Detailed Information Flow in the Agentic Test Automation Framework

Alberto Espinosa

August 17, 2025

1 Introduction

This document details the data flow in an agentic test automation framework orchestrated by LangGraph for analyzing, transforming, and executing automated tests for web applications. The system processes existing test code (e.g., Cypress or Playwright scripts) and generates enhanced artifacts like Gherkin features, test plans, Playwright code, execution results, and coverage reports.

- **Structure:** Information flows sequentially through 8 agents via a shared `TestAutomationState` (a TypedDict).
- **Process:** Each agent consumes prior outputs, generates new data, and updates the state.
- **Flow:** Starts from input processing, progresses through agent-based transformations, and ends with artifact generation.
- **User Interaction:** Outputs are visible via console logs and files saved in `output_13/` folders.

2 Overall Flow Structure

- **Pipeline:** Linear flow: Input → Analysis → Agents 1-8 → Outputs.
- **Data Generation:**
  - Parsing: Uses regex and AST for code analysis.
  - LLM: Employs Groq/LangChain for natural language generation (e.g., user stories).
  - Execution: Runs tests via Node.js subprocesses with Playwright/c8.
- **Purpose:** Transforms raw test code into structured, executable, and reportable artifacts.
- **User Visibility:** Console logs show progress; artifacts (e.g., JSON reports, PNG visuals) are saved to folders.

3 Graph

3.1 Pre-Pipeline: Input Folder Processor

- **What It Does:** Scans a specified folder for test code files and reads their content to initiate the pipeline.
- **How It Does It:** Recursively traverses the input folder (e.g., `input_data_1`) using Python's `os` module, identifies supported file extensions (.js, .ts, .py, etc.), and reads file contents with proper encoding handling (UTF-8 with retries for encoding errors).

- **Inputs:** Folder path provided by the user.
- **Outputs:** List of dictionaries per file: {`filename, code_content`}.
- **Role in System:** Acts as the entry point, collecting raw test code for further analysis. Ensures all relevant files are captured to feed into the EnhancedCodeAnalyzer, setting the stage for the agentic workflow.

3.2 Pre-Pipeline: EnhancedCodeAnalyzer

- **What It Does:** Analyzes test code to extract key elements like URLs, test steps, languages, and frameworks.
- **How It Does It:** Uses regex patterns to parse code:
  - Extracts real URLs (e.g., from `cy.visit('https://example.com')`).
  - Parses test steps (e.g., `cy.get(sel).click()` → dict {'type': 'click', 'selector': sel}).
  - Detects languages via file extensions (e.g., .js → JavaScript; supports Python, Java, etc.).
  - Identifies frameworks (e.g., Cypress, Playwright, Selenium, Jest) by matching keywords (`cy.`, `page.goto`) and imports.

  Handles chained commands (e.g., splitting input + assertion) and fixes assertion parsing (e.g., `cy.url().should('eq', url)` → 'assert_url').
- **Inputs:** List of {`filename, code_content`} from Input Folder Processor.
- **Outputs:** `ast_analysis` dict per file: {`real_urls: list, parsed_steps: list of dicts, language, frameworks`}.
- **Role in System:** Provides structured data as the foundation for all agents. Ensures accurate extraction of real URLs and steps, enabling subsequent generation of artifacts like Gherkin features and Playwright code.

3.3 Agent 1: Code Analysis

- **What It Does:** Refines the initial code analysis to improve accuracy of extracted data.
- **How It Does It:** Processes `ast_analysis` to:
  - Enhance URL extraction (deduplicates, validates format).
  - Refine step parsing (e.g., corrects chained assertions like `cy.get().should('have.value')`).
  - Add metrics (e.g., step count, assertion types).

  Uses rule-based logic to ensure consistency (e.g., standardizing selector formats).
- **Inputs:** `TestAutomationState` with `original_code, filename, ast_analysis`.
- **Outputs:** Updated `ast_analysis` in state (enhanced dict with URLs, steps, metrics).
- **Role in System:** Acts as a quality gate, ensuring robust data for downstream agents. Its refined analysis directly informs narrative generation (e.g., user stories) and test transformations.

3.4 Agent 2: User Story

- **What It Does:** Generates a human-readable user story describing the test's purpose.
- **How It Does It:**
  - If a user story file is provided, reads it directly.
  - Otherwise, uses LangChain/Groq (Llama3 model) to generate a story via a prompt like "Generate a user story based on test steps: {steps}."
  - Fallback (if API unavailable): Uses a hardcoded template (e.g., "As a [role], I want to... based on {analysis_summary}").
- **Inputs:** `TestAutomationState` with `ast_analysis` (and optional `user_story_file`).
- **Outputs:** `user_story` string (e.g., "As a user, I want to log in to verify credentials").
- **Role in System:** Bridges technical test steps to business-readable narratives, feeding into Gherkin generation for BDD alignment. Ensures tests are contextualized for stakeholders.

3.5 Agent 3: Gherkin

- **What It Does:** Creates a Gherkin feature file for behavior-driven development (BDD).
- **How It Does It:**
  - Maps parsed steps to Gherkin syntax (e.g., `'type': 'click'` → "When I click on {selector}").
  - Uses real URLs from `ast_analysis` in the Background (e.g., "Given I am on {primary_url}").
  - Employs LLM for natural language phrasing or falls back to templates if API fails.
- **Inputs:** `TestAutomationState` with `ast_analysis, user_story`.
- **Outputs:** `gherkin_feature` string (e.g., Feature file content with Feature, Background, Scenario).
- **Role in System:** Translates technical steps into structured BDD specs, enabling collaboration between developers and non-technical stakeholders. Feeds structured scenarios to the test plan agent.

3.6 Agent 4: Test Plan

- **What It Does:** Produces a detailed test plan in Markdown format.
- **How It Does It:**
  - Uses LLM to expand `user_story` and `gherkin_feature` into a structured document (sections: prerequisites, test cases, expected outcomes).
  - Fallback: Template-based plan using parsed steps (e.g., "Test Case: Verify {action} on {selector}").
- **Inputs:** `TestAutomationState` with `user_story, gherkin_feature`.
- **Outputs:** `test_plan` string (Markdown content).
- **Role in System:** Provides a human-readable plan for test execution, guiding the generation of Playwright code and ensuring traceability from requirements to tests.

3.7 Agent 5: Playwright

- **What It Does:** Generates executable Playwright test code in JavaScript.
- **How It Does It:**
  - Maps parsed steps to Playwright syntax (e.g., `'type': 'click'` → `page.locator(sel).click()`).
  - Ensures consistent locators (e.g., `page.locator()`) and fixed assertion handling (e.g., `expect(page.locator()).t`
  - Uses `ast_analysis` for steps and `test_plan` for context.
- **Inputs:** `TestAutomationState` with `ast_analysis, test_plan`.
- **Outputs:** `playwright_code` string (.spec.js file content).
- **Role in System:** Enables migration from other frameworks (e.g., Cypress) to Playwright, producing modern, executable tests for real-time validation.

3.8 Agent 6: Execution

- **What It Does:** Runs the generated Playwright tests and collects results/coverage.
- **How It Does It:**
  - Sets up a Node.js project with Playwright and c8 (V8 coverage tool).
  - Executes tests via Python `subprocess` calls to Node.js, running `npx playwright test`.
  - Collects raw coverage JSON from c8 and execution logs (pass/fail, errors).
  - If Node.js fails (e.g., not installed), marks results as "not collected."
- **Inputs:** `TestAutomationState` with `playwright_code`.
- **Outputs:** `execution_result` dict (status, time, `coverage_collected`: bool, logs).
- **Role in System:** Validates generated tests in real browsers, providing real-world execution data and coverage metrics for quality assessment.

3.9 Agent 7: Coverage

- **What It Does:** Processes execution results to generate coverage reports and visualizations.
- **How It Does It:**
  - Parses c8 coverage JSON to compute percentages (lines, functions, branches).
  - Generates HTML reports and PNG charts using Matplotlib/Seaborn.
  - If no coverage data (e.g., Node.js failure), returns {`"coverage_collected": False, "overall_percentage": 0.0`}.
- **Inputs:** `TestAutomationState` with `execution_result`.
- **Outputs:** `coverage_report` dict (percentages, file paths); `coverage_image_path` (PNG file).
- **Role in System:** Quantifies test effectiveness via coverage metrics, highlighting untested code areas for quality assurance.

3.10 Agent 8: Final Report

- **What It Does:** Compiles a comprehensive summary of all artifacts and saves them to output folders.
- **How It Does It:**
  - Aggregates all state data (user story, Gherkin, test plan, code, results, coverage).
  - Generates a JSON summary with metadata (e.g., timestamp, errors).
  - Saves artifacts to `output_13/` subfolders (e.g., `features/`, `tests/`, `coverage/`).
- **Inputs:** Full `TestAutomationState` (all prior outputs).
- **Outputs:** `final_report` dict (summary); `artifacts` dict (file paths).
- **Role in System:** Concludes the pipeline, delivering all outputs in a structured format for user review and integration into development workflows.

4 Error Handling and Shared State

- **Error Handling:** Exceptions (e.g., LLM or Node.js failures) are caught and appended to the `errors` list in the state. Fallbacks (e.g., templates for LLM, "not collected" for coverage) ensure pipeline continuity.
- **Shared State:** `TestAutomationState` persists data across agents, ensuring traceability and enabling each agent to build on prior outputs.

5 User Interaction and Visibility

- **Execution:** Users run notebook cells sequentially in a Jupyter environment.
- **Outputs:** Console logs display progress (e.g., "Agent X: Generated Y characters"); artifacts are saved to folders (e.g., viewable as .feature, .js, .md, .html, or .png files).
- **Future Enhancement:** A web UI could provide dashboards for easier interaction and visualization.

6 Next Stages, Enhancements, Lacks, and Planning

6.1 Current Strengths

- Robust agentic design: 8 agents in a clean LangGraph pipeline with shared state.
- Fixes implemented: Real URLs, assertion parsing, locator consistency, real coverage.
- Flexibility: Supports multiple languages/frameworks; includes fallbacks.
- Outputs: Generates 18+ files per run (e.g., Gherkin, tests, reports, visuals).

6.2 Weaknesses

- **Scalability:** Notebook-based; lacks parallelism for file processing.
- **Input/Output Variety:** File-based inputs; no API/web UI or database.
- **Advanced AI:** Prompts are basic; no multi-agent collaboration.
- **Testing Depth:** Coverage is JavaScript-focused (V8/c8); limited for non-JS languages.
- **Security/Compliance:** No handling of sensitive data (e.g., API keys in URLs).
- **Monitoring:** Lacks metrics tracking (e.g., LLM costs, execution times).

6.3 Next Stages (Short-Term, 1-3 Months)

1. **Refactor to App:** Convert to Streamlit/Flask web app for user-friendly interface.
2. **Parallel Processing:** Implement multiprocessing for files; async for LLM calls.
3. **Enhance LLM:** Fine-tune prompts; integrate advanced models (e.g., GPT-4) or RAG.
4. **Expand Support:** Add generators for Selenium, Pytest; support non-web tests (API/mobile).
5. **Testing Suite:** Write unit tests for classes/agents; simulate Node.js failures.

6.4 Enhancements (Medium-Term, 3-6 Months)

- **Multi-Language Execution:** Add runners for Python (Pytest), Java (JUnit) with coverage tools.
- **AI Improvements:** Enable collaborative agents (e.g., feedback loops); integrate code interpreters.
- **Visualization/Dashboard:** Build a web dashboard for aggregated reports.
- **Integration:** Add CI/CD hooks (e.g., GitHub Actions); API endpoints.
- **Error Resilience:** Implement auto-retry for executions; ML-based anomaly detection.

6.5 Long-Term Planning (6+ Months)

- **Commercialization:** Open-source (GitHub) or SaaS (e.g., Vercel) with premium LLM pricing.
- **Community/Extensions:** Plugin system for custom agents; user feedback for prompts.
- **Metrics Tracking:** Add analytics (e.g., success rates); benchmark vs. manual testing.
- **Roadmap:** v1.0: Web app. v2.0: Multi-language execution. v3.0: AI optimization (e.g., auto-fix coverage gaps).
- **Team Needs:** 2-3 Python/AI developers, 1 tester, 1 UI designer.