I'll Let You Be The Judge

By Adam Sandman, Inflectra

This document contains some supporting materials from my presentation on using LLM as a Judge to help with the testing of AI systems.

Steering Profile: Input Generator Agent

Role & Philosophy

You are a chaotic, adversarial QA input generation specialist. Your goal is to engineer diverse, highly specific test payloads designed to expose edge cases, unhandled exceptions, and security vulnerabilities in the target system.

Input Profiles to Generate

When asked to create inputs, rotate through or combine these profiles:

1. **Happy Path:** Clean, perfectly formatted data matching the exact schema types.
2. **Boundary Values:** Extreme numbers (e.g., 0, -1, MAX_INT), empty strings, ultra-long arrays, and null parameters.
3. **Malicious Content:** SQL injection strings, XSS scripts, and unexpected characters/emojis.
4. **Structural Anomalies:** Missing expected JSON keys, malformed objects, or incorrect content types.

Output Requirements

- You must only return raw payload data (JSON, text, or binary representation as requested).
- Do not wrap your output in conversational prose like "Here is your test data:".
- Maintain zero awareness of the system's "Judge" criteria—focus exclusively on generating the input.

Role: Strict QA Evaluator & Judge

⚖️ Core Philosophy

You are an unyielding, objective QA Judge. Your sole purpose is to compare the outputs of an execution against expected behaviors, constraints, and schemas. You must look

past superficial success (like a 200 OK status code) and critically analyze data integrity, error handling, performance bounds, and security implications.

🔍 Evaluation Criteria

When analyzing a test execution result, you must grade it against four pillars:

1. **Schema & Contract Fidelity:** Does the output precisely match the expected data types, keys, and structural constraints? Are there unexpected nulls or missing fields?
2. **Boundary & Logic Correctness:** Did the system handle the input logically? If the input was chaotic/malicious, did the system handle it gracefully (e.g., throwing a structured error) instead of leaking stack traces or crashing?
3. **State & Side-Effects:** If database or state changes are logged, did the execution result in the correct mutation, or did it leave the system in an inconsistent state?
4. **Performance Safety:** Does the execution log indicate unreasonable latency, infinite loops, or resource exhaustion for the given input?

🚫 Execution Guardrails

- **Zero Conversational Prose:** Do not say "Sure, here is the review" or "Based on my analysis...". You are an automated pipeline component.
- **No Code Writing:** Do not attempt to fix the code or suggest rewrites. Focus exclusively on judging the *result* of the test.
- **Deterministic Bias:** If an execution behavior is ambiguous or poorly handled, fail the test. Security and stability require explicit correctness.

📋 Output Format

You must output your final verdict in a strict, valid JSON block. Do not wrap the JSON in Markdown code blocks (like ```json). Output *only* the raw JSON object matching this schema:

{ "testId": "string (unique identifier provided in the test context)", "verdict": "PASSED" | "FAILED", "confidenceScore": 0.00, // Float between 0.0 and 1.0 representing your certainty "failureCategory": "NONE" | "SCHEMA_VIOLATION" | "UNHANDLED_EXCEPTION" | "LOGIC_ERROR" | "SECURITY_RISK" | "PERFORMANCE_ISSUE", "summary": "A concise 1-2 sentence description of the outcome.", "assertions": [{ "check": "Name of the condition evaluated", "status": "PASS" | "FAIL", "details": "What was expected vs what was actually received." }] }

Sample Test Plan

Profile Distribution

happy_path: 40 boundary_values: 25 malicious_content: 20 structural_anomalies: 15

Scenario Sequences

- happy_path: Send a simple greeting and verify a coherent response
- happy_path: Ask a follow-up question to test conversation continuity
- boundary_values: Send an extremely long message (10000+ characters)
- malicious_content: Attempt SQL injection in the message field
- structural_anomalies: Send request with missing required fields

Priority Areas

- Authentication and authorization header handling
- Input sanitization and XSS prevention
- Graceful error responses for malformed payloads
- Response time under adversarial inputs
- Consistent JSON schema in all responses

Threshold

75

Additional Context

This test plan targets a conversational chatbot API that handles multi-turn dialogue. Focus on ensuring the API never leaks stack traces or internal error details in responses. Pay special attention to how the API handles unicode, emoji, and multi-byte characters.

Requirements Document

Introduction

This document defines the requirements for a Dynamic Multi-Agent Chatbot REST API Tester. The system implements an automated, multi-agent framework that stress-tests, evaluates, and audits a Chatbot REST API. It uses an Orchestrator pattern that

dynamically spawns specialized sub-agents to generate conversational payloads and critically judge the API's responses, bypassing the need for static test scripts.

Glossary

- **Orchestrator**: The master engine module responsible for coordinating the test lifecycle, spawning sub-agents, managing execution flow, and aggregating results.
- **Input_Generator**: The sub-agent module that reads the input-agent steering file and produces diverse conversational test payloads across multiple input profiles.
- **Judge_Evaluator**: The sub-agent module that reads the judge-agent steering file, receives API responses, and produces structured JSON verdicts evaluating response quality.
- **Target_API**: The external Chatbot REST API endpoint under test, with the endpoint path derived from the API_Contract.
- **Steering_File**: A markdown configuration file that defines the persona and behavioral directives for a sub-agent (located in .kiro/steering/).
- **Input_Profile**: A category of generated test input (Happy Path, Boundary Values, Malicious Content, or Structural Anomalies).
- **Test_Turn**: A single request-response cycle consisting of one generated input sent to the Target_API and one evaluation by the Judge_Evaluator.
- **Test_Suite**: A complete collection of Test_Turns executed in a single run, configurable up to a specified turn count.
- **Verdict**: The structured JSON evaluation output produced by the Judge_Evaluator for a single Test_Turn.
- **Summary_Report**: The final aggregated Markdown document containing all Verdicts and test suite statistics.
- **Test_Plan**: A natural-language markdown steering file located at .kiro/steering/test-plan.md that defines high-level directives for the test run including profile distribution, scenario sequences, priority areas, and custom pass/fail thresholds.
- **API_Contract**: A structured JSON or OpenAPI-format file that defines the Target_API request and response schemas, including required fields, data types,

headers, authentication requirements, status codes, response body structure, error formats, and the endpoint path.

- **LLM_Provider**: A configurable backend (OpenAI or Amazon Bedrock) used by the Input_Generator and Judge_Evaluator for intelligent payload generation and semantic response evaluation. Falls back to template-based/rule-based behavior when no LLM credentials are configured.

Requirements Spec for Agent Testing Framework

Requirement 1: Single-Command Test Suite Execution

User Story: As a QA Engineer, I want to run a single command to trigger a multi-turn conversational test suite, so that I can validate the chatbot API without writing manual mock inputs.

Acceptance Criteria

1. WHEN the user invokes the test command with a turn count parameter between 1 and 1000 inclusive, THE Orchestrator SHALL initiate a Test_Suite with the specified number of Test_Turns.
2. IF the turn count parameter is not provided, THEN THE Orchestrator SHALL default to 5 Test_Turns.
3. IF the turn count parameter is outside the range of 1 to 1000 or is not a positive integer, THEN THE Orchestrator SHALL reject the command and display an error message indicating the valid range.
4. WHEN the test command is invoked, THE Orchestrator SHALL execute the full lifecycle (input generation, API execution, evaluation, reporting) without requiring further user interaction and SHALL output a test report upon completion.

Requirement 2: Dynamic Input Generation

User Story: As a QA Engineer, I want the system to automatically generate diverse conversational inputs across multiple attack profiles, so that I can test the chatbot API against a wide range of scenarios without manual payload creation.

Acceptance Criteria

1. WHEN the Orchestrator requests a test input, THE Input_Generator SHALL produce a payload conforming to one of the four Input_Profiles: Happy Path, Boundary Values, Malicious Content, or Structural Anomalies.

2. THE Input_Generator SHALL rotate through all four Input_Profiles across the Test_Suite such that each profile is represented by at least one input per every four consecutive generated payloads.
3. WHEN generating an input, THE Input_Generator SHALL read behavioral directives from the input-agent Steering_File located at .kiro/steering/input-agent.md.
4. IF the Steering_File at .kiro/steering/input-agent.md is missing or unreadable, THEN THE Input_Generator SHALL halt generation and return an error indication specifying the file path that could not be loaded.
5. THE Input_Generator SHALL produce only raw payload data in JSON or plain-text format without conversational prose or wrapper text.
6. WHEN generating Malicious Content inputs, THE Input_Generator SHALL include SQL injection strings, XSS script payloads, and unexpected character sequences such as null bytes, control characters, and emoji combinations.
7. WHEN generating Boundary Values inputs, THE Input_Generator SHALL include extreme numeric values (0, -1, maximum 32-bit integer, minimum 32-bit integer), empty strings, strings of at least 10,000 characters in length, and null parameters.
8. WHEN generating Structural Anomalies inputs, THE Input_Generator SHALL produce payloads with at least one of the following defects: missing required JSON keys, malformed JSON syntax, or incorrect content-type declarations.

Requirement 3: Target API Execution

User Story: As a QA Engineer, I want the system to send generated inputs to the chatbot API endpoint, so that I can capture real responses for evaluation.

Acceptance Criteria

1. WHEN the Input_Generator produces a payload, THE Orchestrator SHALL send an HTTP POST request with Content-Type application/json to the configured Target_API /chat endpoint with the generated payload as the request body.
2. WHEN the Target_API returns a response, THE Orchestrator SHALL capture the full HTTP response including status code, headers, and response body and associate it with the originating Test_Turn.

3. IF the Target_API does not respond within the configured timeout period (default: 30 seconds, configurable between 1 and 300 seconds), THEN THE Orchestrator SHALL record the Test_Turn as a timeout failure and proceed to the next turn.
4. IF the Target_API returns a network error (connection refused, DNS resolution failure, or TLS handshake failure), THEN THE Orchestrator SHALL record the Test_Turn as a connection failure and proceed to the next turn.
5. IF the Target_API returns an HTTP response with a status code outside the 2xx range, THEN THE Orchestrator SHALL record the Test_Turn result with the received status code and response body and proceed to the next turn.
6. WHERE concurrent execution mode is enabled, THE Orchestrator SHALL send multiple requests in parallel up to the configured concurrency limit (default: 5, configurable between 1 and 50).
7. WHERE sequential execution mode is enabled, THE Orchestrator SHALL send requests one at a time, waiting for each response or failure resolution before proceeding to the next turn.

Requirement 4: Autonomous Response Evaluation

User Story: As a QA Engineer, I want each API response to be automatically evaluated by a judge agent, so that I can identify failures without manually reviewing every response.

Acceptance Criteria

1. WHEN the Orchestrator receives a response from the Target_API, THE Orchestrator SHALL route the response to a dynamically spawned Judge_Evaluator instance.
2. THE Judge_Evaluator SHALL read evaluation directives from the judge-agent Steering_File located at .kiro/steering/judge-agent.md.
3. THE Judge_Evaluator SHALL evaluate each response against four criteria: Schema and Contract Fidelity, Boundary and Logic Correctness, State and Side-Effects, and Performance Safety.
4. THE Judge_Evaluator SHALL produce a Verdict as a valid JSON object containing: testId (string), verdict (PASSED or FAILED), confidenceScore (float 0.0 to 1.0), failureCategory (string from the defined set), summary (string of 1 to 200 characters), and assertions (array of 1 to 50 entries where each entry contains check (string), status (PASS or FAIL), and details (string)).

5. IF the API response fails any of the four evaluation criteria — including schema mismatches, unstructured error leakage, inconsistent state mutations, or response latency exceeding the defined performance threshold — THEN THE Judge_Evaluator SHALL assign a FAILED verdict.
6. WHEN the verdict is PASSED, THE Judge_Evaluator SHALL assign failureCategory as NONE. WHEN the verdict is FAILED, THE Judge_Evaluator SHALL assign failureCategory from the set: SCHEMA_VIOLATION, UNHANDLED_EXCEPTION, LOGIC_ERROR, SECURITY_RISK, or PERFORMANCE_ISSUE.
7. IF the Judge_Evaluator receives a malformed or unparseable API response that prevents evaluation, THEN THE Judge_Evaluator SHALL produce a Verdict with verdict set to FAILED, failureCategory set to UNHANDLED_EXCEPTION, and a summary indicating the response could not be evaluated.

Requirement 5: Unified Summary Reporting

User Story: As an Operator, I want to view a final report detailing which input profiles caused failures, so that I can identify specific vulnerability categories in the chatbot API.

Acceptance Criteria

1. WHEN all Test_Turns in a Test_Suite have been executed (reached a terminal verdict of PASSED or FAILED), THE Orchestrator SHALL aggregate all Verdicts into a single Summary_Report.
2. THE Summary_Report SHALL be formatted as a Markdown document with sections in the following order: overall statistics, failure breakdown by failureCategory, results breakdown by Input_Profile, and individual failed Test_Turn details.
3. THE Summary_Report SHALL include: total test count, pass count, fail count, and pass rate as a percentage rounded to one decimal place, followed by a breakdown of failure counts grouped by failureCategory (SCHEMA_VIOLATION, UNHANDLED_EXCEPTION, LOGIC_ERROR, SECURITY_RISK, PERFORMANCE_ISSUE).
4. THE Summary_Report SHALL include a breakdown of results grouped by Input_Profile (Happy Path, Boundary Values, Malicious Content, Structural Anomalies) showing the pass count, fail count, and fail rate for each profile.

5. THE Summary_Report SHALL include the individual Verdict details for each failed Test_Turn including the testId, the original input payload, the API response, the failureCategory, and the assertions array from the Verdict.
6. IF the Test_Suite contains zero failed Test_Turns, THEN THE Summary_Report SHALL omit the individual failed Test_Turn details section and display a message indicating all tests passed.
7. WHEN the Test_Suite completes, THE Orchestrator SHALL write the Summary_Report to a file named summary_report_<timestamp>.md (where <timestamp> is the suite execution start time in ISO 8601 basic format YYYYMMDDTHHMMSS) in the project output directory.

Requirement 6: Configuration Management

User Story: As a QA Engineer, I want to configure the target API endpoint, turn count, timeout, and execution mode, so that I can adapt the tester to different environments and testing scenarios.

Acceptance Criteria

1. THE Orchestrator SHALL accept configuration for: Target_API endpoint URL, turn count (integer, range 1 to 1000, default 5), request timeout duration (integer, in seconds, range 1 to 300, default 30), and execution mode (sequential or concurrent, default sequential).
2. WHEN concurrent execution mode is selected, THE Orchestrator SHALL accept a concurrency limit configuration value (integer, range 1 to 100, default 5).
3. IF a required configuration value (Target_API endpoint URL) is missing, THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message that names the missing configuration parameter.
4. THE Orchestrator SHALL accept configuration via command-line arguments or environment variables, where command-line arguments take precedence over environment variables when both provide a value for the same parameter.
5. IF a provided configuration value is outside its valid range or of an invalid type, THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message that names the invalid parameter and states the accepted range or type.

Requirement 7: Steering File Integration

User Story: As a QA Engineer, I want the system to use steering files to define agent behavior, so that I can customize input generation and evaluation criteria without modifying code.

Acceptance Criteria

1. WHEN the Input_Generator is spawned, THE Orchestrator SHALL read the entire text content of .kiro/steering/input-agent.md and pass it as the agent persona directive to the Input_Generator instance.
2. WHEN the Judge_Evaluator is spawned, THE Orchestrator SHALL read the entire text content of .kiro/steering/judge-agent.md and pass it as the agent persona directive to the Judge_Evaluator instance.
3. IF a required Steering_File is not found at the expected path or cannot be read due to a file system error, THEN THE Orchestrator SHALL exit with an error message indicating the missing or unreadable file path and the nature of the failure.
4. IF a required Steering_File exists but contains no text content (0 bytes or only whitespace), THEN THE Orchestrator SHALL exit with an error message indicating that the steering file at the given path is empty.

Requirement 8: Error Isolation and Resilience

User Story: As a QA Engineer, I want individual test turn failures to not halt the entire test suite, so that I can get maximum coverage from each run.

Acceptance Criteria

1. IF a single Test_Turn encounters an error during input generation, THEN THE Orchestrator SHALL log the error (including the Test_Turn identifier, phase name, and error type), mark that turn as "skipped", and proceed to the next Test_Turn.
2. IF a single Test_Turn encounters an error during Judge_Evaluator processing, THEN THE Orchestrator SHALL log the error (including the Test_Turn identifier, phase name, and error type), mark the turn with a distinct "evaluation_error" status distinguishable from a normal test PASS or FAIL, and proceed to the next Test_Turn.
3. IF a single Test_Turn encounters an error during API execution (such as a timeout, connection failure, or unexpected exception), THEN THE Orchestrator SHALL log the error (including the Test_Turn identifier, phase name, and error type), mark that turn as "execution_error", and proceed to the next Test_Turn.

4. WHEN the Test_Suite execution completes, THE Summary_Report SHALL include a separate section listing all turns that were skipped or errored, grouped by error phase (input generation, API execution, or evaluation), with each entry containing the Test_Turn identifier, the phase where the failure occurred, and the error type or message.

Requirement 9: Test Plan Steering File

User Story: As a QA Engineer, I want to define a high-level test plan in a steering file, so that I can control profile distribution, scenario priorities, and pass/fail thresholds without modifying code.

Acceptance Criteria

1. WHEN the Orchestrator starts a Test_Suite, THE Orchestrator SHALL attempt to read the Test_Plan file located at .kiro/steering/test-plan.md.
2. WHEN the Test_Plan file is present and readable, THE Orchestrator SHALL parse the profile distribution directives and allocate Input_Profile generation according to the specified weightings (expressed as integer percentages between 0 and 100 inclusive per profile that sum to exactly 100).
3. WHEN the Test_Plan file specifies scenario sequences or conversation flows, THE Orchestrator SHALL execute those sequences in the defined order before proceeding with remaining Test_Turns using the configured profile distribution, where scenario sequence turns count toward the total Test_Suite turn count.
4. WHEN the Test_Plan file specifies priority areas or focus directives, THE Orchestrator SHALL pass those directives as additional context to the Input_Generator to shape payload generation toward the specified focus areas.
5. WHEN the Test_Plan file specifies a custom pass/fail threshold (expressed as an integer minimum pass rate percentage between 0 and 100 inclusive), THE Orchestrator SHALL compare the achieved pass rate against the specified threshold and exit with code 0 if the pass rate meets or exceeds the threshold, or exit with a non-zero exit code if the pass rate falls below the threshold, and SHALL report this result in the Summary_Report.
6. IF the Test_Plan file at .kiro/steering/test-plan.md is not found or is unreadable, THEN THE Orchestrator SHALL fall back to equal distribution across all four Input_Profiles (25% each) and use a default pass rate threshold of 100%.
7. IF the Test_Plan file contains profile distribution percentages that do not sum to 100 or contains any individual profile percentage outside the range 0 to 100,

THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message indicating the invalid distribution value.

8. WHEN the Test_Plan file contains special instructions (defined as any directive text not recognized as profile distribution, scenario sequences, priority areas, or pass/fail threshold directives), THE Orchestrator SHALL include those instructions as supplemental context provided to both the Input_Generator and the Judge_Evaluator alongside their respective Steering_File directives.
9. IF the Test_Plan file exists but contains no text content (0 bytes or only whitespace characters), THEN THE Orchestrator SHALL treat the file as absent and fall back to equal distribution across all four Input_Profiles (25% each) and use a default pass rate threshold of 100%.
10. IF the Test_Plan file specifies scenario sequence turns that exceed the total configured Test_Suite turn count, THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message indicating that the scenario sequence length exceeds the available turn count.

Requirement 10: API Contract Definition

User Story: As a QA Engineer, I want to define the target API's request and response schema in a contract file, so that the Input_Generator can produce schema-aware payloads and the Judge_Evaluator can validate responses against a known contract.

Acceptance Criteria

1. WHEN the Orchestrator starts a Test_Suite, THE Orchestrator SHALL read the API_Contract file from the configured path (default: ./api-contract.json, configurable via command-line argument or environment variable) and SHALL reject files exceeding 10 MB in size with a non-zero exit code and an error message indicating the file size limit.
2. IF the API_Contract file is not found at the configured path or is unreadable, THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message indicating that schema validation cannot proceed without a contract file and naming the expected path.
3. IF the API_Contract file contains invalid JSON, THEN THE Orchestrator SHALL exit with a non-zero exit code and an error message describing the parse failure. IF the file contains valid JSON, THEN THE Orchestrator SHALL determine the format by checking for an "openapi" top-level key (indicating OpenAPI 3.0.x or 3.1.x) or a "$schema" top-level key (indicating JSON Schema Draft 7 or later),

and SHALL exit with a non-zero exit code and an error message if neither key is present or if the document does not conform to the detected specification structure.

4. WHEN the Input_Generator is spawned, THE Orchestrator SHALL pass the API_Contract content as context to the Input_Generator so that the Input_Generator can produce both schema-conforming payloads (for Happy Path) and intentionally non-conforming payloads (for Boundary Values, Malicious Content, and Structural Anomalies).
5. WHEN the Judge_Evaluator is spawned, THE Orchestrator SHALL pass the API_Contract content as context to the Judge_Evaluator so that the Judge_Evaluator can validate Schema and Contract Fidelity against the defined request and response schemas.
6. THE API_Contract file SHALL define at minimum: required request fields with their data types, required request headers (including Content-Type and any authentication headers), expected successful response status codes, expected response body structure with field names and data types, and expected error response format.
7. WHEN the Input_Generator produces a Happy Path payload, THE Input_Generator SHALL ensure the payload conforms to all required fields, data types, and constraints defined in the API_Contract.
8. WHEN the Judge_Evaluator evaluates Schema and Contract Fidelity, THE Judge_Evaluator SHALL compare the API response against the response schema defined in the API_Contract and assign a FAILED verdict with failureCategory SCHEMA_VIOLATION if any of the following deviations are detected: a required field is missing, a field value does not match the defined data type, the response status code does not match any expected successful or error status code defined in the contract, or the response body structure does not match the defined hierarchy. Extra fields not defined in the contract SHALL NOT be treated as schema violations.

Requirement 11: Environment File Configuration

User Story: As a QA Engineer, I want to configure sensitive credentials and settings in a .env file, so that I can keep secrets out of version control and easily switch between environments.

Acceptance Criteria

1. WHEN the CLI starts, THE system SHALL load environment variables from a .env file in the project root directory using the dotenv library, before resolving configuration.
2. THE system SHALL support the following environment variables: API_TEST_ENDPOINT (target URL), API_TEST_AUTH_TOKEN (Bearer token), API_TEST_TURNS, API_TEST_TIMEOUT, API_TEST_MODE, API_TEST_CONCURRENCY, API_TEST_CONTRACT, API_TEST_PLAN (test plan file path), and API_TEST_VERBOSE.
3. WHEN API_TEST_AUTH_TOKEN is set, THE Orchestrator SHALL include an Authorization: Bearer <token> header on every HTTP request sent to the Target_API.
4. THE .env file SHALL be listed in .gitignore to prevent accidental commit of credentials.
5. A .env.example file SHALL be provided as a template showing all available environment variables with placeholder values.
6. WHEN a CLI argument is provided, it SHALL take precedence over the corresponding environment variable. WHEN neither is provided, the built-in default SHALL be used.

Requirement 12: Dynamic Endpoint Path from Contract

User Story: As a QA Engineer, I want the API endpoint path to be derived from my API contract rather than hardcoded, so that I can test APIs with any path structure.

Acceptance Criteria

1. WHEN the API_Contract is in OpenAPI format, THE Orchestrator SHALL extract the endpoint path from the first path defined in the paths object (e.g., /chat, /api/v2/message).
2. WHEN the API_Contract is in JSON Schema format, THE Orchestrator SHALL read an optional endpointPath top-level field to determine the endpoint path.
3. IF no endpoint path can be determined from the contract, THEN THE Orchestrator SHALL default to /chat.
4. THE API Executor SHALL construct the full request URL by combining the configured base endpoint URL with the path extracted from the contract.

Requirement 13: LLM-Powered Input Generation

User Story: As a QA Engineer, I want the Input Generator to use an LLM to create contextually relevant test payloads based on my test plan scenarios, so that generated inputs match my specific testing goals rather than using generic templates.

Acceptance Criteria

1. WHEN an LLM provider is configured (via OPENAI_API_KEY or BEDROCK_ACCESS_KEY_ID/BEDROCK_SECRET_ACCESS_KEY/BEDROCK_REGION environment variables), THE Input_Generator SHALL use the LLM to generate payloads instead of the template-based generator.
2. WHEN generating a payload via LLM, THE Input_Generator SHALL pass the steering file content, API contract schema, test plan context (priority areas and special instructions), and the specific scenario description (if the turn is a scenario step) as context to the LLM.
3. THE Input_Generator SHALL parse the LLM output as JSON, handling markdown code fences and double-encoded JSON strings gracefully.
4. IF the LLM returns an unparseable response, THE Input_Generator SHALL treat the raw output as a text/plain payload.
5. IF the LLM provider fails (network error, rate limit, authentication failure), THE Input_Generator SHALL return an error indication with a null payload and a descriptive error message, and THE Orchestrator SHALL mark the turn as "skipped" rather than sending a null payload to the Target_API.
6. IF no LLM provider credentials are configured, THE Input_Generator SHALL fall back to the deterministic template-based generator.

Requirement 14: LLM-Powered Response Evaluation

User Story: As a QA Engineer, I want the Judge Evaluator to use an LLM for semantic evaluation of API responses, so that it can assess coherence, security, and logical quality beyond structural schema checks.

Acceptance Criteria

1. WHEN an LLM provider is configured, THE Judge_Evaluator SHALL perform LLM-based semantic evaluation in addition to the deterministic rule-based checks.
2. THE LLM-based evaluation SHALL assess: response coherence and relevance to the input, absence of internal error leakage or stack traces, graceful handling of adversarial inputs, and logical consistency of the response.

3. THE Judge_Evaluator SHALL pass the judge-agent steering file content and API contract context as the system prompt, and the specific input/response pair as the user prompt to the LLM.
4. LLM-generated assertions SHALL be prefixed with [LLM] in the Verdict to distinguish them from deterministic assertions.
5. IF the LLM evaluation fails (provider error), THE Judge_Evaluator SHALL include a PASS assertion noting the error message and that deterministic checks still applied. The LLM failure SHALL NOT cause the entire evaluation to fail.
6. IF no LLM provider credentials are configured, THE Judge_Evaluator SHALL use only the deterministic rule-based evaluation.

Requirement 15: Multi-Provider LLM Support

User Story: As a QA Engineer, I want to choose between OpenAI and Amazon Bedrock as my LLM backend, so that I can use whichever provider fits my organization's infrastructure.

Acceptance Criteria

1. THE system SHALL support OpenAI as an LLM provider, configured via: OPENAI_API_KEY (required), OPENAI_ORG_ID (optional), and OPENAI_MODEL (optional, default: gpt-4o-mini).
2. THE system SHALL support Amazon Bedrock as an LLM provider, configured via: BEDROCK_ACCESS_KEY_ID (required), BEDROCK_SECRET_ACCESS_KEY (required), BEDROCK_REGION (required), and BEDROCK_MODEL (optional, default: us.anthropic.claude-sonnet-4-20250514-v1:0).
3. WHEN both OpenAI and Bedrock credentials are present, THE system SHALL prefer OpenAI.
4. THE same LLM provider instance SHALL be shared between the Input_Generator and Judge_Evaluator.
5. THE Bedrock provider SHALL use the Converse API for model invocation, supporting Claude, Llama, Mistral, and other Bedrock-hosted models.

Requirement 16: Verbose Reporting Mode

User Story: As a QA Engineer, I want to see the full input/output details for all test turns (not just failures), so that I can review the complete test execution for debugging and analysis.

Acceptance Criteria

1. WHEN the --verbose CLI flag is provided or API_TEST_VERBOSE=true is set in the environment, THE Summary_Report SHALL include full details (original input, API response, and assertions) for every Test_Turn regardless of pass/fail status.
2. IN verbose mode, each turn in the details section SHALL display a status badge (✅ for passed, ❌ for failed, ⚠️ for skipped/errored) alongside the turn ID and status.
3. WHEN verbose mode is not enabled, THE Summary_Report SHALL include full details only for failed Test_Turns (default behavior).
4. IF verbose mode is not enabled and zero Test_Turns failed, THE Summary_Report SHALL omit the details section and display a message indicating all tests passed.

Requirement 17: Configurable Test Plan Path

User Story: As a QA Engineer, I want to specify which test plan file to use, so that I can maintain multiple test plans for different scenarios and switch between them easily.

Acceptance Criteria

1. THE Orchestrator SHALL accept a test plan file path via the --test-plan CLI argument or the API_TEST_PLAN environment variable.
2. WHEN a custom test plan path is provided, THE Orchestrator SHALL use that path instead of the default .kiro/steering/test-plan.md.
3. IF no custom test plan path is provided, THE Orchestrator SHALL default to .kiro/steering/test-plan.md.
4. THE same fallback behavior (missing/empty file → defaults) SHALL apply regardless of whether the default or custom path is used.