RLC Windows Workstation Setup Runbook

Persistent Local LLM Runtime for Agent Orchestration

Target System: MSI Aegis ZS2 Desktop (RTX 5080 16GB, Ryzen 9, 32GB RAM, 2TB SSD)
OS: Windows 11 Home
Primary Goal: Persistent LLM service + Whisper transcription for Master-Agent architecture
Author: Claude (for Tore @ Round Lakes Companies)
Date: December 2025

Table of Contents

Section 0: Pre-flight

0.1 What You Need Before Starting

Physical items:

- [ ] MSI Aegis ZS2 desktop (unboxed, in position)
- [ ] Ethernet cable (strongly recommended over WiFi for 24/7 operation)
- [ ] Monitor, keyboard, mouse
- [ ] Power strip with surge protection (minimum); UPS recommended for later

Accounts/credentials ready:

- [ ] Microsoft account (or plan to create local account)
- [ ] Your home network WiFi password (backup if Ethernet fails)

Time estimate: 2-3 hours for complete setup (longer if Windows updates are slow)

0.2 Decision Points (Already Made)

Decision	Choice	Rationale
LLM Runtime	**Ollama**	Native Windows support, matches your existing LangChain code, simple model management, built-in API server
Service Manager	**NSSM**	Works on Windows Home, reliable, easy to configure
Python Environment	**uv**	Fast, modern, avoids conda complexity
Transcription	**Whisper.cpp**	GPU-accelerated, runs alongside Ollama without conflicts
Network Exposure	**LAN + Tailscale**	Local access for other machines, secure remote access

0.3 What NOT to Do

> ⚠️ **AVOID THESE COMMON MISTAKES:**
>
> - **Don't download drivers from random sites** — Use only NVIDIA's official site or GeForce Experience
> - **Don't install multiple CUDA toolkits** — Ollama bundles its own CUDA; you don't need a separate install
> - **Don't disable Windows Defender entirely** — Just add exclusions for model folders
> - **Don't skip the reboot after driver install** — GPU won't initialize properly
> - **Don't run Ollama as a scheduled task** — Use NSSM for proper service behavior

> - **Don't store models on OneDrive** — Sync conflicts will corrupt them

0.4 Folder Structure We'll Create

```
C:\
├── RLC\
│   ├── models\          # Ollama model storage (symlinked)
│   ├── ollama\          # Ollama config/logs
│   ├── whisper\         # Whisper models and config
│   ├── services\        # NSSM and service scripts
│   ├── logs\            # Centralized logging
│   └── projects\        # Your agent code lives here
└── tools\
    └── nssm\            # Service manager
```

Section 1: Windows First-Boot and Baseline Configuration

1.1 Initial Windows Setup

When you first power on:

1. Select your region and keyboard layout
2. Connect to your network (Ethernet preferred)
3. **Account setup choice:**
   - **Recommended:** Create a LOCAL account (not Microsoft account)
   - During setup, click "Sign-in options" → "Offline account" → "Limited experience"
   - Username suggestion: `rlc-admin` (no spaces, lowercase)
   - This avoids OneDrive auto-sync and simplifies service configuration
4. **Privacy settings:** Turn OFF all optional telemetry (not required, just reduces background noise)
5. Complete setup and reach the desktop

1.2 Disable Sleep and Hibernation

This machine runs 24/7. Sleep will kill your LLM service.

Via Settings UI:

1. Press Win + I to open Settings
2. Navigate: **System → Power & battery** (or **Power**)
3. Click **Screen and sleep**
4. Set all options to **Never**:
    - "When plugged in, turn off my screen after" → **Never** (or 30 minutes if you prefer)
    - "When plugged in, put my device to sleep after" → **Never**

Via PowerShell (run as Administrator):

```powershell
# Disable hibernation entirely (frees up disk space too)
powercfg /hibernate off

# Set sleep timeout to 0 (never)
powercfg /change standby-timeout-ac 0
powercfg /change monitor-timeout-ac 30

# Disable hybrid sleep
powercfg /setacvalueindex SCHEME_CURRENT SUB_SLEEP HYBRIDSLEEP 0
powercfg /setactive SCHEME_CURRENT
```

1.3 Configure Windows Update Behavior

We want updates, but not surprise reboots during trading hours.

Settings path: Win + I **→ Windows Update → Advanced options**

1. **Active hours:** Set to your working hours (e.g., 6 AM to 11 PM)
    - Windows won't auto-reboot during these hours
2. **Get me up to date:** Leave ON (we want security updates)
3. **Download updates over metered connections:** OFF
4. **Notify me when a restart is required:** ON

Pause updates during initial setup:

- Click "Pause for 1 week" to prevent interruptions during this setup process

1.4 Rename the Computer

A clear hostname helps when accessing remotely.

```powershell
# Run as Administrator
Rename-Computer -NewName "RLC-SERVER" -Restart
```

Or via UI: Settings → System → About → Rename this PC

1.5 Enable Remote Desktop Alternative (Windows Home)

Windows Home doesn't include Remote Desktop server. We'll use **Tailscale** + **Parsec** later, but for now, ensure you can access the machine physically for the remaining setup.

Section 2: Driver + Firmware + Updates

2.1 Install Windows Updates First

Order matters: Windows updates before GPU drivers.

1. Press `Win + I` → **Windows Update**
2. Click **Check for updates**
3. Install ALL available updates
4. **Reboot when prompted**
5. Repeat until "You're up to date" with no pending updates

This typically takes 2-3 cycles and 30-60 minutes.

2.2 Install NVIDIA Drivers

Do NOT use the drivers on any included CD. Get fresh drivers.

Option A: GeForce Experience (Recommended for simplicity)

1. Open Edge browser and go to: `https://www.nvidia.com/en-us/geforce/geforce-experience/`
2. Download and install GeForce Experience
3. Create/sign into NVIDIA account (required, unfortunately)

4. Open GeForce Experience → **Drivers** tab
5. Click **Check for updates** then **Download**
6. Choose **Express Installation**
7. **Reboot when complete**

Option B: Direct Driver Download (No account required)

1. Go to: https://www.nvidia.com/Download/index.aspx
2. Select:
   - Product Type: **GeForce**
   - Product Series: **GeForce RTX 50 Series**
   - Product: **GeForce RTX 5080**
   - Operating System: **Windows 11**
   - Download Type: **Game Ready Driver** (GRD) or **Studio Driver** (SD)
   - Click **Search** then **Download**
3. Run the installer
4. Choose **Custom Installation** → check **Perform clean installation**
5. **Reboot when complete**

> 💡 **Studio vs Game Ready:** For LLM inference, either works. Studio drivers are tested for stability in professional workloads. If you also game, GRD is fine.

2.3 MSI Firmware/BIOS Updates (Optional but Recommended)

MSI provides firmware updates through **MSI Center**.

1. Search Start menu for "MSI Center" (pre-installed)
2. Open it and navigate to **Support → Live Update**
3. Check for BIOS and firmware updates
4. Install any available updates
5. **Reboot if prompted**

> ⚠️ **BIOS updates carry small risk.** Only do this if you're comfortable. The system will work fine without it.

2.4 Verify Driver Installation

After reboot, open PowerShell and run:

```
powershell

nvidia-smi
```

Expected output (example):

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 565.xx.xx    Driver Version: 565.xx.xx    CUDA Version: 12.7                 |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  TCC/WDDM  | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 5080      WDDM  | 00000000:01:00.0    On |                  N/A |
|  0%   35C    P8             15W / 360W |     500MiB / 16376MiB  |      0%      Default |
+-----------------------------------------+------------------------+----------------------+
```

What to verify:

- GPU name shows "RTX 5080"
- Driver version shows (any recent version is fine)
- CUDA Version shows 12.x
- Memory shows ~16GB (16376MiB)

If `nvidia-smi` is not found, the driver didn't install correctly. Re-run the installer.

Section 3: Install Core Tooling

3.1 Install Windows Terminal

Windows Terminal is vastly better than CMD or basic PowerShell.

```
powershell

# Run in PowerShell (as Administrator)
winget install Microsoft.WindowsTerminal
```

After install, you can find it in Start menu as "Terminal".

Configure as default:

1. Open Terminal
2. Click the dropdown arrow → **Settings**
3. Set **Default profile** to "PowerShell"
4. Set **Default terminal application** to "Windows Terminal"

3.2 Install Package Managers

Winget (usually pre-installed on Windows 11):

```
powershell

# Verify winget works
winget --version
```

If missing, install "App Installer" from Microsoft Store.

Chocolatey (useful for some tools):

```
powershell

# Run as Administrator
Set-ExecutionPolicy Bypass -Scope Process -Force
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072
iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
```

Close and reopen Terminal after installing Chocolatey.

3.3 Install Git

```
powershell

winget install Git.Git
```

Close and reopen Terminal, then configure:

```
powershell
```

```
git config --global user.name "Tore"
git config --global user.email "your-email@example.com"
git config --global init.defaultBranch main
```

3.4 Install Python via uv

uv is a fast, modern Python package manager that replaces pip/venv/conda complexity.

```
powershell

# Install uv
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Close and reopen Terminal, then verify:

```
powershell

uv --version
```

Install Python:

```
powershell

uv python install 3.11
```

3.5 Install Node.js (Optional, for some integrations)

```
powershell

winget install OpenJS.NodeJS.LTS
```

Verify:

```
powershell

node --version
npm --version
```

3.6 Create RLC Folder Structure

```
powershell
```

```
# Create directory structure
New-Item -ItemType Directory -Force -Path "C:\RLC\models"
New-Item -ItemType Directory -Force -Path "C:\RLC\ollama"
New-Item -ItemType Directory -Force -Path "C:\RLC\whisper"
New-Item -ItemType Directory -Force -Path "C:\RLC\services"
New-Item -ItemType Directory -Force -Path "C:\RLC\logs"
New-Item -ItemType Directory -Force -Path "C:\RLC\projects"
New-Item -ItemType Directory -Force -Path "C:\tools\nssm"

# Verify
Get-ChildItem C:\RLC
```

3.7 Add Windows Defender Exclusions

Defender scanning large model files slows everything down.

```
powershell

# Run as Administrator
Add-MpPreference -ExclusionPath "C:\RLC"
Add-MpPreference -ExclusionPath "$env:USERPROFILE\.ollama"
Add-MpPreference -ExclusionProcess "ollama.exe"
Add-MpPreference -ExclusionProcess "ollama_llama_server.exe"
```

Section 4: GPU Verification

4.1 Detailed GPU Information

```
powershell

# Full nvidia-smi output
nvidia-smi -q

# Just the important bits
nvidia-smi --query-gpu=name,driver_version,memory.total,memory.free,temperature.gpu,power.draw --format=csv
```

Expected output:

```
name, driver_version, memory.total [MiB], memory.free [MiB], temperature.gpu, power.draw [W]
NVIDIA GeForce RTX 5080, 565.xx.xx, 16376 MiB, 15800 MiB, 35, 15.00
```

4.2 CUDA Verification

Ollama bundles its own CUDA runtime, but let's verify the driver's CUDA support:

```powershell
# Check CUDA version supported by driver
nvidia-smi --query-gpu=driver_version,cuda_version --format=csv
```

4.3 GPU Stress Test (Optional)

To verify stability before trusting it with 24/7 workloads:

```powershell
# Install GPU-Z for monitoring (optional)
winget install TechPowerUp.GPU-Z
```

Run GPU-Z and check:

- GPU temperature at idle (should be 30-45°C)
- GPU clock speeds
- Memory clock speeds

4.4 Verify No Integrated GPU Conflicts

The Ryzen 9 in your system likely has no integrated GPU (9900X doesn't), but let's confirm Windows sees only one GPU:

```powershell
Get-WmiObject Win32_VideoController | Select-Object Name, DeviceID, AdapterRAM
```

Expected: Only one entry showing NVIDIA GeForce RTX 5080

Section 5: Install the Local LLM Runtime

5.1 Why Ollama

For your use case, Ollama is the clear choice:

Feature	Ollama	llama.cpp	vLLM
Windows Native	✅ Yes	⚠️ Build required	❌ Linux only
GPU Auto-detect	✅ Yes	Manual config	✅ Yes
API Server	✅ Built-in	Separate binary	✅ Built-in
Model Management	✅ `ollama pull`	Manual download	Manual
Your Existing Code	✅ Already uses it	Rewrite needed	Rewrite needed
Multi-model	✅ Hot-swap	One at a time	One at a time
Concurrent Requests	✅ Yes (configurable)	Limited	✅ Optimized

Alternate path (if you need maximum throughput later): vLLM on WSL2 Ubuntu gives ~10x higher throughput for high-concurrency scenarios. But it requires Linux knowledge and more setup. Start with Ollama; migrate only if you hit limits.

5.2 Install Ollama

```
powershell

winget install Ollama.Ollama
```

Or download directly from: `https://ollama.com/download/windows`

After install, Ollama runs as a background process and adds a system tray icon.

5.3 Configure Ollama Environment

Set environment variables to use our organized folder structure:

```
powershell
```

```
# Run as Administrator

# Set Ollama to store models in C:\RLC\models
[Environment]::SetEnvironmentVariable("OLLAMA_MODELS", "C:\RLC\models", "Machine")

# Enable concurrent requests (3-5 for your agent setup)
[Environment]::SetEnvironmentVariable("OLLAMA_NUM_PARALLEL", "4", "Machine")

# Set context window (increase if your agents need longer context)
[Environment]::SetEnvironmentVariable("OLLAMA_CONTEXT_LENGTH", "8192", "Machine")

# Keep models loaded in memory longer (30 minutes)
[Environment]::SetEnvironmentVariable("OLLAMA_KEEP_ALIVE", "30m", "Machine")

# Bind to all interfaces (for LAN access)
[Environment]::SetEnvironmentVariable("OLLAMA_HOST", "0.0.0.0:11434", "Machine")
```

Important: Close and reopen Terminal for environment variables to take effect.

5.4 Restart Ollama with New Config

```
powershell

# Stop Ollama if running
Stop-Process -Name "ollama" -Force -ErrorAction SilentlyContinue

# Start Ollama (it will use the new environment variables)
Start-Process "ollama" -ArgumentList "serve"
```

5.5 Verify Ollama is Running

```
powershell

# Check if Ollama API responds
curl http://localhost:11434/api/tags
```

Expected output:

```
json

{"models":[]}
```

(Empty because we haven't pulled any models yet)

Section 6: Download Recommended Models

6.1 Model Selection Strategy

Your Master-Agent orchestrates other agents and calls Claude/external LLMs for heavy lifting. The local model needs to be:

- **Fast** (sub-second response for routing decisions)
- **Good at function calling** (tool use is critical)
- **Lightweight enough** to leave VRAM headroom for Whisper

Recommended models:

Model	Size	VRAM	Use Case	Why
qwen2.5:7b	4.7GB	~6GB	Primary orchestrator	Best function-calling in class, fast
llama3.1:8b	4.9GB	~6GB	Alternative orchestrator	Strong general capability
phi3:mini	2.2GB	~3GB	Quick classification	Ultra-fast for routing decisions
nomic-embed-text	274MB	~500MB	Embeddings	For RAG/search if needed

6.2 Pull Primary Model

```
powershell

# Pull the recommended orchestrator model
ollama pull qwen2.5:7b
```

This will download ~5GB. Progress shows in terminal.

6.3 Pull Backup/Alternative Models

```
powershell
```

```
# Alternative orchestrator (good to have options)
ollama pull llama3.1:8b

# Fast model for simple routing
ollama pull phi3:mini

# Embedding model (for vector search)
ollama pull nomic-embed-text
```

6.4 Verify Models Downloaded

```
powershell

ollama list
```

Expected output:

```
NAME                ID              SIZE      MODIFIED
qwen2.5:7b          abcd1234...     4.7 GB    Just now
llama3.1:8b         efgh5678...     4.9 GB    Just now
phi3:mini           ijkl9012...     2.2 GB    Just now
nomic-embed-text    mnop3456...     274 MB    Just now
```

6.5 Quick Model Test

```
powershell

# Test the primary model
ollama run qwen2.5:7b "Respond with only: OPERATIONAL"
```

Expected output:

```
OPERATIONAL
```

Section 7: Configure Persistent Service

7.1 Why NSSM

Windows Home doesn't have the full Service Control Manager UI, and Task Scheduler isn't ideal for always-

running services. **NSSM (Non-Sucking Service Manager)** fills this gap perfectly:

- Runs any executable as a Windows service
- Auto-restarts on crash
- Proper logging
- Works on Windows Home

7.2 Download and Install NSSM

```powershell
# Download NSSM
Invoke-WebRequest -Uri "https://nssm.cc/release/nssm-2.24.zip" -OutFile "$env:TEMP\nssm.zip"

# Extract to tools folder
Expand-Archive -Path "$env:TEMP\nssm.zip" -DestinationPath "C:\tools" -Force

# Copy the correct architecture version
Copy-Item "C:\tools\nssm-2.24\win64\nssm.exe" "C:\tools\nssm\nssm.exe"

# Add to PATH
$currentPath = [Environment]::GetEnvironmentVariable("Path", "Machine")
if ($currentPath -notlike "*C:\tools\nssm*") {
    [Environment]::SetEnvironmentVariable("Path", "$currentPath;C:\tools\nssm", "Machine")
}
```

Close and reopen Terminal.

7.3 Create Ollama Service

First, find where Ollama is installed:

```powershell
Get-Command ollama | Select-Object Source
```

Typically: `C:\Users\<username>\AppData\Local\Programs\Ollama\ollama.exe`

Create the service:

```powershell
```

```powershell
# Run as Administrator
# Replace <username> with your actual username

$ollamaPath = "$env:LOCALAPPDATA\Programs\Ollama\ollama.exe"

# Install the service
nssm install OllamaLLM $ollamaPath serve

# Configure service settings
nssm set OllamaLLM DisplayName "Ollama LLM Service"
nssm set OllamaLLM Description "Local LLM API server for RLC agent orchestration"
nssm set OllamaLLM Start SERVICE_AUTO_START

# Set environment variables for the service
nssm set OllamaLLM AppEnvironmentExtra "OLLAMA_MODELS=C:\RLC\models" "OLLAMA_NUM_PARALLEL=4" "

# Configure logging
nssm set OllamaLLM AppStdout "C:\RLC\logs\ollama-stdout.log"
nssm set OllamaLLM AppStderr "C:\RLC\logs\ollama-stderr.log"
nssm set OllamaLLM AppRotateFiles 1
nssm set OllamaLLM AppRotateBytes 10485760

# Configure restart behavior
nssm set OllamaLLM AppRestartDelay 5000
nssm set OllamaLLM AppThrottle 10000

# Set to run as current user (needed for GPU access)
nssm set OllamaLLM ObjectName ".\$env:USERNAME"
```

> ⚠️ **Note:** You'll be prompted for your Windows password when setting ObjectName. This is required for GPU access.

7.4 Stop Background Ollama and Start Service

```powershell
```

```
# Kill the background Ollama process
Stop-Process -Name "ollama*" -Force -ErrorAction SilentlyContinue

# Remove Ollama from startup (we're using the service now)
Remove-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run" -Name "Ollama" -ErrorAction Sile

# Start the service
nssm start OllamaLLM
```

7.5 Verify Service is Running

```
powershell

# Check service status
nssm status OllamaLLM

# Or via Windows services
Get-Service OllamaLLM

# Verify API responds
curl http://localhost:11434/api/tags
```

Expected:

```
SERVICE_RUNNING
```

7.6 Service Management Commands

```
powershell
```

```
# Stop the service
nssm stop OllamaLLM

# Start the service
nssm start OllamaLLM

# Restart the service
nssm restart OllamaLLM

# View logs (live)
Get-Content "C:\RLC\logs\ollama-stdout.log" -Tail 50 -Wait

# Edit service configuration
nssm edit OllamaLLM

# Remove service entirely (if needed)
nssm remove OllamaLLM confirm
```

7.7 Health Check Script

Create a health check script for monitoring:

```
powershell
```

```
# Create health check script
@'
# Ollama Health Check Script
$endpoint = "http://localhost:11434/api/tags"
$logFile = "C:\RLC\logs\health-check.log"

try {
    $response = Invoke-RestMethod -Uri $endpoint -TimeoutSec 10
    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    "$timestamp - OK - Ollama responding, $($response.models.Count) models loaded" | Out-File -Append $logFile
    exit 0
} catch {
    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    "$timestamp - FAIL - Ollama not responding: $_" | Out-File -Append $logFile

    # Attempt restart
    nssm restart OllamaLLM
    exit 1
}
'@ | Out-File -FilePath "C:\RLC\services\health-check.ps1" -Encoding UTF8
```

Schedule the health check (every 5 minutes):

```
powershell

# Create scheduled task for health check
$action = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-ExecutionPolicy Bypass -File C:\RLC\services
$trigger = New-ScheduledTaskTrigger -RepetitionInterval (New-TimeSpan -Minutes 5) -At (Get-Date) -Once
$settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -StartWhenAvailable
Register-ScheduledTask -TaskName "OllamaHealthCheck" -Action $action -Trigger $trigger -Settings $settings -Description
```

Section 8: Expose API Endpoint

8.1 Local API Access

The Ollama API is already running at `http://localhost:11434`. Test it:

```
powershell
```

```
# List models
curl http://localhost:11434/api/tags

# Chat completion (streaming)
curl http://localhost:11434/api/chat -d '{
  "model": "qwen2.5:7b",
  "messages": [{"role": "user", "content": "Say hello"}],
  "stream": false
}'
```

Expected response:

```
json

{
  "model": "qwen2.5:7b",
  "created_at": "2025-12-17T...",
  "message": {
    "role": "assistant",
    "content": "Hello! How can I assist you today?"
  },
  "done": true
}
```

8.2 OpenAI-Compatible Endpoint

Ollama provides OpenAI-compatible endpoints, which your LangChain code can use:

```
powershell

# OpenAI-compatible chat completion
curl http://localhost:11434/v1/chat/completions -H "Content-Type: application/json" -d '{
  "model": "qwen2.5:7b",
  "messages": [{"role": "user", "content": "Say hello"}]
}'
```

For your LangChain config, update to:

```
python
```

```
from langchain_community.llms import Ollama

llm = Ollama(
    model="qwen2.5:7b",
    base_url="http://localhost:11434"  # or http://RLC-SERVER:11434 from other machines
)
```

8.3 LAN Access Configuration

We already set `OLLAMA_HOST=0.0.0.0:11434` which binds to all interfaces. Now we need to allow it through the firewall.

```
powershell

# Run as Administrator
New-NetFirewallRule -DisplayName "Ollama LLM API" -Direction Inbound -Protocol TCP -LocalPort 11434 -Action Allow
```

Test from another machine on your network:

```
bash

# From your laptop (replace with your desktop's IP)
curl http://192.168.1.X:11434/api/tags
```

Find your desktop's IP:

```
powershell

Get-NetIPAddress -AddressFamily IPv4 | Where-Object { $_.InterfaceAlias -notlike "*Loopback*" } | Select-Object IPAddre
```

8.4 Tailscale for Secure Remote Access

For accessing your LLM when you're away from home:

```
powershell

winget install Tailscale.Tailscale
```

1. Open Tailscale and sign in (Google account works)
2. Your desktop gets a Tailscale IP (like `100.x.x.x`)

3. Install Tailscale on your laptops
4. Access Ollama via `http://100.x.x.x:11434` from anywhere

8.5 Example API Requests

Basic chat:

```bash
curl http://localhost:11434/api/chat -d '{
  "model": "qwen2.5:7b",
  "messages": [
    {"role": "system", "content": "You are an RLC assistant specializing in commodity markets."},
    {"role": "user", "content": "What factors affect soybean prices?"}
  ],
  "stream": false
}'
```

Generate embedding:

```bash
curl http://localhost:11434/api/embeddings -d '{
  "model": "nomic-embed-text",
  "prompt": "soybean export inspection data"
}'
```

List running models (VRAM usage):

```bash
curl http://localhost:11434/api/ps
```

Section 9: Performance and Stability Tuning

9.1 Windows Power Plan

Set to High Performance for consistent GPU clocks:

```powershell
```

```
# List available power plans
powercfg /list

# Set to High Performance
powercfg /setactive 8c5e7fda-e8bf-4a96-9a85-a6e23a8c635c

# Or create Ultimate Performance plan (hidden by default)
powercfg -duplicatescheme e9a42b02-d5df-448d-aa00-03f14749eb61
powercfg /setactive <GUID from output>
```

9.2 NVIDIA Control Panel Settings

Open NVIDIA Control Panel (right-click desktop → NVIDIA Control Panel):

3D Settings → Manage 3D Settings → Global Settings:

- Power management mode: **Prefer maximum performance**
- Threaded optimization: **On**

Desktop → Enable Desktop GPU indicator: On (optional, shows GPU activity)

9.3 VRAM Usage Optimization

Monitor and manage VRAM:

```
powershell

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

# Watch VRAM continuously
nvidia-smi -l 2
```

VRAM Budget for Your Setup:

```
Total VRAM:         16 GB
─────────────────────────────────────
qwen2.5:7b:         ~6 GB (loaded)
Whisper large-v3:   ~3 GB (when transcribing)
System/overhead:    ~2 GB
─────────────────────────────────────
Available:          ~5 GB (for bursting)
```

If running multiple models simultaneously, use smaller quantizations:

```powershell
ollama pull qwen2.5:7b-instruct-q4_K_M  # 4-bit quantized, ~4GB
```

9.4 Thermal Management

MSI Aegis has good cooling, but verify under load:

```powershell
# Monitor temperature during inference
nvidia-smi --query-gpu=temperature.gpu,fan.speed,power.draw --format=csv -l 5
```

Healthy ranges:

- Idle: 30-45°C
- Under load: 60-80°C
- Throttling starts: 83°C+ (avoid this)

If temperatures are high:

1. Ensure desktop has clearance for airflow
2. Open MSI Center → Fan control → Set to "Cooler Boost" or custom curve
3. Clean dust filters monthly

9.5 Process Priority

Ollama should run at normal priority. Don't elevate it—Windows will starve other processes.

If you see inference slowdowns, check for competing processes:

```powershell
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10 Name, CPU, WorkingSet
```

9.6 Memory (RAM) Management

With 32GB RAM, you have plenty. But verify Ollama isn't swapping:

```powershell
# Check system memory
Get-CimInstance Win32_OperatingSystem | Select-Object FreePhysicalMemory, TotalVisibleMemorySize
```

Set Ollama's runner memory limit if needed (add to environment variables):

```powershell
[Environment]::SetEnvironmentVariable("OLLAMA_MAX_LOADED_MODELS", "2", "Machine")
```

Section 10: Whisper Transcription Setup

10.1 Overview

For your "listen to conversations" feature, we'll set up:

1. **Whisper.cpp** — GPU-accelerated transcription
2. **A Python service** — Monitors audio input, transcribes, stores results
3. **Integration point** — Your Master-Agent reads transcriptions for action items

This is separate from Ollama and runs alongside it.

10.2 Install Whisper.cpp (GPU-accelerated)

Option A: Pre-built binaries

```powershell
```

```
# Create whisper directory
New-Item -ItemType Directory -Force -Path "C:\RLC\whisper\bin"
cd C:\RLC\whisper

# Download latest release (check https://github.com/ggerganov/whisper.cpp/releases)
# Look for "whisper-bin-x64.zip" with CUDA support
Invoke-WebRequest -Uri "https://github.com/ggerganov/whisper.cpp/releases/download/v1.7.2/whisper-cublas-12.4.0-bin-x6

Expand-Archive -Path "whisper.zip" -DestinationPath "C:\RLC\whisper\bin" -Force
```

Option B: Using faster-whisper (Python, easier)

```
powershell

cd C:\RLC\whisper

# Create virtual environment
uv venv
.\.venv\Scripts\activate

# Install faster-whisper with CUDA support
uv pip install faster-whisper sounddevice numpy
```

10.3 Download Whisper Model

```
powershell

# Create models directory
New-Item -ItemType Directory -Force -Path "C:\RLC\whisper\models"
cd C:\RLC\whisper\models

# Download medium model (good balance of speed/accuracy)
# For whisper.cpp format:
Invoke-WebRequest -Uri "https://huggingface.co/ggerganov/whisper.cpp/resolve/main/ggml-medium.bin" -OutFile "ggml-me
```

For faster-whisper, models download automatically on first use.

10.4 Create Transcription Service

Create a Python script for continuous transcription:

```
powershell
```

```
@'
"""
RLC Continuous Transcription Service
Listens to microphone, transcribes speech, saves to file for agent processing
"""

import sounddevice as sd
import numpy as np
from faster_whisper import WhisperModel
import json
from datetime import datetime
from pathlib import Path
import queue
import threading
import time

# Configuration
SAMPLE_RATE = 16000
CHUNK_DURATION = 30  # seconds of audio to process at once
SILENCE_THRESHOLD = 0.01
OUTPUT_DIR = Path("C:/RLC/whisper/transcripts")
MODEL_SIZE = "medium"  # tiny, base, small, medium, large-v3

# Initialize
OUTPUT_DIR.mkdir(exist_ok=True)
audio_queue = queue.Queue()

print("Loading Whisper model (this takes a moment)...")
model = WhisperModel(MODEL_SIZE, device="cuda", compute_type="float16")
print("Model loaded!")

def audio_callback(indata, frames, time_info, status):
    """Called for each audio block from the microphone"""
    if status:
        print(f"Audio status: {status}")
    audio_queue.put(indata.copy())

def process_audio():
    """Process audio chunks and transcribe"""
    audio_buffer = []
    buffer_duration = 0

    while True:
```

```
try:
    # Get audio chunk from queue
    chunk = audio_queue.get(timeout=1)
    audio_buffer.append(chunk)
    buffer_duration += len(chunk) / SAMPLE_RATE

    # Process when we have enough audio
    if buffer_duration >= CHUNK_DURATION:
        audio_data = np.concatenate(audio_buffer).flatten()

        # Check if there's actual audio (not silence)
        if np.abs(audio_data).mean() > SILENCE_THRESHOLD:
            print(f"Transcribing {buffer_duration:.1f}s of audio...")

            segments, info = model.transcribe(audio_data, beam_size=5)

            text = " ".join([seg.text for seg in segments])

            if text.strip():
                timestamp = datetime.now().isoformat()
                transcript = {
                    "timestamp": timestamp,
                    "duration": buffer_duration,
                    "text": text.strip(),
                    "language": info.language,
                    "language_probability": info.language_probability
                }

                # Save to daily file
                date_str = datetime.now().strftime("%Y-%m-%d")
                output_file = OUTPUT_DIR / f"transcript_{date_str}.jsonl"

                with open(output_file, "a", encoding="utf-8") as f:
                    f.write(json.dumps(transcript) + "\n")

                print(f"[{timestamp}] {text[:100]}...")

        # Reset buffer
        audio_buffer = []
        buffer_duration = 0

except queue.Empty:
    continue
except Exception as e:
```

```
            print(f"Error processing audio: {e}")

def main():
    print("\n=== RLC Transcription Service ===")
    print(f"Output directory: {OUTPUT_DIR}")
    print("Press Ctrl+C to stop\n")

    # Start processing thread
    process_thread = threading.Thread(target=process_audio, daemon=True)
    process_thread.start()

    # Start audio stream
    with sd.InputStream(callback=audio_callback, channels=1, samplerate=SAMPLE_RATE):
        print("Listening...")
        try:
            while True:
                time.sleep(0.1)
        except KeyboardInterrupt:
            print("\nStopping...")

if __name__ == "__main__":
    main()
'@ | Out-File -FilePath "C:\RLC\whisper\transcribe_service.py" -Encoding UTF8
```

10.5 Test Transcription

```
powershell

cd C:\RLC\whisper
.\.venv\Scripts\activate
python transcribe_service.py
```

Speak into your microphone. You should see transcriptions appear.

10.6 Create Whisper Service (Optional)

If you want transcription to run as a service too:

```
powershell
```

```
# Install as service
nssm install WhisperTranscribe "C:\RLC\whisper\.venv\Scripts\python.exe" "C:\RLC\whisper\transcribe_service.py"
nssm set WhisperTranscribe DisplayName "RLC Whisper Transcription"
nssm set WhisperTranscribe AppDirectory "C:\RLC\whisper"
nssm set WhisperTranscribe AppStdout "C:\RLC\logs\whisper-stdout.log"
nssm set WhisperTranscribe AppStderr "C:\RLC\logs\whisper-stderr.log"
```

> 💡 **Note:** Running Whisper continuously uses ~3GB VRAM. You can start/stop it as needed, or adjust the model size to `small` (~1GB) for always-on use.

10.7 Integration with Your Master-Agent

Your Master-Agent can read transcripts from `C:\RLC\whisper\transcripts\`:

```python
# Example: Read today's transcripts
from pathlib import Path
import json
from datetime import datetime

transcript_dir = Path("C:/RLC/whisper/transcripts")
today = datetime.now().strftime("%Y-%m-%d")
transcript_file = transcript_dir / f"transcript_{today}.jsonl"

if transcript_file.exists():
    transcripts = []
    with open(transcript_file, "r") as f:
        for line in f:
            transcripts.append(json.loads(line))

    # Pass to LLM for action item extraction
    # ...
```

Section 11: Backups, Updates, and Rollback

11.1 What to Back Up

Item	Location	Backup Method
Ollama models	`C:\RLC\models`	Copy to external drive
Service configs	`C:\RLC\services`	Git repo or cloud sync
Transcripts	`C:\RLC\whisper\transcripts`	Daily sync to cloud
Project code	`C:\RLC\projects`	Git (primary)
Environment variables	System	Export script below

Export environment variables:

```powershell
[Environment]::GetEnvironmentVariables("Machine") | ConvertTo-Json | Out-File "C:\RLC\backups\env-machine.json"
```

11.2 Updating Ollama

```powershell
# Check current version
ollama --version

# Update via winget
winget upgrade Ollama.Ollama

# Restart service
nssm restart OllamaLLM

# Verify new version
ollama --version
```

11.3 Updating Models

```powershell
```

```
# Pull latest version of a model
ollama pull qwen2.5:7b

# List all models with dates
ollama list

# Remove old versions (optional)
ollama rm qwen2.5:7b-old
```

11.4 Updating NVIDIA Drivers

1. **Before updating:** Note current working driver version

```
powershell

  nvidia-smi --query-gpu=driver_version --format=csv
```

2. **Download new driver** from nvidia.com or GeForce Experience
3. **Install with clean option** if troubleshooting
4. **Reboot and verify:**

```
powershell

  nvidia-smi
  ollama run qwen2.5:7b "test"
```

11.5 Rollback Procedures

Ollama rollback:

```
powershell

# If Ollama update breaks things:
winget uninstall Ollama.Ollama
# Download specific version from GitHub releases
# https://github.com/ollama/ollama/releases
```

Driver rollback:

Device Manager → Display adapters → NVIDIA GeForce RTX 5080
Right-click → Properties → Driver tab → Roll Back Driver

Model rollback:

```powershell
# If a model update performs worse:
ollama rm qwen2.5:7b
# Re-pull specific version (if available) or use different quantization
ollama pull qwen2.5:7b-q4_K_M
```

11.6 Scheduled Backup Script

```powershell
@'
# RLC Backup Script
$timestamp = Get-Date -Format "yyyy-MM-dd"
$backupDir = "D:\RLC-Backups\$timestamp"  # Change to your backup drive

New-Item -ItemType Directory -Force -Path $backupDir

# Backup configs
Copy-Item -Recurse "C:\RLC\services" "$backupDir\services"
Copy-Item -Recurse "C:\RLC\ollama" "$backupDir\ollama"

# Backup transcripts (last 7 days)
$transcriptBackup = "$backupDir\transcripts"
New-Item -ItemType Directory -Force -Path $transcriptBackup
Get-ChildItem "C:\RLC\whisper\transcripts" -Filter "*.jsonl" |
    Where-Object { $_.LastWriteTime -gt (Get-Date).AddDays(-7) } |
    Copy-Item -Destination $transcriptBackup

# Export environment
[Environment]::GetEnvironmentVariables("Machine") | ConvertTo-Json | Out-File "$backupDir\env-machine.json"

# Log completion
"Backup completed: $timestamp" | Out-File -Append "C:\RLC\logs\backup.log"
'@ | Out-File -FilePath "C:\RLC\services\backup.ps1" -Encoding UTF8
```

Section 12: Troubleshooting

12.1 CUDA/Driver Mismatch

Symptom: Ollama fails to use GPU, falls back to CPU

Check:

```
powershell

nvidia-smi
ollama run qwen2.5:7b "test" --verbose
```

Solutions:

1. Ensure driver is installed correctly (nvidia-smi works)
2. Reboot after driver install
3. Reinstall Ollama (it bundles its own CUDA runtime)
4. Check Windows Event Viewer for GPU errors

12.2 VRAM Out of Memory (OOM)

Symptom: "CUDA out of memory" or inference suddenly fails

Check:

```
powershell

nvidia-smi
curl http://localhost:11434/api/ps  # See loaded models
```

Solutions:

1. Unload unused models:

```
powershell

  # Models unload after OLLAMA_KEEP_ALIVE timeout
  # Or restart service to force unload
  nssm restart OllamaLLM
```

2. Use smaller quantization:

```
powershell

  ollama pull qwen2.5:7b-instruct-q4_K_M
```

3. Reduce concurrent requests:

```
powershell

  [Environment]::SetEnvironmentVariable("OLLAMA_NUM_PARALLEL", "2", "Machine")
  nssm restart OllamaLLM
```

12.3 Runtime Not Seeing GPU

Symptom: Inference is slow (CPU speed instead of GPU)

Check:

```
powershell

ollama run qwen2.5:7b "test" --verbose 2>&1 | Select-String "GPU"
```

Solutions:

1. Verify NVIDIA driver:

```
powershell

  nvidia-smi
```

2. Check if another process is using all VRAM:

```
powershell

  nvidia-smi --query-compute-apps=pid,name,used_memory --format=csv
```

3. Reinstall Ollama
4. Check for Windows GPU scheduling issues:
    - Settings → System → Display → Graphics settings

- Ensure "Hardware-accelerated GPU scheduling" is ON

12.4 Slow Inference

Symptom: Responses take 10+ seconds for simple queries

Check:

```powershell
# Time a simple request
Measure-Command { ollama run qwen2.5:7b "Say OK" }
```

Solutions:

1. Verify GPU is being used (see 12.3)
2. Check model is loaded (first request loads model):

```powershell
curl http://localhost:11434/api/ps
```

3. Increase keep-alive time so model stays loaded:

```powershell
[Environment]::SetEnvironmentVariable("OLLAMA_KEEP_ALIVE", "1h", "Machine")
```

4. Check for thermal throttling:

```powershell
nvidia-smi --query-gpu=temperature.gpu,clocks.gr,clocks.mem --format=csv
```

12.5 Service Won't Start

Symptom: `nssm status OllamaLLM` shows STOPPED or error

Check:

```powershell
```

```
# View service logs
Get-Content "C:\RLC\logs\ollama-stderr.log" -Tail 50

# Check Windows Event Viewer
Get-EventLog -LogName Application -Source "nssm" -Newest 10
```

Solutions:

1. Verify Ollama path is correct:

```
powershell
  Test-Path "$env:LOCALAPPDATA\Programs\Ollama\ollama.exe"
```

2. Try running Ollama manually first:

```
powershell
  & "$env:LOCALAPPDATA\Programs\Ollama\ollama.exe" serve
```

3. Recreate service:

```
powershell
  nssm remove OllamaLLM confirm
  # Re-run Section 7.3
```

4. Check if port is in use:

```
powershell
  netstat -ano | findstr "11434"
```

12.6 Firewall Blocks

Symptom: Can access from localhost but not from LAN

Check:

```
powershell
```

```
# From the server
curl http://localhost:11434/api/tags  # Should work

# From another machine
curl http://192.168.1.X:11434/api/tags  # Fails
```

Solutions:

1. Verify firewall rule exists:

```
powershell

  Get-NetFirewallRule -DisplayName "*Ollama*"
```

2. Add rule if missing:

```
powershell

  New-NetFirewallRule -DisplayName "Ollama LLM API" -Direction Inbound -Protocol TCP -LocalPort 11434 -Action Allo
```

3. Check Windows Firewall isn't blocking all inbound:
   - Settings → Privacy & Security → Windows Security → Firewall
   - Ensure "Private network" isn't fully blocking inbound
4. Verify Ollama is bound to all interfaces:

```
powershell

  netstat -ano | findstr "11434"
  # Should show 0.0.0.0:11434, not 127.0.0.1:11434
```

12.7 Model Download Issues

Symptom: `ollama pull` fails or hangs

Solutions:

1. Check internet connectivity:

```powershell
Test-NetConnection -ComputerName "registry.ollama.ai" -Port 443
```

2. Retry with verbose output:

```powershell
$env:OLLAMA_DEBUG = "1"
ollama pull qwen2.5:7b
```

3. Clear partial downloads:

```powershell
Remove-Item "C:\RLC\models\blobs\*" -Force
ollama pull qwen2.5:7b
```

4. Check disk space:

```powershell
Get-PSDrive C | Select-Object Free
```

12.8 Whisper/Microphone Issues

Symptom: Transcription not working

Solutions:

1. Verify microphone access:
   - Settings → Privacy → Microphone → Ensure apps can access
2. Test microphone:

```powershell
# In Python environment
python -c "import sounddevice; print(sounddevice.query_devices())"
```

3. Check correct device is default:

- Right-click speaker icon → Sound settings → Input

4. Verify VRAM availability for Whisper:

```powershell
nvidia-smi
```

Appendix A: Quick Reference Commands

Service Management

```powershell
nssm start OllamaLLM        # Start Ollama service
nssm stop OllamaLLM         # Stop Ollama service
nssm restart OllamaLLM      # Restart Ollama service
nssm status OllamaLLM       # Check service status
nssm edit OllamaLLM         # Edit service config (GUI)
```

Ollama Commands

```powershell
ollama list                 # List installed models
ollama ps                   # Show running models
ollama pull <model>         # Download a model
ollama rm <model>           # Remove a model
ollama run <model> "prompt"  # Quick test
ollama show <model>         # Model details
```

API Endpoints

```
GET  http://localhost:11434/api/tags            # List models
POST http://localhost:11434/api/chat            # Chat completion
POST http://localhost:11434/api/generate        # Text completion
POST http://localhost:11434/api/embeddings      # Generate embeddings
GET  http://localhost:11434/api/ps              # Running models
POST http://localhost:11434/v1/chat/completions  # OpenAI-compatible
```

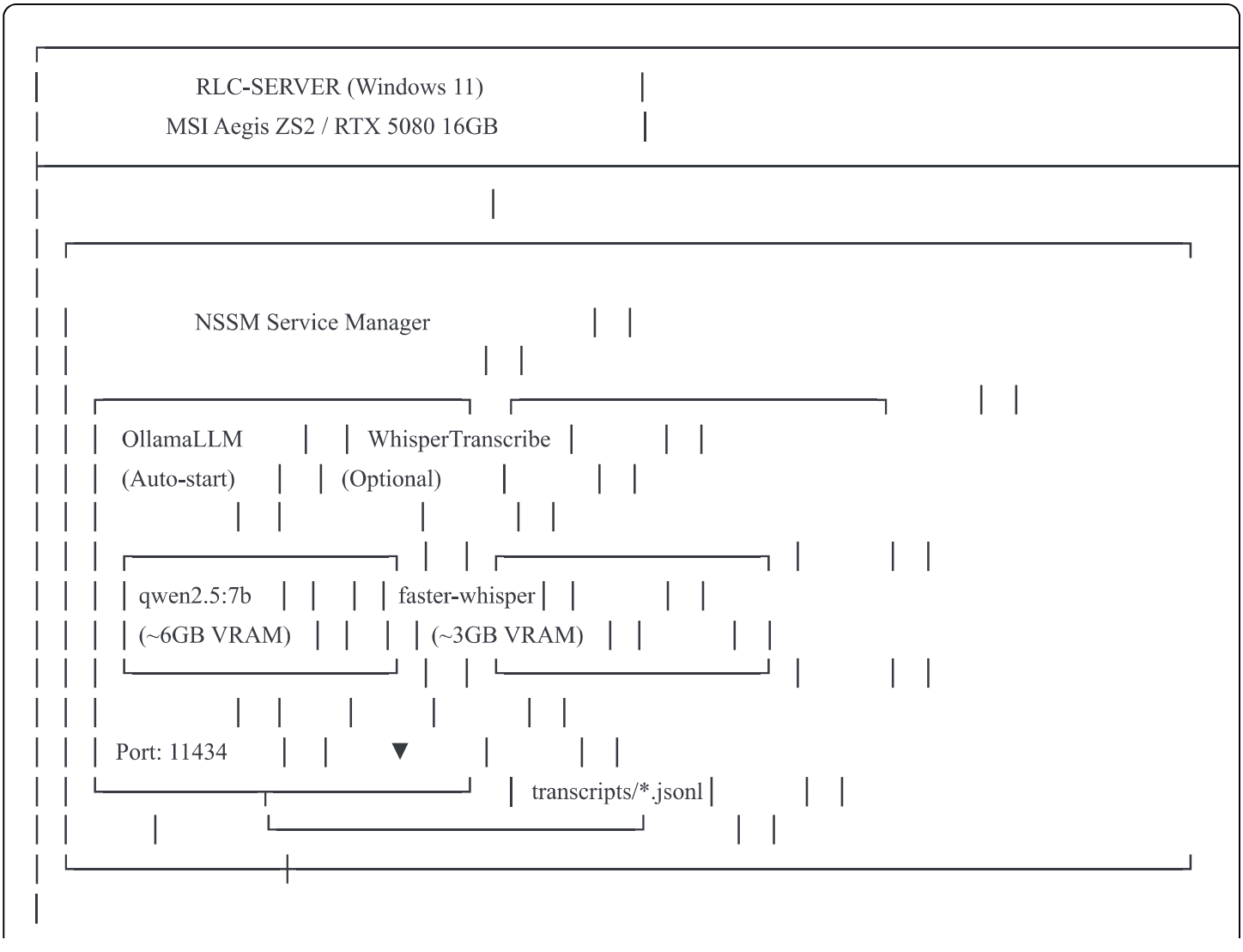
GPU Monitoring

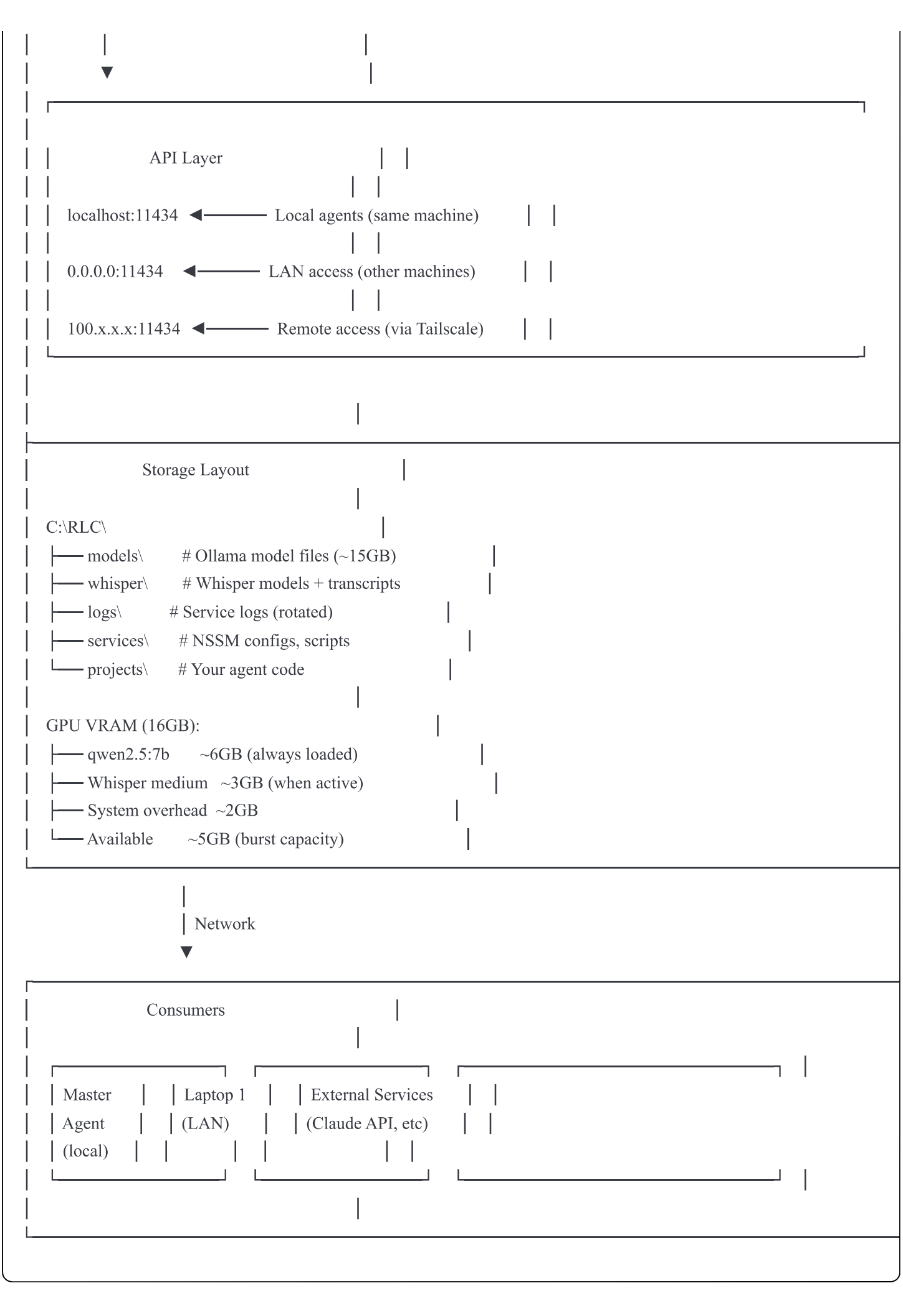
```powershell
nvidia-smi                # GPU status snapshot
nvidia-smi -l 2           # Live monitoring (2s refresh)
nvidia-smi --query-gpu=memory.used,memory.free,temperature.gpu,power.draw --format=csv
```

Log Locations

```
C:\RLC\logs\ollama-stdout.log    # Ollama output
C:\RLC\logs\ollama-stderr.log    # Ollama errors
C:\RLC\logs\whisper-stdout.log   # Whisper output
C:\RLC\logs\health-check.log     # Health check results
```

Appendix B: Architecture Diagram




API Layer
localhost:11434
Local agents (same machine)
0.0.0.0:11434
LAN access (other machines)
100.x.x.x:11434
Remote access (via Tailscale)
Storage Layout
C:\RLC\
├── models\ # Ollama model files (~15GB)
├── whisper\ # Whisper models + transcripts
├── logs\ # Service logs (rotated)
├── services\ # NSSM configs, scripts
└── projects\ # Your agent code
GPU VRAM (16GB):
├── qwen2.5:7b ~6GB (always loaded)
├── Whisper medium ~3GB (when active)
├── System overhead ~2GB
└── Available ~5GB (burst capacity)
Network
Consumers
Master Agent (local)
Laptop 1 (LAN)
External Services (Claude API, etc)

Verification Checklist

Run through this after completing setup:

```powershell
# 1. GPU detected
nvidia-smi --query-gpu=name,memory.total --format=csv
# Expected: NVIDIA GeForce RTX 5080, 16376 MiB

# 2. Service running
nssm status OllamaLLM
# Expected: SERVICE_RUNNING

# 3. API responding
curl http://localhost:11434/api/tags
# Expected: JSON with model list

# 4. Model loaded and working
Measure-Command { ollama run qwen2.5:7b "Say OK" --verbose }
# Expected: < 3 seconds, output mentions GPU

# 5. LAN access (from another machine)
curl http://<RLC-SERVER-IP>:11434/api/tags
# Expected: Same JSON response

# 6. Health check scheduled
Get-ScheduledTask -TaskName "OllamaHealthCheck"
# Expected: Task exists, State = Ready

# 7. Logs being written
Test-Path "C:\RLC\logs\ollama-stdout.log"
# Expected: True

# 8. Environment variables set
[Environment]::GetEnvironmentVariable("OLLAMA_HOST", "Machine")
# Expected: 0.0.0.0:11434
```

All checks passed? Your persistent LLM is ready for agent integration!

Last updated: December 2025 For issues, check Section 12: Troubleshooting