Agentic Chatbot Project

Developer Guide & Build/Test Documentation

Group Project Submission  
MSAI 631 – Human-Computer Interaction

# Developer Documentation

Developer Documentation

\*\*Audience:\*\* Team members and new contributors. This document not only lists directories and endpoints, but also explains \*why\* each exists, and how to extend or test.

The Agentic Chatbot project is built as a modular experiment in conversational AI. It combines lightweight rule-based responses, cloud-provider AI services, and research-oriented retrieval features. This doc gives context for both course deliverables and real-world maintainability.

The system is structured around three main flows:

1. \*\*Anonymous mode\*\* – deterministic, rule-based, safe for local demos.

2. \*\*Logged-in mode\*\* – provider-backed, supports Azure, Hugging Face, OpenAI, etc.

3. \*\*Agentic demo\*\* – lightweight FastAPI proof of concept that integrates the `ChatBot` wrapper.

These flows share common guardrails, memory management, and UI components. This prevents duplication and makes testing easier.

- `app/` hosts the AIOHTTP web app. Routes like `/plain-chat` are meant for quick, anonymous tests, while `/api/messages` provides Bot Framework compatibility.

- `anon\_bot/` contains the rules (`reverse`, `echo`, `help`, and greetings). This path proves you can build minimal logic without relying on external APIs.

- `logged\_in\_bot/` adds advanced features, including Azure sentiment analysis. This is the bridge to provider-backed AI.

- `agenticcore/` is the abstraction layer. Here, the provider selection happens automatically based on available API keys.

- `memory/` supports session management and a retrieval-augmented generation prototype. This is extensible for experiments with datasets.

- `guardrails/` enforces privacy and safety. For example, PII redaction runs before text is passed to any provider.

- `nlu/` contains intent routing and prompt templates. This is where new intents can be added.

- \*\*`/healthz`\*\*: Always returns `{"status":"ok"}`. This is useful for Docker health checks or CI pipelines.

- \*\*`/plain-chat`\*\*: Takes `{text: ...}` JSON. If `reverse` is included, it returns the reversed string. If `help` is included, it shows available commands.

- \*\*`/api/messages`\*\*: Compliant with the Microsoft Bot Framework activity schema.

- \*\*`/agentic`\*\*: FastAPI route; demonstrates cloud integration without the AIOHTTP layer.

All API keys are externalized. You can choose your provider with `AI\_PROVIDER`, but if it is not set, the system tries to detect which keys exist and falls back to offline sentiment. This ensures that local builds always work.

Anonymous mode never leaves the machine. Logged-in mode always applies guardrails first. Logging is structured but intentionally avoids raw message dumps to prevent accidental leaks.

The CLI is designed to probe providers (`status`) and send test queries (`agentic`). It is both a dev tool and a debugging aid for key misconfiguration.

- To add a new intent: extend `nlu/router.py` and add corresponding rules or provider calls.

- To add a new provider: register it in `providers\_unified.py`.

- To extend memory: expand `memory/rag/` with new indexers or retrievers.

Tests are not just unit checks—they simulate real user flows:

- `test\_anon\_bot.py`: Ensures reverse/help/greet/echo work.

- `test\_routes.py`: Ensures routes respond correctly.

- `test\_memory.py`: Confirms session persistence.

This makes grading and regression tracking straightforward.

# Build and Test Documentation

Python 3.10 or later is recommended. A virtual environment is strongly advised.

Steps:

1. Clone the repository.

2. Run `python -m venv .venv` and activate it.

3. Install dependencies with `pip install -r requirements.txt`.

Two main paths exist:

- \*\*AIOHTTP backend\*\*: Run `python app/app.py`. This brings up the routes on `http://127.0.0.1:3978`.

- \*\*Gradio UI\*\*: Set `APP\_MODE=gradio` before running the same command. This launches a full Blocks interface.

- \*\*FastAPI demo\*\*: Run `uvicorn agenticcore.web\_agentic:app --reload --port 8000`. Use `http://127.0.0.1:8000/agentic?msg=hello` to test.

Automated tests are included:

- Run all: `pytest -q`

- Focus on anon bot: `pytest -q tests/test\_anon\_bot.py`

- Focus on routes: `pytest -q tests/test\_routes.py`

Manual tests:

- `curl` to `/healthz` → expect `{"status":"ok"}`.

- `curl` to `/plain-chat` with `{text:"reverse hello"}` → expect `{reply:"olleh"}`.

- Hugging Face: set `HF\_API\_KEY` and `HF\_MODEL\_SENTIMENT`.

- Azure: set `MICROSOFT\_AI\_SERVICE\_ENDPOINT` and `MICROSOFT\_AI\_API\_KEY`.

- OpenAI: set `OPENAI\_API\_KEY`.

- Cohere, DeepAI also supported.

Without any of these, the system runs in offline sentiment mode.

- If `/plain-chat` returns errors, confirm `Content-Type: application/json` is in headers.

- If Gradio UI fails, ensure `APP\_MODE=gradio` is exported.

- If provider calls fail, run the CLI with `status` to confirm API keys.

Scripts exist (planned or in-progress) to generate README, Quickstart DOCX, Design DOCX, and Results DOCX. This keeps academic deliverables in sync with code changes.

With the build, test, and documentation automation in place, the team can focus on iterating the chatbot logic and meeting course deliverables. The skeleton is fully operational—further work should target improved NLU, more provider integrations, and polished UI/UX.