**Documentation: Context-Aware Testing System Using Generative AI**

**Overview**

This document describes the architecture and process for implementing context-aware system testing using Generative AI. The system dynamically generates test scenarios based on contextual inputs to simulate real-world usage and edge cases effectively.

**System Components**

1. Web Form (Context Collector)

* Purpose: Serves as the user interface for gathering essential test context.
* Inputs Collected:
* GitHub Repository URL
* Deployment/Live API URL
* Optional user-defined context (e.g., app domain, expected behaviours, user personas)
* Outcome: Initializes the testing session with relevant environment and application details.

2. File Reader (GitHub Crawler)

* Purpose: Traverses the given GitHub repository and fetches file contents using the GitHub API.
* Functionality:
* Recursively parses all directories and submodules.
* Reads each file (mainly `.js`, `.ts`, `.py`, etc.) and extracts code blocks for analysis.
* Outcome: Prepares the raw code base for functional mapping and analysis.

3. Function Description & API Mapping Engine

* Purpose: Uses Generative AI (Gemini) to understand the structure and behaviour of the codebase.
* Functionality:
* Creates function descriptions for each function as a json which contains different fields(name, description ,parameters).
* Reference:( [Gemini\_function\_calling](https://ai.google.dev/gemini-api/docs/function-calling))
* Identifies and maps API routes (from Express.js, Flask, FastAPI, etc.) to their corresponding function implementations.
* Outcome: Generates an OpenAPI-style map linking endpoints to underlying logic.

4. BDD Scenario Generator

* Purpose: Uses Gemini to create Behavior-Driven Development (BDD) test scenarios from the provided context.
* Functionality:
  + Converts user context + API behavior into Given-When-Then structured test cases.
  + Accounts for edge cases, conditional flows, and user roles.
* Outcome: Produces a comprehensive set of BDD test cases tailored to real-world usage.

5. Function Call Generator

* Purpose: Translates BDD scenarios into executable API function calls.
* Functionality:
  + Converts "When" steps into HTTP requests (GET, POST, etc.) with appropriate headers, payloads, and parameters.
  + Embeds expected outcomes from the "Then" clauses.
* Outcome: A structured test suite ready for automated execution

6. Test Execution Engine (Executioner)

* Purpose: Automatically executes the generated test cases against the live API.
* Functionality:
  + Sends real HTTP requests created by Gemini
* Outcome: Collects actual system behaviour data for comparison.

7. Test Evaluator (Judge)

* Purpose: Evaluates the accuracy of API behaviour against expected outcomes.
* Functionality:
  + Performs pass/fail analysis with optional semantic comparison using LLM.
  + Logs discrepancies, anomalies, and unexpected side effects.
* Outcome: Provides a structured, verdict-based test report with actionable insights.

**Workflow**

1. Context Acquisition

Example:

```json

{

"githubRepo": "PSAK02/test",

"context": "Login",

“deploymentLink”:” https://www.screener.in”

}

```

2. Generation of functional\_description

Example:

{

   "name": "authUser",

    "description": "Authenticates a user or creates a new user if one doesn't exist.",

    "parameters": {

      "type": "object",

      "properties": {

        "email": {

          "type": "string",

          "description": "User's email address"

        },

        "password": {

          "type": "string",

          "description": "User's password"

        }

      },

      "required": [

        "email",

        "password"

      ]

    }

  }

3.BDD Generation:  
Example:

 "BDD": "```gherkin Feature: User Login Scenario: Successful login with valid credentials    Given a user is on the login page    And the user enters a valid username \"testuser\"   And the user enters a valid password \"password123\"    When the user clicks the \"Login\" button   Then the user should be redirected to the home page   And a welcome message should be displayed ``` "

4. API Mapping  
Example:

{

"authUser": {

        "api": "/user/login",

        "request\_type": "post",

        "header": {

            "Content-Type": "application/json"

        }

    }

}

5.Execution & Evaluation

Output is evaluated using rule-based or LLM-assisted validation.

**Sample Test Report**[

{

"passes": 1,

"reason": ""

},

{

"passes": 1,

"reason": ""

},

{

"passes": 0,

"reason": "The test case expects an error message indicating an invalid username and for the user to remain on the login page. However, the response shows a successful authentication with an access token. This indicates the login was successful, contradicting the expected outcome."

},

{

"passes": 0,

"reason": "The test case failed because the expected outcome was an error message indicating an invalid password on the login page. However, the actual response from the authUser API call returned \"{\"email\":\"error\"}\", which does not directly indicate an invalid password. This could represent a different error state, or the front-end may not be properly handling or displaying the specific error reason returned by the API. More information is needed to understand the context of this \"error\" and how it relates to invalid password scenarios."

  }

]

**Benefits:**

* Dynamic adaptability to context changes .
* Improved coverage for edge cases and complex scenarios.
* Human-like reasoning using LLMs to validate real-world conditions.

**Limitations:**

* Requires strict prompt engineering for deterministic results.
* Model hallucination risks—test outputs must be validated.
* May need fine-tuning or system constraints for production testing.

**Tools & Technologies:**

* React js
* Flask
* Python
* Gemini API