Statement of the IGL Lab

🎓 The Refactoring Swarm

Duration	2 to 3Weeks (Single Sprint)
Organisation	Teams of 4 students
Assessment	**100% Automated** (Performance & Data-Driven)

---

1. Scientific Context 🔬

You are participating in an Empirical Software Engineering research experiment.

The goal is not just to produce code, but to design an autonomous agent architecture (LLM) capable of performing software maintenance without human intervention.

Your mission: Building a multi-agent system,**"The Refactoring Swarm"**capable of taking as input a folder containing "badly made" Python code (buggy, undocumented, untested) and delivering as output a clean, functional version validated by tests.

---

3. System Architecture 🏗️

Your system must orchestrate collaboration between at least 3 specialized agents:

1. **The Auditor:**Reads the code, runs static analysis and produces a refactoring plan.
2. **The Fixer:**Reads the plan, modifies the code file by file to correct errors.
3. **The Judge:**Executes unit tests.
   - *If unsuccessful:*It sends the code back to the Corrector with the error logs (Self-Healing Loop).
   - *If successful:*He confirms the end of the mission.

2. Team Organization (4 Roles) 👥

This project requires a variety of skills. Divide the following roles among yourselves from day one. While everyone is responsible for their part, the final code must be integrated into a single Git repository.

1. **L'Orchestrateur (Lead Dev)**🧠
   - Designs the execution graph (via*LangGraph*, *CrewAI*or*AutoGen*).
   - Manages the relay handover logic:*When to switch from the Listener to the Corrector? When to stop the loop?*
   - Responsible for main.py and CLI argument handling.
2. **The Toolsmith** 🛠️
   - Develops the Python functions that the agents call (the internal API).
   - Implements security: prohibits agents from writing outside the "sandbox" folder.
   - Manages the interfaces to the analysis (pylint) and testing (pytest) tools.
3. **The Prompt Engineer** 💬
   - Writes and versions system prompts.
   - Optimize prompts to minimize hallucinations and token cost.
   - Manages the context: ensuring that the agent has access to the relevant code without overloading its memory.
4. **The Quality & Data Manager (Data Officer)**📊
   - **CRITIQUE :**Responsible for telemetry. He ensures that every action of the agents is recorded in the logs/experiment_data.json file according to the imposed schema.
   - Create the internal test dataset to validate that the system works before submission.

In this lab, your code depends on the code of others. If the Tools Engineer changes the name of a function without informing the Prompt Engineer, the agent will stop working. You must communicate constantly. Consider that you are building a single complex machine, not four separate small machines.

---

5. Automated evaluation criteria 🤖

At the end of the 3 weeks, the Correction Bot will clone your repository and launch your system on a**"Hidden Dataset"**(A minimum of 5 instances of buggy code that you have never seen before). The score is calculated automatically according to the following formula:

Dimension	Weight	Criteria verified by the Bot
Performance	**40%**	• Does the final code pass the unit tests? • Has the quality score (Pylint) increased?
Technical Robustness	**30%**	• Does the system run without crashing? • No infinite loop (max 10 iterations)? • Argument respect --target_dir.
Data Quality	**30%**	• Is the experiment_data.json file valid? • Does it contain the history?completeStocks?

6. Recommended approach 📅

- **Step 1: Hello World & Tools**
  - Clean installation (Virtualenv). Run “python check_setup.py” to validate.
  - Creation of tools (file reading/writing, pylint execution).
  - First simple agent that analyzes a file.
- **Step 2: The Feedback Loop**
  - Connection: Listener -> Proofreader.
  - Iteration management: The corrector must be able to retry if the listener is not satisfied.
  - Prompt optimization by the Prompt engineer.
- **Step 3: Testing & Robustness**
  - Added the Test Agent (pytest).
  - Complete validation of the JSON format of the logs.
  - Test on your own "trap files" to verify that the system does not break everything.

Anti-Plagiarism & Integrity Policy ⚠️

This project is subject to strict monitoring to ensure the validity of the research results.

1. **Code Similarity :**All Python code will be analyzed with a plagiarism detection tool.
2. **Prompt Forensics :**Your prompt files will be compared. A high degree of similarity (>70%) between two teams is considered plagiarism.
3. **Git History :**We analyze the commit history. A project submitted with only one commit on the last day will be rejected (Note = 0).
4. **Log Signature :**The logs contain unique signatures (ID/Timestamps).**Copying another group's logs is mathematically detectable and will result in immediate exclusion.**