Mobile_qa_demo video: https://youtube.com/shorts/LY8z5KU5y8c?feature=share

Decision

I will implement the multi-agent system in Python, following a light and framework-optional approach, based mainly on the standard library; minimal external dependencies are allowed, if necessary. The system will orchestrate several specialized agents to plan, execute, and evaluate Android UI tests on an emulator by means of ADB commands and screenshot-based verification.

Rationale for Python

A choice of Python is based on it being seamless with Android tooling using subprocess calls to ADB, LogCat, and file system operations. It will allow for a seamless creation of a multi-agent orchestration level without incurring a heavy framework overhead, keeping this implementation focused on system logic rather than dealing with complexity. Additionally, it is empowered with image processing capabilities if added in the future, making this implementation minimalistic and debuggable

Proposed Architecture

A supervisor serves as a centralized core with which four major agents interact. The supervisor oversees the control loop of testing with which task allocation and shared state information, such as screen information, last actions, results of passing or failing tests, and retrying, are concerned. Furthermore, a test planner will parcel out high-level objectives of a given test, such as a login function check, into a step-by-step plan with which a predicted graphical interface state is concerned. A device executor will execute these steps on an emulator via ADB operations such as touching/interaction, entering texts, swipes, and application launches with which artifacts, such as screenshots, GUI information dump, and a log file, of these operations are involved. Additionally, a verifier will assess each step with a focus on non-negotiable tests based on GUI information, if available, or resorting to an image or AI-powered analysis in supplementary perspectives.

Tooling Approach

ADB is used as a control interface for devices. The information of UI hierarchy is extracted using uiautomator dumps if feasible, with screenshots used as a fallback solution in a universal way for verification and debugging purposes. A structured log directory with steps, screenshots, dumps, and accompanying logcats is produced after each execution of tests.

Tradeoffs and Mitigation

The access for the UI hierarchy might be inconsistent across different apps, so screenshots will always be safely captured. Waits and safe checkpoints among steps can handle time-related flakiness. As a non-deterministic approach, LLM verification will simply be a follow-up action with a clearly-defined threshold on passing or failing.

Conclusion

This approach will allow Part 3 of this challenge to remain feasible in scope, focusing on reliability, debuggability, and clarity of system behavior. No additional complexity such as Node.js tooling will be used needlessly in this design.