ChatGPT

Peer-to-Peer GPU Cloud Platform Architecture

We propose a Replit-based marketplace where **GPU owners ("hosts")** register their devices and **renters** submit AI workloads. The platform is composed of modular components for the host agent, API backend, payments, front-end UI, and model repository, deployed with robust infrastructure and security.

1. GPU Host CLI Agent

- **Python CLI (cross-platform)** – Implement a CLI tool (e.g. using Typer or Click) that runs on the host's machine (Windows/Linux). On startup it authenticates to the backend (e.g. via an API token or short-lived JWT) and **registers the host and its GPU(s)**. It collects GPU details (NVIDIA model name, VRAM, current load, driver) via NVIDIA's management library (e.g. NVML) or `nvidia-smi` and sends these to the server.

- **Persistent WebSocket** – The agent maintains a single outbound WebSocket to the central server (no inbound ports open). All communication (new jobs, status updates, logs) flows over this persistent connection. This follows a common pattern of pushing events from server to host and back: e.g. the System Initiative team used "a persistent WebSocket connection that could push events such as function execution results" [1].

- **Signed Job Manifests** – When a renter submits a job, the backend sends the host a **job manifest** (JSON) describing the workload. Each manifest is signed by the server (using HMAC or an RSA private key) to ensure integrity. The agent verifies the signature (using the shared secret or server's public key) before execution, rejecting any tampered or unsigned jobs. In practice one can use HMAC-SHA256 for symmetric signing or RSA/ECDSA for asymmetric security (as discussed in JWT signing guides [2]).

- **Job Execution** – The agent either pulls a Docker image or unpacks a zipped code bundle. For Docker, it runs `docker run --gpus all ...` with NVIDIA Docker runtime to enable GPU access. For raw Python scripts, it can create a Python virtualenv or container. Resource limits (e.g. `--memory` for RAM, `timeout` for wall-clock time) are enforced. Upon process exit, the agent streams exit codes and artifacts back.

- **Real-Time Logging** – As the job runs, the agent streams logs line-by-line back to the backend over the WebSocket. For example, a FastAPI WebSocket endpoint can continuously send log lines to the client browser [3]. Implement an async loop that reads from the job's stdout/stderr and sends each line over the socket. The backend will relay these logs to the renter's dashboard in real-time.

- **Result Upload** – When the job finishes, the agent uploads result files (trained models, outputs) to an S3-compatible store. Use AWS S3 for production or MinIO for development. The agent can use Python's `boto3` or the `minio` library to PUT files. On success or failure, it notifies the backend (via a WebSocket message or HTTP callback) with the outcome and a link to results.

- **Heartbeats and Recovery** – The agent sends periodic "heartbeat" pings to the server (e.g. every 30 seconds). If the connection drops, it auto-reconnects. On reconnect it re-registers, re-sends any cached GPU info, and asks for pending work. Simple retries and exception handling ensure the agent recovers from transient errors.

This agent thus behaves like a worker node: it **registers securely**, only accepts authenticated jobs, runs Docker/Python with GPU support, and streams logs/results back over a single secure channel [1] [3].

2. Backend API (FastAPI)

- **FastAPI Framework** – We use FastAPI (Python) for the HTTP/WebSocket API. FastAPI automatically generates an OpenAPI schema and interactive docs (`/docs`) for all routes. The API is divided into user routes (authentication, profile), host routes (device registration, pricing), renter routes (job submission), and admin routes. PostgreSQL stores all metadata (users, devices, jobs, models) and Redis serves as a fast job queue/message broker (e.g. via Celery or RQ).

- **Authentication & Roles** – User accounts are managed via OAuth and JWT. Renters and hosts register via OAuth providers (Google/GitHub) or email/password. FastAPI provides built-in support for OAuth2/OpenID Connect flows, making it straightforward to integrate Google/GitHub login [4]. After OAuth, issue a JWT to the client. Roles (host vs renter vs admin) are recorded in the user record; route dependencies enforce role-based access.

- **Host Management** – Hosts call an endpoint (e.g. `POST /api/hosts/register`) to add a device, sending its unique ID, GPU specs (model, VRAM, driver version), pricing (price per GPU-hour), availability tags, and public key (for JWT or job verification). The backend saves this to PostgreSQL. Hosts can later update pricing/availability via API. An admin can view all hosts via the dashboard.

- **Job Submission** – Renters use an endpoint (e.g. `POST /api/jobs`) to submit a job. They include either:

- A ZIP archive of code + data, **plus** a command (e.g. `python train.py`).
- A Docker image reference (e.g. Docker Hub name) **plus** a runtime command.

The renter specifies resource needs (e.g. number of GPUs, memory). The backend stores the code or records the image URI, and enqueues a job record.

- **Scheduler** – A scheduler process pulls pending jobs from Redis and matches them to eligible hosts. Matching logic looks at GPU type, free capacity, price, and tags. For example, choose the cheapest host meeting the job's GPU/memory requirements and marked "available". The scheduler then sends the job manifest to that host's WebSocket channel and marks the job as "running". If no host is available, the job stays queued. A simple first-fit or priority queue can be used; advanced features (like spot pricing with bidding) can be added later.

- **Real-Time Updates** – The backend provides WebSocket endpoints for real-time updates. Renters' browsers connect (e.g. `/ws/job/{job_id}`) to receive status and log updates pushed from the worker agent. Similarly, hosts receive new job manifests over their persistent WebSocket. This live channel pattern is common in FastAPI (see example log-streaming

websocket [3]). Completed job artifacts (models, logs) are stored with S3 URLs; the renter UI fetches these from the backend.

- **Artifact Handling** – Completed jobs produce artifacts (trained model weights, logs, metrics). The agent uploads these to S3 and the backend records the URLs in the job record. The renter can then download results via a secure link (signed URL).

- **Admin Dashboard** – In addition to user dashboards, an admin interface (protected by admin role) lets staff review platform activity: all jobs history, hosts status/performance, and any disputes. The backend exposes APIs for reporting (jobs per day, host reliability, refunds processed).

- **Security (JWT & HTTPS)** – All API endpoints require HTTPS. Authentication uses JWT bearer tokens for API calls. The OpenAPI docs and FastAPI security utilities guide this setup. Integrating with OAuth2 providers like Google and GitHub is straightforward (these use OpenID Connect under the hood) [4].

- **OpenAPI Documentation** – FastAPI auto-generates an OpenAPI schema. We document every endpoint (request/response schemas), enabling tools (Swagger UI, ReDoc) for easy testing. This fulfills the requirement for documented REST API via OpenAPI.

3. Payments (Stripe)

- **Stripe Connect Marketplace** – We treat hosts as **connected accounts** and renters as customers. Using Stripe Connect, the platform can **collect payments and split funds** automatically [5] [6]. Each GPU host must onboard via Stripe Connect (usually Express or Custom account). The host's Stripe Account ID is stored in their profile.

- **Pricing & Billing** – Hosts set an hourly price on their device. When a renter's job completes, the backend calculates total cost = price × runtime. The renter's saved payment method (card) is charged that amount via a Stripe charge. Using Connect, we then **transfer 90%** of the charge to the host's Stripe account and keep a 10% platform fee (configurable). This split can be done using Stripe's "application_fee" or separate Transfers API.

- **Stripe Connect Flow** – Typical flow: the renter's browser initiates payment (`POST /api/pay`) once job is done. Backend creates a Stripe PaymentIntent on the connected host's account with an application fee. Stripe handles 3D Secure etc. On success, funds settle: host's account gets payout later, platform keeps fee.

- **Refunds and Failures** – If a job fails or renter cancels, the system automatically issues a refund via Stripe's API. The refund process reverts the transfer so the host isn't paid. Webhooks listen for payment failures or disputes; the backend then marks the job as refunded and releases any held funds.

- **Global Compliance** – Stripe manages KYC/PCI compliance. By using Connect, we ensure payments are secure and compliant (Stripe is PCI Level 1 certified). Our platform never handles raw card data; Stripe hosts the checkout.

This Connect-based setup is proven for marketplaces: “Collect payments from customers and automatically pay out a portion to sellers or service providers on your marketplace” [6] . It cleanly handles multi-party splits and currency flows for our GPU rentals.

4. Frontend (React + Tailwind)

- **Tech Stack** – Build the UI with React and Tailwind CSS for rapid, responsive development. We can use Create React App or Next.js. Tailwind provides utility classes to style layouts, forms, dashboards, and make the site mobile-friendly.

- **Landing Page** – A static home page explains features (community GPU pool, pricing, how it works) and includes calls-to-action to sign up.

- **Auth Pages** – “Sign In / Sign Up” pages use OAuth buttons (Google/GitHub). After login, the user is routed to either the Host or Renter dashboard based on role.

- **Host Dashboard** – Hosts see a dashboard summarizing their GPUs: current availability, pricing, and a log of recent jobs (successful, failed). Show total earnings and balance (based on Stripe payouts). Hosts can update each device’s price, availability (online/offline), and metadata tags (e.g. “RTX 4090, 24GB”).

- **Renter Dashboard** – Renters see a form to submit new jobs: select code ZIP or Docker image, specify command, resource needs, and an optional “Make public” toggle (for model publishing). After submission, a live status widget shows the assigned host, start/finish time, and streaming logs. On completion, a download button appears for the results/model.

- **GPU Browsing Page** – A public marketplace page lists all available GPU hosts, with filters (GPU model, VRAM, price range, tags). Users can search or sort by price/performance. Clicking a host shows detailed specs and uptime.

- **Public Models Section** – A “Model Zoo” page where users browse community-published models. Each model has metadata (name, description, author, citation info) and a download button. Users can fork or re-run these models on the cloud GPUs.

- **Real-time Logs & Notifications** – Use WebSocket (or a library like Socket.IO) on the front-end to receive log lines from the backend in real time (piped from the agent). Render logs in a scrollable box. Optionally use a toast system for notifications (job started/finished).

- **Admin UI** – Using the same React codebase, admins get extra pages showing job dispute resolution, host verification steps, and overall platform metrics (jobs per day, total spend).

This React/Tailwind UI is designed for ease of use. (Many modern admin dashboards use React+Tailwind.) Every API call from the frontend goes through JWT auth to the FastAPI backend.

5. Public Model Repository

- **User-Published Models** – After a job completes, if the renter checks “Make public”, the resulting model files (e.g. `.pt` or `.onnx`) are uploaded to S3 and a database record is created. A model

entry includes metadata: name, description, tags, size, citation (so others can credit the author), and download URLs.

- **Browsing & Download** – The frontend has a "Models" page showing thumbnails and descriptions of published models. Users can click to view full details and download files. Behind the scenes, an API endpoint (e.g. `GET /api/models`) returns JSON of available models and their S3 links.

- **Similar to Hugging Face Hub** – This is analogous to Hugging Face's model hub: a collaborative library of user-contributed models [7]. By providing a citation and versioning, we encourage reuse and credit. The API is RESTful and returns model cards in JSON (name, author, metrics, download link).

- **Retraining & Forking** – We can allow users to "fork" a public model: this duplicates the model record as a new job input so a user can further train or modify it on rented GPUs.

By integrating a model zoo, we add community value – users can share pre-trained networks and build on each other's work. The Hugging Face Hub (with millions of models) shows the importance of such a repository [7].

6. Infrastructure & Deployment

- **Databases** – Use PostgreSQL to store all relational data. Replit offers Postgres databases which can be used for development; production can use AWS RDS or a managed Postgres. Redis is used for the job queue and for caching.

- **S3 Storage** – For file storage, use MinIO (self-hosted S3-compatible) during development, and switch to AWS S3 (or DigitalOcean Spaces) in production. All job inputs/outputs and model files live in S3 buckets.

- **Replit Deployment** – We host the backend and frontend on Replit. Replit's new "Deployments" allow apps to run 24/7 with dedicated CPU/RAM [8]. We set the FastAPI backend to an Always-On repl with a custom `.replit` config. Tailwind's JIT build or PostCSS is configured in the React repl.

- **CI/CD** – Use GitHub Actions (or GitLab CI) to automate tests and deployment. For example, a GitHub Actions workflow can run on each push to `main`, build the React app, run Python tests, and then call the Replit API to trigger a deployment [9]. Secrets (like REPLIT API token) are stored in GitHub. This ensures any code merge auto-deploys to our Replit instances.

- **OpenAPI Docs** – The backend's API schema is automatically served at `/openapi.json` and a Swagger UI at `/docs` (FastAPI default). This satisfies the requirement for documented REST API with OpenAPI.

- **Containerization** – In production, each component (backend, frontend) runs in its own Replit container or instance. In a mono-repo, subfolders (or sub-repls) handle separation: e.g. `/backend`, `/frontend`, `/agent`. Each has its own README and config. For example, the backend's README details environment variables (DB URL, JWT secret, Stripe keys, S3 creds).

Thanks to Replit Deployments, our services are “always on” – the VMs rarely restart and don’t sleep [8]. This means once deployed, the API and web UI stay up without periodic pings.

7. Security & Networking

- **No Inbound Ports on Hosts** – GPU hosts never open public ports. They only make **outbound** connections to the central server’s WebSocket (over TLS). This avoids NAT/traversal issues. If a host is behind a very restrictive NAT or CGNAT, we can instruct the user to join a Tailscale/ZeroTier mesh network so the host gets a stable endpoint. In essence, all host-server communication is via the persistent WebSocket channel (or HTTPS polling as a fallback).

- **Signed Communications** – All job manifests and critical messages are signed. We use either shared HMAC secrets (fast to verify) or RSA keys. (Symmetric HMAC is simpler but requires storing secrets on both sides; RSA uses a keypair and is better for many clients [2].) Similarly, all API calls use JWT tokens signed by the server.

- **TLS Everywhere** – The backend uses HTTPS/WSS (TLS). Replit provides free HTTPS by default on `<project>.replit.app` domains. We store secrets (JWT signing key, database creds, Stripe API key) in environment variables so code never has raw keys in it.

- **Resource Limits** – In Docker invocations, we enforce `--gpus`, `--memory`, and `--cpus` flags. We also use `timeout` or the host OS to kill runaway processes (e.g. `timeout` Unix command or Docker’s `--stop-timeout`). Disk usage is sandboxed: each job runs in a clean container or a fresh working directory. This prevents any job from consuming more resources than paid for.

- **Job Isolation** – Each job runs in an isolated Docker container or virtualenv so that one renter’s code cannot affect another’s. GPUs are naturally partitioned by container.

- **Signature Verification** – The agent validates each job’s signature before execution. The security blog on JWT signing explains that symmetric (HMAC) uses a shared secret key, while asymmetric (RSA/ECDSA) uses public/private keys [2]. We likely generate a keypair on the server and give each agent the public key (or use a shared secret per agent).

- **Networking Instructions** – We will document in the README how to run the agent behind NAT. For example, installing Tailscale on the host creates a stable hostname (via the tailnet) to which the server can send commands. However, because we only need outbound, Tailscale is optional unless the user wants SSH access.

- **Platform Security** – The backend enforces authentication on every route and validates all inputs. CORS and rate-limiting are configured. Users can only see their own jobs/hosts (enforced by user ID checks in FastAPI dependencies).

This security model ensures a host device never exposes itself, only trusting signed instructions. The use of an always-on WebSocket and signed payloads is a known pattern for secure distributed execution [1] [2].

8. Optional Enhancements

- **Job Versioning** – Keep a history of job runs. Each submission gets a unique version ID or commit hash. Reruns of the same job can be linked to previous runs.

- **Email Notifications** – Use an SMTP service (SendGrid/AWS SES) to email users when jobs complete (or fail). This way renters get notified without watching the UI.

- **API Tokens** – In addition to OAuth, allow advanced users to create personal access tokens (like GitHub tokens) for scripting. These bearer tokens can be used with the REST API for submitting jobs without OAuth flows.

- **Scaling** – Though optional on Replit, we could containerize the scheduler and increase Redis/DB capacity as use grows.

By organizing the code into distinct repos (or a mono-repo with clear directories), each component (agent, backend, frontend, docs) has its own README and deployment instructions. The final deliverable on Replit will include a working demo mode (e.g. with a mocked GPU agent that simulates job execution) and full API specs.

Sources: We leverage FastAPI's features for auth and WebSockets [4] [3], Stripe Connect for marketplace payments [5] [6], and best practices for distributed task queues [1]. The design parallels decentralized GPU rental services (e.g. Nebula AI's P2P GPU marketplace) [10] and model-sharing platforms (e.g. Hugging Face Hub [7]) to create a secure, modular GPU cloud.

---

[1] NATS at System Initiative | Synadia
https://www.synadia.com/blog/system-init-nats-synadia

[2] HMAC vs. RSA vs. ECDSA: Which algorithm should you use to sign JWTs? — WorkOS
https://workos.com/blog/hmac-vs-rsa-vs-ecdsa-which-algorithm-should-you-use-to-sign-jwts

[3] Create a streaming log viewer using FastAPI
https://h3xagn.com/create-a-streaming-log-viewer-using-fastapi/

[4] Security - FastAPI
https://fastapi.tiangolo.com/tutorial/security/

[5] [6] Platforms and marketplaces with Stripe Connect | Stripe Documentation
https://docs.stripe.com/connect

[7] Hugging Face Hub documentation
https://huggingface.co/docs/hub/en/index

[8] Replit — Replit Deployments - the fastest way from idea → production
https://blog.replit.com/deployments-launch

[9] How to integrate a CI/CD pipeline with Replit for automated deployments? | Rapid Dev
https://www.rapidevelopers.com/replit-tutorial/how-to-integrate-a-ci-cd-pipeline-with-replit-for-automated-deployments

[10] Renting a GPU | Nebula AI
https://docs.nebulanetwork.ai/gpu-marketplace/overview/detailed-rental-guide/renting-a-gpu