Latency as Utility

Optimizing Serverless ASR Orchestration for Recursive LLM Workflows

Jack ten Bosch

2025-12-11

Latency as Utility: Optimizing Serverless ASR Orchestration for Recursive LLM Workflows

Abstract

While Word Error Rate (WER) remains the standard metric for Automatic Speech Recognition (ASR) evaluation, the integration of transcription into Large Language Model (LLM) workflows shifts the primary constraint from accuracy to latency. For the intended user who utilizes transcripts as dynamic prompts for querying rather than static text, processing delays introduce friction that degrades the utility of the feedback loop, and this friction compounds in ways that are difficult to quantify but immediately felt. This study evaluates the impact of orchestration parameters, specifically segment duration and concurrency, on wall-clock processing time within a serverless architecture. By optimizing segment granularity from 180 seconds to 360 seconds, we observed a 41% reduction in total processing time (2 minutes 15 seconds to 1 minute 19 seconds for 60 minutes of audio), validating that orchestration overhead, rather than GPU compute capacity, was the primary system bottleneck.

1. Introduction: The Temporal Constraints of Recursive Workflows

The utility of transcription technology is shifting from archival storage to active interrogation, and this shift changes everything about how we should evaluate these systems. In the SiftText workflow, the transcript serves as a "recursive learning loop," where audio data is converted into text not for passive consumption, but as prompt context for immediate interaction via an LLM. The transcript is, in a sense, a bridge, and the speed at which you can cross that bridge determines whether the destination remains cognitively accessible.

While some users engage in traditional reading, the intended high-utility workflow involves querying the transcript to extract specific insights. In this context, the transcript is an intermediate input, a means rather than an end. Consequently, value in this workflow is highly time-sensitive, because if the feedback loop is

delayed, the friction of time degrades the cognitive continuity required for effective querying. The user's mental model of the content decays while waiting, and by the time the transcript arrives, the original question may have lost its urgency or, worse, its context. Therefore, minimizing latency is both a performance enhancement and a requirement for maintaining the tool's core utility.

2. Architecture and Scaling Analysis

The experimental architecture utilizes a Python-based orchestrator that dispatches segmented audio to an auto-scaling pool of serverless GPU workers running the transcription ASR model. The initial optimization hypothesis, and this is where I started before the data told me otherwise, centered on vertical scaling: the addition of multiple GPUs per worker node to parallelize processing locally.

However, telemetry analysis of production workloads invalidated this hypothesis. Logs consistently revealed that the aggregate `gpu_elapsed_sec` (the total GPU time consumed across all segments) significantly exceeded the `wall_clock_time` (the actual time elapsed for the job). This observation can be formalized as a parallelism ratio:

$$k_{\text{effective}} = \frac{T_{\text{gpu\_elapsed}}}{T_{\text{wall\_clock}}}$$

For example, a 60-minute file recording 352 seconds of cumulative GPU processing was completed in only 132 seconds of wall time, yielding $k_{\text{effective}} = \frac{352}{132} \approx 2.7$. This ratio indicates that, on average, 2 to 3 workers were processing segments concurrently. The interpretation is straightforward: when $k > 1$, the infrastructure is already parallelizing horizontally, and adding local GPU resources would be redundant. You cannot outscale a system that is already scaling for you.

This discrepancy confirms that the serverless provider inherently scales horizontally, distributing distinct segments across multiple independent nodes in parallel. Consequently, increasing local GPU resources would offer negligible benefit, as the workload is already effectively parallelized at the infrastructure level. The bottleneck, it turns out, was never compute.

3. Methodology: Orchestration Efficiency

Recognizing that compute throughput was not the limiting factor, the methodology shifted to reducing the "orchestration tax," which can be defined as the cumulative latency incurred by network requests, R2 storage I/O, and container initialization for each independent job segment.

This tax can be modeled simply:

$$T_{\text{total}} = T_{\text{cold}} + n \cdot T_{\text{overhead}} + \frac{T_{\text{compute}}}{k}$$

Where:

- n = segment count
- T_{overhead} = per-segment orchestration cost (HTTP round-trips, storage I/O, container initialization)
- k = effective parallelism (determined externally by the cloud provider)
- T_{cold} = cold start latency (a one-time cost at job initiation)

The key insight from this model is that k is externally determined, you don't control how many workers the provider spins up, and T_{compute} is bounded by the ASR model's inference speed. The only controllable variable, then, is n. Reducing segment count directly reduces cumulative overhead, and this became the optimization target.

The primary variable manipulated was segment duration. We modulated the segmentation strategy from a baseline of 180 seconds up to 420 seconds. The rationale was that increasing segment size would reduce the total number of segments required for a given file, thereby reducing the frequency of control-plane interactions and data transfer events. Fewer segments means fewer round-trips, fewer container initializations, fewer opportunities for the orchestration tax to compound.

4. Results and Analysis

Moving from the baseline of 180-second segments to 360-second segments resulted in the optimal reduction of wall-clock time. For a standard 60-minute audio file, total turnaround time dropped from an average of 2 minutes 15 seconds to 1 minute 19 seconds, a reduction of approximately 41%.

The following query surfaced the parallelism insight by computing a parallelism factor across production jobs:

```sql
SELECT
  id,
  duration_sec / 60.0 AS audio_minutes,
  gpu_elapsed_sec,
  EXTRACT(EPOCH FROM (completed_at - started_at)) AS wall_seconds,
  gpu_elapsed_sec / EXTRACT(EPOCH FROM (completed_at - started_at)) AS parallelism_factor,
  chunk_count
FROM transcription_jobs
```

```
WHERE status = 'completed'
 AND duration_sec > 3000
ORDER BY completed_at DESC;
```

Values consistently exceeding 1.0 confirmed that horizontal scaling was already in effect, which validated the decision to optimize orchestration rather than compute capacity. The data told the story before I had to guess at it.

Extending segment duration further to 420 seconds yielded marginal gains (<5 seconds) while increasing operational risk, because larger segments increase the "blast radius" of a single failure, where a transient error in one segment impacts a larger portion of the total job. The diminishing returns pattern is visible in the following table:

Segment Duration	Segments (60-min file)	Wall Clock	Marginal Δ
180s	22	135s	baseline
300s	13	95s	−40s (30%)
360s	11	79s	−16s (17%)
420s	9	~75s	−4s (5%)

The marginal improvement diminishes beyond 360 seconds, while operational risk continues to increase. This intersection, where the cost of further optimization begins to outweigh the benefit, defines the equilibrium point. The results indicate that 360 seconds represents the efficiency equilibrium for this architecture, and pushing further would be optimizing for benchmarks rather than reliability.

5. Discussion: Variance and Infrastructure Limits

Despite the stabilization of average processing times at approximately 1 minute 19 seconds, identical datasets processed sequentially exhibited noticeable variance, with completion times ranging between 84 seconds and 109 seconds. Telemetry attributes this delta to "cold start" latency: the variable time required for the cloud provider to provision and initialize GPU resources upon receiving a request.

This variance can be framed as an irreducible stochastic component:

$$T_{\text{total}} = \mathbb{E}[T_{\text{process}}] + \epsilon_{\text{cold}}$$

Where ϵ_{cold} represents cold-start latency, observed empirically as a roughly 25-second delta between best and worst runs on identical inputs. The expected processing time is now optimized, and the remaining

variance is infrastructure-level noise, reducible only by maintaining idle warm capacity, which trades cost for consistency.

This is, I think, an important distinction: there is a difference between variance you can engineer away and variance that is simply the cost of doing business with a particular infrastructure model. Cold starts fall into the latter category. While this could be eliminated by maintaining a pool of provisioned "warm" workers, such a solution would incur significant idle costs for a marginal reduction in variance, and for a bootstrapped project, that tradeoff does not make sense.

6. Conclusion

We have successfully shifted the system bottleneck from architectural inefficiency to infrastructure availability. By aligning the segmentation strategy with the characteristics of serverless horizontal scaling, we achieved a 41% performance improvement without increasing computational costs. The optimization was not about adding resources, but about understanding where the time was actually going and attacking the right term in the equation.

The remaining variance in turnaround time is strictly a function of cold start mechanics. While this could be eliminated by maintaining a pool of provisioned "warm" workers, such a solution would incur significant idle costs for a marginal reduction in variance. For the target use case, where transcripts serve as LLM context rather than archival documents, the current performance envelope is sufficient.

The system's core workflow, transcribing technical content and querying it via LLM for accelerated comprehension, was applied recursively during development. Distributed systems concepts, serverless architecture patterns, and the optimization methodology documented in this paper were learned through the same transcribe-query-apply loop the system was designed to enable. The tool, in a sense, taught me how to build it, and hopefully this paper demonstrates that the loop works.