Statement of Work

I. Introduction

Our group was established in the first semester of 2025, with the primary objective of developing a container-based AI application development platform. This platform is built upon ollama (for running large language models), langchain (as the application framework), and gradio (for frontend interaction). By integrating these tools into a custom Docker image, we aim to create a reusable and modular development pipeline, which allows users to rapidly build and deploy AI applications locally across different platforms.

We chose ollama, langchain, and gradio because they are lightweight, open-source, and easy to combine, which makes them very suitable for efficient development and deployment in local environments. By building a custom image, we hope to provide a unified starting point for local AI application development, improving compatibility, deployment efficiency, and development experience.

Our team consists of six members. Their names and student numbers are as follows:

Boyang Zhang u7760642
Diming Xu u7705332
Dongze Yu u7775416
Qingchuan Rui u7776331
Xiangyu Tan u7779491
Zhuiqi Lin u7733924

II. Purpose of Project

Our project primarily focuses on developing a reusable, containerized AI application development platform using ollama, langchain, and gradio. Instead of simply building a local chatbot, our goal is to create a custom Docker image that integrates these three components, forming a ready-to-use development pipeline. This pipeline allows users to run large language models offline, while also offering a modular structure for building AI applications efficiently across different local environments. This approach ensures better cross-platform compatibility, privacy, and deployment convenience, and helps our team gain

hands-on experience in containerization and modular AI application design. Specifically, we will focus on the following deliverables:

1. Build a custom container image that can successfully load and run large language models via ollama in local environments
2. Integrate langchain into the image to form a reusable modular chain logic, realizing a complete prompt → chain → response flow
3. Use gradio to build a frontend interaction interface, which will run automatically as a web service after the container starts and support user-model interaction
4. Provide a complete Dockerfile, deployment scripts, and usage documentation to help users reproduce the setup and develop their own AI applications
5. (Optional) Extend model capabilities, such as enabling document-based question answering through local knowledge base

III. Scope of Work

Milestone 1: Write the Statement of Work (SOW)
Schedule: March 17 - March 23, 2024
Deliverables (due March 21):

Milestone 1: Foundation Environment Setup & Model Loading

During the first week, we will establish a containerized development environment using Docker/Podman. The multi-stage build process will optimize image size, leveraging lightweight base images like Alpine Linux or Debian Slim. Core work involves Ollama runtime integration with dual support for CPU and GPU acceleration modes, particularly addressing CUDA/cuDNN compatibility for NVIDIA GPUs. A dynamic model weight loading mechanism will be implemented through volume mounts, enabling flexible model switching. Environment variable injection will facilitate runtime configuration, while liveness/readiness probes ensure container stability.

Deliverables include a multi-architecture (x86/ARM) Dockerfile incorporating security best practices like non-root user contexts. The accompanying build script features model pre-download capabilities. Test documentation will capture cold/warm start times and GPU VRAM utilization metrics. The basic manual covers Ollama model registry setup and tag-based model switching protocols.

Milestone 2: **LangChain Integration & Pipeline Construction**
Schedule: March 17 - March 29, 2024 (2 weeks)
Deliverables (due March 29):

Week two focuses on deep LangChain framework integration using LCEL (LangChain Expression Language) for chain orchestration. The system employs Jinja2 templating for dynamic prompt generation and Redis/memory-based conversation memory management. Structured output parsing utilizes Pydantic models, with Celery-powered asynchronous task queues enhancing concurrency. Modular agent design ensures future extensibility.

The updated container image pre-loads LangChain core and community packages with prebuilt chain templates (QA Chain, Summarization Chain). Technical documentation details the retrieval-augmented generation pipeline and fallback mechanisms. Example implementations include entity-aware multi-turn dialogue systems, with test cases validating end-to-end latency and output consistency under load.

Milestone 3: **User Interface Development**
Schedule: April 1 - April 19, 2024 (3 weeks)
Deliverables (due April 19):

Week three delivers a production-grade interface using Gradio's Blocks API for custom layouts. The responsive design employs CSS Grid/Flexbox with WebSocket-based real-time communication. Session persistence leverages browser LocalStorage, while integrated Markdown/KaTeX rendering supports technical content. Prometheus metrics enable performance monitoring.

The container auto-launches a Gradio service on port 7860 with dark/light mode toggles. Core UI components include streaming response chat windows and LLM parameter debugging panels. Architecture documentation specifies message protocols and includes WCAG compliance checklists. Load testing data from Locust/k6 provides percentile latency breakdowns.

Milestone 4: **Deployment & Documentation Finalization**
Schedule: April 22 - May 11, 2024 (3 weeks)
Deliverables (due May 11):

Week four implements production deployment through Infrastructure as Code (Ansible/Terraform), with Kubernetes support via optimized Helm Charts. The ELK stack

handles log aggregation while OpenTelemetry enables distributed tracing. Semantic versioning governs image tagging, complemented by auto-generated Swagger UI documentation.

Deliverables include both standalone Docker Compose files and cluster deployment packages using Distroless images for enhanced security. User manuals feature visual troubleshooting guides and hardware sizing recommendations. Developer documentation details plugin APIs and fine-tuning integration points.

Milestone 5(Optional): **Knowledge Base Expansion**

Schedule: May 12 - June 1, 2024 (3 weeks)

Deliverables (due June 1):

The optional fifth week builds hybrid retrieval systems combining BM25 algorithms with vector similarity scoring. The document processing pipeline incorporates semantic chunking and LLM-powered query rewriting. Optimized HNSW algorithms support multiple vector backends (FAISS/Chroma).

Specialized images include multi-format document parsers, with technical documentation covering incremental indexing strategies. Performance guidelines address embedding cache warm-up procedures and precision-recall balanced threshold tuning frameworks.

IV. Task

1. Set up a unified containerized development environment
   A. Write a custom Dockerfile to build a base image with ollama, and unify installation paths, model cache location, and entry point
   B. Configure default ENTRYPOINT to enter usable model environment upon container startup
   C. Download and preload target models (e.g., llama3, mistral), and verify model files can load and run in container
   D. Create unified image build and run documentation to help teammates quickly reproduce environment for further development
2. Basic model invocation tests
   A. Run ollama run in container to interact with LLM, verifying prompt input yields correct output
   B. Record model response time, CPU/memory usage, output structure, etc., to evaluate in-container performance

C. (Optional) Wrap common test processes into shell scripts or CLI tools to simplify testing and demo procedures

3. Build Langchain base flow
   A. Install and configure langchain in the image, connecting to local ollama model as LLM interface
   B. Build minimal functionality chain (e.g., LLMChain): input prompt → processed by langchain → output response
   C. Verify stability and accuracy of each step in the chain, debug interface between langchain and model
   D. Design prompt templates and parameter structures for later expansion to various scenarios (writing, Q&A, summarization)
4. Integrate Gradio frontend
   A. Create a local web frontend using gradio in the container, with input box, output area, and submit button
   B. Connect gradio interface with langchain module to allow user interaction with backend model via web page
   C. Configure the container to auto-run gradio service on startup and listen to designated port for accessible local UI
   D. Test multi-turn conversation flow, abnormal input handling, and interaction experience to ensure stable connection
5. Function extension and module optimization
   A. Add optional modules such as local knowledge base QA (load PDF/text, build RAG document chain)
   B. Introduce system prompt control and role setting to enhance targeted responses
   C. Implement basic dialogue history mechanism to support context management or memory
   D. Modularize image structure: deploy model service (ollama), logic controller (langchain), and frontend (gradio) separately for better maintenance and replacement
6. Optimization and delivery preparation
   A. Optimize overall container platform for performance, reduce resource use, and evaluate performance across devices
   B. Write complete technical documentation including image build steps, config file explanations, module interfaces, troubleshooting guide
   C. Prepare final demo materials (slides, video, demo scripts) to showcase platform functions and development process

D. Propose future development ideas, such as ASR, TTS, mobile compatibility, remote image publishing

V. Responsibilities

A. Tech Lead
Coordinates overall technical architecture and task progress, responsible for writing and maintaining Dockerfile, building custom container image with ollama, langchain, and gradio. Coordinates interface specifications among modules and solves key integration issues.
— Zhuiqi Lin (u7733924)

B. Model Deployment and Performance Testing Engineer
Deploys and validates ollama model service in the container, tests CLI interaction, records model response time, resource usage, stability, and provides performance reference for later modules.
— Boyang Zhang (u7760642)

C. Langchain Engineer
Sets up langchain module in container, connects to local model, builds minimum chain (e.g., LLMChain), and designs reusable prompt templates, ensuring clear response logic and stable interfaces.
— Qingchuan Rui (u7776331)

D. System Integration and Logic Control Engineer
Connects langchain backend and gradio frontend, ensures user input reaches the model and response returns to UI, manages internal services, port mapping, and module communication in the container.
— Xiangyu Tan (u7779491)

E. Frontend Interaction and UI Engineer
Designs and implements local web UI using gradio in container, builds input/output area and buttons, optimizes interaction experience, and participates in frontend-backend communication debugging and style configuration.
— Diming Xu (u7705332)

F. Extension Function and Future Planning Engineer
Researches possible platform extensions (e.g., local knowledge base, ASR/TTS,

mobile deployment), implements optional prototype modules, proposes future development directions, and writes reflection and improvement suggestions.
— Dongze Yu (u7775416)

VI. Requirements and Rules

VII. Define Success

System success criteria include:

- The custom container image can be built and run locally, compatible across platforms (Linux/macOS/Windows)
- After container startup, model service (ollama), application framework (langchain), and frontend (gradio) auto-load, without manual configuration
- Full human-AI interaction: users input via Gradio and model returns stable and reasonable responses
- Model response time is within 5 seconds, system resource use is efficient, and interaction experience is smooth
- Frontend loads and interacts correctly, with no errors or lag, providing good usability
- Platform supports at least one basic function extension (e.g., PDF QA), proving modular capability
- Complete user and deployment documentation available, enabling others to develop custom AI apps using the image

VIII. Closure

IX. Constraint

X. Communication and Resources

XI. Wish List

1. Support for voice input and output (voice-based AI chatbot)
   Description:
   Add local ASR and TTS capabilities on top of current text-based interaction, so users can talk to the AI with voice and receive spoken responses
   Challenges:

- Integrating local voice models (like Whisper, Coqui TTS) requires high hardware resources
- Audio processing involves latency, noise reduction, and clarity tuning for good experience
- Speech-to-text errors may affect response accuracy

2. Access to local document knowledge base (RAG)
   Description:
   By accessing local PDFs, Word, or Markdown files and using vector retrieval mechanisms, support document-based QA, enhancing performance in domain-specific knowledge
   Challenges:

- Document parsing, chunking, and embedding are complex; tools like langchain.document_loaders must be chosen carefully
- Vector databases (e.g., FAISS, Chroma) require proper local storage and tuning
- Need to design reasonable question-document matching to avoid wrong answers

3. Modular parameter configuration panel
   Description:
   Extend Gradio UI to include parameter controls like model selection, temperature, system prompts, for different task needs (writing, summarizing, QA)
   Challenges:

- UI should remain clear and simple to avoid user confusion
- Model switching involves cache management and state sync, which is technically difficult
- Parameters vary across models, requiring unified backend handling

4. Multi-language support and language switching
   Description:
   Allow users to input in different languages (Chinese, English, Japanese, etc.) and set target language for model responses to expand global use
   Challenges:

- Different models vary in multi-language support, affecting answer quality

- Target language must be explicitly set in prompt to avoid mixed-language output
- If voice module is integrated, TTS must also support multiple languages

5. Custom knowledge embedding ("personal AI" mode)
   Description:
   Users can upload personal data (chat logs, profiles, project materials) into the platform as private knowledge base for personalized QA
   Challenges:

- Embedded data must have persistent storage with update/delete support
- Private data requires strict protection, especially in multi-user settings
- System must design prompt or chain control to prioritize private knowledge without conflicting with base model output