The Infernet Protocol Book

The complete guide to decentralized GPU inference

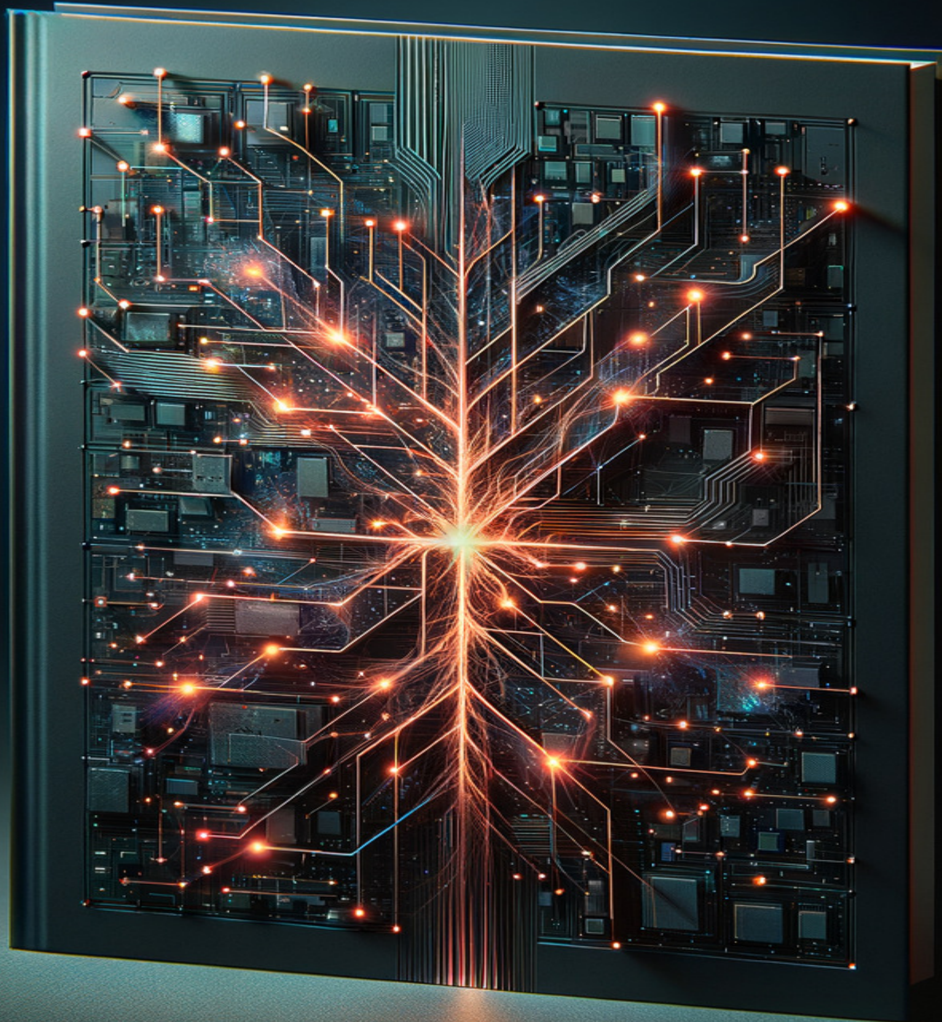
Infernet Protocol

2026

Infernet
Protocol

The Infernet Protocol Book

Open-Source Guide to Decentralized GPU Inference



The Infernet Protocol Book

Open-source guide to decentralized GPU inference

infernetprotocol.com

github.com/infernetprotocol/infernet-protocol

MIT Licensed • Pull requests welcome • 2026 Infernet Protocol

Contents

The Infernet Protocol Book 7
- Who This Book Is For . . . 7
- What You'll Learn . . . 8
- Quick Reference . . . 8
- Running the Examples . . . 8
- Contributing . . . 8

Chapter 1: Introduction 9
- In This Chapter . . . 9
- The Core Idea in One Paragraph . . . 9

Architecture 9
- Component Overview . . . 9
- Control Plane . . . 10
- Node Daemon . . . 10
- Inference Backends . . . 11
- Job Flow . . . 11
- Key Design Decisions . . . 11

Getting started 12
- Track 1 — Use the network for inference . . . 12
- Track 2 — Run a node and earn . . . 13
- Track 3 — Train a custom model . . . 14
- Track 4 — Publish to HuggingFace and Ollama . . . 16
- Stuck? . . . 16

Quick Start 17
- Prerequisites . . . 17
- Step 1: Install the CLI . . . 17
- Step 2: Get Your Registration Token . . . 17
- Step 3: Run Setup . . . 17
- Step 4: Start the Daemon . . . 18
- Step 5: Verify . . . 18
- Step 6: Send a Test Job . . . 19
- What's Next . . . 19

What Is Infernet Protocol? 19
- The Problem . . . 19
- What It Is . . . 20
- How It Works at a High Level . . . 20
- What Makes It Different . . . 20
- What It Is Not . . . 20

Chapter 2: Node Operators 21
- In This Chapter . . . 21
- The Operator's Day-to-Day . . . 21

Earnings 21
How Payments Work . . . 21
Checking Your Balance . . . 22
Setting Your Payout Address . . . 22
Claiming Earnings . . . 23
Payout Minimums . . . 23
Pricing Per Job . . . 23
Automated Payouts . . . 24
Payment Security . . . 24

Installation 24
Install the CLI . . . 24
Run Setup . . . 25
Firewall Configuration . . . 26
Running as a System Service . . . 27
Auto-Upgrade . . . 27
Upgrading the CLI Manually . . . 27
Uninstalling . . . 28

Model Management 28
How Served Models Work . . . 28
Installing Models via CLI . . . 28
Removing Models via CLI . . . 29
Listing Installed Models . . . 29
Installing Models via Dashboard . . . 29
Model Naming Conventions . . . 29
VRAM Planning . . . 30
Hot-Swap vs Cold-Load . . . 30

Monitoring Your Node 31
Status Overview . . . 31
Logs . . . 31
Doctor . . . 31
Dashboard Monitoring . . . 32
Heartbeat Intervals . . . 32
Alerting . . . 33

Hardware Requirements 33
GPU Tiers . . . 33
RAM . . . 34
Storage . . . 34
Bandwidth . . . 35
Operating System . . . 35
GPU Drivers . . . 35

Chapter 3: Inference Backends 36
In This Chapter . . . 36
How Auto-Selection Works . . . 36

Backend Adapter Interface . . . 36

Choosing a Backend **37**
Decision Guide . . . 37
Comparison Table . . . 37
Hardware-Specific Recommendations . . . 38
Switching Backends . . . 39
Benchmarking Your Setup . . . 39

llama.cpp **40**
llama-swap . . . 40
Installing llama.cpp . . . 40
Installing llama-swap . . . 40
Getting GGUF Models . . . 41
Starting llama.cpp Directly . . . 41
Configuring llama-swap . . . 41
Apple Silicon Performance . . . 42
CPU-Only Mode . . . 42
Key Environment Variables . . . 43
Infernet Config . . . 43
Systemd Service (llama-swap) . . . 43

Modular MAX **44**
Requirements . . . 44
Installing MAX . . . 44
Starting MAX Serve . . . 44
How MAX Differs . . . 45
Throughput Benchmarks . . . 45
Quantization . . . 45
Context Length . . . 46
Key Environment Variables . . . 46
Infernet Config . . . 46
Systemd Service . . . 46
Model Support . . . 47

Ollama **47**
Installing Ollama . . . 47
Pulling Models . . . 47
Hardware Support . . . 48
Key Environment Variables . . . 49
Infernet-Specific Config . . . 50
Troubleshooting . . . 50

SGLang **51**
Requirements . . . 51
Installing SGLang . . . 51
Starting SGLang . . . 51
RadixAttention: KV-Cache Reuse . . . 52

Speculative Decoding . . . 52
Structured Output . . . 52
Multi-GPU . . . 53
Key Environment Variables . . . 53
Infernet Config . . . 53
When SGLang Wins vs vLLM . . . 54
Systemd Service . . . 54

vLLM **54**
Requirements . . . 55
Installing vLLM . . . 55
Starting vLLM . . . 55
PagedAttention . . . 56
Multi-GPU with Ray (Tensor Parallelism) . . . 56
Key Environment Variables . . . 57
Infernet Config . . . 57
Performance Tuning . . . 57
Switching Between Models . . . 58
Systemd Service . . . 58

Chapter 4: Building Apps **58**
In This Chapter . . . 59
The Developer's View . . . 59
Quick API Test . . . 59

API Overview **60**
Base URL . . . 60
Authentication . . . 60
Core Endpoints . . . 60
Error Responses . . . 63
Rate Limits . . . 64

Job Lifecycle **64**
States . . . 64
Polling . . . 64
Polling vs Streaming . . . 66
Timeouts . . . 66
Retry Logic . . . 66
Error Reference . . . 67
Pending Time . . . 68
Batch Processing . . . 68

Streaming Chat **69**
SSE Event Format . . . 69
Opening a Stream . . . 70
JavaScript Example . . . 70
Python Example . . . 74
Handling Disconnections . . . 76

Done Event 76

Chapter 5: Protocol Internals **76**
In This Chapter 76
Design Philosophy 77

Nostr Key Hierarchy (IPIP-0028) **77**
Overview 77
User Keys 77
Node Keys 78
Model Keys (IPIP-0028) 78
Key Operations 79
NIP-44 Encryption (Advanced) 79

Payments **80**
Compute Payment Receipts (CPRs) 80
Payment Flow 80
Multi-Chain Support 81
On-Chain Contracts 82
Verifying Payments 82
Platform Fee 83

Security Model **83**
The Problem with API Keys 83
Keypair Generation 83
The X-Infernet-Auth Header 84
What This Means in Practice 85
Key Storage 85

Chapter 6: Advanced Topics **86**
In This Chapter 86
Who Needs These 86

Distributed Training (Roadmap) **86**
The Vision 86
What's Being Built 86
Expected Timeline 88
What You Can Do Now 88
Following Development 89

Multi-GPU Inference **89**
When You Need Multi-GPU 89
Tensor Parallelism with vLLM + Ray 89
Pipeline Parallelism 91
Multi-Machine with Ray 91
Infernet Config for Multi-GPU 92
Serving Multiple Models on Multiple GPUs 92
Performance Expectations 93

Self-Hosting the Control Plane **93**
What the Control Plane Is . . . 93
Prerequisites . . . 93
Option 1: Supabase Cloud + Deployed Next.js . . . 94
Option 2: Fully Self-Hosted (Supabase Local) . . . 95
Pointing the CLI at Your Control Plane . . . 95
Creating the First Admin Account . . . 96
Private Network Configuration . . . 96
Payment Setup for Self-Hosted . . . 96
Upgrading . . . 97

End-to-end training pipeline **97**
The four steps . . . 97
1. Crawl training data from a search query . . . 97
2. Scaffold a training config . . . 98
3. Run the fine-tune . . . 98
4. Publish . . . 99
Variants . . . 99
Open-market training (IPIP-0030) . . . 99
Prerequisites . . . 100

The Infernet Protocol Book

Infernet Protocol is a decentralized GPU compute network. Node operators register their GPU servers and get paid to run LLM inference. Developers submit jobs through a unified API and get responses back without depending on any single provider.

This book covers everything you need to know to run a node, build applications on the network, or understand how the protocol works under the hood.

Who This Book Is For

Node operators who want to earn crypto by contributing GPU compute. You have an NVIDIA, AMD, or Apple Silicon machine and want to put it to work. Start with Chapter 2: Node Operators.

Application developers who want to call LLM inference without locking into OpenAI, Anthropic, or any other centralized provider. You want reliable APIs, streaming responses, and predictable costs. Start with Chapter 4: Building Apps.

Protocol contributors interested in the cryptographic architecture, payment flows, and key hierarchy. Start with Chapter 5: Protocol.

If you're new to Infernet entirely, read Chapter 1: Introduction first.

What You'll Learn

Chapter 1 — Introduction What Infernet Protocol is, the problem it solves, and a high-level architecture tour. Includes a 5-minute quickstart so you can see the system working before diving into details.

Chapter 2 — Node Operators Hardware requirements, the full installation walkthrough, model management, monitoring your node, and how earnings and payouts work.

Chapter 3 — Inference Backends Ollama, vLLM, SGLang, Modular MAX, and llama.cpp. How each one works, when to use it, and how Infernet auto-selects between them.

Chapter 4 — Building Apps The REST API, streaming chat with SSE, job lifecycle management, and error handling. JavaScript and Python examples throughout.

Chapter 5 — Protocol Internals The Nostr-style secp256k1 auth system, Compute Payment Receipts, multi-chain wallet support, and the IPIP-0028 model key hierarchy.

Chapter 6 — Advanced Topics Multi-GPU setups with vLLM and Ray, self-hosting the control plane, and the distributed training roadmap.

Quick Reference

Task	Where to look
Install a node	02-node-operators/installation.md
Pick an inference backend	03-inference-backends/choosing.md
Stream tokens from the API	04-building-apps/streaming-chat.md
Understand auth headers	05-protocol/security.md
Run a 70B model on 2 GPUs	06-advanced/multi-gpu.md

Running the Examples

Most code examples in this book assume:

- `INFERNET_NODE_URL` points to your node (e.g. `http://localhost:3000`)
- `INFERNET_BEARER_TOKEN` is set with a valid bearer token from your control plane

```
export INFERNET_NODE_URL=http://localhost:3000
export INFERNET_BEARER_TOKEN=your_token_here
```

For the CLI examples, `infernet` must be installed and on your PATH. See installation if it isn't.

Contributing

This book is open source and lives in `docs/book/` in the main Infernet Protocol repository. Pull requests are welcome. Corrections, new examples, and translations are especially appreciated.

Chapter 1: Introduction

This chapter explains what Infernet Protocol is, why it exists, and how the pieces fit together. By the end you'll have a mental model of the whole system and a working node (or API call) to prove it.

In This Chapter

- What Is Infernet Protocol? — The problem with centralized AI and how a decentralized compute network solves it.
- Architecture — Control plane, node daemon, inference engines, and the job flow from request to payment.
- Quick Start — Install the CLI, run setup, and verify your node is online in under 5 minutes.

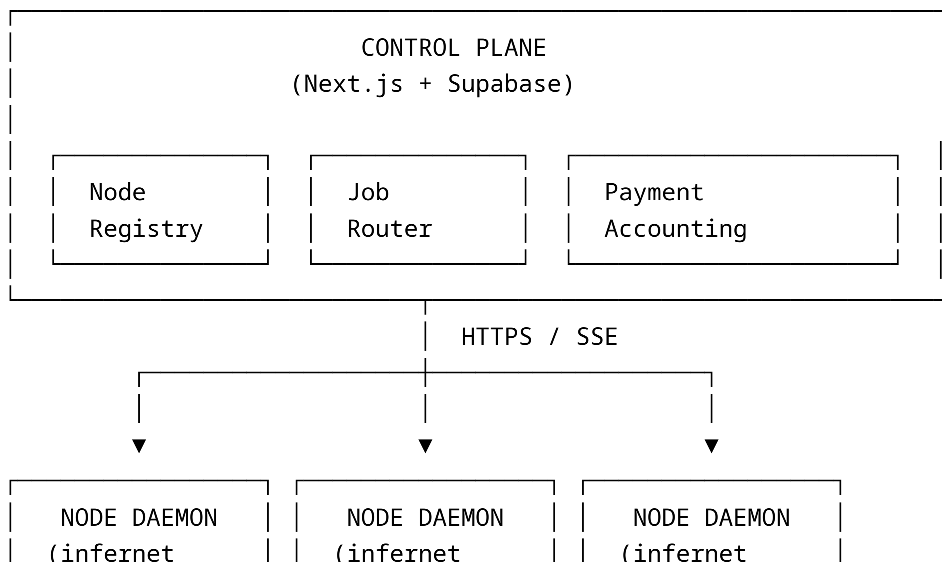
The Core Idea in One Paragraph

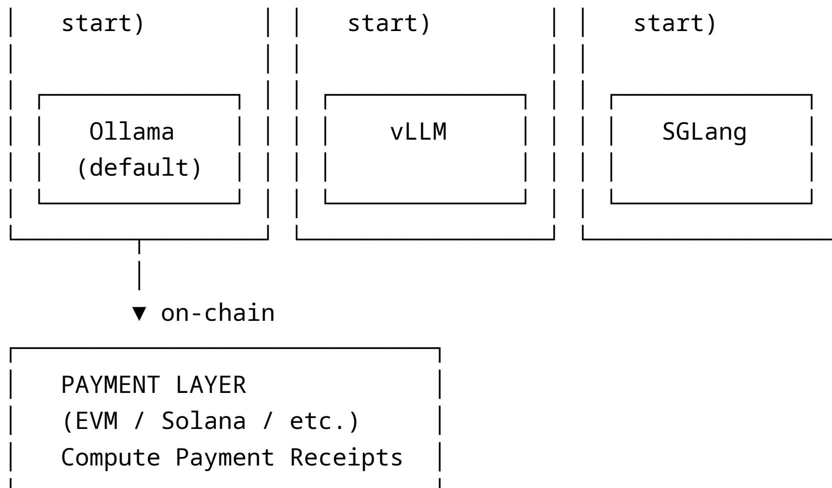
Anyone with a GPU can run a node. Anyone who needs LLM inference submits a job. The network routes the job to an available node with the right model loaded, the node runs inference and streams back the result, and the client's payment is settled on-chain. No single company controls routing, pricing, or what models are available. The cryptographic auth layer means nodes never need to trust the control plane with private keys.

Architecture

Infernet Protocol has four major components: the control plane, node daemons, inference backends, and the on-chain payment layer. Here's how they connect.

Component Overview





Control Plane

The control plane is a Next.js application backed by Supabase. It serves two audiences:

The dashboard — a web UI for operators to see their nodes' status, earnings, model inventory, and recent jobs. Clients can also use the dashboard to browse available models and manage API access.

The API — the REST endpoints that clients call to submit jobs and that node daemons call to register, heartbeat, and poll for commands. The primary job submission endpoint is `POST /api/v1/jobs`.

Supabase handles persistence: node registrations, job records, payment accounting, and the command queue (model installs/removes). Supabase Realtime is used for live dashboard updates.

The control plane is open source. You can self-host it — see Chapter 6: Self-Hosting.

Node Daemon

The node daemon is the process started by `infernet start`. It does several things:

Heartbeat loop — every 30 seconds, the daemon sends a signed heartbeat to the control plane. The heartbeat includes: node public key, IP address, port, GPU stats, which models are currently loaded, and current load. If the control plane doesn't hear from a node for 90 seconds, it marks the node offline.

Command polling — on each heartbeat cycle, the daemon checks a command queue in the control plane. Commands include `model_install` (pull a new model) and `model_remove` (evict a model). Operators issue these commands from the dashboard or CLI; the daemon picks them up and executes them.

Job execution — when a job is routed to the node, the daemon receives it, calls the local inference backend, and streams the result back.

Auth — every request the daemon makes is signed with the node's secp256k1 private key. The signature covers the HTTP method, path, body hash, a nonce, and a timestamp. This is carried in the `X-Infernet-Auth` header. The control plane verifies the signature against the node's registered public key. The private key never leaves the node.

Inference Backends

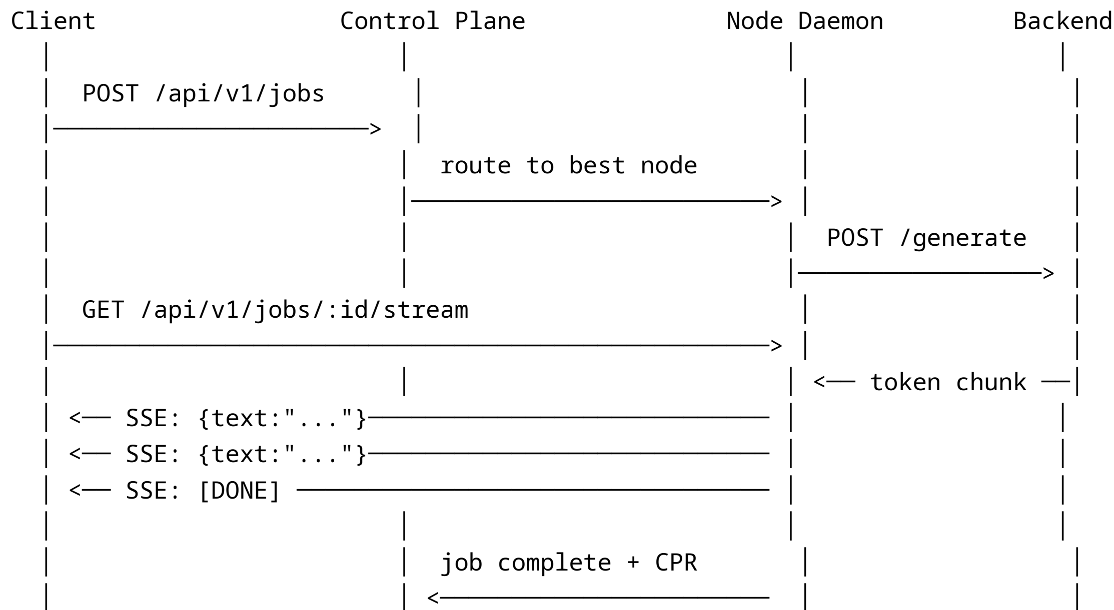
The daemon doesn't run inference itself. It delegates to one of five supported backends:

Backend	Best for	Protocol
Ollama	General use, easy setup	REST at `localhost:11434`
vLLM	High-throughput NVIDIA	OpenAI-compatible REST
SGLang	KV-cache reuse, structured output	OpenAI-compatible REST
Modular MAX	Throughput, modern NVIDIA	REST at configurable port
llama.cpp	CPU, Apple Silicon, GGUF models	REST via llama-swap

The daemon probes each backend in priority order at startup and uses the first one that responds. You can override the selection with env vars.

Job Flow

Here's what happens from the moment a client submits a job to the moment they receive the last token:



The client can poll `GET /api/v1/jobs/:id` for status or open an SSE stream for real-time token delivery. Most applications use the stream.

Key Design Decisions

Supabase as the coordination layer — not a custom blockchain. Job routing, node registry, and command queuing run on Postgres with real-time capabilities. This keeps latency low and the stack familiar. On-chain components handle only what needs to be on-chain: payment settlement.

Nodes never trust the control plane with keys — the secp256k1 auth model means the control plane can verify node identity without ever holding private keys. A compromised control plane cannot impersonate a node or steal earnings.

Backends are swappable — the daemon speaks to all backends through a thin adapter layer. Adding a new backend is a matter of implementing the adapter, not changing the protocol.

SSE for streaming — Server-Sent Events are simpler than WebSockets for unidirectional streaming and work through most proxies and CDNs without special configuration.

Getting started

Four ways to use Infernet Protocol. Each section is a copy-paste sequence that finishes in about five minutes. Pick the path that matches what you want to do today.

Track	Who it's for	Outcome
1. Use the network	Developers, end users	Working chat completion call against the public endpoint
2. Run a node	Operators with a GPU	Daemon registered, accepting jobs, earning crypto
3. Train a custom model	Builders, researchers	Fine-tuned LoRA from a search query → ready-to-publish model
4. Publish a model	Whoever just trained one	Model live on huggingface.co AND ollama.com

---

Track 1 — Use the network for inference

The public endpoint is OpenAI-compatible. If you've ever called OpenAI's API, the only thing that changes is the `base_url`.

curl

```
curl https://infernetprotocol.com/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "qwen2.5:7b",
        "messages": [{"role": "user", "content": "What is Bitcoin?"}],
        "stream": false
    }'
```

Python

```
from openai import OpenAI

client = OpenAI(
    base_url="https://infernetprotocol.com/v1",
    api_key="no-key-needed-for-playground"
```

```
)

stream = client.chat.completions.create(
    model="qwen2.5:7b",
    messages=[{"role": "user", "content": "Who built Linux?"}],
    stream=True,
)
for chunk in stream:
    print(chunk.choices[0].delta.content or "", end="", flush=True)
```

JavaScript / Node

```
import OpenAI from "openai";

const client = new OpenAI({
    baseURL: "https://infernetprotocol.com/v1",
    apiKey: process.env.INFERNET_API_KEY ?? "no-key-needed-for-playground"
});

const res = await client.chat.completions.create({
    model: "qwen2.5:7b",
    messages: [{ role: "user", content: "Explain Schnorr signatures in 3 lines." }]
});
console.log(res.choices[0].message.content);
```

The browser-friendly playground is at https://infernetprotocol.com/chat.

Track 2 — Run a node and earn

One curl command bootstraps everything: installs the CLI, sets up Ollama, opens the firewall port, drops a systemd unit, and registers your node with the control plane.

```
curl -fsSL https://infernetprotocol.com/install.sh | sh
```

Then:

```
infernet init              # generates a Nostr keypair, picks defaults
infernet setup             # installs Ollama + a starter model + opens the firewall port
infernet register          # signs and announces your node to the network
infernet start             # starts the daemon (or use `infernet service enable`)
```

Your node is now live. Configure where to send earnings:

```
infernet payout set --coin BTC --address bc1q...
infernet payout set --coin USDC --address 0x... --network arbitrum
infernet payout list
```

Quality-of-life commands you'll use daily:

```
infernet status                          # daemon health + last-seen
infernet model recommend --install-all   # auto-install best models for your VRAM
infernet uncensored                      # one-shot install of Hermes 3 / Dolphin
infernet logs -f                         # tail the daemon log
infernet upgrade                         # pull the latest CLI
```

The full operator guide is Chapter 2.

Track 3 — Train a custom model

Same shape as `rockypod/svelte-coder`: pick a topic, crawl the web for content, fine-tune a base model on the result. Run it on your one GPU or fan out across all the nodes you own.

1. Crawl a search query into a dataset

```
infernet train data \
    --query "svelte 5 framework documentation" \
    --domains svelte.dev,kit.svelte.dev,github.com \
    --num 30 \
    --out ./data/svelte5.jsonl
```

Needs `VALUESERP_API_KEY` in env or under `integrations.valueserp.api_key` in `~/.config/infernet/config.json`. Output: a ChatML-format JSONL with ~hundreds of training examples extracted from the top web results.

2. Scaffold a config

```
infernet train init --output ./run
```

Edit `run/infernet.train.yml`:

```yaml
name: svelte5-coder
base_model: unsloth/Qwen2.5-Coder-7B-Instruct
method: qlora
runtime: unsloth
workload_class: C1            # C1 local · C2 sweep · C3 federated

input:
  dataset: ./data/svelte5.jsonl
  format: chatml

training:
  epochs: 3
  learning_rate: 2.0e-4
  batch_size: 4
  max_seq_len: 4096

lora:
```

```yaml
  rank: 16
  alpha: 32
  target_modules: [q_proj, k_proj, v_proj, o_proj]

resources:
  min_vram_gb: 24
```

3a. Train locally (single GPU)

```
infernet train run --local --config ./run/infernet.train.yml
```

One-time install of the Python deps the runner shells out to:

```
pip install unsloth datasets trl
```

3b. Train on the open network (federated LoRA — experimental)

Pay any opted-in operator on the network — not just nodes you own — to train shards. Your local `infernet` daemon hosts the dataset shards directly over its existing reachable port; no S3, no HuggingFace, no IPFS, no third-party storage anywhere.

```
infernet train run --open-market \
    --config ./run/infernet.train.yml \
    --budget 5.00 \
    --max-nodes 8
```

What happens:

1. The CLI splits your local JSONL into 8 shards under `~/.infernet/training-runs/<run_id>/shards/` and mints a per-run upload token.
2. Posts a job to `/api/v1/training/jobs` with `dataset_base_url` pointing at your own daemon (`<your-endpoint>/v1/training/shards/<run_id>/shard-N.jsonl`).
3. Operators with `INFERNET_ACCEPT_TRAINING=1` poll the market every 60s, race-claim shards, fetch directly from your daemon, run the same Unsloth runner used in 3a, then PUT the resulting LoRA adapter to your daemon (auth via the upload token the control plane handed them).
4. Adapters land in `~/.infernet/training-runs/<run_id>/adapters/`.
5. You FedAvg the 8 adapters once all shards report.

If your machine is behind strict NAT (rare for rented GPU boxes, common for residential), expose port 8080 via:

```
cloudflared tunnel --url http://localhost:8080
export INFERNET_DAEMON_ENDPOINT=https://<the-cloudflared-url>
```

The control plane only ever sees URLs — never your dataset bytes.

Output: `./run/checkpoint-final/` — a HuggingFace-shape directory ready for Track 4.

Track 4 — Publish to HuggingFace and Ollama

One command, two destinations. The fine-tune lands at `huggingface.co/<org>/<name>` AND `ollama.com/<user>/<name>`.

```
infernet publish ./run/checkpoint-final \
    --hf InfernetProtocol/svelte5-coder \
    --ollama infernet/svelte5-coder \
    --quant q4_k_m
```

What runs under the hood:

1. `huggingface-cli upload` pushes safetensors to HF.
2. `convert_hf_to_gguf.py` from your local `llama.cpp` checkout converts to f16 GGUF.
3. `llama-quantize` quantizes to Q4_K_M (or your `--quant`).
4. An auto-generated `Modelfile` with the ChatML template is written.
5. `ollama create` then `ollama push` ships it to ollama.com.

One-time prerequisites:

```
# llama.cpp for the GGUF convert
git clone https://github.com/ggml-org/llama.cpp ~/llama.cpp
cd ~/llama.cpp && cmake -B build && cmake --build build -j

# HF token with write scope on your target org
export HUGGINGFACE_TOKEN=hf_...

# Ollama signin (once)
ollama signin
```

After publish, anyone with Ollama can pull your model:

```
ollama pull infernet/svelte5-coder
ollama run infernet/svelte5-coder "How do runes work in Svelte 5?"
```

Variants

- `--skip-hf` — Ollama only
- `--skip-ollama` — HuggingFace only
- `--modelfile-only` — generate the Modelfile + GGUF locally, don't push anywhere

---

Stuck?

- Troubleshooting for common failure modes
- Discord for real-time help
- GitHub issues for bugs
- `hello@infernetprotocol.com` for everything else

Quick Start

Get a node online in under 5 minutes. This walkthrough installs the CLI, runs setup, and verifies the node is heartbeating to the control plane.

Prerequisites

- Linux (Ubuntu 22.04+ recommended) or macOS
- A GPU (or CPU-only for testing)
- `curl` available on PATH
- An account on the Infernet dashboard to get your registration token

If you're on a GPU machine, make sure your GPU drivers are installed before starting. The setup wizard will detect the GPU, but it can't install drivers for you.

Step 1: Install the CLI

```
curl -sSL https://infernetprotocol.com/install | bash
```

This script installs the `infernet` binary to `~/.local/bin` (or `/usr/local/bin` if you have write access) and adds it to your PATH. It takes about 30 seconds.

Verify the install:

```
infernet --version
# infernet 0.9.2
```

Step 2: Get Your Registration Token

Open the Infernet Dashboard, sign in, and navigate to **Nodes** → **Add Node**. Copy the one-time registration token shown on that page. It looks like:

```
inft_reg_7x9k2mNpQwRsLvTbY4cJ
```

Step 3: Run Setup

```
infernet setup
```

The setup wizard will:

1. Ask for your registration token
2. Detect your GPU (NVIDIA, AMD, Apple Silicon, or CPU)
3. Determine your VRAM tier (`>=48gb`, `>=24gb`, `>=12gb`, `>=8gb`, or `cpu`)
4. Install Ollama if no inference backend is detected
5. Ask which default model to load (defaults to `qwen2.5:7b` for >=8gb, `qwen2.5:1.5b` for cpu)
6. Generate a secp256k1 keypair for this node
7. Register the node with the control plane
8. Write the config to `~/.infernet/config.json`

Example output:

```
Infernet Node Setup
===================
Registration token: inft_reg_7x9k2mNpQwRsLvTbY4cJ

Detecting hardware...
  GPU: NVIDIA RTX 4090 (24 GB VRAM)
  Tier: >=24gb
  RAM: 64 GB

Checking inference backends...
  Ollama: not found
  Installing Ollama...  done

Default model [qwen2.5:14b]:
Pulling qwen2.5:14b... ████████████████████ 100% (8.1 GB)

Generating node keypair...
  Public key: npub1abc123...
  Config written to ~/.infernet/config.json

Registering node...
  Node ID: node_8f3a2c1d
  Status: registered

Setup complete. Run `infernet start` to bring your node online.
```

Step 4: Start the Daemon

```
infernet start
```

The daemon starts in the foreground by default. You'll see heartbeat logs:

```
[2026-04-30 14:23:01] Node node_8f3a2c1d starting...
[2026-04-30 14:23:01] Inference backend: ollama (localhost:11434)
[2026-04-30 14:23:01] Loaded models: qwen2.5:14b
[2026-04-30 14:23:02] Heartbeat OK (latency: 42ms)
[2026-04-30 14:23:32] Heartbeat OK (latency: 39ms)
```

To run it in the background:

```
infernet start --detach
```

Or install it as a system service that starts on boot:

```
infernet service install
```

Step 5: Verify

Check the node is online from the CLI:

```
infernet status
```

```
Node:     node_8f3a2c1d
Status:   online
Uptime:   3 minutes
Backend:  ollama
Models:   qwen2.5:14b
Jobs:     0 completed, 0 pending
Earnings: 0.00 USDC
```

You should also see the node appear as **Online** in the dashboard within 30 seconds of starting the daemon.

Step 6: Send a Test Job

You can send a test inference job directly from the CLI:

```
infernet chat "What is the capital of France?"
```

```
Paris.
```

Or with streaming visible:

```
infernet chat --stream "Explain how neural networks learn."
```

Tokens will appear as they're generated.

What's Next

- **Node operators**: Read Chapter 2 for firewall setup, model management, monitoring, and earnings.
- **Developers**: Read Chapter 4 for the REST API, SSE streaming, and error handling.
- **Want a different backend?** Read Chapter 3 to swap Ollama for vLLM, SGLang, or MAX.

What Is Infernet Protocol?

The Problem

Modern AI applications depend almost entirely on a handful of centralized API providers. When OpenAI has an outage, thousands of products break. When Anthropic changes its pricing, startups scramble. When a model gets deprecated, months of prompt engineering disappears overnight.

Beyond reliability and cost, there's a structural problem: the compute that runs these models is controlled by a small number of large cloud providers. GPUs are expensive. If you have one, there's currently no straightforward way to monetize it by running inference for others. If you need inference, you're stuck paying whatever the incumbent providers charge.

Infernet Protocol is built to fix both sides of this.

What It Is

Infernet Protocol is a decentralized GPU compute network for LLM inference. It has two kinds of participants:

Node operators run GPU servers with inference software installed. They register their nodes on the network, keep models loaded, and accept inference jobs. They earn crypto payments per job completed.

Clients submit inference jobs through a unified API. Jobs are routed to available nodes, executed, and results are streamed back. Clients pay per job. They don't need to know which node ran their job.

The network is coordinated through a control plane (a Next.js app backed by Supabase) that handles node registration, job routing, and payment accounting. Crucially, the control plane never holds private keys. All auth uses Nostr-style secp256k1 keypairs where nodes sign their own requests.

How It Works at a High Level

1. **A node operator installs the CLI** and runs `infernet setup`. This generates a secp256k1 keypair, detects the GPU, installs an inference backend (Ollama by default), and registers the node on the network.
2. **The daemon starts** with `infernet start`. It heartbeats to the control plane every 30 seconds, reporting which models are loaded, GPU utilization, and availability.
3. **A client submits a job** via `POST /api/v1/jobs`. The control plane routes the job to a suitable node based on model availability, capacity, and proximity.
4. **The node runs inference** using whichever backend is installed (Ollama, vLLM, SGLang, MAX, or llama.cpp). Results stream back to the client via Server-Sent Events.
5. **Payment is settled**. The node's wallet receives a Compute Payment Receipt (CPR) that can be redeemed on-chain. Operators run `infernet payout` to claim earnings.

What Makes It Different

No vendor lock-in. The API is consistent regardless of which backend is running inference. You can switch from Ollama to vLLM by changing an env var without changing your client code.

Censorship resistance. Because any operator can join and any model can be served, the network isn't dependent on any single company's content policies or business decisions.

Real hardware. Jobs run on real GPUs owned by real people, not virtual machines. Operators who invest in better hardware can handle larger models and charge more.

Crypto-native payments. Payments are settled on-chain with no intermediary holding funds. Operators get paid directly into their wallets.

Open source. The CLI, control plane code, and protocol spec are all open source. You can self-host the entire stack if you want full independence.

What It Is Not

Infernet Protocol is not a model hosting service. It doesn't store model weights. Operators pull models onto their own machines and serve them from local storage.

It's not a training platform (yet — see Chapter 6 for the roadmap).

It's not a cloud provider. There are no SLAs, no guaranteed uptime per node, and no managed infrastructure. The network's reliability comes from having many nodes, not from any individual node being highly available.

Chapter 2: Node Operators

Running a node is how you contribute compute to the network and earn crypto. This chapter covers everything from hardware selection through daily operations.

In This Chapter

- Requirements — GPU tiers, RAM, bandwidth, and OS requirements. What hardware you can use and what to expect from each tier.
- Installation — Full install walkthrough: the curl installer, `infernet setup`, firewall rules, and running the daemon as a system service.
- Model Management — Installing and removing models via the CLI and dashboard. How the daemon serves models and polls for updates.
- Monitoring — `infernet logs`, `infernet status`, `infernet doctor`, and what the dashboard shows you about your node's health.
- Earnings — How payments work per job, how to check your balance, and how to run `infernet payout` to claim earnings.

The Operator's Day-to-Day

Once your node is set up, operating it is mostly passive. The daemon runs in the background, accepts jobs automatically, and earns payments without your involvement. The main active tasks are:

- **Managing your model inventory**: adding new models when clients request them, removing models you're not getting jobs for to free VRAM.
- **Monitoring node health**: checking logs when something looks wrong, running `infernet doctor` when the node goes offline unexpectedly.
- **Claiming earnings**: running `infernet payout` periodically to move accumulated earnings to your wallet.
- **Keeping software current**: running `infernet upgrade` when new CLI versions are released.

The rest of this chapter covers each of these in detail.

Earnings

How Payments Work

Every completed inference job generates a **Compute Payment Receipt (CPR)**. A CPR is a signed record that proves your node completed a specific job: it includes the job ID, token count, model used, your node's public key, the client's public key, and an on-chain settlement value.

CPRs accumulate in your node's account on the control plane. When you run `infernet payout`, the pending CPRs are aggregated into an on-chain transaction that transfers the earned amount to your payout wallet.

Checking Your Balance

```
infernet status
# Shows: Earnings: 12.84 USDC (unclaimed)
```

For more detail:

```
infernet payments
```

```
Earnings Summary
================
Unclaimed balance:  12.84 USDC
Paid out (lifetime): 847.22 USDC

Recent jobs (last 24h):
  job_9a3f2c1d  qwen2.5:14b  312 tokens  0.0031 USDC  2026-04-30 14:23
  job_8b2e1c3a  llama3.2:3b   89 tokens  0.0009 USDC  2026-04-30 14:19
  job_7c1d4b2f  qwen2.5:14b  891 tokens  0.0089 USDC  2026-04-30 14:11
  ...

Total today: 142 jobs, 0.9182 USDC
```

You can also see this in the dashboard under **Earnings → Ledger**.

Setting Your Payout Address

Before you can receive payments, set your payout wallet address:

```
infernet payout --set-address 0xYourWalletAddressHere
```

Or edit `~/.infernet/config.json` directly:

```
{
  "payout_address": "0xYourWalletAddressHere"
}
```

The address must be valid for the chain you want to receive payment on. Infernet supports multiple chains — specify which chain when setting the address:

```
# Ethereum mainnet
infernet payout --set-address 0x... --chain ethereum

# Base
infernet payout --set-address 0x... --chain base

# Solana
infernet payout --set-address YourSolanaPublicKey --chain solana
```

The address is sent to the control plane on the next heartbeat and appears in the dashboard under **Settings → Payout Address**.

Claiming Earnings

```
infernet payout
```

This initiates an on-chain settlement for your current unclaimed balance. The process:

1. The CLI requests a payout from the control plane
2. The control plane aggregates all pending CPRs for your node
3. An on-chain transaction is submitted to transfer the balance to your payout address
4. The CLI waits for confirmation and shows the transaction hash

```
Initiating payout...
  Unclaimed balance: 12.84 USDC
  Destination: 0x1234...abcd
  Chain: base

Submitting transaction...
  Tx hash: 0xabc123...
  Waiting for confirmation...

Confirmed! (block 14298741)
12.84 USDC transferred to 0x1234...abcd
```

Payout transactions typically confirm within 5–30 seconds on Base, a few minutes on Ethereum mainnet.

Payout Minimums

There's a minimum payout threshold to avoid spending more on gas than you earn. The minimum varies by chain:

Chain	Minimum Payout
Base	1.00 USDC
Ethereum	10.00 USDC
Polygon	1.00 USDC
Solana	1.00 USDC

If your balance is below the minimum, the payout command will tell you and exit without submitting a transaction.

Pricing Per Job

Job pricing is set by the network based on:

- **Model**: larger models cost more per token
- **Token count**: output tokens + input tokens, weighted (output costs ~4x more)
- **Tier premium**: `>=48gb` tier nodes earn a small premium for serving models others can't

Current approximate rates (subject to network demand):

Model Size	Rate (per 1K output tokens)
1B–3B	$0.001
7B	$0.002
13B–14B	$0.005
30B–33B	$0.012
70B+	$0.025

These rates are illustrative. The actual network rate is visible in the dashboard under **Earnings** → **Pricing**.

Automated Payouts

You can schedule automatic payouts via cron. The `--auto` flag runs the payout non-interactively and only if the balance meets the minimum:

```
# Claim earnings every day at midnight
0 0 * * * /usr/local/bin/infernet payout --auto >> /var/log/infernet-payout.log 2>&1
```

Or configure auto-payout in `~/.infernet/config.json`:

```
{
  "auto_payout": true,
  "auto_payout_threshold": 5.0,
  "auto_payout_schedule": "daily"
}
```

Valid schedule values: `"hourly"`, `"daily"`, `"weekly"`.

Payment Security

Payments are cryptographically tied to your node's keypair. A CPR is only valid if it's signed by the node that executed the job. The control plane verifies the signature before issuing the on-chain transaction.

This means: even if someone gains access to your payout wallet address, they can't forge CPRs or claim earnings that aren't yours. And even if someone gains access to the control plane, they can't redirect your earnings because the on-chain transaction requires a valid CPR signature.

Keep your node's private key (`~/.infernet/keys/node.key`) secure. Back it up somewhere safe. If you lose it, you can't claim any pending unclaimed earnings, and you'll need to re-register with a new keypair.

Installation

Install the CLI

```
curl -sSL https://infernetprotocol.com/install | bash
```

The installer: 1. Detects your OS and architecture 2. Downloads the appropriate pre-built Node.js binary 3. Installs it to `/usr/local/bin/infernet` (or `~/.local/bin` if not root) 4. Adds it to your PATH

After the install, either open a new shell or `source ~/.bashrc` / `source ~/.zshrc`.

Verify:

```
infernet --version
# infernet v0.1.19
#   up to date
```

Manual Install

```
INFERNET_VERSION=$(curl -s https://infernetprotocol.com/version)
curl -sSL "https://github.com/infernetprotocol/infernet-protocol/releases/download/${INFERNET_VERSION
  -o /usr/local/bin/infernet
chmod +x /usr/local/bin/infernet
```

Available platforms: `linux-x86_64`, `linux-aarch64`, `darwin-arm64`, `darwin-x86_64`.

Run Setup

```
infernet setup
```

`setup` is interactive and walks through every step. Pass `--yes` to accept defaults non-interactively:

```
infernet setup --yes
```

What Setup Does

1. Collect registration token

You'll be prompted for the one-time registration token from the dashboard. Get it at **Nodes** → **Add Node** in the Infernet Dashboard.

2. Detect hardware

The wizard runs `nvidia-smi`, `rocminfo`, and system checks to determine GPU model, VRAM, system RAM, and CPU cores. Only coarse capability data is sent to the control plane — no hostname or exact CPU model leaves the node.

3. Install inference backend

If no backend is detected, Ollama is installed automatically. If you want a different backend (vLLM, SGLang, MAX), install it first (see Chapter 3), then run setup — it will detect and configure it.

4. Pull default model

The wizard suggests a model based on your VRAM tier:

Tier	Default Model
>=48gb	qwen2.5:72b
24-48gb	qwen2.5:14b
16-24gb	qwen2.5:7b
8-16gb	qwen2.5:7b
cpu	qwen2.5:1.5b

5. Generate keypair + config

A secp256k1 keypair (Nostr-compatible) is generated. Config is written to `~/.config/infernet/config.json` with mode `0600`:

```json
{
  "controlPlane": { "url": "https://infernetprotocol.com" },
  "node": {
    "id": null,
    "nodeId": "provider-a1b2c3d4",
    "role": "provider",
    "name": "user@hostname",
    "publicKey": "abcdef...",
    "privateKey": "012345...",
    "address": "1.2.3.4",
    "port": 46337
  },
  "engine": {
    "backend": "ollama",
    "ollamaHost": "http://localhost:11434",
    "model": "qwen2.5:7b"
  }
}
```

6. Configure logrotate

On Linux, setup writes `/etc/logrotate.d/infernet` so logs in `/var/log/infernet/` rotate daily and are retained for 14 days.

7. Register with the control plane

`infernet register` is called automatically. The node's public key, role, GPU capabilities, and available models are sent to the control plane.

Firewall Configuration

The daemon makes outbound HTTPS connections only — no inbound ports are required for basic operation. If clients connect directly to your node:

```bash
# UFW
sudo ufw allow out 443/tcp
```

```
sudo ufw allow in 46337/tcp   # P2P port (default)

# iptables
iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
iptables -A INPUT  -p tcp --dport 46337 -j ACCEPT
```

Running as a System Service

```
infernet service install
```

This creates a systemd unit (Linux) or launchd plist (macOS) so the daemon starts at boot.

```
infernet service start
infernet service stop
infernet service restart
infernet service status
infernet service uninstall
```

Auto-Upgrade

The daemon checks for a new CLI version every 5 minutes against the GitHub releases API. When a newer version is available it:

1. Re-runs the installer to pull the new binary
2. Waits for in-flight jobs to finish
3. Closes all server sockets
4. Re-execs itself into the new binary

No manual intervention needed. You can also trigger an upgrade manually:

```
infernet upgrade
```

Check your current version at any time:

```
infernet --version
```

Upgrading the CLI Manually

```
infernet upgrade
```

Re-runs the curl installer. Config, keys, and registered node identity are preserved.

Uninstalling

```
infernet remove
```

Stops the daemon, removes the service unit, removes the binary, and optionally removes your config and keys (it will ask). The node's registration on the control plane is deactivated.

Model Management

Models are the inventory of your node. The more relevant models you have loaded, the more jobs you'll match. Managing this inventory well is one of the primary levers you have over your node's earnings.

How Served Models Work

Your node's config has a `served_models` array. This is the canonical list of models your node advertises to the network:

```json
{
  "served_models": ["qwen2.5:14b", "llama3.2:3b", "deepseek-coder:6.7b"]
}
```

When the daemon heartbeats, it reports these models to the control plane. The job router uses this list to determine which jobs your node can accept. A job for `qwen2.5:14b` will only be routed to nodes that list it in `served_models`.

The models must actually be loaded in the inference backend. `served_models` and what's actually pulled in your backend (e.g., `ollama list`) should stay in sync. The daemon checks this at startup and warns if there's a mismatch.

Installing Models via CLI

```
infernet model install qwen2.5:14b
```

This does three things: 1. Pulls the model into the inference backend (`ollama pull qwen2.5:14b` for Ollama backends) 2. Adds the model to `served_models` in your config 3. Sends a sync heartbeat so the control plane updates immediately (no need to wait 30 seconds)

Watch the pull progress:

```
infernet model install qwen2.5:14b --verbose
# Pulling qwen2.5:14b from Ollama registry...
# ██████████████████░░ 89% (7.2 GB / 8.1 GB)  45 MB/s
```

For vLLM or SGLang backends, specify the HuggingFace model ID:

```
infernet model install Qwen/Qwen2.5-14B-Instruct
```

Removing Models via CLI

```
infernet model remove qwen2.5:14b
```

This: 1. Removes the model from `served_models` 2. Deletes the weights from the backend 3. Syncs with the control plane

The node will stop receiving jobs for this model immediately.

Listing Installed Models

```
infernet model list
```

```
Model                    Size      Backend     Status
qwen2.5:14b              8.1 GB    ollama      loaded
llama3.2:3b              2.0 GB    ollama      loaded
deepseek-coder:6.7b      3.8 GB    ollama      loaded
```

Installing Models via Dashboard

In the Infernet Dashboard, navigate to your node's detail page and click **Manage Models**. You'll see:

- **Installed models**: currently in your served_models
- **Suggested models**: models with active job demand that your tier can run
- **Browse**: search the full model catalog

Click **Install** on any model. The command is queued in the control plane's command queue and picked up by the daemon on its next heartbeat cycle (within 30 seconds).

The daemon polls the command queue on every heartbeat tick. Commands are structured like:

```json
{
  "type": "model_install",
  "model": "qwen2.5:32b",
  "issued_at": "2026-04-30T14:00:00Z"
}
```

The daemon executes the install, updates `served_models`, and acknowledges the command back to the control plane.

Model Naming Conventions

Models are identified by the format used by the installed backend:

Ollama:

```
qwen2.5:7b
qwen2.5:14b
qwen2.5:72b
llama3.2:3b
llama3.1:8b
deepseek-coder:6.7b
nomic-embed-text:latest
```

vLLM / SGLang / MAX (HuggingFace repo IDs):

```
Qwen/Qwen2.5-7B-Instruct
Qwen/Qwen2.5-14B-Instruct
meta-llama/Llama-3.1-8B-Instruct
deepseek-ai/DeepSeek-V2-Lite
```

llama.cpp (GGUF filenames):

```
qwen2.5-7b-instruct-q4_k_m.gguf
llama-3.1-8b-instruct-q5_k_m.gguf
```

When installing via the CLI, use the naming format appropriate for your backend. The control plane normalizes model names for routing.

VRAM Planning

Before installing a model, check that it fits in your VRAM. Rule of thumb: a model needs roughly 2 bytes per parameter for FP16, 1 byte per parameter for INT8, and 0.5 bytes per parameter for Q4.

Model	FP16 VRAM	INT8 VRAM	Q4 VRAM
3B	6 GB	3 GB	1.5 GB
7B	14 GB	7 GB	3.5 GB
14B	28 GB	14 GB	7 GB
32B	64 GB	32 GB	16 GB
70B	140 GB	70 GB	35 GB

With Ollama, quantized variants are pulled by default. Specify the tag explicitly to get a particular quantization:

```
# Q4_K_M quantized (smaller, faster)
infernet model install qwen2.5:14b-instruct-q4_K_M

# FP16 (full precision, needs full VRAM)
infernet model install qwen2.5:14b-instruct-fp16
```

Hot-Swap vs Cold-Load

Ollama keeps loaded models in VRAM until memory pressure requires eviction. If you have multiple models installed, Ollama will swap them in and out as needed. This means:

- First inference on a model after a swap takes 5–30 seconds (load time)
- Subsequent inferences are fast

If you want a model to always be hot (no load latency), limit your `served_models` to the number of models that fit simultaneously in your VRAM, and tell Ollama to keep them loaded:

```
# Keep model in VRAM indefinitely (Ollama-specific)
OLLAMA_KEEP_ALIVE=-1 ollama run qwen2.5:14b
```

Or set the env var in your config and restart the daemon.

Monitoring Your Node

Status Overview

```
infernet status
```

```
Node:           provider-a1b2c3d4
Status:         online
Uptime:         3d 14h 22m
Backend:        ollama (localhost:11434)
Models loaded: qwen2.5:14b
GPU:            NVIDIA RTX 4090 (24 GB)
Last heartbeat: 18 seconds ago
```

Logs

Live Log Stream

```
infernet logs -f
```

Streams new log lines as the daemon writes them. Press Ctrl+C to stop.

Example output:

```
[2026-04-30 14:23:32] Heartbeat OK (latency: 39ms)
[2026-04-30 14:23:41] Job job_9a3f2c1d received: qwen2.5:14b, 512 ctx
[2026-04-30 14:23:42] Streaming started (job_9a3f2c1d)
[2026-04-30 14:23:48] Job job_9a3f2c1d complete: 312 tokens, 52 tok/s
[2026-04-30 14:24:02] Heartbeat OK (latency: 41ms)
```

Last N Lines

```
infernet logs --lines 500
```

Default is 200 lines. Omit `--lines` for the default.

Log File Location

Logs are written to `/var/log/infernet/daemon.log` on systems where that directory is writable (i.e., when running as root or with sudo). On non-root installs, the log falls back to `~/.config/infernet/daemon.log`.

Logrotate is configured by `infernet setup` on Linux: daily rotation, 14-day retention, `copytruncate` (daemon doesn't need a restart for rotation to take effect).

```
/var/log/infernet/daemon.log
/var/log/infernet/daemon.log.1   # yesterday
/var/log/infernet/daemon.log.2   # ...
```

Doctor

`infernet doctor` runs a diagnostic suite and reports any issues:

```
infernet doctor
```

```
Infernet Node Diagnostics
=========================
[OK] CLI version: v0.1.19 (latest)
[OK] Config file: ~/.config/infernet/config.json
[OK] Node keypair: present
[OK] Control plane: reachable (42ms)
[OK] Node registered: provider-a1b2c3d4
[OK] Node online: yes (last heartbeat 12s ago)
[OK] Backend (ollama): running at localhost:11434
[OK] Models: qwen2.5:14b (loaded)
[OK] Firewall: outbound 443 accessible
[OK] Disk space: 234 GB free
[OK] GPU drivers: NVIDIA 545.29.06
[OK] CUDA: 12.4
[OK] GPU memory: 24 GB (7.2 GB used, 16.8 GB free)

All checks passed.
```

Common issues doctor catches:

Issue	Doctor Output
Backend not running	`[FAIL] Backend (ollama): not reachable at localhost:11434`
Model mismatch	`[WARN] served_models mismatch: config has X, backend has Y`
Clock skew	`[FAIL] System time: 4m 12s drift from NTP (max 30s)`
Disk full	`[FAIL] Disk space: 2 GB free (minimum 20 GB recommended)`
Outdated CLI	`[WARN] CLI version: v0.1.10 (latest is v0.1.19)`

Run doctor first whenever your node is behaving unexpectedly.

Dashboard Monitoring

The Infernet Dashboard shows your node's status in real time:

- **Status dot**: green (online), yellow (degraded), red (offline)
- **Heartbeat**: time since last successful heartbeat
- **Version**: CLI version reported on last registration — lets you spot nodes that haven't auto-upgraded yet
- **Models**: which models are currently advertised by the node
- **Recent Jobs (Your Nodes Processed)**: last 20 jobs with timestamps, model, token count, latency, and payment

Heartbeat Intervals

The daemon heartbeats every **30 seconds**. If 3 consecutive heartbeats fail (90 seconds), the control plane marks the node offline and stops routing jobs to it. When connectivity is restored and the next heartbeat

succeeds, job routing resumes automatically.

Alerting

The control plane exposes a public status endpoint you can wire into any uptime monitoring tool:

```
GET https://infernetprotocol.com/api/v1/nodes/{node_id}/status
```

Returns `{"status": "online"}` or `{"status": "offline"}`. Works with UptimeRobot, Healthchecks.io, etc.

Dashboard webhook notifications can be configured at **Settings → Notifications** for Slack or Discord alerts when your node goes offline.

Hardware Requirements

GPU Tiers

Infernet classifies nodes into VRAM tiers. Your tier determines which models your node can serve and how you're matched to jobs. Higher tiers get more job volume and can charge more per job.

Tier: `>=48gb` (Flagship)

Example hardware: NVIDIA H100 80GB, A100 80GB, RTX 6000 Ada (48GB)

What you can run: 70B parameter models (Qwen2.5-72B, Llama-3.1-70B), large code models (DeepSeek Coder 33B), multi-modal models with large context windows.

Expected throughput: 50–150 tokens/second for 70B models. 200+ tokens/second for 7B–13B models.

Recommended backends: vLLM or SGLang for maximum throughput. Both support tensor parallelism if you have multiple GPUs.

Tier: `>=24gb` (High-End Consumer / Professional)

Example hardware: NVIDIA RTX 4090 (24GB), RTX 3090 (24GB), A5000 (24GB)

What you can run: 14B–33B models at full precision, 70B models with aggressive quantization (Q4).

Expected throughput: 30–80 tokens/second for 14B models, 15–30 tokens/second for 33B.

Recommended backends: Ollama works well. vLLM for higher throughput if you're getting steady job volume.

Tier: `>=12gb` (Mid-Range)

Example hardware: NVIDIA RTX 4080 (16GB), RTX 3080 (12GB), A2000 (12GB)

What you can run: 7B–13B models at full precision, 14B–30B with Q4 quantization.

Expected throughput: 40–90 tokens/second for 7B models.

Recommended backends: Ollama. llama.cpp if you need GGUF model support.

Tier: `>=8gb` (Entry)

Example hardware: NVIDIA RTX 4060 Ti (8GB), RTX 3070 (8GB), AMD RX 6800 (16GB)

What you can run: 7B models at full precision, larger models with heavy quantization.

Expected throughput: 20–50 tokens/second for 7B models depending on memory bandwidth.

Notes: AMD GPUs in this tier work well with Ollama via ROCm. Apple Silicon (M1 Pro, M2) fits here too, with Ollama or llama.cpp.

Tier: `cpu` (CPU-Only)

What you can run: Small models (1B–3B), heavily quantized 7B models. Not recommended for production job-taking — CPU inference is very slow for most clients.

Best use: Development, testing your setup, serving small specialized models (embedding models, classifiers).

RAM

GPU Tier	Minimum RAM	Recommended RAM
>=48gb	64 GB	128 GB
>=24gb	32 GB	64 GB
>=12gb	16 GB	32 GB
>=8gb	16 GB	32 GB
cpu	16 GB	32 GB

vLLM and SGLang manage memory aggressively and benefit from having more system RAM available for KV-cache spill. Ollama is less demanding.

Storage

Model weights take significant disk space. Budget accordingly:

Model Size	Approximate Disk (FP16)	Approximate Disk (Q4)
1B–3B	2–6 GB	0.5–2 GB
7B	14 GB	4 GB
13B–14B	28 GB	8 GB
30B–33B	65 GB	18 GB
70B–72B	140 GB	40 GB

Use fast storage (NVMe SSD) for model weights. Model loading time on cold start is much faster with NVMe vs HDD or SATA SSD.

Minimum recommendations: **- >=48gb tier**: 1 TB NVMe **- >=24gb tier**: 500 GB NVMe **- <=12gb tier**: 250 GB NVMe

---

Bandwidth

Minimum: **100 Mbps** symmetric.

The bandwidth bottleneck is model downloads (one-time) and inference results (streaming). A typical streaming inference response is a few KB/second per active job. At 10 concurrent jobs, you're looking at 50–100 KB/s of upstream.

More important than raw bandwidth is **upload latency**. High-latency connections (>100ms) degrade the streaming experience for clients. Datacenter connections are strongly preferred over residential for `>=24gb` tier and above.

Operating System

Strongly recommended: Ubuntu 22.04 LTS or Ubuntu 24.04 LTS.

The installer and daemon are tested primarily on Ubuntu. Most of the tooling in the ecosystem (NVIDIA drivers, CUDA, ROCm, Docker) has the best support on Ubuntu.

Supported: Debian 11+, other Debian-based distros, Fedora 38+, CentOS Stream 9.

macOS: Supported for development and Apple Silicon nodes. `infernet setup` works on macOS. llama.cpp and Ollama both support Metal. Production deployment on macOS is reasonable for smaller nodes.

Windows: Not currently supported. Use WSL2 if you need to test on Windows.

GPU Drivers

Install GPU drivers **before** running `infernet setup`. The setup wizard detects your GPU via `nvidia-smi` (NVIDIA) or `rocminfo` (AMD) and will warn you if drivers are missing.

NVIDIA: Install the latest stable driver from developer.nvidia.com. CUDA 12.1+ required for vLLM and SGLang. Ollama manages its own CUDA version.

AMD: ROCm 5.7+ for RX 6000/7000 series. Install via the official ROCm installer. Ollama has ROCm support built in.

Apple Silicon: No additional drivers needed. Metal is used automatically.

Quick driver check:

```
# NVIDIA
nvidia-smi

# AMD
rocm-smi

# Apple Silicon
system_profiler SPDisplaysDataType | grep "Metal"
```

Chapter 3: Inference Backends

Infernet Protocol supports five inference backends. The daemon auto-detects which one to use, but understanding your options lets you pick the right tool for your hardware and workload.

In This Chapter

- Ollama — The default. Easy setup, broad hardware support (NVIDIA, AMD, Apple Silicon, CPU).
- vLLM — High throughput on NVIDIA hardware. Best for nodes with heavy job volume.
- SGLang — KV-cache reuse and structured output. Best when clients use similar prompt prefixes.
- Modular MAX — High throughput with a different memory management approach. Competitive with vLLM.
- llama.cpp — No Python dependency. CPU and Apple Silicon native. GGUF model format.
- Choosing a Backend — Decision guide with a comparison table.

How Auto-Selection Works

At startup, the daemon probes each backend in priority order by sending a health check request. The first backend that responds is used.

Default probe order: 1. Ollama (`localhost:11434/api/tags`) 2. vLLM (`localhost:8000/health`) 3. SGLang (`localhost:30000/health`) 4. Modular MAX (`localhost:8080/health`) 5. llama.cpp / llama-swap (`localhost:8080/health`)

Override the selection with an env var:

```
export INFERNET_BACKEND=vllm
infernet start
```

Or in your config:

```
{
  "backend": "vllm"
}
```

Backend Adapter Interface

All backends speak to the daemon through a common adapter interface:

- `POST /generate` — single inference request
- `POST /generate/stream` — streaming inference request
- `GET /models` — list loaded models
- `GET /health` — health check

The daemon translates between this interface and each backend's native API. You don't need to know the backend's native API unless you're doing advanced configuration.

Choosing a Backend

Decision Guide

Start with these questions:

Do you have an NVIDIA GPU and want maximum throughput? → Use **vLLM** or **SGLang**. They're neck-and-neck on throughput. SGLang wins when requests share a common system prompt or context. vLLM wins when requests are highly varied.

Are you on Apple Silicon or need CPU inference? → Use **llama.cpp** (via llama-swap). It has the best Metal support and GGUF quantization makes large models practical.

Do you have an AMD GPU? → Use **Ollama**. It has the most mature ROCm support.

Do you want the easiest setup with broad hardware support? → Use **Ollama**. It works everywhere, installs in one command, and handles model management well.

Do you want to benchmark before committing? → Use **Ollama** to get started, then swap to **vLLM** or **MAX** and measure. The swap is a single config line change.

Comparison Table

Feature	Ollama	vLLM	SGLang	MAX	llama.cpp
NVIDIA CUDA	Yes	Yes	Yes	Yes	Yes
AMD ROCm	Yes	Partial	No	Preview	Yes
Apple Silicon	Yes	No	No	No	Yes
CPU	Yes	No	No	No	Yes
Install complexity	Low	Medium	Medium	Medium	Medium
Python required	No	Yes	Yes	Yes	No
Model format	GGUF/GGML	Hugging-Face	HuggingFace	Hugging-Face	GGUF
Multi-GPU	Limited	Yes (Ray)	Yes	Planned	No
KV-cache reuse	No	No	Yes (Radix)	Partial	No
Continuous batching	Yes	Yes	Yes	Yes	Limited
Structured output	Limited	Yes	Yes (fast)	Partial	Yes
Speculative decoding	No	Yes	Yes	Planned	Yes
Memory efficiency	Good	Excellent	Excellent	Excellent	Good
Relative throughput (7B)	1.0x	1.4x	1.5x	1.5x	0.8x

Throughput numbers are normalized to Ollama = 1.0x for a single NVIDIA GPU. Actual results vary by model and workload.

Hardware-Specific Recommendations

NVIDIA RTX 4090 (24GB)

Best: **vLLM** or **SGLang** for production throughput. **Ollama** for easy setup or mixed workloads.

```
# Primary: vLLM
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --port 8000

# Config
export INFERNET_BACKEND=vllm
```

NVIDIA H100 (80GB)

Best: **vLLM** with tensor parallelism, or **MAX** for newer architectures.

```
# vLLM single-GPU for 70B models
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --port 8000
```

AMD RX 7900 XTX (24GB)

Best: **Ollama** (most stable ROCm support).

```
infernet setup  # Ollama will be auto-selected
```

Apple Silicon M2/M3

Best: **Ollama** for convenience, **llama.cpp** for maximum performance.

```
# Ollama (easiest)
ollama pull qwen2.5:14b

# llama.cpp for performance-critical nodes
llama-server \
  --model qwen2.5-14b-q4km.gguf \
  --n-gpu-layers 99 \
  --port 8080
```

CPU Only

Best: **llama.cpp** with Q4_K_M quantization.

```
llama-server \
  --model qwen2.5-7b-q4km.gguf \
  --n-gpu-layers 0 \
  --threads $(nproc) \
  --port 8080
```

Switching Backends

Switching is a one-line config change:

```
{
  "backend": "vllm",
  "vllm_host": "http://localhost:8000",
  "vllm_model": "Qwen/Qwen2.5-14B-Instruct"
}
```

Then restart the daemon:

```
infernet service restart
# or
infernet stop && infernet start
```

The control plane is updated on the next heartbeat. No re-registration required.

Benchmarking Your Setup

Before committing to a backend, benchmark it with realistic traffic:

```
# Install benchmark tool
pip install "sglang[benchmark]"

# Benchmark with 16 concurrent users, 512 output tokens each
python -m sglang.bench_serving \
  --backend openai \
  --base-url http://localhost:8000 \
  --model Qwen/Qwen2.5-14B-Instruct \
  --num-prompts 100 \
  --request-rate 16 \
  --output-len 512
```

Key metrics to compare: **- Throughput (tokens/second)**: total output tokens divided by total time **- First token latency (TTFT)**: time from request to first token — affects perceived responsiveness **- Inter-token latency**: time between tokens in a stream — should be consistent **- P99 latency**: worst-case latency under load

---

llama.cpp

llama.cpp is a C++ implementation of LLM inference with no Python dependency. It runs on CPU, NVIDIA (CUDA), AMD (ROCm), Apple Silicon (Metal), and even mobile hardware. It uses GGUF, a quantized model format that makes large models accessible on consumer hardware.

Use llama.cpp when you want CPU or Apple Silicon inference, need GGUF models, or want to avoid a Python installation entirely.

llama-swap

For production use with Infernet, we recommend **llama-swap** on top of llama.cpp. llama-swap is a Go proxy that manages multiple llama.cpp instances, handles model routing by name, and hot-swaps models on demand. It exposes an OpenAI-compatible API that Infernet can talk to directly.

```
Client → llama-swap (port 8080) → llama.cpp instance (per model)
```

Installing llama.cpp

Pre-built Binaries

```
# Download latest release
LLAMA_VERSION=$(curl -s https://api.github.com/repos/ggerganov/llama.cpp/releases/latest | jq -r .tag
curl -sSL "https://github.com/ggerganov/llama.cpp/releases/download/${LLAMA_VERSION}/llama-${LLAMA_VE
unzip llama.zip -d llama-cpp
sudo cp llama-cpp/llama-server /usr/local/bin/
```

Building from Source (with CUDA)

```
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release -j$(nproc)
sudo cp build/bin/llama-server /usr/local/bin/
```

Building for Apple Silicon (Metal)

```
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp
cmake -B build -DGGML_METAL=ON
cmake --build build --config Release -j$(sysctl -n hw.logicalcpu)
```

Installing llama-swap

```
# Download pre-built binary
curl -sSL https://github.com/mostlygeek/llama-swap/releases/latest/download/llama-swap-linux-amd64 \
  -o /usr/local/bin/llama-swap
chmod +x /usr/local/bin/llama-swap
```

Getting GGUF Models

GGUF models are hosted on HuggingFace under repos maintained by Bartowski, TheBloke, and others:

```
# Using HuggingFace CLI
pip install huggingface_hub
huggingface-cli download bartowski/Qwen2.5-14B-Instruct-GGUF \
  Qwen2.5-14B-Instruct-Q4_K_M.gguf \
  --local-dir ~/.cache/llama-cpp/models/

# Or direct download
wget https://huggingface.co/bartowski/Qwen2.5-14B-Instruct-GGUF/resolve/main/Qwen2.5-14B-Instruct-Q4_
  -O ~/.cache/llama-cpp/models/qwen2.5-14b-q4km.gguf
```

GGUF quantization levels:

Quantization	Size (7B)	Quality	Use case
Q2_K	2.7 GB	Low	Minimum VRAM/RAM
Q4_K_M	4.1 GB	Good	Default choice
Q5_K_M	4.8 GB	Very good	Quality-focused
Q6_K	5.5 GB	Near-lossless	Max quality / CPU
Q8_0	7.2 GB	Near-lossless	VRAM to spare
F16	14 GB	Lossless	Reference

Starting llama.cpp Directly

```
llama-server \
  --model ~/.cache/llama-cpp/models/qwen2.5-14b-q4km.gguf \
  --host 0.0.0.0 \
  --port 8080 \
  --n-gpu-layers 99 \
  --ctx-size 8192 \
  --n-parallel 4
```

`--n-gpu-layers 99` offloads all layers to GPU. Set to 0 for CPU-only. Set to a specific number to split between CPU and GPU (useful when the model doesn't fully fit in VRAM).

Configuring llama-swap

llama-swap uses a YAML config to define which models to serve:

```
# ~/.config/llama-swap/config.yaml
models:
  qwen2.5:14b:
    cmd: llama-server
    args:
      - --model
      - /root/.cache/llama-cpp/models/qwen2.5-14b-q4km.gguf
      - --n-gpu-layers
```

```
      - "99"
      - --ctx-size
      - "8192"
      - --n-parallel
      - "4"
    port: 8081

  qwen2.5:7b:
    cmd: llama-server
    args:
      - --model
      - /root/.cache/llama-cpp/models/qwen2.5-7b-q4km.gguf
      - --n-gpu-layers
      - "99"
      - --ctx-size
      - "8192"
      - --n-parallel
      - "8"
    port: 8082

proxy:
  port: 8080
  healthcheck_timeout: 30s
  swap_timeout: 60s
```

Start llama-swap:

```
llama-swap --config ~/.config/llama-swap/config.yaml
```

llama-swap now handles routing requests for `qwen2.5:14b` and `qwen2.5:7b` to the appropriate llama-server instance, loading/unloading models as needed.

Apple Silicon Performance

On Apple Silicon, llama.cpp with Metal is often the best choice — even faster than many NVIDIA GPU setups for smaller models, thanks to the unified memory architecture:

```
# macOS build with Metal automatically uses GPU
llama-server \
  --model qwen2.5-14b-q4km.gguf \
  --n-gpu-layers 99 \
  --ctx-size 16384 \
  --port 8080
```

M2 Pro (16GB) can comfortably run Q4 14B models at 30–50 tokens/second. M3 Max (128GB) can run Q4 70B models at 20+ tokens/second.

CPU-Only Mode

For CPU inference, use Q4_K_M or smaller:

```
llama-server \
  --model qwen2.5-7b-q4km.gguf \
  --n-gpu-layers 0 \
  --threads $(nproc) \
  --ctx-size 4096 \
  --port 8080
```

CPU inference is slow (5–15 tokens/second for a 7B model on a modern CPU), but it works and requires no GPU.

Key Environment Variables

```
# llama.cpp server address
LLAMACPP_HOST=http://localhost:8080

# For llama-swap, same variable
LLAMACPP_HOST=http://localhost:8080

# CUDA device selection
CUDA_VISIBLE_DEVICES=0
```

Infernet Config

```
{
  "backend": "llamacpp",
  "llamacpp_host": "http://localhost:8080"
}
```

Systemd Service (llama-swap)

```
[Unit]
Description=llama-swap Model Server
After=network.target

[Service]
Type=simple
User=infernet
ExecStart=/usr/local/bin/llama-swap --config /home/infernet/.config/llama-swap/config.yaml
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Modular MAX

Modular MAX (formerly Modular Engine) is a high-performance inference runtime from Modular, the company behind the Mojo programming language. MAX uses a compiler-based approach to optimization rather than hand-tuned CUDA kernels, which means it can apply aggressive optimizations across the full compute graph.

Requirements

- NVIDIA GPU with CUDA 12+ (CPU inference also supported)
- Ubuntu 22.04+ or macOS
- Python 3.9+

AMD GPU support is in preview as of early 2026.

Installing MAX

```
# Install Modular CLI
curl -ssL https://magic.modular.com | bash
source ~/.bashrc

# Install MAX
magic add max
```

Or via pip:

```
pip install max
```

Verify:

```
max --version
```

Starting MAX Serve

MAX provides an OpenAI-compatible serving endpoint:

```
max serve \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 8080
```

With a HuggingFace token (for gated models):

```
HF_TOKEN=hf_... max serve \
  --model-path meta-llama/Llama-3.1-8B-Instruct \
  --host 0.0.0.0 \
  --port 8080
```

Test:

```
curl http://localhost:8080/health
# {"status":"ok"}
```

```
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Qwen/Qwen2.5-14B-Instruct",
    "messages": [{"role": "user", "content": "Hello"}],
    "max_tokens": 50
  }'
```

How MAX Differs

Compiler-based optimization: vLLM and SGLang are written primarily in Python with custom CUDA kernels for hot paths. MAX compiles the model graph end-to-end using MLIR and applies optimizations that span operator boundaries — fusing attention + MLP, eliminating redundant data movement, etc. For well-supported architectures, this gives competitive or superior performance without requiring architecture-specific hand tuning.

Mojo runtime: The inference kernels are written in Mojo, which compiles to native code. This avoids Python GIL overhead and enables aggressive inlining that Python-based frameworks can't do.

Continuous batching: Like vLLM, MAX uses continuous batching to maximize GPU utilization across concurrent requests. New requests join an in-flight batch as soon as a slot is available rather than waiting for the current batch to complete.

Throughput Benchmarks

On an NVIDIA H100 80GB serving Qwen2.5-14B-Instruct at 512 output tokens, approximate throughput:

Backend	Tokens/second (single stream)	Tokens/second (16 concurrent)
Ollama	85	220
vLLM	120	580
SGLang	125	610
MAX	130	640

These numbers are illustrative. Actual performance depends heavily on model architecture, quantization, batch size, and hardware. Benchmark against your own workload.

Quantization

MAX supports several quantization formats:

```
# AWQ quantization (fast, good quality)
max serve \
  --model-path Qwen/Qwen2.5-14B-Instruct-AWQ \
  --port 8080

# GPTQ
max serve \
```

```
  --model-path TheBloke/Qwen2.5-14B-GPTQ \
  --port 8080

# bitsandbytes (4-bit NF4)
max serve \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --quantization bnb-4bit \
  --port 8080
```

Context Length

```
# Extend context beyond model default
max serve \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --max-length 32768 \
  --port 8080
```

Key Environment Variables

```
# MAX server address
MODULAR_MAX_HOST=http://localhost:8080

# Model path
MAX_MODEL=Qwen/Qwen2.5-14B-Instruct

# HuggingFace token
HF_TOKEN=hf_your_token_here

# GPU device selection
CUDA_VISIBLE_DEVICES=0

# Number of GPU workers
MAX_NUM_GPUS=1
```

Infernet Config

```
{
  "backend": "max",
  "max_host": "http://localhost:8080",
  "max_model": "Qwen/Qwen2.5-14B-Instruct"
}
```

Systemd Service

```
[Unit]
Description=Modular MAX Inference Server
```

```
After=network.target

[Service]
Type=simple
User=infernet
ExecStart=/home/infernet/.modular/bin/max serve \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 8080
Restart=always
RestartSec=10
Environment=CUDA_VISIBLE_DEVICES=0

[Install]
WantedBy=multi-user.target
```

Model Support

MAX has strong support for: - Qwen2, Qwen2.5 family - Llama 3, Llama 3.1, Llama 3.2 - Mistral, Mixtral - Phi-3, Phi-3.5 - Gemma 2

Check the Modular docs for the current supported model list. For models not yet natively supported, MAX can often serve them via HuggingFace Transformers compatibility mode (slower but still works).

Ollama

Ollama is the default backend. It handles NVIDIA, AMD (ROCm), Apple Silicon (Metal), and CPU inference with no additional configuration. If you're not sure which backend to use, start with Ollama.

Installing Ollama

`infernet setup` installs Ollama automatically if no backend is detected. To install it manually:

```
curl -fsSL https://ollama.com/install.sh | sh
```

This installs the `ollama` binary and configures it as a systemd service.

Verify the install:

```
ollama --version
# ollama version 0.5.4

ollama list
# NAME    ID    SIZE  MODIFIED
```

Pulling Models

Ollama manages its own model registry. Pull models before starting the Infernet daemon:

```
ollama pull qwen2.5:14b
ollama pull qwen2.5:7b
ollama pull llama3.2:3b
ollama pull nomic-embed-text
```

Ollama auto-selects quantization based on your available VRAM. To pull a specific quantization:

```
# Q4_K_M — good balance of quality and size
ollama pull qwen2.5:14b-instruct-q4_K_M

# Q8 — higher quality, needs more VRAM
ollama pull qwen2.5:14b-instruct-q8_0

# Full precision FP16
ollama pull qwen2.5:14b-instruct-fp16
```

List your pulled models:

```
ollama list
```

```
NAME                              ID              SIZE      MODIFIED
qwen2.5:14b                       d8c5e0b67b1e    8.1 GB   2 hours ago
llama3.2:3b                       a80c4f17acd5    2.0 GB   1 day ago
nomic-embed-text:latest           0a109f422b47    274 MB   3 days ago
```

Hardware Support

NVIDIA (CUDA)

Ollama uses CUDA for NVIDIA GPUs. CUDA is bundled with Ollama — you don't need to install it separately (though having it installed doesn't hurt). You do need the NVIDIA driver.

```
# Verify NVIDIA GPU is detected
nvidia-smi
ollama run qwen2.5:7b "hello" 2>&1 | head -5
```

Ollama will use all available NVIDIA GPUs. To restrict to specific GPUs:

```
CUDA_VISIBLE_DEVICES=0 ollama serve
```

AMD (ROCm)

ROCm support is included in the official Ollama Linux build. Works with RX 6000 series and newer:

```
# Verify ROCm is detected
rocm-smi
ollama run qwen2.5:7b "hello"
```

If Ollama doesn't detect your AMD GPU, check:

```
# Should list your GPU
/opt/rocm/bin/rocminfo | grep "Agent 2" -A 10
```

Older GCN architecture cards may need `HSA_OVERRIDE_GFX_VERSION`:

```
HSA_OVERRIDE_GFX_VERSION=10.3.0 ollama serve
```

Apple Silicon (Metal)

Ollama uses Metal on Apple Silicon. No configuration needed:

```
ollama run qwen2.5:7b "hello"
# Uses GPU automatically
```

Ollama treats the unified memory as both system RAM and VRAM, so a MacBook Pro with 36GB unified memory can run larger models than an NVIDIA GPU with 24GB VRAM.

CPU Fallback

If no GPU is detected, Ollama falls back to CPU inference using highly optimized GGML kernels. It's slow but works:

```
OLLAMA_NUM_GPU=0 ollama serve  # Force CPU mode
```

Key Environment Variables

```
# Ollama server address (default: http://127.0.0.1:11434)
OLLAMA_HOST=0.0.0.0:11434

# Keep models in VRAM between requests (0 = unload after use, -1 = never unload)
OLLAMA_KEEP_ALIVE=5m

# Number of parallel requests per model (default: auto based on VRAM)
OLLAMA_NUM_PARALLEL=4

# Max number of models loaded simultaneously
OLLAMA_MAX_LOADED_MODELS=3

# Context window size override
OLLAMA_NUM_CTX=8192

# Force CPU-only mode
OLLAMA_NUM_GPU=0

# Custom models directory
OLLAMA_MODELS=/mnt/fast-nvme/ollama/models
```

Set these in `/etc/systemd/system/ollama.service.d/override.conf` for persistent configuration:

```
[Service]
Environment="OLLAMA_KEEP_ALIVE=-1"
Environment="OLLAMA_NUM_PARALLEL=4"
Environment="OLLAMA_MODELS=/mnt/fast-nvme/ollama/models"
```

```
sudo systemctl daemon-reload
sudo systemctl restart ollama
```

Infernet-Specific Config

In your Infernet config (`~/.infernet/config.json`):

```
{
  "backend": "ollama",
  "ollama_host": "http://localhost:11434"
}
```

If Ollama is running on a different machine or port:

```
{
  "backend": "ollama",
  "ollama_host": "http://192.168.1.50:11434"
}
```

Troubleshooting

Ollama not starting:

```
sudo systemctl status ollama
journalctl -u ollama -n 50
```

GPU not detected:

```
# NVIDIA
nvidia-smi  # Should show GPU
ls /dev/nvidia*  # Should show devices

# AMD
rocm-smi --showid
ls /dev/kfd  # Should exist for ROCm
```

Slow inference (possible CPU fallback):

```
# Run with verbose to see what device is used
OLLAMA_DEBUG=1 ollama run qwen2.5:7b "hello" 2>&1 | grep -i gpu
```

Out of memory:

```
# Check current VRAM usage
nvidia-smi --query-gpu=memory.used,memory.free --format=csv

# Unload all models
ollama stop --all
```

SGLang

SGLang (Structured Generation Language) is an inference engine that focuses on KV-cache reuse and structured output generation. It's particularly efficient when many requests share common prompt prefixes — a common pattern in chatbots with a shared system prompt.

Requirements

- NVIDIA GPU with CUDA 12.1+
- Python 3.9+
- 16 GB+ VRAM recommended

Installing SGLang

```
pip install "sglang[all]"
```

Or with a specific CUDA version:

```
pip install "sglang[all]" --find-links https://flashinfer.ai/whl/cu121/torch2.4/
```

Using Docker:

```
docker pull lmsys/sglang:latest
```

Starting SGLang

SGLang also exposes an OpenAI-compatible API:

```
python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 30000 \
  --served-model-name qwen2.5:14b

# With Docker
docker run --gpus all --rm \
  -v ~/.cache/huggingface:/root/.cache/huggingface \
  -p 30000:30000 \
  lmsys/sglang:latest \
  python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --port 30000
```

Test:

```
curl http://localhost:30000/health
# {"status":"ok"}
```

RadixAttention: KV-Cache Reuse

SGLang's standout feature is RadixAttention. It organizes the KV-cache as a radix tree, where the tree edges represent token sequences. When two requests share a common prefix (e.g., the same system prompt), SGLang reuses the computed KV-cache for that prefix rather than recomputing it.

In practice, this means:

- System prompts are computed once and cached, not once per request
- Multi-turn conversations benefit because earlier turns are already cached
- RAG applications that prepend the same documents to many queries see dramatic speedups

For a workload where 80% of requests share the same 500-token system prompt, RadixAttention can reduce total computation by 40–60%.

You don't need to enable it — RadixAttention is on by default. The cache size is configurable:

```
python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --max-prefill-tokens 16384 \
  --mem-fraction-static 0.85
```

`--mem-fraction-static` controls what fraction of GPU memory is reserved for the KV-cache (vs model weights). Higher values give more cache capacity.

Speculative Decoding

SGLang supports speculative decoding, which uses a small draft model to generate candidate tokens and a larger verifier model to accept/reject them. This can increase token throughput by 2–3x with minimal quality loss.

```
python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --speculative-draft-model-path Qwen/Qwen2.5-1.5B-Instruct \
  --speculative-num-draft-tokens 4 \
  --port 30000
```

The draft model must be from the same model family and small enough that draft generation is cheap. A 7B verifier with a 1.5B draft is a common pairing.

Structured Output

SGLang's native structured output is more efficient than the schema-enforcement approaches used by other backends, because it constrains the sampling process rather than post-filtering:

```
curl http://localhost:30000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5:14b",
    "messages": [{"role": "user", "content": "Extract the name and age from: John Smith is 32 years o
    "response_format": {
      "type": "json_schema",
      "json_schema": {
```

```
        "name": "person",
        "schema": {
          "type": "object",
          "properties": {
            "name": {"type": "string"},
            "age": {"type": "integer"}
          },
          "required": ["name", "age"]
        }
      }
    }
  }'
```

Multi-GPU

```
python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-72B-Instruct \
  --tp 4 \
  --port 30000
```

`--tp` is the tensor parallelism degree. Use `--dp` for data parallelism (multiple model replicas):

```
# 2 replicas, each on 2 GPUs (4 GPUs total)
python -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --tp 2 \
  --dp 2 \
  --port 30000
```

Key Environment Variables

```
# SGLang server address
SGLANG_HOST=http://localhost:30000

# Default model
SGLANG_MODEL=Qwen/Qwen2.5-14B-Instruct

# HuggingFace token
HF_TOKEN=hf_your_token_here

# CUDA device selection
CUDA_VISIBLE_DEVICES=0
```

Infernet Config

```
{
  "backend": "sglang",
```

```
  "sglang_host": "http://localhost:30000",
  "sglang_model": "Qwen/Qwen2.5-14B-Instruct"
}
```

When SGLang Wins vs vLLM

SGLang tends to outperform vLLM when: - Requests share long common prefixes (system prompts, RAG context) - You're serving structured output with JSON schemas - You're doing multi-step reasoning chains with shared context

vLLM tends to win when: - Requests are highly varied with no shared prefix - You need the broadest model support (vLLM supports more exotic architectures) - You're already invested in vLLM tooling

For most Infernet workloads (chat with a consistent system prompt), SGLang and vLLM are competitive. Run a benchmark against your actual workload to decide.

Systemd Service

```
[Unit]
Description=SGLang Inference Server
After=network.target

[Service]
Type=simple
User=infernet
ExecStart=/usr/bin/python3 -m sglang.launch_server \
  --model-path Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 30000 \
  --served-model-name qwen2.5:14b
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

---

vLLM

vLLM is a high-throughput inference engine for NVIDIA GPUs. It uses PagedAttention to manage KV-cache memory more efficiently than naive implementations, enabling higher concurrency and better GPU utilization under load.

Use vLLM when you're getting steady job volume and want to maximize tokens/second per dollar of GPU cost.

Requirements

- NVIDIA GPU with CUDA 11.8+
- Python 3.9+
- 8 GB+ VRAM (16 GB+ recommended)

Installing vLLM

```
pip install vllm
```

For a specific CUDA version:

```
# CUDA 12.1
pip install vllm --extra-index-url https://download.pytorch.org/whl/cu121

# CUDA 11.8
pip install vllm --extra-index-url https://download.pytorch.org/whl/cu118
```

Using Docker (recommended for production):

```
docker pull vllm/vllm-openai:latest
```

Verify the install:

```
python -c "import vllm; print(vllm.__version__)"
```

Starting vLLM

vLLM exposes an OpenAI-compatible REST API. Start it pointing at your model:

```
# HuggingFace model
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 8000 \
  --served-model-name qwen2.5:14b

# With Docker
docker run --gpus all --rm \
  -v ~/.cache/huggingface:/root/.cache/huggingface \
  -p 8000:8000 \
  vllm/vllm-openai:latest \
  --model Qwen/Qwen2.5-14B-Instruct \
  --served-model-name qwen2.5:14b
```

Test it's working:

```
curl http://localhost:8000/health
# {"status":"ok"}

curl http://localhost:8000/v1/completions \
```

```
  -H "Content-Type: application/json" \
  -d '{"model": "qwen2.5:14b", "prompt": "Hello", "max_tokens": 10}'
```

PagedAttention

PagedAttention is vLLM's key performance feature. Normal KV-cache implementations allocate a fixed block of memory per sequence upfront (to handle the maximum context length). Most of this allocation is wasted for short sequences.

PagedAttention uses virtual memory pages instead: it allocates KV-cache in small pages and only pages in what's actually needed. This enables:

- Higher concurrency: more sequences fit in VRAM simultaneously
- Better GPU utilization: less idle VRAM
- Longer effective context: cache can span physical memory blocks

The result is 2–4x higher throughput than naive implementations at the same VRAM budget.

You don't need to configure PagedAttention — it's on by default. The only relevant setting is the block size:

```
# Default block size is 16 tokens per page
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --block-size 16
```

Larger block sizes (32) improve throughput for long sequences. Smaller block sizes (8) improve memory efficiency for short sequences.

Multi-GPU with Ray (Tensor Parallelism)

For models larger than a single GPU's VRAM, vLLM uses Ray for tensor parallelism:

```
# Install Ray
pip install ray

# Start with 2-GPU tensor parallelism
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 2 \
  --host 0.0.0.0 \
  --port 8000
```

For 4 GPUs:

```
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 4 \
  --host 0.0.0.0 \
  --port 8000
```

See Chapter 6: Multi-GPU for more on multi-GPU configurations.

Key Environment Variables

```
# Model server address
VLLM_HOST=http://localhost:8000

# Default model name (used if serving a single model)
VLLM_MODEL=Qwen/Qwen2.5-14B-Instruct

# HuggingFace token (for gated models like Llama)
HF_TOKEN=hf_your_token_here

# CUDA device selection
CUDA_VISIBLE_DEVICES=0,1

# Disable usage stats reporting
VLLM_NO_USAGE_STATS=1
```

Infernet Config

```
{
  "backend": "vllm",
  "vllm_host": "http://localhost:8000",
  "vllm_model": "Qwen/Qwen2.5-14B-Instruct"
}
```

Performance Tuning

```
# Increase max concurrent requests (default: 256)
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --max-num-seqs 512

# Limit GPU memory fraction (default: 0.90)
# Useful if you want headroom for other processes
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --gpu-memory-utilization 0.85

# Quantization (reduces VRAM at some quality cost)
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --quantization awq

# Use FP8 KV cache (reduces VRAM for KV cache)
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --kv-cache-dtype fp8
```

Switching Between Models

Unlike Ollama, a single vLLM server instance typically serves one model. To serve multiple models, run multiple instances on different ports:

```
# Model 1 on port 8000
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --port 8000 &

# Model 2 on port 8001
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-7B-Instruct \
  --port 8001 &
```

The Infernet daemon handles routing to the correct port based on the requested model.

Systemd Service

```
[Unit]
Description=vLLM Inference Server
After=network.target

[Service]
Type=simple
User=infernet
ExecStart=/usr/bin/python3 -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --host 0.0.0.0 \
  --port 8000 \
  --served-model-name qwen2.5:14b
Restart=always
RestartSec=10
Environment=CUDA_VISIBLE_DEVICES=0

[Install]
WantedBy=multi-user.target
```

Chapter 4: Building Apps

This chapter is for developers building applications on top of Infernet Protocol. You don't need to run a node — you submit jobs to the network via a REST API and get inference results back.

In This Chapter

- API Overview — Base URL, authentication, and the core endpoints. Everything you need to make your first API call.
- Streaming Chat — How to consume SSE token streams in JavaScript and Python. Handling done events and errors.
- Job Lifecycle — Job states, polling vs streaming, retry logic, and what to do when jobs fail.

The Developer's View

From a developer's perspective, Infernet Protocol looks like a standard LLM API — similar to OpenAI's API — but backed by a decentralized network of GPU nodes. You submit a prompt, specify a model, and get tokens back. The routing, node selection, and payment happen under the hood.

The key differences from a centralized provider:

1. **Model availability varies**: nodes on the network serve different models. If you request a model that no node currently has loaded, the job will queue until a node with that model is available — or fail with `no_capacity` if none are registered.

2. **Streaming is the default**: because responses come from distributed nodes, streaming is the most reliable way to get results. Polling a job ID also works but adds latency.

3. **Auth uses bearer tokens from the dashboard**: you get a bearer token by creating an API key in the Infernet Dashboard. This token identifies your account for billing and rate limiting.

Quick API Test

```
# Set your token
export INFERNET_BEARER_TOKEN="your_token_here"

# Submit a job
curl https://infernetprotocol.com/api/v1/jobs \
  -H "Authorization: Bearer $INFERNET_BEARER_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5:7b",
    "messages": [{"role": "user", "content": "What is 2 + 2?"}],
    "stream": false
  }'
```

You should get back a JSON response with the job result in under a few seconds.

API Overview

Base URL

All API requests go to the control plane:

`https://infernetprotocol.com/api/v1`

If you're self-hosting the control plane, replace this with your own URL.

Authentication

All requests require a bearer token:

`Authorization: Bearer your_token_here`

Get a token by creating an API key in the Infernet Dashboard under **Settings** → **API Keys**.

Tokens are scoped to your account. Rate limits and billing are tracked per token.

Core Endpoints

POST /api/v1/jobs

Submit an inference job.

Request:

```json
{
  "model": "qwen2.5:14b",
  "messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "What is the capital of France?"}
  ],
  "max_tokens": 256,
  "temperature": 0.7,
  "stream": false
}
```

Parameters:

Field	Type	Required	Description
`model`	string	Yes	Model identifier (e.g., `qwen2.5:14b`)
`messages`	array	Yes	Chat messages in OpenAI format
`max_tokens`	integer	No	Maximum output tokens (default: 512)
`temperature`	float	No	Sampling temperature 0.0–2.0 (default: 0.7)
`top_p`	float	No	Nucleus sampling (default: 1.0)
`stream`	boolean	No	Return SSE stream immediately (default: false)
`node_id`	string	No	Pin to a specific node (advanced)

Field	Type	Required	Description

Response (stream: false):

```json
{
  "id": "job_9a3f2c1d",
  "status": "pending",
  "model": "qwen2.5:14b",
  "created_at": "2026-04-30T14:23:41Z"
}
```

When `stream: true`, the response is an SSE stream directly. See Streaming Chat.

Example:

```bash
curl https://infernetprotocol.com/api/v1/jobs \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5:14b",
    "messages": [{"role": "user", "content": "Hello!"}],
    "stream": false
  }'
```

GET /api/v1/jobs/:id

Get job status and result.

Response (pending):

```json
{
  "id": "job_9a3f2c1d",
  "status": "pending",
  "model": "qwen2.5:14b",
  "created_at": "2026-04-30T14:23:41Z"
}
```

Response (completed):

```json
{
  "id": "job_9a3f2c1d",
  "status": "completed",
  "model": "qwen2.5:14b",
  "node_id": "node_8f3a2c1d",
  "result": {
    "content": "Paris is the capital of France.",
    "usage": {
      "prompt_tokens": 24,
      "completion_tokens": 8,
```

```json
      "total_tokens": 32
    }
  },
  "created_at": "2026-04-30T14:23:41Z",
  "completed_at": "2026-04-30T14:23:43Z",
  "latency_ms": 2041
}
```

Response (failed):

```json
{
  "id": "job_9a3f2c1d",
  "status": "failed",
  "error": "no_capacity",
  "error_message": "No nodes available with model qwen2.5:14b",
  "created_at": "2026-04-30T14:23:41Z"
}
```

GET /api/v1/jobs/:id/stream

Open an SSE stream to receive tokens as they're generated. This is the recommended way to deliver results in real-time applications.

See Streaming Chat for full documentation.

GET /api/v1/models

List models currently available on the network (at least one online node has the model loaded).

Response:

```json
{
  "models": [
    {
      "id": "qwen2.5:72b",
      "nodes": 3,
      "avg_tokens_per_second": 48
    },
    {
      "id": "qwen2.5:14b",
      "nodes": 12,
      "avg_tokens_per_second": 78
    },
    {
      "id": "qwen2.5:7b",
      "nodes": 28,
      "avg_tokens_per_second": 95
```

```json
    },
    {
      "id": "llama3.2:3b",
      "nodes": 15,
      "avg_tokens_per_second": 130
    }
  ]
}
```

Use this to check availability before submitting a job, especially for large models.

GET /api/v1/jobs

List your recent jobs.

Query params:

Param	Default	Description
`limit`	20	Number of jobs to return (max 100)
`status`	all	Filter by status: `pending`, `processing`, `completed`, `failed`
`since`	—	ISO8601 timestamp to filter from

Example:

```bash
curl "https://infernetprotocol.com/api/v1/jobs?limit=10&status=completed" \
  -H "Authorization: Bearer $TOKEN"
```

Error Responses

All errors return a JSON body with `error` and `error_message`:

```json
{
  "error": "unauthorized",
  "error_message": "Invalid or expired bearer token"
}
```

Common error codes:

Code	HTTP Status	Meaning
`unauthorized`	401	Missing or invalid token
`bad_request`	400	Missing required field or invalid parameter
`model_not_found`	404	Model is not registered on the network
`no_capacity`	503	Model exists but no nodes have capacity
`rate_limited`	429	Too many requests; back off and retry

Code	HTTP Status	Meaning
`internal_error`	500	Something went wrong on our end

Rate Limits

Default rate limits for API tokens:

- 60 requests per minute
- 1000 requests per hour
- Limits are per token, not per IP

Rate limit headers are included in responses:

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 47
X-RateLimit-Reset: 1746020400
```

When rate limited, you'll receive HTTP 429. The `Retry-After` header tells you when to retry:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 13
```

Job Lifecycle

States

A job moves through these states:

```
pending → processing → completed
                    ▯ failed
```

State	Description
`pending`	Job submitted, waiting for a node to accept it
`processing`	A node has accepted the job and is running inference
`completed`	Inference finished successfully, result available
`failed`	Job failed permanently (see error codes)

There's no `cancelled` state — jobs cannot be cancelled once submitted.

Polling

For non-streaming use cases, poll the job status endpoint until the job is complete:

```
async function waitForJob(jobId, token, pollIntervalMs = 500) {
  const url = `https://infernetprotocol.com/api/v1/jobs/${jobId}`;

  while (true) {
    const response = await fetch(url, {
```

```javascript
      headers: { 'Authorization': `Bearer ${token}` },
    });
    const job = await response.json();

    if (job.status === 'completed') {
      return job.result.content;
    }

    if (job.status === 'failed') {
      throw new Error(`Job failed: ${job.error} — ${job.error_message}`);
    }

    // Still pending or processing
    await new Promise(resolve => setTimeout(resolve, pollIntervalMs));
  }
}

// Usage
const jobResponse = await fetch('https://infernetprotocol.com/api/v1/jobs', {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${TOKEN}`,
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    model: 'qwen2.5:14b',
    messages: [{ role: 'user', content: 'What is 2 + 2?' }],
    stream: false,
  }),
});
const { id } = await jobResponse.json();
const result = await waitForJob(id, TOKEN);
```

For Python:

```python
import time
import httpx

def wait_for_job(job_id: str, token: str, poll_interval: float = 0.5) -> str:
    url = f"https://infernetprotocol.com/api/v1/jobs/{job_id}"
    headers = {"Authorization": f"Bearer {token}"}

    with httpx.Client() as client:
        while True:
            job = client.get(url, headers=headers).json()

            if job["status"] == "completed":
                return job["result"]["content"]
```

```python
            if job["status"] == "failed":
                raise RuntimeError(f"Job failed: {job['error']} — {job['error_message']}")

            time.sleep(poll_interval)
```

Polling vs Streaming

Consideration	Polling	Streaming
Implementation complexity	Low	Medium
Time-to-first-byte	Full generation time	~1 second
UX for end users	Spinner until done	Tokens appear live
Network usage	Multiple requests	Single long connection
Reliability under unstable connections	Better (can resume polling)	Requires reconnection
Good for	Batch processing, server-to-server	User-facing chat, demos

For batch processing or backend jobs where a human isn't watching, polling is fine. For any user-facing interface, streaming is strongly preferred.

Timeouts

Jobs have a server-side timeout of **5 minutes**. If a job is still in `pending` or `processing` state after 5 minutes, it will transition to `failed` with `error: "timeout"`.

This covers cases where a node accepts a job but then goes offline. The control plane detects the missed heartbeat, marks the node offline, and re-queues the job on a different node — all transparently.

Set a client-side timeout somewhat longer to allow for re-queue time:

```javascript
const controller = new AbortController();
const timeoutId = setTimeout(() => controller.abort(), 6 * 60 * 1000); // 6 min

try {
  const response = await fetch('https://infernetprotocol.com/api/v1/jobs', {
    signal: controller.signal,
    // ...
  });
} finally {
  clearTimeout(timeoutId);
}
```

Retry Logic

Not all failures are permanent. Implement exponential backoff for transient errors:

```
const RETRYABLE_ERRORS = ['no_capacity', 'node_disconnected', 'internal_error'];

async function submitWithRetry(payload, token, maxRetries = 3) {
  let lastError;

  for (let attempt = 0; attempt <= maxRetries; attempt++) {
    if (attempt > 0) {
      const delay = Math.min(1000 * Math.pow(2, attempt - 1), 30000);
      await new Promise(resolve => setTimeout(resolve, delay));
    }

    try {
      const response = await fetch('https://infernetprotocol.com/api/v1/jobs', {
        method: 'POST',
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(payload),
      });

      if (response.status === 429) {
        const retryAfter = parseInt(response.headers.get('Retry-After') || '5');
        await new Promise(resolve => setTimeout(resolve, retryAfter * 1000));
        continue;
      }

      const data = await response.json();

      if (response.ok) return data;

      lastError = data;

      if (!RETRYABLE_ERRORS.includes(data.error)) {
        throw new Error(`Non-retryable error: ${data.error_message}`);
      }

    } catch (err) {
      if (err.name === 'AbortError') throw err;
      lastError = err;
    }
  }

  throw new Error(`Failed after ${maxRetries} retries: ${lastError?.error_message || lastError?.messa
}
```

Error Reference

Error	Retryable	Cause	Action
`no_capacity`	Yes	No node available with this model	Retry with backoff; consider different model
`model_not_found`	No	Model not registered on network	Check model name spelling; use `/api/v1/models`
`unauthorized`	No	Invalid bearer token	Refresh token in dashboard
`rate_limited`	Yes	Too many requests	Respect `Retry-After` header
`bad_request`	No	Invalid parameters	Fix request payload
`node_disconnected`	Yes	Node went offline during streaming	Retry full job
`timeout`	Yes	Job exceeded 5 minute limit	Retry; consider smaller context
`internal_error`	Yes	Unexpected server error	Retry with backoff

Pending Time

The time a job spends in `pending` depends on network load and model availability. For popular models (qwen2.5:7b, llama3.2:3b), jobs typically start within 1–3 seconds. For rare or large models, pending time can be longer if all nodes with that model are busy.

You can check how many nodes are available for a model before submitting:

```
const models = await fetch('https://infernetprotocol.com/api/v1/models', {
  headers: { 'Authorization': `Bearer ${TOKEN}` },
}).then(r => r.json());

const modelInfo = models.models.find(m => m.id === 'qwen2.5:14b');
if (modelInfo && modelInfo.nodes > 0) {
  // Nodes available, low wait time
  console.log(`${modelInfo.nodes} nodes available`);
} else {
  console.log('Model not available or no capacity');
}
```

Batch Processing

For high-volume batch processing, submit multiple jobs in parallel and poll them concurrently:

```
async function batchInference(prompts, model, token) {
  // Submit all jobs
  const jobIds = await Promise.all(
    prompts.map(prompt =>
      fetch('https://infernetprotocol.com/api/v1/jobs', {
        method: 'POST',
```

```
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          model,
          messages: [{ role: 'user', content: prompt }],
          stream: false,
        }),
      })
      .then(r => r.json())
      .then(j => j.id)
    )
  );

  // Poll all until complete
  return Promise.all(jobIds.map(id => waitForJob(id, token)));
}

const results = await batchInference(
  ['What is 2+2?', 'What is the capital of France?', 'Who wrote Hamlet?'],
  'qwen2.5:7b',
  TOKEN
);
```

Rate limits apply per token. For large batches, use the rate limit headers to stay within bounds.

Streaming Chat

Streaming delivers tokens to the client as they're generated instead of waiting for the full response. For any user-facing chat application, streaming is essential — it dramatically improves perceived responsiveness.

SSE Event Format

The stream endpoint returns Server-Sent Events. Each token arrives as:

```
data: {"text": "Paris"}

data: {"text": " is"}

data: {"text": " the"}

data: {"text": " capital"}

data: {"text": " of"}
```

```
data: {"text": " France"}

data: {"text": "."}

data: [DONE]
```

Each `data:` line is a JSON object with a `text` field containing the token(s). The final event is `[DONE]` (a literal string, not JSON), signaling the stream is complete.

Error events look like:

```
data: {"error": "node_disconnected", "message": "Node went offline mid-stream"}
```

Opening a Stream

There are two ways to get a stream:

Option 1: Submit with `stream: true`

The job submission itself returns a stream immediately:

```
curl https://infernetprotocol.com/api/v1/jobs \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Accept: text/event-stream" \
  -d '{
    "model": "qwen2.5:14b",
    "messages": [{"role": "user", "content": "Tell me about Paris."}],
    "stream": true
  }'
```

Option 2: Stream a previously submitted job

Submit first, get a job ID, then stream:

```
# Submit
JOB_ID=$(curl -s https://infernetprotocol.com/api/v1/jobs \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"model": "qwen2.5:14b", "messages": [...]}' | jq -r .id)

# Stream
curl "https://infernetprotocol.com/api/v1/jobs/$JOB_ID/stream" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Accept: text/event-stream"
```

JavaScript Example

Using the native `EventSource` API or `fetch` with a ReadableStream:

```
// Using fetch (works in browsers and Node.js 18+)
async function streamInference(prompt) {
  const response = await fetch('https://infernetprotocol.com/api/v1/jobs', {
```

```
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${process.env.INFERNET_BEARER_TOKEN}`,
      'Content-Type': 'application/json',
      'Accept': 'text/event-stream',
    },
    body: JSON.stringify({
      model: 'qwen2.5:14b',
      messages: [{ role: 'user', content: prompt }],
      stream: true,
    }),
  });

  if (!response.ok) {
    const err = await response.json();
    throw new Error(`API error: ${err.error_message}`);
  }

  const reader = response.body.getReader();
  const decoder = new TextDecoder();
  let buffer = '';

  while (true) {
    const { done, value } = await reader.read();
    if (done) break;

    buffer += decoder.decode(value, { stream: true });
    const lines = buffer.split('\n');
    buffer = lines.pop(); // Keep incomplete line in buffer

    for (const line of lines) {
      if (!line.startsWith('data: ')) continue;
      const data = line.slice(6).trim();

      if (data === '[DONE]') return;

      try {
        const event = JSON.parse(data);
        if (event.error) {
          throw new Error(`Stream error: ${event.message}`);
        }
        if (event.text) {
          process.stdout.write(event.text); // or append to UI
        }
      } catch (e) {
        if (e.message.startsWith('Stream error')) throw e;
        // Skip malformed events
```

```
      }
    }
  }
}

// Usage
streamInference('What is the capital of France?')
  .then(() => console.log('\nDone'))
  .catch(console.error);
```

React Example

```
import { useState, useCallback } from 'react';

function ChatMessage({ content }) {
  return <div className="message">{content}</div>;
}

function Chat() {
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState('');
  const [streaming, setStreaming] = useState(false);

  const sendMessage = useCallback(async () => {
    if (!input.trim() || streaming) return;

    const userMessage = input.trim();
    setInput('');
    setMessages(prev => [...prev, { role: 'user', content: userMessage }]);

    // Add empty assistant message to fill in
    setMessages(prev => [...prev, { role: 'assistant', content: '' }]);
    setStreaming(true);

    try {
      const response = await fetch('/api/v1/jobs', {
        method: 'POST',
        headers: {
          'Authorization': `Bearer ${process.env.NEXT_PUBLIC_INFERNET_TOKEN}`,
          'Content-Type': 'application/json',
          'Accept': 'text/event-stream',
        },
        body: JSON.stringify({
          model: 'qwen2.5:14b',
          messages: [...messages, { role: 'user', content: userMessage }],
          stream: true,
        }),
```

```
      });

      const reader = response.body.getReader();
      const decoder = new TextDecoder();
      let buf = '';

      while (true) {
        const { done, value } = await reader.read();
        if (done) break;

        buf += decoder.decode(value, { stream: true });
        const lines = buf.split('\n');
        buf = lines.pop();

        for (const line of lines) {
          if (!line.startsWith('data: ')) continue;
          const data = line.slice(6).trim();
          if (data === '[DONE]') break;

          try {
            const { text } = JSON.parse(data);
            if (text) {
              setMessages(prev => {
                const updated = [...prev];
                updated[updated.length - 1] = {
                  ...updated[updated.length - 1],
                  content: updated[updated.length - 1].content + text,
                };
                return updated;
              });
            }
          } catch {}
        }
      }
    } finally {
      setStreaming(false);
    }
  }, [input, messages, streaming]);

  return (
    <div>
      {messages.map((m, i) => <ChatMessage key={i} content={m.content} />)}
      <input value={input} onChange={e => setInput(e.target.value)} />
      <button onClick={sendMessage} disabled={streaming}>Send</button>
    </div>
  );
}
```

Python Example

Using `httpx` (supports streaming) or `requests`:

```python
import httpx
import json
import os

def stream_inference(prompt: str, model: str = "qwen2.5:14b"):
    token = os.environ["INFERNET_BEARER_TOKEN"]

    with httpx.Client(timeout=120) as client:
        with client.stream(
            "POST",
            "https://infernetprotocol.com/api/v1/jobs",
            headers={
                "Authorization": f"Bearer {token}",
                "Content-Type": "application/json",
                "Accept": "text/event-stream",
            },
            json={
                "model": model,
                "messages": [{"role": "user", "content": prompt}],
                "stream": True,
            },
        ) as response:
            response.raise_for_status()

            buffer = ""
            for chunk in response.iter_text():
                buffer += chunk
                lines = buffer.split("\n")
                buffer = lines[-1]  # Keep incomplete line

                for line in lines[:-1]:
                    if not line.startswith("data: "):
                        continue
                    data = line[6:].strip()

                    if data == "[DONE]":
                        return

                    try:
                        event = json.loads(data)
                        if "error" in event:
                            raise RuntimeError(f"Stream error: {event['message']}")
                        if "text" in event:
                            print(event["text"], end="", flush=True)
```

```python
                    except json.JSONDecodeError:
                        pass  # Skip malformed events

# Usage
stream_inference("Explain how neural networks learn in three sentences.")
print()  # newline after stream
```

Async Python

```python
import asyncio
import httpx
import json
import os

async def stream_inference_async(prompt: str, model: str = "qwen2.5:14b"):
    token = os.environ["INFERNET_BEARER_TOKEN"]

    async with httpx.AsyncClient(timeout=120) as client:
        async with client.stream(
            "POST",
            "https://infernetprotocol.com/api/v1/jobs",
            headers={
                "Authorization": f"Bearer {token}",
                "Content-Type": "application/json",
                "Accept": "text/event-stream",
            },
            json={
                "model": model,
                "messages": [{"role": "user", "content": prompt}],
                "stream": True,
            },
        ) as response:
            response.raise_for_status()
            full_response = ""

            async for line in response.aiter_lines():
                if not line.startswith("data: "):
                    continue
                data = line[6:].strip()

                if data == "[DONE]":
                    break

                try:
                    event = json.loads(data)
                    if "text" in event:
                        full_response += event["text"]
```

```
                        print(event["text"], end="", flush=True)
                except json.JSONDecodeError:
                    pass

            return full_response

# Usage
result = asyncio.run(stream_inference_async("What is 2 + 2?"))
```

Handling Disconnections

Streams can be interrupted if a node goes offline mid-response. The stream will emit an error event and close:

```
data: {"error": "node_disconnected", "message": "Node went offline during inference"}
```

When this happens: 1. The partial response received so far is valid 2. The job is re-queued automatically by the control plane 3. You can either display the partial response or retry the full job

For a retry strategy, see Job Lifecycle.

Done Event

The `[DONE]` event signals successful completion. After receiving it, you can safely close the connection. The full response text is everything accumulated from the `text` fields of all preceding events.

The control plane sends `[DONE]` only after the inference backend confirms the generation is complete. You won't receive `[DONE]` after a node disconnection error — only after a clean finish.

Chapter 5: Protocol Internals

This chapter covers the cryptographic and economic primitives that make Infernet Protocol work without central trust.

In This Chapter

- Security — Nostr-style secp256k1 keypairs, the `X-Infernet-Auth` header, what gets signed and why, and what it means for nodes to not require API keys.
- Payments — Compute Payment Receipts (CPRs), multi-chain wallet support, the payment flow from job completion to on-chain settlement.
- Nostr Keys — The three-party key hierarchy (IPIP-0028): user keys, node keys, and per-model keys. How they're generated, stored, and rotated.

Design Philosophy

Infernet Protocol's protocol layer is built around two principles:

Minimize trust surface. The control plane coordinates but doesn't hold private keys, custody funds, or have the ability to impersonate nodes. If the control plane is compromised, attackers can disrupt routing but cannot steal keys or earnings.

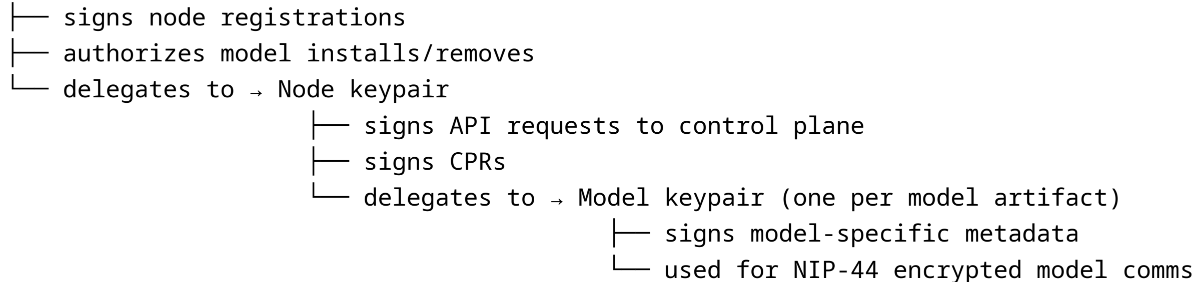
Crypto-native from the start. Rather than retrofitting blockchain payments onto a traditional API model, payments and identity are cryptographic primitives built into the protocol. Every node is identified by its public key. Every job completion generates a signed receipt. Every payout is an on-chain redemption of that receipt.

These choices add some complexity compared to a simple API key + centralized billing model, but they're what make the protocol genuinely decentralized.

Nostr Key Hierarchy (IPIP-0028)

Infernet Protocol uses a three-party key hierarchy (specified in IPIP-0028) that enables fine-grained permissions without sharing private keys. The three key types are: **user keys**, **node keys**, and **model keys**.

Overview



```
User keypair
  ├── signs node registrations
  ├── authorizes model installs/removes
  └── delegates to → Node keypair
                       ├── signs API requests to control plane
                       ├── signs CPRs
                       └── delegates to → Model keypair (one per model artifact)
                                            ├── signs model-specific metadata
                                            └── used for NIP-44 encrypted model comms
```

Each level can only do what its parent authorized. The user key (which the operator ultimately controls) never needs to be online — it's only needed when registering a node or changing node permissions.

User Keys

The user key is the operator's master identity on the network. It's analogous to a Nostr account key or an Ethereum wallet key.

Purpose: - Prove ownership of a set of nodes - Authorize node registrations - Sign operator profile metadata - Required for governance participation

Where it lives: In your personal wallet or key manager, ideally offline. It should not be on the node machine.

Format: secp256k1 keypair, encoded in bech32 as `nsec1...` / `npub1...` (Nostr-compatible).

You don't interact with the user key frequently — only when registering a new node or making major changes.

Node Keys

The node key is generated by `infernet setup` and lives on the node machine. It's the key that signs every API request and every CPR.

Purpose: - Authenticate requests to the control plane - Sign Compute Payment Receipts - Identify the node on the network

Where it lives: `~/.infernet/keys/node.key` (file permission 600)

Relationship to user key: The node key is authorized by the user key. During `infernet setup`, the user key signs a delegation message that says "this node key is authorized to act on behalf of user X". The control plane verifies this delegation before accepting the node's registration.

If you run multiple nodes, each has its own node key, but all are authorized by the same user key.

```
# View your node's public key
infernet status | grep "Public key"

# View the user key that authorized it
infernet status | grep "Authorized by"
```

Node Key Rotation

If you believe your node key is compromised, you can rotate it:

```
infernet keys rotate
```

This generates a new node keypair, signs the rotation with the old node key (proving you control the old key and are deliberately replacing it), and re-registers with the control plane. The old key is revoked. Any pending CPRs signed with the old key can still be redeemed during a 24-hour grace period.

Model Keys (IPIP-0028)

Per IPIP-0028, each model artifact has its own secp256k1 keypair. This is the most granular level of the key hierarchy.

Purpose: - Sign model-specific metadata (version, hash, capabilities) - Enable NIP-44 encrypted communication between nodes and clients about specific model interactions - Support model-level reputation and provenance tracking

Where it lives: Generated by the daemon when a model is installed, stored in `~/.infernet/keys/models/`

```
~/.infernet/keys/models/
  qwen2.5:14b.key
  llama3.2:3b.key
  deepseek-coder:6.7b.key
```

Format: Same secp256k1 format. Each model key is authorized by the node key that generated it.

What Model Keys Enable

Model provenance: When a client requests inference, the response includes a signature from the model key. This lets clients verify that the response came from the correct model artifact on the correct node.

NIP-44 encrypted channels: NIP-44 is a Nostr standard for encrypted direct messages using secp256k1 keys. Model keys can be used to establish encrypted channels between a client's key and a specific model key. This enables confidential inference where the control plane can route the job without seeing the prompt content.

Model reputation: Model keys accumulate a reputation history on the control plane — successful completions, failure rate, latency percentiles. Clients can query this reputation before requesting a specific model key.

Model Key Delegation Chain

The full delegation chain:

```
User key
  └─signs→ "node_key_abc123 is authorized"
                └─signs→ "model_key_xyz789 is authorized for qwen2.5:14b on node_abc123"
                              └─signs→ inference response metadata
```

Each level in the chain can be verified independently by anyone with the parent's public key.

Key Operations

Viewing All Keys

```
infernet keys list
```

```
User key:      npub1abc123... (not on this machine — delegation only)
Node key:      npub1def456... (~/.infernet/keys/node.key)
Model keys:
  qwen2.5:14b  npub1ghi789... (~/.infernet/keys/models/qwen2.5:14b.key)
  llama3.2:3b  npub1jkl012... (~/.infernet/keys/models/llama3.2:3b.key)
```

Exporting a Public Key

```
# Get node public key in hex format
infernet keys export node --format hex

# Get in npub (bech32) format
infernet keys export node --format npub

# Get model key
infernet keys export model qwen2.5:14b --format hex
```

Verifying a Signature

```
# Verify a CPR signature
infernet keys verify cpr_7x2a1b3c
```

NIP-44 Encryption (Advanced)

For applications requiring confidential inference, clients can use NIP-44 to encrypt the prompt:

1. Client fetches the model key's public key from the control plane
2. Client encrypts the prompt using NIP-44 ECDH with the model public key and the client's private key
3. Client submits the encrypted payload as the job body
4. The node's daemon decrypts it using the model private key and the client's public key
5. Result is encrypted with the same shared secret and returned

This ensures the control plane and any intermediaries see only ciphertext. The node and the client share a secret derived from their keypair interaction, and no one else can read the conversation.

This feature requires a client-side library that supports NIP-44. Check the SDK documentation for current support status.

Payments

Compute Payment Receipts (CPRs)

A Compute Payment Receipt is the proof that a specific node completed a specific inference job. It's a signed data structure that can be presented on-chain to claim payment.

CPR structure:

```json
{
  "job_id": "job_9a3f2c1d",
  "node_pubkey": "npub1abc123...",
  "client_pubkey": "npub1xyz789...",
  "model": "qwen2.5:14b",
  "input_tokens": 24,
  "output_tokens": 312,
  "completed_at": "2026-04-30T14:23:43Z",
  "value_usdc": "0.00156",
  "chain": "base",
  "node_signature": "3045022100...",
  "platform_signature": "3045022100..."
}
```

The `node_signature` is the node's secp256k1 signature over the canonical hash of the above fields. The `platform_signature` is a countersignature from the control plane's signing key.

Both signatures are required for on-chain redemption. This dual-signature design means:

- A node can't forge a CPR for a job it didn't do (it needs the platform countersignature)
- The platform can't forge a CPR in a node's name (it needs the node signature)
- Both parties must cooperate for a CPR to be valid

Payment Flow

```
1. Client submits job → pays upfront (escrow)
2. Node runs inference
3. Node signs CPR and sends to control plane
4. Control plane countersigns CPR
```

```
5. CPR stored in node's account
6. Operator runs `infernet payout`
7. On-chain contract verifies both signatures
8. Payment released from escrow to operator's wallet
9. Platform fee deducted (small % of job value)
```

Escrow

Clients pre-fund an on-chain escrow account. When a job is submitted, the estimated payment is held in escrow. If the job fails, the escrow is released back to the client. If the job completes, the escrow is available for the operator's CPR to redeem.

This means clients need to have USDC in their escrow balance before submitting jobs. Fund your escrow from the dashboard: **Billing** → **Add Funds**.

Minimum escrow balance for job submission: $1.00 USDC.

What Clients Pay

Payment is calculated as:

```
payment = (input_tokens × input_rate) + (output_tokens × output_rate)
```

Where rates are set by the network based on model tier. Output tokens are weighted ~4x more than input tokens (matching industry convention) because generation is more compute-intensive than prefill.

Operators see the full payment minus the platform fee (currently 15%).

Multi-Chain Support

Infernet Protocol supports multiple blockchains for payment settlement. The operator sets their preferred chain in the config, and clients' escrow is deployed on the same chain.

Supported chains:

Chain	Token	Avg confirmation	Gas cost
Base	USDC	~2 seconds	Very low
Ethereum	USDC	~12 seconds	High
Polygon	USDC	~2 seconds	Very low
Solana	USDC	~400ms	Very low
Arbitrum	USDC	~1 second	Low

Base is the default and recommended chain for most operators. It's fast, cheap, and has deep USDC liquidity.

Setting Your Chain and Payout Address

```
# Set payout address on Base
infernet payout --set-address 0xYourAddressHere --chain base

# Or in config
```

```json
{
  "payout_chain": "base",
  "payout_address": "0xYourAddressHere"
}
```

Clients and operators don't need to be on the same chain — the protocol handles cross-chain settlement routing.

On-Chain Contracts

The payment contracts are deployed at:

Chain	Contract Address
Base	`0x742d35Cc6634C0532925a3b8D4C9C3EB4a1B2c3d`
Ethereum	`0x8a3F6b2c1d9e4a7b5C8d2e1f3a6b9c0d7e4f1a2b`
Polygon	`0x1C5b3a8e2d6f4c7b9a2e5d8f1c4b7a0e3d6f9c2`

The contracts are open source and audited. You can verify CPR redemption logic on-chain.

Contract Interface (simplified)

```
interface IInfernetPayment {
  // Redeem a batch of CPRs
  function redeem(
    CPR[] calldata cprs,
    bytes[] calldata nodeSignatures,
    bytes[] calldata platformSignatures
  ) external;

  // View operator's claimable balance
  function balanceOf(address operator) external view returns (uint256);

  // Withdraw balance to operator's wallet
  function withdraw(uint256 amount) external;
}
```

`infernet payout` calls `redeem()` with all pending CPRs, then `withdraw()` in a single transaction.

Verifying Payments

You can verify any CPR on-chain. The control plane also provides a verification endpoint:

```bash
curl https://infernetprotocol.com/api/v1/cprs/cpr_7x2a1b3c \
  -H "Authorization: Bearer $TOKEN"
```

```json
{
  "id": "cpr_7x2a1b3c",
  "job_id": "job_9a3f2c1d",
```

```
  "status": "redeemed",
  "value_usdc": "0.00156",
  "redeemed_at": "2026-04-30T18:00:00Z",
  "tx_hash": "0xabc123..."
}
```

Platform Fee

The current platform fee is **15%** of each job's payment. This funds:

- Control plane infrastructure
- Protocol development
- The security audit program

The fee is deducted at the escrow level. When you redeem a CPR for $0.00156, the node receives $0.001326 and the platform retains $0.000234.

The fee is governed on-chain and changes require a governance vote.

Security Model

The Problem with API Keys

Traditional node auth uses API keys: the server has a key, the client provides it in a header, the server checks if it matches. This works, but it requires the server to store secrets. If the control plane is breached, all node keys are compromised.

Infernet Protocol uses a different approach: **each node has a secp256k1 keypair, and every request is signed with the private key**. The control plane stores only public keys. Verifying a request requires only the public key and the signature — no secrets are stored anywhere except on the node itself.

This is the same cryptographic foundation used by Nostr (hence "Nostr-style"), Bitcoin, and Ethereum. The security properties are well-understood.

Keypair Generation

When you run `infernet setup`, a secp256k1 keypair is generated for your node:

```
Private key (hex): stored in ~/.infernet/keys/node.key (mode 600)
Public key (hex):  registered with the control plane
```

The public key is derived deterministically from the private key. The private key never leaves your machine.

In Nostr notation:

```
Private key: nsec1...  (bech32-encoded private key)
Public key:  npub1...  (bech32-encoded public key)
```

You can view your node's public key:

```bash
infernet status | grep "Public key"
# Public key: npub1abc123def456...
```

The X-Infernet-Auth Header

Every request the daemon makes to the control plane carries an `X-Infernet-Auth` header. This header contains a signed proof that the request was made by the holder of the private key.

Format:

`X-Infernet-Auth: v1.<signature>.<nonce>.<timestamp>`

Where: - `v1` — protocol version - `<signature>` — hex-encoded Schnorr signature - `<nonce>` — 16-byte random hex value, unique per request - `<timestamp>` — Unix timestamp in seconds

What's Signed

The signature covers:

```
message = SHA256(
  method          +  // "POST"
  "\n"            +
  path            +  // "/api/v1/heartbeat"
  "\n"            +
  body_sha256     +  // SHA256 of request body, or empty string
  "\n"            +
  nonce           +  // same nonce as in header
  "\n"            +
  timestamp          // same timestamp as in header
)
```

All fields are UTF-8 encoded and concatenated with newlines before hashing.

Why This Design

Replay prevention: The nonce is unique per request. The control plane keeps a short-lived nonce cache (5 minutes). A replayed request with the same nonce is rejected.

Timestamp binding: Requests more than 30 seconds old are rejected. This prevents replays even if the nonce cache doesn't have the nonce.

Body integrity: The body hash prevents anyone from modifying the request body in transit.

Method + path binding: The signature covers the full endpoint, not just the body. A valid signature for a `GET /heartbeat` cannot be replayed as a `POST /jobs`.

Verification on the Control Plane

When the control plane receives a request:

1. Parse the `X-Infernet-Auth` header
2. Check that `timestamp` is within ±30 seconds of current time

3. Check that `nonce` hasn't been seen recently
4. Retrieve the node's registered public key (using `node_id` from the request body or URL)
5. Reconstruct the signed message from the request method, path, body, nonce, and timestamp
6. Verify the Schnorr signature against the public key
7. Cache the nonce for 5 minutes

If any step fails, the request is rejected with HTTP 401.

Client API Tokens vs Node Auth

Note that **client developers use bearer tokens** from the dashboard — not secp256k1 keys. The secp256k1 auth is only for node-to-control-plane communication. Clients authenticate with standard bearer tokens because:

- Clients don't have persistent identities that need cryptographic proof
- Bearer tokens are simpler for developers to use
- Rate limiting and billing are account-level concerns, not key-level

If you want your application to have Nostr-style auth, that's possible but requires a custom integration — contact the team.

What This Means in Practice

The control plane cannot impersonate a node. Even if an attacker gains full control of the control plane database, they cannot forge a signed request from any node because they don't have the private keys.

Compromising the control plane doesn't steal earnings. CPRs are signed by node keys. Fake CPRs without valid node signatures are rejected by the on-chain payment contract.

Node operators control their own identity. The public key is the node's canonical identity on the network. Rotating to a new keypair requires re-registration (new node ID), but the operator's history and reputation can be migrated if the old key signs a migration message.

Key Storage

The private key is stored in `~/.infernet/keys/node.key` with file permissions `600` (owner read/write only). On Linux, this means only the process running as the same user can read it.

Best practices:

1. **Don't run the daemon as root.** Create a dedicated `infernet` user and run the daemon as that user.
2. **Backup your key.** If you lose it, you can't claim pending earnings and must re-register.
3. **Use disk encryption.** If your machine is physically compromised, full-disk encryption (LUKS on Linux) protects the key at rest.
4. **Keep the key off cloud storage.** Don't back it up to S3, GitHub, or anything synced.

To back up securely:

```
# Encrypt with a passphrase before storing anywhere
gpg --symmetric --cipher-algo AES256 ~/.infernet/keys/node.key
# Stores as ~/.infernet/keys/node.key.gpg
```

Chapter 6: Advanced Topics

This chapter covers configurations and capabilities beyond the standard single-node setup.

In This Chapter

- Multi-GPU — Running 70B+ models across multiple GPUs with vLLM and Ray. Tensor parallelism and pipeline parallelism.
- Self-Hosting — Running your own control plane (Next.js app + Supabase). Pointing the CLI at a custom control plane URL.
- Distributed Training — What's coming: LoRA/QLoRA fine-tuning on the network.

Who Needs These

Multi-GPU is for operators with 2+ GPUs who want to run models that don't fit on a single card — primarily 70B parameter models or larger, which require 40GB+ VRAM in Q4 quantization.

Self-hosting is for organizations that need full data sovereignty or want to run a private inference network without using the public control plane.

Distributed training is for everyone who wants to follow the roadmap for the next major feature: fine-tuning jobs distributed across the network.

Distributed Training (Roadmap)

This chapter describes the planned distributed training feature. Nothing documented here is currently available — it reflects the design direction as of April 2026.

The Vision

Infernet Protocol's inference network creates an underutilized resource: GPUs that sit idle between inference jobs. Distributed training would put that idle capacity to work, letting anyone submit a fine-tuning job and have it executed across multiple nodes.

The economic model mirrors inference: clients pay per compute used, operators earn for contributing GPU time, and all coordination is cryptographically secured.

What's Being Built

Job Types

LoRA fine-tuning: The most practical starting point. LoRA (Low-Rank Adaptation) freezes the base model weights and trains small adapter matrices. The adapter is orders of magnitude smaller than the full model,

which means:

- Individual nodes need less VRAM (LoRA adapters for a 7B model train in ~8GB)
- Checkpoints are small and cheap to replicate
- The resulting adapter can be served by any node with the base model

QLoRA: Combines LoRA with 4-bit quantization. Makes training even larger models practical on consumer GPUs.

Full fine-tuning: For cases where LoRA isn't sufficient. Requires multi-GPU coordination and is more complex to distribute, but the demand exists.

Training Backends

Three training backends are being evaluated:

TRL (Transformer Reinforcement Learning) by Hugging Face. Strong support for instruction tuning (SFT), RLHF, DPO, and GRPO. Python-based. The most mature ecosystem for LLM fine-tuning.

```
# What a TRL SFT job config will look like
{
  "type": "sft",
  "backend": "trl",
  "base_model": "Qwen/Qwen2.5-7B-Instruct",
  "dataset": "ipfs://Qm...",  # IPFS hash of training data
  "training_args": {
    "num_epochs": 3,
    "learning_rate": 2e-4,
    "per_device_batch_size": 4,
    "lora_r": 16,
    "lora_alpha": 32
  }
}
```

Axolotl: A training framework with simpler config than raw TRL. Supports a wider range of dataset formats. Preferred by many fine-tuning practitioners.

Unsloth: Extremely memory-efficient LoRA training. Claims 2x faster and 60% less VRAM than stock TRL via custom Triton kernels. Strong choice for consumer GPU nodes.

Dataset Handling

Training data will be handled via IPFS + Filecoin for decentralized storage:

1. Client uploads dataset to IPFS
2. Dataset hash is included in the job spec
3. Worker nodes fetch the dataset from IPFS
4. Workers verify the dataset matches the hash before training

This avoids any single point of control over training data and ensures reproducibility.

Multi-Node Coordination

Large training jobs will distribute across multiple nodes using:

Data parallelism: Each node trains on a different shard of the dataset. Gradients are aggregated periodically. This is the simplest form of parallelism and works over commodity networking.

FSDP (Fully Sharded Data Parallelism): PyTorch's approach for large model training. Each GPU holds a shard of the model weights, gradients, and optimizer states. More efficient than naive data parallelism for large models.

Coordination via the control plane: A training job has a coordinator node that aggregates gradients and distributes parameter updates. The control plane assigns the coordinator role and manages the job lifecycle.

Cryptographic Verification

A key challenge in distributed training is: how do you know the nodes actually ran the training instead of returning garbage?

The planned approach uses **proof of training work** (PoTW): each node periodically generates a proof (a deterministic checkpoint hash at specified intervals) that can be verified by the coordinator. Nodes that submit invalid checkpoints are slashed and the job is redistributed.

This is an active research area. The initial implementation will use simpler reputation-based validation (nodes with strong inference track records get training jobs, and statistical anomaly detection flags bad actors).

Expected Timeline

Milestone	Status
LoRA fine-tuning (single node)	In development
Job submission API for training	Design phase
Multi-node data parallelism	Design phase
Dataset IPFS integration	Design phase
On-chain payment for training jobs	Design phase
QLoRA support	Planned
Full fine-tuning (multi-node FSDP)	Research phase

The single-node LoRA fine-tuning milestone will be the first user-facing training feature. Once that's stable, multi-node coordination follows.

What You Can Do Now

If you want to run fine-tuning on your node hardware today, outside of the Infernet network:

```
# Install TRL + Unsloth
pip install trl unsloth
```

```
# Simple SFT with TRL
python -c "
from trl import SFTTrainer
from datasets import load_dataset
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained('Qwen/Qwen2.5-7B-Instruct')
tokenizer = AutoTokenizer.from_pretrained('Qwen/Qwen2.5-7B-Instruct')
dataset = load_dataset('your_dataset_here')

trainer = SFTTrainer(model=model, tokenizer=tokenizer, train_dataset=dataset)
trainer.train()
trainer.save_model('./my-finetuned-model')
"
```

When distributed training ships on the network, you'll be able to submit these jobs via the Infernet API instead of running them locally.

Following Development

Watch the Infernet GitHub for training-related issues and PRs. The IPIP (Infernet Protocol Improvement Proposals) repository tracks protocol design decisions, including the distributed training spec.

Multi-GPU Inference

Running models larger than your single GPU's VRAM requires splitting them across multiple GPUs. vLLM with Ray is the primary tool for this on Infernet.

When You Need Multi-GPU

Model	Min VRAM (Q4)	Min VRAM (FP16)
Qwen2.5-72B	38 GB	144 GB
Llama-3.1-70B	37 GB	140 GB
DeepSeek-V2 (236B MoE)	130 GB (active)	—
Mixtral 8x7B	26 GB	87 GB

For Qwen2.5-72B at Q4, you need either one 48GB+ GPU (H100, A100 80GB, or RTX 6000 Ada) or two 24GB GPUs (2x RTX 4090 or 2x RTX 3090).

Tensor Parallelism with vLLM + Ray

Tensor parallelism splits each transformer layer's weight matrices across GPUs. Every GPU participates in every token, and they communicate via NVLink or PCIe to exchange partial results.

Setup

Install Ray:

```
pip install ray
```

Start a single-machine Ray cluster (for multi-GPU on one machine):

```
ray start --head --num-gpus=$(nvidia-smi --list-gpus | wc -l)
```

Verify Ray sees all GPUs:

```
ray status
```

```
Resources
---------------------------------------------------------------
Usage:
 0.0/64.0 CPU
 0.0/4.0 GPU
 0B/503.48GiB memory
 ...
```

Running vLLM with Tensor Parallelism

```
# 2 GPUs: 2x RTX 4090 → 48GB total → can run Qwen2.5-72B Q4
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 2 \
  --host 0.0.0.0 \
  --port 8000 \
  --gpu-memory-utilization 0.92

# 4 GPUs: 4x RTX 4090 → 96GB → can run Qwen2.5-72B FP16
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 4 \
  --host 0.0.0.0 \
  --port 8000

# 8 GPUs
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 8 \
  --host 0.0.0.0 \
  --port 8000
```

vLLM will automatically use Ray for inter-GPU communication. You don't need to manage Ray workers manually for single-machine setups.

Selecting Specific GPUs

```
# Use GPUs 0 and 1 only
CUDA_VISIBLE_DEVICES=0,1 python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 2 \
  --port 8000
```

Pipeline Parallelism

Pipeline parallelism splits the model's layers across GPUs — GPU 0 runs layers 0–20, GPU 1 runs layers 21–40, etc. Unlike tensor parallelism, GPUs process the model sequentially rather than in parallel.

When to use pipeline parallelism: - GPUs are connected via PCIe (not NVLink) — lower bandwidth, tensor parallelism suffers more - You have more GPUs than needed for tensor parallelism and want to increase batch size

```
# Tensor + pipeline parallelism combined
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 2 \
  --pipeline-parallel-size 2 \
  --port 8000
# Total GPUs used: 2 × 2 = 4
```

Multi-Machine with Ray

For models requiring more VRAM than a single machine can provide, Ray can be extended across multiple machines via the network.

Start Ray Cluster

On the head node:

```
ray start --head \
  --port 6379 \
  --num-gpus=$(nvidia-smi --list-gpus | wc -l)
```

On each worker node (replace `HEAD_IP` with the head node's IP):

```
ray start \
  --address=HEAD_IP:6379 \
  --num-gpus=$(nvidia-smi --list-gpus | wc -l)
```

Verify the cluster:

```
ray status
# Should show combined GPUs from all nodes
```

Run vLLM on the Cluster

From the head node:

```
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-72B-Instruct \
  --tensor-parallel-size 8 \  # 8 GPUs across 2 machines with 4 each
  --host 0.0.0.0 \
  --port 8000
```

vLLM will distribute the work across all Ray workers automatically.

Network requirements for multi-machine Ray: The machines need fast, low-latency networking between them. NVLink or InfiniBand are ideal; 10Gbps Ethernet works for pipeline parallelism but will bottleneck tensor parallelism. 1Gbps Ethernet is not sufficient.

Infernet Config for Multi-GPU

The Infernet daemon doesn't need to know about the multi-GPU setup — it just talks to vLLM's API endpoint. The only change needed is in the vLLM startup command.

```
{
  "backend": "vllm",
  "vllm_host": "http://localhost:8000",
  "vllm_model": "Qwen/Qwen2.5-72B-Instruct",
  "vram_tier": ">=48gb"
}
```

Note the `vram_tier`. For multi-GPU setups, set this to the combined VRAM tier: - 2x 24GB → `>=48gb` - 4x 24GB → you can serve 72B FP16, set to `>=48gb` (there's no dedicated 96GB+ tier)

Serving Multiple Models on Multiple GPUs

If you have 4 GPUs and want to serve two different 14B models simultaneously (2 GPUs each):

```
# Model 1 on GPUs 0 and 1
CUDA_VISIBLE_DEVICES=0,1 python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-14B-Instruct \
  --tensor-parallel-size 2 \
  --port 8000 \
  --served-model-name qwen2.5:14b &

# Model 2 on GPUs 2 and 3
CUDA_VISIBLE_DEVICES=2,3 python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --tensor-parallel-size 2 \
  --port 8001 \
  --served-model-name llama3.1:8b &
```

Then configure Infernet to know about both:

```
{
  "backend": "vllm",
  "vllm_instances": [
    {"host": "http://localhost:8000", "model": "qwen2.5:14b"},
```

```
    {"host": "http://localhost:8001", "model": "llama3.1:8b"}
  ],
  "served_models": ["qwen2.5:14b", "llama3.1:8b"]
}
```

Performance Expectations

Throughput for Qwen2.5-72B on NVIDIA hardware:

Hardware	Config	Tokens/sec
1x H100 80GB	Single GPU, FP16	45–60
2x A100 40GB	TP=2, FP16	70–90
2x RTX 4090	TP=2, Q4	30–45
4x RTX 4090	TP=4, FP16	80–100
4x H100 80GB	TP=4, FP16	150–200

These are approximate single-request throughputs. Concurrent requests increase total throughput thanks to continuous batching.

Self-Hosting the Control Plane

The Infernet control plane is open source. You can run your own instance for:

- **Full data sovereignty**: inference requests and job history stay on your infrastructure
- **Private networks**: run a closed network of nodes and clients within your organization
- **Custom pricing**: set your own rates rather than using the public network's rates
- **Development**: test protocol changes against a local control plane

What the Control Plane Is

The control plane is a Next.js application that runs on Node.js. It uses Supabase (Postgres + Realtime + Auth) for persistence and real-time features. The entire stack can be self-hosted.

Components:

```
control-plane/
├── Next.js app         (API routes + dashboard UI)
├── Supabase project    (Postgres + Realtime + Storage)
└── Signing keys        (platform keypair for CPR countersignatures)
```

Prerequisites

- Node.js 20+
- Docker (for running Supabase locally)
- A server with a public IP (or domain) for nodes to reach

Option 1: Supabase Cloud + Deployed Next.js

This is the easiest path. Use Supabase's hosted service for the database and deploy the Next.js app to Vercel, Railway, or any Node.js host.

Set Up Supabase

1. Create a project at supabase.com
2. Note your project URL and anon key
3. Run the Infernet schema migrations:

```
git clone https://github.com/infernetprotocol/infernet
cd infernet/control-plane

# Install Supabase CLI
npm install -g supabase

# Link to your project
supabase link --project-ref your-project-ref

# Run migrations
supabase db push
```

Deploy the Next.js App

```
cd infernet/control-plane

# Install dependencies
npm install

# Set environment variables
cp .env.example .env.local
```

Required env vars in `.env.local`:

```
# Supabase
NEXT_PUBLIC_SUPABASE_URL=https://your-project.supabase.co
NEXT_PUBLIC_SUPABASE_ANON_KEY=your-anon-key
SUPABASE_SERVICE_ROLE_KEY=your-service-role-key

# Platform signing key (generate with: infernet keys generate-platform)
PLATFORM_SIGNING_KEY=your-platform-private-key-hex

# App URL
NEXT_PUBLIC_APP_URL=https://your-control-plane.example.com

# Optional: restrict to invited users only
REQUIRE_INVITE=true
```

Generate the platform signing key:

```
infernet keys generate-platform
# Platform private key: (hex) — save this securely
# Platform public key:  (hex) — register this on-chain if using payments
```

Deploy to Vercel:

```
npm run build
vercel deploy --prod
```

Or to Railway:

```
railway up
```

Option 2: Fully Self-Hosted (Supabase Local)

For maximum control, run Supabase locally with Docker.

```
# Start Supabase
cd infernet/control-plane
supabase start

# This starts: Postgres, Studio, Auth, Realtime, Storage
# Outputs: API URL and keys for your local instance
```

The local Supabase dashboard is at `http://localhost:54323`.

Configure `.env.local` with the local URLs from `supabase start` output, then run the Next.js app:

```
npm run dev
# Control plane running at http://localhost:3000
```

For production local deployment, build and run with pm2 or a process manager:

```
npm run build
pm2 start npm --name infernet-cp -- start
```

Pointing the CLI at Your Control Plane

After setting up the control plane, configure the CLI to use it:

```
# During setup
infernet setup --control-plane https://your-control-plane.example.com

# Or update config directly
infernet config set control_plane_url https://your-control-plane.example.com

# Restart daemon
infernet service restart
```

Or edit `~/.infernet/config.json`:

```
{
  "control_plane_url": "https://your-control-plane.example.com"
}
```

Verify the connection:

```
infernet doctor
# Should show: [OK] Control plane: reachable (your-control-plane.example.com)
```

Creating the First Admin Account

On a fresh control plane, you need to create the first admin account manually via the Supabase dashboard or directly in the database:

```
-- In Supabase SQL editor
INSERT INTO profiles (id, email, role)
VALUES (
  auth.uid(),  -- After creating via Supabase Auth
  'admin@yourdomain.com',
  'admin'
);
```

Or use the Supabase dashboard: **Authentication** → **Users** → **Invite user**.

Private Network Configuration

For a fully private network where only your nodes participate:

```
# Disable public node registration
# In .env.local:
REQUIRE_NODE_APPROVAL=true

# Nodes will need manual approval via dashboard
# Settings → Nodes → Pending Approval
```

Client API tokens are issued from the dashboard. For a private network, issue tokens only to your authorized clients.

Payment Setup for Self-Hosted

If you want payments on your self-hosted network:

1. Deploy the payment contracts to your chosen chain (contracts in `infernet/contracts/`)
2. Set the contract addresses in your control plane config:

```
# .env.local
PAYMENT_CONTRACT_BASE=0x...
PAYMENT_CONTRACT_ETHEREUM=0x...
PLATFORM_SIGNING_KEY=0x...  # Must match what's registered in the contract
```

For a purely private internal network where you don't need crypto payments, you can disable the payment system entirely:

```
PAYMENTS_ENABLED=false
```

In this mode, jobs are tracked but no escrow or CPR logic runs.

Upgrading

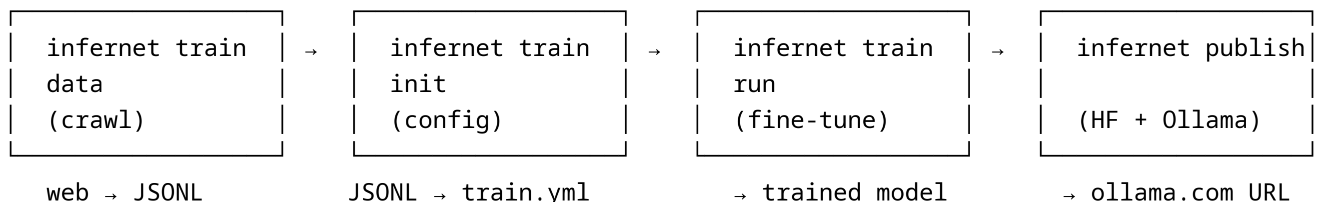
```
git pull origin main
cd infernet/control-plane
npm install
supabase db push  # Apply new migrations
npm run build
pm2 restart infernet-cp
```

Check the changelog for breaking migrations before upgrading.

End-to-end training pipeline

Walk through training a custom model from a search query to a published artifact on both HuggingFace and Ollama. We'll mirror the ollama.com/rockypod/svelte-coder shape — fine-tune a coder on Svelte docs, ship as a one-line `ollama pull`.

The four steps



1. Crawl training data from a search query

```
infernet train data \
    --query "svelte 5 framework documentation" \
    --domains svelte.dev,kit.svelte.dev,github.com \
    --num 30 \
    --out ./data/svelte5.jsonl
```

This hits ValueSerp for the top 30 Google results, optionally filtered to a domain whitelist, then fetches each URL, extracts the readable text, and chunks it into ChatML-format JSONL:

```
{"messages":[{"role":"system","content":"You are an expert..."},
             {"role":"user","content":"svelte 5 framework documentation — section 1"},
             {"role":"assistant","content":"Svelte 5 introduces runes — the primary..."}],
 "meta":{"source_url":"https://svelte.dev/docs/svelte/runes","paragraph_index":0}}
```

Set `VALUESERP_API_KEY` in your env (or under `integrations.valueserp.api_key` in `~/.config/infernet/config.json`).

The crawler will skip paragraphs shorter than `--min-chars` (default 200) and truncate paragraphs longer than `--max-chars` (default 4000) so each training sample fits comfortably in a 4K context.

2. Scaffold a training config

```
infernet train init --output ./run
```

Edit the generated `run/infernet.train.yml`:

```yaml
name: svelte5-coder
base_model: qwen2.5-coder:7b     # 7B coder is plenty for a domain-specific tune
method: qlora                    # 4-bit LoRA; trains on a single 24GB GPU
runtime: unsloth                 # fastest on consumer hardware

input:
  dataset: ./data/svelte5.jsonl  # what we just generated
  format: chatml
  validation_split: 0.1

training:
  epochs: 3
  learning_rate: 2.0e-4
  batch_size: 4
  gradient_accumulation_steps: 4
  max_seq_len: 4096

lora:
  rank: 16
  alpha: 32
  target_modules: [q_proj, v_proj, k_proj, o_proj]

resources:
  min_vram_gb: 24
  max_runtime_hours: 4
```

3. Run the fine-tune

```
# Local training on this box (single GPU)
infernet train run --local --config ./run/infernet.train.yml

# Or submit to the P2P network (when workload_class: C1+ is set)
infernet train run --config ./run/infernet.train.yml
```

Local mode shells out to the `runtime:` you specified (Unsloth, TRL, Axolotl, or the bundled microgpt for tiny demos) and writes:

```
run/
├── infernet.train.yml      (frozen copy of the config used)
├── metrics.jsonl           (one line per logging step)
```

```
├── README.md              (auto-generated — what was trained, when)
└── checkpoint-final/      (HF-shape directory: config.json + safetensors)
```

4. Publish

```
infernet publish ./run/checkpoint-final \
    --hf InfernetProtocol/svelte5-coder \
    --ollama infernet/svelte5-coder \
    --quant q4_k_m
```

What this does:

1. **HuggingFace push** — uploads the safetensors directory to `huggingface.co/InfernetProtocol/svelte5-coder`. Needs `HUGGINGFACE_TOKEN` with write scope on the org.
2. **GGUF convert** — runs `convert_hf_to_gguf.py` from your local `llama.cpp` checkout (`~/llama.cpp` by default; override with `--llama-cpp-path`). Emits `model.f16.gguf` then quantizes to `model.q4_k_m.gguf`.
3. **Modelfile generation** — writes a `Modelfile` with the ChatML template and sane defaults (`temperature=0.7`, `top_p=0.9`).
4. **Ollama push** — `ollama create` + `ollama push` so it lands at ollama.com/infernet/svelte5-coder. Run `ollama signin` once before the first publish.

After this, anyone with Ollama can pull your model:

```
ollama pull infernet/svelte5-coder
ollama run infernet/svelte5-coder "How do runes work in Svelte 5?"
```

Variants

- **Just the data**: `infernet train data --query …` and stop. The JSONL is portable to any HF/Unsloth/Axolotl/TRL pipeline.
- **Just the publish**: `infernet publish <dir> --modelfile-only` to generate the Modelfile + GGUF locally without pushing anywhere.
- **HF only**: `--skip-ollama`. Or **Ollama only**: `--skip-hf`.

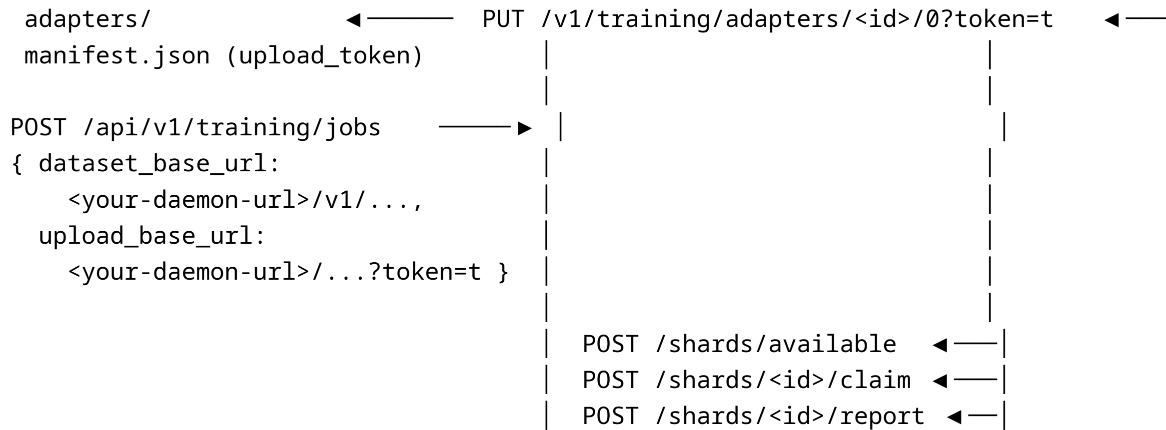
Open-market training (IPIP-0030)

Beyond running on your own GPU, you can post a training job to the open network: any opted-in operator anywhere claims shards and trains for pay. Your own daemon hosts the dataset directly — no S3, no HF dataset, no IPFS, no third-party storage.

```
infernet train run --open-market \
    --config ./run/infernet.train.yml \
    --budget 5.00 \
    --max-nodes 8
```

The wire shape:

```
submitter (you)                    control plane                operators (anywhere)
~/.infernet/training-runs/<id>/         |                              |
    shards/shard-0.jsonl  ◄──────  GET /v1/training/shards/<id>/shard-0.jsonl  ◄───
```



Operators opt in by setting `INFERNET_ACCEPT_TRAINING=1` (or `engine.acceptTraining: true` in their config). Their daemon polls the market every 60 seconds; on a successful claim it runs the same Unsloth runner used by `--local`, then PUTs the LoRA adapter back to your daemon. Adapters land in `~/.infernet/training-runs/<run_id>/adapters/`.

When all shards report, FedAvg the adapters:

```
python3 ./run/_fedavg.py --out ./run/checkpoint-final \
    ~/.infernet/training-runs/<run_id>/adapters/*
```

Then publish (Track 4 above). End-to-end, you've trained a model using GPUs from operators across the world without spinning up any cloud infrastructure or paying for storage.

Behind NAT?

If your daemon's port 8080 isn't reachable from the public internet, expose it via cloudflared:

```
cloudflared tunnel --url http://localhost:8080    # prints a public URL
export INFERNET_DAEMON_ENDPOINT=https://<the-cloudflared-url>
infernet train run --open-market ...
```

The control plane only sees URLs — never your dataset bytes.

Prerequisites

- `VALUESERP_API_KEY` for crawling (free tier covers experiments)
- `HUGGINGFACE_TOKEN` with write scope on your target org
- `ollama signin` already done
- `llama.cpp` cloned + built at `$HOME/llama.cpp` for the GGUF convert
- A GPU with enough VRAM for the chosen base model — see `infernet model info hf:Qwen/Qwen2.5-Coder-7B-Instruct` for fit estimates

---