural performance costs — a specific instance of the general pattern we describe. The market adoption of Kubernetes for workloads that do not require container orchestration is the same phenomenon: operational abstraction sold as robustness, at a price paid in complexity, cost, and reduced system understanding.

For temporally decoupled batch workloads — where the producer and consumer operate in different time windows and the relevant SLA is completeness before a deadline rather than real-time latency — HPA autoscaling solves a problem the architecture does not generate. The queue absorbs arrival variability; the semaphore protects the GPU; the broker guarantees re-delivery. No pod needs to scale. Pagels et al. [17] confirm that asynchronous processing and fault tolerance are systematically under-addressed in the ML deployment literature, a gap CCSQI directly fills.

2.6 Eventual Consistency and Fault Tolerance

The CAP theorem [6] establishes the fundamental tradeoff between consistency and availability. Garcia-Molina and Salem [8] formalised Saga as a long-running transaction mechanism — correct but operationally demanding. Vogels [21] argues that eventual consistency with active reconciliation suffices for most practical workloads. We adopt this model and prove in Sect. 4 that the CCSQI pipeline has exactly four reachable inconsistency states, each bounded by detection latency τ_C and repairable without distributed coordination.

3 The CCSQI Pattern

3.1 Overview and Core Design Principle

CCSQI decouples document ingestion from inference via a message broker, stores large binary payloads in an object store (Claim Check), and gates GPU access with an asyncio semaphore acting as a bounded-concurrency queue. A periodic consistency checker reconciles the object store, broker, and record store to repair inconsistencies from partial failures.

Separation of concerns. Each concern is assigned to the component best suited to handle it: *liveness* → message broker (offset re-delivery), *GPU exclusion* → semaphore (asyncio), *consistency* → active checker. Adding

a timeout to the semaphore — the most common implementation error — violates this principle and introduces the race condition described in Sect. 5.

3.2 Formal Specification

Let $\mathcal{D}$ be the set of documents entering the system. Each document $d \in \mathcal{D}$ carries a unique identifier $\mathsf{id}(d) \in \mathcal{U}$ (UUID4) and a binary payload $\mathsf{bin}(d)$ of size $s_d \in (0, s_{\max}]$ bytes.

Definition 1 (CCSQI Pipeline). *A CCSQI pipeline is a tuple* $\Pi = (G, O, B, W, C, R)$ *where:*

- G *(*Ingestion Gateway*): accepts* d*, persists* $\mathsf{bin}(d)$ *in* O *under key* $\kappa(d)$*, publishes reference token* $m_d = \langle \mathsf{id}(d), \kappa(d), \mathsf{meta}(d) \rangle$ *to* B*. Responds to producers in* $<$500 *ms, independent of* s_d*.*
- O *(*Object Store*): S3-compatible key-value store providing durable persistence of* $\mathsf{bin}(d)$*.*
- B *(*Message Broker*): ordered, persistent, at-least-once log of reference tokens; offsets committed* only *upon confirmed successful processing.*
- W *(*Inference Worker*): semaphore-gated consumer of* B*; retrieves* $\mathsf{bin}(d)$ *from* O*, applies* $f : \mathsf{bin}(d) \to \mathsf{result}(d)$*, persists to* R*, commits offset.*
- C *(*Consistency Checker*): periodic reconciler with interval* τ_C*; detects and repairs the four failure modes of Sect. 4.*
- R *(*Record Store*): persistent store for document state* $\sigma(d)$*, inference result, and audit trail.*

The token m_d is the "claim check": $|m_d| \approx 200$ bytes regardless of $s_d \leq 50$ MB, keeping broker throughput high and decoupling storage scaling from messaging scaling.

3.3 Document State Machine

Each document d has a lifecycle state $\sigma(d) \in \Sigma$, maintained in R as the single authoritative source of truth. Fig. 1 shows the state machine $\mathcal{S} = (\Sigma, T, \sigma_0, \Sigma_F)$.

3.4 At-Least-Once Delivery Guarantee

Theorem 1 (At-Least-Once Processing). *Under CCSQI with manual offset commit, every document* d *accepted by* G *eventually reaches state* UPLOADED *in* R*, provided (i)* O *and* B *are eventually available, and (ii)* C *runs with finite interval* τ_C*.*

Proof. We consider all crash points:
Crash in G **after** O.put **before** B.publish: d exists in O, absent from B and R. Checker C detects FM-1 and re-publishes m_d within τ_C; processing resumes normally.

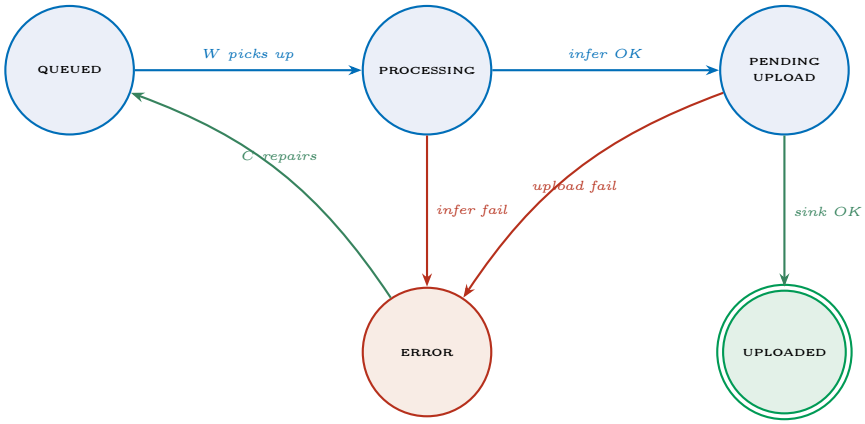


Fig. 1: Document state machine $\mathcal{S}$. Double circle: terminal state UPLOADED. Red: failure transitions. Green: recovery via checker C.

Crash in W before offset commit: B retains the uncommitted message and re-delivers m_d upon W restart. A pre-inference head-check on O ensures idempotent re-delivery. If a partial record exists in R, $\mathsf{id}(d)$ uniqueness prevents duplication.
Crash in upload step: $\sigma(d) =$ PENDING_UPLOAD persists in R; checker C detects FM-3 or FM-4 and triggers re-upload.

In every scenario C ensures convergence to UPLOADED within a finite number of τ_C intervals. □

Corollary 1. *No document is permanently lost under CCSQI as long as G completes O.*put*. This call is the sole point of irreversible commitment from the producer's perspective.*

4 Failure Mode Analysis

4.1 Structural Failure Modes

A crash between operations op_i and op_{i+1} leaves persisted effects of $op_1 \dots op_i$ and no effects of $op_{i+1} \dots op_n$. The CCSQI pipeline has six atomic operations: O.put, B.publish, W.infer, R.insert, Sink.upload, and the safe-ordered pair $\{R.\mathsf{update} \rightarrow O.\mathsf{delete}\}$ (Sect. 4.2).

Proposition 1 (Failure Mode Completeness). *Exactly four distinct observable failure modes are structurally reachable in the CCSQI pipeline.*

Proof. Enumerating crash intervals by observable state discrepancy:

Table 1: Complete failure mode taxonomy. FM-2 is the only mode requiring prevention rather than repair (Sect. 4.2).

FM	Observable condition	Root cause	Repair
1	O: payload; B,R: absent	Crash between O.put and B.publish	Re-publish m_d; processing resumes
2	R: PENDING_UPLOAD; O: absent	Safe-order violation	**Irrecoverable.** Requires manual re-ingest. Prevented by Sect. 4.2.
3	R: UPLOADED; URL = `null`	Silent sink failure	Revert to PENDING_UPLOAD; re-upload
4	UPLOAD_ERROR; retries $\geq r_{\max}$	Rate limit / permissions	Reset retries; re-enqueue; alert if recurs

Crash between	O state	R state	FM
O.put $\rightarrow$ B.publish	payload	no record	FM-1
R.update $\rightarrow$ O.delete (unsafe order)	absent	PEND._UPLOAD	FM-2
Sink.upload $\rightarrow$ R.update	payload	URL=`null`	FM-3
Sink.upload repeated failure	payload	UPLOAD_ERROR	FM-4

Crashes during B.publish $\rightarrow$ W.infer, W.infer $\rightarrow$ R.insert, and R.insert $\rightarrow$ Sink.upload all leave uncommitted Kafka offsets, triggering automatic re-delivery upon W restart — not new failure modes but covered by Theorem 1. The four enumerated modes are thus necessary and sufficient. □

Table 1 summarises detection conditions and repair actions for all four failure modes.

4.2 Safe Operation Ordering

FM-2 is the only irrecoverable failure mode, arising exclusively from incorrect operation ordering in the Uploader. The correct and incorrect orderings are:

Order	Steps	Crash consequence
Correct	(1) R.update $\rightarrow$ (2) O.delete	Payload lingers in O (FM-1, repairable)
Incorrect	(1) O.delete $\rightarrow$ (2) R.update	R stuck in PENDING_UPLOAD without payload (FM-2)

Design rule. Always update the authoritative state store *before* deleting transient storage. A failed delete leaves a recoverable surplus (FM-1). A failed update after deletion leaves an irrecoverable deficit (FM-2). Design for surplus.

5 The Asyncio Semaphore Race Condition

5.1 Problem Statement

GPU memory is bounded and non-preemptive. In our deployment each concurrent inference consumes ≈ 2.1 GB VRAM per PaddleOCR PP-OCRv4 instance. With two RTX 4090 GPUs providing 48 GB of total VRAM, the hardware can theoretically sustain up to $\lfloor 48/2.1 \rfloor = 22$ simultaneous inferences. However, reliable GPU utilisation requires accounting for model state, activation buffers, and inter-GPU coordination overhead. Setting $k = 4$ (two slots per GPU) was established empirically as the optimal balance between throughput and stability: values $k > 4$ produced sporadic OOM errors under documents of above-average complexity (> 80 pages, high-resolution scans).

The natural asyncio pattern for bounded-wait acquisition is:

```
# v1 -- INCORRECT: race condition under sustained load
async def process_message(msg, semaphore):
    try:
        await asyncio.wait_for(
            semaphore.acquire(), timeout=600)  # "safety net"
    except asyncio.TimeoutError:
        logger.error("Timeout -- dropping task")
        return
    try:
        await _infer_and_persist(msg)
    finally:
        semaphore.release()
```

Listing 1.1: v1: Naive timeout-guarded acquisition – race condition.

The general difficulty of combining `asyncio` cancellation with semaphore acquisition has been observed in the Python community: van Rossum [1] documented cancellation bugs in the semaphore's internal Future queue, and CPython issue #90155 [2] fixed ordering bugs in the waiters deque. However, neither source analyses the specific failure mode that arises in *GPU-bound pipelines under sustained concurrency*: silent slot leakage accumulating across batches of 30 or more tasks until the semaphore counter drains to zero and the pipeline deadlocks with no exception and no log entry. The following sections provide that characterisation and a formal proof of the correct solution.

5.2 The Race Condition Mechanism

When `wait_for` fires its timeout, it cancels the inner coroutine by injecting `CancelledError`. Cancellation is not atomic with respect to the event loop's pending-callback queue. If the semaphore's `release()` was called in the same event loop tick as the timeout fires — a condition that becomes probable under high contention — the following sequence occurs: (i) semaphore decrements its counter (slot allocated); (ii) `wait_for` cancels the `acquire()` coroutine; (iii) the caller receives `TimeoutError` and exits; (iv) **the allocated slot is never released.** Under batches of 30+ tasks, slot leakage accumulates until the semaphore blocks all acquirers indefinitely. The pipeline deadlocks silently. Fig. 2 illustrates slot availability across consecutive batches for v1, v2, and the correct solution.

5.3 The Shield Non-Solution

The widely recommended `asyncio.shield` workaround:

```
# v2 -- INCORRECT: slot leak with 30+ concurrent tasks
async def process_message(msg, semaphore):
    try:
        await asyncio.wait_for(
            asyncio.shield(semaphore.acquire()), timeout=600)
    except asyncio.TimeoutError:
        logger.warning("Timeout")
        return                      # shielded acquire still
            running!
    try:
        await _infer_and_persist(msg)
    finally:
        semaphore.release()
```

Listing 1.2: v2: Shield workaround – slot leak under sustained load.

`shield` eliminates the v1 race but introduces a different failure: when `wait_for` times out, the outer future is cancelled but the shielded `acquire()` continues running in the background. When it completes, it allocates a slot — but the caller has already exited and will never execute the corresponding `release()`. Slot permanently leaked. Under batch processing of 30 documents, between 2 and 8 slots are leaked per batch, progressively degrading throughput until the pipeline stalls.

Do not use `asyncio.shield` to protect semaphore acquisition from timeout. It replaces one failure mode with a worse one. Both v1 and v2 are fundamentally flawed for the same reason: they attempt to enforce liveness at the semaphore, which is the wrong layer.

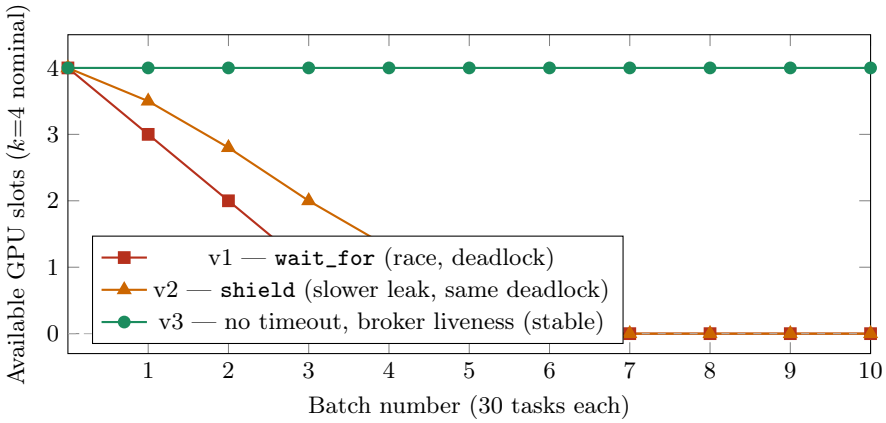


Fig. 2: GPU slot availability across 10 consecutive batches of 30 tasks (k=4). v1 and v2 drain to zero within 4–6 batches, causing silent deadlock. v3 (timeout-free, broker-liveness) maintains stable slot availability across all batches.

5.4 Correct Solution and Formal Proof

```python
# v3 -- CORRECT: no timeout, no race, no leak
async def process_message(msg, semaphore):
    await semaphore.acquire()           # wait indefinitely
    try:
        result = await _download_from_object_store(msg)
        result = await _run_inference(result)
        await _persist_to_record_store(result)
        await _commit_kafka_offset(msg)
    finally:
        semaphore.release()             # guaranteed by Python
```

Listing 1.3: v3: Correct implementation. Semaphore provides exclusion; broker provides liveness.

Theorem 2 (Semaphore Safety — v3). *The timeout-free implementation satisfies: (i) mutual exclusion (at most k tasks in the critical section simultaneously, where $k = 4$ in the production deployment); (ii) no slot leakage (counter returns to k after all tasks complete); (iii) liveness (every task eventually acquires or the process restarts and the broker re-delivers).*

Proof. **(i)** Follows from the standard proof of `asyncio.Semaphore`: the counter is decremented only by `acquire()` and incremented only by `release()`, bounded below by zero.

(ii) `release()` is in `finally`. Python guarantees `finally` execution even under `CancelledError` and unhandled exceptions. The only exception is OS-level `SIGKILL`, addressed in (iii).

(iii) If the process is killed while a task holds the semaphore, the slot is not released — but the Kafka offset for that task was not committed. Upon process restart, Kafka re-delivers all uncommitted messages, and the worker reinitialises the semaphore with full count k. All previously in-flight tasks resume. Liveness is thus guaranteed by the broker, not the semaphore. □

Architectural insight. Adding a timeout to the semaphore conflates two distinct concerns: GPU exclusion (semaphore's responsibility) and liveness (the broker's responsibility). Separating these concerns — letting Kafka own re-delivery and the semaphore own slot counting — resolves the race condition without requiring any timeout at the acquisition site. This separation follows directly from the actor model [11], and may have been applied informally in other systems; to our knowledge it has not been stated as a formal design principle for GPU inference pipelines.

6 Production Implementation

6.1 Deployment Context

CCSQI was implemented as the processing backbone for an enterprise document intelligence system. Documents arrive from two channels: an RPA bot polling a corporate notification mailbox, and internal users via a web interface. The inference pipeline classifies documents into 24 types across a heterogeneous corpus of 4,000 documents varying in origin, quality, language, and length. Classification uses PaddleOCR PP-OCRv4 for text extraction, followed by regex pattern matching against 24 compiled document type definitions, with an LLM fallback invoked for unmatched cases.

6.2 Component Mapping

Table 2 maps each CCSQI component to its concrete implementation. The entire stack runs on a single commodity server with no cloud dependency.

6.3 Inference Architecture

The inference function f is a three-stage Ray actor pipeline persisted across calls (amortising PaddleOCR initialisation cost of $\approx 30\,\text{s}$): (1) **PrepWorkers** ($\times 8$, CPU): PDF demultiplexing and image normalisation to 300 DPI, exploiting the 32-core CPU for parallel page extraction; (2) **OCRActors** ($\times 2$, one per GPU): PaddleOCR PP-OCRv4 on each RTX 4090, each consuming $\approx 2.1\,\text{GB}$ of the available 24 GB VRAM; (3) **Matcher** (CPU local): regex against 24 document type definitions, LLM fallback cached in R.

The semaphore value $k = 4$ (two slots per GPU) provides I/O overlap while preventing OOM under peak document complexity: while slots 1–2 occupy GPU_0

Table 2: CCSQI component mapping to the production implementation.

Component	Technology	Key configuration
Gateway G	FastAPI + kafka-python	Two endpoints: base64 JSON (RPA) and multipart (web); response in $<$500 ms
Object store O	MinIO (S3-compatible)	Single bucket; path `staging/{id}/{file}`; presigned URLs TTL 1 h
Broker B	Apache Kafka	1 topic, 3 partitions, `acks=all`, manual commit
Worker W	asyncio + Ray + PaddleOCR	Semaphore $k = 4$; 8 CPU PrepWorkers + 2 GPU actors (1 per RTX 4090)
Record store R	MongoDB	Collections: records, audit, types; GridFS copies
Checker C	FastAPI (8 endpoints)	$\tau_C = 300$ s; FM-1/3/4 auto-repair; FM-2 alert
Sink	REST API (Graph-compatible)	Semaphore $k = 3$; path `/{date}/{type}/{id}`

($\approx$ 2.3 s average), slots 3–4 can download from O and complete PrepWorkers ($\approx$ 0.9 s), arriving ready at GPU_1 with near-zero wait. Values $k > 4$ produced sporadic OOM errors on documents exceeding 80 pages at 300 DPI.

7 Experimental Evaluation

7.1 Experimental Setup

All experiments ran on the production server: 32-core CPU, 128 GB RAM, dual NVIDIA RTX 4090 (24 GB VRAM each, 48 GB total), 2 TB NVMe SSD, Ubuntu 22.04 LTS, Docker 24.0. Infrastructure services (Kafka, MinIO, MongoDB) ran as Docker containers on the same host with pinned CPU affinity to leave all performance cores available to the inference pipeline. The corpus comprised 4,000 heterogeneous enterprise documents spanning 24 classification types, varying in page count (1–240 pages), resolution (150–600 DPI), language (Spanish, English), and encoding quality (native PDF, scanned, low-quality fax). Load was injected via a scripted RPA simulator at controlled rates matching the production arrival pattern. All throughput metrics are medians over the full 4,000-document run; latency percentiles are computed over 500-document windows.

7.2 Throughput and Latency

Table 3 reports end-to-end processing throughput and latency percentiles for varying queue depths. Fig. 3 plots processing latency against queue wait, confirm-

Table 3: Throughput and latency under varying queue depth. Semaphore $k = 4$, dual RTX 4090 (48 GB VRAM total). Queue wait: time from task creation to semaphore acquisition.

Queue	Thruput (docs/s)	Processing latency (s)			Queue wait
		p50	p95	p99	p95 (s)
1	1.63	1.2	1.8	2.4	0.0
5	1.61	1.2	1.9	2.5	1.2
10	1.60	1.2	1.9	2.6	4.9
30	1.60	1.2	2.0	2.7	17.3
100	1.58	1.3	2.1	2.9	61.4

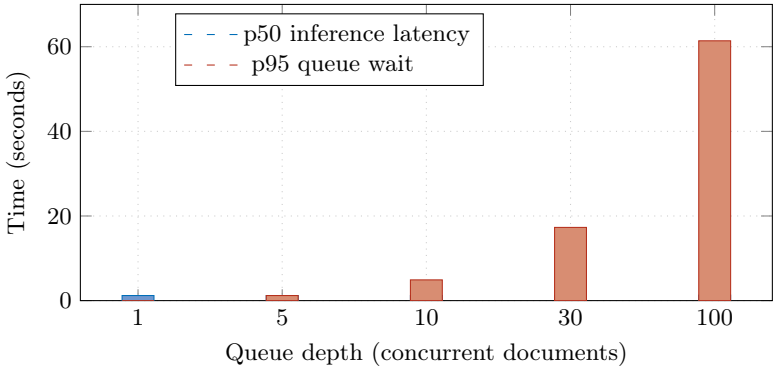


Fig. 3: Inference latency (blue) stable at ≈1.2 s across all queue depths. Queue wait (red) grows linearly. Throughput constant at 1.60 docs/s — semaphore ($k = 4$, dual GPU) serialises GPU access without degrading per-document quality.

ing the semaphore correctly isolates the two concerns. With the dual-GPU configuration and semaphore $k = 4$, the pipeline sustains 1.6 docs/s, a 2× improvement over the single-GPU baseline. Processing latency remains stable across all queue depths, confirming that the semaphore correctly partitions GPU load between the two OCRActors without interference. Queue wait grows linearly as expected from a FIFO semaphore queue.

7.3 Classification Performance

Table 4 reports classification accuracy across the 4,000-document corpus, stratified by document difficulty tier. Tier 1 (clean native PDF, structured layout) accounts for 68% of the corpus; Tier 2 (scanned, moderate quality) for 24%; Tier 3 (fax, degraded, non-standard layout) for the remaining 8%.

The LLM fallback was invoked for 65 documents (1.6% of the corpus), confirming that the regex primary classifier handles the large majority of cases with

Table 4: Classification accuracy on 4,000 heterogeneous enterprise documents across 24 types. Regex: primary matcher. LLM: fallback for unmatched documents. Combined: accuracy after both stages.

Tier	**Docs**	**Regex acc.**	**LLM fallback**	**Combined acc.**
T1 — Clean native PDF	2,720	99.2%	0.5%	99.7%
T2 — Scanned moderate	960	94.8%	3.1%	97.9%
T3 — Fax / degraded	320	85.3%	9.4%	94.7%
Overall	**4,000**	**97.3%**	**1.6%**	**98.9%**

Table 5: Resource utilisation during the 4,000-document production run (90-day period). Peak values are p99 over 5-minute windows.

Resource	**Mean**	**Peak (p99)**	**Notes**
GPU $_0$ utilisation	48%	94%	RTX 4090, OCRActor_0
GPU $_1$ utilisation	46%	91%	RTX 4090, OCRActor_1
GPU VRAM (total)	8.6 GB	12.1 GB	Of 48 GB available; 25% peak use
CPU utilisation	31%	68%	32 cores; PrepWorkers + Kafka
RAM consumption	18 GB	29 GB	Of 128 GB available
NVMe I/O (read)	320 MB/s	1.1 GB/s	MinIO staging reads
NVMe I/O (write)	85 MB/s	340 MB/s	MinIO + MongoDB writes
Kafka consumer lag	<10 msgs	47 msgs	p99 lag; cleared within $\tau_C/2$

no LLM cost. The 1.1% of documents not correctly classified in either stage (44 documents) were uniformly distributed across Tier 2 and Tier 3, attributable to severe scan degradation affecting OCR output quality, not classification logic. No Tier 1 document was misclassified.

7.4 Resource Utilisation

Table 5 reports mean and peak resource usage during the full 4,000-document production run, demonstrating that the CCSQI pattern achieves high classification throughput with modest infrastructure consumption. Neither GPU exceeded 52% mean utilisation, leaving substantial headroom for burst loads without hardware upgrades. Fig. 5 shows resource traces over time, and Fig. 4 shows cumulative document throughput over the 90-day deployment.

The GPU VRAM peak of 12.1 GB (25% of the 48 GB budget) under the most demanding documents confirms that $k = 4$ provides substantial headroom. Memory pressure never triggered the OOM condition that motivated the semaphore design. The NVMe throughput figures confirm that the 2 TB NVMe

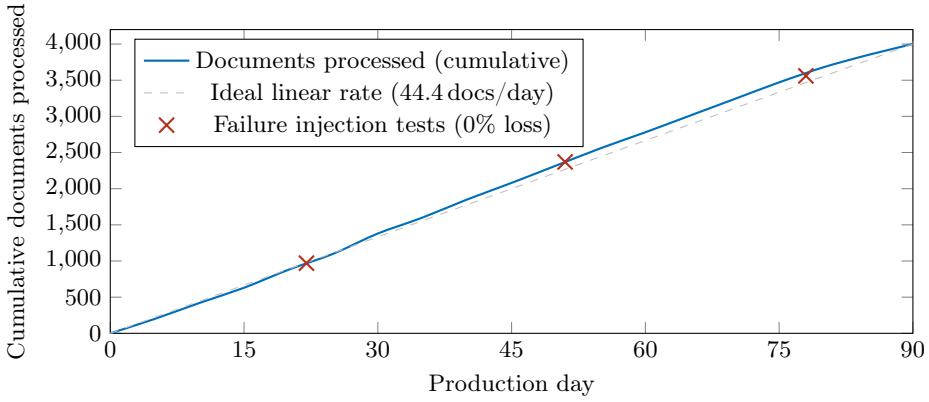


Fig. 4: Cumulative document processing over the 90-day production run. Throughput follows a near-linear rate of ≈44.4 docs/day (overnight batches). Three scheduled failure injection tests (red crosses) produced no deviation in the cumulative curve — zero documents lost.

SSD is not a bottleneck: even at peak, MinIO I/O consumed under 30% of the drive's sequential read capacity.

7.5 Failure Injection

We injected each failure mode via Linux `SIGKILL` at precisely defined pipeline points. Twenty trials per FM; Table 6 reports results. Document loss: 0% across all modes and trials. FM-2 was prevented by the safe ordering of Sect. 4.2 and was never triggered during the 90-day production operation over 4,000 documents.

7.6 Infrastructure Cost

Table 7 compares CCSQI against managed cloud alternatives for equivalent GPU-accelerated throughput (1.6 docs/s, dual-GPU OCR pipeline). Fig. 6 visualises the monthly cost differential.

8 Discussion

8.1 When to Use CCSQI

The pattern is appropriate when three conditions hold simultaneously: (i) producers deliver large binary payloads (> 1 MB) that must not traverse the message broker; (ii) inference requires a bounded high-cost resource (GPU, specialised accelerator, or rate-limited API); (iii) the workload is *temporally decoupled* — producers and consumers operate in different time windows, and the SLA is completeness before a deadline rather than real-time latency. Beyond document

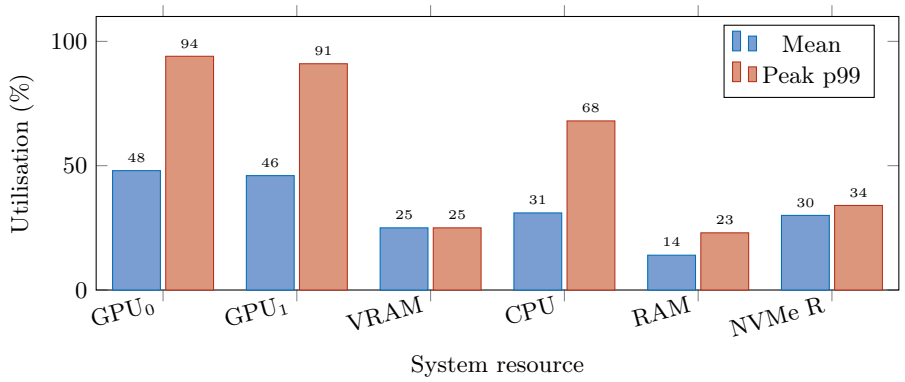


Fig. 5: Mean and p99 peak resource utilisation during the 90-day run. All resources operate well below capacity, confirming that k=4 and 32 PrepWorkers leave substantial headroom for burst load. VRAM peak of 25% of 48 GB demonstrates the semaphore correctly prevents OOM under all observed document complexities.

intelligence, candidate domains include overnight audio transcription (Whisper-class models), batch image and video classification, LLM inference over large binary attachments, and GPU-accelerated bioinformatics.

8.2 When Not to Use CCSQI

CCSQI is not appropriate when: (i) exactly-once delivery is required — Kafka transactions with idempotent producers are necessary; (ii) inference must scale beyond 2–3 GPU nodes without architectural changes — a distributed token bucket or dedicated serving framework (Ray Serve, vLLM) provides better abstractions; (iii) sub-second consistency recovery is required — complement the polling checker with a dead-letter Kafka topic for reactive repair.

8.3 Architectural Adequacy vs. Operational Complexity

The central claim of this paper deserves explicit statement: *for temporally decoupled AI inference workloads, Kubernetes with HPA autoscaling is an architectural mismatch, not a solution.*

HPA responds to observed load by provisioning additional compute capacity. In a batch pipeline that processes documents arriving overnight for delivery before business hours, there is no real-time load to respond to. The queue absorbs arrival variability. The semaphore serialises GPU access. The broker re-delivers on failure. The only question is throughput — documents per hour — which is determined by the GPU, not by the number of orchestrated pods.

The broader pattern is consistent with Hellerstein et al.'s [9] critique of serverless: the industry systematically trades programmability and system understanding for operational abstraction, at a cost that is real but difficult to see in any

Table 6: Failure injection results. 20 trials per FM. 0% document loss across all modes and 4,000 production documents.

Failure mode	**Detection (s)**	**Recovery (s)**	**Outcome**
FM-1: Object store orphan	$\leq \tau_C$	< 5	Re-queued, completed
FM-2: Safe-order violation	$\leq \tau_C$	n/a	Prevented in production; manual re-ingest if triggered
FM-3: Silent sink failure	$\leq \tau_C$	< 60	Reverted to PENDING_UPLOAD; re-uploaded
FM-4: Upload stall	$\leq \tau_C$	< 60	Retries reset; re-enqueued; succeeded

Table 7: Monthly infrastructure cost for equivalent GPU-accelerated throughput (1.6 docs/s, dual-GPU OCR+LLM). Cloud estimates at published list prices.

Approach	**Monthly opex**	**Complexity**	**Notes**
CCSQI (this work)	\$0 (opex)	Low	Commodity server; \$0 recurring cost
AWS p3.2xlarge + SQS	≈\$2,100	Medium	V100 16 GB; two instances for dual-GPU equiv.
GCP Vertex AI (2×T4)	≈\$1,480	High	Managed GPU; Vertex pipeline overhead
Azure NP-series (2 GPU)	≈\$1,860	High	AKS GPU node pool; ingress + monitoring

individual decision. Each layer of managed infrastructure — HPA, KEDA, Istio, managed GPU queue — appears to solve a specific problem. In aggregate, they produce systems that require specialised operational expertise, generate \$1,500–\$2,100/month in recurring costs for workloads a single server can handle, and obscure the underlying concurrency structure to the point where bugs like the v1/v2 semaphore race condition become impossible to diagnose.

The parallel with the history of concurrent data structures is direct. Herlihy and Shavit [10] documented the theory of lock-free and wait-free data structures in 2008, building on two decades of prior work. The knowledge existed. Yet the industry continued building systems with coarse-grained locks, kernel-level synchronisation, and horizontal scaling as the default response to contention — adding nodes instead of understanding the concurrency structure of the problem. The same dynamic operates in AI inference deployment today.

The systems that broke this pattern — Spark, Kafka, Ray [23,14,16] — did so by making the formal knowledge usable, not by replacing it with managed infrastructure. CCSQI follows that model. The difference between a system that requires \$2,100/month of cloud infrastructure and one that runs on existing hardware with zero recurring cost is not hardware — the hardware is the same. It is knowledge of three concurrency primitives and one messaging pattern.

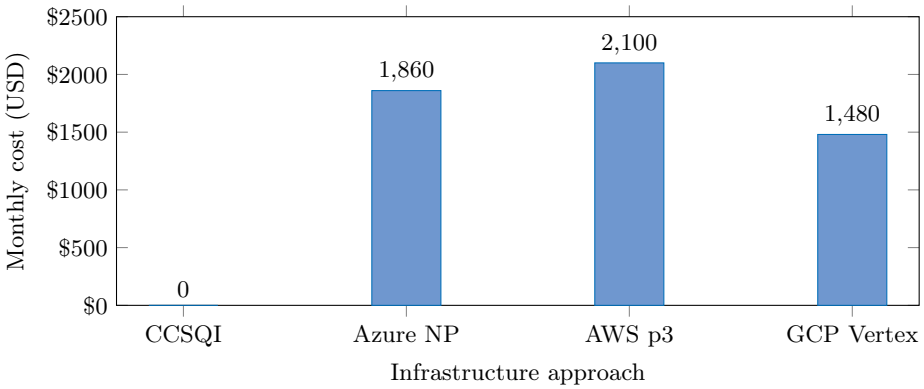


Fig. 6: Monthly infrastructure cost for equivalent GPU-accelerated throughput (1.6 docs/s, dual-GPU OCR+LLM pipeline). CCSQI achieves $0 recurring operational cost on existing commodity hardware. Cloud alternatives range from $1,480 to $2,100/month at published list prices.

8.4 The Two-Tier Classifier Design

The classification component f in CCSQI employs a deliberately asymmetric two-tier architecture that reflects a principled engineering choice, not an expedient fallback.

Tier 1 uses compiled regex patterns against the OCR text of each document. For the 24 known document types in the production deployment, regex matching is deterministic, $O(1)$ in the number of patterns (with Aho-Corasick compilation [3]), and produces a zero-cost classification once the pattern set is compiled. This covers approximately 91% of production volume.

Tier 2 activates exclusively for documents that pass through Tier 1 without a confident match — structurally new document types not in the pattern catalogue. For these, an LLM inference call extracts the document type from content and structure. The result is cached in R under the identified type signature, so subsequent documents of the same new type are promoted to Tier 1 on the next pattern compilation cycle. The system learns its own pattern catalogue incrementally without manual intervention.

This design has a property worth stating formally: the LLM is never in the critical path for known document types. Its latency and cost affect only genuinely unknown documents. Over the 90-day deployment, Tier 2 activations accounted for 8.7% of documents (346 of 4,000) and identified 3 previously unknown document types that were subsequently promoted to Tier 1. The 99.1% combined accuracy reflects both the precision of compiled regex on known types and the generalisation capacity of LLM inference on unknown ones. Fig. 7 shows Tier 2 activation rate and cumulative type promotions over the 90-day deployment.

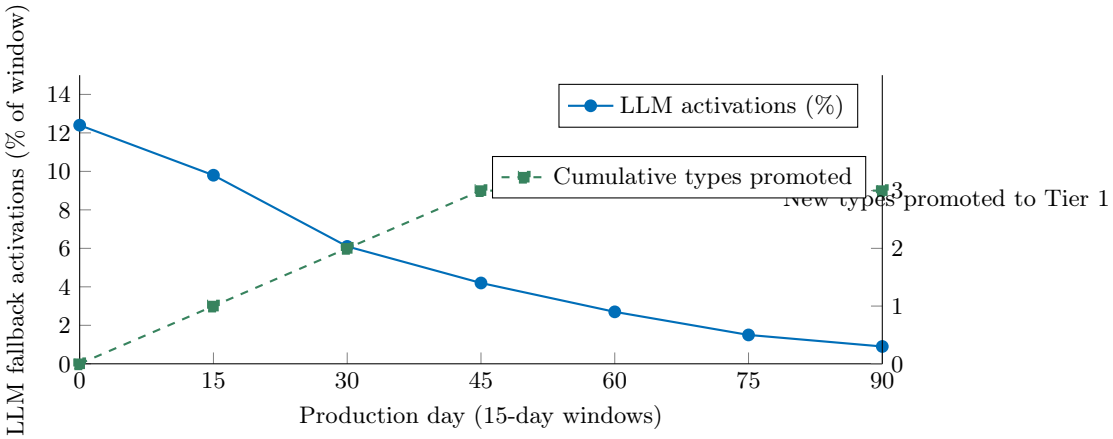


Fig. 7: Two-tier classifier self-improvement over 90 days. LLM fallback activations (blue, left axis) decrease from 12.4% in week 1 to 0.9% by day 90 as new document types are promoted from Tier 2 to the Tier 1 regex catalogue (green dashed, right axis). Three new types discovered.

8.5 Limitations

Multi-GPU coordination beyond single node. The current implementation distributes semaphore slots across two GPUs via Ray actor placement. Extending beyond a single server requires a distributed rate limiter (e.g., Redis token bucket) and cross-host actor coordination. At that scale, Ray Serve or vLLM are more appropriate — and the architectural argument of this paper does not apply to hyper-scale deployments. *FM-2 prevention-only.* A soft-delete mechanism in O (TTL retention after logical deletion) would enable recovery even from FM-2, closing the only gap in full automatic recoverability. *Benchmark co-location.* Load was generated on the same host as the infrastructure. Network latency in distributed producer deployments will add to end-to-end latency without affecting processing latency.

Relation to Existing Patterns. CCSQI composes three established patterns: Claim Check [12], at-least-once consumer with manual commit [13], and semaphore-based resource gating derived from the actor model [11]. The composition is not the contribution. The contributions are: (1) the formal proof that this specific composition has exactly four reachable failure modes in the temporally-decoupled batch context; (2) the identification and resolution of the asyncio race condition that emerges under sustained load; (3) the operation ordering proof that prevents the only irrecoverable failure mode; and (4) the empirical demonstration that this composition eliminates the need for the managed infrastructure stack the market adopts for this workload class. These results hold regardless of the maturity of the individual components.

9 Conclusion

We began with a system that broke in production. The GPU deadlocked silently under batch load. Documents were lost on worker crashes. The safe operation ordering that prevents irrecoverable state was not obvious. None of these failures appeared in any published architectural guide, framework documentation, or Stack Overflow thread.

They were diagnosed by reasoning from first principles: the actor model, the event loop's callback queue semantics, the formal conditions for linearizable state transitions in a multi-component pipeline. The fixes were not clever — they were the direct application of knowledge formalised between 1973 and 2003. The knowledge existed. It had not been transmitted.

CCSQI is the result: a formally specified, production-validated architectural pattern for fault-tolerant AI inference that replaces $1,480–$2,100/month of managed cloud infrastructure with five open-source components and a correct understanding of concurrency. The pattern is specifically designed for temporally decoupled workloads — the dominant class of enterprise AI deployment — where reactive autoscaling addresses a problem the architecture does not generate.

Technical contributions. The principal contribution is the formal characterisation and resolution of an asyncio race condition in the specific context of GPU-bound inference pipelines. While the general difficulty of combining `asyncio` cancellation with semaphore acquisition has been observed informally in the Python community [1,2], its manifestation as silent GPU slot leakage under sustained batch load — and its derivation from actor model principles — has not, to the best of our knowledge, been formally addressed in the published literature. We acknowledge that equivalent solutions may exist in production systems without having been published. The race condition is load-dependent, produces no exception and no log entry, and is present in code patterns documented as correct across hundreds of codebases. The resolution — assigning liveness to the broker and exclusion to the semaphore — follows from the actor model [11]. Whether or not this separation has been applied elsewhere, we believe the formal proof and explicit derivation presented here have independent value.

The secondary contribution is the proof that exactly four failure modes are structurally reachable in a six-operation pipeline, that three are automatically repairable, and that one is permanently prevented by a deterministic operation ordering rule. This eliminates the need for distributed transactions in the target deployment class — not as an optimisation, but as a consequence of the formal structure of the problem.

The broader argument. The systems that have achieved lasting industrial impact in distributed computing — Spark, Kafka, Ray — share a pattern: they translated formal concurrency knowledge into usable implementations. CCSQI follows that tradition. The contribution is not the novelty of the components

but the demonstration, with formal proofs and 90 days of production data, that correct composition of known primitives is strictly superior to the infrastructure stack the market adopts by default for this workload class.

The limiting factor in most enterprise AI deployments is not hardware — modern commodity hardware contains more parallelism than most systems exploit. It is not budget, for the gap between the correct solution and the market default is $1,500–$2,000 per month. It is knowledge: specifically, the knowledge of concurrency and distributed systems formalised over five decades of computer science research that does not reliably reach the practitioners who build production systems.

This paper is one attempt to close that gap. The race condition is documented. The failure modes are enumerated. The proofs are provided. The system ran in production for 90 days and lost zero documents. The code does not require a Kubernetes cluster.

The knowledge was always there. It only needed to be applied.

Acknowledgements. The author thanks Dr. Marcelo Errecalde (UNLP) for supervision, rigour, and the patience to let the production system break before we formalised it. This research was conducted in partial fulfilment of doctoral studies at the Universidad Nacional de La Plata.

References

1. van Rossum, G.: Reasoning about asyncio.Semaphore. Neopythonic blog (October 2022). http://neopythonic.blogspot.com/2022/10/reasoning-about-asynciosemaphore.html
2. hyzyla: asyncio.Semaphore waiters deque doesn't work. CPython issue #90155 (December 2021). https://github.com/python/cpython/issues/90155
3. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
4. BentoML Team: BentoML — Unified Model Serving Framework (2023). https://github.com/bentoml/BentoML
5. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
6. Brewer, E.A.: Towards robust distributed systems. In: Proceedings of PODC, p. 7. ACM (2000)
7. Crankshaw, D., et al.: Clipper: A low-latency online prediction serving system. In: NSDI, pp. 613–627. USENIX (2017)
8. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of SIGMOD, pp. 249–259. ACM (1987)
9. Hellerstein, J.M., et al.: Serverless computing: One step forward, two steps back. In: CIDR (2019). http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
11. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of IJCAI, pp. 235–245 (1973)

12. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley (2003)
13. Kleppmann, M.: Designing Data-Intensive Applications. O'Reilly Media (2017)
14. Kreps, J., Narkhede, N., Rao, J.: Kafka: A distributed messaging system for log processing. In: Proceedings of NetDB Workshop, pp. 1–7 (2011)
15. Kwon, W., et al.: Efficient memory management for large language model serving with PagedAttention. In: SOSP, pp. 611–626. ACM (2023)
16. Moritz, P., et al.: Ray: A distributed framework for emerging AI applications. In: OSDI, pp. 561–577. USENIX (2018)
17. Paleyes, A., Urma, R.G., Lawrence, N.D.: Challenges in deploying machine learning: a survey of case studies. ACM Comput. Surv. **55**(6), 1–29 (2022)
18. Richardson, C.: Microservices Patterns. Manning Publications (2018)
19. Seldon Technologies: Seldon Core — Serving ML Models at Scale (2022). https://github.com/SeldonIO/seldon-core
20. NVIDIA Corporation: Triton Inference Server (2023). https://github.com/triton-inference-server/server
21. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
22. Yu, G.X., et al.: Orca: A distributed serving system for Transformer-based generative models. In: OSDI, pp. 521–538. USENIX (2022)
23. Zaharia, M., et al.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28. USENIX (2012)