Two AI conversation Simulator SDK: Design Report

Sreenivasu Akella

July 30, 2025

1 Thought Process and Reasoning

The assignment was to build a flexible and observable SDK to simulate two-agent conversations with text and audio output. I followed a phased design approach:

- **Initial Prototype with CrewAI:** I began prototyping using CrewAI—its role/task abstractions made scripted flows easy to define. However, it lacked mechanisms for managing agent memory, dynamic turn orchestration, and extensible evaluation hooks.
- **Evaluating LangGraph:** I switched to LangGraph for its DAG-based orchestration, native support for memory nodes, clear traceability, and strong alignment with LangChain workflows. This enabled fine-grained agent behaviors, persona conditioning, and cleaner extension for future metrics and dashboards.
- **Modularization :** I abstracted agents, TTS providers, configuration, output, and observers into separate modules, enabling swapping or extension with minimal changes.
- **CLI-First Development:** I prioritized an SDK/CLI-first interface for simplicity, repeatability, and easy scripting, while leaving room for future dashboard/UI integration.
- **Cross-Platform Support:** The SDK is designed to work on both Windows and Linux/macOS environments, with clear instructions for each.

2 Architecture Overview

The **Two AI Conversation Simulator SDK** is a modular Python framework for simulating two-agent conversations, supporting both scripted and LLM-powered unscripted dialogue. The SDK is designed for extensibility, observability, and reproducibility, with clear separation of concerns across its components.

- **Core Orchestration (`AgentSimulator`):** The main controller class (in `agentic_sdk/agent.py`) manages the conversation loop, agent turn-taking, state updates, and output generation. It exposes a standard SDK lifecycle: `init` → `configure` → `run` → `observe`. It loads configuration, initializes agents, manages observers, and coordinates the overall simulation.
- **Agent Nodes (`agent_a_node`, `agent_b_node` in `utils/nodes.py`):** Each agent node is responsible for generating a response based on the current conversation state. In unscripted mode, nodes use LLMs (e.g., OpenAI GPT models via LangChain) to generate context-aware responses, conditioned on persona, tone, and previous messages. Scripted mode (if implemented) uses predefined utterances. Each node also handles emotion detection and evaluation integration.

- **State Management (`state.py`):** The `ConversationState` object tracks the full message history, current turn, speaker, and configuration. It is passed between nodes to maintain context and enable memory throughout the conversation.

- **Text-to-Speech (TTS) Integration (`tts.py`):** After each message is generated, it is converted to audio using a pluggable TTS backend (gTTS, Coqui-TTS, or ElevenLabs), producing both per-turn and full-conversation audio files. The TTS module is designed for easy extension to new providers.

- **Observability and Logging (`observer.py`, `logger.py`):** The SDK uses an observer pattern and structured logging. Observers can be registered to receive real-time events (e.g., message generated, emotion detected, evaluation completed), enabling integration with dashboards or analytics. All key events are logged for traceability.

- **Evaluation Hooks (`utils/nodes.py`):** Each agent response can be evaluated for tone, coherence, and resolution using the FutureAGI SDK. Results are logged and can be used for downstream analysis or quality monitoring. Evaluation is optional and non-blocking.

- **Configuration Layer:** YAML files (in `examples/`) define agent personas, conversation context, number of turns, tone, TTS provider, and evaluation settings. This enables reproducible experiments and easy scenario switching.

- **Output Assets:**
  - **Transcripts:** Saved as both plain text and JSON, capturing all turns and metadata.
  - **Audio Files:** Generated for each turn and for the full conversation.
  - **Logs:** Structured logs record all key events, agent actions, and evaluation results.

- **Packaging and Testing:** The project uses `pyproject.toml` for packaging and dependency management, and supports standard Python testing tools for maintainability. The structure is compatible with PyPI publication.I included run_unscripted.py and run_scripted.py for test cases in example folder.

Execution Flow:

1. User initializes `AgentSimulator` and loads configuration (YAML or dict).
2. Conversation alternates between agent nodes, each generating a response based on the current state.
3. Each response is cleaned, emotion is detected, and evaluated via FutureAGI.
4. Audio is synthesized for each message.
5. Observers/loggers capture all events.
6. Outputs (transcripts, audio, logs, evaluations) are saved after each turn and at session end.

Extensibility and Modularity:

- New agent types, TTS providers, or evaluation backends can be added by implementing the respective interfaces.
- Observers can be registered for custom event handling (e.g., live dashboards, metrics collection).

- Configuration is decoupled from code, supporting reproducible experiments and easy scenario switching.

This architecture ensures modularity, extensibility, and observability, supporting both research and production use cases for

3 Key Design Decisions & Trade-offs

- **CrewAI vs. LangGraph:** CrewAI was fast to prototype but limited for memory and extensibility—LangGraph offered richer agent orchestration and traceability.
- **Scripted and Unscripted Support:** Enables reproducible tests via scripted mode, while unscripted mode adds realism; at cost of complexity and API dependency.
- **Observer Pattern:** Introduced minor runtime overhead, but critical for logging and enabling integration with dashboards or analytics.
- **Multi-TTS Backend Support:** Offers free (gTTS), local (Coqui-TTS), and high-quality (ElevenLabs) options, balancing cost, control, and voice naturalness.
- **Evaluation Integration:** Optional FutureAGI evaluation provides advanced metrics, but requires additional API keys and may introduce latency.
- **YAML over Python Config:** YAML was chosen for clarity, readability, and ease of CI/CD, despite reduced dynamism.

4 4. Alternative Approaches Considered

- **Web UI-First Design:** Not implemented initially to stay aligned with SDK-focused instructions and to speed iteration.
- **Flat-Text Only Outputs:** Initially considered minimal output; final design includes JSON and audio to support richer demos and analysis.
- **Direct API Calls vs. Orchestration Library:** Although direct OpenAI calls were simpler, LangGraph/LangChain provides modular orchestration, retries, agent memory, and better maintainability.
- **Single-Mode Only:** Avoided to maintain both deterministic testing and real-world conversation capability.

5 Known Limitations & Next Steps

- **LLM/API Dependency:** Unscripted mode requires a valid OpenAI API key; incurs latency, cost, and rate limits.
- **Audio Quality Variance:** TTS voices differ—ElevenLabs yields the most natural output but requires paid access.
- **No Web UI Yet:** Only CLI and SDK interfaces are available; a minimal dashboard is a planned enhancement.
- **Evaluation Latency:** Integrating FutureAGI metrics may slow batch runs due to API round-trip times.

- **Packaging Not Yet Published to PyPI:** While the SDK is package-ready (via `pyproject.toml`), it's not yet published. Sensitive environment keys (e.g. in `.env`) prevent safe public release without secret management refactoring, pending secret management refactoring.
- **Limited Parallel Simulation:** Current implementation runs sequentially; future support for concurrent agent sessions would improve scalability.

Planned Next Steps

- Refactor secret management (OpenAI key, TTS credentials) for secure PyPI publishing.
- Add multilingual and additional TTS voice support.
- Enhance memory/persona continuity using LangGraph memory nodes.
- Provide optional real-time dashboard or web UI for demos and visualization.
- Support concurrent simulations and batch streaming outputs.

6 AI Assistance Used

- **GitHub Copilot:** Assisted with boilerplate code, scaffolding, and routine refactorings.
- **LangChain/LangGraph Documentation & Community:** Regularly consulted for design patterns, agent memory usage, and workflow orchestration.

GitHub Repository: `github.com/SreenivasuAkella/2-Conversational-AIAgents`