3. **Set Up llama.cpp:** The script clones the `llama.cpp` repository from GitHub, compiles it with optimizations (making use of all CPU cores). This yields the `main` binary (and others like `convert.py`). We also download at least one small test model (like LLaMA 2 7B or an open variant) to ensure everything is working. The model files (GGML format) are stored in a designated directory (e.g., `/opt/llm/models`).
4. **Deploy Open WebUI:** The script fetches the Open WebUI project (possibly via Git or a Docker image). We have a preference for using a Docker container for Open WebUI for consistency. The script can install Docker if not present, then pull the `ghcr.io/<open-webui-image>` container. Alternatively, if not using Docker, it sets up a Python virtual environment, installs required packages, and configures the Open WebUI service. After setup, it will configure Open WebUI to know about the models installed (linking it to llama.cpp and Ollama backends).
5. **Install LM Studio:** LM Studio might come as an AppImage or installer. The script will download the latest release of LM Studio (for example, a .deb package or AppImage on Linux, or an MSI on Windows). On a headless server, LM Studio might not run, so this is more for clients who have a GUI system; however, if the server has a desktop environment, we place the LM Studio app for them. The script can also place a shortcut or instructions for connecting to it remotely (like via Remote Desktop or X forwarding).

After installation, the script performs basic configuration: it might create systemd services to auto-start the AI tools on boot. For instance, setting up **Ollama** to run as a background service listening on a certain port for API requests, or launching **Open WebUI** on boot so that the web interface is accessible at `https://<server-ip>:8000` internally. We also configure firewall rules so that these services are only reachable within the LAN or via VPN (no external exposure).

Usage Instructions: The document provides guidance on running the deployment and using the tools:

- To execute the deployment, an admin would copy the `On-Prem AI Deployment Scripts` archive to the target machine (which should be a Linux server or VM with sufficient resources). Unzip the archive, and run the main script: e.g. `bash deploy_llm_stack.sh`. The script will prompt for confirmation and then proceed. It logs its progress to a file (for troubleshooting). After completion, it will indicate success or any issues.
- **Post-Install Testing:** We include steps to verify each component:
- For **Ollama**: run `ollama run llama2` (assuming a model name "llama2" was downloaded) with a sample prompt to see if it returns an answer.
- For **llama.cpp**: run the `./main` binary on a test prompt file to ensure it works in pure CLI mode.
- For **Open WebUI**: open a browser on a client PC and go to the designated URL (the script will output something like `Open WebUI running at http://10.0.0.10:8000` for example). Ensure the interface loads and you can chat with the model.
- For **LM Studio**: if installed on a user workstation, launch it and confirm it detects the model files or can download models to the correct folder (the document offers a brief walkthrough of LM Studio's UI to load a local model and run a prompt).

We also emphasize resource requirements: Running large models can be hardware-intensive. The guide notes that for a full 70B parameter model, a high-end GPU and >100GB RAM might be needed, whereas smaller 7B or 13B models will run on a modest server (especially with int4 quantization). The script by default might download a smaller model (like a 7B) to ensure functionality, and we include instructions for the admin to download larger models manually if needed (since they might be copyrighted, we point them to where and how to obtain them, e.g., having the client log in to download LLaMA weights if they have access).

Privacy Considerations: All these AI tools run **completely offline**. None of the installed components will send data to external servers. We even configure them to not automatically check for updates without approval. This ensures that any proprietary or sensitive prompts/data the client uses with these AI models never leave the premises. We mention that maintaining this isolation is crucial: if the client later connects a tool to the internet (for model downloading or some plugin), they should be aware of the data implications. By default, our deployment is locked down – for example, if Open WebUI has a feature to pull community prompts or extensions, we disable it or advise caution.

Maintenance: The guide recommends periodic updates of the AI tools (since the AI field evolves rapidly). We provide a short script or instructions for updating: e.g., how to pull the latest llama.cpp and recompile, how to update Ollama (`ollama pull` if supported), etc. We also instruct how to add new models. The installed system comes with a directory structure (perhaps `/opt/llm_models`) and the user can drop new model files there and update config for Open WebUI or Ollama to register them.

By using the **On-Prem AI Deployment Scripts**, our clients can harness AI capabilities similar to cloud AI services, but entirely within their own secure environment. This empowers them to ask questions, analyze data, or automate tasks with AI while **ensuring data confidentiality** (no queries or responses are exposed to a third-party). The automation saves countless hours in setup and configuration, providing a ready-to-go AI environment with minimal technical effort.

Proxmox GPU Passthrough Guide

This guide offers a step-by-step walkthrough for enabling **GPU passthrough on Proxmox Virtual Environment (VE)**, specifically targeting NVIDIA GPUs (though many steps apply to other GPUs as well). By following this, one can dedicate a physical GPU installed in the Proxmox host to a virtual machine (VM), allowing that VM to use the GPU for graphics or computation as if it were directly attached. This is critical for workloads like AI model inference (as per our AI deployment) or other GPU-accelerated tasks, and even for running a full desktop VM with proper graphics.

1. Verify and Prepare the Hardware:

Ensure the Proxmox host machine supports virtualization extensions (Intel VT-x/VT-d or AMD-V/AMD-Vi for IOMMU). In the system BIOS/UEFI, **enable IOMMU** (often called Intel VT-d or AMD IOMMU) and also enable virtualization for CPU if not already. Enable “Above 4G Decoding” and “Resizable BAR” if those options exist, as they can help especially with GPUs that have large BAR memory (mostly for AMD GPUs or very large VRAM NVIDIA). Physically install the GPU in the server and boot Proxmox.

On the Proxmox host, check that the GPU is detected by the OS: for example, run `lspci | grep -E "VGA|NVIDIA"` and note the device ID (like `01:00.0` for the GPU and maybe `01:00.1` for the GPU’s audio function). Also, ensure no other processes (like a host GUI or a mining service) are using the GPU – our standard Proxmox installs are command-line only, so the GPU should be idle.