ChatGPT

GPL Licensing and Repository Structure

The core **llama-orch** orchestrator code is released under GPLv3 (see the LICENSE file) [1]. Because GPL is a "copyleft" license, any code *linked* into the GPL-covered code would normally also need to be GPL. To keep Stripe billing and usage-metering closed-source, those components must be in separate modules or services. In practice this means putting the Stripe integration and metering logic in a **distinct repository or process** so it isn't part of the GPL codebase. For example, you can deploy the metering as a separate microservice or Docker container that the GPLv3 orchestrator *calls* over a well-defined API. This "mere aggregation" approach – running the GPL code and a proprietary service as isolated programs – does **not** force your private code to be open-sourced [2]. In short: keep the metering/billing code out of the GPL repo, and treat it as an independent service or plugin.

Open-Sourcing DevOps vs. Private Metering

You can safely make the **infrastructure/deployment (DevOps) code open-source** so long as it does **not embed proprietary code**. For example, Kubernetes manifests, Terraform scripts, CI/CD pipelines, and containers for the open-source parts can all live in the public repo. The DevOps code can reference configuration values (API keys, endpoints, etc.) for the metering service, but it should not **contain** the metering logic. In practice, this means keeping the metering service's implementation (and any Stripe libraries) out of the open repo and instead injecting them via environment variables or separate secrets. This separation ensures that anyone can clone the open DevOps repo and deploy the core orchestrator on any NVIDIA-GPU cloud, but only your private environment includes the proprietary metering components. In short, *DevOps scripts can orchestrate deployment of both public and private components, but should not merge their codebases.*

- **Keep modules separate.** Follow a modular architecture (as seen in llama-orch) where the orchestrator, adapters, and other subsystems are distinct crates or services [3]. The **metering** function can live in its own module or service (in a private repo) that communicates over an API, decoupled from the open orchestrator code.
- **Clear API boundaries.** Define simple interfaces between the orchestrator and the metering service (e.g. REST calls, message queues). The orchestrator can emit usage events or request billing actions without exposing its internals. By treating metering as an "external" service, the GPL-covered orchestrator just invokes it like any other backend.

Hobbyist Deployment (GPU Requirements)

Yes, a hobbyist user **can deploy llama-orch on any cloud platform** as long as they have access to NVIDIA GPUs. The orchestrator is hardware-agnostic and runs as a service, and it plugs into LLM engine backends through *worker adapters*. For example, the project scaffolds adapters for GPU-accelerated engines like **llamacpp** and **vLLM** [3]. These engines require CUDA support (i.e. NVIDIA GPUs) to run large models efficiently. In practice, a user could provision a cloud VM or container with an NVIDIA GPU, install the orchestrator and one of the provided adapters (e.g. `worker-adapters-llamacpp-http` or `worker-`

`adapters-vllm-http`), and they would have a complete *agentic LLM API*. All of the deployment code (Dockerfiles, Helm charts, etc.) for this is open-source, so the user can customize it freely.

- **Choose NVIDIA GPUs.** Many LLM libraries (especially those for 13B+ models) rely on CUDA. Ensure your cloud instance has an NVIDIA GPU and drivers so that the worker adapter (llama.cpp, vLLM, etc.) can utilize hardware acceleration. The open-source adapters in `llama-orch` target these backends by default [3].
- **Agentic API access.** Once deployed, llama-orch provides an HTTP/SSE API for submitting tasks and streaming responses. A hobbyist on the cloud would use this same API (via the open-source SDK or CLI) to run "agentic" workflows. No special hardware beyond the GPU is needed for clients – they just need the orchestrator's endpoint.

Architectural Considerations ("Deep Architectural Decision")

Deciding how tightly to couple the DevOps layer, orchestrator, and metering is indeed a major design choice. **Loose coupling** is key. Ideally, treat each concern (orchestrator core, deployment/infra, metering/billing) as separate layers or services: this maximizes flexibility and keeps private parts isolated.

- **Modular design.** In llama-orch's architecture, functionality is split across crates (e.g. `orchestrator-core`, `orchestratord`, `worker-adapters`, etc.) [3]. You should similarly isolate the metering logic. For instance, deploy the orchestrator and worker adapters as one service, and run the metering service (with Stripe) as another, communicating over a network protocol. This way each service can be developed and scaled independently.
- **API vs Library.** Avoid compiling or linking the metering code into the GPL binary. Instead, let the orchestrator make HTTP or RPC calls to a billing endpoint. This turns the meter into a black-box service. From a license standpoint, this follows the "mere aggregation" model, since you're just *aggregating* separate programs at runtime [2].
- **Configuration and Secrets.** Keep all Stripe keys and proprietary configs out of the open repo. The open DevOps pipeline can reference a placeholder or environment variable (e.g. `STRIPE_API_KEY`) so that anyone can run the deployment, but only your private deployment injects the real secrets.
- **Scalability and Ownership.** Think about who "owns" each service. If metering is a separate team or has its own release schedule, it makes sense to fully isolate it. The orchestrator's DevOps can simply assume "some billing service" will consume its usage events.

In summary, decouple where possible. Have the orchestrator emit events or metrics, and let the metering service subscribe to them (or pull them) on its own schedule. This approach is common in microservice architectures: each service does one job and exposes a clear API. By keeping these boundaries sharp, you can open-source your orchestration and deployment code without entangling the proprietary billing logic.

Sources: The llama-orch architecture emphasizes modular, layered components [3]. Legal guidance (FSF/experts) confirms that separate processes communicating over an API are *not* considered a single "derivative work" under the GPL [2], so you can license them independently.

---

[1] LICENSE
https://github.com/veighnsche/llama-orch/blob/02728075dea5cdfe25b1ba75eb35b66117c85487/LICENSE

2 software - GPL Licensed code as separate process - Law Stack Exchange
https://law.stackexchange.com/questions/40420/gpl-licensed-code-as-separate-process

3 README.md
https://github.com/veighnsche/llama-orch/blob/02728075dea5cdfe25b1ba75eb35b66117c85487/README.md