ChatGPT

Pilot Implementation Guide: Tether Hyperswarm Inference Cluster

This guide walks you through building a **local pilot** of the Tether inference network using the design described earlier. The goal is to stand up all core services on your machine so you can observe end-to-end request routing and inference using a small language model.

Architecture recap:

- **Gateway (Node.js/TypeScript)** – Exposes an HTTP API for clients, validates requests, enforces quotas and forwards jobs into the Hyperswarm control topic.
- **Orchestrator/Scheduler (Node.js/TypeScript)** – Subscribes to presence and model topics to maintain a live snapshot of available workers, performs affinity scoring and retries, reserves capacity in the metadata store, then dispatches jobs to workers via model topics.
- **Metadata store (rqlite)** – Lightweight distributed SQL database built on SQLite and Raft. Stores jobs, quotas and model metadata. In a pilot you can run a single node for simplicity.
- **Worker controller (Node.js/TypeScript)** – Announces presence and subscribed models, listens for `infer.start` messages, and invokes a Python model process.
- **Model runtime (Python)** – Uses Hugging Face's `transformers` to load a small model (e.g. `phi2` or `TinyGPT-2`) and perform inference.

Prerequisites

1. **Install Node.js 18+ and npm** (use nvm if you don't have Node installed). Example:

```
# install nvm (if not already installed)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
source ~/.bashrc
nvm install 18
nvm use 18
```

1. **Install Python 3.9+** and a virtual environment manager (e.g. `venv` or `conda`).

2. **Install rqlite** (SQLite with Raft replication). Download the latest binary from the rqlite releases page, unpack it, and add it to your `$PATH`:

```
wget -qO rqlite.tar.gz https://github.com/rqlite/rqlite/releases/download/
v7.13.0/rqlite-v7.13.0-linux-amd64.tar.gz
```

```
tar -xzf rqlite.tar.gz
sudo mv rqlite-v7.13.0-linux-amd64/rqlited /usr/local/bin/
```

1. **Create a project directory** and the sub-directories for each service. For example:

```
mkdir tether-pilot && cd tether-pilot
mkdir gateway orchestrator worker
```

Step 1: Set up the Metadata Store

Run a single rqlite node. In production you would form a cluster, but a standalone instance suffices for a pilot.

```
# choose a data directory
mkdir -p ~/rqlite-data
rqlited ~/rqlite-data &  # runs on http://localhost:4001

# create a jobs and models table (we will use simple schemas)
curl -s -XPOST localhost:4001/db/execute -d "CREATE TABLE IF NOT EXISTS jobs
(\n    id TEXT PRIMARY KEY,\n    tenant_id TEXT,\n    model_id TEXT,\n
version TEXT,\n    input TEXT,\n    status TEXT,\n    result TEXT,\n
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP\n);"
curl -s -XPOST localhost:4001/db/execute -d "CREATE TABLE IF NOT EXISTS models
(\n    model_id TEXT,\n    version TEXT,\n    description TEXT,\n    PRIMARY
KEY(model_id, version)\n);"
```

Choosing Small Language Models for the Pilot

The pilot uses small models so that inference can run on local hardware. Select one of the following models based on your hardware and requirements:

- **Microsoft Phi-2** (~2.7 B parameters): a high-quality small model that excels at reasoning, code generation and comprehension [1]. Optimized for low-latency inference, it works well for edge AI, on-device assistants and enterprise copilots [1].
- **Mistral 7B** (7 B): an open-source model with strong multilingual accuracy and fast reasoning [2]. Suitable for multilingual chatbots, content generation and cross-border customer service [2].
- **LLaMA 3 8B** (8 B): a refined small LLaMA model offering high accuracy with low compute demand and state-of-the-art performance in reasoning, summarization and dialogue [3].
- **IBM Granite 4.0 Micro** (≈3 B): a long-context instruct model fine-tuned for improved instruction following and tool-calling capabilities [4]. It supports many languages, including English, German, Spanish, French, Japanese, Portuguese, Arabic, Czech, Italian, Korean, Dutch and Chinese [5], and performs tasks such as summarization, text classification, question answering, retrieval-augmented generation and code-related tasks [6].

To use a specific model, set the `MODEL_NAME` constant in `worker/model_runtime.py` to the corresponding Hugging Face identifier (for example `"ibm-granite/granite-4.0-micro"` for IBM Granite 4.0 Micro). Make sure your machine has enough memory to run the chosen model.

Step 2: Implement the Worker Model Runtime (Python)

Create a Python script `worker/model_runtime.py` that loads a small transformer model and listens on `stdin` for inference requests. Each request is a JSON object with an `input` field. It returns a JSON response with the generated text to stdout. Install dependencies:

```
cd worker
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
pip install transformers torch

# create model_runtime.py (see below)
```

`worker/model_runtime.py`:

```
#!/usr/bin/env python
import json
import sys
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

# Load a small model.  Update MODEL_NAME to one of the recommended small models
(Phi-2, Mistral 7B, LLaMA 3 8B, IBM Granite 4.0 Micro, etc.).
MODEL_NAME = "microsoft/phi-2"

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForCausalLM.from_pretrained(MODEL_NAME)
model.eval()

def generate_response(prompt: str, max_new_tokens: int = 64):
    inputs = tokenizer(prompt, return_tensors="pt")
    with torch.no_grad():
        outputs = model.generate(**inputs, max_new_tokens=max_new_tokens,
do_sample=False)
    generated = tokenizer.decode(outputs[0][inputs.input_ids.size(1):],
skip_special_tokens=True)
    return generated

for line in sys.stdin:
    try:
```

```
        req = json.loads(line)
        prompt = req.get("input", "")
        response = generate_response(prompt)
        sys.stdout.write(json.dumps({"result": response}) + "\n")
        sys.stdout.flush()
    except Exception as exc:
        sys.stdout.write(json.dumps({"error": str(exc)}) + "\n")
        sys.stdout.flush()
```

Make the script executable and return to the project root:

```
chmod +x model_runtime.py
cd ..  # back to project root
```

Step 3: Implement the Worker Controller (Node.js/TypeScript)

The worker controller is a Node.js program that uses Hyperswarm RPC to announce itself and handle inference requests. It spawns the Python model runtime as a child process and proxies requests to it.

1. Initialize the Node.js project inside the `worker` directory and install dependencies:

```
cd worker
npm init -y
npm install typescript ts-node @types/node hyperswarm hyperswarm-rpc
npx tsc --init
```

1. Create `worker/controller.ts` with the following content:

```
import Hyperswarm from 'hyperswarm';
import { once } from 'events';
import { spawn } from 'child_process';

const REGION = process.env.TETHER_REGION || 'us-east';
const MODEL_ID = process.env.MODEL_ID || 'phi-2';
const VERSION = process.env.MODEL_VERSION || '1.0';

// Start the Python model process
const modelProc = spawn('python', ['model_runtime.py'], { stdio: ['pipe',
'pipe', 'inherit'] });

const swarm = new Hyperswarm();

// Join presence and model topics
const presenceTopic = `tether/presence/${REGION}`;
```

```
const modelTopic = `tether/models/${MODEL_ID}/${VERSION}/${REGION}`;
swarm.join(Buffer.from(presenceTopic), { lookup: true, announce: true });
swarm.join(Buffer.from(modelTopic), { lookup: true, announce: true });

// Periodic heartbeats
setInterval(() => {
  const heartbeat = JSON.stringify({
    workerId: MODEL_ID + '-' + Math.random().toString(36).substring(2, 8),
    modelId: MODEL_ID,
    version: VERSION,
    region: REGION,
    capacity: 1,
    gpuClass: 'cpu'
  });
  const hbConn = swarm.stream();
  hbConn.write(heartbeat);
  hbConn.end();
}, 5000);

swarm.on('connection', async (conn) => {
  for await (const data of conn) {
    try {
      const msg = JSON.parse(data.toString());
      if (msg.type === 'infer.start') {
        const jobId = msg.jobId;
        const input = msg.input;
        modelProc.stdin.write(JSON.stringify({ input }) + '\n');
        const [respRaw] = await once(modelProc.stdout, 'data');
        const resp = JSON.parse(respRaw.toString());
        const response = { type: 'infer.result', jobId, result: resp.result };
        conn.write(JSON.stringify(response));
      }
    } catch (e) {
      console.error('Worker processing error', e);
    }
  }
});
```

1. Run the worker controller:

```
npx ts-node controller.ts
```

Step 4: Implement the Orchestrator/Scheduler

1. In the `orchestrator` directory, initialize a Node.js project and install dependencies:

```
cd ../orchestrator
npm init -y
npm install typescript ts-node @types/node hyperswarm hyperswarm-rpc axios uuid
npx tsc --init
```

1. Create `orchestrator/index.ts`:

```
import Hyperswarm from 'hyperswarm';
import axios from 'axios';
import { v4 as uuidv4 } from 'uuid';

const REGION = process.env.TETHER_REGION || 'us-east';
const presenceTopic = `tether/presence/${REGION}`;

const swarm = new Hyperswarm();
swarm.join(Buffer.from(presenceTopic), { lookup: true, announce: true });

interface WorkerInfo { peer: any; modelId: string; version: string; }
const workers: WorkerInfo[] = [];

swarm.on('connection', async (conn, info) => {
  for await (const data of conn) {
    try {
      const hb = JSON.parse(data.toString());
      workers.push({ peer: info.peer, modelId: hb.modelId, version:
hb.version });
    } catch {}
  }
});

export async function handleJob(job: { tenantId: string; modelId: string;
version: string; input: string; }): Promise<string> {
  const candidate = workers.find(w => w.modelId === job.modelId && w.version
=== job.version);
  if (!candidate) throw new Error('No workers available');
  const jobId = uuidv4();
  // Persist job in rqlite
  await axios.post('http://localhost:4001/db/execute', `INSERT INTO jobs(id,
tenant_id, model_id, version, input, status) VALUES('${jobId}','$
{job.tenantId}','${job.modelId}','${job.version}','${job.input}','pending')`);
  // Dispatch job via Hyperswarm
  const modelTopic = `tether/models/${job.modelId}/${job.version}/${REGION}`;
  const conn = swarm.stream(Buffer.from(modelTopic));
  conn.write(JSON.stringify({ type: 'infer.start', jobId, input: job.input }));
  conn.end();
```

```
  return jobId;
}
```

1. Run the orchestrator:

```
npx ts-node index.ts
```

Step 5: Implement the Gateway (Node.js/TypeScript)

The gateway exposes an HTTP REST endpoint for clients. It validates the request, enforces a simple rate limit and forwards to the orchestrator.

1. In the `gateway` directory, initialize a Node.js project and install dependencies:

```
cd ../gateway
npm init -y
npm install typescript ts-node @types/node express body-parser axios
npx tsc --init
```

1. Create `gateway/index.ts`:

```
import express from 'express';
import bodyParser from 'body-parser';
import axios from 'axios';
import { handleJob } from '../orchestrator/index';

const app = express();
app.use(bodyParser.json());

const limits: Record<string, { count: number; reset: number }> = {};
const limit = 5;
const windowMs = 60 * 1000;

app.post('/infer', async (req, res) => {
  const { tenantId, modelId, version, input } = req.body || {};
  if (!tenantId || !modelId || !version || !input) {
    return res.status(400).json({ error: 'Missing fields' });
  }
  const now = Date.now();
  const record = limits[tenantId] || { count: 0, reset: now + windowMs };
  if (now > record.reset) { record.count = 0; record.reset = now + windowMs; }
  if (record.count >= limit) return res.status(429).json({ error: 'Rate limit
exceeded' });
  record.count += 1;
  limits[tenantId] = record;
```

```
  try {
    const jobId = await handleJob({ tenantId, modelId, version, input });
    const interval = setInterval(async () => {
      const q = `SELECT status,result FROM jobs WHERE id='${jobId}'`;
      const resp = await axios.get(`http://localhost:4001/db/query?q=$
{encodeURIComponent(q)}`);
      const row = resp.data.results?.[0]?.values?.[0];
      if (row && row[0] === 'completed') {
        clearInterval(interval);
        return res.json({ jobId, result: row[1] });
      }
    }, 1000);
  } catch (err) {
    res.status(500).json({ error: (err as Error).message });
  }
});

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Gateway listening on port ${port}`);
});
```

1. Run the gateway:

```
npx ts-node index.ts
```

Step 6: Test the Pilot

Once all services are running (rqlite, worker runtime and controller, orchestrator and gateway), you can send a test request:

```
curl -XPOST http://localhost:3000/infer \
  -H 'Content-Type: application/json' \
  -d '{"tenantId":"demo-tenant","modelId":"phi-2","version":"1.0","input":"What
is the capital of France?"}'
```

The gateway will return a `jobId` and then respond with the final inference result once the worker completes. Logs from the services will show heartbeats, job dispatches and results.

Notes and Extensions

- **rqlite limitations:** rqlite is not designed for high write throughput [7], but it's perfect for a pilot and provides strong consistency across nodes.

- **Model choice:** Replace `MODEL_NAME` in `model_runtime.py` with any Hugging Face model that fits your hardware. Smaller models like `phi-2` or `TinyLLama` will run comfortably on a CPU.
- **Scaling:** For multiple workers, run additional instances of the worker controller with different random worker IDs and separate Python model processes.
- **Security:** This pilot omits authentication, encryption and signed announcements. In production you would generate key pairs for each node, sign presence messages and encrypt traffic over Hyperswarm.
- **Streaming responses:** The simplified gateway polls rqlite for job completion. To stream results token-by-token, forward `infer.result` messages through a WebSocket or Server-Sent Events endpoint.

Following this guide will set up a functioning local inference network using Node.js, Hyperswarm, rqlite and Hugging Face transformers. You can extend and refine it as you move towards a production implementation.

---

[1] [2] [3] 15 Best Small Language Models [SLMs] in 2025 | Dextralabs
https://dextralabs.com/blog/top-small-language-models/

[4] [5] [6] ibm-granite/granite-4.0-micro · Hugging Face
https://huggingface.co/ibm-granite/granite-4.0-micro

[7] RQLITE: Distributed SQLite You Can Actually Use | by Sergio Rafael Coyopae | Medium
https://medium.com/@sergiorch/rqlite-distributed-sqlite-you-can-actually-use-b38925e8af32