

Volume 1, Issue 1

Research Article

Date of Submission: 28 Aug, 2026

Date of Acceptance: 29 Sep, 2026

Date of Publication: 09 Oct, 2026

Reducing Tail Latency in LLM-Backed Microservices Through Asynchronous Decomposition and Request Hedging

Chandrashekhar Singh* 

AI Engineer, Independent Researcher, Toronto, Canada

***Corresponding Author:** Chandrashekhar Singh, AI Engineer, Independent Researcher, Toronto, Canada.

Citation: Singh, C. (2026). Reducing Tail Latency in LLM-Backed Microservices Through Asynchronous Decomposition and Request Hedging. *J Digi Health Rec Med AI Info Sys*, 1(1), 01-12.

Abstract

Large language model inference services can exhibit substantial latency variability, particularly for long-context workloads. In fan-out applications, the completion time of an application-level request is determined by the slowest downstream call. Consequently, a small number of delayed inference calls can dominate end-to-end response time.

This paper presents an anonymized production-motivated case study of a document-processing microservice that invokes a managed, OpenAI-compatible inference endpoint. We evaluate three designs: a synchronous monolithic request, asynchronous decomposition into independently processed batches, and asynchronous batching combined with delayed request hedging. In the hedged design, a duplicate call is launched only when an original batch exceeds an empirically selected latency threshold, and the first successfully completed response is retained.

Across 100 production-shaped replay requests, asynchronous decomposition reduced typical latency by processing independent batches concurrently, while request hedging reduced the remaining slow tail. In the observed workload, P95 end-to-end latency decreased from approximately 60 seconds to approximately 13 seconds. The improvement was achieved without changing the model family, prompt semantics, or inference infrastructure, although it introduced approximately 10% additional inference-token consumption under the measured latency distribution.

The paper contributes an implementation pattern, an empirical procedure for selecting a hedge threshold, and operational safeguards covering idempotency, cancellation, bounded concurrency, and cost containment. The technique is applicable to read-only, latency-sensitive distributed operations when duplicate execution is safe and the latencies of duplicate calls are sufficiently independent.

Keywords: Tail Latency, Request Hedging, Asynchronous Processing, Large Language Models, Microservices, Distributed Systems, Latency Optimization

Introduction

Modern software systems increasingly depend on externally managed large language model (LLM) inference APIs. These APIs provide convenient access to capable models, but application owners typically cannot control replica placement, accelerator scheduling, model-server autoscaling, or admission decisions. The application nevertheless remains responsible for meeting user-facing latency objectives.

Latency from a hosted inference service is often right-skewed. Most calls may complete within a predictable interval, while a smaller fraction experiences substantially longer delay. Possible causes include queueing, accelerator contention, model-server cold starts, long-context prefill, network variation, autoscaling events, or routing to an overloaded backend replica.

The problem becomes more severe when an application request depends on several downstream calls. Let L_i represent the latency of the i -th downstream inference call. When all results are required before the application can respond, the

end-to-end downstream latency is approximately

$$L_{\text{fanout}} = \max L_i, \quad (1)$$

$$1 \leq i \leq n$$

Even when $n - 1$ calls complete quickly, one delayed call can determine the response time of the complete request. This behaviour is commonly called tail amplification.

This paper examines a document-processing microservice that accepts up to 100 textual items and produces a structured aggregate summary. The initial implementation submitted the complete corpus through one large synchronous inference call.

Two successive changes were evaluated:

- decomposing the input into small batches and dispatching those batches concurrently; and
- applying delayed request hedging independently to batches that exceed a measured latency threshold.

The final design reduced observed P95 latency from approximately 60 seconds to approximately 13 seconds in the evaluation workload.

Contributions

This work does not claim asynchronous execution or request hedging as new distributed-systems primitives. Instead, it studies their combined application to long-context LLM microservices whose inference infrastructure is externally managed.

The Paper Makes the Following Contributions

Production-Motivated Case Study: We characterize how long-context prefill and backend latency variance affect an application that aggregates multiple LLM results.

Combined Latency-Control Design: We present a design that decomposes a large inference workload into bounded concurrent batches and applies delayed hedging independently to slow batches.

Empirical Hedge Selection: We describe how to select and periodically retune the hedge threshold observed latency percentiles, a service-level objective, and duplicate-call cost.

Operational Safeguards: We identify requirements covering idempotency, bounded concurrency, cancellation propagation, timeout budgeting, and hedge-rate monitoring.

Latency-Cost Evaluation: We compare synchronous, asynchronous, and hedged implementations using equivalent production-shaped workloads.

All organization names, customer information, internal endpoint names, proprietary schemas, and domain-specific business rules have been removed. Only technical details needed to explain and evaluate the latency optimization are retained.

Background

Tail Latency in Distributed Services

Distributed services commonly route requests across multiple replicas. Although load balancers attempt to distribute traffic evenly, individual replicas may temporarily become slower because of queueing, resource contention, garbage collection, cache misses, network congestion, or autoscaling activity.

The resulting latency distribution may contain a pronounced slow tail. When an application performs fan-out and waits for every branch, the probability that at least one call is slow increases with the number of branches.

If each call independently has probability p of exceeding a latency threshold, the probability that at least one of n calls exceeds that threshold is

$$P(\text{one or more slow}) = 1 - (1 - p)^n. \quad (2)$$

For example, with $p = 0.05$ and $n = 25$, the probability that at least one branch is slow is approximately 72%. Independence is an idealization, but the equation demonstrates why fan-out can amplify moderately rare latency events.

Long-Context LLM Inference

LLM request latency generally contains at least two major components:

Prefill Latency: Processing input tokens before the first output token is generated.

Generation Latency: Producing output tokens after prefill. A monolithic request containing tens of thousands of input tokens can incur substantial prefill delay. Decomposing the input into smaller independent prompts reduces the input size of each call. When these calls execute concurrently, their latencies overlap rather than add sequentially.

Batch decomposition does not eliminate prefill. It reduces per-call prefill work and changes the end-to-end critical path from one large request to the slowest of several smaller requests.

Request Hedging

Request hedging is a delayed redundancy technique. A primary request is issued normally. If it remains incomplete after a threshold T_{hedge} , a duplicate is issued. The first successful response wins, and the slower operation is cancelled when possible.

Hedging differs from an immediate retry:

- the original request remains active;
- the duplicate is issued only after a delay;
- the two operations race; and
- the first acceptable result is retained.

The technique improves tail latency when a duplicate request has a reasonable probability of avoiding the condition that delayed the primary request. With managed inference services, the application may not know whether two calls are routed to different physical replicas. Therefore, latency independence is an empirical property rather than a guaranteed infrastructure feature.

System Context

The evaluated system is a Python 3.12 microservice implemented using an asynchronous web framework. It exposes a document-processing endpoint that accepts a corpus containing up to 100 textual items. Individual items contain on the order of hundreds of tokens.

The service performs three logical operations:

- validate and normalize the incoming items;
- submit one or more prompts to a managed LLM inference API; and
- combine batch-level results into one structured response.

The inference dependency exposes an OpenAI-compatible HTTP interface. The application does not control the model-serving infrastructure, accelerator allocation, replica routing, or autoscaling policy.

The API-layer target was a P95 latency of approximately 20 seconds. This target is an application objective used for the case study and does not identify any organization-specific service commitment.

Design

V0: Synchronous Monolithic Inference

The initial implementation submitted the complete corpus through one synchronous HTTP request. Depending on the number and size of input items, the prompt contained approximately 60,000–80,000 input tokens.

The design had two major limitations.

Large Prefill Cost: The model had to process the complete input before producing the first output token. Long-context prefill accounted for a substantial portion of the observed latency.

Blocking I/O: The application waited for the inference response in a blocking request path. Backend latency variance therefore appeared directly as application latency.

In the observed workload, V0 had a P50 near 50 seconds and a P95 near 60 seconds.

Asynchronous Batch Decomposition

The second implementation partitioned the corpus into batches of four items. For a 100-item request, this produced approximately 25 batch-level inference calls.

The calls were dispatched concurrently. Conceptually,

Algorithm 1 Delayed request hedging

Require: Asynchronous operation f , hedge threshold T_{hedge}

- 1: $p \leftarrow \text{STARTTASK}(f)$
- 2: $r \leftarrow \text{WAITUNTIL}(p, T_{\text{hedge}})$
- 3: if p completed successfully then
- 4: return result of p
- 5: end if
- 6: $s \leftarrow \text{STARTTASK}(f)$
- 7: $w \leftarrow \text{FIRSTSUCCESSFUL}(p, s)$
- 8: cancel the non-winning task
- 9: await cancellation cleanup
- 10: return result of w

$$R = \text{gather} (f (B_1), f (B_2), \dots, f (B_m)) \quad (3)$$

where B_j is a batch, f is the inference operation, and m is the number of batches.

Because batches execute concurrently, end-to-end inference latency is approximately the latency of the slowest batch rather than the sum of all batch latencies:

$$LV \approx \max$$

$$1 \leq j \leq m$$

$$L(B_j). \quad (4)$$

Batch size four was selected empirically for the studied workload. Larger batches increased per-call prefill latency, while very small batches increased request count and scheduling overhead.

Asynchronous decomposition reduced typical response time, but P95 remained influenced by the slowest concurrently executing batch.

V2: Asynchronous Batching with Delayed Hedging

The final design wrapped each batch-level inference operation in a hedging primitive.

For every batch:

- issue the primary inference call;
- wait up to T_{hedge} ;
- if the primary finishes, return it;
- otherwise issue a duplicate;
- return the first successful result; and
- cancel and drain the losing task.

Algorithm 1 summarizes the policy.

The studied configuration used a hedge threshold of approximately 10 seconds. This value was selected from the observed batch-level latency distribution and the 20-second application objective.

Concurrency and Timeout control

Bounded Concurrency

Asynchronous execution does not imply unlimited parallelism. The client uses a shared semaphore to limit simultaneously active inference calls.

Let B be the number of batches and C be the concurrency limit. At most

$$\min (B, C) \quad (5)$$

primary batch calls may execute concurrently. Hedged duplicates consume capacity from the same concurrency budget.

This prevents the optimization from producing unbounded traffic during provider degradation. The concurrency limit should be chosen using:

- provider requests-per-minute limits;
- provider tokens-per-minute limits;
- HTTP connection-pool capacity;
- application worker capacity;
- expected concurrent users; and
- acceptable duplicate-call overhead.

Layered Timeouts

The implementation separates three timeout concepts:

Hedge threshold: when to launch a duplicate.

Per-call timeout: maximum duration of an individual inference call.

Application deadline: maximum end-to-end request time. The hedge threshold is not the final timeout. It must leave enough time for a healthy duplicate to complete before the application deadline.

Selecting the Hedge Threshold

The hedge threshold is the most important hedging parameter. If T_{hedge} is too small, duplicates are launched for healthy requests, increasing traffic and cost without substantial latency gain. If T_{hedge} is too large, the duplicate is started too late to

meet the application objective.

Initial Threshold

Let L be the batch-level latency distribution. A useful initial condition is

$$T_{\text{hedge}} > P50(L), \quad (6)$$

so fewer than half of healthy calls are duplicated.

A practical initial value is near the observed P75:

$$T(0)_{\text{hedge}} = P75(L). \quad (7)$$

This is not a universal optimum. It is a starting point for evaluation.

SLA-Aware Constraint

Let S be the end-to-end latency objective, and let $E[L_{\text{healthy}}]$ be the expected latency of a healthy duplicate. A useful condition is

$$T_{\text{hedge}} + E[L_{\text{healthy}}] + \delta < S, \quad (8)$$

where δ represents aggregation, serialization, and network headroom.

For the studied workload, the application objective was approximately 20 seconds, healthy small-batch latency was approximately 2–5 seconds, and the selected hedge threshold was approximately 10 seconds.

Threshold Sensitivity

A publishable evaluation should compare several thresholds, for example:

$$\{\text{disabled}, P50, P60, P75, P90\}. \quad (9)$$

For each threshold, the system should report:

- P50, P95, and P99 latency;
- hedge activation rate;
- calls per application request;
- token-cost multiplier;
- timeout rate; and
- rate-limit or overload errors.

Safety Constraints

Hedging should be enabled only when the following conditions hold.

Idempotency and Side Effects

The hedged operation must be safe to execute more than once. Read-only LLM inference is application-level idempotent in the sense that it does not mutate external state, although model outputs may not be identical.

Hedging is unsafe without additional protection for:

- database inserts or updates;
- payment or authorization operations;
- email or push-notification dispatch;
- webhook fan-out;
- resource creation;
- job submission; and
- any other externally visible side effect.

Such operations require receiver-side idempotency keys or deduplication. The receiver, rather than the hedging client alone, must guarantee that duplicate execution cannot produce duplicate effects.

Nondeterministic LLM Output

Two executions of the same prompt may produce different text. Returning the first completed result is acceptable only when either result satisfies the application contract.

Applications requiring reproducibility should control, where available:

- model version;
- system prompt and template version;

- temperature;
- random seed;
- output schema; and
- decoding parameters.

Cancellation

When one request wins, the losing task should be cancelled and awaited to avoid local task leakage.

Client-side cancellation does not guarantee that provider-side inference stops immediately. For streaming endpoints, a provider may stop token generation after detecting the disconnected client. For non-streaming endpoints, inference may continue even though the client no longer waits for the response.

Therefore, cancellation provides a local resource benefit, but cost analysis should use provider-reported token usage rather than assuming that losing requests are free.

Overload Protection

Hedging can worsen an overloaded backend because delayed responses cause more duplicates, which create additional traffic.

Production deployment should therefore include:

- a global concurrency limit;
- a maximum hedge rate;
- a retry or duplicate budget;
- circuit breaking;
- rate-limit awareness;
- jittered backoff for ordinary retries; and
- automatic hedge disablement during severe incidents.

Experimental Methodology

Workload

The evaluation used 100 anonymized, production-shaped replay requests derived from the observed request-size distribution of the application.

Each request contained up to 100 textual items. Customer identifiers, organization names, account data, domain-specific terminology, internal field names, and proprietary values were removed or replaced.

All three versions received equivalent logical input and produced the same response schema:

- V0: one synchronous monolithic inference call;
- V1: asynchronous fixed-size batches; and
- V2: asynchronous batches with delayed hedging.

The model family, prompt semantics, generation settings, output schema, and client environment were held constant across versions.

Measurement Boundary

End-to-end latency was measured from API request admission until the complete application response was ready to return. This measurement included:

- request validation;
- prompt construction;
- inference calls;
- waiting and hedging;
- result aggregation; and
- response serialization.

Reported Metrics

The evaluation reports:

- mean latency;
- P50, P75, P90, P95, and P99 latency;
- maximum observed latency;
- timeout and error rates;
- hedge activation rate;
- average inference calls per application request;

Metric	V0 Sync	V1 Async	V2 Hedged
P50	50 s	30 s	12 s
P75	55 s	33 s	12 s
P95	60 s	38 s	13 s
Maximum	65 s	45 s	15 s

Table 1: End-to-End Latency Across 100 Production-Shaped Requests

- input and output token consumption;
- token-cost multiplier; and
- losing-request cancellation outcome.

Because the study contains only 100 requests, its extreme-tail estimates are descriptive rather than statistically precise. In a sample of 100, P99 is effectively determined by approximately one observation. A larger sample is required for strong P99 or three-nines reliability claims.

Experimental Controls

To reduce experimental bias, future expanded experiments should:

- execute versions in randomized or interleaved order;
- use identical logical inputs;
- run across multiple time periods;
- record provider throttling events;
- repeat each configuration;
- report confidence intervals; and
- separate warm-up requests from measured requests.

Results

Table 1 reports latency from the 100-request case-study sample.

The observed P95 decreased from approximately 60 seconds in V0 to approximately 13 seconds in V2. This corresponds to an approximately

60
 ≈ 4.6 (10)
 13

fold reduction in P95 latency.

The observed median decreased from approximately 50 seconds to approximately 12 seconds:

50
 ≈ 4.2 . (11)
 12

These values describe the measured case-study workload and should not be interpreted as universal guarantees.

Operational Metrics

Table 2 lists additional measurements needed for a complete evaluation. Replace every TODO value with data from experiment logs before submission.

Metric	V0	V1	V2
Timeout rate	TODO	TODO	TODO
Error rate	TODO	TODO	TODO
Hedge activation rate	0%	0%	TODO
Calls per application request	1	TODO	TODO
Input-token multiplier	1.00	TODO	TODO
Output-token multiplier	1.00	TODO	TODO
Total cost multiplier	1.00	~ 1.00	~ 1.10
Cancelled losing calls	N/A	N/A	TODO
Rate-limit errors	TODO	TODO	TODO

Table 2: Operational and Cost Measurements

Cost Model

Let h denote the fraction of batch calls that activate a hedge. Let r denote the average billed fraction of a duplicate relative to a complete batch call. An approximate cost multiplier is

$$M_{\text{cost}} \approx 1 + hr. \quad (12)$$

This approximation should be replaced with measured provider usage metadata whenever available.

In the observed workload, the measured or estimated token-spend multiplier for V2 was approximately 1.10×. The value is sensitive to backend conditions. If the slow-call probability increases, more calls cross Thedge, increasing both hedge rate and cost.

Output Quality and Semantic Preservation

Latency reduction is meaningful only when the optimized implementation continues to satisfy the original task.

Batch decomposition can change output semantics because each batch has less global context than a monolithic prompt. A complete evaluation should therefore measure both latency and summary quality.

The studied pipeline should explicitly document whether it:

- concatenates batch summaries directly;
- performs a second-stage aggregation;
- removes duplicate information;
- preserves source ordering;
- handles cross-batch relationships; and
- validates the final structured schema. Recommended quality metrics include:
 - schema-valid response rate;
 - factual consistency against source text;
 - coverage of required information;
 - duplicate-content rate;
 - human reviewer preference;
 - semantic similarity to the baseline response; and
 - task-specific field accuracy.

Todo

Insert the actual aggregation strategy and measured quality results. Do not claim equivalent quality without an evaluation.

Discussion

Why Asynchronous Execution and Hedging Complement Each Other

Asynchronous decomposition and hedging address different latency components.

Asynchronous decomposition:

- reduces the context size of individual calls;
- overlaps independent network and inference operations; and
- lowers typical latency.

However, because the application waits for every batch, the slowest branch can still dominate P95 and P99 latency.

Hedging:

- targets unusually slow calls;
- leaves healthy calls unchanged;
- reduces the probability that one delayed batch controls the complete response; and
- introduces additional traffic and cost.

The combination improves both the typical path and the slow tail. Asynchronous decomposition tightens the central part of the latency distribution, while hedging clips delayed branches.

Why Not Immediate Duplication?

Launching two copies of every call would provide redundancy but would nearly double traffic and inference cost. Delayed hedging preserves the single-call path for requests that complete before Thedge.

Why Not Caching?

Caching is complementary but was not the primary optimization studied here. Requests may contain newly uploaded or user-specific text, which reduces the frequency of exact cache hits. Semantic caching also introduces correctness and privacy considerations.

The presented technique targets cold, latency-bound requests that still require model inference.

Why Not Prompt Compression Alone?

Prompt compression, retrieval filtering, and context reduction may also reduce latency and cost. They can, however, alter information coverage.

This study focuses on a case where the application must process the complete logical corpus and the optimization should preserve the required input content. Prompt optimization remains complementary.

Generalization Beyond LLM Inference

The technique may apply when:

- the downstream operation has a meaningful slow tail;
- duplicate execution is safe;
- cancellation is supported or duplicate cost is acceptable;
- the duplicate has some probability of avoiding the original delay; and
- the application has sufficient capacity for controlled redundancy.

Potential examples include:

- read-only database queries against replicated read pools;
- object-storage reads;
- search-index queries across replicas;
- read-only calls to distributed caches;
- DNS resolution through multiple resolvers; and
- selected read-only service-to-service requests.

Write paths require receiver-side idempotency before hedging is safe.

Limitations

The study has several limitations.

Small Evaluation Sample: Only 100 production-shaped requests were measured. This is sufficient for a preliminary case study but not for stable extreme-percentile estimation.

Managed Backend Opacity: The client could not observe the physical replica, queue state, accelerator, or routing decision used by the inference provider.

Potential Temporal Variation: Provider latency may drift over time. A hedge threshold selected from historical measurements must be periodically retuned.

Cancellation Uncertainty: Client cancellation may not terminate provider-side computation or billing.

Increased Traffic: Hedging consumes additional requests and tokens and can amplify load during provider incidents.

Quality Risk from Decomposition: Smaller prompts may lose global context or cross-batch relationships. A quality evaluation is required.

Single Application Pattern: Results from document aggregation may not directly transfer to conversational, agentic, multimodal, or tool-calling workloads.

No Server-Side Changes: Hedging reduces client-observed variance but does not repair or improve the underlying inference platform.

Future Work

Future work should include:

- experiments with at least 1,000–10,000 requests;
- confidence intervals obtained through bootstrapping;
- adaptive hedge thresholds based on rolling latency distributions;
- load testing with concurrent application users;
- token-aware concurrency control;
- comparison with retries and immediate duplication;
- dynamic batch sizing;
- quality evaluation across batch sizes;
- automatic hedge disabling during provider degradation;
- comparison across multiple inference providers; and
- analysis of streaming time-to-first-token behaviour.

An adaptive controller could periodically compute a new threshold:

$$T_{\text{hedge}}(t) = \text{clip}(P75(L_{t-w:t}), T_{\text{min}}, T_{\text{max}}), \quad (13)$$

where $L_{t-w:t}$ is a rolling latency window and the clipping bounds protect against abrupt threshold changes.

Related Work

Dean and Barroso described the importance of tail latency in large-scale online services and discussed techniques such as hedged requests for reducing user-visible delay [1].

Vulimiri et al. analysed the use of redundancy to reduce latency and studied the conditions under which duplicate requests improve response time [2].

Suresh et al. proposed adaptive replica selection for reducing tail latency in cloud data stores [3].

Modern LLM-serving systems use techniques such as continuous batching, memory-efficient attention, scheduling, and prefix reuse to improve accelerator utilization and throughput [4]. Those approaches primarily operate within the model-serving layer. In contrast, the present work considers client-side latency mitigation when the application cannot modify the externally managed inference backend.

Python's `asyncio` runtime provides the task, waiting, timeout, and cancellation primitives used in the reference implementation [5].

Conclusion

This paper presented an anonymized case study of reducing tail latency in an LLM-backed document-processing microservice.

The initial implementation submitted a large corpus through one synchronous inference request. The optimized implementation first decomposed the corpus into small batches and dispatched them through bounded asynchronous execution. It then applied delayed request hedging to batches that exceeded an empirically selected threshold.

Across 100 production-shaped replay requests, the measured P95 latency decreased from approximately 60 seconds to approximately 13 seconds. The optimized design did not require modification of the model-serving infrastructure, but introduced approximately 10% additional token consumption under the observed latency distribution.

The main engineering lesson is that typical-case acceleration alone does not eliminate tail latency in fan-out systems. Tail events require explicit treatment as probabilistic occurrences. Delayed hedging can reduce their impact, but it must be combined with idempotency, bounded concurrency, deadline management, cancellation handling, cost monitoring, and periodic threshold retuning.

Author Contributions

The author designed the system, implemented the asynchronous and hedged execution strategies, constructed the anonymized evaluation workload, conducted the experiments, analysed the results, and wrote the manuscript.

Data and Code Availability

The production-derived data used in this study cannot be released because it may contain confidential or proprietary information. The evaluation uses anonymized, production shaped inputs.

A standalone reference implementation containing no organizational code, credentials, proprietary schemas, or customer information may be released subject to applicable organizational and publication approvals.

Ethics and Confidentiality Statement

The manuscript intentionally excludes organization names, client names, user identifiers, internal service names, deployment identifiers, credentials, proprietary document contents, and business-specific rules. Results are reported only in aggregate.

Appendix A Reference Implementation

Listing 1 provides a generic reference implementation. It is independently written pseudocode-style Python and should not be replaced with proprietary production source code.

Listing 1: Generic asynchronous batching and request hedging

```
import asyncio
from collections.abc import Awaitable, Callable
from typing import TypeVar

T = TypeVar("T")

async def first_successful( tasks: set[asyncio.Task[T]],
) -> T:
    """Return the first successful task result."""
    errors: list[BaseException] = []
    pending = set(tasks)
    while pending:
        done, pending = await asyncio.wait( pending, return_when=asyncio.FIRST_COMPLETED,
```

```

)
for task in done: try:
    result = task.result() except BaseException as exc:
    errors.append(exc) continue
for loser in pending: loser.cancel()
await asyncio.gather(
    *pending, return_exceptions=True,
)
return result
raise RuntimeError(
    "All hedged calls failed"
) from errors[-1]

async def hedged_call(
    operation: Callable[[], Awaitable[T]], hedge_after_seconds: float,
) -> T:
    """
    Start one operation. If it has not completed after
    hedge_after_seconds, start one duplicate and return
    the first successful result. """
    primary = asyncio.create_task(operation())
    done, _ = await asyncio.wait(
        {primary}, timeout=hedge_after_seconds,
    )
    if primary in done:
        return primary.result()
    secondary = asyncio.create_task(operation())
    return await first_successful(
        {primary, secondary}
    )

async def process_batches( items: list[str], process_batch: Callable[
    [list[str]], Awaitable[T]
],
    *,
    batch_size: int = 4,
    hedge_after_seconds: float = 10.0, max_concurrency: int = 8,
) -> list[T]: """
    Partition input, process batches concurrently, and use one shared concurrency budget for
    primary
    and hedged calls. """
    if batch_size <= 0: raise ValueError(
        "batch_size must be positive"
    )
    if max_concurrency <= 0: raise ValueError(
        "max_concurrency must be positive"
    )
    semaphore = asyncio.Semaphore( max_concurrency
    )
    batches = [
        items[index:index + batch_size] for index in range(
            0,
            len(items), batch_size,
        )
    ]
    async def limited_call( batch: list[str],
    ) -> T:
        async def invoke() -> T: async with semaphore:
            return await process_batch(batch)
        return await hedged_call( invoke, hedge_after_seconds,
        )
    tasks = [
        asyncio.create_task( limited_call(batch)
        )
    for batch in batches

```

```

]
try:
    return await asyncio.gather(*tasks) except BaseException:
    for task in tasks: task.cancel()
    await asyncio.gather(
        *tasks, return_exceptions=True,
    )
    raise

```

Appendix B Development Checklist

- Before enabling request hedging, verify:
- The operation is read-only or protected by receiver-side idempotency.
- The HTTP client propagates cancellation or connection closure.
- The provider's billing behaviour for cancelled calls is understood.
- At least 1,000 batch-level latency observations are available.
- The hedge threshold is above the observed median.
- The threshold leaves time for a healthy duplicate before the application deadline.
- Primary and duplicate calls share one concurrency budget.
- Hedge activation rate, cost, timeout rate, and errors are monitored.
- A circuit breaker or kill switch can disable hedging.
- Output quality remains acceptable after batch decomposition.

References

1. Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56(2), 74-80.
2. Vulimiri, A., Michel, O., Godfrey, P. B., & Shenker, S. (2012, October). More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (pp. 13-18).
3. Suresh, L., Canini, M., Schmid, S., & Feldmann, A. (2015). C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (pp. 513-527).
4. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., ... & Stoica, I. (2023, October). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles* (pp. 611-626).
5. Python Software Foundation, "asyncio—Asynchronous I/O," Python 3 documentation.