Aratta

The land that traded with empires but was never conquered.

Why

You got rate-limited again. Or your API key got revoked. Or they changed their message format and your pipeline broke at 2am. Or you watched your entire system go dark because one provider had an outage.

You built on their platform. You followed their docs. You used their SDK. And now you depend on them completely — their pricing, their uptime, their rules, their format, their permission.

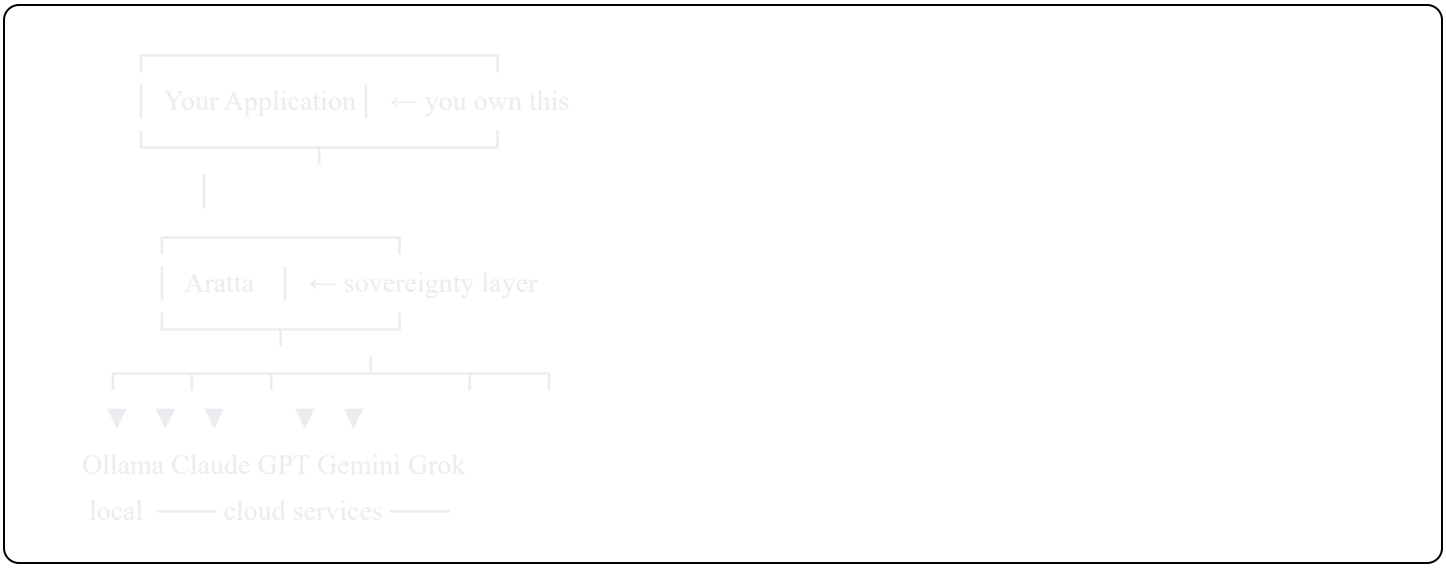
That's not infrastructure. That's a leash.

Aratta takes it off.

What Aratta Is

Aratta is a sovereignty layer. It sits between your application and every AI provider — local and cloud — and inverts the power relationship.

Your local models are the foundation. Cloud providers — Claude, GPT, Gemini, Grok — become callable services your system invokes when a task requires specific capabilities. They're interchangeable. One goes down, another picks up. One changes their API, the system self-heals. You don't depend on any of them. They work for you.



The Language

Aratta defines a unified type system for AI interaction. One set of types for messages, tool calls, responses, usage, and streaming — regardless of which provider is on the other end.

```python
from aratta.core.types import ChatRequest, Message, Role

request = ChatRequest(
    messages=[Message(role=Role.USER, content="Explain quantum computing")],
    model="local",     # your foundation
    # model="reason",  # or invoke Claude when you need it
    # model="gpt",     # or GPT — same code, same response shape
)
```

The response comes back in the same shape regardless of which provider handled it. Same fields, same types, same structure. Your application logic is decoupled from every provider's implementation details.

You never change your code when you switch providers. You never change your code when they change their API. You write it once.

What that replaces

Every provider does everything differently:

Concept	Anthropic	OpenAI	Google	xAI
Tool calls	`tool_use` block	`function_call`	`functionCall`	`function`
Tool defs	`input_schema`	`function.parameters`	`functionDeclarations`	`function.parameter`
Finish reason	`stop_reason`	`finish_reason`	`finishReason`	`finish_reason`
Token usage	`usage.input_tokens`	`usage.prompt_tokens`	`usageMetadata.promptTokenCount`	`usage.prompt_toke`
Streaming	`content_block_delta`	`choices[0].delta`	`candidates[0]`	OpenAI-compat
Thinking	`thinking` block	`reasoning` output	`thinkingConfig`	encrypted
Auth	`x-api-key`	`Bearer` token	`x-goog-api-key`	`Bearer` token

Aratta: `Message`, `ToolCall`, `Usage`, `FinishReason`. One language. Every provider.

Quick Start

```bash
pip install aratta
aratta init                # pick providers, set API keys, configure local
aratta serve               # starts on :8084
```

The `init` wizard walks you through setup — which providers to enable, API keys, and local model configuration. Ollama, vLLM, and llama.cpp are supported as local backends. Local is the default. Cloud is optional.

Use it

```python
import httpx

# Local model — your foundation
resp = httpx.post("http://localhost:8084/api/v1/chat", json={
    "messages": [{"role": "user", "content": "Hello"}],
    "model": "local",
})

# Need deep reasoning? Invoke a cloud provider
resp = httpx.post("http://localhost:8084/api/v1/chat", json={
    "messages": [{"role": "user", "content": "Analyze this contract"}],
    "model": "reason",
})

# Need something else? Same interface, different provider
resp = httpx.post("http://localhost:8084/api/v1/chat", json={
    "messages": [{"role": "user", "content": "Generate test cases"}],
    "model": "gpt",
})

# Response shape is always the same. Always.
```

Define tools once

Every provider has a different tool/function calling schema. You define

tools once. Aratta handles provider-specific translation:

```python
from aratta.tools import ToolDef, get_registry

registry = get_registry()
registry.register(ToolDef(
    name="get_weather",
    description="Get current weather for a location.",
    parameters={
        "type": "object",
        "properties": {"location": {"type": "string"}},
        "required": ["location"],
    },
))

# Works with Claude's tool_use, OpenAI's function calling,
# Google's functionDeclarations, xAI's function schema — automatically.
```

Model Aliases

Route by capability, not by provider model ID. Define your own aliases or use the defaults:

Alias	Default	Provider
`local`	llama3.1:8b	Ollama
`fast`	gemini-3-flash-preview	Google
`reason`	claude-opus-4-5-20251101	Anthropic
`code`	claude-sonnet-4-5-20250929	Anthropic
`cheap`	gemini-2.5-flash-lite	Google
`gpt`	gpt-4.1	OpenAI
`grok`	grok-4-1-fast	xAI

Aliases are configurable. Point `reason` at your local 70B if you want. Point `fast` at GPT. It's your routing. Your rules.

Full reference: docs/model-aliases.md

What Makes the Sovereignty Real

The sovereignty isn't a metaphor. It's enforced by infrastructure:

Circuit breakers — if a cloud provider fails, your system doesn't. The breaker opens, traffic routes elsewhere, and half-open probes test recovery automatically.

Health monitoring — continuous provider health classification with pluggable callbacks. Transient errors get retried. Persistent failures trigger rerouting.

Self-healing adapters — each provider adapter handles API changes, format differences, and auth mechanisms independently. Your code never sees it.

Local-first — Ollama is the default provider. Cloud is the fallback. Your foundation runs on your hardware, not someone else's.

API

Endpoint	Method	Description
`/health`	GET	Liveness probe
`/api/v1/chat`	POST	Chat — any provider, unified in and out
`/api/v1/chat/stream`	POST	Streaming chat (SSE)
`/api/v1/embed`	POST	Embeddings
`/api/v1/models`	GET	List available models and aliases
`/api/v1/health`	GET	Per-provider health and circuit breaker states

Agent Framework

Aratta includes a ReAct agent loop that works through any provider:

```python
from aratta.agents import Agent, AgentConfig, AgentContext

agent = Agent(config=AgentConfig(model="local"), context=ctx)
result = await agent.run("Research this topic and summarize")
```

Sandboxed execution, permission system, tool calling. Switch the model alias and the same agent uses a different provider. No code changes.

Details: docs/agents.md

Project Structure

```
src/aratta/
├── core/              The type system — the language
├── providers/
│   ├── local/         Ollama, vLLM, llama.cpp (the foundation)
│   ├── anthropic/     Claude (callable service)
│   ├── openai/        GPT (callable service)
│   ├── google/        Gemini (callable service)
│   └── xai/           Grok (callable service)
├── tools/             Tool registry + provider format translation
├── resilience/        Circuit breaker, health monitoring, metrics
├── agents/            ReAct agent loop, executor, sandbox
├── config.py          Provider config, model aliases
├── server.py          FastAPI application
└── cli.py             CLI (init, serve, health, models)
```

Development

```bash
git clone https://github.com/scri-labs/aratta.git
cd aratta
python -m venv .venv
.venv/Scripts/activate      # Windows
# source .venv/bin/activate # Linux/macOS
pip install -e ".[dev]"
pytest                      # 82 tests
ruff check src/ tests/      # clean
```

Docs

- Architecture — how it works
- Providers — supported providers + writing your own
- Model Aliases — routing by capability
- Agent Framework — ReAct agents across providers

License

Apache 2.0 — see LICENSE.