■ Cortex-Flow

A Distributed Multi-Agent AI System

Powered by LangChain, LangGraph, and FastAPI

Version:	v1.1 Stable
Date:	October 2025
Status:	Production Ready
License:	MIT

Table of Contents

1. Introduction

Cortex-Flow is a cutting-edge distributed multi-agent AI system that revolutionizes how we build and deploy intelligent workflows. By combining the power of LangChain, LangGraph, and FastAPI, it provides a robust platform for creating sophisticated AI applications.

Key Highlights

- Production-ready distributed architecture
- Template-based workflow system with visual execution
- Specialized AI agents using the ReAct pattern
- Model Context Protocol (MCP) for tool integration
- Custom Python library system for extensibility
- OpenAI-compatible API for seamless integration
- Powerful web interface for workflow management

2. Features Overview

2.1 Multi-Agent System

The multi-agent system is the core of Cortex-Flow, featuring specialized AI agents that work together using the ReAct (Reasoning-Action-Observation) pattern. Each agent has a specific role and expertise, enabling complex task decomposition and parallel execution.

Agent	Role	Capabilities
Supervisor	Orchestrator	Task decomposition, delegation, coordination
Researcher	Information Gathering	Web research, data collection, fact-checking
Analyst	Data Analysis	Pattern recognition, data processing, insights
Writer	Content Generation	Reports, summaries, documentation
Custom	User-Defined	Any specialized task or domain

2.2 Workflow Templates

The workflow template system enables you to define complex AI pipelines using JSON-based configurations. Workflows support conditional routing, parallel execution, and composable sub-workflows, all compiled to native LangGraph for optimal performance.

- JSON-based workflow definitions
- Conditional routing with dynamic branching
- Parallel node execution for performance
- Composable sub-workflows for modularity
- Native LangGraph compilation
- Visual workflow builder interface

2.3 MCP Integration

The Model Context Protocol (MCP) integration allows Cortex-Flow to connect with external tools and services. Any MCP-compliant server can be integrated, providing access to filesystems, databases, APIs, and custom tools.

2.4 Python Libraries System

The new Python library system (v1.1) enables you to integrate any Python functionality into your workflows. Using a simple decorator-based approach, you can expose Python functions as workflow nodes with automatic type validation and security controls.

Built-in Libraries:

Library	Functions	Use Cases
REST API	GET, POST, PUT, DELETE	API integrations, webhooks, data fetching
Filesystem	Read, Write, JSON operations	Data persistence, file processing, logs
Email	Send notifications	Alerts, reports, user communication
Database	Query, Insert, Update	Data storage, analytics, CRUD operations

3. System Architecture

Cortex-Flow follows a microservices architecture where each component operates independently but works together seamlessly. This distributed design ensures scalability, resilience, and flexibility.

Architecture Layers

Application Layer:

Web UI (React), CLI Tools, REST API, WebSocket connections

Agent Layer:

Supervisor, Researcher, Analyst, Writer, Custom Agents

Core Engine:

LangGraph compiler, Workflow engine, MCP client, Library executor

Infrastructure:

FastAPI servers, Redis/PostgreSQL, Docker, Kubernetes

Key Design Principles

- Microservices architecture for independent scaling
- Event-driven communication between agents
- Stateless design with external state persistence
- Container-native deployment with Docker/Kubernetes
- Async I/O for optimal performance
- Capability-based security model

4. Getting Started

4.1 Installation

```
# Clone the repository git clone
https://github.com/cortex-flow/cortex-flow.git cd cortex-flow # Create
virtual environment python -m venv .venv source .venv/bin/activate # On
Windows: .venv\Scripts\activate # Install dependencies pip install -r
requirements.txt # Configure environment cp .env.example .env # Edit .env
with your API keys # Start the system python scripts/start_all.py
```

4.2 Quick Examples

Workflow Template Example:

```
{ "name": "research_and_report", "nodes": [ { "id": "research", "agent":
"researcher", "instruction": "Research {topic} trends in 2024" }, { "id":
"analyze", "agent": "analyst", "instruction": "Analyze the research
findings", "depends_on": ["research"] }, { "id": "save_data", "agent":
"library", "library_name": "filesystem", "function_name": "write_json",
"function_params": { "path": "./output/analysis.json", "data":
"{analyze_output}" }, "depends_on": ["analyze"] } ] }
```

Custom Library Example:

```
from libraries.base import library_tool, LibraryResponse @library_tool(
name="send_email", description="Send email notification", parameters={ "to":
{"type": "string", "required": True}, "subject": {"type": "string",
"required": True}, "body": {"type": "string", "required": True} }, timeout=30
) async def send_email(to: str, subject: str, body: str): # Your
implementation here await smtp_client.send(to, subject, body) return
LibraryResponse( success=True, data="Email sent successfully",
metadata={"recipient": to} )
```

9. Security & Best Practices

Security is a core design principle in Cortex-Flow. The system implements multiple layers of security controls to ensure safe execution of AI workflows and protection of sensitive data.

Security Features

Capability-Based Access Control:

Libraries must declare required capabilities (filesystem, network, etc.)

Path Validation:

Filesystem operations restricted to allowed directories

Resource Limits:

CPU, memory, and execution time limits for library functions

Sandboxing:

Optional process isolation for critical operations

Input Validation:

Automatic type checking and sanitization

Secrets Management:

Environment-based configuration with encrypted storage

Best Practices

- Always use environment variables for API keys and secrets
- Implement input validation in custom libraries
- Set appropriate timeouts for long-running operations
- Use rate limiting for resource-intensive functions
- Enable audit logging for security-critical operations

- Regularly update dependencies and security patches
- Follow the principle of least privilege for capabilities
- Test workflows in isolated environments before production

10. Release Notes

v1.1 - October 2025

Python Library Integration System

- Custom Python libraries with decorator-based registration
- Type validation with Pydantic
- Security capabilities system
- Built-in REST API and Filesystem libraries
- Variable substitution from workflow state

v1.0 - October 2025

Multi-Project Configuration

- JSON-based configuration system
- Multi-environment support
- Project isolation
- Secrets separation
- Backward compatibility

v0.4 - September 2025

LangGraph Integration

- Native LangGraph compilation
- Streaming support
- Checkpointing system
- Human-in-the-loop workflows
- Performance optimizations