ChatGPT

Designing a Robust LLM Inference System with Queueing (Groq-Style Flow)

Overview and Objectives

We aim to build a **reliable multi-user LLM service** on a Windows PC (with an RTX 3060 GPU) that can handle 2–3 concurrent users via an ngrok tunnel. The key requirements are:

- **Queue-based request handling:** Fix issues with unreliable internet or concurrent access by queueing incoming LLM requests. This ensures each request is processed one at a time (preventing GPU overload) and no request is lost if connectivity fluctuates.
- **Groq-like flow:** Emulate the workflow of Groq's platform – i.e., accept requests quickly, queue them for processing, and deliver results asynchronously (with users polling or being notified when done) [1]. This includes measuring performance metrics for each request (latency, processing time, etc.).
- **Logging and monitoring:** Record how long each request ("record") takes, from queue wait to completion, and log these details locally on the host PC for analysis. We want insight into queue delays, processing times, and overall latency (similar to Groq's metrics like TTFT and total latency) [2].
- **State-of-the-art (SOTA) design:** Use modern, robust backend components suitable for ML workloads – ensuring scalability, resilience to crashes or restarts, and a good user experience (e.g. streaming responses for faster feedback).

By achieving the above, the system will provide a **stable LLM API** that can be accessed remotely (via ngrok) without dropping requests or overwhelming the hardware, even if multiple users hit it simultaneously or the network connection is unstable.

System Architecture and Components

Our recommended architecture consists of the following components, each chosen for reliability and performance:

- **1. FastAPI Web Server (API Layer):** A FastAPI application will serve as the front-end API. Users send their prompts/requests to this API (e.g., an HTTP POST request via the ngrok URL). FastAPI is high-performance and easy to implement, providing automatic docs (useful for testing) and support for asynchronous handling [3]. The API's job is mainly to **enqueue the request** and immediately respond with an acknowledgment (and possibly a job ID). This keeps the request handling snappy and frees the user from waiting online for long computations.

- **2. Task Queue and Broker (Redis + Celery):** For robust background processing, we use **Celery** (a distributed task queue) with **Redis** as the message broker and result backend. When the FastAPI server receives a request, it will send a task to Celery (which enqueues it in Redis) instead of processing it inline [4]. **Redis** (an in-memory data store) will **queue the task** and hold it until a worker is ready to process it. This decouples request submission from execution. Multiple incoming requests are stored in the queue and handled in order (first-in, first-out). The use of a

broker ensures tasks aren't lost even if the web server or network hiccups – they remain in the queue until processed. In fact, "when a task is called, Celery sends it to the broker; the broker holds the task until a worker process retrieves it and starts processing" [5]. This pattern guarantees that tasks will eventually run, providing a **permanent fix for transient connectivity issues** (if the client disconnects, the task still completes on the server).

- **3. Worker Process (Celery Worker with GPU access):** A Celery worker runs on the same PC, listening for queued tasks in Redis. This worker will have the **LLM model loaded on the RTX 3060 GPU**, and it pulls tasks one by one to execute [4]. By using a separate worker process, we ensure heavy LLM computations do not block the FastAPI server. Even if the main API process restarts or the user's connection drops, the worker can continue and finish the task [6]. We will configure Celery with an appropriate concurrency (likely **1** in this case, to only run one task at a time on the single GPU). The worker starts up, loads the model into GPU memory once, and then serves all requests using that loaded model. Each task consists of running the model inference on the given prompt and preparing the result (the model's generated response).

- **4. Logging and Monitoring Module:** On the worker side, for each task we will capture timing metrics. For example: when the task started execution, when it finished, and how long it took. We can also log queue waiting time (difference between task enqueue time and start time) to identify if requests are waiting long in the queue. These metrics are written to a local log file (e.g., `llm_requests.log`) on the PC. Optionally, we can log additional info like the number of tokens in the prompt/response or GPU memory usage if needed. The goal is to have a **detailed log of each request's performance**, similar to Groq's dashboard metrics (e.g., **"Latency: total server time from API request to completion"** and **"Time to First Token"** if streaming) [2]. This will help in optimizing the system and diagnosing issues.

- **5. Client Interaction (ngrok + Async Polling/Notifications):** Users will connect to the FastAPI endpoints via an **ngrok** URL (which exposes the local server to the internet). Because we aren't returning the LLM result immediately (since it's queued), we need a way to deliver the response when ready. There are two common approaches: **polling** or **callbacks**. In a simple implementation, when the user submits a request, the API immediately responds with a JSON containing a `task_id` (an ID for their queued job). The user can then call another endpoint (e.g. `/result/{task_id}` or `/status/{task_id}`) to check if the result is ready, or the front-end can poll periodically. This is akin to Groq's batch processing flow, where after submission "completion notification" can be via **webhook or polling**, and then the client retrieves the result when ready [1]. For 2–3 users, polling is manageable and easiest to implement. Alternatively, one could implement server-sent events or WebSocket to **push** the result to the user when done (more complex, but provides instant notification).

Below is a **workflow summary** tying these components together:

1. **User request:** A user (via a UI or script) sends a prompt to the FastAPI server (through the ngrok public URL). This is typically a POST request like `/generate` with the prompt and any parameters.
2. **Acknowledgment & Queue:** FastAPI immediately enqueues the request as a Celery task (e.g., `generate_text(prompt)`) in Redis and returns a response like `{"status": "queued", "task_id": "..."}` to the user. The user's request thus returns quickly without waiting for full completion.
3. **Task Processing:** The Celery worker (which has the LLM model loaded on GPU) picks up the task from Redis. It records a start timestamp, runs the model inference on the prompt, generates the

completion, records the end timestamp, and stores the result (e.g., in Redis backend or an in-memory store keyed by `task_id`). It also appends the timing info to the log file on disk. Because we use Celery, tasks are executed in the background worker and **"the Celery worker will then fetch and process the tasks one by one"** in the queue [4]. This guarantees only one uses the GPU at a time, preventing out-of-memory errors on the 3060 and providing consistent performance.

4. **Result Retrieval:** The user can now obtain the result. If we implement a polling endpoint, the user (or their client app) can call `/result/{task_id}` to get the output when it's ready. The FastAPI server can use Celery's AsyncResult or a shared database to fetch the stored result. If the task is not done yet, it can return a "pending" status; if done, it returns the generated text. (If we had implemented a callback or WebSocket, the server/worker could directly send the result to the user's waiting connection, but polling is simpler initially).
5. **Logging and Monitoring:** Meanwhile, the log file (and any Celery monitoring tool like Flower, if set up) will have an entry for this request, including how long it waited in queue and how long the model took to generate the answer. Over time, these logs show throughput and can identify any bottlenecks (for example, if multiple users cause long queue times or if some prompts take exceptionally long due to length).

Diagram (Conceptual): *FastAPI API -> (enqueue task) -> Redis Queue -> Celery Worker -> LLM Model (GPU) -> Result stored -> User polls for result.* (This is conceptually similar to typical Celery architectures where *"FastAPI sends tasks to a broker (Redis); Celery workers retrieve and process these tasks, saving results to the result backend"* [7].)

Backend Technology Choices

Why FastAPI + Celery + Redis? This combination is a proven stack for handling long-running tasks in web apps, including ML inference.

- **FastAPI** is a modern async framework that can easily handle incoming requests without blocking, and it integrates well with background task patterns. It allows adding tasks via `BackgroundTasks` or integrating with Celery for more robust needs [8] [9]. In our case, we use FastAPI mainly to receive requests and query task status.

- **Celery** is a mature task queue system for Python. It excels at asynchronous job processing and can run tasks outside the main web process. This prevents blocking and allows the web server to remain responsive even under heavy workloads [10]. Celery brings features like task retries, scheduling, and result storage out-of-the-box, aligning with our need for a full-fledged solution. Crucially, Celery tasks run in separate worker processes, so *"even if the server is restarted, the tasks keep running in the worker, and their status/results can be stored to check progress even after a crash"* [6]. This is essential for reliability – if our FastAPI app (or ngrok connection) goes down temporarily, any ongoing LLM jobs won't be interrupted; they'll continue on the worker and we can retrieve the results later.

- **Redis** acts as both the **message broker** and the **result backend** for Celery (it could be split, but using one Redis instance for both is convenient). Redis is lightweight, in-memory, and works on Windows (it can be run via Docker or WSL if needed). When a task is queued, it's essentially stored in Redis until a worker picks it up. The worker, after finishing, can also put the result into Redis (mapped by task ID), allowing FastAPI to fetch it. According to an architecture summary, *"the process begins with FastAPI sending tasks to Redis; Celery workers retrieve and process these tasks, then save results to the backend (Redis). FastAPI can monitor task status and outcomes via this setup."* [7] This nicely encapsulates why we use Redis+Celery: it provides a **persistent queue**

and a way to track results. Even if the friend's PC loses internet for a moment, the tasks remain in Redis and the worker can finish them; once connectivity is back, clients can fetch their results.

- **Hardware Utilization:** By leveraging the RTX 3060 via frameworks like PyTorch or TensorFlow (likely PyTorch/Transformers for LLMs), the worker can perform model inference on GPU. We will ensure the backend libraries (e.g. Hugging Face Transformers or llama.cpp for LLM) are installed and configured to use CUDA. The design guarantees that only one job runs on the GPU at a time (to avoid VRAM exhaustion), but if we ever needed to scale up, Celery could manage multiple workers or even workers on different machines (with more GPUs). For now, one GPU worker suffices, and Celery will simply queue any extra simultaneous requests.

Alternative Considerations: For a simpler setup (given only 2–3 users), one might consider using FastAPI's built-in `BackgroundTasks` or a simple Python thread queue. However, these approaches have limitations. FastAPI's `BackgroundTasks` runs in-process and isn't suitable for very long tasks or tracking after response [8] [11]. A custom thread-based queue could work, but you'd lose persistence (tasks would be lost if the process restarts). Celery+Redis is a bit more setup but is the **state-of-the-art solution for robust background job management** in Python – providing reliability, the ability to track task state, and easy scaling in the future [12] [5].

LLM Model Deployment on the GPU

With the infrastructure in place, we need to load and serve an LLM efficiently:

- **Model Choice:** Since the hardware is an RTX 3060 (12GB VRAM), choose an LLM model that fits in memory and provides good performance. For example, a 7B-13B parameter model (like Llama-2 7B/13B, GPT-J, etc.) possibly with 4-bit quantization to reduce memory usage. If higher quality is needed, one could use a larger model but likely with quantization or offloading some layers to CPU to fit. The model should be loaded once at worker startup to avoid re-loading on each request (which would be slow).

- **Framework:** Use Hugging Face Transformers with PyTorch (which supports CUDA) or an optimized inference library. The Celery worker can import the model and keep it in a global variable. For example, at worker initialization, do something like:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model_name = "TheModelName"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name,
device_map="auto")  # load to GPU
model.half()  # use FP16 for efficiency if supported
```

  This way, each `generate_text(prompt)` task can reuse `model` and `tokenizer` to produce an output (using `model.generate()` or similar). This **amortizes the loading cost** and ensures faster responses after the first load.

- **Sequential Processing:** As noted, we will run one generation at a time. Celery's default configuration spawns multiple worker processes (by default equal to CPU cores), but we will **limit concurrency to 1** (or simply run a single worker process) to avoid two processes fighting over the single GPU. This can be done by launching Celery with `--concurrency=1`.

Alternatively, if using threads within one process (Celery can use threads or an event loop with gevent), ensure only one thread invokes the model at once.

- **Handling Timeouts and Long Responses:** If some prompts might take very long (due to many tokens), we may set a reasonable limit on output length to control processing time. Also consider implementing a timeout in Celery (task time limit) to avoid any single job hanging forever. Celery allows setting a `time_limit` for tasks. If a task exceeds it (e.g., 2 minutes), it can be killed and marked failed – preventing the queue from freezing if something goes wrong.

- **Error Handling:** The task function should be wrapped in try/except to catch any model errors or memory issues. If an error occurs (e.g., out-of-memory), log the error, and mark the task as failed (so the API can report failure status). This ensures one bad request doesn't crash the whole system.

Queue Workflow and User Experience

From a user's perspective, using the service will look like this:

1. **Submit Prompt:** User sends a request (for example, via a small frontend or directly via cURL/Postman) to the server's endpoint (the ngrok URL forwarding to FastAPI). The request might be `POST /generate` with a JSON payload `{"prompt": "Your question here", "user_id": 123}` or similar.

2. **Immediate Response (Queued):** The server responds within a second with something like:

   ```
   { "status": "queued", "task_id": "abcd-1234-efgh", "message": "Your request is being processed." }
   ```

   This confirms the server received the query and queued it. The `task_id` is a unique identifier (UUID) for that job. The user can now disconnect or do other things; the request is safely in the system.

3. **Processing & Waiting:** The user waits for the result. Because only one job runs at a time, if two users submit simultaneously, the second one will have to wait in queue. Suppose User A's job starts processing immediately and User B's job is queued behind it. User B's `status` might remain "queued" for a short period. This waiting time is captured in our logs as well (queue time).

4. **Result Retrieval:** When the result is ready, the user has two main options to get it:

5. **Polling:** The user (or their client app) periodically calls `GET /result/{task_id}`. When the task is done, this endpoint returns `{"status": "completed", "result": "...generated text..."}` (and possibly some stats like tokens or latency). If the task is still running or queued, it might return `{"status": "in progress"}` or `{"status": "queued"}` accordingly. This method is straightforward and was implemented in our design (similar to the example where they added a `/task_status/{id}` route to check the state [13]).

6. **Notification (advanced):** Alternatively, the system could *push* the result. For example, if the user provided a callback URL or if we implemented WebSockets: the worker/API could send the result to the user as soon as it's ready. This would require the user to keep a connection open (or an endpoint to receive a webhook). Since ngrok can expose any port, one could even have the user run a local server to receive a callback. However, given only a few users, polling each second or two is simplest to implement and quite effective.

7. **Streaming Responses (optional for UX):** To mimic a truly *real-time* feel (like ChatGPT or Groq's on-demand service), we can implement **streaming tokens** as they are generated. This means, when a task is processing, the worker could yield partial output chunks. If using FastAPI, we can create a streaming response (using Server-Sent Events or chunked transfer) that the client can listen to for incremental tokens. Groq's platform highlights this as a feature: *"Streaming support for immediate perceived response"* is key for interactive applications [14]. Implementing this would involve more complexity (the client needs to handle a stream or we use WebSockets), so it's an optional enhancement. In our scenario, since the plan is to decouple via queue and polling, streaming might be less straightforward. But if we wanted, we could allow a single user (in on-demand mode) to get a streamed response when no queue is needed. For now, it's enough to note that streaming improves perceived latency and user engagement [15], and we can consider it once the basic system is stable.

Logging and Performance Tracking

We fulfill the requirement of recording how long each request took and other metrics by implementing comprehensive logging:

- **Log Format:** Each job will produce a log entry. We can choose a format like CSV or JSON lines in a file, e.g.:

  ```
  [Timestamp] task_id=abcd-1234, prompt_length=50 tokens,
  result_length=200 tokens, queue_wait=2.3s, processing_time=5.8s,
  total_time=8.1s, status=success
  ```

  This captures when the task was processed, how big the input/output were (if tokenization info available), how long it waited in queue, how long the model took to generate, and the total from submission to completion. We'll append each record to a log file on disk.

- **Implementation:** In the Celery task function (the function that runs the model), we can do something like:

  ```
  import time
  start_queue_time = ...  # passed in or captured when enqueuing if
  possible
  start_time = time.time()
  # run model generate...
  result = model.generate( ... )
  end_time = time.time()
  # Compute durations
  wait_time = start_time - start_queue_time
  ```

```
proc_time = end_time - start_time
total_time = end_time - start_queue_time
# Log to file
with open("llm_requests.log", "a") as f:
    f.write(f"{time.strftime('%Y-%m-%d %H:%M:%S')} task={task_id}
queue_wait={wait_time:.2f}s proc_time={proc_time:.2f}s
total={total_time:.2f}s\n")
```

We can obtain `start_queue_time` by capturing the enqueue time (for example, store the current time on the task object when we queue it – Celery allows passing args, or simpler, note it in a global dict as shown in a simpler FastAPI example [16]). Alternatively, the API could include the current timestamp as part of task submission arguments. Regardless, we want to measure how long the job sat in Redis waiting.

- **Monitoring Tools:** In addition to logs, since we are using Celery, we can use tools like **Flower** (a Celery monitoring web UI) or simply Celery's event logs to monitor tasks. Celery will log each task start and finish with timestamps if `--loglevel=info` is used, giving another source of truth. For a lightweight setup, our own log file is sufficient.

- **Analyzing the Logs:** Over time, these logs will let us answer questions like: What's the average processing time? Are users sending very large prompts that slow things down? Is the queue wait time usually zero or are jobs piling up? This can guide future improvements (e.g., if many concurrent requests, maybe adding another GPU or moving to a bigger model server). It's worth noting that Groq's console provides similar data (TTFT, tokens/sec, etc.) for their hosted models [2] – we are essentially building a mini version of that for our local system.

Ensuring Reliability and Resilience

This design inherently improves reliability in several ways:

- **Tasks survive disconnections:** If the user's internet or the ngrok tunnel drops mid-way, the task still lives in the Celery worker. The user can reconnect later and retrieve the result by the task ID. This addresses the "permanent internet issue" – no query will be lost due to a flaky connection, thanks to the decoupling via the queue. The only caveat is that the initial request must reach the server to be queued. If it didn't, the user should retry. But once queued, the result will be generated whether or not the user remains connected.

- **Isolation of heavy processing:** The web API and the LLM processing are separated (different processes via Celery). So a crash in the model (or a GPU memory error) won't crash the API process handling incoming requests. Similarly, if we need to update the API or restart uvicorn, we can do so without interrupting the worker. Celery tasks in progress will continue unaffected. This is a best practice for production systems – you can deploy updates to the API or handle errors gracefully while background jobs continue [6].

- **Retry and Failure Handling:** Celery can be configured to automatically retry failed tasks a certain number of times, which could be useful if, say, the model occasionally fails due to an out-of-memory error that could resolve on a second attempt with a cleared cache. We can also catch exceptions in the task and perhaps return a safe error message as the result (so the user gets something like "Sorry, your request could not be processed." instead of hanging indefinitely).

- **Security considerations:** Since the service is exposed via ngrok, we should secure the API. FastAPI makes it easy to add an API key or token auth for each request. Given only trusted users (2–3 people) will use it, we could simply use a shared secret or ngrok's built-in access control. This prevents abuse of the endpoint by others (ngrok URLs are hard to guess, but if leaked, someone could spam the model). Rate limiting can also be implemented (though Celery's queue inherently serializes execution, we might still limit how fast one user can enqueue tasks if needed).

- **Performance tuning:** The logging will reveal if the GPU is underutilized or overloaded. We might find that tasks are quick and queue is usually empty – great. If tasks start queueing often (users sending requests faster than the model can handle), we might consider scaling up: e.g., upgrade hardware or spin up another worker on another machine (Celery can distribute tasks to multiple workers across machines). With 2–3 users, this is unlikely an issue, but it's good that the architecture can scale – just by adding more workers (and possibly a load-balancer at the API layer to distribute requests, or having tasks specify a routing key to different workers for different models etc.). The design is flexible for future growth.

Conclusion and Summary

By introducing a queued, asynchronous processing pipeline **just like Groq's high-performance inference setup**, we ensure that our LLM service is **reliable, efficient, and user-friendly**. The system uses **state-of-the-art practices** from industry: a FastAPI REST interface for ease of integration, a Celery + Redis task queue for managing background jobs, and a dedicated GPU worker for heavy LLM computations. This design resolves prior internet stability issues by not tying the request/response directly to a single connection – requests are retained until completion and can survive disconnects. It also prevents the GPU from being overwhelmed by concurrent jobs, as tasks line up in an orderly fashion.

Furthermore, we've built in comprehensive **logging and monitoring**. Each inference's queue time and processing duration are recorded, so we have full transparency into performance (similar to how GroqConsole tracks latency metrics) [1]. These logs will enable continuous improvement and tuning of the system (for example, optimizing prompts if TTFT is high due to large inputs, or enabling streaming to cut perceived latency).

In essence, the proposed solution provides everything one would expect from a production-grade LLM deployment: **request queuing, asynchronous execution, progress tracking, result retrieval mechanisms, error handling, and performance logging**. By using a robust backend and queue, both the users and the system owner (your friend) can trust that every request will be handled fairly and efficiently, even under less-than-ideal network conditions.

Finally, to implement this: set up the environment with FastAPI, Celery, and Redis on the Windows PC (Docker can run Redis if needed). Use ngrok to expose the FastAPI port, and test with a couple of simultaneous requests to see the queue in action. You'll notice that one request will wait until the first finishes, and the log file will show the timing. This architecture is **scalable, resilient, and modeled on SOTA practices**, ensuring a smooth experience for the 2–3 users (and easily more in the future).

Sources: The design draws upon known best practices and examples in the industry: using background task queues for heavy ML workloads [8] [4], leveraging Celery for asynchronous processing [5] [6], and even mirroring features from Groq's AI inference platform such as streaming and queued batch

processing [14] [1], to guarantee reliability and performance. The references cited provide additional context and confirmation of these approaches.

---

[1] [2] [14] [15] Understanding and Optimizing Latency - GroqDocs
https://console.groq.com/docs/production-readiness/optimizing-latency

[3] [8] [9] [11] Run ML Models on API endpoints using FastAPI — BackgroundTasks | by Rajesh Pyne | Medium
https://medium.com/@rajesh.pyne/run-ml-models-on-api-endpoints-using-fastapi-backgroundtasks-ac3791d70b06

[4] [6] [13] [16] Celery and Background Tasks. Using FastAPI with long running tasks | by Hitoruna | Medium
https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d

[5] [10] [12] Mastering Celery: A Guide to Background Tasks, Workers, and Parallel Processing in Python | by Khairi BRAHMI | Medium
https://khairi-brahmi.medium.com/mastering-celery-a-guide-to-background-tasks-workers-and-parallel-processing-in-python-eea575928c52

[7] FastAPI Template for LLM SaaS Part 2 — Celery and Pg-vector | by Euclidean AI | Towards AI
https://pub.towardsai.net/fastapi-template-for-llm-saas-part-2-celery-and-pg-vector-0db158b6c1b0?gi=9fedf0e5ca1c