Cestus Network

A Decentralized AI Execution Substrate

Whitepaper · cestus.network · 2026

Abstract

AI inference today works like a utility where a small number of operators own the grid. OpenAI keeps the models. AWS keeps the margin. The open-weight ecosystem has produced models that rival their proprietary counterparts on most tasks, yet accessing them at production scale still means building your own infrastructure or accepting someone else's terms. The arrangement should be different. A permissionless network where GPU operators worldwide contribute hardware, serve open-weight models, and earn from every token their machines produce. Where developers access the full range of what open AI can do through a single endpoint, without asking permission from any provider. Where no content policy is enforced at the platform level, because no single party controls the platform. The result is an inference market where the cost of intelligence is set by the hardware that produces it and access belongs to everyone.

> *"AI is no longer a single breakthrough or application. It is essential infrastructure. Every company will use it. Every nation will build it. "*
>
> -- Jensen Huang, Founder and CEO, NVIDIA.

Contents

---

1. Introduction

The platforms that control how intelligence is served will define the next decade of software. Right now, that control belongs to a handful of companies.

On one side: centralized providers. OpenAI, Google, Anthropic, AWS. Polished products, reliable uptime, and a set of terms you accept or walk away from. Their content policies are non-negotiable. Their model selection is theirs to control. Their pricing reflects the margins of a data center business. And the models themselves are closed: you can use them but never audit, verify, or own them.

On the other side: a growing ecosystem of decentralized compute networks. Projects across the DePIN space have demonstrated that idle GPU supply can be aggregated and monetized at scale. The infrastructure argument is proven. But these projects are predominantly supply-side plays: they solve the hardware coordination problem, not the developer experience problem. To get production inference out of most of them, you still need to provision nodes, manage model serving, build an API layer, handle billing, and wire together a stack that the centralized providers hand you in one call. Most developers never make it that far.

Cestus closes that gap. Not by competing with one side or the other, but by being something neither has delivered: a complete, permissionless inference platform where open models are served through a polished developer API and consumer-facing interfaces, with no closed weights, no centralized content policy, and no single company controlling the infrastructure. One API. Every open model. Live on mainnet today.

Several design decisions set Cestus apart from both categories.

Unified multi-modal pipeline. Text inference, image generation, and video generation all run on the same swarm, routed by the same algorithm, billed through the same token economy. Image and video jobs use an async queue with artifact storage (necessary because generation times for these modalities range from seconds to

many minutes) while text inference remains synchronous. No configuration change is needed to switch between modalities.

VRAM-aware model management with cache pricing. The Rebus node software dynamically loads and unloads models from VRAM based on live demand signals from the swarm. Input tokens that hit the node's cache are billed at a lower rate than uncached tokens, because they consume less compute.

Full-stack product, single infrastructure. The same swarm that serves API requests also backs a chat interface, an image generation interface, and a video generation interface accessible to any end user in a browser. There is no separate infrastructure for consumer products. A developer building on the API and an end user opening the chat UI are hitting the same nodes, routed by the same algorithm, billed by the same protocol.

Single-token economy. All platform activity: developer API charges, consumer UI charges, node operator earnings, platform fees; flows through CES. Prices are denominated in CES with a USD mapping updated periodically. There is no fiat payment path at the protocol layer and no multi-token complexity.

The network compounds on itself as it grows. More operators mean more models available in more regions. Broader model coverage attracts more developers and users. More usage increases earnings per node, which attracts more operators. Each new capability (video generation today, model training in Phase 2) runs on the same swarm, multiplying its value without requiring new infrastructure from anyone but the operators who choose to join.

---

2. Problem Statement

2.1 The Centralization Failure

Modern AI inference is dominated by a small number of centralized providers. This concentration produces four structural failure modes that worsen as AI becomes more deeply embedded in critical applications.

Vendor lock-in. API schemas, model availability, rate limits, and pricing are controlled entirely by the provider. Any change in policy propagates immediately to all dependent products. Developers building on centralized inference APIs are building on borrowed ground.

Model opacity. Closed providers do not publish model weights, training data, or architectural specifications. Developers cannot audit, reproduce, or verify the behavior of the model their product depends on. A silent model update can change the behavior of every downstream application without notice.

Unilateral censorship. Content policies are applied without developer consent. Healthcare, legal, scientific, and Web3 applications that require specific model behaviors face modification at any time. There is no recourse and no alternative within the centralized paradigm.

Web3 incompatibility. On-chain applications, DAOs, and autonomous agents that call centralized inference APIs introduce a critical point of trust failure. A nominally decentralized protocol with a centralized AI dependency is not decentralized.

2.2 The Fragmentation of Existing Alternatives

Decentralized compute exists. What does not exist is a complete, developer-ready platform built on top of it.

Raw compute marketplaces provide GPU access but require the developer to design and operate their own model serving infrastructure, autoscaling, and API layer. They solve a hardware procurement problem. Cestus solves the entire developer experience.

Decentralized compute networks coordinate the supply side of GPU infrastructure but provide no unified developer API, no metered billing, and no consumer product surface. Access varies per subnet or validator and is not standardized across the network. The architecture is designed to reward compute contributors, not to serve developers calling an endpoint. These networks and Cestus operate at different layers of the stack and address different participants: one coordinates hardware providers, the other serves builders and end users.

Decentralized storage networks offer data persistence, not inference execution. They solve a different problem entirely.

A separate category of decentralized network uses token incentives to coordinate compute contributors directly, rewarding participants based on peer ranking or task completion scores. These networks are designed to produce and evaluate intelligence, not to serve it. Access is mediated by subnet-specific validators rather than a standardized API, and pricing is not metered per token. Cestus operates at a different layer: it assumes the compute exists and focuses entirely on making that compute accessible to developers and end users through a production-grade platform.

The missing piece is not compute. It is the complete platform layer: one that makes compute useful to a developer without a PhD in distributed systems, and simultaneously accessible to an end user without any technical knowledge at all. No existing platform delivers all of the following simultaneously: a production inference API, consumer-facing interfaces for chat, image generation, and video generation, intelligent routing across a heterogeneous node network, per-token metered billing, Web3-native authentication, and a roadmap to training and model ownership. Cestus does.

2.3 Content Neutrality as an Architectural Property

Centralized inference providers apply content policies at the platform layer, uniformly and without developer override. The practical consequence is that entire categories of legitimate application, certain healthcare queries, legal reasoning over sensitive case material, security research, scientific investigation into restricted domains encounter model behavior that is modified or refused at the provider's discretion, with no alternative within the same platform.

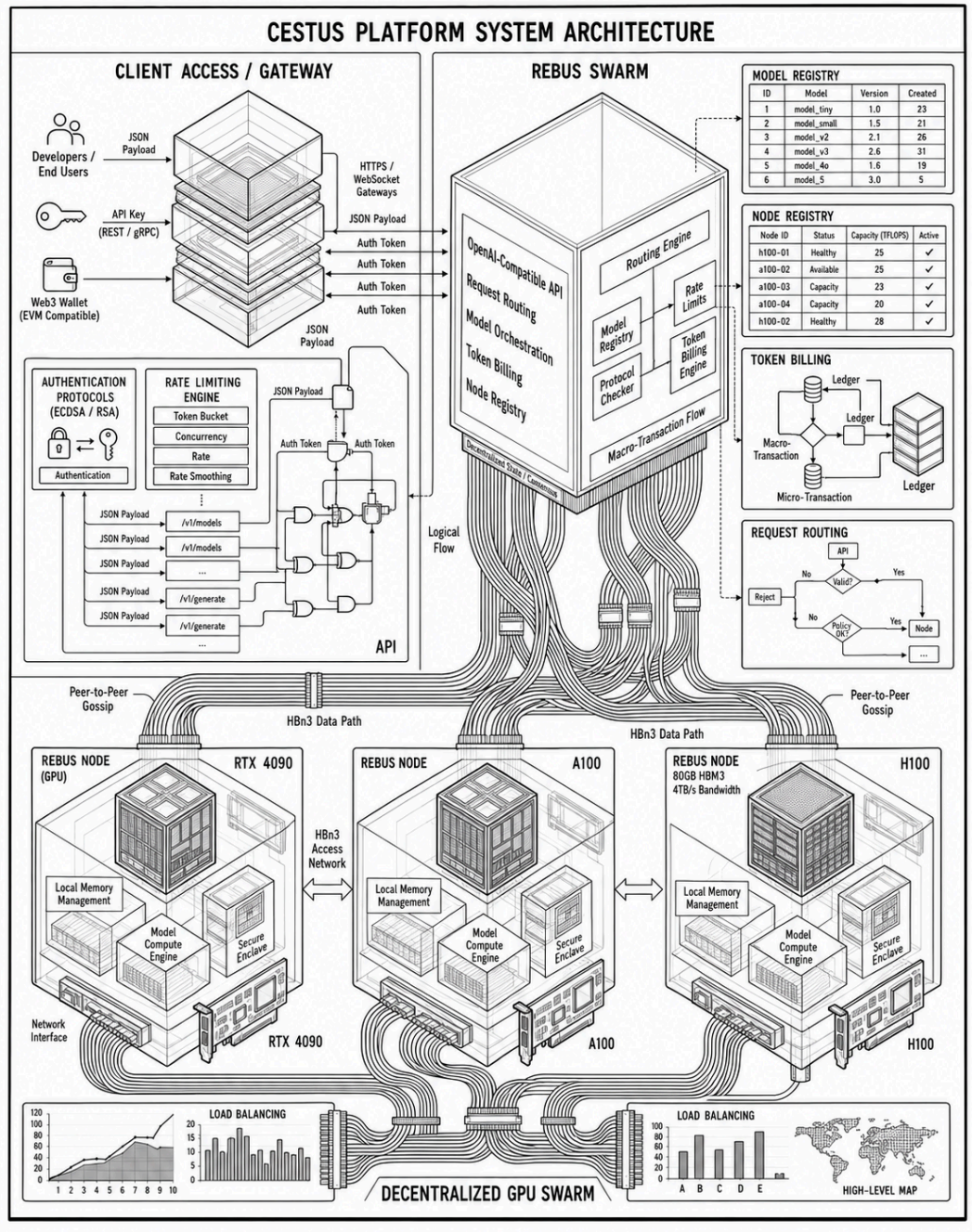
Because Cestus operates as a permissionless network with no centralized enforcement layer, content policy is not a platform-level concept. From the swarm's perspective, any registered open-weight model is equivalent: it occupies VRAM on a node, receives routed requests, and generates billed output. Whether a given model applies internal content filtering is a property of the weights themselves, determined by how the model was trained, not by any Cestus protocol rule. Operators who run nodes choose which models to serve. Developers who call the API choose which models to request. The platform enforces neither selection.

This is an architectural property, not a policy position, and it has direct commercial significance. A substantial portion of inference demand exists in domains where centralized content enforcement is incompatible with the application's requirements. That demand currently has no production-grade API to call.

---

3. System Architecture

Cestus operates across two primary components: the **Rebus Node**, software run by GPU operators, and the **Rebus Swarm**, the coordination and API layer that orchestrates the network. On top of the swarm sits the **Cestus Platform**, which exposes both a developer API and a suite of consumer-facing interfaces (a chat UI, an image generation UI, and a video generation UI) all powered by the same underlying infrastructure.



The separation of concerns between the two components is deliberate and consequential. Node software handles local hardware and model execution; the swarm handles routing, billing, and protocol logic. Because

neither layer has knowledge of or dependency on the internals of the other, each can evolve independently. Protocol upgrades do not require node operators to redeploy. New hardware joins the routing pool automatically. The swarm scales horizontally without coordination with individual nodes.

Both components are implemented in Elixir, chosen for its fault-tolerant concurrency model and native support for the high-connection-count workloads that distributed inference demands.

---

4. The Rebus Node

The Rebus node is software installed by any GPU operator wanting to contribute compute and earn rewards. It manages models, executes inference, and reports state to the swarm.

Rebus handles the complete model lifecycle on the node. It is responsible for downloading models from Hugging Face and maintaining their state in VRAM, with full support for multi-file model formats. Models are loaded into or unloaded from VRAM dynamically based on demand signals from the swarm: the routing algorithm directs traffic toward whichever models are currently hot, and Rebus responds by loading requested models and unloading idle ones to free capacity. This behaviour is transparent to the developer and requires no manual intervention from the operator.

Rebus implements input token caching to reduce the effective volume of input tokens processed across repeated or similar requests. The billing model reflects this: cached input tokens carry a separate, lower price than uncached tokens, meaning developers pay for actual compute rather than for redundant re-processing of context they have already paid for.

The node collects detailed usage metrics per model and transmits them to the swarm for billing settlement. These metrics include input token counts (broken down by cached and uncached), output token counts, and per-model request volume. Rebus also collects GPU power draw in Watts alongside standard hardware metrics, giving operators visibility into energy consumption and providing the swarm with data to support power-aware routing in future protocol versions.

4.1 Node States

At any moment, a node is in one of five states:

State	Description
`connecting`	Establishing connection to swarm
`idle`	Connected, no active inference tasks
`busy`	One or more inference tasks in flight
`downloading`	Fetching a model as instructed by swarm

State	Description
error	Fault state; excluded from routing until recovery

4.2 GPU Capability Profiling

On registration, each node reports its hardware profile. The fundamental constraint determining which models a node can serve is VRAM. The swarm enforces the following fit condition before assigning any model:

$$\text{VRAMnode} \geq M\text{size} \times Q_{\text{factor}} + \text{KV}_{\text{cache}}$$

Where M_{size} is model parameter count in bytes, Q_{factor} is the quantization compression ratio (Q4_K_M $\approx$ 0.5, FP16 = 2.0), and KV_{cache} is the key-value cache requirement at the target context length:

$$\text{KVcache} = 2 \times L \times nkv \times d_k \times P_{\text{bytes}} \times S_{\text{ctx}}$$

Where L is transformer layer count, n_{kv} is the number of key-value heads, d_k is the per-head dimension, P_{bytes} is precision in bytes, and S_{ctx} is context length in tokens. The factor of 2 accounts for separate key and value caches. For Gemma 2 9B (42 layers, 8 KV heads, head dimension 256, FP16, 4096-token context):

$$\text{KV}_{\text{cache}} = 2 \times 42 \times 8 \times 256 \times 2 \times 4096 \approx 1.3 \text{ GB}$$

This calculation runs before assignment, preventing out-of-memory failures during inference.

4.3 Model Thermal States

Models on a node exist in one of three thermal states, which directly determine routing priority:

- **Hot.** Weights loaded in VRAM. Zero cold-start latency. The swarm routes to hot nodes first.
- **Cold.** Weights on disk, loadable into VRAM before serving (typically 2-10 minutes depending on model size and disk speed).
- **Unloaded.** Not present on the node. Requires download before any serving.

The swarm maintains a live thermal map across all nodes and routes to hot nodes wherever possible, minimizing time-to-first-token.

4.4 Node Operator Dashboard

Each node exposes a local web UI providing real-time visibility into GPU utilization and temperature, VRAM allocation, active and completed tasks, model inventory with thermal states, and cumulative earnings attributed to the operator's wallet.

---

5. The Rebus Swarm

The Rebus Swarm is the coordination and API layer, the component that developers and end users interact with directly. It performs six functions simultaneously: exposes the OpenAI-compatible API; authenticates requests; maintains a live registry of all connected nodes, their hardware profiles, and model availability; selects the optimal node for each incoming request; instructs nodes to download and warm models based on demand patterns; and meters token consumption per request for billing.

5.1 API Compatibility

The swarm implements the OpenAI Chat Completions schema exactly. A valid request to Cestus is structurally identical to a request to OpenAI the only difference is the base URL and the model identifier. Any existing OpenAI SDK in any language connects to Cestus with one configuration change.

```
https://api.cestus.network/v1
```

API keys follow the format `rbs_live_*` and are issued through the Cestus developer dashboard.

5.2 Node Registry

For each connected node, the swarm maintains a live record tracking hardware profile, model inventory with thermal states, active task counts, rolling P95 latency, and current status. This registry is the source of truth for all routing decisions and is updated continuously as nodes report state changes.

```
NodeRecord {
  node_id:           string
  hardware: {
    gpu_model:       string
    vram_total_gb:   integer
    vram_used_gb:    integer
    power_watts:     float
  }
  models: [
    { model_id: string, state: hot | cold | unloaded, size_gb: float }
  ]
  tasks: {
    in_flight:       integer
    completed:       integer
    errors:          integer
  }
  latency_p95_ms:    float
  status:            idle | busy | downloading | error
}
```

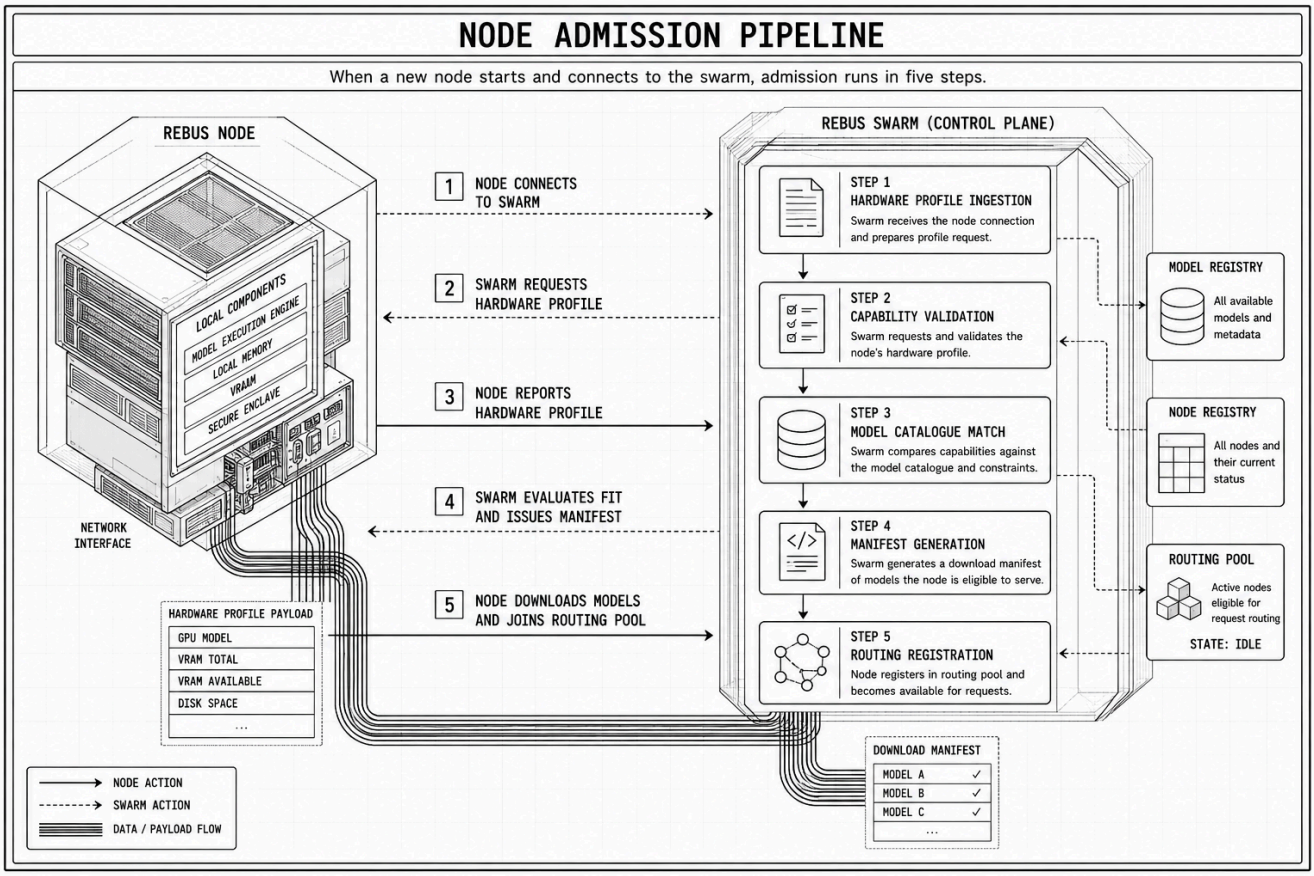
6. Swarm Formation and Distributed Inference

6.1 Network Structure

The Cestus Swarm is not a fixed cluster of owned hardware. It is a dynamically forming network of independently operated GPU nodes. The swarm has no central hardware dependency: any qualifying machine can join, contribute compute, and earn rewards, and any machine can leave without disrupting the network. Total capacity, model coverage, and geographic distribution grow organically as operators join. There is no capacity planning, no procurement cycle, and no scaling ceiling imposed by owned infrastructure.

6.2 Node Admission

When a new node starts and connects to the swarm, admission runs in five steps:



1. Node connects to swarm
2. Swarm requests hardware profile
3. Node reports: GPU model, VRAM total, VRAM available, disk space
4. Swarm evaluates fit against model catalogue and issues download manifest
5. Node downloads assigned models -> enters routing pool (state: idle)

6.3 Emergent Capacity

Network capacity is an emergent property of swarm size. Let $N(t)$ be the active node count at time t. Total inference capacity is:

$$\Theta_{\text{swarm}}(t) = \sum_{n=1}^{N(t)} \Theta_n \cdot U_n(t)$$

Where Θ_n is node n's peak throughput in tokens per second and $U_n(t)$ is its utilization. As $N(t)$ grows, Θ_{swarm} grows proportionally without any centralized provisioning step. The network's capacity is bounded below by:

$$\Theta_{\text{swarm}} \geq N \times \Theta_{\text{min}}$$

Where Θ_{min} is the throughput of the minimum-spec qualifying node (24 GB VRAM, RTX 3090 class).

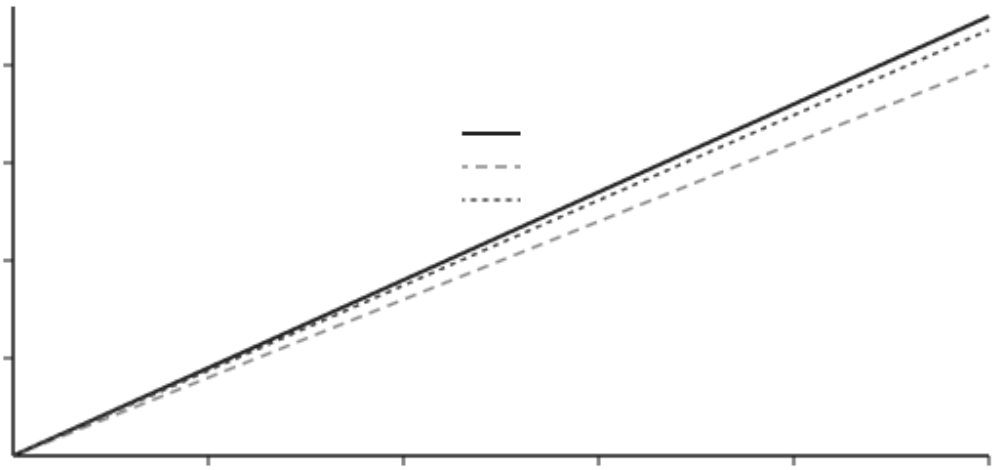


FIGURE 1 / Network throughput Θ_{swarm} scales linearly with active node count N. The lower bound is set by the minimum qualifying hardware (RTX 3090, 24 GB VRAM). The upper bound assumes H100-class nodes. Mean throughput reflects a heterogeneous mix of consumer and data centre GPUs. No centralized provisioning step is required at any point on the curve.

Each inference request is served by a single node. The swarm distributes at the request level different concurrent requests route to different nodes. This is the appropriate model for a heterogeneous, permissionless network where nodes join and leave unpredictably: it avoids the coordination complexity of splitting individual requests across nodes while delivering full parallelism at scale.

The effective parallelism for model m is:

$$P_m = |\mathcal{N}_m^{\text{hot}}| + \alpha \cdot |\mathcal{N}_m^{\text{cold}}|$$

Where $\mathcal{N}_m^{\text{hot}}$ and $\mathcal{N}_m^{\text{cold}}$ are the sets of nodes with model m hot or cold respectively, and $\alpha \in (0, 1)$ discounts cold-start latency. Maximum concurrent requests without queueing:

$$\text{MaxConcurrent}(m) = \sum_{n \in \mathcal{N}_m^{\text{hot}}} \lambda \max, n$$

6.4 Dynamic Model Replication

For popular models, the swarm maintains multiple hot nodes. The target replication factor R_m is set dynamically by demand:

$$R_m = \left\lceil \frac{\text{demand}(m, \Delta t)}{\bar{\Theta} \times \lambda_{\max}} \right\rceil$$

Where $\text{demand}(m, \Delta t)$ is request volume over rolling window Δt. When R_m exceeds the current hot node count, the swarm triggers pre-warming on eligible cold nodes. Popular models attract more hot nodes automatically; rarely-requested models stay cold to preserve VRAM for higher-demand workloads. The system is self-regulating.

6.5 Fault Tolerance

The swarm treats node failure as a normal operating condition. When a node fails its health check it is immediately removed from the routing pool, in-flight requests are requeued to other eligible nodes, and if the failure drops the hot node count for any model below R_m pre-warming is triggered on the next eligible cold node. The system targets zero dropped requests under single-node failure with a healthy swarm.

---

7. API Reference

7.1 List Available Models

Returns all models currently available on the network across text, image, and video modalities. Each entry includes a `kind` field (`"text"`, `"image"`, or `"video"`) and `context_length` (null for image and video models). Cestus imposes no fixed model catalogue: any open-weight model that meets the hardware fit conditions of Section 4.2 can be registered and served. The available set expands continuously as operators join and new open-source releases are added. Always query this endpoint at runtime rather than hardcoding model identifiers.

```
curl -s https://api.cestus.network/v1/models \
  -H 'Authorization: Bearer <YOUR_API_KEY>'
```

```
{
  "object": "list",
  "data": [
    {
      "id": "flux2-klein-9b",
      "object": "model",
      "kind": "image",
      "owned_by": "cestus",
      "context_length": null
    },
    {
      "id": "wan2.1-t2v-1.3b",
      "object": "model",
      "kind": "video",
      "owned_by": "cestus",
      "context_length": null
    },
    {
      "id": "qwen3.6-27b-uncensored",
      "object": "model",
      "kind": "text",
      "owned_by": "cestus",
      "context_length": 8192
    },
    {
      "id": "gemma-4-26B",
      "object": "model",
      "kind": "text",
      "owned_by": "cestus",
      "context_length": 8192
    },
    {
      "id": "glm-4.7-flash",
      "object": "model",
      "kind": "text",
      "owned_by": "cestus",
      "context_length": 8192
    },
    {
      "id": "qwen3.6-35b",
      "object": "model",
      "kind": "text",
      "owned_by": "cestus",
      "context_length": 8192
    }
  ]
}
```


Note: All model identifiers shown throughout this document are illustrative examples of open-weight models that can run on Cestus. The actual available set is dynamic and returned at

runtime by the `/v1/models` endpoint. The whole point of Cestus is the ability to use any open model: the catalogue is not fixed and grows continuously as operators register new models.

7.2 Text Inference

Follows the OpenAI `/v1/chat/completions` schema exactly. Model identifiers in the examples below are illustrative; the actual available set is returned by the `/v1/models` endpoint.

```
curl -X POST https://api.cestus.network/v1/chat/completions \
  -H 'Authorization: Bearer <YOUR_API_KEY>' \
  -H 'Content-Type: application/json' \
  -d '{
    "model": "gemma-4-26B",
    "messages": [{"role": "user", "content": "Explain how transformer attention
works."}]
  }'
```

```
{
  "id": "chatcmpl-qZdbmIm1LFmRqW1R",
  "object": "chat.completion",
  "created": 1780264698,
  "model": "gemma-4-26B",
  "node": 229,
  "usage": {
    "cancelled": false,
    "input_tokens": 17,
    "output_tokens": 99,
    "total_tokens": 116
  },
  "choices": [{
    "index": 0,
    "message": {"role": "assistant", "content": "Hello! How can I help you today?"},
    "finish_reason": "stop"
  }]
}
```

The `usage` object provides exact token counts for billing. Input and output tokens are metered separately as they carry different compute costs (see Section 11). The `cancelled` field indicates whether the request was terminated before completion, which affects billing. The `node` field records which node served the request.

SDK compatibility (Python):

```
from openai import OpenAI

client = OpenAI(
```

```
    base_url="https://api.cestus.network/v1",
    api_key="<YOUR_API_KEY>"
)

response = client.chat.completions.create(
    model="gemma-4-26B",
    messages=[{"role": "user", "content": "Hello, Cestus."}]
)
```

No changes to the OpenAI SDK are required beyond `base_url` and `api_key`.

7.3 Image Generation

Image generation requests are **asynchronous**. Because generation may take anywhere from seconds to several minutes depending on model, resolution, and step count, the API uses a job queue model rather than a synchronous HTTP response. All requests are submitted to the same endpoint as before; the difference is in the response and the follow-up polling step.

Submitting a generation request

```
curl -X POST https://api.cestus.network/v1/images/generations \
  -H 'Authorization: Bearer <YOUR_API_KEY>' \
  -H 'Content-Type: application/json' \
  -d '{
    "model": "flux2-klein-9b",
    "prompt": "a red panda riding a bicycle on the moon, cinematic lighting",
    "n": 8,
    "size": "1024x1024",
    "steps": 20,
    "seed": 5000
  }'
```

Request parameters:

Parameter	Type	Description
`model`	string	Image model identifier (illustrative; see `/v1/models` for available models)
`prompt`	string	Text description of the image to generate
`n`	integer	Number of images to generate (default: 1)
`size`	string	Output dimensions, e.g. `512x512`, `768x768`, `1024x1024`
`steps`	integer	Diffusion sampling steps. Higher values yield better quality at the cost of latency (default: 20)

Parameter	Type	Description
seed	integer	Reproducibility seed. Identical seed and prompt produce identical output

Immediate response (HTTP 202 Accepted):

```
{
  "id": "img-4a724_6lR64OmdUO",
  "status": "queued",
  "created": 1780437050
}
```

The system has accepted the request and added it to the generation queue. The id value is used to track the job.

Tracking job status

```
curl -X GET https://api.cestus.network/v1/jobs/img-4a724_6lR64OmdUO \
  -H 'Authorization: Bearer <YOUR_API_KEY>'
```

In-progress response (HTTP 200):

```
{
  "id": "img-4a724_6lR64OmdUO",
  "status": "processing"
}
```

Completed response (HTTP 200):

```
{
  "data": [
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/0.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/1.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/2.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/3.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/4.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/5.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/6.png" },
    { "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/7.png" }
  ],
  "id": "img-4a724_6lR64OmdUO",
  "status": "completed",
  "created": 1780437185
}
```

The `data` array contains one entry per generated image. All artifacts are stored on `storage.cestus.network` and accessible via their URLs for download or direct use.

Python polling example:

```
import time, requests

headers = {"Authorization": "Bearer <YOUR_API_KEY>"}

# Submit the job

resp = requests.post(
    "https://api.cestus.network/v1/images/generations",
    headers=headers,
    json={
        "model": "flux2-klein-9b",
        "prompt": "a red panda riding a bicycle on the moon, cinematic lighting",
        "n": 8,
        "size": "1024x1024",
        "steps": 20,
        "seed": 5000
    }
)
job_id = resp.json()["id"]

# Poll until complete

while True:
    status = requests.get(
        f"https://api.cestus.network/v1/jobs/{job_id}",
        headers=headers
    ).json()
    if status["status"] == "completed":
        for i, item in enumerate(status["data"]):
            print(f"Image {i}: {item['url']}")
        break
    time.sleep(2)
```

Generation time scales linearly with steps K:

$$t_{\text{generate}} \approx K \times t_{\text{step}}$$

For `flux2-klein-9b` on an RTX 4090 at 1024x1024, $t_{\text{step}} \approx 40\text{ms}$, giving approximately 800ms at the default 20 steps. For larger resolutions and higher step counts, generation may take several minutes; the async job model is designed for exactly this range.

7.4 Video Generation

Video generation uses the same asynchronous job pattern as image generation. Requests are submitted to `/v1/videos/generations` and tracked via the shared `/v1/jobs/:job_id` endpoint. Generation time can range from two minutes to considerably longer depending on model, resolution, and frame count.

Submitting a video generation request

```
curl -X POST https://api.cestus.network/v1/videos/generations \
  -H 'Authorization: Bearer <YOUR_API_KEY>' \
  -H 'Content-Type: application/json' \
  -d '{
    "model": "wan2.1-t2v-1.3b",
    "prompt": "a lovely cat walking",
    "size": "832x480",
    "frames": 81,
    "fps": 16
  }'
```

Request parameters:

Parameter	Type	Description
`model`	string	Video model identifier (illustrative; see `/v1/models` for available models)
`prompt`	string	Text description of the video to generate
`size`	string	Output resolution, e.g. `832x480`, `1280x720`
`frames`	integer	Total number of frames to generate
`fps`	integer	Frames per second for playback

Immediate response (HTTP 202 Accepted):

```
{
  "id": "img-4a724_6lR64OmdUO",
  "status": "queued",
  "created": 1780437050
}
```

Tracking job status

Use the same `/v1/jobs/:job_id` endpoint as for image jobs.

In-progress response (HTTP 200):

```
{
  "id": "img-4a724_6lR64OmdUO",
  "status": "processing"
}
```

Completed response (HTTP 200):

```
{
  "data": [
    {
      "url": "https://storage.cestus.network/jobs/img-4a724_6lR64OmdUO/0.webm"
    }
  ],
  "id": "img-4a724_6lR64OmdUO",
  "status": "completed",
  "created": 1780433704
}
```

The completed response returns a `data` array with one entry per generated video clip. Video artifacts are stored as `.webm` files on `storage.cestus.network` and available by URL immediately upon job completion.

The example above generates an 81-frame clip at 16 fps, giving approximately 5 seconds of video. Duration is controlled by `frames / fps`.

---

8. Consumer Interfaces

The Cestus platform is not exclusively a developer API. The same inference infrastructure that powers the API also backs a set of consumer-facing web interfaces, accessible to any user without any API key or SDK knowledge. All three interfaces are built directly on top of the Rebus Swarm and share the same routing, billing, and model availability as the developer API: there is no separate infrastructure layer for consumer products.

8.1 Chat UI

The Cestus Chat UI provides a browser-based interface for conversational interaction with any text model available on the network. Users can select from the full catalogue of registered open-weight models, switch models mid-session, and access models that centralized chat products do not offer. Billing is metered per token and drawn from the user's CES balance.

8.2 Image Generation UI

The image generation interface allows users to generate images through a visual form without writing any code. Users specify a model, prompt, resolution, step count, and seed, submit the job, and poll for results: the same

async job flow described in Section 7.3, surfaced as a point-and-click interface. Completed images are delivered as downloadable files hosted on storage.cestus.network.

8.3 Video Generation UI

The video generation interface extends the same pattern to video. Users configure a model, prompt, resolution, frame count, and frame rate, submit the job, and receive a downloadable `.webm` file on completion. Given that generation times for video can range from two minutes to considerably longer, the UI surfaces live job status updates while the job is in progress.

8.4 Unified Infrastructure

Because all three consumer interfaces and the developer API are powered by the same swarm, every improvement to the network - more nodes, more models, faster routing, lower latency - benefits all access patterns simultaneously. There is no distinction in model availability or performance between a request originating from the Chat UI and a request from an application built against the API.

---

9. Node Operator Deployment

9.1 Hardware Requirements

GPU Series	Examples	Min VRAM
NVIDIA RTX 30xx	RTX 3090, RTX 3090 Ti	24 GB
NVIDIA RTX 40xx	RTX 4090, RTX 4080 Super	24 GB
NVIDIA RTX 50xx	RTX 5090, RTX 5080	24 GB
NVIDIA L-Series	L40, L40S, L40G	48 GB
NVIDIA H-Series	H100, H200	80 GB

The 24 GB minimum is set by the smallest production-quality models in the catalogue. Higher VRAM nodes are assigned larger, higher-demand model slots by the routing algorithm and earn proportionally more. The network also hosts text-to-speech and speech-to-text models, which operate within lower VRAM budgets and can be served by nodes that do not meet the threshold for large language models.

Software prerequisites: Docker (latest stable), NVIDIA drivers (525+), NVIDIA Container Toolkit, Linux (Ubuntu 22.04 LTS recommended).

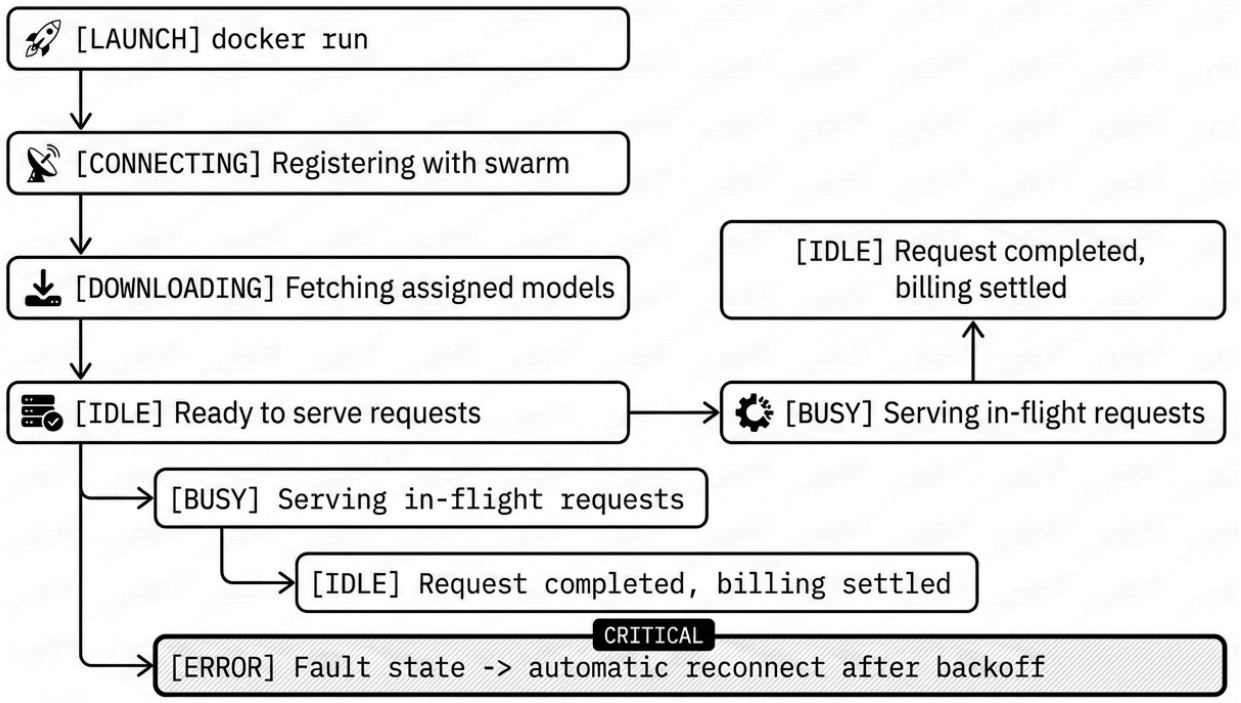
9.2 Launch

Each node is registered to a Web3 wallet address that receives operator earnings. Launch is a single command:

```
docker run --gpus all \
  -e WALLET="<YOUR_WALLET_ADDRESS>" \
  cestusnetwork/rebus:latest
```

On first launch, the container detects GPU hardware, registers its hardware profile with the swarm, receives a model download manifest, downloads assigned models, and enters the idle state once at least one model is ready to serve.

9.3 Node Lifecycle



9.4 Model Assignment by VRAM

The swarm assigns models to nodes based on the VRAM fit conditions in Section 4.2. The table below illustrates representative assignments for a 24 GB node using models available at launch. This is not a fixed catalogue: any open-weight model that satisfies the hardware fit conditions can be registered and served, and the available set expands continuously.

Model	Kind	Params	Quant	Est. Size	Assignable (24 GB)?
flux2-klein-9b	image		Q8	~8 GB	Yes

Model	Kind	Params	Quant	Est. Size	Assignable (24 GB)?
wan2.1-t2v-1.3b	video	1.3B	Q8	~2 GB	Yes
glm-4.7-flash	text	~9B	Q4_K_M	~6 GB	Yes
gemma-4-26B	text	26B	Q4_K_M	~16 GB	Yes
qwen3.6-27b-uncensored	text	27B	Q4_K_M	~17 GB	Yes
qwen3.6-35b	text	35B	Q4_K_M	~22 GB	Yes (VRAM-limited)
70B-class models	text	70B	Q4_K_M	~42 GB	No

A note on uncensored models. Because Cestus is a permissionless network with no centralized content enforcement layer, it can serve uncensored model variants that centralized providers do not offer. This is a structural property of the platform, not a policy exception. Healthcare, legal, security research, and creative applications that require uninhibited model outputs can access them through Cestus in the same way they access any other model: via the standard API, with the same routing and billing mechanics. The distinction between a censored and uncensored model is, from the network's perspective, purely a matter of which weights are loaded on a node. Content decisions belong to the operator and the developer, not to the platform.

---

10. Routing and Load Allocation

The routing algorithm is the most performance-critical component of the platform. It runs on every incoming request and must select the optimal node within milliseconds.

10.1 Routing Objectives

The algorithm optimizes for four objectives in priority order:

1. **Model availability.** The selected node must have the requested model in a hot or cold state.
2. **Minimum TTFT.** Prefer nodes with the model hot in VRAM.
3. **Load distribution.** Spread requests across available nodes to prevent single-node saturation.
4. **Latency.** Among equivalent candidates, prefer nodes with lower measured P95 latency.

10.2 Node Scoring Function

For each incoming request r for model m, the swarm computes a score for every eligible node n:

$$S(n, r) = w_1 \cdot T(n, m) + w_2 \cdot \left(1 - \frac{\lambda_n}{\lambda_{\max}}\right) + w_3 \cdot \left(1 - \frac{L_n}{L}\right)$$

Where:

- $T(n, m)$ is the thermal availability score: 1.0 if model m is **hot**, 0.5 if **cold**, 0 if **unloaded**
- $\lambda_n / \lambda_{\max}$ is node n's current load relative to the maximum observed across all nodes
- L_n / L is node n's P95 latency relative to the network mean
- $w_1 = 0.6$, $w_2 = 0.3$, $w_3 = 0.1$ are the default weighting coefficients

The coefficients are normalized such that $w1 + w2 + w3 = 1$. They are configurable at the protocol level.

The request routes to the highest-scoring node:

$$n^* = \arg\max_{n \in \mathcal{N}_m} S(n, r)$$

The weighting structure ensures thermal availability dominates a hot node is always preferred over a cold one while load and latency serve as tiebreakers. The coefficients are configurable.

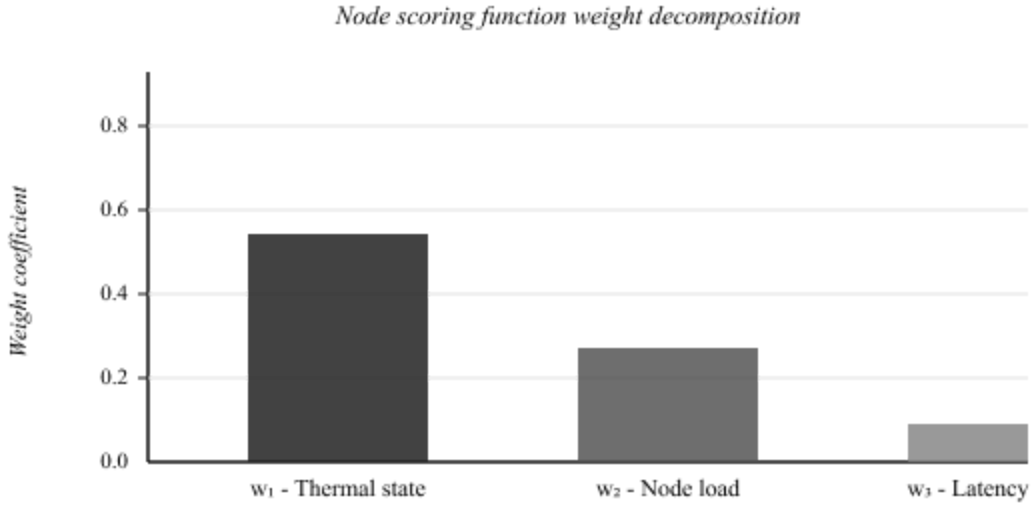


FIGURE 2 / Default weight decomposition of the node scoring function $S(n, r)$. Thermal state (hot/cold/unloaded) carries the dominant weight at 0.60, ensuring TTFT is minimized by preferring models already resident in VRAM. Current node load carries 0.30, distributing requests evenly across eligible nodes. Measured P95 latency carries 0.10 as a final tiebreaker. All weights are configurable at the protocol level.

When all eligible nodes for model m are at capacity, the swarm places the request in a priority queue and dispatches it to the next node that completes a task. Mean queue wait time under Poisson arrivals is:

$$\mathbb{E}[W] = \frac{\rho}{\mu(1-\rho)}$$

Where $\rho = \lambda / \mu$ is utilization, λ is arrival rate, and μ is the service rate. For $\rho < 0.8$, mean queue time is negligible relative to inference latency. The swarm monitors queue depth and triggers pre-warming on additional nodes when pressure builds.

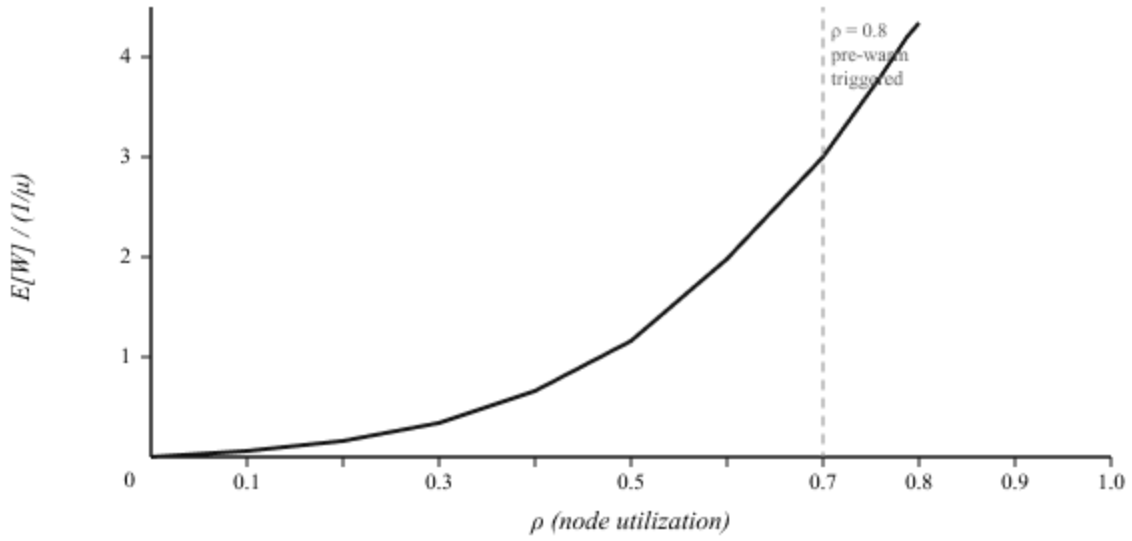


FIGURE 3 / Mean queue wait time $\mathbb{E}[W]$ as a function of node utilization ρ, normalised to mean service time. Below $\rho = 0.8$, queue contribution to total request latency is negligible. The swarm triggers model pre-warming on additional nodes when utilization crosses this threshold, maintaining the system in the low-wait operating region.

10.3 Model Assignment Priority

When assigning models to a newly joined node, the swarm ranks candidates by expected yield:

$$\text{Priority}(m, n) = \frac{\text{demand}(m) \times P_{\text{node}}(m)}{\text{size}(m)}$$

Where $\text{demand}(m)$ is recent request volume, $P_{\text{node}}(m)$ is the operator's per-token revenue share, and $\text{size}(m)$ is model size in GB. This maximizes operator earnings per gigabyte of VRAM committed.

---

11. Token Economics and Billing

11.1 Billing Model

Cestus uses per-token metered billing. All charges on the platform are denominated in **CES**, the native network token. Users deposit CES to fund their inference budget; node operators receive CES for compute contributed; node operators may use CES for their own inference or trade it freely.

Input and output tokens are priced separately, reflecting their different compute profiles: input tokens are processed in a single parallelized forward pass (compute-bound), while output tokens are generated sequentially

(memory-bandwidth-bound). Input tokens are further split into **uncached** and **cached** categories: uncached tokens are processed in full; cached tokens have been retained in Rebus's input cache from a prior request and are billed at a lower rate reflecting the reduced compute.

Total request cost:

$$C_{\text{request}} = T_{\text{in,uncached}} \times P_{\text{in}} + T_{\text{in,cached}} \times P_{\text{in,cached}} + T_{\text{out}} \times P_{\text{out}}$$

Where $T_{\text{in,uncached}}$ and $T_{\text{in,cached}}$ are uncached and cached input token counts, T_{out} is output token count, and P_{in}, $P_{\text{in,cached}}$, P_{out} are the corresponding per-token prices for the requested model, all denominated in CES. Pricing is expressed in CES with a USD mapping updated periodically; for example, 100k tokens for a Qwen-class model priced at $0.10 USD maps to the equivalent CES value at the prevailing rate.

11.2 Revenue Distribution

For each request, protocol revenue is split between the node operator, the protocol treasury, and a network development reserve. The operator receives the majority share, directly tied to the compute they contributed:

$$R_{\text{operator}} = T_{\text{out}} \times P_{\text{node}}(m)$$

Where $P_{\text{node}}(m)$ is the operator's per-output-token revenue share, denominated in CES. Operator revenue is attributed to output tokens, which represent the memory-bandwidth-bound decode phase that constitutes the dominant share of compute cost per request. Input token processing costs are recovered through the platform pricing margin rather than allocated directly to individual operators. Weekly earnings aggregate across all requests served in the 7-day period:

$\sum_r R_{\text{operator},r}$

over the billing period. Accumulated CES is transferred to the operator's registered wallet address at the end of each billing cycle.

11.3 Metering Integrity

Each response from a node includes a usage record signed by that node. The swarm verifies this record before crediting the node and deducting from the user's balance, preventing any node from inflating reported token counts:

$$\text{Valid if: verify}(k_{\text{node}}, \sigma_{\text{usage}}, H(\text{usage}_{\text{r}}\text{ecord})) = \text{true}$$

11.4 Alignment Properties

The billing structure creates direct incentive alignment across all participants. Users pay only for compute consumed, with no idle capacity billed to their account. Node operators earn in direct proportion to what they

serve, with no fixed cost to participate and no penalty for periods of low demand. Protocol revenue scales with network usage rather than with infrastructure overhead. There is no mechanism by which any participant profits from another participant's idleness, and no intermediary extracting margin between the compute that does the work and the developer who pays for it.

11.5 Platform Fee

A platform fee is applied to each settled request. The fee is expressed as a percentage of the total request cost and retained by the protocol treasury to fund ongoing development and network operations. The fee percentage is set by protocol governance and is visible to developers in the API response. Operator earnings are calculated after the platform fee is deducted from gross request revenue.

---

12. The CES Token

CES is the native utility token of the Cestus Network and the sole currency for all economic activity on the platform.

12.1 Token Utility

CES serves four functions within the network.

Payment for inference. Users deposit CES to fund their inference budget. Every API request is charged in CES at the prevailing per-token rate for the requested model. There is no fiat payment path for API access; CES is the exclusive medium of exchange for compute.

Operator earnings. Node operators receive CES for every inference request their hardware serves. Earnings accumulate over each 7-day billing cycle and are distributed to the operator's registered wallet at the end of the period. Operators receive the majority share of gross request revenue after the platform fee is deducted.

Operator consumption. Node operators can use CES earned through inference to pay for their own API access - running models on the network, fine-tuning, or any other platform service, without converting to another currency. This creates a closed loop where operators are both producers and consumers of the same token.

Trade and liquidity. CES is freely tradable. Operators who accumulate more CES than they consume in platform services may trade it on external markets. Users who need CES to fund inference may acquire it through the same channels.

12.2 Pricing and USD Mapping

Inference prices are denominated in CES and mapped to USD equivalents that are updated periodically based on the prevailing CES market price. A model priced at $0.10 USD per 100k tokens maps to the corresponding

CES quantity at the update time. This design preserves predictable USD-equivalent cost for developers while keeping the on-chain settlement layer entirely in CES.

12.3 Single-Token Architecture

The platform does not support multi-token settlement or fiat on-ramp at the protocol layer. Deposits are in CES; distributions are in CES; fees are deducted in CES. This simplifies the billing model, eliminates currency conversion risk within the protocol, and ensures that all value flows through the same token, directly tying token demand to network usage.

---

13. Inference Performance Model

13.1 Key Metrics

Three metrics characterize inference performance on the Cestus network.

Time to First Token (TTFT) is the elapsed time from request arrival to the first generated token reaching the client. It is the dominant user-perceived latency metric and decomposes as:

$$\mathrm{TTFT} = t_{\mathrm{routing}} + t_{\mathrm{queue}} + t_{\mathrm{prefill}} + t_{\mathrm{network}}$$

Routing overhead t_{routing} completes in under 5ms for networks of thousands of nodes due to the constant-time scoring computation. Under normal load, $t_{\mathrm{queue}} \approx 0$.

Time Per Output Token (TPOT) is the mean time to generate each successive token after the first. This phase is memory-bandwidth-bound, not compute-bound, for reasons described in Section 13.3.

$$\mathrm{TPOT} = \frac{t_{\mathrm{decode_{t}otal}}}{T_{\mathrm{out}}}$$

Network Throughput is total tokens generated per second across the active swarm:

$$\Theta_{\mathrm{network}} = \sum_{n \in \mathcal{N}\mathrm{active}} \frac{T\mathrm{out},n}{\mathrm{TPOT}_n}$$

13.2 Prefill Phase (Compute-Bound)

The prefill phase processes the entire input prompt in a single parallelized forward pass. Compute cost scales linearly with input length:

$$\mathrm{FLOPsprefill} \approx 2 \times T\mathrm{in} \times N_{\mathrm{params}}$$

For an 8B parameter model with 512 input tokens:

$$\approx 8.19 \times 10^{12} \text{ FLOPs}$$

Prefill time is bounded by GPU compute throughput (TFLOPS).

13.3 Decode Phase (Memory-Bound)

The decode phase generates output tokens one at a time. Each step requires loading the full model weight matrix from VRAM, making decode memory-bandwidth-bound rather than compute-bound. The theoretical floor is:

$$\text{TPOT} \approx \frac{2 \times N_{\text{params}} \times P_{\text{bytes}}}{\text{BW}_{\text{mem}}}$$

For an RTX 4090 (1,008 GB/s memory bandwidth) serving an 8B model in FP16:

$$\text{TPOT} \approx \frac{2 \times 8 \times 10^9 \times 2}{1.008 \times 10^{12}} \approx 31.7 \text{ ms/token}$$

This yields a theoretical maximum of approximately **31.5 tokens per second** (1000 / 31.7 ms) per RTX 4090 for an 8B model. Actual throughput varies with quantization, batch size, and KV cache efficiency. The figures above assume FP16 precision. In practice, nodes on the Cestus network typically serve models at Q4_K_M quantization, which reduces the bytes-per-parameter from 2 to approximately 0.5 and yields roughly 4× higher throughput than the FP16 theoretical floor shown here. The distinction between prefill and decode also drives the split pricing model in Section 11.1: the two phases have fundamentally different cost structures, and pricing them identically would misrepresent their compute profiles.

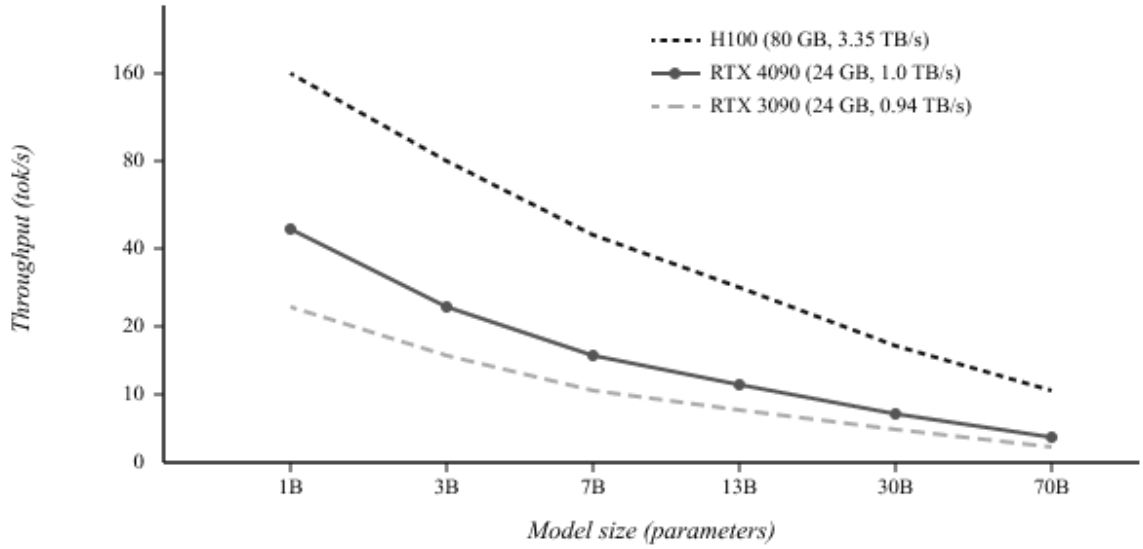


FIGURE 4 / Theoretical peak throughput (tokens per second) per node as a function of model size, across three representative GPU tiers available on the Cestus network. All figures assume FP16 precision. Decode phase throughput is memory-bandwidth-bound: larger models

saturate VRAM bandwidth faster, reducing tokens per second. The H100's 3.35 TB/s bandwidth advantage is most pronounced for large models.

Throughput scales with batch size up to memory saturation:

$$\Theta_{\text{node}} = B \times \frac{1}{\text{TPOT}(B)}$$

Where $\text{TPOT}(B)$ increases with batch size due to KV cache pressure. The optimal batch size maximizing throughput subject to a latency SLA τ is:

$$B^* = \max\left\{B : \text{TPOT}(B) \leq \frac{\tau}{T_{\text{out,max}}}\right\}$$

Continuous batching where new requests enter an in-progress batch as earlier requests complete allows nodes to approach theoretical peak throughput under sustained load.

13.4 Attention Complexity

The transformer self-attention mechanism (Vaswani et al., 2017) computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The quadratic complexity $O(T^2 \cdot d_k)$ in sequence length T makes long-context inference disproportionately expensive. This is reflected in Cestus's pricing structure for extended context windows, where longer contexts carry higher per-request costs that correspond to their actual compute profile.

---

14. Training Layer (Phase 2)

Consider what it means to own a model.

Not API access to someone else's model. Not a fine-tune you uploaded to a platform that can revoke your access, change pricing, or shut down tomorrow. Actual ownership of the weights, running on infrastructure you do not depend on any single company to maintain, with the ability to monetize that model by selling access to it through a permissionless marketplace and earning on every inference call it serves.

Nobody can offer this today. No centralized provider does, because ownership of the weights is incompatible with their business model of charging for access to models they control. No decentralized alternative does either, because none have built the training infrastructure, the model registry, and the inference marketplace as integrated components of the same platform.

Phase 2 introduces that capability to Cestus: model training as a service, at two levels. Fine-tuning adapts an existing open model to proprietary data without the cost or complexity of training from scratch. Full model training uses the Cestus swarm as the distributed compute substrate for training a new model end to end. In both cases, the resulting model belongs entirely to the party that commissioned it, stored in the Cestus registry under their control, and they choose what happens next: keep it private, sell access through the marketplace, export the weights, or release it publicly.

A model trained on Cestus and listed on the marketplace is not just a product. It is a revenue-generating asset with zero ongoing infrastructure cost to its creator. Every inference call it serves earns the creator a share of the billing revenue, automatically, indefinitely. This is a new economic primitive in AI. It does not exist anywhere else in the market today.

The training infrastructure is designed to run on the same node hardware used for inference. Fine-tuning via LoRA runs on the minimum 24 GB consumer GPU. Full distributed training scales across as many nodes as the job requires, using gradient compression to manage synchronization costs over a wide-area network.

14.1 Fine-Tuning via LoRA

Cestus implements **Low-Rank Adaptation (LoRA)** as its primary fine-tuning mechanism. LoRA decomposes weight updates into low-rank matrices, dramatically reducing the number of trainable parameters while preserving expressive capacity.

For a weight matrix $W \in \mathbb{R}^{d \times k}$, LoRA represents the update as:

$$\Delta W = BA, \quad B \in \mathbb{R}^{d \times r},\ A \in \mathbb{R}^{r \times k},\ r \ll \min(d, k)$$

The modified forward pass becomes:

$$h = W_0 x + BAx$$

Only A and B are updated during training; W_0 (the pre-trained weights) is frozen. For a typical layer with $d = k = 4096$ and $r = 16$:

$$\text{Trainable parameters} = r(d + k) = 16 \times 8192 = 131,072$$

Compared to $d \times k = 16,777,216$ for full fine-tuning a 128x reduction. This makes fine-tuning feasible on the same consumer GPU hardware used for inference, with no additional hardware tier required.

14.2 Full Model Training

Full training uses data-parallel distributed training across multiple nodes. Each node processes a different mini-batch and computes local gradients, aggregated via all-reduce:

$$g_{\text{global}} = \frac{1}{N_{\text{workers}}} \sum_{i=1}^{N_{\text{workers}}} g_i, \qquad \theta_{t+1} = \theta_t - \eta \cdot g_{\text{global}}$$

Gradient synchronization across a wide-area network is the primary performance constraint relative to co-located data center training. Cestus addresses this through gradient compression via top-k sparsification:

$$\tilde{g}_i = \text{TopK}(g_i, k), \quad k = \rho \cdot |g_i|, \quad \rho \in (0, 1)$$

At $\rho = 0.01$ transmitting the top 1% of gradients by magnitude convergence quality is preserved while synchronization bandwidth drops by 99%.

14.3 Training Cost and Scaling

Training jobs are billed by GPU-hours:

$$C_{\text{train}} = t_{\text{train}} \times P_{\text{gpu/hr}} \times N_{\text{gpus}}$$

For full training, total compute follows the Chinchilla scaling relationship (Hoffmann et al., 2022):

$$\text{FLOPstotal} \approx 6 \times N\text{params} \times D$$

Where D is dataset size in tokens. The factor of 6 accounts for forward pass, backward pass, and gradient application, giving the minimum compute budget required to train a model of a given size to convergence on a given dataset.

14.4 Model Ownership

Fine-tuned models are stored as LoRA adapter weights in the Cestus model registry. Base model weights remain on the network; only the adapter delta ΔW is associated with the user's account. Fully trained models are stored as complete weight checkpoints owned entirely by the training job initiator.

In both cases, the creator may keep the model private, publish it to the Cestus model marketplace with custom per-token pricing to earn revenue on every inference call, export weights for local deployment, or release the model publicly. The model belongs to its creator, not to Cestus.

---

15. Security and Trust Model

15.1 Authentication

Cestus supports two authentication mechanisms.

Bearer token. Standard API key issued on account creation. Suitable for server-side integrations.

Web3 wallet authentication. Clients sign a challenge message with a Web3 wallet. The swarm verifies the signature cryptographically:

$$\sigma = \text{sign}(k_{\text{private}},\ H(\text{challenge}|\text{timestamp}))$$

$$\text{Valid if: } \text{verify}(k_{\text{public}},\ \sigma,\ H(\text{challenge}|\text{timestamp})) = \text{true}$$

Where H is SHA-256 and k is the wallet key pair. This enables fully non-custodial, wallet-native authentication with no password and no centralized identity, a first-class requirement for Web3-native applications.

15.2 Node Trust and Fault Isolation

The swarm assumes individual nodes will fail, and designs around that assumption rather than against it. A node that fails its health check is immediately removed from the routing pool:

$$\text{NodeState}(n) = \text{error if } \Delta t_{\text{last}_{\text{h}}\text{eartbeat}} > \delta_{\text{heartbeat}}$$

In-flight requests are requeued automatically. Combined with the metering integrity mechanism in Section 11.3, where every usage record is cryptographically signed by the serving node, the system provides both availability guarantees and tamper-proof billing without requiring trust in any individual operator. Trust is placed in the protocol, not in participants.

15.3 Output Integrity

The metering integrity mechanism prevents a node from inflating reported token counts. A separate concern is whether a malicious node could return corrupted or fabricated outputs rather than genuine model completions. The current architecture addresses this through two mechanisms.

First, the signed usage record commits the node to specific input and output token counts for each request. Any attempt to return a trivially short or empty response while claiming full output token billing would produce a verifiable mismatch between the claimed token count and the actual response length, detectable by the swarm at settlement time.

Second, operator nodes are registered to wallet addresses that accumulate reputation over time through their completion and error rate statistics, visible in the node registry. Nodes with high error rates are deprioritized by the scoring function in Section 10.2 via the load distribution term, progressively routing them fewer requests. This creates an economic incentive for operators to maintain genuine, well-functioning inference capacity.

Probabilistic output verification, where the swarm periodically reruns a sample of requests on a second node and compares outputs, is under consideration for a future protocol version as the network reaches sufficient scale to make redundant verification cost-effective.

15.4 Swarm Availability

The Rebus Swarm is the single coordination layer for the network, which raises the question of what happens if the swarm coordinator becomes unavailable. The swarm is designed as a stateless router with shared state in a distributed key-value store, allowing multiple swarm instances to run in parallel behind a load balancer. No single swarm process holds exclusive state. The failure of any individual swarm instance does not affect in-flight requests on other instances, and the node registry state is replicated across the cluster.

Individual node software, Rebus, maintains its WebSocket connection to the swarm and reconnects automatically after a configurable backoff interval if the connection is dropped. Nodes do not lose their model inventory or hardware state during swarm downtime. The network resumes full operation as soon as swarm connectivity is restored, without any manual intervention from node operators.

---

16. Network Scalability

16.1 Capacity

The swarm is a stateless router. Multiple swarm instances run in parallel, allowing the API layer to scale independently of the node network. Total inference capacity grows linearly with node count:

$$\Theta_{\text{total}} = \sum_{n=1}^{N} \Theta_n \approx N \times \bar{\Theta}$$

There is no architectural ceiling on this scaling. The bottleneck is GPU operator participation, not platform design.

16.2 Model Availability

As the network grows, the probability that any given model is available in a hot state on at least one node rapidly approaches certainty:

$$P(\text{model } m \text{ hot}) = 1 - \prod_{n \in \mathcal{N}_m} (1 - pn, m)$$

For a model served by 10 nodes each with 80% probability of being hot:

$$P = 1 - (1 - 0.8)^{10} \approx 99.99999\%$$

This availability property emerges from network size alone, without any centralized coordination or redundancy planning. It is a consequence of scale, not an engineered guarantee.

16.3 Latency

Total end-to-end latency decomposes as:

$$L_{\text{total}} = t_{\text{routing}} + t_{\text{queue}} + t_{\text{prefill}} + T_{\text{out}} \times \text{TPOT} + L_{\text{network}}$$

The dominant terms are t_{prefill} and $T_{\text{out}} \times \text{TPOT}$, both model-execution-bound. For geographically distributed nodes, network overhead is second-order. Routing overhead completes in under 5ms regardless of swarm size. The latency profile of a Cestus request is therefore determined almost entirely by model size and output length, the same variables that determine latency on any inference platform, centralized or otherwise.

---

17. Conclusion

The centralization of AI inference is not an accident. It is a business model. The providers who control the intelligence layer today have every incentive to keep controlling it. They will price-discriminate, enforce content policies, and retain model opacity for as long as the market permits. Developers, researchers, Web3 protocols, and anyone who needs something different will not get it from those providers. By design.

Cestus is the architecture that makes that control irrelevant.

A permissionless GPU swarm serving inference across open language, image, and video models. Any developer integrates via an OpenAI-compatible API with one configuration change. Any end user accesses the platform directly through the Chat UI, image generation interface, or video generation interface: no API key, no SDK, no setup. Any GPU operator joins with one Docker command. Wallet-based authentication, per-token metered billing, uncensored model access, consumer-facing products, and a node operator dashboard are all components of the same unified platform.

Phase 2 introduces a new economic primitive that does not exist in the current AI market: training a model on decentralized infrastructure, owning the resulting weights outright, and monetizing that model through a permissionless marketplace that distributes earnings on every inference call served, indefinitely.

The architecture guarantees a self-reinforcing flywheel with no ceiling. More nodes expand model coverage. Broader coverage attracts more developers. More developer demand drives higher earnings per node. Higher earnings recruit more operators. Each phase of the roadmap places additional demand on the same underlying

infrastructure. Every new capability makes the network more valuable to operate. The compounding is structural.

The intelligence layer is the most valuable real estate in the next decade of software. Right now, a handful of companies own it. Cestus is how that changes.

---

18. References

1. Vaswani, A. et al. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, NeurIPS 2017*.
2. Hu, E. et al. (2022). LoRA: Low-Rank Adaptation of Large Language Models. *International Conference on Learning Representations (ICLR 2022)*. arXiv:2106.09685.
3. Kwon, W. et al. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023)*.
4. Hoffmann, J. et al. (2022). Training Compute-Optimal Large Language Models. *Advances in Neural Information Processing Systems 36 (NeurIPS 2022). arXiv:2203.15556.*
5. Lin, Y. et al. (2018). Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *International Conference on Learning Representations (ICLR 2018)*. arXiv:1712.01887.
6. Olshansky, D., Rodriguez Colmeiro, R., and Li, B. (2024). Decentralized AI: Permissionless LLM Inference on POKT Network. *arXiv:2405.20450*.
7. Hamadanian, P. and Fouladi, S. (2025). Glinthawk: A Two-Tiered Architecture for Offline LLM Inference. *arXiv:2501.11779*.
8. Frantar, E. et al. (2022). GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *arXiv:2210.17323*. Foundational work underlying the GGUF quantization format used for model storage and serving on Rebus nodes.
9. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal, 2014*(239).
10. Touvron, H. et al. (2023). LLaMA: Open and Efficient Foundation Language Models. *arXiv:2302.13971*.
11. Jiang, A. et al. (2023). Mistral 7B. *arXiv:2310.06825*.
