Production Requirements Document (PRD)

Program Name

Local LLM Framework (LLF)

Program Purpose

Local LLM Framework (LLF) is a Python-based application designed to run Large Language Models (LLMs) **entirely on local computer hardware** using **vLLM** as the runtime engine.

The primary goal is to provide **maximum flexibility, zero token costs, and full local control**, while exposing the LLM through multiple access methods over time.

Phase 1 scope focuses exclusively on CLI-based interaction.

Goals & Objectives

Primary Objectives (Phase 1)

- Run a modern LLM locally using **vLLM**
- Download and manage a local LLM model
- Provide a **CLI interface** for sending prompts and receiving responses
- Ensure production-quality code with:
    - Modular architecture
    - Unit testing (≥ 80% coverage)
    - Clear documentation
    - Clean installation and uninstall process

Non-Goals (Out of Scope for Phase 1)

- GUI interface
- API server exposure
- Voice input/output
- Internet access
- Tool execution (commands, filesystem access)
- Multi-model switching via configuration

These features are **explicitly deferred** but **must be considered in architectural design** to avoid rework later.

Technology Stack

Programming Language

- **Python 3.11+**

LLM Runtime

- **vLLM**

Target Model (Phase 1)

- **Qwen3-Coder**

Operating Environment

- Local machine (Mac, Linux, Windows supported where vLLM allows)
- No cloud dependency required

Architectural Decision (Phase 1)

Selected Architecture

✅ Single-Process CLI Application

A single Python program that:

- Loads the LLM
- Accepts user input via CLI
- Executes inference
- Returns output to the user

Rationale

- Simplest architecture for initial development
- Easier debugging and testing
- Faster iteration with Claude Code

- Avoids early complexity of IPC, background services, or API layers

⚠️ **Design Constraint**
Code **must be modularized** so the LLM execution logic can later be extracted into:

- A background service
- An OpenAI-compatible API server
- A GUI or web frontend

High-Level System Components

1. Model Management Module

Responsible for:

- Downloading the Qwen3-Coder model locally
- Storing models in a predictable directory structure
- Verifying model availability before runtime

Example responsibilities:

- Check if model exists locally
- Download model if missing
- Validate integrity (basic checks)

---

2. LLM Runtime Module

Responsible for:

- Initializing vLLM
- Loading the model
- Managing inference parameters
- Executing prompt → response

Constraints:

- Use vLLM's **OpenAI-compatible APIs**
- Designed so backend can later be swapped without breaking the CLI

---

3. CLI Interface Module

Responsible for:

- Accepting user input from terminal
- Sending prompts to the LLM runtime
- Displaying formatted responses
- Gracefully handling errors and exits

CLI Features (Phase 1):

- Interactive prompt loop
- Exit command (`exit`, `quit`, or Ctrl+C)
- Basic help command (`--help`)

---

4. Configuration Module

Phase 1 scope:

- Minimal configuration via constants or simple config file
- Model name
- Model storage path
- Default inference parameters

⚠️ Must be designed so this evolves into:

- Multi-model configuration
- Tool permissions
- Future module toggles

---

5. Logging Module

Responsible for:

- Logging startup
- Model load status
- Errors and exceptions

Constraints:

- Use Python's standard `logging` library
- Console output only (Phase 1)

Directory Structure (Proposed)

```
local_llm_framework/
│
├── llf/
│   ├── __init__.py
│   ├── cli.py
│   ├── config.py
│   ├── model_manager.py
│   ├── llm_runtime.py
│   └── logging_config.py
│
├── tests/
│   ├── test_cli.py
│   ├── test_model_manager.py
│   └── test_llm_runtime.py
│
├── requirements.txt
├── README.md
└── setup.py (optional, if packaging is desired)
```

Libraries & Dependencies

Required Libraries

- vllm
- openai (used against vLLM's OpenAI-compatible API)
- huggingface_hub (for model downloading)
- rich (optional but recommended for CLI UX)
- pytest
- pytest-cov

If multiple libraries could be used for a task, the **best-fit library should be selected**, consulting the Internet if necessary.

Testing Requirements

Unit Testing

- Framework: **pytest**
- Minimum coverage: **80%**
- Tests must cover:
    - Model download logic

- LLM initialization
- Prompt execution flow
- CLI input handling (mocked)

Coverage Enforcement

- Use pytest-cov
- Coverage results must be documented in README

Installation Requirements

Python Virtual Environment (Required)

The application **must be installed and run inside a Python virtual environment**.
For Phase 1, the virtual environment will be created **inside the repository**.

Setup Virtual Environment

```
python -m venv llf_venv
```

Start Virtual Environment

```
source llf_venv/bin/activate
```

Verify Active Environment

```
echo $VIRTUAL_ENV
```

Stop Virtual Environment

```
deactivate
```

Dependency Management

- All dependencies listed in requirements.txt
- Install via:
- pip install -r requirements.txt

Model Download

- Model is downloaded on first run or via a dedicated command

- No manual user steps required

Documentation Requirements

README.md Must Include:

1. Overview of LLF
2. System requirements
3. Installation steps
4. How to run the CLI
5. How the model is downloaded
6. Configuration (if applicable)
7. How to run tests
8. How to uninstall
9. Known limitations (Phase 1)

Future-Proofing Requirements (Non-Functional)

Even though the following are **out of scope**, the code must not block them:

- Multiple LLMs via config file
- API-based access
- GUI frontend
- Voice input/output
- Tool execution (commands, filesystem, internet)
- Permission-based tool access

Key Rule:

No logic should be tightly coupled to CLI-only assumptions.

Success Criteria (Phase 1)

- User can install LLF locally
- User can run a CLI command
- LLM loads successfully via vLLM
- User can send prompts and receive responses
- Tests pass with ≥ 80% coverage
- Codebase is clean, modular, and extensible