Mistral 22B EXL2 Q4 Setup

I want to load Mistral 22B using ExLlamaV2 EXL2 quantization at Q4. I also want you to consider those configurations as well:

```
gpu_memory_utilization=0.97,
max_model_len=16384,
max_num_seqs=2,
cpu_offload_gb=4,
swap_space=8
```

To help you load Mistral 22B using ExLlamaV2 EXL2 quantization at Q4 with your listed configurations, could you please confirm the following:

1. What inference framework or environment are you using (e.g., Text Generation WebUI, Oobabooga, vLLM, custom Python script)?
2. Are you using a single GPU or multiple GPUs? If multiple, how many and what are their VRAM sizes?
3. Is the model already quantized, or do you need steps to quantize it to EXL2 Q4?

These details will help tailor the guidance to your setup.

What inference framework or environment are you using (e.g., Text Generation WebUI, Oobabooga, vLLM, custom Python script)? Whatever suitable but I' prefer vLLM if it proves easy to load the quantized model for inference.

Are you using a single GPU or multiple GPUs? If multiple, how many and what are their VRAM sizes? Single GPU

Is the model already quantized, or do you need steps to quantize it to EXL2 Q4? If there's an open source quantization with the above criteria just load it with vLLM. If not then quantize it to EXL2 Q4 with suitable Calibration Dataset to measure quantization errors.

Thanks! I’ll look into whether Mistral 22B is available in EXL2 Q4 format and compatible with vLLM, and if not, I’ll explore how to quantize it using ExLlamaV2 with a suitable calibration dataset. I’ll also ensure the configuration supports your single GPU setup with high memory utilization, CPU offloading, and large sequence lengths.

I’ll let you know once I have all the necessary information and instructions.

Running Mistral 22B in 4-Bit (EXL2) with vLLM

Availability of Mistral 22B in EXL2 Q4 Format

Yes – **Mistral-22B v0.2** (a 22-billion parameter model by Vezora) has been quantized in ExLlamaV2's EXL2 4-bit format. For example, the user *bartowski* provides an EXL2-quantized version on Hugging Face. In that repository, different branches correspond to various average bits-per-weight; the **"4_25"** branch (~4.25 bits per weight) is roughly equivalent to a GPTQ 4-bit quantization. There is also a "4_0" branch (exact 4.0 bits) for maximum compression, though at some quality loss. These EXL2 quantizations significantly reduce VRAM requirements – e.g. ~13.3 GB for 4k context and ~15.8 GB for 16k context at ~4-bit precision, making single-GPU usage feasible.

However, **vLLM does *not* natively support loading the EXL2 format**. vLLM's quantization support focuses on **AWQ** (Activation-aware Weight Quantization) and **GPTQ** formats, not the custom EXL2 format used by ExLlamaV2. This means you **cannot directly load** bartowski's EXL2 quantized model into vLLM's engine. Instead, the recommended approach is to use an AWQ 4-bit quantized model of Mistral-22B, which vLLM can handle out-of-the-box .

Fortunately, a suitable **4-bit AWQ** model of Mistral-22B v0.2 *does* exist. For example, **solidrust/Mistral-22B-v0.2-AWQ** on Hugging Face is a 4-bit AWQ quantization of the same model. This model is specifically designed for low VRAM usage – it's part of a "22B AWQ Collection" intended to run on a single ~24 GB GPU (or two 12 GB GPUs). Using the AWQ model with vLLM will give you similar memory savings to EXL2, while being compatible with vLLM's high-throughput inference engine.

Loading the 4-bit Model in vLLM

To run the AWQ-quantized Mistral-22B in vLLM, follow these steps:

1. **Install vLLM and AutoAWQ** (for AWQ support):

   ```bash
   pip install vllm autoawq
   ```

vLLM 0.7+ has built-in support for AWQ models, but installing `autoawq` ensures the necessary components are available .

2. **Download the AWQ model** from Hugging Face. You can either use the model ID directly in vLLM (it will download on first run), or manually clone it:

```bash
git lfs install
git clone https://huggingface.co/solidrust/Mistral-22B-v0.2-AWQ
```

3. **Launch vLLM with the AWQ model and desired settings.** You can use the **OpenAI-compatible HTTP server** or the offline API. Here's an example using the CLI to start the server with the given configuration:

```bash
vllm serve solidrust/Mistral-22B-v0.2-AWQ \
    --quantization awq --trust-remote-code \
    --gpu-memory-utilization 0.97 \
    --max-model-len 16384 \
    --max-num-seqs 2 \
    --cpu-offload-gb 4 \
    --swap-space 8
```

Explanation:

- `--quantization awq` explicitly tells vLLM to treat the model as AWQ 4-bit . (vLLM may auto-detect AWQ, but this flag is safe to include.)
- `--trust-remote-code` is added in case the model or quantization code requires custom code (the Mistral model uses the standard Llama architecture, but trusting remote code is often needed for custom model classes).
- The other options correspond to the user's requirements:
    - `--gpu-memory-utilization 0.97` – use 97% of GPU memory for the model execution engine . This helps vLLM allocate nearly all of the GPU's VRAM (e.g. ~23.3 GB on a 24 GB card) for model weights and activation buffers.
    - `--max-model-len 16384` – sets the maximum sequence length (context length) to 16384 tokens. This ensures vLLM limits memory allocations appropriately for 16k context. It's important for models with RoPE scaling (extended context) to avoid vLLM overestimating needed memory.

- `--max-num-seqs 2` – allows at most 2 sequences (requests) to be processed concurrently . We set this small since we likely only need a couple of parallel sequences (and it keeps memory usage predictable).
- `--cpu-offload-gb 4` – reserves 4 GiB of CPU RAM for offloading weights if needed . vLLM will move less-used layers to CPU memory to stay within the GPU utilization limit if necessary.
- `--swap-space 8` – allocates up to 8 GiB of host memory as swap space for paging out model data . This acts as an overflow buffer if both GPU and CPU-offload memory are saturated. (In some cases, reducing swap-space can avoid certain OOM issues, but 8 GB is a reasonable starting point.)

Once launched, vLLM will load the 4-bit model and host an API server. You can send prompts to it using the OpenAI API format or via the `LLM` Python interface. The above configuration is tuned for a single high-VRAM GPU setup (e.g. an RTX 6000/ A6000 48GB, or RTX 4090 24GB plus CPU offloading for spillover). **Running Mistral-22B 4-bit with 16k context on a 24 GB GPU** is feasible with these settings.

Note: AWQ quantization in vLLM is still being optimized. The vLLM team notes that AWQ models currently have slightly lower throughput than full-precision models (due to memory bandwidth overheads), but they significantly reduce memory footprint . This trade-off is acceptable for single-GPU scenarios where memory is the main constraint.

Quantizing Mistral 22B to EXL2 Q4 with ExLlamaV2

If no suitable pre-quantized model was available (or if you prefer ExLlamaV2's quantization), you can quantize Mistral-22B yourself using ExLlamaV2's conversion script. ExLlamaV2's **EXL2** format uses a GPTQ-like approach with mixed-precision weight allocation for optimal accuracy. Here's how to do it:

1. **Prepare the base model:** Download the original FP16 Mistral-22B v0.2 weights. The model is hosted by Vezora on Hugging Face . You can use the HF Hub Python API or `git lfs` to fetch the model files (which will total ~45 GB in FP16). Ensure you have enough disk space.
2. **Install ExLlamaV2:** Clone the ExLlamaV2 repository and install its requirements:

   ```bash
   git clone https://github.com/turboderp-org/exllamav2.git
   cd exllamav2
   pip install -r requirements.txt
   ```

(Alternatively, install via pip if available: `pip install exllamav2`.)

3. **Select a calibration dataset:** EXL2 quantization requires a calibration dataset to measure the error introduced by quantizing each layer. A good choice is a sample of diverse text similar to the model's training data:

   - For a general instruct/chat model like Mistral-22B, **The Pile** is often used. For example, the ExLlamaV2 author notes that *"ThePile 0007.parquet"* (a shard of the Pile) is used for general chat/instruct models.
   - Another common choice is **WikiText**. In fact, Turboderp used the WikiText test set to calibrate a Mistral-7B instruct model .

   You can obtain a suitable calibration file by using the 🤗 **Datasets** library. For example, to use WikiText-2:

   ```python
   import datasets
   ds = datasets.load_dataset("wikitext", "wikitext-2-raw-v1", split="test")
   ds.to_parquet("wikitext_test.parquet")
   ```

   Or to use a portion of The Pile (if you have access to it), ensure it's saved in a similar format (parquet with one text sample per row).

4. **Run the ExLlamaV2 conversion script:** ExLlamaV2 provides `convert.py` to handle quantization. The process is usually done in two passes – first **measurement**, then **quantization** – to achieve the target average bitrate:

   - **Measurement pass:** Evaluate the model's layers with the calibration data to record quantization errors.
   - **Quantization pass:** Actually quantize weights based on those measurements, targeting the desired bits-per-weight.

   For example, Turboderp shared the commands he used for the 7B model:

   ```bash
   # Measurement step (example for 7B model, using WikiText)
   python convert.py -nr \
       -i /path/to/Mistral-22B-v0.2-FP16/ \
       -o /path/to/work_dir/ \
       -c /path/to/wikitext_test.parquet \
       -om /path/to/work_dir/measurement.json \
   ```

```
    -gr 100

# Quantization step (target ~4.0 bits)
python convert.py -nr \
    -i /path/to/Mistral-22B-v0.2-FP16/ \
    -o /path/to/work_dir/ \
    -c /path/to/wikitext_test.parquet \
    -m /path/to/work_dir/measurement.json \
    -cf /path/to/output_dir/Mistral-22B-exl2-4bit/ \
    -b 4.0
```

In the first command:

- `-i` specifies the input model folder (FP16 weights).
- `-o` is a working directory for temporary files.
- `-c` is the calibration data file (parquet).
- `-om` outputs the measured errors to a JSON file.
- `-gr 100` (optional) increases the number of measurement *rounds* per layer (for more precise error profiling at the cost of VRAM). Turboderp used `-gr 100` for thorough measurement , but note this can be memory-intensive (he suggests omitting `-gr` if you have <24 GB VRAM).

The second command uses the measurement to quantize:

- `-m` provides the measurement JSON from the first pass.
- `-cf` is the output directory for the quantized model.
- `-b 4.0` sets the target average bits per weight (4.0 in this case). You can adjust this (e.g. 4.25 for slightly higher quality at a minor memory cost).

After the quantization completes, the output directory (here “Mistral-22B-exl2-4bit”) will contain the 4-bit model (split into *.safetensors shards, plus config and tokenizer files). **Verify the model files** (`*.safetensors`, `config.json`, etc.) are present.

5. **Using the quantized model:** Since vLLM cannot load EXL2 format, you should use an ExLlamaV2-compatible inference solution. For instance, you can use the TabbyAPI server or a local UI like text-generation-webui with ExLlamaV2 backend to load the quantized model. The prompt format for Mistral v0.2 is typically the Guanaco style (as noted in the model card). With ExLlamaV2's optimized kernels, you should be able to generate text with Mistral-22B on a single GPU in 4-bit mode, even at extended context lengths.

   Optional: If you **must** use vLLM with a quantized model that you created, consider quantizing to **AWQ** instead of EXL2. You can use the **AutoAWQ** tool to quantize Mistral-22B to 4-bit AWQ format (similar to the example in vLLM's docs). The process is simpler (one-step) and the resulting AWQ model will be directly loadable by vLLM. For example, using AutoAWQ in Python:

   ```python
   from awq import AutoAWQForCausalLM
   model = AutoAWQForCausalLM.from_pretrained("Vezora/Mistral-22B-v0.2",
   trust_remote_code=True)
   model.quantize(q_config={"w_bit":4, "q_group_size":128, "zero_point":True})
   model.save_quantized("Mistral-22B-v0.2-awq")
   ```

   This will produce a folder `Mistral-22B-v0.2-awq` with the 4-bit weights. You can then launch vLLM with `--quantization awq` on that folder as shown above. (In practice, using the existing solidrust AWQ model is easier, but this is how you'd do it yourself.)

References and Resources

- **ExLlamaV2 GitHub:** turboderp's ExLlamaV2 library and README – explains EXL2 quantization and provides the conversion script.
- **Mistral-22B-v0.2 Model:** Base FP16 model by Vezora on Hugging Face .
- **EXL2 Quantized Model:** bartowski's 4-bit ExLlamaV2 quantization of Mistral-22B.
- **AWQ Quantized Model:** solidrust's 4-bit AWQ version of Mistral-22B (suitable for vLLM).
- **vLLM Documentation:** *AutoAWQ* usage and AWQ support in vLLM , and engine arguments reference for memory settings .
- **Quantization Example:** Turboderp's notes on quantizing Mistral using ExLlamaV2 (HuggingFace discussion) – includes command-line usage and tips on sequence length and rounds for calibration.
- **Calibration Data:** LoneStriker's EXL2 calibration dataset choices for different model types, confirming The Pile for general models.

Sources

install bartowsk 4k EXL2-quantized version from Hugging Face locally without necessarily using vLLM. With those parameters if possible.

gpu_memory_utilization=0.97,
max_model_len=16384,
max_num_seqs=2,
cpu_offload_gb=4,
swap_space=8

To guide you on installing and running Bartowski's 4k EXL2-quantized Mistral-22B model locally (without vLLM), could you confirm the following:

1. Are you planning to use a specific inference tool like Text Generation WebUI, TabbyAPI, LM Studio, or a custom Python script?
2. What GPU do you have, and how much VRAM does it provide?
3. Are you targeting inference through a web UI, API, or CLI?

This will help me tailor the instructions to your environment.

Are you planning to use a specific inference tool like Text Generation WebUI, TabbyAPI, LM Studio, or a custom Python script? Pick a suitable tool for the above mention parameters/configuration

What GPU do you have, and how much VRAM does it provide? RTX Ti 4070 with 16GB VRAM

Are you targeting inference through a web UI, API, or CLI? API

Great, I'll find the best way to install and run Bartowski's EXL2 4k quantized Mistral-22B model locally on an RTX 4070 Ti with 16GB VRAM using an API interface. I'll choose the most suitable tool (likely TabbyAPI or Text Generation WebUI with ExLlamaV2) that supports EXL2 quantization and respects your configuration settings, including:

- `gpu_memory_utilization=0.97`
- `max_model_len=16384`
- `max_num_seqs=2`
- `cpu_offload_gb=4`

- `swap_space=8`

I'll include setup steps, installation commands, and how to launch the model via an API. I'll let you know once everything's ready.

Running Bartowski's EXL2-Quantized Mistral-22B Locally via API

Running Bartowski's EXL2-Quantized Mistral-22B Locally via an API

Selecting an ExLlamaV2-Compatible Inference Tool

To serve the **EXL2-quantized Mistral-22B (Bartowski's model)** with a local API, you should use an inference backend that supports **ExLlamaV2** (the library needed for EXL2 quantizations). The two main options are:

- **TabbyAPI:** A FastAPI-based server that is the *official* API backend for ExLlamaV2 . It supports the new EXL2 quantized models (as well as GPTQ and FP16) and provides an OpenAI-compatible API out of the box . This is the most straightforward choice for your use case, since it's lightweight, fast, and built specifically for ExLlama V2 models.
- **Text Generation WebUI (Oobabooga):** A web UI that can also load ExLlamaV2 models (via the `exllamav2` loaders) and offers an API mode. This is viable if you prefer a GUI, but it may require manually updating the ExLlamaV2 backend and enabling the API extension. It's less streamlined for pure API serving compared to TabbyAPI.

Given that you **don't need a UI and want an API interface**, **TabbyAPI** is recommended for simplicity and performance. Below are detailed steps to install TabbyAPI, download the model, configure it for a 16k context on a 16GB GPU (with CPU offloading), and serve it via an API.

Installing TabbyAPI and ExLlamaV2 Backend

Follow these steps to set up TabbyAPI on your system (with an RTX 4070 Ti, CUDA 12.x compatible):

1. **Install Python 3.10+ (ideally 3.11).** Make sure you have a proper Python installation with `pip` available (avoid the MS Store version on Windows as it can cause pip issues). On Windows, ensure you have Visual Studio Build Tools installed (for compiling any extensions) . On Linux, ensure you have CUDA drivers for your GPU (CUDA 12) or install the CUDA toolkit if needed (PyTorch can often install CUDA libraries if not present) .
2. **Clone the TabbyAPI repository:** Open a terminal and run:

   ```bash
   git clone https://github.com/theroyallab/tabbyAPI.git
   cd tabbyAPI
   ```
3. **Launch the TabbyAPI setup script:**
   - On **Windows**, run `start.bat`.
   - On **Linux/Mac**, run `./start.sh`.
     The first run will set up a Python virtual environment and prompt you to choose the GPU backend. Select the option for **CUDA 12.x** (appropriate for an RTX 4070 Ti). The script will then install TabbyAPI's dependencies, including the **ExLlamaV2** library and optimized kernels. (If for some reason the script fails, you can manually create a venv and do `pip install -U .[cu121]` inside the repo to install with CUDA 12 support .)
4. **Start the TabbyAPI server (no model loaded yet):** After installation, the script should automatically launch the API. If it doesn't, you can rerun `start.bat` / `start.sh` to start the server. Initially, TabbyAPI will run with **no model loaded** by default . You should see a message that the server is listening (on `127.0.0.1:5000` by default) and waiting for a model to be loaded. We'll load the model next.

Downloading the Mistral-22B EXL2 Model (Bartowski's Quantization)

Bartowski's repository provides Mistral-22B v0.2 in the EXL2 quantized format on Hugging Face. There are multiple quantization branches (from 3-bit up to 8-bit) – higher bit means better quality but more VRAM usage . For a **16GB VRAM** GPU at 16k context, you'll likely use either the **5.0-bit** or **4.25-bit** branch to fit memory constraints:

- **5.0 bits:** ~18.0 GB VRAM needed at 16k context . This offers high performance/quality (Bartowski marks it as recommended) . With 16GB VRAM, you'd need to offload ~2GB to CPU (which is within your 4 GB CPU offload budget).

- **4.25 bits:** ~15.8 GB VRAM at 16k . This will comfortably fit in 16GB (no offload needed for the model weights) and still has good quality (comparable to 4-bit GPTQ) .

Using a 6.5-bit quant (near FP16 quality) would require ~21.9 GB at 16k – beyond your GPU's capacity – so it would depend heavily on offloading to CPU and even disk. It's safer to choose 5.0 or 4.25 for smooth operation. Here's how to download the model:

- **Option 1: Use TabbyAPI's built-in downloader.** Tabby can fetch Hugging Face models directly. For example, to download the 5.0-bit branch of Bartowski's model, run (in the `tabbyAPI` directory):

  ```bash
  # Linux/Mac
  ./start.sh download bartowski/Mistral-22B-v0.2-exl2 --revision 5_0

  # Windows
  start.bat download bartowski/Mistral-22B-v0.2-exl2 --revision 5_0
  ```

  This will clone the Hugging Face repo for that branch into Tabby's `models` directory . (If the model is gated or requires acceptance, you might need to supply a Hugging Face token; however, this Mistral model is Apache-2.0 licensed and should download without a token.)

- **Option 2: Manual download via git or HF client.** You can manually clone the repo. For example:

  ```bash
  mkdir -p models/Mistral-22B-v0.2-exl2
  git lfs install
  git clone --single-branch --branch 5_0 \
    https://huggingface.co/bartowski/Mistral-22B-v0.2-exl2 \
    models/Mistral-22B-v0.2-exl2
  ```

  Ensure you have `git-lfs` installed, as the model weights are large. This will download all necessary files (model `*.safetensors`, `config.json`, tokenizer, etc.) into a folder under `models/`. You can substitute `5_0` with `4_25` or another branch if you choose a different quantization.

After downloading, you should have a directory like `tabbyAPI/models/Mistral-22B-v0.2-exl2/` containing the model files. (The folder name will likely already be `Mistral-22B-v0.2-exl2` if you used the commands above – if not, you can rename it accordingly.)

Configuring the Model for 16K Context & Memory Offloading

Next, configure TabbyAPI to load the model with the desired context length and memory usage parameters. There are two ways to set configurations in Tabby: via the `config.yml` file or via command-line flags when starting the server. We'll outline the key settings you need:

- **Model Selection:** In Tabby's `config.yml`, set the `model_name` to the folder name of the model. For example:

  ```yaml
  model_name: "Mistral-22B-v0.2-exl2"
  model_dir: "models"          # (if you placed the model in a non-default path, adjust this)
  ```

  This ensures Tabby loads that model on startup. (Alternatively, you can specify `--model Mistral-22B-v0.2-exl2` as a flag when launching the server.)

- **Max sequence length (context window):** Set `max_seq_len: 16384` to allow a 16k token context . By default, Tabby will use the model's built-in context length (which for Mistral-22B v0.2 is likely 4096 or 8192). We override it to 16384. This will allocate a larger KV cache and inform ExLlamaV2 to use rope scaling for extended context. In `config.yml` you can add:

  ```yaml
  max_seq_len: 16384
  cache_size: 16384   # (cache_size defaults to max_seq_len, but you can set it explicitly)
  ```

  RoPE scaling: Because we are extending beyond the model's native context, we need to adjust RoPE (rotary positional embedding) scaling. TabbyAPI can handle this via config. You can specify:

  ```yaml
  rope_scale: 0.25
  rope_alpha: null
  ```

Here, `rope_scale: 0.25` tells ExLlama to **compress** the positional embeddings by 4x (since 1/4 = 0.25) to achieve roughly 4x context length (4096 × 4 = ~16384) . We leave `rope_alpha` as null so that Tabby will auto-calculate it if needed . *(Note: Some models might have an explicit `rope_scaling` field in their HF config.json; if so, ExLlamaV2 will respect that. If not, the above settings apply a linear scaling.)*

- **Batch/sequence settings:** Set `max_num_seqs: 2`. This defines the maximum number of sequences (requests) that can be processed in parallel (Tabby uses continuous batching to handle concurrent requests). In practice, `max_num_seqs: 2` means the engine can handle up to 2 prompt+generation sequences at once, which suits your desire for at most 2 concurrent sequences. You can include this in `config.yml` or use the flag `--max-num-seqs 2` when launching. (If not set, it might default higher; limiting it to 2 can save memory.)
- **GPU memory utilization:** By default, ExLlamaV2 doesn't automatically grab *all* GPU memory, leaving a safety margin. You can increase the fraction of VRAM to use. Setting `gpu_memory_utilization: 0.97` (97%) will tell the backend to utilize almost the entire 16GB for model layers and cache . This helps fit the large model + context. (Keep a small buffer for CUDA kernels to avoid OOM.) If using CLI, add `--gpu-memory-utilization 0.97`.
- **CPU offloading:** You indicated **4 GB CPU offload**. ExLlamaV2/Tabby can offload some model layers or KV cache to CPU RAM when VRAM is insufficient. Set `cpu_offload_gb: 4` (or CLI flag `--cpu-offload-gb 4`) to allow up to 4 GiB of RAM usage for offloaded data . With the 5.0-bit model (~18 GB at 16k), this offload will be used to hold the overflow (~2 GB) that doesn't fit in VRAM. *(Ensure your system has at least 4 GB of free RAM for this.)*
- **Disk swap (paging):** If you want to be safe with memory, you can also allocate disk swap space for the model. Setting `swap_space: 8` will allow up to ~8 GB to be used from disk (NVMe recommended) if both GPU and RAM are filled . This is a fallback to avoid out-of-memory crashes by swapping out least-used weights to disk. Use `swap_space: 8` in config or `--swap-space 8` flag. In practice, if your VRAM (16GB at 97% util) + 4GB RAM are still not enough (unlikely in this configuration), it will start using up to 8GB on disk.

How to apply these settings: You can either edit Tabby's `config.yml` to include the above options under the **Model Options** section, or simply pass them as command-line arguments when starting Tabby. For example, you could launch the server with:

```bash
# Starting TabbyAPI with model and overrides (Linux/Mac example)
./start.sh --model Mistral-22B-v0.2-exl2 \
  --max-seq-len 16384 --max-num-seqs 2 \
  --gpu-memory-utilization 0.97 --cpu-offload-gb 4 --swap-space 8
```

This will start TabbyAPI and automatically load the `Mistral-22B-v0.2-exl2` model with a 16k context window and the specified memory usage parameters. (On Windows, you'd modify the `start.bat` invocation similarly.) The CLI flags override the config file settings , so you can use whichever method you prefer. Just ensure the values match those above.

> 💡 *Tip:* If you edit `config.yml`, you might also set `use_as_default` for these options so they persist on every model load . For instance: `use_as_default: [max_seq_len, gpu_memory_utilization, cpu_offload_gb, swap_space, max_num_seqs]` to make those the defaults for any model you load.

Finally, consider disabling authentication for ease of use (if you're the only one using the API locally). In `config.yml` set `disable_auth: true` to turn off API key requirements . By default, TabbyAPI generates an API key for OpenAI compatibility, but if you disable auth you won't need to provide any key when making requests.

Launching the Server and Using the API

With the model downloaded and configuration in place, you're ready to run the server and use the API:

- **Start TabbyAPI with the model:** If you configured `model_name` in config or used the `--model` flag, Tabby will load Mistral-22B on startup. Launch the `start` script (with the flags as needed). The first time loading this 22B model, **expect a bit of a delay** – ExLlamaV2 may compile some CUDA kernels (if not already cached) and then load the quantized weights into memory. Monitor the console output; you should see logs about allocating cache, loading layers, etc. Once finished, TabbyAPI will report that the model is loaded and the server is running on your specified host/port (default `127.0.0.1:5000`).
- **Verify memory usage:** Tabby/ExLlama will try to use ~97% of the 16GB VRAM. With the 5.0-bit model at 16k context, roughly 14–15GB will go to the model and 1–2GB to the KV cache. The remaining overflow should be sitting in CPU RAM (up to 4GB). This fits within our plan (16GB * 0.97 ≈ 15.5GB on GPU, plus 2-3GB on CPU for safety, under the 4GB limit, and possibly a few GB swap unused). If something doesn't fit, the swap file (up to 8GB) will catch it. In short, the model should now be running without OOM, courtesy of ExLlamaV2's efficient paging system.
- **Using the API:** TabbyAPI exposes an **OpenAI-compatible REST API** . This means you can use OpenAI client libraries or tools like cURL to hit the endpoints. For example:
  - **Chat completions:** `POST http://localhost:5000/v1/chat/completions` with a JSON body containing your conversation (`model` field can be set to the model name or left as "gpt-3.5-turbo" if you enabled dummy names, since Tabby can pretend to be an OpenAI model). e.g.:

```json
  {
    "model": "Mistral-22B-v0.2-exl2",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Hello, how are you?"}
    ]
  }
  ```

  Tabby will format this into the model's prompt template and generate a completion. (It has a flexible template system and even built-in ones for chat; by default it should use a generic chat format unless you specified a custom one .) For Bartowski's model, the expected chat format is: **"### System: ... ### Human: ... ### Assistant:"** . Tabby's default template for chat likely conforms to this format, but you can double-check or create a template if needed.

  - **Completion endpoint:** `POST http://localhost:5000/v1/completions` for single-turn prompt->completion (if you prefer to send a raw prompt instead of chat format).

Because we set `disable_auth: true`, you won't need an API key in the header. Otherwise, Tabby would require an `Authorization: Bearer <token>` header (and it prints the token on startup or uses one from `api_keys.yml` if configured).

- **Testing the model:** Try a simple request to ensure everything works. For example, using `curl`:

  ```bash
  curl http://localhost:5000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{"model": "Mistral-22B-v0.2-exl2", "prompt": "### Human: Hello, how are you?\n### Assistant:", "max_tokens": 100}'
  ```

  This should return a JSON with a completion (the model's answer). You can also integrate this endpoint in apps like Postman, or point an OpenAI-compatible UI (e.g. SillyTavern, etc.) at `localhost:5000` to chat with the model .

Note on performance: ExLlamaV2 is very optimized, so even with a 22B model at 4–5 bit quant, you should get decent speeds on a 4070 Ti. The `max_num_seqs=2` setting means the server can handle two requests interleaved, but if you only send one at a time, it will devote full GPU to it. If you ever need to adjust performance vs. memory, you can tweak the `max_num_seqs` (higher allows more parallel generation but uses more VRAM for batch), or adjust `gpu_memory_utilization` down if you encounter any instability at 0.97 (though 0.97 is typically safe).

Finally, remember that this model (Mistral-22B v0.2) is **uncensored and experimental** (as noted by the creators). Use it responsibly. You now have a local API running the model – you can integrate it into applications or interact with it just like you would with an OpenAI model, but entirely on your own hardware.

Sources:

- TabbyAPI (ExLlamaV2 server) documentation and features .
- Bartowski's Mistral-22B EXL2 model card (quantization branches and VRAM requirements) .
- TabbyAPI usage examples for downloading models and configuring context length & RoPE .

Sources

keep max_seq_len: 4096 at native value

Perfect, keeping the native context length (4096 tokens) simplifies the setup and ensures the best performance and stability.

Here's your simplified, recommended configuration with native context length:

Updated Configuration:

Parameter	Value
`gpu_memory_utilization`	**0.97**
`max_seq_len`	**4096** (native)
`max_num_seqs`	**2**
`cpu_offload_gb`	**4** GB

Parameter	Value
`swap_space`	**8** GB

✅ Final Setup & Launch Instructions:

1. Launch TabbyAPI Server with Correct Model:

Navigate to your `tabbyAPI` directory and start the server with these flags:

```bash
./start.sh --model Mistral-22B-v0.2-exl2 \
  --gpu-memory-utilization 0.97 \
  --max-seq-len 4096 \
  --max-num-seqs 2 \
  --cpu-offload-gb 4 \
  --swap-space 8
```

For Windows:

```cmd
start.bat --model Mistral-22B-v0.2-exl2 ^
  --gpu-memory-utilization 0.97 ^
  --max-seq-len 4096 ^
  --max-num-seqs 2 ^
  --cpu-offload-gb 4 ^
  --swap-space 8
```

2. Verify the Model Loads Correctly:

Check your console logs to confirm:

- Model files are loaded from `models/Mistral-22B-v0.2-exl2`.
- GPU memory utilization (~15-16 GB out of your 16 GB VRAM).
- CPU offloading occurring if necessary (up to 4 GB).
- Swap space (8 GB) available if GPU+CPU RAM isn't sufficient.

3. Using Your API Endpoint:

Your local OpenAI-compatible API endpoint will run at:

```bash
http://localhost:5000/v1/completions
```

Example test request via `curl`:

```bash
curl http://localhost:5000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{"model": "Mistral-22B-v0.2-exl2", "prompt": "### Human: Hello!\n### Assistant:", "max_tokens": 100}'
```

This ensures optimal memory use, stability, and native performance for your GPU with Mistral-22B at EXL2 4-bit quantization (Bartowski).

You’re good to go! Let me know if you encounter any issues or need more help.