DEFENSIVE PUBLICATION — TECHNICAL DISCLOSURE (TDCOMMONS DEPOSIT COPY) · 2026-06-25

Keywords: defensive-publication, prior-art, ai-agents, llm, queueing, amortized-triage, workload-management

Two-Tier Attention Queue with Amortized Triage

A Bounded In-Memory Attention Tier Overflowing to an Unbounded Persistent Tier, with an Exactly-Once Triage Invariant for Burst-Resilient, Cost-Bounded LLM Agents

Field	Value
Document type	Technical Defensive Publication (public prior art)
Publisher / Copyright holder	Gus IT LLC (Florida, USA)
Author	Gustavo Assuncao, PhD
Publication date	2026-06-25
Version	1.0
Classification	Public
License	AGPL-3.0-or-later (copyleft; commercial license available)
Deposit channel	To be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is claimed at time of writing.
Status	Published — intended to bar later patenting of the disclosed technique by third parties.

Abstract

An autonomous agent backed by a large language model (LLM) frequently receives inputs in bursts: two hundred emails delivered at once, a flood of webhook events, a queue of chat messages that arrived during a network partition. The dominant cost in processing such a burst is *triage* — the act of classifying each input into a type and a priority so the agent knows what to do with it. When triage itself invokes the LLM, a burst of *N* inputs produces *N* LLM calls *before any useful work begins*, a thundering-herd that spikes token spend and latency in lock-step with arrival rate. This disclosure describes a **two-tier attention queue** that decouples burst *absorption* from burst *cost*. A bounded in-memory **hot tier** (default capacity 50) holds items awaiting cognitive focus; when it overflows, items spill to an unbounded persistent **cold tier** backed by a durable store. The defining invariant is **amortized, exactly-once triage**: each item is triaged at most once, at intake, and its assigned type and priority are *persisted with the item* so they survive every subsequent memory↔disk transition. Overflow to the cold tier carries the triage result with the payload and never re-triages; promotion back to the hot tier reads the stored classification rather than recomputing it. Dequeue is **memory-first**: the hot tier is drained before the cold tier, exploiting cache and context warmth. The combination yields burst absorption *without* proportional LLM spend, a steady-state processing rate bounded by worker throughput rather than arrival rate, and a hard ceiling on triage cost equal to the number of *distinct* items rather than the number of *queue transitions*. We present the architecture, the exactly-once triage state machine, the data model, an original clean-room reference implementation, a worked 200-email scenario, failure-mode analysis, framework alignment, an evaluation

methodology, and a set of inventive claims. This document is published defensively to keep the technique freely practiceable.

---

1. Executive Summary

1.1 Thesis

The cost of handling an input burst in an LLM agent is dominated not by the work the burst represents but by the *classification* of the burst. If classification is coupled to queue mechanics — re-run whenever an item moves between buffers, retried on every dequeue, recomputed after a restart — then a system that merely *buffers* a burst still *pays* for it proportionally. The novel contribution is to make triage a **property of the item**, computed once and carried forever, so that buffering becomes free of classification cost. A bounded hot tier plus an unbounded persistent cold tier provides the buffering; an exactly-once triage invariant provides the cost bound; memory-first dequeue provides the locality benefit.

1.2 Contributions

#	Contribution	Section
C1	A two-tier attention queue: bounded in-memory hot tier + unbounded persistent cold tier, with a defined overflow and promotion protocol.	§6, §7
C2	The **exactly-once triage invariant**: triage runs at most once per item, at intake; the result is persisted with the item and reused across all memory↔disk transitions.	§7.2
C3	**Triage-result migration**: overflow and promotion move the *classification together with the payload*, so no buffer transition triggers re-triage.	§7.3
C4	**Memory-first dequeue** that drains the hot tier before the cold tier to exploit context/cache warmth.	§7.4
C5	A cost model proving triage invocations are bounded by *distinct items*, not *queue transitions* or *arrival rate*.	§7.6, §13
C6	A fail-open posture: a triage failure degrades to a safe default classification rather than blocking intake, and the persistence layer is the source of truth across restarts.	§11
C7	A clean-room reference implementation reducing the method to practice.	§9, Appendix C

1.3 Headline Claim

> A method for handling input bursts in an LLM-backed agent that maintains a bounded in-memory primary queue and an unbounded persistent secondary queue, triages each item **exactly once** at intake, persists the triage result with the item, transfers items to the secondary queue on overflow **without re-triage**, and on dequeue drains the primary queue before the secondary — such that the number of triage (LLM) invocations is bounded by the number of distinct items rather than the number of queue transitions or the arrival rate.

1.4 Scope — What It Is / Is Not

It is: a workload-management pattern for the *intake and scheduling* stage of an LLM agent; a way to make burst absorption cost-independent of arrival rate; a two-tier buffering discipline with a once-only

classification invariant.

It is not: a general-purpose message broker, a replacement for a durable queue product, a scheduler for distributed compute, a rate limiter, or a model-inference cache. It does not claim novelty in two-level buffering *per se* (that is ancient); its novelty lies in the exactly-once triage invariant and its migration across tiers in the *LLM-triage* setting where each classification is an expensive model call.

---

2. Introduction & Motivation

2.1 The concrete problem

Consider a cognitive persona — an autonomous agent that monitors a mailbox, a chat channel, and a webhook endpoint. Inputs arrive asynchronously. For each input the agent must decide *what kind of thing this is* (an email needing a reply? a multi-step task? a tool invocation? a voice callback?) and *how urgent it is*. This decision is **triage**. In a modern agent, triage is often performed by the LLM itself: the raw input is handed to the model with an instruction to emit a structured `{type, priority}` classification, because heuristics are brittle against natural-language inputs.

Now deliver a burst. A mail server that was offline for an hour reconnects and delivers 200 queued emails in one second. A CRM integration replays 500 webhook events. A chat client flushes a backlog. Each input demands triage; each triage is an LLM call. The agent faces a **thundering herd**: 200 concurrent (or rapidly serial) model calls, *before a single reply is sent or a single task is done*.

2.2 The "tax" — why this is expensive

The cost has three components:

1. **Token spend.** Each triage call consumes input tokens (the item body, truncated) and output tokens (the classification). At burst scale this is a step-function spike in spend that tracks arrival rate, not value delivered.
2. **Latency and rate-limit exposure.** Provider rate limits (requests-per-minute, tokens-per-minute) are hit during the burst, causing 429s, backoffs, and retries — which, if naively coded, *re-triage* and multiply the cost.
3. **Head-of-line blocking.** If triage and work share a budget or a worker pool, the burst of triage starves the actual work the agent exists to do.

A naive buffer does not fix this. Suppose the agent buffers the burst in a durable queue and processes items one at a time. If triage is performed *at dequeue* — the natural place, since that is "when you look at the item" — then every dequeue is a triage call, and a restart that re-reads the queue re-triages everything. The buffer absorbed the burst in *time* but not in *cost*.

2.3 Why existing approaches fall short

- **Plain durable queues / brokers** (RabbitMQ, SQS, Kafka, Linux IPC message queues) provide buffering and overflow-to-disk, but they are *content-agnostic*: they carry an opaque payload and have no notion of a per-item *classification* that must be computed once and reused. Re-classification

on redelivery is the application's problem, and the common application pattern re-classifies on every consume.
- **In-memory work queues** with priority ordering (heaps, ring buffers) bound memory but have no overflow story and lose everything on restart.
- **LLM response caches** (semantic caches, prompt caches) reduce *repeat* inference for *identical or similar* inputs, but a burst of 200 *distinct* emails has 200 distinct triages — caching does not help.
- **Admission control / rate limiting** throttles intake, trading latency for cost, but does not reduce the per-item triage count; it just spreads it over time. It also drops or delays work.

None of these enforces the specific invariant *triage is computed once per distinct item and migrates with the item across a bounded→unbounded tier boundary*.

2.4 Why now

LLM agents that *autonomously* consume external inputs (mailboxes, webhooks, chat) are newly common. Their triage step is newly *expensive* because it is an LLM call, not a regex. And their inputs are newly *bursty* because they ride on integrations (mail, webhooks) that batch and replay. The conjunction of expensive triage and bursty arrival is what makes the exactly-once invariant valuable now.

---

3. Problem Statement

3.1 Formal framing

Let an agent receive a stream of items $i_1, i_2, \dots$ Each item i requires a triage function $T(i) \rightarrow (\text{type}, \text{priority})$ where T is an LLM call of cost c_T (tokens, latency, rate-limit budget). The agent processes items via a worker of throughput μ items/second. Arrivals occur in bursts: over a window the arrival rate λ may exceed μ by orders of magnitude.

A correct system must (a) not lose items, (b) keep memory bounded, (c) prioritize urgent items, and (d) **minimize total triage cost** $\sum c_T$. The number of triage invocations K is the cost driver. We seek a discipline in which

$$K = |{\text{distinct items}}|$$

— each item triaged exactly once — independent of how many times items move between buffers, how many restarts occur, or how high λ spikes. A naive design instead yields $K = \sum_i (\text{number of times } i \text{ is dequeued or re-read})$, which is unbounded above by item count.

3.2 Derived requirements

Req	Requirement	Rationale	Addressed in
R1	Absorb arrival bursts where $\lambda \gg \mu$ without unbounded memory growth.	Bursts are the trigger condition.	§6, §7.1
R2	Triage each distinct item **at most once**, ever.	Triage is the cost driver.	§7.2
R3	The triage result (type, priority) must persist with the item across memory↔disk transitions.	So no transition re-triages.	§7.3, §8
R4	Overflow from the bounded tier to the unbounded tier must be lossless and carry the triage result.	Durability + R2.	§7.3
R5	Promotion from the unbounded tier back to the bounded tier must reuse the stored classification, not recompute it.	R2 across the round trip.	§7.3
R6	Dequeue order prefers the in-memory tier (memory-first) to exploit warmth.	Locality benefit.	§7.4
R7	Within a tier, items are ordered by priority/weight, not raw FIFO.	Urgency must win.	§7.5
R8	A triage failure must not block intake; it must degrade to a safe default.	Fail-open availability.	§11
R9	The persistent tier is the source of truth across process restarts.	Crash safety.	§8, §11
R10	Configuration (capacity, thresholds, default classification) must be externally configurable, not hardcoded.	Operability.	§7.7
R11	The total triage-invocation count must be observable and bounded by distinct-item count.	Auditability of the cost claim.	§13
R12	Urgent items may interrupt in-flight focus without being re-triaged.	Responsiveness without cost.	§7.5, §10

4. Related Work & Prior Art

This work *builds on* well-established queueing and scheduling foundations and contributes a specific invariant in a specific setting. We name the foundations explicitly.

- **Two-level / hierarchical buffering** — the idea of a fast bounded buffer that overflows to a slower unbounded store is foundational and ancient: CPU cache → RAM → disk hierarchies, the buffer-pool / spill-to-disk design in databases (e.g., sort and hash operators spilling to temp), and OS page caches. We adopt the hot/cold structure directly and claim no novelty in it.
- **Durable message brokers** — RabbitMQ, Apache Kafka, Amazon SQS, Google Pub/Sub, and POSIX/SysV message queues provide durable buffering, overflow, redelivery, and (sometimes) priority. We adopt durable overflow semantics. These systems are *payload-agnostic* and have no per-item once-only classification step.
- **Priority queues and weighted scheduling** — binary heaps, calendar queues, weighted fair queueing (WFQ), and multilevel feedback queues (MLFQ) inform the in-tier ordering by weight. We adopt priority ordering within a tier.
- **Producer–consumer with backpressure** — bounded-buffer producer/consumer (Dijkstra), reactive-streams backpressure, and admission control inform the overflow trigger. We adopt overflow-on-bound.

- **LLM cost-control patterns** — prompt caching and semantic response caching (e.g., GPTCache-style approaches) reduce *repeat* inference. We note that these are orthogonal: they help when inputs *repeat*; the exactly-once triage invariant helps when distinct inputs *re-traverse buffers*.
- **Memoization** — caching a pure function's result keyed by its input is classical. The exactly-once triage invariant is a *persisted, item-bound* memoization that survives serialization to a durable store and crosses a tier boundary — distinguished by (a) the key being item identity, (b) the cache being co-located with the item in a durable queue rather than a side cache, and (c) being applied to an LLM-classification call in a queue-overflow setting.
- **Agent frameworks** — contemporary agent SDKs (orchestration loops, tool routers) classify inbound events but, to the authors' knowledge, do not publish a *two-tier overflow with a triage-result-migration invariant*. Their classification typically runs per event at handling time.

The synthesis: each ingredient is known; the *combination* — a bounded→unbounded overflow tier pair in which an *LLM triage result is computed once and migrated with the item across the tier boundary so that buffering is cost-free in triage* — is the contribution.

---

5. Prior-Art Delta

Prior source	What it has	What it LACKS	What THIS adds
Durable broker (Kafka/SQS/RabbitMQ)	Durable buffering, overflow, redelivery, priority	A per-item *classification* computed once and carried with the payload; LLM-triage awareness	Triage result bound to the item, migrated across the bounded↔unbounded boundary, never recomputed
In-memory priority heap / ring buffer	Bounded memory, priority ordering	Overflow to durable store; crash survival; once-only triage	A cold persistent tier with lossless overflow that preserves the triage result
DB / OS spill-to-disk (buffer pool, page cache)	Hot/cold tiering, overflow	Notion of an expensive per-item classification; memory-first *work* dequeue	An exactly-once *triage* (LLM) step amortized across the spill
LLM semantic / prompt cache	Avoids repeat inference for similar inputs	Help for bursts of *distinct* inputs; queue-transition coverage	Cost bound on *distinct* items via item-bound persistence, independent of repetition
Producer–consumer w/ backpressure	Bounded buffer, overflow trigger	Persisted classification; memory-first dequeue; triage cost bound	Overflow that *moves the classification*, not just the payload
Admission control / rate limiter	Bounds concurrency / spend over time	Reduction in per-item triage *count*; lossless absorption	Per-item triage count reduced to one regardless of arrival rate
Plain memoization	Caches a function result by key	Persistence across serialization to a durable queue; tier-crossing; LLM-classification setting	Memoization *co-located in the durable item* and migrated across the tier boundary
Agent SDK event classifier	Classifies inbound events	Two-tier overflow; once-only-across-tiers invariant; cost bound	The two-tier + exactly-once triage discipline as a named, enforced invariant

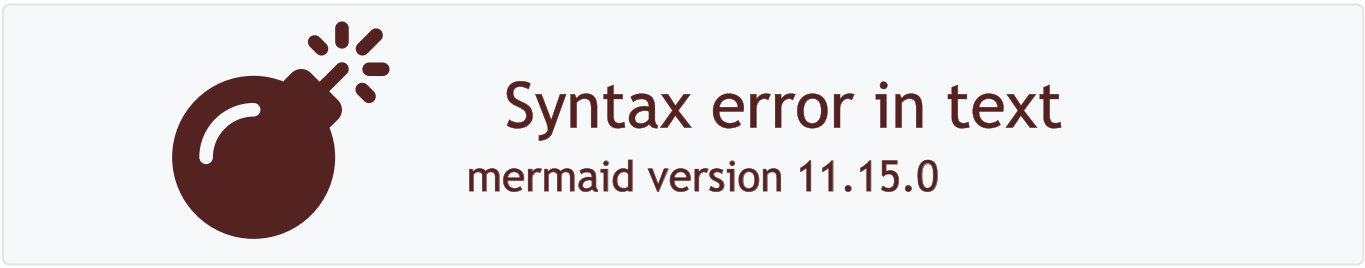
---

6. System Architecture

6.1 Components

- **Intake** — receives raw items from sources (mailbox, webhook, chat, tool callbacks). The *only* place triage runs.
- **Triage classifier** — an LLM-backed function `T(item) → {type, priority, reasoning}`, invoked exactly once per item, with a heuristic fast-path that skips the LLM for already-typed items.
- **Hot tier (primary queue)** — a bounded in-memory queue (default capacity 50) ordered by computed weight. Holds items awaiting focus.
- **Cold tier (secondary queue)** — an unbounded persistent queue backed by a durable store. Holds overflow. Each row carries the full payload *and* the persisted triage result.
- **Overflow/promotion controller** — on hot-tier saturation, moves items (with their triage result) to the cold tier; when the hot tier has free capacity, promotes the highest-priority cold items back, reusing their stored classification.
- **Dequeue scheduler** — memory-first: pops from the hot tier; only when the hot tier is empty does it pull from the cold tier (which, on pull, is itself a promotion).
- **Worker / focus engine** — consumes one item at a time, performs the actual work (which may itself call the LLM), and reports completion.
- **Config provider** — supplies capacity, default classification, weights, and thresholds from external configuration.

6.2 Architecture diagram



6.3 Cross-cutting properties

- **Cost-independence of buffering.** Moving an item hot→cold→hot incurs zero triage cost.
- **Durability boundary.** The cold tier is the crash-safe source of truth; the hot tier is a performance cache of the front of the queue.
- **Priority preservation.** Weight derived from the (once-computed) priority orders both tiers.
- **Fail-open intake.** Triage failure yields a default classification; intake never blocks on the model.

---

7. Detailed Mechanics

7.1 Overflow trigger

On intake of item i:

1. Triage i exactly once (see §7.2), producing `(type, priority)` and a derived `weight`.

2. If `|hot| < capacity`, insert i into the hot tier and re-sort by weight.
3. Else (overflow), persist i — *including its triage result* — to the cold tier.

The bound `capacity` is configurable (default 50). The cold tier has no bound other than storage.

7.2 The exactly-once triage invariant

Triage is invoked **at intake and nowhere else**. Once an item carries a non-default, non-ambiguous `type` and a `priority`, no code path re-invokes `T`. Concretely:

- **Fast-path skip.** If the item already arrives typed (e.g., a webhook with a known event type, or a tool callback), the LLM triage is skipped entirely; a heuristic assigns the classification. This handles the common case at zero LLM cost.
- **Once-only LLM call.** Only ambiguous items (free-text email/chat with `type = unknown`) invoke the LLM classifier, and only at intake.
- **Idempotence marker.** The item is stamped `_triaged = true`. Any later code that *could* triage checks this marker (and the presence of a concrete `type`) and returns immediately.

Pseudocode:

```
function intake(item):
    item = triage_once(item)              # at most one LLM call
    weight = computeWeight(item)
    if hot.size < capacity:
        hot.insert(item); hot.sortByWeight()
        return ADDED_HOT
    else:
        cold.persist(item)                # carries item.type, item.priority, item._triaged
        return OVERFLOW_COLD

function triage_once(item):
    if item._triaged or (item.type != "unknown" and item.type != null):
        return item                       # already classified → NO LLM call
    body = truncate(item.body, MAX_TRIAGE_CHARS)
    if len(body) < MIN_BODY:              # too little to classify
        item.type = item.type or DEFAULT_TYPE
        return item
    try:
        c = LLM_classify(body)            # the ONE expensive call
        item.type = c.type
        item.priority = c.priority
        item._triaged = true
        item._triageReasoning = c.reasoning
    catch:
        item.type = item.type or DEFAULT_TYPE      # fail-open
        item.priority = item.priority or DEFAULT_PRIORITY
    return item
```

7.3 Triage-result migration across tiers

The key to R3–R5 is that the **cold tier row stores the triage result alongside the payload**. Overflow serializes `{id, source, type, priority, weight, body, ts, _triaged, reasoning}` — not just the payload. Promotion deserializes the same record and inserts it into the hot tier *without calling* `triage_once` (the

record is already `_triaged = true` and concretely typed, so the idempotence guard short-circuits). The round trip hot→cold→hot is therefore triage-free.

```
function overflow_to_cold(item):
    cold.persist({ ...item.payload, type: item.type, priority: item.priority,
                   weight: item.weight, triaged: true, reasoning: item.reasoning })

function promote_from_cold():
    if hot.size >= capacity: return
    row = cold.popHighestWeight()         # ordered fetch
    if row is null: return
    hot.insert(rowToItem(row))            # rowToItem sets _triaged = true → guard skips triage
    hot.sortByWeight()
```

7.4 Memory-first dequeue

The worker asks the scheduler for the next item. The scheduler **drains the hot tier before the cold tier**:

```
function dequeue():
    if hot.size > 0:
        return hot.popHighestWeight()      # warm: same process memory
    else:
        return promote_one_and_pop()       # pull from cold, then serve
```

Serving from the hot tier exploits warmth: the item is already in process memory (no deserialization), and consecutive same-type items keep the LLM's working context (and any prompt cache) warm, reducing per-item work cost. The cold tier is consulted only when the hot tier is exhausted, at which point a promotion both refills the hot tier and serves the head.

7.5 In-tier ordering and interrupts

Within a tier, items are ordered by a numeric **weight** derived from the once-computed priority (e.g., `low=20, normal=40, high=60, urgent=80, critical=100`), with bonuses (e.g., +20 for a human sender) and FIFO as a tiebreak. Escalation of an item to `urgent` re-weights it *in place* and may trigger an interrupt of in-flight focus — **without re-triaging** (escalation changes priority, not type, and is an explicit operator/heuristic action, not an LLM classification). This satisfies R7 and R12.

7.6 Cost analysis

Let an item be inserted once and traverse the buffers t times (overflows + promotions). Under the invariant, the number of triage calls for that item is exactly 1, independent of t. Therefore total triage cost over a burst of N distinct items is

$$K = N \quad\text{(minus fast-path-skipped items)}, \qquad \text{independent of } \lambda, t, \text{ and restart count.}$$

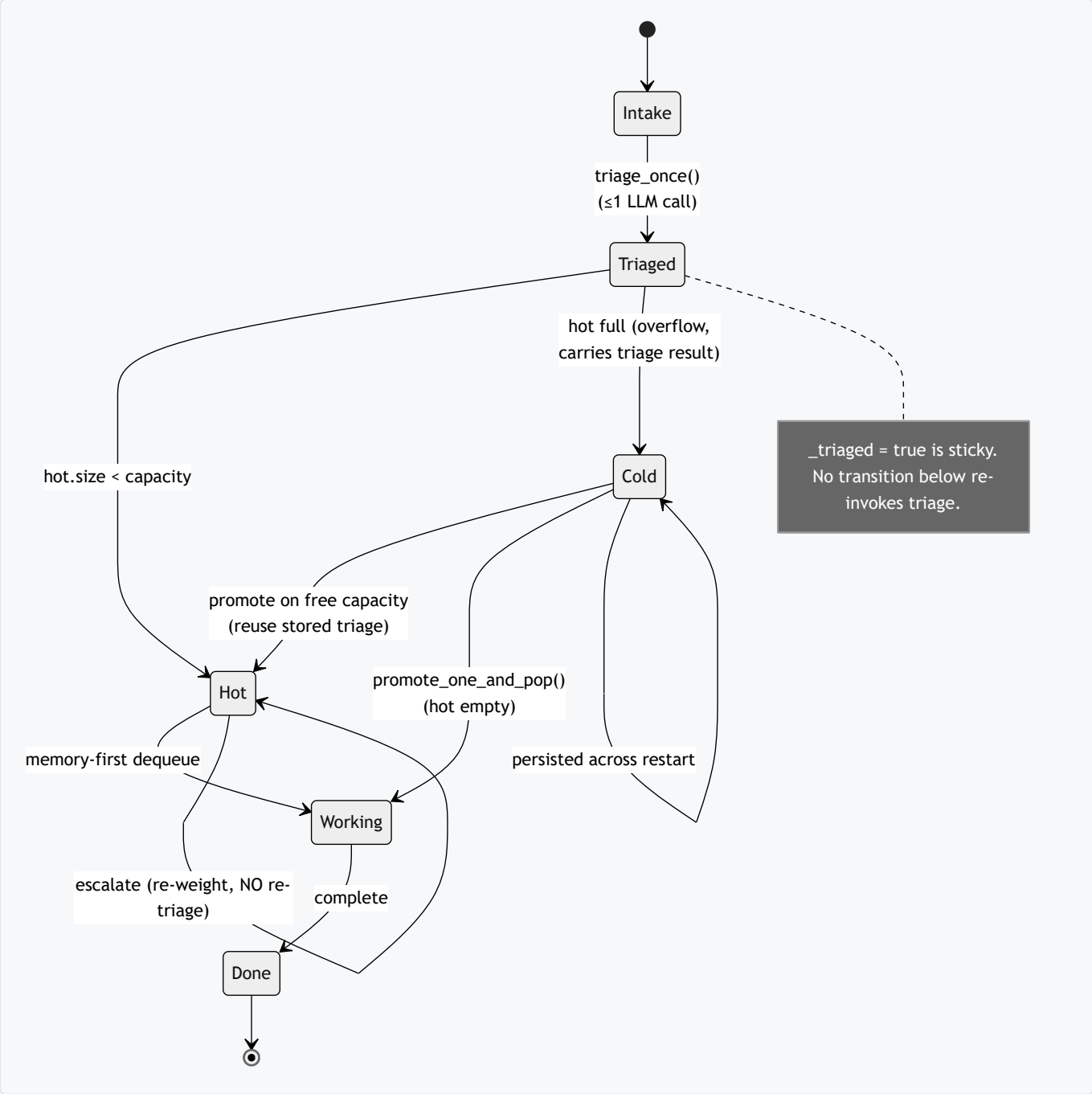
A naive triage-at-dequeue design yields $K_{\text{naive}} = \sum_i (\text{dequeues of } i) \ge N$, strictly worse whenever any item is re-read (restart, promotion, retry). See §13 for the methodology to measure this.

7.7 Configuration

Key	Meaning	Default
`capacity`	Hot-tier bound	50
`defaultType`	Fallback classification on triage failure / thin body	`task`
`defaultPriority`	Fallback priority	`normal`
`priorityWeights`	priority → numeric weight map	low 20 … critical 100
`humanBonus`	weight bonus for human-sourced items	20
`maxTriageChars`	body truncation for the classifier	500
`minBodyChars`	below this, skip LLM and use default	10

All values come from a config provider (env / config service), never hardcoded constants in business logic (R10).

7.8 State machine



8. Data Model

8.1 Hot-tier item (in-memory record)

Field	Type	Notes
`id`	string	Stable item identity (idempotency key)
`source`	string	Sender / origin
`channel`	string	mailbox / webhook / chat / voice
`type`	enum	email, chat, task, tool, agent-session, voice — set once by triage
`priority`	enum	low, normal, high, urgent, critical — set once by triage
`weight`	number	Derived from priority + bonuses
`body`	text	Payload (may be truncated for triage)
`ts`	timestamp	Intake time
`from_human`	bool	Drives weight bonus / interrupt policy
`_triaged`	bool	Idempotence marker (sticky true)
`reasoning`	text	One-line triage rationale (audit)

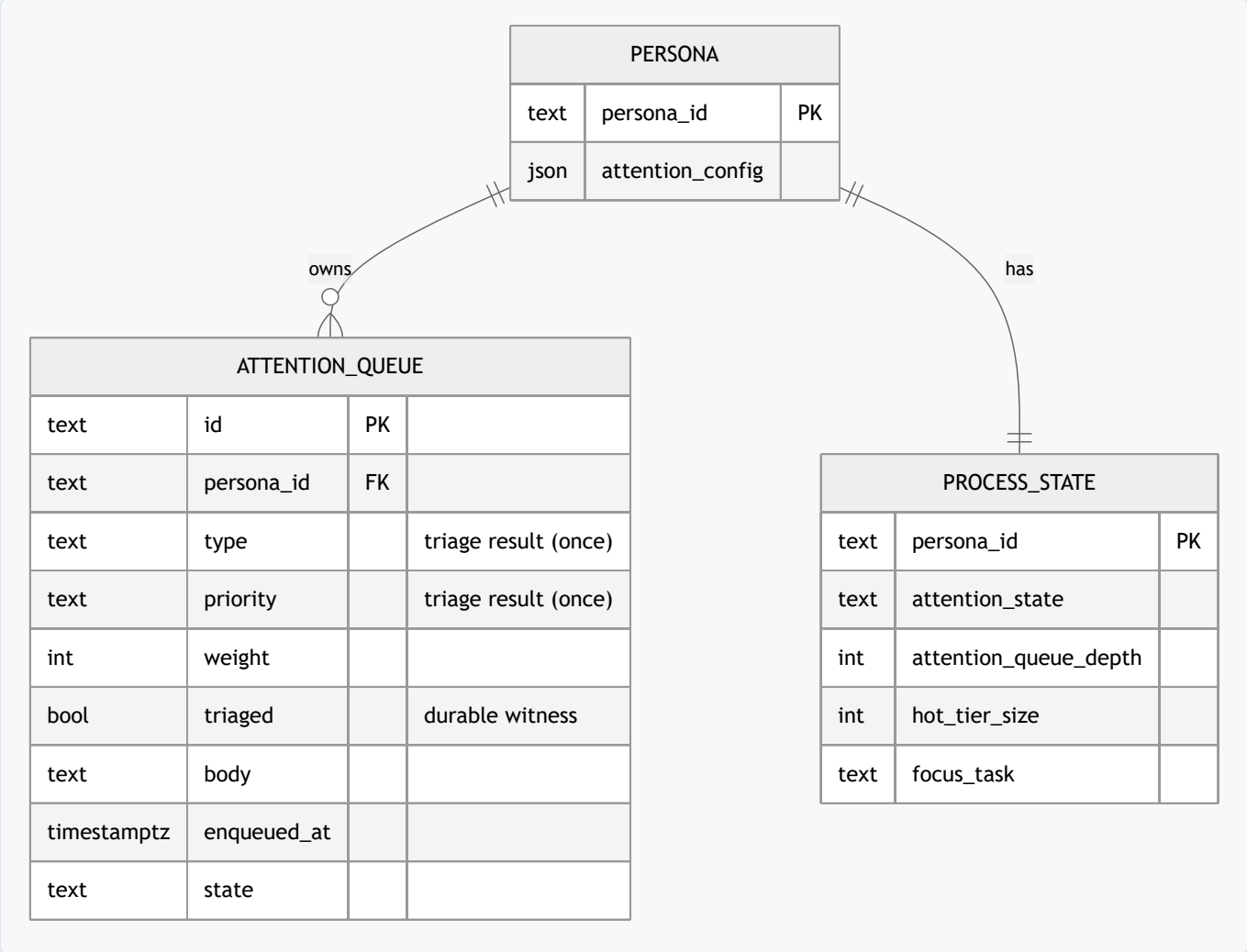
8.2 Cold-tier row (persistent)

The cold tier persists **the same fields**, so promotion is a pure deserialize with no recomputation. A minimal schema:

```
CREATE TABLE attention_queue (
    id            TEXT PRIMARY KEY,          -- idempotency key
    persona_id    TEXT NOT NULL,             -- owning agent
    source        TEXT,
    channel       TEXT,
    type          TEXT NOT NULL,             -- triage result (persisted)
    priority      TEXT NOT NULL,             -- triage result (persisted)
    weight        INTEGER NOT NULL,
    body          TEXT,
    from_human    BOOLEAN DEFAULT FALSE,
    triaged       BOOLEAN DEFAULT TRUE,       -- always true in cold tier
    reasoning     TEXT,
    enqueued_at   TIMESTAMPTZ DEFAULT now(),
    state         TEXT DEFAULT 'queued'       -- queued | promoting | done
);
CREATE INDEX idx_aq_promote ON attention_queue (persona_id, state, weight DESC, enqueued_at ASC);
```

The `triaged` column being persisted is the crux: it is the durable witness that this item's classification cost has already been paid.

8.3 ER diagram



9. Reference Implementation & Enablement

The `src/` directory contains an **original, clean-room** implementation of the two-tier attention queue, written specifically for this disclosure. It is illustrative — dependency-light, standard-library JavaScript — and is *not* production code.

What it demonstrates:

- `TwoTierAttentionQueue` (`src/two-tier-queue.js`) — the bounded hot tier, an in-memory stand-in for the persistent cold tier (a `PersistentColdTier` whose interface is the seam where a real durable store plugs in), the overflow/promotion controller, memory-first dequeue, and the **exactly-once triage invariant** with a sticky `_triaged` marker.
- A pluggable `triageFn` whose invocation count is tracked, so the cost claim (triage calls == distinct items) is *machine-checkable* rather than asserted.
- `src/example.js` — a self-checking harness that pushes a 200-item burst through a capacity-50 hot tier, drains it, restarts (re-reading the cold tier), and asserts that the triage function was invoked **exactly once per distinct item** despite many overflow/promotion transitions and a simulated restart.

How it reduces the invention to practice: running `node src/example.js` exercises every requirement R1–R12 against a concrete burst and prints the triage-invocation count, proving $K = N_{\text{distinct}}$ regardless of buffer churn.

Configuration points (R10) are constructor options: `capacity`, `defaultType`, `defaultPriority`, `priorityWeights`, `humanBonus`. The `triageFn` and the cold-tier store are injected, so a real deployment swaps the in-memory cold tier for a durable store and the stub classifier for an LLM call without touching the queue logic.

---

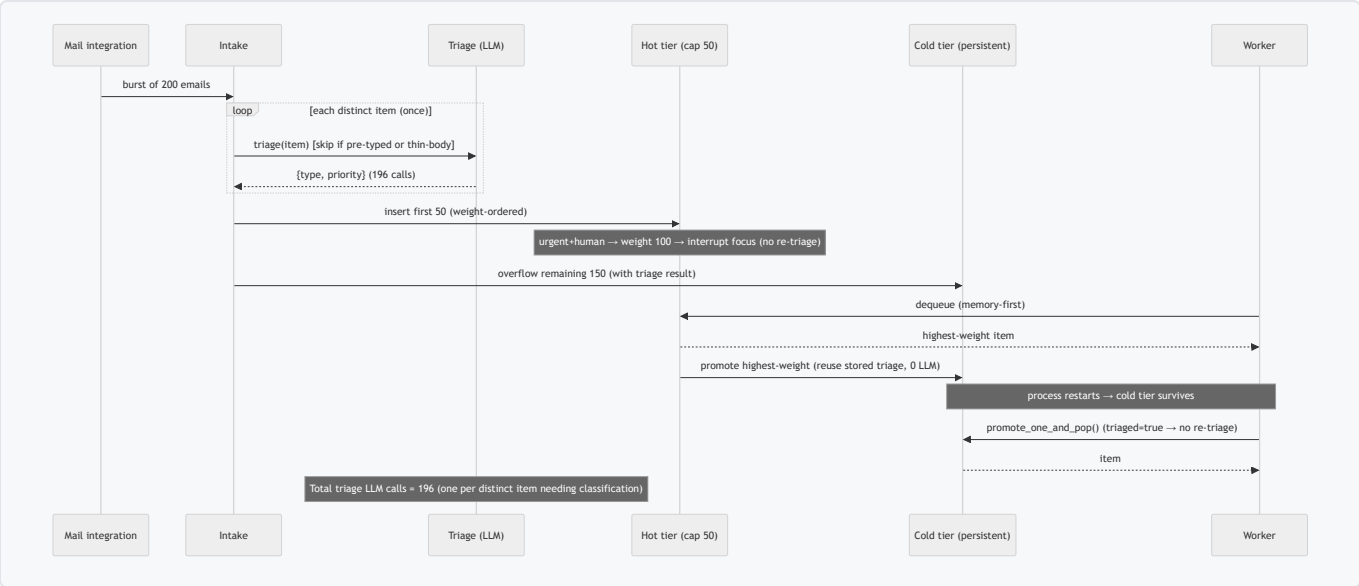
10. Worked Example / Scenario

Setup. An autonomous agent is in `focused` state working on task A. Its hot tier capacity is 50. A mail integration reconnects and delivers **200 emails** in one burst; one of them, from a human, contains the word "urgent".

Walkthrough.

1. Intake receives all 200. Each is triaged **once at most**: free-text emails → LLM classifier; pre-typed items (e.g. calendar invites) → heuristic fast-path, **no LLM call**; and any item whose body is below the minimum length → heuristic default, **no LLM call**. In the bundled reference harness the 200 items decompose as 3 pre-typed + 1 thin-body skipped, so **196** items invoke the classifier. The narrative below uses the same shape with 3 pre-typed skips; the precise count is whatever the instrumented `triageFn` reports. Triage LLM calls so far: **196** (not 200) — one per distinct item that actually needs classification.
2. The first 50 (by weight) fill the hot tier. The "urgent + human" email gets `priority = urgent`, `weight = 80 + 20 = 100`, sorts to the head, and **interrupts** the persona's focus on A — *without re-triage*.
3. The remaining 150 overflow to the cold tier, each row carrying its already-computed `type`, `priority`, `weight`, and `triaged = true`.
4. The worker drains the hot tier memory-first. As it empties, the controller **promotes** the highest-weight cold items back into the hot tier — reading their stored classification, calling the classifier **zero** more times.
5. Mid-drain the process restarts. On boot, the hot tier is empty but the cold tier holds the survivors. The scheduler promotes from cold; because each row is `triaged = true`, **no re-triage** occurs.
6. All 200 are processed. **Total triage LLM calls: 196** — one per distinct item that actually needed classification — despite 150 overflows, ~150 promotions, an interrupt, and a restart. (The runnable harness in `src/` reports exactly this count.)

A naive triage-at-dequeue design would have re-triaged on every promotion and again after the restart, yielding hundreds of extra LLM calls.



11. Security, Safety & Failure Modes

11.1 Posture

- **Intake is fail-open.** A triage failure (model error, timeout, rate limit) does **not** block intake; the item is admitted with a safe default classification (`defaultType`/`defaultPriority`) and `_triaged` is *not* set to true on a failed LLM call — so a later, deliberate re-triage is permitted *only* via an explicit re-classification path, never via ordinary buffer churn. (Implementations may choose to mark a failed item with a bounded retry budget; the invariant is that *ordinary* hot↔cold transitions never trigger triage.)
- **Cold tier is fail-safe (source of truth).** Items in the cold tier survive process crashes; the hot tier is a rebuildable cache.
- **Bounded blast radius.** Because triage is once-only, a triage outage cannot multiply into a cost storm: items simply queue with default classifications.

11.2 Failure-mode table

Failure mode	Trigger	Effect without mitigation	Mitigation (this design)
Triage LLM down	Provider outage / 429	Intake blocks or retries storm	Fail-open default classification; intake proceeds
Process crash mid-drain	Pod restart	Lost in-flight items / re-triage	Cold tier is durable & `triaged=true` → no loss, no re-triage
Hot-tier unbounded growth	$\lambda \gg \mu$	OOM	Hard capacity bound; overflow to cold
Re-triage on promotion	Naive code path	Cost storm	Sticky `_triaged` + concrete type guard short-circuits triage
Duplicate intake	Source replays an item	Double work / double triage	`id` idempotency key; existing item skipped
Priority inversion	Urgent item stuck behind low	Missed SLAs	Weight ordering across both tiers; promote-by-weight
Poison item	Body crashes classifier	Repeated failures	Body truncation + min-length guard + fail-open default

11.3 STRIDE-style notes

- **Tampering / Spoofing:** `id` and `source` are inputs; downstream authorization is out of scope for the queue but the queue must not *elevate* trust — weight bonuses (e.g., human bonus) should be set by trusted intake, not by item-supplied fields.
- **Denial of service:** a burst is the *designed* condition, not an attack surface; the capacity bound and fail-open default cap the damage. An attacker cannot force re-triage to multiply cost, because the invariant forbids it.
- **Repudiation:** the persisted `reasoning` and `enqueued_at` provide an audit trail of why and when each item was classified.

---

12. Standards & Framework Mapping

This is *semantic alignment*, not certified compliance.

Framework / standard	Relevant concept	Alignment in this design
ISO/IEC 25010 (software quality)	Performance efficiency, reliability (recoverability), resource utilization	Bounded memory, durable recovery, triage-cost bound
ISO/IEC 42001 (AI management)	Controlled, measurable AI resource use	Observable, bounded LLM-call count per burst
NIST AI RMF (MANAGE / cost & reliability)	Managing AI system resource and reliability risks	Cost-storm prevention; fail-open degradation
Reactive Manifesto	Resilience, elasticity, message-driven backpressure	Overflow-on-bound; bounded hot tier; durable spillover
Queueing theory (Little's Law)	$L = \lambda W$; stability requires $\lambda < \mu$ at steady state	Tiers decouple transient $\lambda \gg \mu$ from steady-state throughput
Twelve-Factor (config)	Strict separation of config from code	All thresholds externally configured (R10)
FinOps principles	Variable-cost visibility and control	Per-burst triage spend is bounded and observable

13. Evaluation Methodology

We describe how to evaluate the design. Any figures here are **illustrative** and flagged as such; the reference harness emits real measured counts.

13.1 Dimensions and metrics

Dimension	Metric	Interpretation
Triage cost bound (the core claim)	`triage_calls / distinct_items`	Must equal 1.0 (minus fast-path skips). >1.0 means the invariant is violated.
Burst absorption	max queue depth vs. burst size; loss count	Loss must be 0; memory bounded by capacity + cold storage
Memory bound	peak hot-tier size	Must never exceed `capacity`
Recovery	items lost / re-triaged across restart	Both must be 0
Latency to first work	time from burst start to first dequeue	Should be ~one triage, not N triages
Priority correctness	rank-order of dequeue vs. weight	Urgent items served before low
Steady-state throughput	items/s drained	Bounded by μ, independent of λ

13.2 Procedure

1. Generate a synthetic burst of N distinct items (mix of pre-typed and free-text), with a known urgent subset.
2. Run through the queue with an instrumented `triageFn` that counts invocations and records the item id.
3. Force overflow ($N >$ capacity), drain partially, simulate a restart (re-read cold tier), then drain fully.

4. Assert: `triage_calls == distinct_free_text_items`; `loss == 0`; `peak_hot <= capacity`; urgent served first.

13.3 Illustrative result (not measured production data)

Scenario	Distinct items	Overflows	Promotions	Restarts	Naive triage calls	This design's triage calls
200-email burst (cap 50)	200	150	~150	1	~500+	**196** (3 pre-typed + 1 thin-body fast-path-skipped)

All numbers in this table are illustrative for exposition; the reference harness prints the actual measured counts on each run.

---

14. Novelty & Inventive Claims

The following claims are presented as prose for clarity of disclosure (defensive publication, not an application). One independent claim and fifteen dependent claims.

Claim 1 (independent). A computer-implemented method for handling bursts of input in an agent backed by a large language model, the method comprising: maintaining a bounded in-memory primary queue and an unbounded persistent secondary queue; on intake of an item, performing exactly once a triage step that assigns to the item a type and a priority by invoking a classification function, and persisting the assigned type and priority together with the item; responsive to the primary queue reaching its bound, transferring a subsequent item to the secondary queue together with its persisted type and priority and without re-invoking the triage step; on dequeue, draining the primary queue before the secondary queue; and promoting items from the secondary queue back into the primary queue using their persisted type and priority without re-invoking the triage step — characterized in that the classification function is invoked at most once per distinct item across the primary and secondary queues, such that the number of classification invocations for a burst is bounded by the number of distinct items rather than by the number of queue transitions or the arrival rate.

Claim 2. The method of claim 1, wherein the classification function is a call to the large language model, and the bound on classification invocations is a bound on language-model calls.

Claim 3. The method of claim 1, wherein persisting the assigned type and priority comprises storing, in a durable record of the secondary queue, the item payload together with the type, the priority, and an idempotence marker indicating the item has been triaged.

Claim 4. The method of claim 3, wherein promoting an item from the secondary queue comprises reading the durable record and inserting the item into the primary queue while a triage guard, conditioned on the idempotence marker, suppresses any re-invocation of the classification function.

Claim 5. The method of claim 1, further comprising computing, from the assigned priority, a numeric weight, and ordering items within each queue by the weight, such that higher-priority items are dequeued first across both queues.

Claim 6. The method of claim 1, wherein the triage step is skipped, without invoking the classification function, for items that arrive already bearing a concrete type, a heuristic instead assigning their

classification.

Claim 7. The method of claim 1, wherein the triage step is skipped, without invoking the classification function, for items whose body is shorter than a configured minimum length, a configured default type and priority instead being assigned.

Claim 8. The method of claim 1, wherein a failure of the classification function during the triage step causes the item to be admitted with a configured default type and priority and without blocking intake, the method thereby being fail-open with respect to the classification function.

Claim 9. The method of claim 1, wherein draining the primary queue before the secondary queue exploits warmth of in-process memory and of a model context or prompt cache shared across consecutively processed items.

Claim 10. The method of claim 1, wherein the secondary queue is the source of truth across a process restart, the primary queue being rebuilt by promoting items from the secondary queue using their persisted type and priority without re-invoking the triage step.

Claim 11. The method of claim 1, further comprising escalating the priority of a queued item in place, re-computing its weight, and optionally interrupting an in-flight focus, all without re-invoking the triage step.

Claim 12. The method of claim 1, wherein the bound on the primary queue, a default type, a default priority, a mapping from priority to weight, and a body-truncation length are each obtained from external configuration rather than hardcoded.

Claim 13. The method of claim 1, further comprising assigning each item a stable idempotency identifier and suppressing intake of a duplicate item bearing an identifier already present in either queue, thereby preventing duplicate triage and duplicate work.

Claim 14. The method of claim 1, wherein each item processed by the worker may itself invoke the large language model, and the bound established by the triage invariant is a bound on the classification calls specifically, separate from work-stage model calls.

Claim 15. The method of claim 1, further comprising recording, for each item, a triage rationale and an enqueue timestamp in the durable record, providing an audit trail of why and when the item was classified.

Claim 16. A non-transitory computer-readable medium storing instructions that, when executed, cause a system to perform the method of any of claims 1–15, the system comprising the bounded in-memory primary queue, the unbounded persistent secondary queue, the triage classifier invoked at most once per distinct item, and the memory-first dequeue scheduler.

---

15. Limitations & Threats to Validity

- **Two-level buffering is not novel.** The hot/cold structure is classical; the contribution is the exactly-once *triage* invariant migrating with the item in the LLM setting. A reviewer must read the claims as narrowing on that invariant, not on tiering per se.
- **Producer–consumer overflow prior art is close.** Examiners may cite bounded-buffer overflow patterns; the distinguishing limitation is the persisted, item-bound, tier-crossing

classification result and its enforcement that buffer churn never re-triages.
- **Triage correctness is out of scope.** This work bounds the *cost* of triage, not its *accuracy*. A wrong-but-once classification is still wrong; re-classification is an explicit, separate path.
- **Cold-tier ordering cost.** Promote-by-weight requires an ordered fetch from the persistent store; very large cold tiers shift cost to the index. This is a standard durable-queue concern, intentionally delegated to the store.
- **Single-consumer assumption in the reference.** The illustrative implementation assumes one worker per persona; multi-consumer fan-out with the same invariant is straightforward but not exercised here.
- **Withheld detail.** Production weighting heuristics, interrupt policies, and integration with a specific persistence backend are intentionally generalized; the disclosure teaches the mechanism, not a particular deployment.

---

16. Future Work & Open-Source Reference App

The planned open-source reference app (see `docs/OPEN-SOURCE-APP.md`) is a small, self-contained "burst-resilient agent inbox" service: it accepts items over HTTP, triages each once, buffers across the two tiers backed by a real durable store, and exposes the triage-invocation counter as a Prometheus metric so operators can *see* the cost bound hold. Roadmap:

1. Pluggable cold-tier adapters (Postgres, SQLite, Redis Streams).
2. A multi-consumer variant preserving the once-only invariant via row-level promotion locks.
3. A dashboard visualizing hot/cold depths and the `triage_calls / distinct_items` ratio over time.
4. A Helm chart for generic Kubernetes/AKS deployment (no internal cluster identifiers).

---

17. Conclusion

Buffering a burst is easy; buffering it *cheaply* is the problem when classification is an LLM call. By making triage a once-computed, item-bound, durably-persisted property that migrates with the item across a bounded→unbounded tier boundary, and by draining memory-first, the two-tier attention queue absorbs arbitrary input bursts while bounding triage cost to the number of distinct items — independent of arrival rate, buffer churn, or restarts. We publish this defensively so the technique remains free for all to practice.

Appendix A — Prior-Art Landscape

Well-trodden (not claimed as novel):

- Hierarchical buffering / spill-to-disk (CPU caches, DB buffer pools, OS page cache).
- Durable message brokers with overflow, redelivery, and priority (Kafka, SQS, RabbitMQ, POSIX/SysV mqueues).
- Priority queues, weighted fair queueing, multilevel feedback queues.
- Bounded-buffer producer/consumer and reactive backpressure.
- Memoization and semantic/prompt caching for LLMs.

Candidate-novel (claimed):

- The exactly-once triage invariant: an LLM classification computed once at intake and *persisted with the item* such that it migrates across the bounded↔unbounded tier boundary and survives restarts, so that no buffer transition ever re-triages.
- The resulting cost bound: classification invocations == distinct items, independent of arrival rate and queue churn.
- Memory-first dequeue coupled to the above to exploit context/cache warmth in the work stage.

Honesty attestation. The prior-art searches behind this document are *directional, not exhaustive*. They reflect the authors' good-faith knowledge of queueing systems, message brokers, and LLM cost-control patterns as of the publication date. No claim is made that every relevant reference has been found. This document's purpose is defensive: to place the described combination in the public record as of 2026-06-25.

Appendix B — Glossary

Term	Definition
Triage	Classifying an input into a `type` and `priority`; here, an LLM call for ambiguous items.
Hot tier / primary queue	Bounded in-memory queue holding the front of the work line.
Cold tier / secondary queue	Unbounded persistent queue holding overflow; crash-safe source of truth.
Overflow	Moving an item from the saturated hot tier to the cold tier, carrying its triage result.
Promotion	Moving a cold-tier item back into the hot tier, reusing its stored triage result.
Exactly-once triage invariant	The rule that the classification function runs at most once per distinct item, across all tiers and restarts.
Memory-first dequeue	Draining the hot tier before the cold tier.
Weight	Numeric ordering key derived from priority plus bonuses.
`_triaged` marker	Sticky idempotence flag that short-circuits re-triage.
Thundering herd	A burst of inputs each independently triggering an expensive operation.

Appendix C — Reference-Implementation Index

File	Role
`src/two-tier-queue.js`	The `TwoTierAttentionQueue`, `PersistentColdTier` (durable-store seam), overflow/promotion controller, memory-first dequeue, exactly-once triage guard.
`src/example.js`	Self-checking harness: 200-item burst, overflow, partial drain, simulated restart, full drain; asserts `triage_calls == distinct_items`.
`src/README.md`	What the sample shows, how to run it, and the clean-room / illustrative disclaimer.

Appendix D — Defensive-Publication Deposit & Timestamp

- **Publication date:** 2026-06-25
- **Publisher / copyright holder:** Gus IT LLC (Florida, USA)
- **Author:** Gustavo Assuncao, PhD
- **Version:** 1.0
- **License:** AGPL-3.0-or-later (copyleft; commercial license available)
- **Deposit channel:** To be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is claimed at the time of writing.

This disclosure is intentionally made public to establish prior art as of the publication date and to **bar later patenting of the disclosed technique by others**. The technique is contributed to the public domain of practice under the accompanying AGPL-3.0-or-later license; the express patent grant ensures downstream users receive a patent license from the contributor and discourages patent assertion.