# **Technical Research Report: A Modular Testing Framework for Agentic Systems**

Version: 26.0

Date: 18 June 2025

Status: Approved

## **1. Abstract**

This report addresses a critical bottleneck in the development and verification of the multi-agent R&D pipeline: the inherent difficulty of testing a complex, non-deterministic system end-to-end. Relying solely on full-pipeline runs for testing has proven to be inefficient for debugging and insufficient for guaranteeing the reliability of individual agents. This document proposes a shift in methodology towards Modular Testing, a best practice in traditional software engineering adapted for agentic systems. This involves testing each agent (Design, Coding) in isolation using stable, hand-crafted input artifacts, or "mocks." This approach enables rapid, repeatable, and predictable testing, which is a prerequisite for robust development. Furthermore, this framework is the key to making the future Project Manager (PM) agent's task feasible, as it allows the PM to be developed and audited against a series of simple, verifiable unit tasks rather than the chaotic output of the full, generative pipeline.

## **2. The Problem with End-to-End Testing in Agentic Pipelines**

Our current testing methodology involves running the entire Research -> Design -> Code pipeline. While this serves as a valuable final "smoke test," it is a fundamentally flawed approach for iterative development and debugging for two key reasons:

* Error Cascading: The pipeline is a chain of non-deterministic LLM outputs. A minor, un-noticed flaw in the research\_synthesis.json generated by the Research Agent can cause the Design Agent to produce a slightly incorrect design\_synthesis.json. This flawed "work order" is then passed to the Coding Agent, which may execute it perfectly but still produce a nonsensical or buggy final code artifact. It becomes nearly impossible to trace the original source of the error, as the problem "cascades" down the pipeline.
* Non-Reproducibility: Due to the generative nature of LLMs, running the same high-level task twice can produce different intermediate artifacts. This lack of reproducibility makes it extremely difficult to confirm if a bug has been fixed or if a new feature is working correctly. We cannot reliably isolate variables.

## **3. Best Practice: A Modular Testing Framework**

The solution, standard in all professional software development, is to test components in isolation. We will adopt this by creating a "test harness" that allows us to bypass upstream agents and inject a "perfect," hand-crafted JSON artifact directly into a specific agent.

### **3.1. Unit Tests for Agents**

This methodology allows us to create unit tests for each agent. A unit test is designed to verify the smallest possible piece of functionality. Our simplest unit test will be a "Hello, World!" project.

* Unit Test for the Coding Agent: We can create a perfect design\_synthesis.json that instructs the agent to create a single new file, hello.py, with the content print("Hello, World!"). If we inject this file and the agent succeeds, we know its core logic for file creation is sound. If it fails, we know the bug is unequivocally within the Coding Agent itself.
* Unit Test for the Design Agent: Similarly, we can create a research\_synthesis.json that conceptually describes the "Hello, World!" program. We then inject this into the Design Agent and verify that it produces a sensible design\_spec.md and a valid design\_synthesis.json for the Coding Agent.

### **3.2. Building a Library of Test Cases**

This "Hello, World!" test is the foundational block. We can build upon it to create a library of increasingly complex, yet perfectly predictable, test cases:

* A test for modifying an existing file.
* A test for deleting a function.
* A test for handling multiple file modifications in one run.

## **4. Enabling the Project Manager Agent**

The implementation of a modular testing framework is the single most important prerequisite for developing the Master Project Manager (PM) agent. The PM's core task is to observe the outcome of a task and decide what to do next. This is an exercise in auditing.

* Auditing the Full Pipeline is Intractable: Asking the PM agent to look at the complex, multi-file output from a full pipeline run and determine "if it worked" is an incredibly difficult, high-level reasoning task. It would be nearly impossible to train or prompt the PM agent to do this reliably.
* Auditing a Unit Test is Trivial: With our modular framework, the PM's task becomes simple and auditable.
  + Goal: The PM is given the "Hello, World!" design\_synthesis.json as the goal.
  + Observation: The PM is given the coding\_execution\_report.json produced by the Coding Agent.
  + Audit Task: The PM's prompt is simple: "Does the final\_code\_artifacts key in the execution report contain one entry for hello.py with the content print(\"Hello, World!\")?"
  + Decision: This is a simple, verifiable string comparison. The PM can confidently decide "Yes, the task was completed successfully" or "No, the task failed."

By developing the PM against a suite of these simple, modular unit tests, we can build its core decision-making logic on a foundation of certainty. Once it masters auditing these simple tasks, we can gradually increase the complexity, but always from a baseline of predictable, verifiable behavior. This is the only reliable path to creating an autonomous PM that can effectively orchestrate the R&D pipeline.