**Building a Jarvis-like Local AI Assistant on an M2 Mac (16GB RAM)**

Building a responsive **Jarvis-like assistant** locally involves choosing the right model, fine-tuning it for structured outputs, optimizing the voice-command pipeline, integrating automation tools, and implementing an efficient wake word. Below is a step-by-step guide covering each aspect:

**Step 1: Model Choice – DeepSeek R1 vs. Mistral 7B**

**Performance & Efficiency:** Mistral 7B is designed for speed and low resource use, whereas DeepSeek R1 is more complex. Mistral’s architecture is a **fully dense transformer** optimized with techniques like Grouped-Query and Sliding Window attention, giving it a **compact VRAM footprint** ideal for edge devices​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Mistral%207B%3A%20Optimized%20for%20Speed,Efficiency)

. In contrast, DeepSeek R1 is a **retrieval-augmented (RAG) model** that can pull in external context for better long-term knowledge, but this adds overhead​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=DeepSeek%20R1%3A%20Retrieval)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=,dynamically%2C%20but%20requires%20additional%20infrastructure)

. On a GPU benchmark, Mistral 7B reached ~35 tokens/s versus ~31 tokens/s for DeepSeek R1, and required less memory (14GB vs 16GB VRAM in FP16)​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Training%20Dataset%20Size%203,turn%20chat)

. This implies that on a local Mac, Mistral will generally run faster and lighter. The trade-off is that DeepSeek’s built-in retrieval might handle open-domain queries slightly better, but it **requires additional infrastructure for the knowledge store**​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=,dynamically%2C%20but%20requires%20additional%20infrastructure)

, which complicates a local setup.

**Accuracy & Capabilities:** Both models perform well for their size, with each excelling in different areas. Mistral slightly leads in general knowledge and reasoning tasks, while DeepSeek R1 has a small edge in code generation and multi-turn chat quality​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Metric%20Mistral%207B%20DeepSeek%20R1,publicly%20benchmarked%20Not%20publicly%20benchmarked)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=,performs%20better%20in%20code%20generation)

. For a “Jarvis” assistant that needs to both chat and execute commands, **consistency and responsiveness** may matter more than marginal accuracy differences. Mistral 7B is already tuned for instructions (especially if using an Instruct variant), which helps in following commands. DeepSeek’s retrieval could be advantageous if you plan to give the assistant a large knowledge base or long conversation history (it supports up to 64K token contexts with retrieval)​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=DeepSeek%20R1%3A%20Retrieval)

, but if your use-case is primarily controlling the local machine and casual Q&A, this advantage is not critical.

**Recommendation:** Use **Mistral 7B** as the base model for local inference, given its speed and lower memory footprint on Apple Silicon. It’s more self-contained (no external DB or internet needed) and carries a permissive Apache 2.0 license​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=,good%20for%20commercial%20use)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=%23%20Key%20Architectural%20Trade)

. DeepSeek R1 is powerful, but its RAG architecture would entail running a retriever and storing documents, which isn’t necessary for typical assistant tasks. Keeping the setup simple will yield a snappier assistant.

**Quantization Format for Apple Silicon:** To run a 7B model on a 16GB M2 Mac, use a **4-bit quantized model** in the GGUF format (the successor to GGML). GGUF is specifically designed for efficient CPU/GPU inference on Macs, allowing layers to offload to the Apple Neural Engine/GPU for speed​

[e2enetworks.com](https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq#:~:text=GGUF%2C%20the%20successor%20of%20GGML%2C,approach%20for%20handling%20model%20files)

. This yields excellent performance – for example, using llama.cpp with a 4-bit GGUF model, an M2 Max can generate ~40 tokens/sec with near-zero CPU use by leveraging all GPU cores​

[reddit.com](https://www.reddit.com/r/LocalLLaMA/comments/140nto2/full_gpu_inference_on_apple_silicon_using_metal/#:~:text=Other)

. On an M2 Pro or base M2 (with fewer GPU cores), you can still expect on the order of ~10–20 tokens/sec, which is very fluid. Alternative quantization like **GPTQ** (another 4-bit method) works, but is better suited to NVIDIA GPUs; on Apple Silicon, GPTQ models often run on CPU and won’t match the speed of GGUF with Metal acceleration​

[e2enetworks.com](https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq#:~:text=GGUF%2C%20the%20successor%20of%20GGML%2C,approach%20for%20handling%20model%20files)

. Another option is **AWQ** (Activation-Aware Quantization), which some reports found to produce slightly better output quality for Mistral​

[e2enetworks.com](https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq#:~:text=Also%2C%20you%E2%80%99ll%20see%20the%20fluctuation,compared%20to%20the%20quantized%20models)

, but it’s less common in Mac workflows. The easiest path is to download a pre-quantized Mistral 7B GGUF (in 4-bit or 5-bit) from a repository like Hugging Face (e.g. TheBloke’s quantized models​

[huggingface.co](https://huggingface.co/TheBloke/Mistral-7B-Instruct-v0.2-GGUF#:~:text=TheBloke%2FMistral,kindly%20provided%20by%20Massed)

) and run it with llama.cpp or an app like Ollama on macOS. In summary, **GGUF 4-bit** is the sweet spot for speed and low memory on Apple Silicon, preserving accuracy while enabling fast local inference​

[heidloff.net](https://heidloff.net/article/qlora/#:~:text=Memory,bit%20model%20finetuning)

.

**Step 2: Fine-Tuning Strategy for Structured JSON Output**

**LoRA vs. QLoRA:** Fine-tuning your model with low-rank adaptation will teach it to produce the exact JSON formats you need. Standard **LoRA** fine-tuning (with 16-bit base weights) works, but **QLoRA** is ideal here – it keeps the model weights in 4-bit precision during training, drastically reducing memory usage while maintaining almost the same performance​

[heidloff.net](https://heidloff.net/article/qlora/#:~:text=Memory,bit%20model%20finetuning)

. QLoRA essentially *combines* LoRA’s efficient adapter training with model quantization, making it feasible to fine-tune a 7B model on a MacBook Pro. (In fact, community members have managed to QLoRA-fine-tune Mistral 7B on as little as 8GB RAM by using the GPU with mixed-precision math.) With 16GB RAM, you can fine-tune Mistral 7B using QLoRA on the M2’s GPU (via Metal) or CPU, though it may be slow – alternatively use a cloud VM for training and then run the model locally. In either case, QLoRA will let you train the model’s adapters without needing a high-end GPU, making **continual fine-tuning** viable over time.

**Improving JSON Output:** Prepare a fine-tuning dataset of example commands and their desired JSON responses. Fine-tuning on structured output examples is very effective – the model will learn to format its response as JSON without extra prompting​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=Use%20case%202%3A%20specific%20format)

. For instance, if your assistant should output {"action": "open\_app", "app": "Safari"}, include numerous variations of user requests (“Open Safari”, “Launch the web browser”, etc.) paired with the correct JSON. Mistral’s documentation notes that training on a target format helps the model **conform to that format** reliably​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=Use%20case%202%3A%20specific%20format)

​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=Fine,outputs%20in%20this%20specific%20format)

. Using QLoRA, you’ll update only a small number of parameters (the LoRA layers) – the model retains its general knowledge but gains a specialized skill in JSON structuring.

**Reducing Prompt Overhead:** A key benefit of fine-tuning is removing the need for a lengthy system prompt describing the JSON schema on each query. Today, you might be feeding a big prompt like *“You are an assistant that outputs JSON with the following keys…”* every time. After fine-tuning, you can eliminate or shorten that. The model will have learned the instruction implicitly, which **improves inference speed** by cutting out hundreds of prompt tokens. As an example from Mistral’s guide: instead of including complex instructions about the output format in the prompt, they fine-tuned a model so that just providing the input (e.g. medical notes) was enough for it to output the structured JSON directly​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=)

. You should see the same effect – your assistant will produce well-formed JSON for a command without being reminded of the format each time. This makes responses faster and also reduces the chance of format errors.

**Continual Fine-Tuning:** Treat fine-tuning as an ongoing process. Start with an initial training round on a synthetic dataset of commands, and then **iteratively refine** the model as you encounter new phrases or edge cases. For example, if the assistant ever outputs malformed JSON or misunderstands a request, add that scenario (with the correct output) to your dataset. Because you’re using LoRA/QLoRA, you can re-train new adapters on top of the base model without starting from scratch. You could either *merge* the old LoRA and new data into a fresh adapter, or maintain a cumulative dataset and periodically fine-tune a new LoRA that supersedes the old one. Over time, this incremental approach will make the model more robust. Just be sure to monitor for any regressions (it’s wise to keep a validation set of some critical commands to ensure the model doesn’t drift). Continual fine-tuning in this manner allows your Jarvis to **learn from its mistakes and new tasks**, gradually expanding its capabilities while running locally.

**Step 3: Workflow Optimization – From Voice Command to Execution**

Your current pipeline is: **voice transcription → LLM → JSON → action execution**. This modular approach is actually quite sound, as it cleanly separates concerns (speech recognition, language understanding, and action handling). However, there are a few ways to optimize for efficiency:

* **Avoid Unnecessary LLM Calls:** Identify simple commands that don’t require the full power of the LLM to understand. For instance, if the user just says “open Safari” or “what time is it?”, you could bypass the model and trigger the action directly via a keyword match or regex. A small command parser or intent recognizer can catch these. This way the LLM is reserved for complex or ambiguous requests, reducing latency. You might implement a lightweight rules engine in Node-RED or a Python script to handle a set of known phrases instantly.
* **Direct Intent Parsing (Optional):** For a more formal approach, tools like **Voice2Json** or **Rhasspy** can perform *speech-to-intent* recognition offline. These systems use predefined grammar to map voice input directly to JSON intents. If your primary goal were purely home automation or computer control with a fixed command set, such a system could replace the LLM step. For example, voice2json is designed for “limited domain speech, where each sentence is a specific voice command”​

[news.ycombinator.com](https://news.ycombinator.com/item?id=27240653#:~:text=Author%20here,on%20May%2022%2C%202021)

– you’d define sentences and their target JSON, and it would output structured data in one step. The advantage is speed and reliability (no hallucinations, since it only recognizes known commands). The drawback is lack of flexibility – it won’t handle arbitrary phrasing or engage in back-and-forth conversation. In practice, you might combine approaches: use direct intent parsing for a handful of frequent commands and fall back to the LLM for everything else.

* **Efficient Prompt Design:** When you do invoke the LLM, keep the prompts terse. Leverage the fact that you fine-tuned the model to know the output format, so the prompt can be as short as: "User: <transcribed text>\nAssistant:" and nothing more. If you need the assistant to sometimes respond conversationally vs. output JSON, you can include a short flag in the prompt (or use separate invocation modes). For example, you might decide that any input beginning with “Jarvis,” triggers a conversational response, whereas direct commands produce JSON. Another strategy is to have the model **always output a JSON** with an action key, including an action like "action": "respond", "message": "text to say" for purely conversational replies. This uniform output can simplify the executor logic, though it slightly increases overhead to parse JSON even for a speak-only response. In any case, ensure the model isn’t doing extra work like apologizing or explaining the JSON – it should emit the minimal structure needed. Tight prompt design and schema enforcement will improve response speed and correctness.
* **Tokenization and Context:** Use the model’s context window wisely. Maintain only the necessary context between turns. For a task executor, you often don’t need a long conversation history in context (unlike a normal chatbot). This means you can clear the context or keep it very short for each new voice command, which saves memory and compute. If you *do* want some conversation memory (e.g. follow-up questions), consider limiting it to the last user query and assistant response. Reducing the number of tokens processed per query has a direct impact on speed.
* **Alternate Path – Programmatic Actions:** In some setups, developers have the LLM output *actual code or script* which is then executed (for example, outputting AppleScript or shell commands directly instead of JSON). This can skip the parsing step. However, executing free-form code from an LLM is risky and requires sandboxing. JSON as an intermediate is safer, because you can validate it before execution. Thus, the JSON approach is optimal unless you have very specific scripted sub-tasks that you trust the model to fill in.

Overall, the transcription→LLM→JSON→execute pipeline is quite optimal for a general AI assistant. By short-circuiting trivial commands and streamlining prompts, you minimize any unnecessary latency. The heaviest steps will be the speech-to-text and the LLM inference; using efficient models (e.g. **Whisper small or medium** for transcription, and the quantized Mistral for inference) will ensure each of those steps only takes a second or two on the M2. Any attempt to collapse the pipeline further (like end-to-end speech-to-action models) would require training a custom multi-modal model and likely be less reliable. So sticking with these discrete steps, while refining each, is the best approach.

**Step 4: Automation & GUI Control with Node-RED**

**Integrating Node-RED:** Node-RED will serve as the “brain” to orchestrate the components. It provides a flow-based GUI where you can wire together the microphone input, the AI model, and the action triggers. This visual approach makes it easy to add new commands or adjust logic without digging into code each time. For example, you can have a **flow** that starts when audio text is received: pass it into a node (function) that calls the LLM, then parse the JSON and route to different action nodes (open app, fetch info, etc.) based on the "action" field. Node-RED has a rich set of pre-built nodes and can call external programs or APIs. In fact, the integration of Node-RED with LLMs is becoming common, as it “streamlines the development process and accelerates time-to-market for AI-powered solutions”​

[ubos.tech](https://ubos.tech/news/node-red-and-large-language-models-integration-a-comprehensive-guide/#:~:text=Advantages%20of%20Node)

. You can simply use an HTTP request node or an Exec node in Node-RED to query your local model (for instance, hitting an Ollama local server or running a Python script that loadsllama.cpp). The response (JSON) can then be fed into a JSON node to convert it to a JavaScript object for downstream nodes.

**Fast and Intuitive Automation:** The benefit here is **extensibility** – if you want your assistant to do something new (say control a smart home device or send an email), you can drag in new nodes or write a small function and hook it into the flow, rather than changing the core assistant code. Node-RED is event-driven and ideal for connecting different systems. It’s also *live* programmable, meaning you can tweak your assistant’s behavior on the fly via the Node-RED editor. This makes developing your Jarvis fun and interactive. In terms of speed, Node-RED itself is lightweight. Just ensure that heavy tasks (like running the LLM) are done outside the Node-RED main thread – e.g., using the **exec node** to call a Python or C++ script for the model means Node-RED will wait for the result asynchronously, rather than blocking. This way, the GUI and other flows remain responsive. Node-RED essentially becomes the **conductor**, coordinating the wake word trigger, recording audio, invoking the STT/LLM, and then performing the action.

**Low-Level Control:** Even though Node-RED is a high-level tool, you won’t lose low-level control. You can embed custom code at any point (using a Function node for JavaScript logic, or triggering external scripts for Python/AppleScript/Swift, etc.). For instance, after getting the JSON, if the action is "execute\_script", you might use a Node-RED Exec node to run an AppleScript command on macOS. This gives you the full power of the system when needed. Node-RED basically allows you to mix and match – quick wiring for simple things, code where detailed control is needed. It also makes it easier to monitor and debug the assistant’s behavior, since you can insert debug nodes to see what JSON was produced or if an action was triggered. In summary, Node-RED will make your assistant’s **task automation fast to develop and easy to visualize**. The approach is to let Node-RED handle the flow logic and UI, while delegating AI inference and OS-specific scripting to dedicated nodes or external modules. This keeps the system both **intuitive** (via Node-RED’s GUI) and **flexible** (since you can always drop to code for complex tasks).

*(Tip: There are community Node-RED nodes for Whisper and even local LLMs that you might explore. For example, node-red-contrib-ml-rag includes a “local GPT” node for text generation​*

[*flows.nodered.org*](https://flows.nodered.org/node/node-red-contrib-ml-rag#:~:text=This%20module%20for%20Node,red)

*​*

[*flows.nodered.org*](https://flows.nodered.org/node/node-red-contrib-ml-rag#:~:text=about%20BM42%3A%20https%3A%2F%2Fqdrant.tech%2Farticles%2Fbm42%2F%20%2A%20rag,generation)

*, which could potentially load your quantized model through Hugging Face Transformers. If those work well, you might not even need a separate script – you could configure that node with your model and quantization and call it directly in the flow. Alternatively, using an HTTP endpoint (like an Ollama server or a Flask app wrapping the model) and Node-RED’s HTTP request node is a clean integration​*

[*ubos.tech*](https://ubos.tech/news/node-red-and-large-language-models-integration-a-comprehensive-guide/#:~:text=How%20to%20Integrate%20Node,Large%20Language%20Models)

*.)*

**Step 5: Wake Word Activation on macOS (Low Resource)**

To achieve a “Hey Jarvis” hotword without hogging memory or CPU, you should use a specialized wake-word engine rather than continuously running a full speech model. Here are the best options:

* **Porcupine by Picovoice:** Porcupine is a popular choice for always-listening wake word detection. It’s **highly optimized (very lightweight)** and known for accuracy​

[github.com](https://github.com/Picovoice/porcupine#:~:text=Picovoice%2Fporcupine%3A%20On,enabled%20applications)

. Porcupine offers pretrained wake words (like “Hey Google”, “Alexa”, etc.), and you can train a custom phrase like "Jarvis". On macOS, you can use Picovoice’s SDK (they have a Python and C API, and even Node.js bindings). Porcupine’s footprint is tiny – on the order of a few MB of RAM – and it runs efficiently on the CPU. This means you can have it running in the background constantly without impacting your system’s performance. The upside of Porcupine is it’s essentially plug-and-play and **very reliable** at detecting the wake word while filtering out false alarms. The only consideration is licensing: Porcupine is free for personal/non-commercial use (with access key registration), but not open-source. If that’s not an issue for you, Porcupine is arguably the easiest path. It will trigger your Node-RED flow (or whichever process) as soon as it hears “Jarvis” (or any chosen phrase), and then you can start recording audio for the assistant to process.

* **OpenWakeWord:** This is an **open-source wake word engine** developed by David Scripka, also focused on speed and ease of use​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=Home%20Assistant%E2%80%99s%20wake%20words%20are,of%20their%20own%20wake%20word)

. It has become the default wake word system in Home Assistant’s offline voice projects. OpenWakeWord provides pre-trained models for a number of common wake phrases and allows community training of new words. It runs on commodity hardware (like RPi 4) with a small footprint, using either the ONNX or TFLite runtime for neural inference​

[github.com](https://github.com/dscripka/openWakeWord#:~:text=Installation)

. The design goals are real-world accuracy and quick detection, using a simple model architecture that can be trained with synthesized data​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=openWakeWord%20is%20created%20with%204,goals%20in%20mind)

​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=openWakeWord%20is%20built%20around%20an,generate%20the%20wake%20word%20model)

. For “Jarvis,” you might find that OpenWakeWord already has a model (Home Assistant’s UI suggests “Hey Jarvis” was an option​

[reddit.com](https://www.reddit.com/r/homeassistant/comments/176hz85/year_of_the_voice_chapter_4_wake_words/#:~:text=Year%20of%20the%20Voice%20,able%20as%20the%20wake%20word)

). If not, training one is possible with a short audio dataset or even using their pipeline with a TTS like Piper to generate samples​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=openWakeWord%20is%20built%20around%20an,generate%20the%20wake%20word%20model)

. The memory and CPU usage of OpenWakeWord are minimal – Home Assistant notes they can run multiple instances without issue on a Pi 4​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=2,detection%20on%20an%20external%20server)

. The benefit here is that it’s completely local and free with no licensing friction. You can integrate OpenWakeWord by running a small Python process that listens to the microphone and emits an event when the wake word is heard. For instance, you could have it send an HTTP request or MQTT message to Node-RED to signal “Jarvis is awake.” Since it’s open-source, you can tweak sensitivity or other parameters to suit your environment.

* **Whisper or Speech-to-text based detection:** Using a general speech model like Whisper for wake word is **not recommended** if efficiency is a concern. While you could continuously run Whisper (especially a tiny model) and look for the word “Jarvis” in the transcriptions, this is far more CPU-intensive than dedicated wake word models. Whisper-small or medium would consume a large chunk of the 16GB RAM and significant CPU/GPU if run constantly. That said, a hybrid approach could be: run a very lightweight voice activity detector (VAD) to monitor audio, and only when speech is detected, use Whisper to transcribe and check if the first word was “Jarvis.” This avoids running Whisper when no one is speaking. Still, compared to Porcupine or OpenWakeWord, this approach will have more latency and power draw. It’s essentially re-purposing a full speech recognizer to do a job that a tiny keyword spotter can do more simply. Therefore, consider Whisper-based wake word only if you for some reason can’t use the above options (for example, if you wanted *any* arbitrary wake phrase without training a model – but even then, openWakeWord can be trained relatively easily). In practice, Porcupine or openWakeWord will do the job more efficiently and accurately for a fixed phrase.

**Implementation on macOS:** Both Porcupine and openWakeWord can run locally on macOS without issue. Porcupine’s Mac SDK comes with sample apps – you’d initialize the detector with the “Jarvis” keyword file and run a loop on the audio input stream. OpenWakeWord being Python-based means you can install it via pip (pip install openwakeword) and use its API to load a model and listen from the microphone. You might integrate them with Node-RED by having a separate **daemon thread or process** for wake word: for example, a small Python script that uses pyaudio to capture mic audio and checks the wake word. When detected, it could, say, send an HTTP POST to a Node-RED endpoint or simply use Node-RED’s websocket/tcp in node to signal the flow to start recording audio for the command. This separation is good because Node-RED isn’t real-time – you wouldn’t want Node-RED to continuously process raw audio. Let the wake word engine handle that in a tight loop, and only trigger Node-RED when needed.

**Memory Usage:** Both recommended wake word solutions are extremely light. Porcupine typically uses < 10 MB RAM and under 10% CPU of a single core in always-listening mode. OpenWakeWord’s resource usage is similarly low; it runs an ONNX model that is a few megabytes and the processing involves a short window of audio. The M2 has plenty of horsepower for either, and you can certainly run them alongside the LLM and other components without exceeding 16GB. By contrast, running Whisper continuously could easily use a few hundred MB or more and tax the CPU/GPU. So a dedicated wake listener is the way to go for an “always on” Jarvis that doesn’t drain your system.

**Summary:** By selecting an efficient 7B model (Mistral) and quantizing it for the M2, you ensure the assistant’s brain is both smart and fast. Fine-tuning with QLoRA to produce JSON outputs means the model will naturally respond in a structured way without heavy prompting​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=)

, which speeds up interactions. The overall voice command workflow can be kept lean by handling straightforward commands directly and using the LLM for the harder queries, with concise prompts and context management. Node-RED ties everything together in a maintainable fashion, giving you a **low-code interface** to add functionalities and connect to system-level scripts. Finally, a lightweight wake word engine (Porcupine or openWakeWord) provides snappy hands-free activation, only waking the heavier components when a command is actually incoming. Following this approach, you’ll have a local AI assistant that feels **responsive and efficient**, without the cloud or hefty rigs – essentially your own Jarvis running **entirely on your Mac**. Enjoy building it!

**Sources:**

1. Adyog – *Mistral 7B vs DeepSeek R1 Performance*​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Mistral%207B%3A%20Optimized%20for%20Speed,Efficiency)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=DeepSeek%20R1%3A%20Retrieval)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=,dynamically%2C%20but%20requires%20additional%20infrastructure)

​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Training%20Dataset%20Size%203,turn%20chat)

1. Adyog – *Mistral vs DeepSeek Benchmark Comparison*​

[blog.adyog.com](https://blog.adyog.com/2025/01/31/mistral-7b-vs-deepseek-r1-performance-which-llm-is-the-better-choice/#:~:text=Metric%20Mistral%207B%20DeepSeek%20R1,publicly%20benchmarked%20Not%20publicly%20benchmarked)

1. Reddit – *llama.cpp on Apple Silicon (M2 Max)*​

[reddit.com](https://www.reddit.com/r/LocalLLaMA/comments/140nto2/full_gpu_inference_on_apple_silicon_using_metal/#:~:text=Other)

1. E2E Networks – *Quantization Methods for Mistral 7B (GGUF vs GPTQ vs AWQ)*​

[e2enetworks.com](https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq#:~:text=GGUF%2C%20the%20successor%20of%20GGML%2C,approach%20for%20handling%20model%20files)

​

[e2enetworks.com](https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq#:~:text=Also%2C%20you%E2%80%99ll%20see%20the%20fluctuation,compared%20to%20the%20quantized%20models)

1. Mistral AI Documentation – *Fine-tuning for Specific Formats*​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=Use%20case%202%3A%20specific%20format)

​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=Fine,outputs%20in%20this%20specific%20format)

1. Mistral AI Documentation – *Removing Complex Prompts via Fine-Tuning*​

[docs.mistral.ai](https://docs.mistral.ai/guides/finetuning/#:~:text=)

1. Heidloff.net – *Memory-efficient Fine-tuning with QLoRA*​

[heidloff.net](https://heidloff.net/article/qlora/#:~:text=Memory,bit%20model%20finetuning)

1. Voice2Json HackerNews – *Voice2Json for Limited Domain Commands*​

[news.ycombinator.com](https://news.ycombinator.com/item?id=27240653#:~:text=Author%20here,on%20May%2022%2C%202021)

1. UBOS Tech – *Node-RED and LLM Integration Advantages*​

[ubos.tech](https://ubos.tech/news/node-red-and-large-language-models-integration-a-comprehensive-guide/#:~:text=Advantages%20of%20Node)

​

[ubos.tech](https://ubos.tech/news/node-red-and-large-language-models-integration-a-comprehensive-guide/#:~:text=How%20to%20Integrate%20Node,Large%20Language%20Models)

1. Node-RED ml-rag Documentation – *Local GPT Node for Text Generation*​

[flows.nodered.org](https://flows.nodered.org/node/node-red-contrib-ml-rag#:~:text=This%20module%20for%20Node,red)

​

[flows.nodered.org](https://flows.nodered.org/node/node-red-contrib-ml-rag#:~:text=about%20BM42%3A%20https%3A%2F%2Fqdrant.tech%2Farticles%2Fbm42%2F%20%2A%20rag,generation)

1. Home Assistant Blog – *openWakeWord for Wake Word Detection*​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=Home%20Assistant%E2%80%99s%20wake%20words%20are,of%20their%20own%20wake%20word)

​

[home-assistant.io](https://www.home-assistant.io/voice_control/about_wake_word/#:~:text=openWakeWord%20is%20built%20around%20an,generate%20the%20wake%20word%20model)

1. Picovoice GitHub – *Porcupine Wake Word Engine Description*​

[github.com](https://github.com/Picovoice/porcupine#:~:text=Picovoice%2Fporcupine%3A%20On,enabled%20applications)