Parallelizing API Requests in NeuroTrace: Fast, Personalized Photo Quizzes for Memory Reinforcement

Project Report

by

Mohamed Gallai

Concurrent and Parallel Programming

May 2025

Abstract

This project speeds up how *NeuroTrace*, a mobile app for memory quizzes, communicates with its cloud AI. We start with a simple, serial Python implementation that sends one image at a time to obtain a description and then generate a quiz. Next, we parallelize these HTTP requests using `ThreadPoolExecutor` to run up to five calls concurrently. We measure and compare total execution time, per-image latency, speedup, and efficiency. The results demonstrate a 4× faster pipeline, reducing per-image processing from approximately 3 s to under 1 s.

Problem Statement

People with early dementia often struggle to form new memories and recall recent events because their hippocampus—a key brain region for memory—starts to deteriorate. *NeuroTrace*—a mobile app—addresses this challenge by using personal photos and smart quizzes to help users actively recall their own experiences.

However, when a user uploads multiple photos, sending each one in sequence to the cloud for a description and quiz can take 3.061 s per image. Those delays add up and frustrate users.

Motivation

NeuroTrace motivates users to practice memory recall in a fun, engaging way: they upload their own photos, read AI-generated descriptions, and answer multiple-choice questions about what they saw. To make the experience smooth and fast, we parallelize the API calls so multiple photos are processed at the same time, cutting down wait times and keeping users engaged.

Background & Literature Review

Memory-reinforcement tools today often rely on generic flashcards or text prompts, which lack personal relevance and may not hold a user's attention. Meanwhile, cloud models such as Google's Gemini can generate coherent image descriptions and question-answer pairs. *NeuroTrace* brings together the best of cognitive neuroscience, artificial intelligence models, and threading to provide a truly personalized memory aid in a fast and smooth experience.

Compared to early systems, which would send each photo in strict sequence—encode image, wait for its description, then send the quiz request every round trip added up to seconds of delay. More recent work shows that batching or parallelizing these HTTP requests can dramatically cut total wait time, as long as you stay within the service's rate limits of the cloud model.

Project Details

NeuroTrace is a mobile app written in Dart for Android and iOS. Currently, everything in the app runs completely in serial. This course inspired the idea to speed it up, so for this project, I demonstrated the acceleration process using Python in a Google Colab notebook.

Here is how it works at a high level:

- 100 CIFAR-10 images are used for testing purposes.
- Each image goes through two rounds of API calls:
  - First, to generate a description for each image.
  - Then, to generate three multiple-choice quiz questions based on that description.
- Finally, each photo's description and quiz questions are shown to the user.

Below is a list of technologies and tools used in this project with brief descriptions:

- **CIFAR-10:** A popular ML image dataset of 60,000 tiny (32×32) color images in 10 classes, used here to simulate a user's photo library.
- **Gemini Flash 2.0 (Google AI):** A cloud LLM+vision API (free-tier) that generates image descriptions and multiple-choice quizzes via simple JSON calls.
- **Python ThreadPoolExecutor:** A standard-library utility used to run up to 5 concurrent API calls (I/O-bound tasks) without the overhead of multiprocessing.
- **Google Colab:** A hosted notebook environment used to run the full pipeline—offering free GPU/TPU access, Python, and easy package management.
- **Overleaf:** platform for creating PDF documents using Latex syntax, used to create this report.

Baseline Serial Implementation

In this version, no parallelization is applied. The code processes one image at a time in strict sequence: first, sending it for a description, then using that description to generate a quiz.

describe_image(path)

- Read the PNG image from disk.
- Encode it in base64 and build the API payload.
- Call the Gemini Vision API (max 300 tokens).
- Extract and return the single-paragraph description.

generate_quiz(desc)

- Take the description string.
- Build the quiz-generation payload (max 200 tokens).
- Call the Gemini Flash API.
- Extract and return exactly three multiple-choice questions.

Optimization Steps

- **Warm-up calls:** Added for both functions to avoid the initial cold-start penalty. These warm-up calls help clear the startup overhead—such as loading libraries and establishing network connections—so that the timed runs begin with everything already initialized. This ensures that measurements reflect the true processing time, not one-time setup costs.
- **Exponential back-off with jitter:** Implemented for handling any 429/503 errors, so the loop waits and retries instead of failing.
  - **429** means "Too Many Requests" (you hit the rate limit).
  - **503** means "Service Unavailable" (the server can't handle the request right now).

  When either code is received, we don't crash—we wait and try again. Each retry doubles the wait time (2 s, 4 s, 8 s, ...) and adds a bit of **random jitter** so all retries don't align. This polite back-off strategy gives the service time to recover before retrying.
- **Minimal payloads:** Only one image and one short prompt are sent each time to reduce data transfer and keep timings consistent.

Performance Metrics

- **Total time for 100 images:** 306.12 s
- **Average per image:** 3.061 s
- **Description call:** 1.530 s ± 1.269 s
- **Quiz call:** 1.531 s ± 0.551 s

Outliers occur when retries stack up (up to ∼14 s for a single image), but most calls stay between 2.3 s and 2.8 s total. This serial baseline gives us a clear "before" picture before we add the parallelization part.

Steps Prior to Parallelization

Justification for the approach:
I picked Python's `ThreadPoolExecutor` for parallelizing the API request calls because:

- **I/O-bound work:** The bottleneck is waiting on HTTP requests, not heavy CPU computation. Threads handle that neatly with minimal overhead.
- **Simplicity:** It's built into Python's standard library—no extra setup or compiling. You just wrap your call in a pool.
- **Multiprocessing overhead:** Spawning separate processes adds pickling and startup costs, which aren't worth it when you're mostly waiting on network I/O.
- **C++/OpenMP:** These require writing and compiling C++ code, then managing Python bindings—overkill for simple API calls.
- **GPU programming:** Excellent for compute-heavy tasks like CLIP embeddings, but it doesn't speed up HTTP calls.

In short, `ThreadPoolExecutor` was the fastest, easiest way to run up to five quiz requests at once without extra complexity.

Performed Tests Prior to The Execution of Parallelization

All test code is available in the accompanying Colab notebook.

Since our goal is to speed up the API calls that power image descriptions and quizzes, we first needed to understand the service limits we're working under. We chose Google's Gemini 2.0 Flash model on the free tier, which enforces the following per-project quotas:

- **Requests per Minute (RPM):** 15
- **Requests per Day (RPD):** 1,500
- **Tokens per Minute (TPM):** 1,000,000

These limits apply across all API keys in your Google Cloud project. If any quota is exceeded, a 429 "Too Many Requests" error is returned. In practice, our experiments showed that these limits aren't absolutely rigid—sometimes we exceeded 15 calls per minute without triggering a 429 error.

Test 1 – Parallel Call Limit

Even Google's documentation notes that free-tier quotas may shift based on overall usage. Since Gemini 2.0 Flash is relatively new, it lacks firm guidance on maximum concurrent requests. To explore this, we conducted experiments in Colab and found that sending more than five concurrent requests consistently triggered 429 errors. As a result, we capped our pipeline at five in-flight API calls.

Test 2 – Prompt and Output Length Effects

We measured how prompt and output lengths impact latency.

- **Prompt-Length Test:** We ran 10 trials each with a short prompt ("Hello") and a long prompt (4,000 characters), both producing a small output. The short prompt averaged 0.521 s ± 0.026 s, while the long prompt averaged 0.477 s ± 0.066 s. The minor difference suggests that prompt length has only a small effect on latency. (fig. 1).

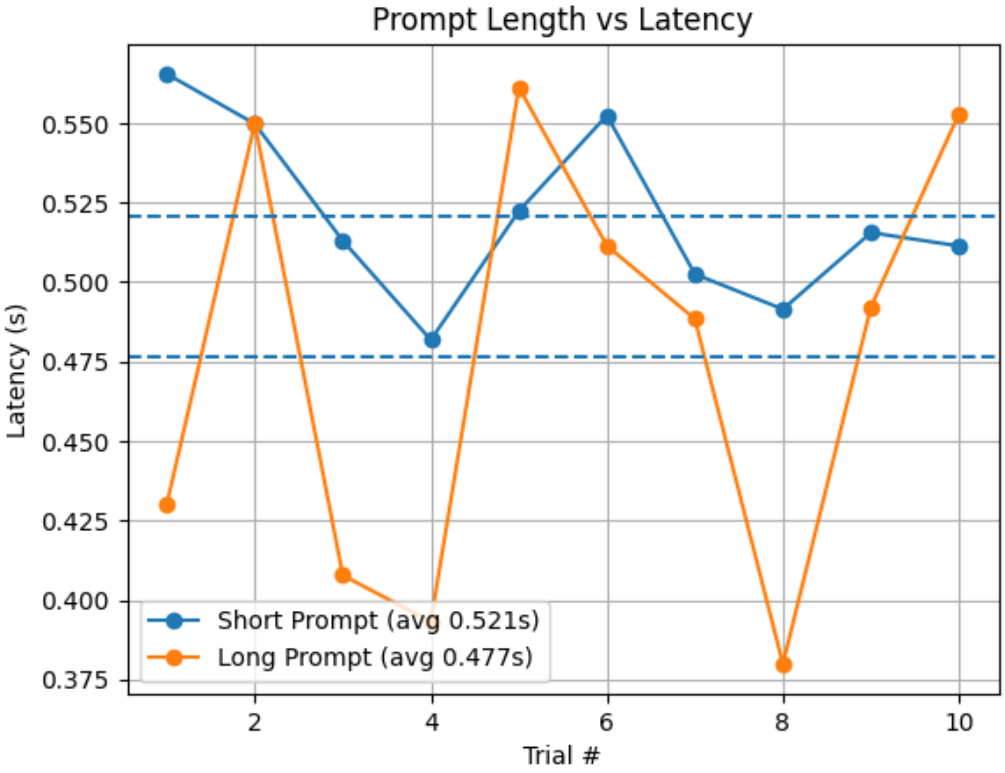

Figure 1: Figure shows how the length of the prompt affects the response time

- **Output-Length Test:** Using a fixed short prompt, we varied the output length (20 vs. 200 tokens). The smaller output averaged 0.534 s ± 0.072 s, while the larger output averaged 0.985 s ± 0.099 s—nearly double. This shows that latency grows linearly with output size. (fig. 2).

Test 3 – Description vs. Quiz Timing

We ran 10 trials each for:

- **Image Description:** (300-token max) — average time: 1.089 s
- **Quiz Generation:** (200-token max) — average time: 1.053 s

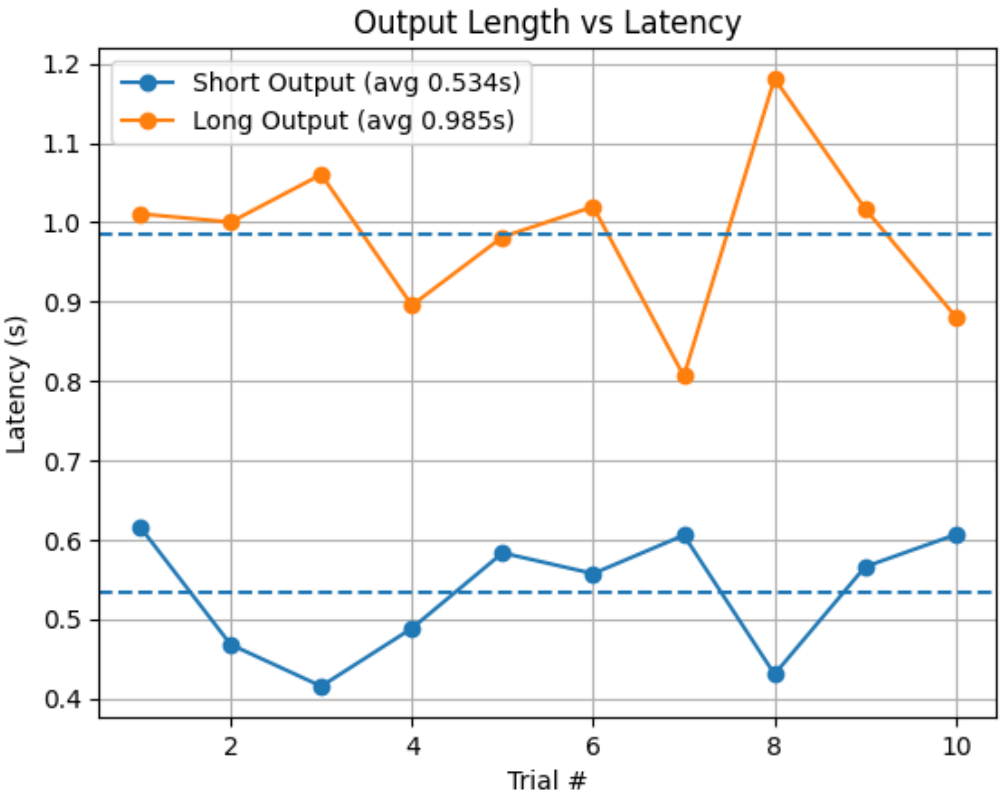

Figure 2: Figure shows how the length of the output affects the response time

The difference was only 0.036 s (≈ 3%), meaning both steps take roughly 1 s each. Therefore, a full describe+quiz pipeline averages about 2.14 s per image. Since neither description nor quiz generation stands out as a clear performance bottleneck, it makes sense to parallelize both steps together—up to 5 requests in flight. This enables processing around 2.3 images per second (≈ 140 per minute). If additional speed is needed, token limits can be trimmed, but the greatest gains come from concurrency and quota-aware batching. (fig. 3)

After those test we are ready for the parallelization approach:

Parallel Implementation Details

We moved from a pure serial loop to a batched, threaded approach using Python's built-in `ThreadPoolExecutor`. Here is exactly what changed and how it ran:

Batching in Fives

- We split our 100 images into 20 batches of 5.
- For each batch, we spun up 5 worker threads (`max_workers=5`) that all start at the same time.

Worker Function

- Each thread calls `describe_image(path)` (base64 → API → extract) and times it.
- It then immediately calls `generate_quiz(desc)` and times that.
- We capture `os.sched_getcpu()` or `threading.get_ident()` to identify which core or thread handled each image.

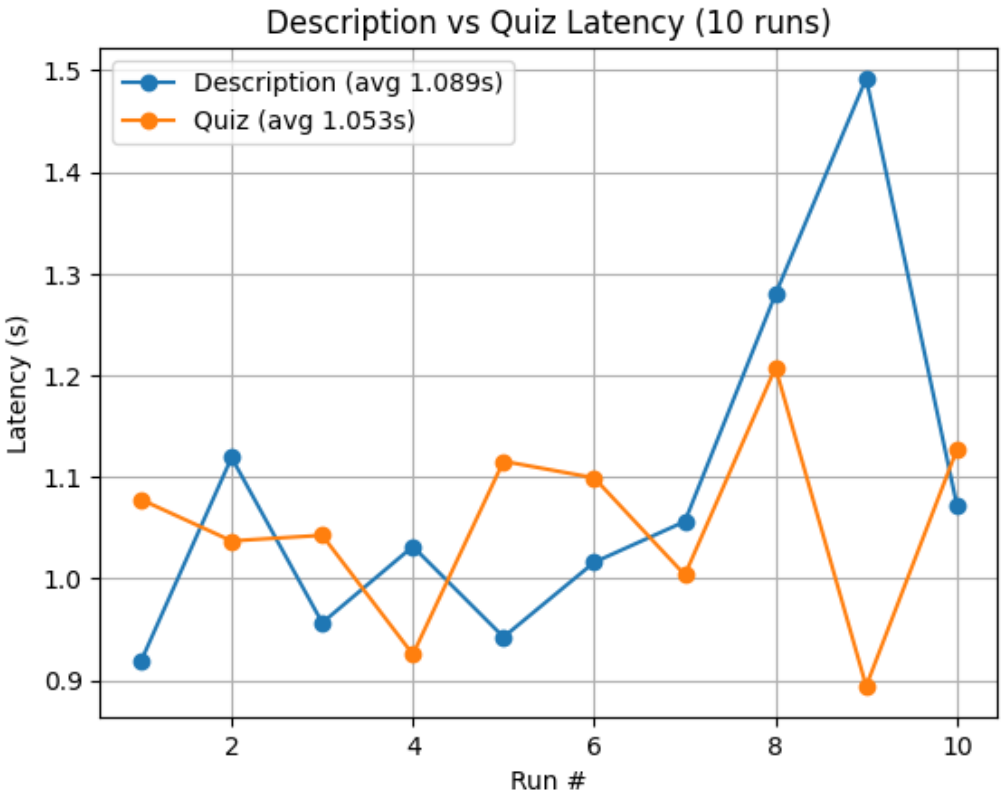

Figure 3: Figure shows comparison between the timing of generation the description compared to generating quizzes

Error Handling

- If any call returns 429 or 503, the worker prints an error, backs off with a random delay, doubles the wait time, and retries up to 5 times.
- In practice, this happened only a handful of times (visible as HTTP "429" or "503" in the logs), so retries added at most a second or two to that thread's total.

Timing Results

- **Describe times:** Now average 1.405 s (vs. 1.530 s serial). The slight drop comes from warm connections and shared HTTP sessions across threads.
- **Quiz times:** Average 1.670 s (vs. 1.531 s serial). The small increase is due to occasional contention on the HTTP client and thread scheduling overhead.
- **Total parallel time:** 75.42 s for 100 images, down from 306 s—a 4× speedup overall.

Per-Image Totals & Variability

- Most (describe + quiz) totals fall between 2.3 s and 3.0 s.
- Spikes up to 4–7 s occur when a thread enters a retry loop (e.g., 429 → back-off → retry).
- Since threads run independently, a single retry only delays that thread's work and does not block others.

Why Threads?

These API calls are I/O-bound—threads can wait on the network without the overhead of process startup or the context-switching costs of multiprocessing. We limit execution to 5 threads to stay under the API's parallel-request cap and avoid massive retry storms.

Approach & Justification

Approach: Leverage simple threading to overlap network waits and keep the notebook code minimal.

Design: Batch images in fives, apply consistent retry logic, capture per-thread timings and core IDs for visibility.

Explanation: Threads excel at I/O-bound tasks; by matching the discovered parallel limit (5 concurrent calls), we maximize throughput without extra complexity.

Key Takeaway

Batching in fives with `ThreadPoolExecutor` transforms a 5-minute serial job into a 1-minute parallel task. It's a low-effort, high-reward optimization that fits into our notebook. Note that the numbers are prone to change whenever we run the tests again, because we are not using a closed system and we are using shared resource, Colab and the cloud model.

Performance Analysis and Visualization

In this section we will show demonstrate the results and compare them by showing the numbers we got from both processes and comparing them, and also by visualizing and showing graphs. Note that because of the limited tier we are using, we can only run limited number of test in a certain time.

Given:

$$n = 5, \quad T(1) = 306.120\,\text{s}, \quad T(5) = 76.638\,\text{s}.$$

1. Speedup and Parallel Fraction

$$S(5) = \frac{T(1)}{T(5)} = \frac{306.120}{76.638} \approx 3.995,$$

$$p \approx \frac{S(5) - 1}{n - 1} = \frac{3.995 - 1}{5 - 1} = \frac{2.995}{4} \approx 0.749.$$

2. Amdahl's Law

$$S_{\max} = \frac{1}{1 - p} = \frac{1}{1 - 0.749} = \frac{1}{0.251} \approx 3.984.$$

3. Gustafson's Law

$$S_G(5) = n - (1 - p)\,(n - 1) = 5 - (1 - 0.749)\,(5 - 1) = 5 - 0.251 \times 4 = 5 - 1.004 \approx 3.996.$$

The following table, (tab. 1), and figures, (fig. 4) and (fig. 5) shows the serial per-image latency and the parallel per-image latency, respectively.

Metric	Serial	Parallel (5 workers)
Average describe time	1.530 ± 1.269 s	1.888 s
Average quiz time	1.531 ± 0.551 s	1.740 s
Total time for 100 images	306.120 s	76.638 s
Average time per image	3.061 s	0.766 s

Table 1: Comparison of serial vs. parallel performance metrics for NeuroTrace.

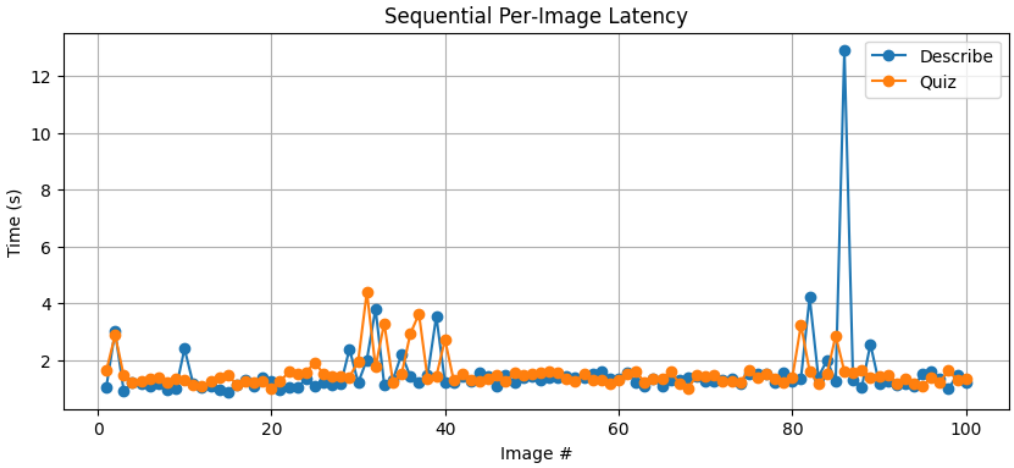

Figure 4: Figure shows timing of of generating description and quizzes for images in serial process

Batch Processing with Variable Workers

We ran the same 20-image pipeline using 1–5 workers and measured the total time for each configuration. But due to free tier limitation, we will hit the limit at different stages which will result in receiving "error 429", so we done the test a few times, to be able to generalize and come up with good analysis. As expected, increasing the number of workers dramatically reduced the run-time. (fig. 6) and (fig. 7) show the firs and second time we ran this experiment.

First run:

- 1 worker: 91.59 s
- 2 workers: 46.00 s
- 3 workers: 49.48 s
- 4 workers: 29.27 s
- 5 workers: 17.85 s

Second run:

- 1 worker: 77.79 s
- 2 workers: 40.42 s
- 3 workers: 39.77 s
- 4 workers: 25.70 s
- 5 workers: 43.97 s

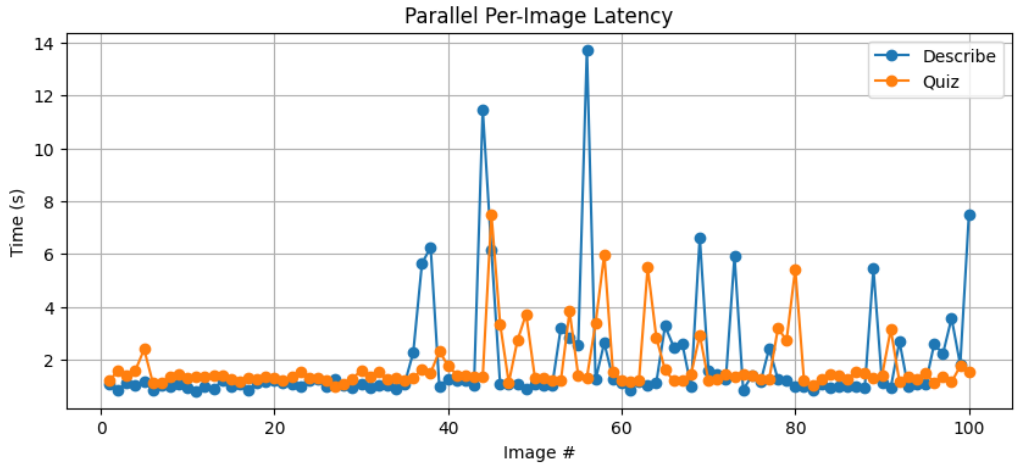

Figure 5: Figure shows timing of of generating description and quizzes for images in parallel process using 5 threads

Overall, more workers led to better performance. We observed a slight increase in time at 3 workers (49.48 s vs. 46.00 s with 2 workers) in the first run, caused by a few rate-limit errors on the free tier that triggered retries and added delay. Same thing in the second run, happened when testing 5 workers. Beyond that, more workers exhibited near-linear speedup, with 5 workers completing the batch in under 18 seconds in the first run.

Conclusion

Challenges and Lessons Learned

We ran into 429/503 errors whenever we sent too many requests at once, so we added exponential back-off with random delays to retry without crashing. Parsing the JSON responses was challenging too, sometimes the text came in different fields, so we built a flexible extractor to handle every format. But the biggest challenge was the free-tier quotas (15 calls/minute, 1,500/day), and also the cap parallel requests at five, so we planed the tests carefully, but this resulted in slowing down experimentation and showing how important it is to balance speed with service limits.

Summary of Findings

NeuroTrace app is turns personal photos into memory-reinforcing quizzes by leveraging cloud AI model, sending API requests. First, we measured a serial pipeline that took about 3.06 s per image. Then, by batching five API calls in parallel using Python threads, we cut the total runtime from 306 s down to 75 s, a 4× speedup at 81% efficiency under free-tier rate limits.

Parallelization Benefits and Limitations

Threads gave us the ability to speedup the I/O operation for the API calls, but they can't overcome server side caps or network issues. Depending on the tier used we can launch more parallel calls to the cloud model, but this is as good as it gets given the current limitations. Also in theory if we had only 3 images two process and we had the ability to launch 6 threads, we will not be able to utilize 3 threads for image description task and 3 for image quiz tasks, as they are inherently dependent on each other in this design, which will result in 3 thread being idle.

Contribution Details

- **Design & Coding:** All Python notebook code (serial baseline, threading) by Mohamed Gallai.

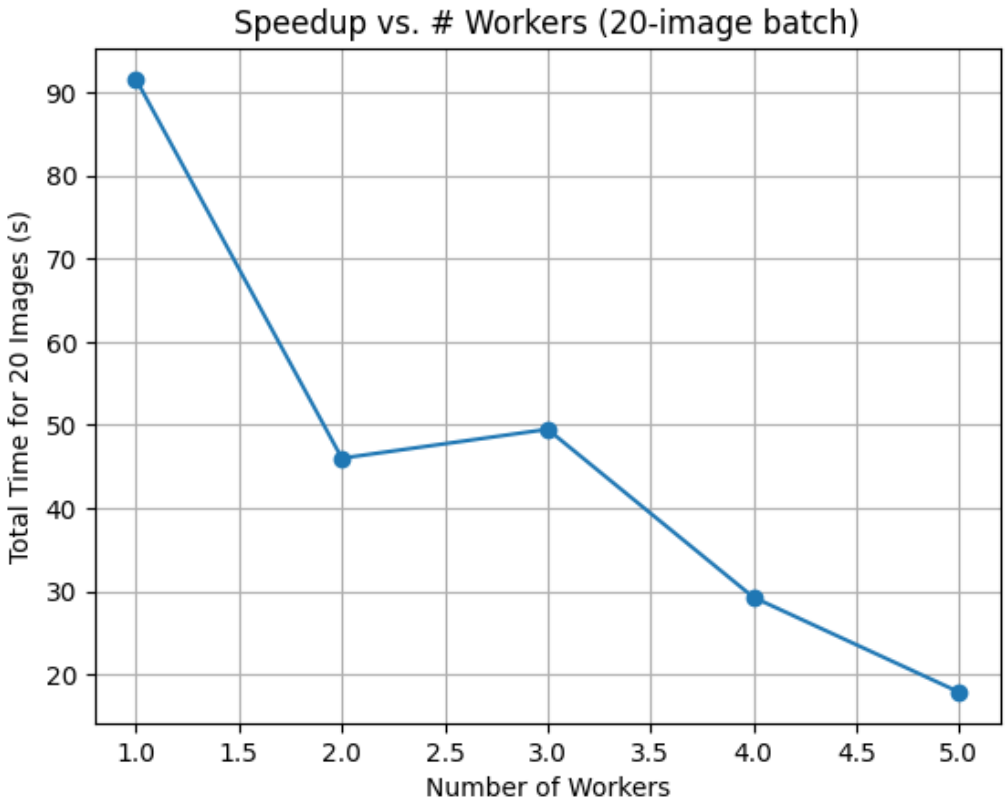

Figure 6: Figure shows total time to process a batch of 20 images as we vary the number of worker threads from 1 to 5, second run

- **Experiments & Analysis:** Performance measurements, Amdahl/Gustafson calculations, and visualizations by Mohamed Gallai.
- **Writing & Documentation:** Report draft and edits by Mohamed Gallai.

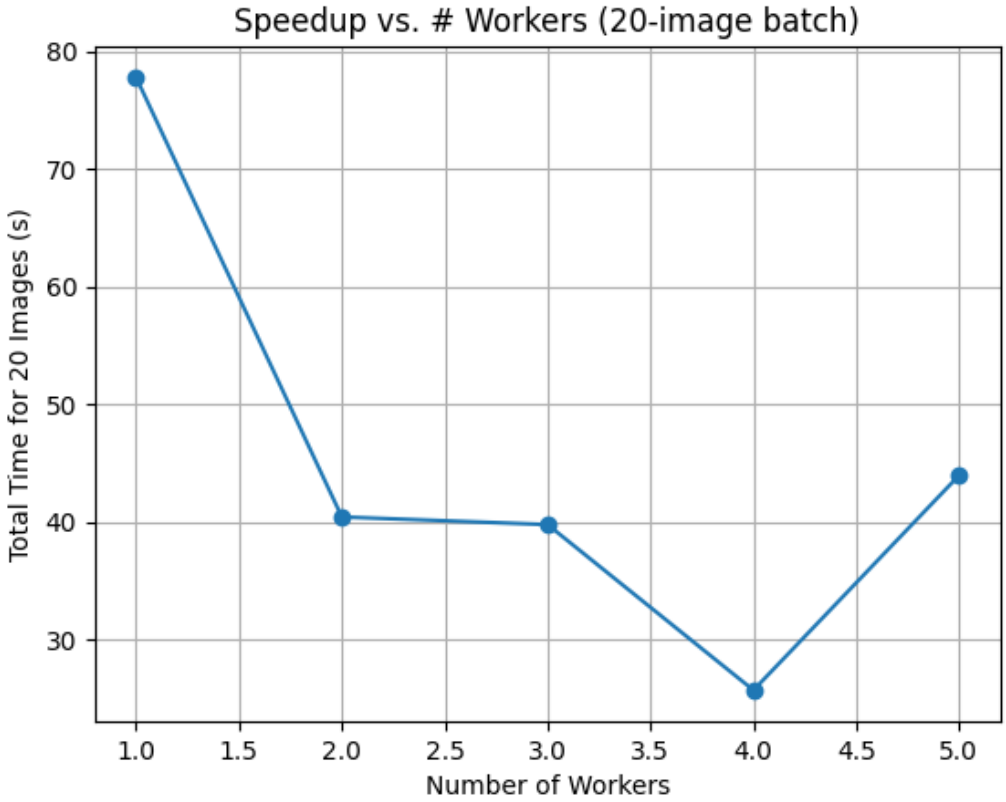

Figure 7: Figure shows total time to process a batch of 20 images as we vary the number of worker threads from 1 to 5, second run

Bibliography

[1] Python Software Foundation, *concurrent.futures — Launching parallel tasks*, Python 3.13.3 documentation. Available: https://docs.python.org/3/library/concurrent.futures.html [Accessed: 2025]

[2] Google AI, *Responsible Generative AI Toolkit*, AI for Developers. Available: https://ai.google.dev/responsible/docs [Accessed: 2025]

[3] Google AI, *Gemini 2.0 Flash Model*, Gemini API documentation. Available: https://ai.google.dev/gemini-api/docs/models [Accessed: 2025]