ChatGPT

AI-Driven Testing Platform Architecture & Strategy

High-Level Architecture

- **Platform Overview:** The solution consists of a **Test Orchestrator** service (Python 3.12 + FastAPI) running on AWS ECS. This orchestrator uses LangChain/LangGraph to manage an AI agent that generates and maintains tests. It integrates with Bitbucket, Jira, CI pipelines, and test execution infrastructure. The design is event-driven – for example, a Harness CI trigger on a Bitbucket commit invokes the orchestrator via webhook or API call [1]. This trigger passes context (repo, branch, commit ID, Jira ticket reference, etc.) to the orchestrator.

- **Source Integration:** Upon trigger, the orchestrator uses the Bitbucket API to discover new feature branches or recent commits (especially those tagged with Jira IDs in commit messages). It then calls the Jira API (via a LangChain **Jira tool** or direct integration) to fetch the user story description and acceptance criteria for the referenced Jira ticket. This provides natural language context about the feature/bug fix. *For example, an AI agent can orchestrate an end-to-end cycle from Jira user stories to code and test generation* [2]. By leveraging a powerful LLM, the platform can **translate Jira requirements into detailed test cases** automatically [3], ensuring test coverage aligns with the story intent.

- **Test Generation Agent:** The heart of the system is an AI **Test Generator Agent** (powered by Azure OpenAI models via LangChain/LangGraph). This agent has access to tools and context:

- It retrieves **code diffs** or relevant source files from Bitbucket to see what changed in this commit/branch.
- It reads **Jira ticket details** (user story, bug description, acceptance criteria).

- It may also load any existing test cases for the component (to update or avoid duplication).
  Using this context, the agent prompts an LLM (e.g. GPT-4) to generate or update test code. The agent can handle multiple test types:

    - *End-to-End tests:* e.g. generating **Cucumber** scenarios or **Cypress** test scripts for UI flows. We support both approaches – the agent might output Gherkin scenarios (which have step definitions in our in-house framework) or directly output a Cypress test in JavaScript/TypeScript for React UI built on the Connect library.
    - *Integration/API tests:* e.g. Python PyTest functions or Postman-like collections to test Spring Boot REST APIs (invoked via AWS API Gateway/Lambda). These validate each service endpoint or Lambda function in isolation.
    - *Regression tests:* if a Jira ticket is a bug fix, the agent can generate a regression test to ensure the issue is resolved and stays resolved.

- **Test Execution Environment:** Once tests are generated, a **Test Runner** module executes them in parallel across projects/environments. For **UI E2E tests**, we use containerized **Cypress** runs (or

Selenium if needed for Cucumber) that can run headlessly and in parallel (Cypress supports parallel execution across multiple containers/nodes). Each test container uses a **Functional ID (FID)** credential (securely stored) to log in via the IDAnywhere OIDC flow – for example, the test might perform an OAuth login with the FID username/password or reuse a saved bearer token to simulate an authenticated user in the dev/test environment. For **API tests**, the runner injects a valid OIDC bearer token (obtained via a client credentials flow or a special test client) in the Authorization header of each request. This ensures secure access to protected endpoints using the same auth mechanism as real clients.

- **Parallel & Scalable Execution:** The architecture supports high scalability. The orchestrator can spin up multiple test execution jobs on ECS (or harness can spawn parallel nodes) to run tests concurrently for different projects or branches. Test suites are containerized and isolated to avoid interference. We design tests to be stateless and idempotent (each test sets up and tears down its own data) to allow safe parallel runs. The agent tags tests by project and environment, and uses AWS EventBridge or SQS internally to queue and manage test-run tasks across a fleet of executors. This **parallelization** accelerates feedback without serial bottlenecks.

- **Observability & Reporting:** A **real-time dashboard** (or chat interface like a Slack bot) is connected to the orchestrator. It shows each test run status, progress, and results. The orchestrator streams logs and metrics:

- Important metrics (e.g. number of tests run, passes/failures, execution time, error rates) are sent to **Dynatrace** for monitoring. (For instance, custom Dynatrace events/metrics can track a spike in failures or slower performance in tests, correlating with recent deployments).
- All test logs and platform logs are shipped to **Splunk** for aggregation and deeper analysis. This includes any error stack traces, AI agent debug info, and screenshots from UI test failures.

- We integrate **Phoenix (Arize)** for AI observability. All LLM interactions (prompts and responses) are instrumented with OpenTelemetry and traced via Arize Phoenix [4] [5]. This allows us to monitor the AI agent's performance, detect anomalies in its outputs, and evaluate prompt effectiveness over time. We can use Phoenix's dashboard to replay prompts and compare model versions or prompt templates, ensuring our prompting strategy remains reliable.

- **Result Publishing:** After execution, the platform collates a **detailed report**. This report might be posted as a comment in the relevant Bitbucket pull request or as a summary in the Jira ticket. It includes which tests were run or created, how many passed/failed, failure details (with links to Splunk logs or screenshots), and any **auto-healing actions** performed. If tests were automatically fixed, the agent will push the updated test code to a new branch (or the feature branch) in Bitbucket. For traceability, the commit message or PR will indicate it's an AI-generated test update (including the Jira ID) and may contain a summary of changes (e.g. "Updated test selectors to fix login test (auto-healed by AI agent)"). Development teams can review these changes, merge them, or provide feedback if something looks off.

AI Prompting Strategy

- **Multi-Stage Prompting:** We employ a **chain-of-thought** prompting approach using LangChain/LangGraph. Rather than a single giant prompt, the agent breaks the problem into stages:

- **Understanding the Change:** The agent first summarizes the Jira ticket and code diff. A prompt to a smaller model (e.g. a GPT-3.5-tier model) could be: *"Summarize the user story and code changes in commit XYZ. What features or fixes were introduced? Identify any new endpoints or UI elements."* This focuses the context.
- **Test Plan Proposal:** Next, the agent asks the LLM (preferably GPT-4 for its reasoning strength) to draft a test plan. For example: *"Given the above feature description and changes, list what scenarios should be tested (end-to-end, integration, regression). Include user flows or API calls that need validation."* The model might output something like a list of scenarios (login success, login failure, new API endpoint returns correct data, etc.). This can be reviewed or even shown in the dashboard for transparency.
- **Test Case Generation:** For each scenario, the agent crafts a prompt to generate actual test code. This prompt includes:
    - The scenario description (from the previous step).
    - The preferred framework and format (e.g. "Generate a Cypress test in JavaScript" or "Generate a Cucumber scenario with Gherkin and step definitions in Java").
    - Any coding standards or fixture data required (possibly providing a snippet of existing test code as an example for consistency).

      The LLM (GPT-4) then produces the test code. We include system instructions to ensure the output is executable and does not hallucinate unavailable identifiers. For instance, *"Use the page object IDs from the Connect library when selecting UI elements, and ensure all steps exist. If the test is for an API, use the given base URL and include the bearer token header."*

- **Self-Consistency Checks:** We utilize LangGraph's capabilities to validate and correct the generated code. After initial generation, the agent attempts to parse or compile the test code (e.g., run a syntax check or load the test file in a sandbox). If there are errors (syntax errors, undefined selectors, etc.), those are fed back into the LLM in a follow-up prompt to fix the code. This *reflection loop* continues until the test code passes basic checks or a retry limit is hit. This strategy is informed by recent research showing that iterative self-correction dramatically improves code generation success rates (e.g. boosting execution success from ~55% to ~81% by checking and fixing errors) [6]. In our context, if the first Cypress test generation has a typo or the selector is slightly wrong, the error (or Cypress's log) is provided to the model: *"The test failed because it couldn't find element [#loginBtn]. Suggest a fix."* The model can then adjust the selector or step. This loop makes the output far more reliable than one-shot generation.

- **Model Selection:** We leverage **Azure OpenAI's model suite** in a tiered manner for efficiency:

- **GPT-4 (e.g. GPT 4.1)** is used for the heavy lifting – understanding requirements and generating complex test code – since it offers superior reasoning and code generation quality.
- Smaller or fine-tuned models (like the GPT-4 "mini" or GPT-3.5 variants) are used for simpler tasks such as summarization, extracting keywords (like finding all API endpoints mentioned in a diff), or for quick fixes of minor issues. Using lighter models for these tasks improves throughput and lowers cost, reserving the most powerful model only when needed.

- We also take advantage of **few-shot prompting** where possible. The agent maintains a few **test examples** (maybe template test cases or previous successful generations) and includes them in the prompt for a new similar test to guide the model. For instance, if it's generating a login flow test, we include a short example of a login test format as a guide.

- **Tool Use via LangChain:** Our agent is an augmented LLM that can call tools mid-prompt. For example, using LangChain's Bitbucket tool to fetch a file or diff, or a custom **IDAnywhere Auth tool** that can simulate the login to retrieve a token. The prompting strategy might involve the agent asking itself: *"Do I have the latest DOM or API spec? If not, call the tool to get it."* These tool outputs then become part of the prompt context. By integrating live data (code, DOM snapshots, API schemas), we mitigate the model's knowledge cutoff issues and reduce hallucination. In short, the model **augments its knowledge with real-time data** before generating test steps [7] [8].

- **Prompt Structure & Governance:** We design prompts to be **robust and safe**. Each prompt has a clear instruction section (system prompt defining the role like *"You are an expert test automation engineer."*), and a bounded context (relevant diff hunks, user story text, etc.). We impose output format rules (using markdown for code, or JSON for structured output if needed). We also monitor for hallucinations or unsafe suggestions. If the LLM ever produces code that could be harmful (e.g., deleting data in environment or something not allowed), we have validation rules to catch and prevent execution. Prompts also encourage the LLM to explain its reasoning in comments in the generated test code, which helps maintainers understand why a certain test was created or a certain selector used (and these comments can be reviewed or stripped out later).

- **Iterative Learning:** The prompting strategy isn't static. All LLM interactions are logged and evaluated (via Phoenix Arize and our own analytics). We track cases where the AI's test failed or a human had to intervene, and refine prompts accordingly. Over time, the system "learns" preferred patterns – e.g. if a certain way of phrasing assertions leads to flaky checks, we adjust the prompt to use a better practice. This **prompt management** (with version control and experimentation via Arize's prompt management features [9]) ensures continuous improvement of the AI agent's performance.

Test Generation & Self-Healing Workflow

1. **Detection & Initialization:** When a new code push or pull request is made, Harness CI triggers the AI testing pipeline. The orchestrator service receives the event (including repo URL, branch name, commit ID, and any Jira IDs from commit messages or branch naming convention). It initializes a **Test Run** record and retrieves necessary context (Bitbucket repo metadata, etc.). If multiple projects have pending changes, multiple such runs are queued – the system can handle these concurrently thanks to the stateless, event-driven design.

2. **Context Gathering:** The agent gathers context for the run:

3. Uses Bitbucket API to fetch the **git diff** (code changes) or at least list of modified files.
4. Identifies the **tech stack** of changed components (e.g., a change in a `.java` file indicates backend service change; a `.jsx` or `.tsx` file indicates a UI change). This informs what tests to generate.
5. Fetches the **Jira ticket** details if an ID is present. The user story description and acceptance criteria give natural language requirements that the tests should fulfill (e.g., "Given a valid user, when they login with correct credentials, they see the dashboard" would map to an E2E login test).

6. If available, fetches **existing test cases** related to those files or the same module (for update or reference). For instance, if the `UserService.java` changed, load any `UserServiceTest.java` (unit tests) or API test files to understand current coverage.

7. **Test Suite Generation:** Based on the context, the agent decides which test types are needed:

8. *New Feature (e.g. new REST endpoint added):* Generate **integration tests** for that endpoint (calling the API with various inputs, checking HTTP 200 and response schema) and possibly an E2E test if it's user-facing.
9. *UI Change (e.g. new React component or flow):* Generate **Cypress E2E tests** that cover the new UI functionality in the dev/test environment. This could include locating new elements, clicking through the user journey, and verifying outcomes on screen.
10. *Bug Fix:* Generate a **regression test** (likely integration or E2E) that reproduces the original bug and demonstrates it's now fixed (and remains covered for future).
11. *Impact Analysis:* For regression, the agent might also run a quick impact analysis – e.g., if a common library was changed, propose running the full regression suite for affected projects.

The agent then produces test code for each identified scenario using the prompting strategy described. For **Cucumber-style** tests, it might produce Gherkin scenarios and even implement step definitions (using our in-house Connect UI library for steps). For **Cypress**, it writes a new spec file (or updates an existing one) with `describe/it` blocks for each scenario. The test code is saved (either directly committed to a git branch or staged in a volume) for execution.

1. **Static Validation:** Before execution, the platform performs a quick sanity check on the generated tests:
2. **Linting/Compilation:** Lint the JavaScript (for Cypress) or compile the Java (for Cucumber step definitions) to catch syntax errors or undefined references. Any issues found here trigger the agent's self-correction loop. The error output is given to the LLM to fix the code. *For example, if Cypress test used a misspelled selector* `cy.get('#loginButon')` *and linting catches an unused variable or our custom rule flags it, the agent corrects it to* `#loginButton`. This loop continues until the tests at least pass syntax and lint checks, ensuring we don't attempt to run broken code in CI.

3. **Environment Prep:** The platform ensures the target **test environment** (Dev or QA environment of the app) is up and configured. It loads the FID credentials and either stores the bearer token for API calls or ensures the Cypress test knows how to log in (possibly by injecting a session cookie or by letting the test perform the login steps using the provided credentials). IDAnywhere may require a special auth flow – the platform could use a headless browser to perform the OIDC login upfront and store the auth cookies for reuse by all tests in that run, to avoid repetitive logins.

4. **Parallel Test Execution:** The orchestrator dispatches the tests to run. If multiple test suites (e.g. one set of Cypress tests for UI, one set of integration tests for APIs) are generated, they run in parallel on separate containers in ECS. Within Cypress itself, we can split tests by spec files and use Cypress parallelization (coordinated via the dashboard or via our own simple controller) to utilize multiple CPUs or machines. The agent tags each test with metadata (project, branch, test type) so results can be grouped. Throughout execution, the platform streams logs. A failure in any test is caught by the framework:

5. For UI tests, we capture screenshots or DOM snapshots on failure.

6. For API tests, we capture the response and error messages.

7. **Failure Analysis (AI Self-Healing):** If any tests **fail**, the self-healing agent kicks in immediately. This is a loop where:

8. The failing test's details are gathered: error logs, stack trace, screenshot, DOM HTML around the failed element, etc.
9. These details are fed into a specialized **failure-analysis prompt** to the LLM. The prompt might say: *"Test X failed. Error: Element '#submitBtn' not found. The DOM at failure time is attached. Suggest how to update the test."* For API tests, it might be: *"Expected status 200 but got 500. The request payload was Y. Suggest if the test expectation should change or this indicates a bug."*
10. The LLM analyzes whether it's a test issue (e.g., locator changed, timing issue, assertion mismatch) or an application bug. If it recognizes a known pattern (like "element not found"), it attempts to fix the test: e.g., find an alternative selector from the DOM (maybe the button's text or a different attribute). This realizes the vision of tests that can **"fix themselves" with LLM assistance** [10]. With self-healing enabled, minor app changes (like a button ID change) won't cause permanent test failures – the AI agent will adjust the test to use the new identifier and rerun it, so the pipeline can still pass [11]. This dramatically reduces flaky failures and maintenance toil.

11. If the failure is due to an application bug (e.g. the API really returned 500), the agent may determine it cannot fix that in the test. In that case, it logs it as a genuine defect (possibly even automatically commenting on the Jira ticket or creating a bug ticket). It will not try to "band-aid" the test in that scenario aside from maybe marking it as skipped. The distinction between test issue vs app issue is crucial for trust in results.

12. **Test Auto-Repair:** For test issues, the agent formulates a fix (e.g., update the selector or add a wait, or correct an assertion to match new requirements). The orchestrator then **applies the code change** to the test files (this can be done by directly modifying the in-memory code or by generating a patch and applying it). After fixing, the specific test (or the relevant subset) is run again on-the-fly. If it passes, the orchestrator notes that the test was "self-healed." If it still fails or a new issue arises, the loop can repeat a couple of times. We configure a limit (say 2-3 attempts) to avoid endless loops. This iterative fix approach is informed by the LangGraph reflection technique – capturing errors and re-prompting the model to improve the solution incrementally [12]. It yields far better stability than one-shot attempts.

13. **Reporting & Feedback:** Once all tests either pass or irrecoverable failures are identified, the platform finalizes the run. It compiles a **report**:

14. Summary of total tests run, pass/fail counts.
15. Any tests that were auto-healed (with notes like "LoginTest: selector updated to .btn-primary as ID changed").
16. Any unresolved failures (potential bugs) with error details.
17. Links to logs in Splunk, performance metrics in Dynatrace (e.g., perhaps the duration of certain key flows, for performance regression tracking).
18. LLM agent stats (like how many prompts were used, success of auto-fixes) can also be included for transparency.

This report is delivered via multiple channels: updated in a web dashboard, possibly posted to a team chat (e.g., a Slack message with the high-level results), and as a comment on the Bitbucket PR. The idea is that developers get immediate feedback within their workflow. If a test was auto-fixed, they can review that

commit on the branch. The new test code will already be in the repo (committed by the service under the functional ID or a bot account), so if the PR is merged, the tests go into the main branch automatically. This closed-loop means tests stay up-to-date with code changes continuously.

1. **Continuous Learning:** Finally, all data from the run (prompts, model outputs, what fixes were made, which prompts succeeded/failed) is logged for analysis. Using Phoenix (Arize) and our own databases, we evaluate where the AI might need improvement – for example, if a certain kind of UI change repeatedly fooled the agent, we can augment its prompt or toolset (maybe adding a tool to query the UI's React component tree, etc.). We also monitor **trends**: perhaps over time fewer tests are failing due to self-healing, indicating more stability. All this feeds into improving the **reliability** of the platform over time.

Reliability, Scalability, and Maintainability Best Practices

- **Robust Error Handling:** Reliability is paramount since this platform will be autonomous. We implement thorough error handling at each stage of the workflow. If the LLM fails to produce valid output (e.g., times out or returns gibberish), the orchestrator has fallback logic – it can retry the prompt with adjusted parameters or switch to a backup model. All critical actions (Git operations, test execution commands) are wrapped with retries and timeouts to avoid stalling the pipeline. Additionally, any changes the AI makes to code are verified (through tests as described and sometimes via code review by an AI or optionally a human) before being trusted. This prevents the AI from introducing bad tests that always pass (false positives) or from not actually testing the requirement.

- **Iterative Verification:** Taking inspiration from research on self-correcting code agents, we treat test generation as an iterative verify-and-refine process rather than one-shot. After initial generation, the code is validated by running it in a safe context. If issues are found, the agent reflects and corrects. This significantly boosts reliability of the final test code [6] [12]. We also use the technique of generating **expected output** descriptions and having the model double-check its own work. For example, after writing a test, we might ask the model "What is this test verifying? Does it align with the acceptance criteria?" to catch any mismatches.

- **Isolation and Security:** Each test run occurs in an isolated environment (Docker containers per test job) to avoid cross-contamination or security risks. The FID credentials and tokens are managed via a secure vault and only injected at runtime. No credentials are hard-coded in test scripts – the platform provides them as environment variables or through secure storage that tests can access. We strictly limit the AI's ability to execute arbitrary code: the agent can suggest test code, but execution happens in a controlled sandbox. This prevents any malicious or accidental destructive commands from harming systems. The OIDC login is performed against test environments only, and all endpoints under test are non-production.

- **Scalability:** The use of AWS ECS and stateless services means we can scale horizontally. If test volume grows, we can run multiple orchestrator instances behind a load balancer and scale out runners to handle parallel suites. We design the test generation to be **asynchronous** – e.g., if multiple commits come in, the agent can queue multiple prompt tasks. The Azure OpenAI service is leveraged with rate limits in mind; we might implement a throttling or priority system if many prompts are needed at once. Caching is applied for redundant steps: if two commits in different

branches affect the same code in similar ways, the platform can recognize a similar test was recently generated and reuse or lightly tweak it instead of asking the LLM from scratch (saving time and tokens).

- **Model Monitoring & Improvement:** Using Arize Phoenix, we continuously monitor the LLM's output quality (coherence, correctness) and prompt effectiveness. We set up evaluation metrics for the LLM in this domain – e.g., **precision of test fixes** (did the AI fix actually resolve the failure?) and **recall of test generation** (did it cover all acceptance criteria?). This monitoring allows us to spot drifts or degradation in the AI behavior. For example, if a model update starts giving more hallucinated selectors, we catch that via failing tests and can roll back or adjust prompts. Phoenix's tracing [4] and evaluation tools help ensure we deploy the AI features responsibly and with confidence.

- **Observability & Alerting:** We treat the testing platform like a production service. Dynatrace monitors the health of the platform (CPU/memory of the containers, response times of the FastAPI endpoints, etc.), and also the performance of the systems under test during the run. We configure alerts for abnormal conditions: e.g., if a normally stable test suddenly fails frequently (possible regression in the app), or if the self-healing mechanism had to intervene too often (maybe indicating underlying instability in either tests or application). Splunk log analysis can detect patterns like "Login failed due to 500 error" appearing across many tests, prompting further investigation. These tools ensure the platform's reliability and the reliability of the applications it's testing.

- **Maintainability & Modularity:** The system is designed in modular components – the AI agent logic (prompt templates, tool definitions) is separate from the test execution engine and separate from the CI integration. This separation of concerns makes it easier to update one part without breaking others. For example, we can upgrade the test runner to a new version of Cypress or add a new test framework (maybe Playwright) without changing how the AI decides what to test. All prompts and agent decision logic are stored in a version-controlled repository so we can track changes and roll back if a new prompt version underperforms. We also incorporate **best coding practices** for the generated tests: the AI is instructed to produce clean, readable code following our internal guidelines (comments, meaningful names, no hard-coded waits, etc.), making the tests maintainable by human engineers in the long run.

- **Knowledge Updates:** To keep the AI up-to-date with our evolving tech stack and application changes, we regularly feed it new information. We can fine-tune smaller models on our internal code base or provide the latest API documentation as context during generation. This mitigates the risk of the model suggesting outdated functions or libraries [13] [8]. For example, if our Connect UI library updates with new components, we update the agent's context or training so it knows how to handle them. The platform could even periodically re-run older tests with the AI to see if it suggests improvements (kind of self-refactoring of test code as the app evolves).

- **Scalability of AI Workflows:** We use LangChain/LangGraph to manage complex workflows, which allows conditional branches and parallel calls in the agent's reasoning. For instance, if there are two unrelated modules changed, the agent can spawn two sub-tasks to generate tests for each concurrently. LangGraph's structured approach helps in building these multi-agent or multi-step pipelines reliably. It also simplifies adding new capabilities – e.g., adding an **Accessibility testing agent** later that uses an LLM to write Axe tests for WCAG compliance, or a **Performance test agent**

that crafts JMeter tests based on new endpoints. The architecture can expand to those by adding new agent roles with minimal disruption.

- **Conclusion:** In summary, the platform uses an **AI-driven approach** to achieve comprehensive, continuous testing. By combining **LLM-powered test generation** with traditional automation frameworks (Cucumber/Cypress) and robust DevOps tooling, we ensure high **coverage** and rapid feedback. The inclusion of self-healing AI agents means the test suite adapts instantly to application changes, embodying true continuous testing with minimal manual intervention [10]. Following best practices in reliability (verification loops, monitoring) and scalability (containerization, parallelism, model optimization) guarantees that this system can be trusted and scaled across many projects in the enterprise. It acts as a smart testing co-pilot for the development teams, catching issues early and reducing the maintenance burden of keeping the test suite in sync with the application [11]. The result is a **robust, scalable, and maintainable** AI-driven testing platform that accelerates delivery while upholding quality.

Sources: The approach leverages insights from industry on AI-driven test generation and self-healing. For example, Atlassian reports using LLMs to convert Jira user stories directly into test cases [3]. Autonomous agent frameworks like CrewAI (built on LangChain) have demonstrated orchestrating Jira-to-code-and-tests workflows [2]. Research in AI-assisted testing (Ministry of Testing) shows that integrating LLMs with frameworks (like Playwright or Cypress) allows tests to “fix themselves” and avoid brittle failures [10] [11]. Moreover, techniques such as iterative prompting with error feedback (as described by LangChain’s LangGraph team) significantly improve code generation accuracy [6], which we apply to test script generation and repair. These practices and tools inform the design and ensure our solution is using proven strategies at the cutting edge of AI and software quality engineering.

---

[1] Trigger pipelines using Git events - Harness Developer Hub
https://developer.harness.io/docs/platform/triggers/triggering-pipelines/

[2] Using Crew AI to Automate Code Generation and Test Cases from Jira | by Sushma Bhat | Medium
https://medium.com/@sushmabhat.shimoga/using-crew-ai-to-automate-code-generation-and-test-cases-from-jira-1c1a31dfa27e

[3] AI Test Case Generator | Unlimited for Jira | Atlassian Marketplace
https://marketplace.atlassian.com/apps/1232448/ai-test-case-generator-unlimited-for-jira

[4] [5] [9] GitHub - Arize-ai/phoenix: AI Observability & Evaluation
https://github.com/Arize-ai/phoenix

[6] [12] LangGraph for Code Generation
https://blog.langchain.com/code-execution-with-langgraph/

[7] [8] [10] [11] [13] Creating self-healing automated tests with AI and Playwright | Ministry of Testing
https://www.ministryoftesting.com/articles/creating-self-healing-automated-tests-with-ai-and-playwright