GitSense: The Autonomous Developer's Co-pilot

GitSense is not just a chatbot. It's an autonomous, multi-agent AI system designed for complex software development tasks. It writes code, reads files, runs tests in a secure sandbox, and *autonomously self-corrects* its own work until the tests pass.

Built on a hybrid cloud/local architecture, it functions both online (leveraging powerful cloud models) and completely offline (using local SLMs).

Core Features

- **Advanced Multi-Agent Team:** Simulates a real software team with specialized agents: a **Router**, an **Architect**, **Coders**, a **QA Reviewer**, and a **Clarifier** to handle ambiguity.
- **Cyclical Self-Correction (Dual-Loop):** Uses **LangGraph** for two nested correction loops:
  1. **Functional Loop:** The `coder_agent` and `test_code_tool` loop until all functional tests pass.
  2. **QA Loop:** The `qa_agent` reviews the *working* code for quality, style, and security, forcing a new loop if standards aren't met.
- **Secure Sandboxed Execution:** A "CI/CD Bot" that uses **Docker Compose** to spin up entire environments (e.g., code + RabbitMQ) and executes tests via the **GitHub Actions API**.
- **Hybrid RAG (Online/Offline):** Seamlessly switches between **Azure AI Search** (cloud) and a local **ChromaDB** (offline) based on internet connectivity.
- **Reliable Tooling (MCP + Instructor):** All agent tools are defined using the **Model Context Protocol (MCP)** standard. All outputs are converted to reliable, structured data (**Instructor**) for predictable logic.

System Architecture

The system is built on a "separation of concerns" model: a central **Orchestrator** (the `master_agent`) manages the flow, while specialized **Agents** (the "brains") and **Tools** (the "hands") execute the tasks.

1. The Orchestrator (`master_agent`)

- **Technology: LangGraph**
- **Role:** This is *not* an LLM. It's a "hardcoded" (reliable) state machine that directs the flow of work. It is the "brain" of the self-correction loops and holds the application state (e.g., `iterations_remaining`, `qa_passes`).

2. The Agents (The "Brains")

The agents act as a coordinated team, managed by the `master_agent` orchestrator.

- **`router_agent`**:
  - **Model:** Local SLM (Ollama `phi-3`).
  - **Role:** The "Triage Lead." Its only job is to analyze the user's prompt and output a structured JSON (via `Instructor`) telling the `master_agent` which agent to call next (e.g., `{"route": "clarifier"}` or `{"route": "architect"}`).
- **`clarifier_agent`**:
  - **Model:** Local SLM (Ollama `phi-3`).
  - **Role:** The "Communicator." If the `router_agent` finds a prompt "ambiguous," this agent takes over. It pauses the workflow and asks the user for clarification (e.g., "Optimize for speed or for cost?").
- **`architect_agent`**:
  - **Model:** Cloud LMM (Azure `GPT-4o` / Claude 3.5 Opus).
  - **Role:** The "Solutions Architect." For complex tasks (e.g., "Build a new API"), this agent does *not* write code. It generates a detailed, step-by-step implementation plan in structured JSON, which the `master_agent` then uses as a checklist.
- **`coder_agent_expert`**:
  - **Model:** Cloud LMM (Azure `GPT-4o`).
  - **Role:** The "Pro Developer." Handles complex logic, multimodal inputs (sees error screenshots), and executes the steps from the `architect_agent`'s plan. This is the agent involved in both correction loops.
- **`coder_agent_low_cost`**:
  - **Model:** Local SLM (Ollama `phi-3` or `Llama-3-8B`).
  - **Role:** The "Offline/Junior Developer." Handles simple tasks or operates when no internet is detected. Uses the local `ChromaDB` for RAG.
- **`qa_agent`**:
  - **Model:** Cloud LMM (Azure `GPT-4o` / Claude 3.5 Sonnet).
  - **Role:** The "QA & Security Reviewer." This agent *only* reviews code that has *already passed all functional tests*. It looks for security flaws, poor documentation, or style issues, and can "reject" the code, forcing another correction loop.
- **`metrics_agent`**:
  - **Model:** Any (e.g., `GPT-3.5-Turbo`).
  - **Role:** The "Analyst." Uses the `get_github_metrics` tool and `Instructor` to return perfectly formatted JSON about a repository's health.

Core Tech Stack

Category	Technology	Purpose
Orchestration	**LangGraph**	The core `master_agent` state machine for cyclical logic.

Category	Technology	Purpose
Tool Standard	Model Context Protocol (MCP)	Defines the "API" for all tools the agents can use.
Models (Local)	Ollama (`Phi-3` , `Llama 3`)	Powers the `router_agent` and `coder_agent_low_cost` .
Models (Cloud)	Azure OpenAI / Amazon Bedrock	Powers the expert agents (Architect, Coder, QA).
RAG (Local)	ChromaDB	Vector store for offline RAG.
RAG (Cloud)	Azure AI Search	Scalable, persistent vector store for online RAG.
Backend	FastAPI	Serves the entire application as a robust API.
Sandboxing	Docker / Docker Compose	Defines the *environment* for code testing (e.g., app + RabbitMQ).
Execution	GitHub Actions API	The *executor* that securely runs the Docker Compose sandbox.
Deployment	Azure Container Apps	Hosts the main FastAPI application.
Security	Azure Key Vault	Securely manages all API keys (Azure, GitHub, etc.).

How it Works: The Full "Team" Workflow

This example shows how all agents collaborate on a complex, ambiguous task.

1. **Ask:** User submits a vague prompt: "My app is slow, make it better."
2. **Route:** `router_agent` (SLM) classifies the task as `"complex"` but `"ambiguous"` .
3. **Clarify:** `master_agent` (LangGraph) sees the "ambiguous" flag and dispatches to `clarifier_agent` . The agent asks the user, "How should I optimize? For database speed or app startup time?"
4. **Update:** The user replies: "Optimize the database login logic."
5. **Re-Route:** The new, clear prompt goes back to the `router_agent` , which classifies it as `"complex_refactor"` .
6. **Plan:** `master_agent` dispatches to `architect_agent` . It returns a 4-step JSON plan (e.g., "1. Add index to User table", "2. Refactor login_service.py", etc.).
7. **Dispatch:** `master_agent` receives the plan and passes **Step 1** to `coder_agent_expert` .
8. **Generate:** `coder_agent_expert` generates the code for Step 1.
9. **Test (Loop 1 - Functional):** `master_agent` calls `test_code_tool` via the GitHub Actions API. The test *fails* (e.g., syntax error).
10. **Correct (Loop 1):** `master_agent` sees the error log, decrements the counter, and sends the task + error log *back* to `coder_agent_expert` .
11. **Succeed (Loop 1):** The agent fixes the bug. The `test_code_tool` now *passes*.
12. **Review (Loop 2 - QA):** `master_agent` now sends the *functionally correct code* to the `qa_agent` .
13. **QA Fail (Loop 2):** `qa_agent` reviews the code and replies: "Test passed, but you left a database password hardcoded in `login_service.py` . Rejecting."
14. **Correct (Loop 2):** `master_agent` sends the code + QA feedback *back* to `coder_agent_expert` .
15. **Succeed (Loop 2):** The `coder_agent_expert` fixes the QA issue (e.g., moves password to an env var). The `qa_agent` reviews it again and *approves*.
16. **Continue Plan:** `master_agent` marks Step 1 as complete and moves to Step 2 of the `architect_agent` 's plan. The entire process (Generate, Test, QA) repeats.
17. **Respond:** Once all steps are complete, the final, validated, and quality-checked code is returned to the user.

Getting Started (Local / Offline Mode)

This setup runs the entire system locally using the `coder_agent_low_cost` and `ChromaDB` .

1. **Clone the repo:**

   ```
   git clone [https://github.com/your-username/gitsense.git](https://github.com/your-username/gitsense.git)
   cd gitsense
   ```

2. **Install dependencies:**

   ```
   pip install -r requirements.txt
   ```

3. **Install & Run Ollama:**
   - Download Ollama
   - Pull the models:

     ```
     ollama pull phi-3
     ollama pull llama-3-8b
     ```

4. **Run the local RAG ingestion:**

    - (You will build a script to "feed" files to ChromaDB here)

    ```
    python ingest_local_rag.py --source /path/to/docs
    ```

5. **Run the application:**

    ```
    uvicorn main:app --host 0.0.0.0 --port 8000
    ```

Deployment (Cloud / Pro Mode)

This deploys the full-power, hybrid application to Microsoft Azure.

1. **Provision Azure Resources:**
    - Create an **Azure OpenAI** service and deploy `gpt-4o`.
    - Create an **Azure AI Search** service for the cloud RAG.
    - Create an **Azure Key Vault** to store your secrets.
2. **Configure GitHub:**
    - Create a GitHub App or Personal Access Token (PAT) with `actions:write` permissions.
    - Store this token in your **Azure Key Vault**.
3. **Deploy the App:**
    - Containerize the FastAPI application using the provided `Dockerfile`.
    - Push the container to Azure Container Registry (ACR).
    - Deploy the container to **Azure Container Apps**.
    - In the Container App's configuration, securely inject the secrets from **Key Vault** as environment variables.

TFG Roadmap & Future Work

This project serves as the foundation (Phase 1) for a TFG focused on high-performance ML systems (Phase 2).

Phase 1: Architectural Foundation (This README)

- [ ] Build `master_agent` (LangGraph) with state management and all agent nodes.
- [ ] Implement all agents (`router`, `clarifier`, `architect`, `coders`, `qa`).
- [ ] Build `test_code_tool` with GitHub Actions API integration.
- [ ] Implement hybrid RAG (`ChromaDB` / `Azure AI Search`).
- [ ] Deploy the full stack to Azure Container Apps.

Phase 2: TFG - Inference Optimization

The goal of the TFG is to **replace the generic `coder_agent_low_cost` with a custom, ultra-optimized inference engine.**

- [ ] **Fine-Tune:** Fine-tune an SLM (e.g., `Phi-3-medium`) on a massive dataset of "code bug -> code fix" examples.
- [ ] **Kernel-Level Optimization:** Identify the key bottleneck in the fine-tuned model (e.g., attention mechanism) and write a **custom GPU kernel using Triton** to accelerate it.
- [ ] **Optimized Serving:** Serve the custom model using a high-performance server like **NVIDIA Triton Inference Server**.
- [ ] **Benchmark:** Conclude the TFG by benchmarking the new, optimized agent against the generic `GPT-4o` agent.
    - **Hypothesis:** The custom agent will be 15x faster and 90% cheaper at 95% of the `GPT-4o`'s accuracy *for this specific task*.

LICENSE

MIT License