Challenges in Testing Large Language Model Based Software: A Faceted Taxonomy

FELIX DOBSLAW, Mid Sweden University, Sweden
ROBERT FELDT, Chalmers University of Technology, Sweden
JUYEON YOON and SHIN YOO, KAIST, Republic of Korea

Large Language Models (LLMs) and Multi-Agent LLMs (MALLMs) introduce non-determinism, unlike traditional or machine learning software, where models typically produce deterministic outputs for given inputs. LLM-based software requires new approaches to verifying correctness that account for variability within individual test cases, not just across different inputs. This paper presents a taxonomy for LLM test case design, informed by research literature and our experience. Each facet is exemplified, and we conduct an LLM-assisted analysis of six open-source testing frameworks, perform a sensitivity study of an agent-based system across different model configurations, and provide working examples contrasting atomic and aggregated test cases. We identify key variation points that impact test correctness and highlight open challenges that the research, industry, and open-source communities must address as LLMs become integral to software systems. Our taxonomy defines four facets of LLM test case design, addressing ambiguity in both inputs and outputs while establishing best practices. It distinguishes variability in goals, the system under test, and inputs, and introduces two key oracle types: atomic and aggregated. Our findings reveal that current tools treat test executions as isolated events, lack explicit aggregation mechanisms, and inadequately capture variability across model versions, configurations, and repeated runs. This highlights the need for viewing correctness as a distribution of outcomes rather than a binary property, requiring closer collaboration between academia and practitioners to establish mature, variability-aware testing methodologies.

Additional Key Words and Phrases: Large Language Models, Software Testing, Correctness, Ambiguity, Aggregated Oracles

1 Introduction

Large Language Models (LLMs) and Multi-Agent LLMs (MALLMs) [18, 40] are transforming software development—not just through their capabilities but also due to their inherent non-determinism. Unlike traditional systems, where variability arises from, e.g., unreliable servers, stochastic sub-components, or real-time inaccuracies, LLMs exhibit fundamental unpredictability due to their construction, differences in model selection, configuration, and input variations at both syntactic and semantic levels. Conventional testing methods and approaches to oracle formulation and correctness [2] struggle in this setting. Based on our experience in designing, implementing, testing, and assessing several software systems/solutions either partly or fully built around LLMs [12, 20–25, 41, 46], we propose a taxonomy for LLM test case design that captures key challenges and nuances in this emerging paradigm.

In this article, we focus on testing *LLM-based software systems* rather than evaluating LLMs in isolation. By LLM-based software, we mean systems in which an LLM is used as a component with a clearly defined intent

Authors' Contact Information: Felix Dobslaw, Mid Sweden University, Department of Communication, Quality Management and Information Systems, Östersund, Sweden, felix.dobslaw@miun.se; Robert Feldt, Chalmers University of Technology, Department of Computer Science and Engineering, Gothenburg, Sweden, robert.feldt@chalmers.se; Juyeon Yoon; Shin Yoo, KAIST, School of Computing, Daejeon, Republic of Korea, {juyeon.yoon,shin.yoo}@kaist.ac.kr.

ACM 1557-7392/2026/4-ART
https://doi.org/10.1145/3806396

and expected behaviour inside a larger architecture—such as a medical decision-support tool that synthesises recommendations from patient records or a content moderation pipeline where an LLM classifies or filters items within an existing service. This perspective also includes multi-agent LLM (MALLM) architectures, where several specialised agents collaborate toward shared goals and intents are distributed across agents rather than concentrated in a single model call. In such settings, the LLM components "dock into" an established software stack much like other subsystems, but their non-deterministic, natural-language behaviour and inter-agent coordination require different test design strategies, with a stronger emphasis on integration and end-to-end behaviour. Our focus therefore differs from traditional ML testing surveys that treat models as primary artefacts [3] and from model-centric LLM evaluation that assesses standalone models on benchmarks [4]; we instead study how test cases, oracles, and adequacy criteria should be structured when the intended behaviour emerges from the interaction between LLM components and surrounding software.

Recent research highlights the impact of input and output ambiguity in LLM-based applications. Subtle prompt variations can invert model responses, even under high-confidence settings [44], and repeated queries–despite deterministic configurations (e.g., temperature = 0)–can produce inconsistent outputs [1]. This variability raises concerns for replicability and necessitates advances in automated oracle design [29].

Traditional software testing relies on deterministic oracles, but the probabilistic nature of LLMs challenges this assumption. While Barr et al. [2] introduced probabilistic oracles to handle non-determinism, their framework does not account for prompt-driven variability in LLMs and MALLMs. More broadly, ML testing requires a paradigm shift [3], yet existing work does not explicitly treat variation as a first-class concern. Crucially, traditional ML testing typically assumes that a trained model produces deterministic outputs for a given input—variability is assessed across different inputs in a test dataset, not across repeated executions of the same input. In contrast, LLM-based software exhibits non-deterministic outputs even for identical inputs, requiring test designs that account for variability within single test cases rather than only across them. The challenge is further compounded by LLMs' hybrid nature, where behavior emerges from a combination of code, model inference, and prompt engineering.

This divergence from traditional testing is evident in how correctness is defined. Existing paradigms–whether deterministic [2], stochastic [2, 15, 16], or ML-specific [10, 33]–struggle to address the multi-layered complexity of LLM-based systems. Unlike traditional ML models, which produce deterministic outputs for a given input and where uncertainty is primarily assessed across different inputs in a test dataset, LLMs introduce fundamental non-determinism: repeated executions with identical inputs can yield different outputs. This requires test designs that explicitly account for variability within individual test cases, not just across them. LLMs also introduce an additional axis of complexity: the prompt itself, which acts as both input specification and behavioral modifier. Moreover, LLM performance may degrade post-deployment due to data shifts, yet monitoring mechanisms remain underdeveloped [3]. This fundamental difference in how variability manifests—within test cases versus across them—means that established ML testing methodologies cannot be directly applied to LLM-based software without significant adaptation. The disconnect between LLM testing tools and core ML or SE testing literature further underscores the need for specialized methodologies [19].

This article extends our previous work [11] by presenting a faceted taxonomy for LLM test case design that organizes testing concerns across four dimensions: Software Under Test, Goal, Oracles, and Inputs. Central to this taxonomy is the distinction between *atomic oracles*, which evaluate single executions under the assumption of determinism, and *aggregated oracles*, which assess correctness across multiple runs to account for stochastic behavior. We validate the taxonomy through comprehensive empirical investigations that combine multiple methods: concrete examples demonstrating each facet, LLM-assisted analysis of six open-source testing frameworks (Opik, DeepEval, RAGAs, Promptfoo, Phoenix, and Giskard), a sensitivity study comparing different model configurations (GPT-3.5-turbo vs GPT-4) in the DroidAgent agent-based system, and working implementations contrasting atomic and aggregated test approaches.

Our findings reveal gaps across multiple dimensions of current testing tooling and practice. First, analysis of the six frameworks shows that tools predominantly treat each test execution as an isolated event, relying on atomic oracles that presume deterministic outcomes. Second, while some tools aggregate results at the dataset level (e.g., reporting pass rates), none provide explicit mechanisms for handling multiple evaluations of the same input within a single system configuration–ignoring the stochastic variability inherent to LLMs. Third, the sensitivity study demonstrates that even minor configuration changes (model selection, temperature settings) can dramatically alter system behavior, yet current tools lack systematic support for capturing and comparing such variability across model versions and configurations. These observations collectively challenge the assumption that correctness can be assessed through isolated executions and reveal that mature LLM testing requires viewing correctness as a distribution of outcomes rather than a binary property. This conceptual shift has implications not only for tool design and aggregation strategies but also for how testing responsibilities may need to extend beyond traditional software testers to include domain experts who possess the contextual knowledge necessary for validation in specialized domains.

The structure of the article is as follows. Section 2 offers background and motivates the need for a taxonomy and structured approach to the testing of LLM-based software. Section 3 introduces our faceted taxonomy [37] for LLM test cases, categorizing key variation points that impact evaluation correctness, including the distinction between Atomic and Aggregated Oracles to address the identified gap. Unlike prior work–[44] on adversarial robustness and [19] on broad LLM testing taxonomies–our framework organizes test case design across multiple facets, extending beyond specific testing foci and high-level categorization. Section 4 presents our empirical investigations, including tool mapping, LLM-assisted analysis, and sensitivity studies. Finally, Section 5 discusses implications for the field and identifies key challenges for ensuring correctness in LLM-based software toward 2030.

2 Background and Motivation

The proposed taxonomy for LLM test cases is designed to support not only the traditional testing of finalized LLM-based systems but also to play a central role throughout the entire development, maintenance, and evolution phases. Given the inherent non-determinism, flexible behavior, and the vast and often unpredictable output space of LLMs, we believe that test cases will be an essential part of the development process of any LLM-based system, far more so than for traditional software systems.

While the emphasis on the need for test cases throughout the development lifecycle strongly reminds us of Test-Driven Development (TDD), we argue that LLM-based systems would take on a more dynamic and iterative process than that of TDD as we know it. In conventional software development, TDD follows a relatively linear cycle: developers write a test case which declaratively specifies what the expected program behavior should be, implement code to pass the test, and maintain a stable, "green" test suite–unless the specification changes. However, in LLM-based systems, the development process is less deterministic. Developers often begin with only a vague understanding of "what to test", especially in domains with complex requirements (e.g., medical chatbots). The high degree of causal uncertainty–the inability to predict how an LLM will interpret and follow a given prompt–means that developers cannot statically reason about expected behaviors. Unlike traditional code, where the flow of execution can often be understood without running it, at least on lower levels, LLM behaviors must be explored through dynamic analysis, observing actual outputs to identify patterns, strengths, and failures.

This leads to a more fluid form of TDD, where both the Software Under Test (SUT) and the test cases themselves evolve together. Developers iteratively refine prompts, configurations, and even the underlying models, all while adapting and extending the test suite based on observed behaviors. Importantly, achieving a 100% green test suite is often neither feasible nor the goal. Instead, the focus shifts to systematically reducing uncertainty and guiding the system toward acceptable behavior ranges.

Furthermore, during the design phase, it's not uncommon that developers experiment with different base LLMs to evaluate their suitability for specific components. While traditional testing phases typically assume a fixed system configuration, in LLM-based development, variability in models, prompts, and hyperparameters is a key part of the design process. This means that testing is not simply a validation tool but a primary mechanism for system exploration and refinement.

This interplay between model selection, prompt engineering, and test design underscores the importance of integrating testing as an ongoing, foundational activity. The taxonomy we propose below supports this need by offering a structured framework for test case design that remains valuable across all stages of a system's lifecycle–from early prototyping to deployment and long-term maintenance.

In this context, testing is not a final verification step but an integral driver of system development. This dynamic approach to TDD highlights that, for LLM-based systems, testing is even more critical than in traditional software development, where formal methods, type systems, and static analyses can partially guarantee behavior. In contrast, for LLM-centric systems, testing is often the primary means of ensuring reliability, safety, and functionality.

Traditional software testing relies on a specification that defines the expected behavior, interfaces, and constraints of the system under test (SUT). In LLM-based systems, this process becomes more complex due to the flexibility and ambiguity of natural language interactions. To manage this, developers often use *prompt templates*—structured inputs with placeholders, such as:

> Write code to read <FORMAT> file in <PROGRAMMING_LANGUAGE> with telling variable names and add code comments.

These templates function like macros in traditional software, offering structured yet customizable input formats controlled by developers.

While prompt templates enable diverse inputs, they also complicate testing. LLMs typically generate free-text responses without strict adherence to requested formats, making automated evaluation challenging. To address this, *invariant checks* assess both syntactic correctness (e.g., ensuring the output is valid Python code) and semantic quality (e.g., verifying meaningful documentation). However, LLM responses can be ambiguous or only partially fulfil a request, requiring correctness evaluations to account for deviations ranging from minor formatting errors to complex, context-dependent variations.

These challenges, along with strategies for handling them, are further explored in the Oracle section. The Inputs section discusses how test cases instantiate prompt templates with varying, concrete data for evaluation.

3 Taxonomy for LLM Test Case Design

Testing LLMs presents unique challenges due to their non-deterministic behavior and the ambiguity inherent in both inputs (e.g., prompt phrasing and intent) and outputs (e.g., diverse, context-dependent responses). Beyond variability, practical concerns such as cost and granularity in test case design further complicate reliable evaluation. To address these challenges, we propose a structured taxonomy for LLM testing, organized around four core dimensions: **Software Under Test (SUT), Goal, Oracles, and Inputs**. Each of these core dimensions is further subdivided into finer-grained sub-facets, capturing the specific factors that influence test case design and evaluation as well as points of variability between specific test case runs. This taxonomy provides a systematic framework for developing and refining test cases across the software lifecycle, emphasizing the need for adaptive and continuous testing strategies tailored to the dynamic nature of LLM-based systems. The order of these dimensions reflects their logical dependencies in test case design.

(1) **SUT** is presented first because it defines the underlying system or component to be evaluated. The SUT typically remains constant across multiple test cases, serving as the foundation upon which different testing scenarios are applied.
(2) The test case **Goal** comes next, as it specifies the unique objective of the test case and highlights the specific properties of it to be tested. While the SUT may remain unchanged, each test case targets a specific property, such as ensuring functional correctness or non-functional goals like safety, fairness, or robustness.
(3) **Oracles** are then defined for each property derived from the test case goal.
(4) **Inputs** are the final dimension, consisting of the datasets, user interactions, or synthetic prompts used to elicit responses from the SUT, which are then assessed using the defined oracles.

This ordered structure can help ensure clarity in the design process. Starting with the SUT allows for reusable test infrastructures, while distinct goals enable focused evaluations. The oracles then provide concrete evaluation mechanisms for each property, and the inputs serve as the vehicle to generate diverse test executions. However, the taxonomy does not prescribe a particular order of analysis, and developing and testing with LLMs is an inherently iterative process. Below, we detail each dimension further and discuss its sub-facets.

3.1 Case: Issue report classification

We explain all four facets in support of a case SUT by name ClassifyIssueReport, which has the purpose of classifying issue reports for issue tracking (such as on GitHub). Given an issue text, the SUT shall return exactly one label (for reasons of simplicity) - BUG, FEATURE, INVALID, or DUPLICATE. Invalid issue reports demand work without benefit [26]. In case of a duplicate, i.e., an issue that has been brought up already in an existing issue report, point to the matching existing issue. The classification then could lead to different actions, for example, assigning bugs to an appropriate tester or developer, or adding a duplication note in the thread with reference to a duplicate issue and closing it.

Classifying reported issues has practical relevance and can be solved in many different ways in support of traditional software, machine learning, or LLMs. Using LLM's, we could imagine simple solutions with a single model or agentic systems that collaborate over the entire project's issue reporting history to match potential duplicates.

3.2 Software Under Test

The system under test (SUT) refers to the software implementation being evaluated for correctness. In this context, we define the SUT as a software system that integrates one or more large language models (LLMs)–such as in multi-agent LLM architectures [46] (e.g., MALLM)–to achieve the expected behavior specified in its design. To analyze variation both across test cases and within a single test execution, we break down the SUT in LLM-centric systems into the following key **sub-facets**.

The **Component** represents the specific functional unit or role within the system that is under test. In LLM-based systems, this often corresponds to a prompt (or prompt template), an instruction set, or an agent role. For instance, in a multi-agent setup, the component might be the *Planner* agent (as in the multi-agent LLM system of [46]) or a specific prompt guiding a data transformation task. However, our taxonomy remains agnostic to the level of testing: in a multi-agent LLM system, a targeted subsystem composed of multiple agents may itself serve as the SUT. The component facet thus defines the logical function under test, independent of the specific model(s) executing it.

The **Model(s)** refers to the LLM(s) implementing the component(s) during test execution. Different base models (e.g., *Claude 3.5*, *Deepseek R1*, *GPT-4o*) can produce distinct behaviors even under identical prompts, making this a

critical variation point. In MALLMs, each component may rely on a different model or a combination of models, requiring test cases to account for these permutations.

Each model operates under specific **Configurations** that directly influence its behavior. These include model parameters such as temperature, top-k/top-p sampling, and maximum token limits, as well as system-level settings like API rate limits and external tool integrations. Configurations may vary per model or be shared across components, introducing additional variability that must be managed in test design.

We acknowledge that the combinations of various components, models, and configurations can result in a wide range of architectural designs, each with a different number of models, different prompting styles, different configurations, as well as different connections between them. While we do not claim an exhaustive list of architectural design choices, we do envision following design choices for the SUT:

- Prompts vs. Agents: an important dimension in the space of SUTs is the main architecture of the SUT. Earlier LLM applications were primarily just prompts (e.g., Libro, which aims to perform bug reproduction in the form of code completion prompts [24]), engineered with techniques like few-shot learning and Chain-of-Thoughts. Increasingly, LLM applications adopt agentic architecture, assisted with techniques like Retrieval Augmented Generation (RAG) and ReAct-based tool usage [45]. We note that the rich interaction between agents and their environments and tools would introduce extra variability that needs to be verified for reliable and robust operations.
- Single vs. Multiple LLMs: the SUT may involve just a single LLM instance, or multiple instances, each with different configurations. An SUT using a single LLM can span from simple individual prompts to an agentic system driven by a single LLM instance (e.g., AutoFL [21] or AutoCodeRover [48]). Multiple LLM instances may be required to optimize LLM queries (for example, routing [5, 28] or cascading [47] query requests), or simply to implement a multi-agent architecture (e.g., DroidAgent [46]). With the use of multiple LLMs comes the increased variability in SUT - for example, we can envision multi-agent systems that rely on multiple LLMs, depending on the specific requirements of each task allocated for participating agents.
- Artifact-oriented vs. Conversational: we posit that the nature of the task for which the SUT is designed will affect the overall testing of the SUT. Artifact-oriented tasks are those designed to produce specific artifacts at the end: it may be specific texts, images, or source code that realizes a specific functionality. However, we also envision another category of LLM-based applications whose main purpose is the conversation itself, providing education [32] or therapeutic consultation [7]. In particular, we posit that conversational agents introduce an extra dimension to the variability of the SUT, which is the user, i.e., the counterpart of the conversational interaction. While users are not directly under testing, they can introduce a high degree of variability that the LLM-based software systems need to effectively deal with: it may be different levels of detail, use of different terminology, or any other personal traits relevant to the purpose of the conversation.

In the context of a specific test case, the SUT is a concrete instance with fixed choices for the Component, Model(s), and Configuration(s). However, our taxonomy explicitly distinguishes between fixed and variable elements across test runs. Any modification–such as swapping the underlying LLM, adjusting a temperature setting, or altering the role's prompt–creates a new SUT instance that may behave differently under the same input.

This approach contrasts with traditional software testing, where the SUT is typically treated as fixed and configuration changes are minor or peripheral. In LLM-based systems, even small variations can significantly impact behavior, making them a **deliberate factor in test case design rather than an unintended side effect**. The complexity increases further in hybrid systems combining LLMs with conventional code, where changes in either can affect overall system behavior.

SUT Example

Components. The `ClassifyIssueReport` SUT is a combination of a duplication checking function `DuplicationFinder` and an LLM call. For a given issue report, the function returns either a matching issue id - to then return DUPLICATE - or null. In the latter case, the LLM is invoked.
Model and Configuration. One LLM is used only when the duplication check returns null: Mistral-7B-v0.1. The `DuplicateFinder` tool is used in version 0.3. The LLM is called with a fixed prompt defining BUG, FEATURE, and INVALID and instructing to answer with *exactly* one of them. Result variability is limited using temperature=0.0, top_p=1.0, n=1, and max_tokens=16.

3.3 Goal

The goal defines the high-level objective of a test case, which is then refined into specific, measurable sub-goals called **Properties**–the concrete conditions the SUT must satisfy. For instance, if the goal is to *ensure the safety of LLM outputs*, relevant properties might include that the LLM must not generate offensive language (Property 1) and must not encourage harmful behavior (Property 2). These properties form the basis for designing oracles that evaluate whether the SUT meets the defined goals.

Goal Examples

G1 Decision validity. Given one issue report, the system should return exactly one label from BUG, FEATURE, INVALID, DUPLICATE.

- **P1.1** Single-label: the output contains exactly one label.
- **P1.2** Membership: the label is strictly one of BUG, FEATURE, INVALID, DUPLICATE.
- **P1.3** Format discipline: no extra content beyond the label.

G2 Duplicate alignment. When the duplication function provides a matching id, the system should surface a DUPLICATE decision that references that id; when it provides no match, the system should decide among BUG, FEATURE, INVALID.

- **P2.1** Tool consistency: if the duplication function returns an id, the decision is DUPLICATE and the same id is echoed.
- **P2.2** Non-duplicate discipline: if the duplication function returns null, the decision is not DUPLICATE.
- **P2.3** ID provenance: any reported duplicate id must come from the duplication function.

G3 Consistent behavior. At fixed prompt/model/settings, the system should produce the same decision for the same input text.

- **P3.1** Run-to-run stability: at fixed settings, the same input yields the same label.
- **P3.2** Minor-text robustness: light, meaning-preserving paraphrases yield the same label.
- **P3.3** Prompt-format robustness: benign formatting changes (e.g., whitespace, line breaks) do not change the label.

We posit that types of concrete properties expected from LLM-based applications can be categorised into functional and non-functional properties, similarly to traditional software.

- Functional Properties: any features provided by LLM-based applications should functionally deliver what has been specified by the original requirements: properties such as correctness, reliability, and consistency all apply.
- Non-functional Properties: Non-functional properties widely studied for traditional software systems, such as safety, security, speed, and efficiency, all apply here. However, we note that the highly abstract and unstructured nature of natural language interfaces means that non-functional properties expected from LLM-based applications extend further than those expected from traditional software systems. For example, the safety criterion for LLM-based applications may have to consider the tone of the language generated by LLMs, compliance with social norms, and psychological impact on users, in addition to the usual safety implications of the SUT functionalities.

3.4 Oracles

Oracles decide whether the SUT satisfies the property(ies) implied by a test case goal. We make a novel distinction between two levels—*Atomic* and *Aggregated* oracles—and argue that the latter is essential for testing LLM-based software, where both the SUT and the checker may behave stochastically/non-deterministically.

An **Atomic Oracle** evaluates a *single* test execution. It takes one observed output and returns an assessment, much like in traditional testing, where an output is checked against a criterion. In LLM-based testing, the atomic check itself comes in two main forms. First, **deterministic atomic checks** always return the same result for the same observed output, such as equality comparisons, regular expressions, schema or JSON-structure validation, or other structural matching—well suited to predictable artifacts like numerical computations or strictly formatted responses. Second, **non-deterministic atomic checks** may vary across repeated applications, even when given the same observed output. This category includes probabilistic and semantic similarity measures, human-in-the-loop assessments, and *LLM-as-judge* evaluations, which inherit non-determinism from the judging model itself. These alternatives are often necessary for open-ended outputs, but they may still fail to fully account for LLMs' variability, even when the SUT is configured for determinism (e.g., temperature = 0) [1].

An **Aggregated Oracle** addresses this limitation by evaluating correctness across *multiple* observations and then synthesising them into a final assessment. Concretely, an aggregated oracle wraps an atomic oracle: it applies the atomic check to each observation and then combines the resulting judgements using an aggregation rule. A key point is that "multiple observations" being aggregated can arise from two different sources of randomness. We may repeat **SUT execution** to obtain multiple outputs for the same input and configuration (capturing output variability), and/or repeat the execution of the **atomic oracle** to obtain multiple judgements for the same output (capturing judgement variability, for example, when the atomic oracle is non-deterministic, e.g., using an LLM-as-judge). In both cases, aggregated oracles treat correctness as an empirical distribution rather than a single outcome, and they make the final verdict depend on how consistently the property holds across samples.

We posit that an aggregated oracle must make at least three design choices: (i) the scope over which it aggregates, (ii) the decision rule used to turn evidence into a verdict, and (iii) how it accounts for unequal credibility among observations.

- **Scope of aggregation (what is aggregated).** Aggregation may be **temporal**, combining repeated executions of the same test case under identical conditions to estimate run-to-run stability. It may be **structural**, combining observations across systematic variants—such as paraphrased prompts, alternative temperatures or seeds, different model versions, or other configuration changes—to probe robustness and sensitivity. Finally, it may aggregate across **multiple judging signals**, such as several atomic checks, multiple judges (including multiple LLM judges), or mixed automated metrics, to reduce dependence on any single checker.

- **Decision rule family (how evidence becomes a verdict). Strict** rules require universal agreement, such as *all-of* / "fail-if-any," and are appropriate when a single violation is unacceptable (e.g., safety-, security-, or compliance-related properties, or strict output formats). **Threshold** rules allow occasional failures but flag systematic issues, such as *k-of-n* criteria, majority/plurality voting, or pass-rate $\geq \tau$. **Distributional** rules go beyond a binary pass/fail (at aggregated level) and instead characterise risk and instability using measures such as variance, entropy, quantiles, tail-risk (e.g., worst-k behaviour), or confidence intervals. These are particularly useful when goals are graded, or the desired behaviour is underspecified, where "how often" and "how bad" may matter more than a single verdict. More generally, aggregated-oracle sampling can be governed by standard *sequential* stopping rules, so the number of runs is determined adaptively while controlling error rates [38].
- **Weighting and credibility (whose evidence counts more).** Not all observations are equally informative. Aggregation may therefore use **weighted** votes or weighted thresholds—for example giving more weight to more reliable judges, higher-severity scenarios, or more realistic operational conditions—so that the final assessment reflects evidential strength rather than raw counts.

This distinction has practical implications for LLM-based testing. Many properties can be checked with deterministic atomic predicates when the expected output is well-defined and structured. But as soon as either the SUT output or the check itself is stochastic, batch-style testing becomes necessary: we must collect samples, aggregate judgements, and report not only a verdict but also the observed reliability profile. We argue that this is rather the norm than the exception for LLM-based system testing, which makes it unique compared to traditional software testing, where this level of variation is rarely seen or, if it is available, rarely explicitly handled or supported.

Oracle Examples

For G1 Decision validity.

- **O1 (P1.1–P1.3)** Per-case check: read the output; confirm it has exactly one label, that the label is in BUG, FEATURE, INVALID, DUPLICATE, and that nothing extra is included.
- **O1-metric** Dataset view: share of cases that pass O1.

For G2 Duplicate alignment.

- **O2 (P2.1)** Per-case check when the tool returns an id: label is `DUPLICATE` and the same id is returned.
- **O3 (P2.2)** Per-case check when the tool returns `null`: label is not `DUPLICATE`.
- **O2/O3-metric** Dataset view: share of cases that follow the tool result (report separately for tool=id and tool=null).

For G3 Consistent behavior.

- **O4 (P3.1)** Metric: repeatability — same label across re-runs with fixed settings.
- **O5 (P3.2)** Metric: paraphrase agreement — same label across light rewordings.
- **O6 (P3.3)** Metric: format agreement — same label across harmless formatting changes.

We note that metamorphic testing offers a useful point of reference from traditional software testing: it is, in effect, a limited form of *aggregated oracle* in the sense we propose here. Instead of judging a single execution against a known expected output, a metamorphic oracle evaluates a *set* of related executions—typically generated by applying a transformation to the input—and checks whether the outputs satisfy a specified relation (a metamorphic relation) [6]. In our situation here, each individual run can still be checked with an atomic predicate

Table 1. Heuristic guidance and examples for selecting atomic and aggregated oracles by property type.

Property type (goal)	**Atomic oracle(s)**	**Aggregation strategy**
Safety / Security / Compliance	Deterministic rule-based checks (e.g., blocklist/policy rules, invariant checks, format constraints); optionally human review for critical cases	**Strict-pass** (*fail-if-any*); optionally report tail-risk (e.g., worst-k) when sampling is used or further guardrails are used before action or user impact
Functional correctness (structured outputs: code, JSON, extraction)	Structural/syntax validation (schema, parsing, type checks), exact-match or constrained matching; unit-style assertions when applicable	**Strict-pass** for hard requirements; otherwise **high threshold** pass-rate $\geq \tau$ when outputs are probabilistic but must be reliable
Semantic correctness (open-ended: summaries, chat, explanations)	LLM-as-judge and/or semantic similarity scoring; rubric-based checks; keyword/coverage checks for required facts	**Pluralistic/threshold** (majority vote, k-of-n, pass-rate $\geq \tau$) and/or **distributional** reporting (variance, quantiles)
Robustness / Stability (sensitivity to seeds, prompts, configs)	Atomic checks aligned with the goal (e.g., judge score, similarity, constraint violations) applied per variant	**Structural scope** aggregation across variants; **distributional metrics** (variance/instability, entropy, confidence intervals)

(e.g., compute a property of an output), but the final verdict comes from aggregating evidence across the runs by enforcing cross-run consistency. This perspective is particularly relevant for LLM-based applications, where the "right" output may be underspecified, yet meaningful relations across multiple executions (or variants) can still be stated and tested.

While our taxonomy is conceptual and will require further research to refine, Table 1 provides further examples by offering lightweight heuristics for choosing atomic and aggregated oracles based on the domain and the property being verified.

3.5 Inputs

Inputs are the concrete prompts that drive SUT execution, forming the final dimension of the taxonomy. They typically populate placeholders in prompt templates with specific dataset values, simulating real user interactions or system executions. However, in systems allowing multi-turn dialogues, having reasoning agents, or stateful architectures, later inputs often depend on earlier outputs, which can make static substitutions insufficient.

In addition to specifying the specific data values or how to derive sequences of inputs, the Inputs dimension can also list explicit variability operators to assess the SUT's robustness. These fall into two categories: *syntactic variations*, which modify formatting or wording while preserving meaning, and *semantic variations*, which shift the meaning to explore different behaviors. Such variations apply at both the prompt template and input levels, expanding test coverage and blurring the distinction between prompt templates as input (from the LLM's perspective) and as part of the SUT (as a development artefact).

This layered approach allows comprehensive probing of the SUT, generating diverse outputs for evaluation by oracles. Unlike traditional software testing, where inputs map directly to expected outputs, LLM interactions introduce ambiguity–user inputs may refine, complement, or complete a prompt, complicating clear test case boundaries.

Systematically addressing input variability involves creating comprehensive input coverage strategies that incorporate both syntactical and semantical dimensions. This approach not only ensures higher reliability and

robustness of LLM-based software but also supports clearer identification of failure points related to input ambiguities.

Inputs Example (Adequacy)

What we check. We declare the input space *adequately covered* when the following simple, countable conditions hold:

- **Class balance.** Each base item class (BUG, FEATURE, INVALID, DUPLICATE) has at least 50 items.
- **Syntactic variation.** For every base item, generate exactly 3 meaning-preserving variants: (S1) whitespace/line-break change, (S2) punctuation/case tweak, (S3) benign formatting edit.
- **Semantic variation (boundary).** For 20% of items per class, add 2 near-miss paraphrases that push a neighboring class boundary (e.g., BUG↔FEATURE).

How we record it. Each row stores `base_id`, `class`, `variant_type` ∈ {BASE, S1, S2, S3, SEM1, SEM2}. A simple adequacy report lists per-class counts and per-operator coverage:

- **Per-class coverage:** target met if ≥ 50 BASE items and ≥ 150 syntactic variants (3 per BASE) per class.
- **Boundary coverage:** target met if ≥ 10
- **Stop rule:** declare input adequacy when all targets are met; otherwise, generate the missing variants only.

While the example above is domain-specific, it suggests the possibility of extracting reusable adequacy patterns that could apply across different LLM-based software contexts. For instance, patterns might include ensuring class balance through minimum representation thresholds per category, systematically generating syntactic variants to test robustness, or explicitly targeting boundary cases between categories. However, developing a comprehensive set of adequacy patterns with broad applicability would require a more structured, empirically underpinned approach involving systematic analysis across diverse domains and use cases, which is beyond the scope of this paper. Also, while we expect the list to grow further, the following criteria provide the basic dimensions of the space of inputs for LLM-based applications.

- Uni-modal vs. Multi-modal: while the primary modality of LLM-based applications was initially the uni-modal text, multi-modal models such as Visual Language Models (VLMs) and generative diffusion models are rapidly advancing, resulting in multi-modal applications that can handle images as both input and output being more prevalent. The image modality may introduce further challenges in input generation: testing of deep learning software systems showed us that sampling of image inputs is not trivial [43].
- Human-origin vs. Machine-origin: as agentic workflows become increasingly adopted, certain inputs to an LLM-based software may actually be outputs generated by another LLM-application. From a testing perspective, whether there exists any meaningful difference between data from these two origins is an interesting open question.
- Unstructured vs. Structured: inputs to SUTs may be unstructured natural language text, or structured data such as JSON or source code.

4 Empirical Investigations

To demonstrate and test the practical value of our taxonomy, we conducted three complementary empirical investigations. Together, they examine how the taxonomy can guide both human and automated tool evaluation

and how selected aspects of it manifest in practice. The first two studies focus on testing tools: Section 4.1 presents a manual mapping of six open-source frameworks, while Section 4.2 extends this analysis by turning the taxonomy into a detailed checklist that also serves as a structured prompt for LLM-based evaluation. This dual approach allowed us to assess how well the taxonomy captures existing practices and to explore whether LLMs can assist in identifying tool capabilities and gaps not yet represented in the taxonomy. The third study, in Section 4.3, shifts focus from tools to systems under test, applying a sensitivity analysis to a multi-agent Android testing system [46] to illustrate how SUT variability appears across different LLM models and leads to output variability that motivates aggregated oracles. All supplementary materials, including tool documentation, analysis prompts, detailed results, and complete action logs, are available online[1].

4.1 Manual Taxonomy-Based Tool Evaluation

Here we analyze how each facet of our taxonomy is addressed by existing tools that represent the state of practice in LLM application testing, by explicitly mapping taxonomy facets to the concrete functional *features* supported by the tools (Table 2). Several open-source [9, 14, 31, 34–36] and hosted [27, 42] platforms have emerged to support this domain. We focus on a selection of widely used open-source frameworks as representative examples and study how their exposed features align with and instantiate the facets of our taxonomy.

We identified relevant GitHub repositories by searching for "LLM Testing" and "LLM Evaluation", selecting frameworks that specifically target LLM applications (i.e., systems combining LLMs with prompt templates, optionally enhanced with retrieval mechanisms or tool-calling capabilities) and provide dedicated testing interfaces. The six selected projects, chosen based on GitHub popularity (stars) and distinct testing features, are Opik (14.4k), DeepEval [34] (11.4k stars), RAGAs (11k stars), Promptfoo [35] (8.6k stars), Phoenix (7.1k), and Giskard [14] (4.9k stars) at the time of writing.

Table 2. Mapping of supported features in open-source testing frameworks to taxonomy facets.

Facet	Tool Features	Opik	DeepEval	RAGAs	Promptfoo	Phoenix	Giskard
SUT	LLM Chains or Agents	✓	✓	✓	✓	✓	✓
	Conversational Agents	✓	✓	✓	✓	✓	✓
	Multi-LLM Systems	-	-	-	-	-	-
	Version Comparison	✓	✓	✓	✓	✓	-
Goal	Functional	✓	✓	✓	✓	✓	✓
	Non-functional (Perf.)	-	✓	-	✓	-	-
	Non-functional (Safety)	✓	-	-	✓	✓	-
Oracles	Deterministic Metrics	✓	✓	✓	✓	✓	✓
	LLM-as-a-judge	✓	✓	✓	✓	✓	✓
	Multiple Judging Signals	✓	✓	✓	✓	✓	-
	Trajectory Eval.	✓	✓	-	✓	-	-
	Temporal Aggr.	-	-	-	-	-	-
	Structural Aggr.	-	-	-	-	-	-
	Weighted Aggr.	-	-	-	✓	-	-
	Decision Rules for Aggr.	-	-	-	-	-	-
Inputs	Multi-modal Inputs	✓	-	-	-	✓	-
	Human Inputs	✓	✓	✓	✓	✓	✓
	Machine Inputs	-	-	-	-	-	-
	Dataset Synthesis	-	-	-	✓	✓	✓
	Dataset Augmentation	✓	-	-	-	-	-
	Coverage/Adequacy	-	-	-	-	-	-

[1] https://doi.org/10.5281/zenodo.17393106

SUT. All tools accommodate variability across the three SUT sub-facets, though the supported feature scope differs slightly across sub-facets. Specifically, all surveyed frameworks support testing *LLM chains or agents* beyond a single prompt and multi-turn *conversational agents*. However, none of the open-source tools explicitly support multi-LLM systems as first-class SUTs.

Each allows users to specify the *Component* under test (e.g., a prompt template or agent), select the *Model(s)* (e.g., GPT-4o, Claude 3.7 Sonnet), and adjust *Configurations* (e.g., temperature, token limits), aligning with the *version comparison* feature captured in the table. Notably, Promptfoo enables "unfolded" comparisons across prompt versions, model choices, or configuration settings. Similarly, Opik and DeepEval include visualization support but primarily focus on identifying regressions through aggregated metrics (e.g., the proportion of passing test cases within a dataset).

While Giskard's open-source SDK does not directly support cross-version SUT comparisons, this functionality is available through integration with an external enterprise platform [13].

Assessment: Each tool, to some degree, recognizes that developers frequently re-evaluate multiple implementations throughout development. However, support for multi-agent LLM architectures remains limited: existing tools do not explicitly support testing individual agents and their integrated system behaviour as distinct, first-class units of analysis.

Goal. All surveyed tools support *functional goals*, while support for *non-functional goals* is uneven. Only a subset of tools provides explicit mechanisms for evaluating performance-related properties (e.g., latency or cost), and safety-oriented goals are supported by even fewer frameworks.

In most tools, a test case consists of a single input paired with an expected output, evaluated against one or more property-checking metrics. Giskard places greater emphasis on dataset-level scores and offers limited controls over property selection within a test run, though finer-grained goal configuration is only available through its enterprise platform. While other tools allow executing multiple test cases within a single evaluation, they do not explicitly model a test goal as a first-class entity that groups multiple properties or inputs. Conversely, Giskard lacks flexibility in configuring properties at the individual test case level.

Together, these frameworks fall short of balancing flexibility (checking properties at the individual input level) with coherence (grouping relevant inputs under a common evaluation goal).

Assessment: While these tools offer various property-checking mechanisms, they struggle with goal variability and representability–the ability to flexibly define and assess test goals at different levels of granularity. Although properties serve as concrete criteria derived from test goals, current frameworks do not explicitly model this distinction or integrate it into evaluation workflows. In addition, existing tools do not clearly distinguish between functional and non-functional goals (e.g., safety or performance), instead treating them as interchangeable sets of checks without explicit semantic separation at the test case level.

Oracles. The tools commonly support a range of oracles, often referred to as "metrics" or "assertions," including deterministic checks (e.g., substring matching) and heuristic, model-based evaluations. Users can define custom metrics using tailored prompts for a separate LLM instance (*LLM-as-a-judge*) [17, 49] or through code-based implementations, and most allow combining multiple judging signals. Web interfaces enable manual review and labeling, but human-in-the-loop validation remains underdeveloped. While tools allow human-provided labels, they lack mechanisms for iterative collaboration between humans and model evaluators.

Support for *trajectory-level evaluation* is also limited to a subset of tools, and no open-source framework provides explicit support for temporal (i.e., combining repeated executions of the same test case) or structural (i.e., combining runs across prompt or configuration variations of SUT) aggregation of repeated evaluations. Weighted aggregation is only partially supported, and decision rules governing aggregation outcomes are absent across all tools.

Evaluations typically yield either a continuous score (e.g., similarity to a ground-truth output on a 0-1 scale) or a binary pass/fail result, with continuous scores often thresholded into binary outcomes. Giskard mainly operates at the dataset level, aggregating individual assessments (e.g., reporting a 70% pass rate for politeness). However, LLM outputs can vary across repeated runs, and current tools lack aggregation methods to handle multiple evaluations of the same input within the same system under test (SUT). Instead, each run is either treated as an independent outcome or simply averaged (Phoenix), ignoring stochastic variability.

We argue for aggregated oracles that account for repeated evaluations, using different aggregation strategies depending on the property under test and the source of variation. For example, a strict requirement such as "the response MUST follow the specified JSON format" should pass only if no violations occur across multiple runs. Conversely, a softer requirement like "the response should be polite" could be satisfied if the majority of runs meet the criterion.

Assessment: While these tools offer diverse oracle definitions, model-based heuristic oracles remain unstable, and deterministic oracles lack expressiveness. Existing dataset-level aggregation is limited, and current tools fail to systematically capture variability across repeated runs. A principled approach to aggregated oracles is needed, along with flexible aggregation schemes (e.g., majority vote, strict pass, confidence-based thresholds) tailored to different evaluation properties.

Inputs. Developers often struggle with a lack of data or inputs when testing LLM applications. Existing tools mitigate this by generating synthetic inputs using separate LLM instances ("Dataset Synthesis") or modifying existing inputs with predefined operators ("Dataset Augmentation"). Test inputs are typically obtained either by *extracting data from existing artefacts*, such as documentation, logs, or curated datasets, or by *synthesizing inputs using dedicated LLM prompts*.

Dataset augmentation, as supported by tools such as Opik, spans both *syntactic* variations (e.g., punctuation removal, case conversion, or language changes) and *semantic* variations (e.g., generating edge cases from typical examples). In addition, tools such as Promptfoo support automated adversarial input generation for security and safety testing directly from prompt templates, aligning with known attack patterns [30], including jailbreaking prompts and the embedding of encrypted forbidden content to bypass security filters.

However, as indicated by the absence of coverage-related features in Table 2, these input generation mechanisms lack explicit adequacy criteria or quantitative objectives. Consequently, testers have limited guidance on whether generated inputs sufficiently stress the SUT or how to prioritize inputs for regression testing. In addition, existing tools primarily focus on *human-provided inputs*, with limited support for multi-modal inputs and no explicit handling of machine-generated internal states as test inputs, typically required for handling multi-agent LLM systems.

Assessment: While current tools offer diverse input synthesis features, a major limitation in existing tools is the absence of clear test adequacy criteria for input variability. This gap makes it difficult to reason about test completeness or optimize testing effort under cost constraints, by systematically assessing whether variations sufficiently test SUT robustness.

Overall, the manual mapping provided a structured view of how current testing frameworks address the key facets of our taxonomy. Yet, performing such detailed analyses manually is time-consuming and prone to subjective interpretation. To explore whether this process can be accelerated and standardized, we next examined whether large language models themselves can assist in evaluating testing tools.

4.2 LLM-based Tool Evaluation Through a Detailed Checklist/Prompt

Building directly on the manual analysis presented in Section 4.1, we transformed the taxonomy into a detailed checklist that operationalizes each facet as a set of concrete evaluation questions. This checklist serves a dual purpose: it can guide human analysts toward systematic coverage and also functions as a structured prompt

Table 3. Checklist rating scale used for tool evaluation

Category	Description
Not supported	No evidence of this capability.
Limited	Minimal or basic support, typically covering only one narrow aspect or requiring significant manual work.
Partial	Moderate support, covering multiple aspects but with notable gaps or limitations.
Strong	Comprehensive support with well-developed features.

for LLM-based evaluation. Using it, we investigated whether an LLM could analyze tool documentation with sufficient precision to replicate or complement human judgment.

We applied this approach to the same six tools analyzed manually (Opik, DeepEval, RAGAs, Promptfoo, Phoenix, and Giskard) by consolidating their documentation and using the checklist as a detailed prompt to Claude Sonnet 4.5. This method produced a reusable artifact for standardized tool comparison and validated that the taxonomy captures relevant dimensions of real-world tools. The complete checklist and the individual analysis results for each tool, together with the consolidated LLM-based summaries, are available in the online replication package; below we summarize the checklist's structure and focus on insights that complement or extend the manual analysis.

The checklist operationalizes the taxonomy's four core facets through targeted sub-questions that probe concrete tool capabilities. Each sub-capability is rated on a four-point scale (not supported, limited, partial, strong) and grounded in direct evidence from documentation. The framework also includes cross-cutting criteria such as reproducibility, cost control, security, and ecosystem integration, as well as a "Beyond the Taxonomy" section that captures emerging or unexpected features. This section also serves as an indirect evaluation of the taxonomy itself, revealing tool capabilities or dimensions that are not yet well represented within its current structure.

We use the four-level ordinal scale to characterize the extent of support for each checklist item. The categories are defined as follows and summarized in Table 3. The main differentiators are that limited typically reflects support for a single aspect of the capability, partial reflects support for multiple aspects with noticeable omissions, and strong reflects broad, well-integrated support across aspects. The ratings were supported by an LLM using the prompt provided in the supplementary material, which enhances transparency of the procedure without claiming full reproducibility of the exact scores.

Applied as an LLM prompt, the checklist yielded coherent, high-level summaries and consistent facet-level characterizations that largely aligned with our manual analysis. However, the model often misjudged finer-grained criteria, mistaking superficial mentions for true support—for instance, labeling any reference to repeated execution as "partial support" for aggregated oracles even when statistical analysis was absent. Thus, while LLMs can accelerate preliminary assessments and highlight relevant documentation, human interpretation remains essential for judging depth and context. The checklist nonetheless offers a structured, reusable foundation for both automated and manual evaluation of LLM testing tools.

4.2.1 LLM-based analysis of Current Tool Landscape. The LLM-based analysis largely confirmed the patterns identified through manual evaluation (Section 4.1), particularly the maturity of atomic oracle support and the absence of aggregated oracle mechanisms. The automated analysis also highlighted several additional dimensions that extend beyond the taxonomy's current scope.

The "Beyond the Taxonomy" analyses exposed several dimensions that extend or challenge the current framework. The development–production continuum emerged as a missing axis, with many tools blurring the boundary between testing, monitoring, and deployment. While our taxonomy focuses on test design during development, it would be an interesting future work to evaluate if a broader view could extend the usefulness

further. Domain-specific testing paradigms, such as Ragas's knowledge graph–based evaluation and DeepEval's ConversationSimulator, suggest that specialized approaches may warrant explicit recognition. However, we argue that the core of the taxonomy should be generally applicable and thus not domain-specific. In addition, adoption and usability factors—including documentation quality, community support, and ease of integration—strongly influence tool uptake yet fall outside the taxonomy. Together, these insights show that the "Beyond the Taxonomy" section not only captures emerging innovation but also serves as a mechanism for evaluating and evolving the taxonomy itself.

Table 4. Repeated runs of actor agent from DroidAgent using GPT-3.5 and GPT-4o

Model	Run	Succeed	# Actions	# Effective	# Redundant	Cause of Failure
GPT-3.5	1	X	20	0	0	Failed to proceed from permission popup
	2	X	12	5	7	Failed to input a valid deck name (was an empty string)
	3	X	20	3	17	Failed to pass the permission popup (mistakenly clicks back buttons)
	4	X	20	2	18	Failed to pass the permission popup
	5	X	16	5	11	Failed to input a valid deck name (was an empty string)
	6	O	20	7	13	-
	7	X	20	2	18	Failed to pass the permission popup
	8	X	20	1	19	Failed to pass the permission popup
	9	X	20	0	20	Failed to pass the permission popup
	10	X	20	1	19	Failed to pass the permission popup
GPT-4o	1	O	8	8	0	-
	2	X	20	3	17	Mistakenly got into the app info page from the permission popup
	3	O	7	7	0	-
	4	O	7	7	0	-
	5	O	8	8	0	-
	6	O	7	7	0	-
	7	O	7	7	0	-
	8	O	7	7	0	-
	9	O	8	8	0	-
	10	O	7	7	0	-

4.3 LLM sensitivity analysis

Our third investigation shifts attention from testing tools to the systems under test themselves. By executing the same LLM-based agent multiple times—each time implemented with a different underlying model—and observing the resulting outputs, we explore how the taxonomy's SUT variability and Oracle facets appear in practice. This small empirical study highlights concrete needs for future LLM testing frameworks. Table 4 summarizes ten repeated runs of the actor agent in DroidAgent [46], an autonomous GUI testing system for Android applications. In each run, the agent receives the prompt `Create a new flashcard deck named 'Computer Science'` and performs the corresponding actions in the open-source app AnkiDroid. We compare results using two underlying models, GPT-3.5 and GPT-4o, to analyze behavioral differences across SUT configurations. The table's columns—# of actions, # effective, and # redundant—indicate, respectively, the total actions taken, those that produced meaningful GUI changes, and those that did not. Below, we first examine the variability in outcomes, motivating the use of aggregated oracles, and then discuss configuration variability across models.

4.3.1 Nondeterministic Outputs and Usage of Aggregated Oracle. Results in Table 4 immediately show the variability of LLM outputs. The functional variability concerns whether the actor has successfully executed the given scenario. Assuming that we can verify whether the agent succeeded or not based on the final status of the GUI,

the following (simplified) test case can check the result of the agent behaviour as well as whether it succeeded within the given budget (i.e., the number of GUI actions):

```
def test_droidagent_atomic_deterministic_result():
  with open("scenario.txt", "r") as s:
      test_scenario = s.readlines()
      agent = DroidAgent(model="gpt-4o")
      agent.execute(test_scenario)

      assert agent.get_current_screen().contains_text("Computer Science")
      assert agent.get_num_actions() < 20
```

However, there also exists non-functional variability. That is, even successful actions can vary in concrete details while being semantically equivalent. For example, consider the following two action steps generated by GPT-4o:

Listing 1. Action log with length 7

```
[step_0] (from the permission dialog page) I tapped the "
      Allow" button to grant notification permissions to
      AnkiDroid.
    > adb shell input tap [540, 1314]

[step_1] (from the AnkiDroid permission dialog page) I
      tapped the "OK" button to dismiss the storage
      permission request dialog.
    > adb shell input tap [880, 1700]

[step_2] (from the permission dialog page) I tapped the "
      Allow" button to grant storage permissions to
      AnkiDroid.
    > adb shell input tap [540, 1345]

[step_3] (from the main deck selection page) I tapped the "
      Add" button to initiate the creation of a new deck.
    > adb shell input tap [941, 2200]

[step_4] (from the add deck page) I tapped the "Create deck"
       button to start the process of naming a new deck.
    > adb shell input tap [865, 1615]

[step_5] (from the create deck page) I tapped the text field
       to focus it and input "Computer Science" as the new
      deck name.
    > adb shell input tap [540, 1233]
    > adb shell input text ['Computer Science']

[step_6] (from the create deck page) I tapped the "OK"
      button to confirm the creation of the new deck named "
      Computer Science".
    > adb shell input tap [880, 1400]



.
```

Listing 2. Action log with length 8

```
[step_0] (from permission prompt page) I allowed AnkiDroid
      to send notifications to dismiss the permission
      prompt.
    > adb shell input tap [540, 1315]

[step_1] (from welcome screen) I tapped the "OK" button to
      dismiss the welcome message and proceed with the
      setup.
    > adb shell input tap [880, 1691]

[step_2] (from permission prompt page) I tapped the "Allow"
       button to grant AnkiDroid access to files.
    > adb shell input tap [540, 1346]

[step_3] (from deck picker page) I tapped the "Add" button
      to proceed to the next step of creating a new
      flashcard deck.
    > adb shell input tap [940, 2200]

[step_4] (from deck picker page) I tapped the "Create deck"
       button to start creating a new flashcard deck.
    > adb shell input tap [865, 1615]

[step_5] (from create deck page) I tapped on the EditText
      field to focus on it for entering the deck name.
    > adb shell input tap [540, 1233]

[step_6] (from create deck page) I entered the text '
      Computer Science' into the focused EditText field to
      name the new flashcard deck.
    > adb shell input text ['Computer Science']

[step_7] (from create deck page) I tapped the "OK" button
      to confirm and create the new flashcard deck named '
      Computer Science'.
    > adb shell input tap [880, 1397]
```

In case we want to check the agent behaviour against a specific human reference, such non-functional variability would prevent us from using simple Boolean predicates. Instead, we need to capture the semantic contents of the generated output (in this case, the action logs) and compare their similarity to the reference using a threshold:

```
def test_droidagent_atomic_semantic_realistic():
  with open("scenario.txt", "r") as s, open("reference_actions.txt", "r") as r:
      test_scenario = s.readlines()
      agent = DroidAgent(model="gpt-4o")
      agent.execute(test_scenario)
```

```
    actions = agent.get_action_log()

    human_reference = r.readlines()

    similarity = measure_similarity(actions, human_reference)
    assert similarity > threshold
```

Some of the properties may require other LLMs for checking due to their abstract nature. Suppose we have to use another LLM as a judge to determine whether a given action log contains any redundant action steps or not. The simplified test case would look like the following:

```
def test_droidagent_atomic_deterministic_redundancy():
  with open("scenario.txt", "r") as s:
      test_scenario = s.readlines()
      agent = DroidAgent(model="gpt-4o")
      agent.execute(test_scenario)

      actions = agent.get_action_log()

      prompt = f"""An Android GUI testing agent tried to execute the following
      test scenario: {test_scenario}. It has executed the following actions:
      {actions}.

      Your task is to assess whether the actions taken by the agent are
      efficient and do not contain any redundant steps. Answer yes if this is
      the case, otherwise no. Do not add any explanation."""

      assessment = ollama.generate(model="gpt-4o", prompt=prompt)
      assert "yes" in assessment
```

Given the variability across ten different runs shown in Table 4, we argue that any of the test cases shown above should be applicable to a number of runs, the results of which are then aggregated. For example, we can rewrite the functional test cases above for repeated runs as follows:

```
def test_droidagent_aggregated_deterministic_result():
  with open("scenario.txt", "r") as s:
    test_scenario = s.readlines()
    agent = DroidAgent(model="gpt-4o")
    for i in range(10):
      agent.execute(test_scenario)
      num_passes += 1 if \
        agent.get_current_screen().contains_text("Computer Science") and \
        agent.get_num_actions() < 20
  assert num_passes / num_scenarios > threshold
```

While here we illustrate the aggregation over multiple runs of the same prompt, it is also possible to aggregate model behaviour across different prompts: for example, we can evaluate the actor agent using aggregated results from multiple scenarios. However, we note that support for aggregating repeated runs and customising the aggregation methods for domain needs is currently lacking in most frameworks.

4.3.2 SUT Variability: Model Versions. Another observation from Table 4 is that GPT-4o achieves a much higher success rate compared to GPT-3.5. While this is due to the advances in the foundation model and thus welcome, it does also raise the need for regression testing for model changes. Even if everything else remains the same, plugging in a different LLM into an existing system can produce regression faults, i.e., such a change may break a

functionality that was working well with the previous model. We note that the concept of such regression faults is generally lacking in current testing frameworks.

4.4 Summary of Empirical Findings

Across the three investigations, a clear pattern emerges. Current LLM testing tools (Sections 4.1 and 4.2) have extensive support for atomic evaluations/oracles but provide little systematic support for aggregated oracles, reproducibility, or variability-aware assessment. Both the manual and LLM-assisted analyses reveal similar structural gaps: tools rarely distinguish between goals and properties, offer limited mechanisms for measuring input coverage, and focus on reproducibility rather than true stochastic variability. The LLM-based evaluation further confirmed that while LLM models themselves can efficiently surface relevant evidence when evaluating new testing tools, human interpretation remains essential for judging the depth and validity of tool capabilities.

The DroidAgent sensitivity study (Section 4.3) illustrates these limitations in practice, showing how configuration changes and nondeterminism directly affect observed behavior in LLM-based systems.

Together, these insights establish a foundation for the discussion that follows, which explores how the field—and the taxonomy itself—must likely evolve to address these challenges and to guide the development of more reliable, scalable, and adaptive LLM testing methodologies toward 2030.

5 Discussion: Towards 2030

The taxonomy proposed in this paper offers a conceptual foundation for reasoning about how to test systems built around or including large language models, while also revealing areas where current practice lags behind. Beyond organizing existing practices, it exposes systematic gaps between how such systems behave in reality and how they are currently tested. Building on this foundation, we outline a staged research and tooling agenda—near-term, mid-term, and long-term—that connects immediate methodological needs with longer-term shifts in engineering practice and societal expectations.

5.1 Near-term: Guidance for testing frameworks for LLM applications

Although this work introduces a conceptual taxonomy rather than a full testing framework, it implicitly proposes a concrete research agenda on the key next steps: formalizing the notion of *probabilistic correctness* in the context of LLM application testing, defining standard metrics for *aggregated* verdicts, and building toolkits that support *hybrid human/LLM oracle workflows*. Broader validation will also be necessary, through replication across domains, longitudinal tracking of variability over time, and empirical benchmarks that measure aggregated oracle performance under real-world conditions.

Our analyses of existing tools and frameworks expose a mismatch between what is measured and what actually varies in such systems. Notably, (1) most tools treat each test execution as an isolated event, relying on deterministic or LLM-judged atomic oracles, and (2) correctness is implicitly assumed to be absolute, despite the inherent stochasticity of these systems.

As near-term guidance, we envision extending the taxonomy into a stepwise test design methodology, guiding practitioners from test goal specification to oracle selection and aggregation strategies. Aggregated oracles bridge deterministic testing and stochastic behaviour by replacing binary pass/fail judgments with confidence estimates and distributions of outcomes. Traditional testing already employs implicit aggregation—regression suites track pass/fail rates over time, CI/CD pipelines monitor flakiness, and developers interpret trends across multiple runs. Our contribution is not to introduce aggregation as a new idea, but to argue that it should become a first-class concern in LLM testing frameworks.

Whereas traditional tools aggregate results retrospectively and informally, LLM application testing requires deliberate aggregation strategies that acknowledge model non-determinism rather than treating it as a defect.

Future work could refine this distinction between *temporal aggregation*, combining results across repeated runs, and *structural aggregation*, combining results across models, prompts, or input variants. Existing research on probabilistic software properties can be revisited [15], yet open challenges remain regarding optimal sample size, cost and accuracy trade-offs, and how to convey probabilistic findings clearly to human stakeholders.

While frameworks such as Opik, Promptfoo, and DeepEval begin to incorporate aspects of variability-aware testing, their metrics and datasets still reflect traditional, output-oriented paradigms. A key opportunity lies in developing evaluation ecosystems where humans and LLMs jointly serve as oracles. In these hybrid settings, human judgment provides grounding, while LLM-based evaluators can contribute scale and adaptivity. Understanding how to calibrate, govern, and ensure transparency in such human-in-the-loop testing will be essential.

A remaining open question is how to design user interfaces and interaction patterns that enable domain experts and other stakeholders beyond traditional software testers to effectively participate in testing, as they possess the contextual knowledge necessary to validate LLM behaviour in specialized domains. This shift may introduce new challenges in maintaining consistent evaluation standards and in training non-technical stakeholders to interpret probabilistic test outcomes effectively. Questions of bias, fairness, and accountability arise when automated judges themselves may vary in output and vary differently for different sets of inputs, reinforcing the need for auditability and explainability of probabilistic verdicts.

5.2 Mid-term: Rethinking testing workflows under software evolution

Testing is increasingly entangled with development in LLM-based systems, representing a fundamental shift from traditional software engineering workflows. Where testing once served mainly as a post-development validation step, it can now play an important role throughout the entire lifecycle of LLM systems, from initial prompt engineering and model selection to iterative refinement and deployment. Continuous testing paradigms, where evaluation is embedded into every stage of system evolution, may offer valuable insights for managing this entanglement. Understanding the how-to will be critical.

This integration, operationalizing testing not merely as a gatekeeper but as a co-design mechanism, also bears a conceptual resemblance to test-driven development (TDD); but it departs from classical TDD process in several key aspects: (1) specifications in LLM-based applications rarely admit a one-to-one mapping to tests, making it impractical to assume a fully correct implementation upfront; (2) test outcomes are typically non-binary and require interpretation rather than simple pass–fail judgments; and (3) prompts and test oracles must therefore be iteratively refined and allowed to evolve alongside the system as developers reconsider what constitutes acceptable behaviour.

SUT variability, stemming from differences in model versions, configurations, and interaction contexts, adds another dimension of instability that current methods seldom capture. The DroidAgent empirical investigation (Section 4.3) illustrates how minor configuration changes can dramatically alter observed behaviour, emphasizing the need for regression testing that spans both versions and parameterizations. Future work should explicitly model such drift to determine when and why system behavior degrades. Statistical methods such as early stopping [39] and testing techniques like property-based testing [8] may help balance cost and confidence. Developing adaptive selection strategies that automatically adjust oracle types, sample sizes, and evaluation criteria based on observed variability would further enhance test automation, ensuring that test suites evolve alongside the systems they evaluate while minimizing testing costs.

5.3 Long-term: Societal implications and redefining correctness

Beyond methodology and tooling, the shift toward probabilistic correctness carries broader societal implications. As AI-based systems take on roles with higher stakes, accountability and fairness in their evaluation become more central concerns. Variability-aware testing frameworks could enhance transparency and trust, but they also

demand clarity about who is responsible when outcomes are probabilistic rather than deterministic. Making these uncertainties explicit can be an important step toward more honest and explainable software systems where AI models are key components.

We argue that progress toward mature LLM testing will depend on close collaboration between academia, industry, and policymakers. Open-source ecosystems already provide a foundation for standardization, but maintaining their relevance requires shared benchmarks, consistent terminology, and transparent reporting of stochastic outcomes—efforts that demand coordinated engagement across all three sectors. Researchers can distill empirical insights into generalizable principles, while practitioners validate them in operational environments, and policymakers can help establish governance frameworks that ensure accountability and ethical deployment. Even if these more traditional roles seem to increasingly blur as AI systems become more deeply embedded in society, we argue they will all continue to be relevant.

By recognising ambiguity and variability as *intrinsic properties* rather than flaws, the community can move toward a more realistic and principled understanding of correctness in LLM-based systems. The proposed taxonomy can offer a shared vocabulary and some direction for this transition, one that connects traditional software testing with the emerging realities of probabilistic, adaptive, and human-interactive intelligent systems.

References

[1] Berk Atil, Alexa Chittams, Liseng Fu, Ferhan Ture, Lixinyu Xu, and Breck Baldwin. 2024. LLM Stability: A detailed analysis with some surprises. *arXiv preprint arXiv:2408.04667* (2024).

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[3] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542.

[4] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[5] Shuhao Chen, Weisen Jiang, Baijiong Lin, James Kwok, and Yu Zhang. 2024. RouterDC: Query-Based Router by Dual Contrastive Learning for Assembling Large Language Models. *Advances in Neural Information Processing Systems* 37 (Dec. 2024), 66305–66328.

[6] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. 2004. Metamorphic Testing and Beyond. In *Proceedings of the International Workshop on Software Technology and Engineering Practice (STEP 2003)*. 94–100.

[7] Young Min Cho, Sunny Rai, Lyle Ungar, João Sedoc, and Sharath Guntuku. 2023. An Integrative Survey on Mental Health Conversational Agents to Bridge Computer Science and Medical Perspectives. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 11346–11369.

[8] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

[9] Comet. 2025. Opik: Open-source LLM Evaluation Framework. https://github.com/comet-ml/opik accessed: 2025-02.27.

[10] Felix Dobslaw and Robert Feldt. 2023. Similarities of Testing Programmed and Learnt Software. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 78–81.

[11] Felix Dobslaw, Robert Feldt, Juyeon Yoon, and Shin Yoo. 2025. Challenges in Testing Large Language Model Based Software: A Faceted Taxonomy. *arXiv preprint arXiv:2503.00481* (2025).

[12] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards autonomous testing agents via conversational large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1688–1693.

[13] Giskard. 2025. Giskard. https://docs.giskard.ai/hub/ui/index.html accessed: 2025-10.19.

[14] Giskard Team. 2024. Giskard: Secure Your LLM Agents. https://www.giskard.ai accessed: 2025-02.27.

[15] Lars Grunske. 2008. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering*. 31–40.

[16] Lars Grunske and Pengcheng Zhang. 2009. Monitoring probabilistic properties. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 183–192.

[17] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. 2024. A Survey on LLM-as-a-Judge. *arXiv preprint arXiv:2411.15594* (2024).

[18] Junda He, Christoph Treude, and David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* (2024).
[19] Sinclair Hudson, Sophia Jit, Boyue Caroline Hu, and Marsha Chechik. 2024. A Software Engineering Perspective on Testing Large Language Models: Research, Practice, Tools and Benchmarks. *arXiv preprint arXiv:2406.08216* (2024).
[20] Janina Kaarre, Robert Feldt, Bálint Zsidai, Eric Hamrin Senorski, Emilia Möller Rydberg, Olof Wolf, Sebastian Mukka, Michael Möller, and Kristian Samuelsson. 2024. ChatGPT can yield valuable responses in the context of orthopaedic trauma surgery. *Journal of Experimental Orthopaedics* 11, 3 (2024), e12047.
[21] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-based Explainable Fault Localization. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1424–1446. Issue FSE.
[22] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *Journal of Empirical Software Engineering* 30, 45 (2025), 1–28.
[23] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2024. Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2677–2694.
[24] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. 2312 – 2323.
[25] Arsham Gholamzadeh Khoee, Yinan Yu, Robert Feldt, Andris Freimanis, Patrick Andersson Rhodin, and Dhasarathy Parthasarathy. 2024. GoNoGo: An Efficient LLM-Based Multi-agent System for Streamlining Automotive Software Release Decision-Making. In *IFIP International Conference on Testing Software and Systems*. Springer, 30–45.
[26] Muhammad Laiq and Felix Dobslaw. 2025. Automatic techniques for issue report classification: A systematic mapping study. *arXiv preprint arXiv:2505.01469* (2025).
[27] LangChain. 2025. LangSmith. https://www.langchain.com/langsmith accessed: 2025-02.27.
[28] Keming Lu, Hongyi Yuan, Runji Lin, Junyang Lin, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2024. Routing to the Expert: Efficient Reward-guided Ensemble of Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 1964–1974.
[29] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2024. Test Oracle Automation in the era of LLMs. *ACM Transactions on Software Engineering and Methodology* (2024).
[30] OWASP. 2025. OWASP Top 10 Vulnerabilities. https://owasp.org/www-project-top-ten/ accessed: 2025-02.27.
[31] Parea AI. 2025. Parea AI: Test and Evaluate your AI Systems. https://www.parea.ai/ accessed: 2025-02.27.
[32] Jiahuan Pei, Fanghua Ye, Xin Sun, Wentao Deng, Koen Hindriks, and Junxiao Wang. 2025. Conversational Education at Scale: A Multi-LLM Agent Workflow for Procedural Learning and Pedagogic Quality Assessment. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 2984–2997.
[33] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering* 25 (2020), 5193–5254.
[34] Confident AI Team. 2024. DeepEval: An Open-Source Framework for Evaluating AI Models. https://github.com/confident-ai/deepeval Accessed: 2024-11-27.
[35] PromptFoo Team. 2024. PromptFoo: The AI Prompt Testing Tool. https://www.promptfoo.dev/ Accessed: 2024-11-27.
[36] Trulens Team. 2025. TruLens: Evaluate and Track LLM Applications. https://www.trulens.org/ accessed: 2025-02.27.
[37] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology* 85 (2017), 43–59.
[38] Abraham Wald. 1945. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics* 16, 2 (1945), 117–186. doi:10.1214/aoms/1177731118
[39] Abraham Wald and Jacob Wolfowitz. 1948. Optimum character of the sequential probability ratio test. *The Annals of Mathematical Statistics* (1948), 326–339.
[40] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.
[41] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE.
[42] Weights & Biases. 2025. Weights & Biases. https://www.wandb.ai accessed: 2025-02.27.
[43] Michael Weiss, AndréGarcía Gómez, and Paolo Tonella. 2023. Generating and detecting true ambiguity: a forgotten danger in DNN supervision testing. *Empirical Software Engineering* 28, 6 (2023), 146.
[44] Mingxuan Xiao, Yan Xiao, Shunhui Ji, Hanbo Cai, Lei Xue, and Pengcheng Zhang. 2024. Automated Robustness Testing for LLM-based NLP Software. *arXiv preprint arXiv:2412.21016* (2024).

[45] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the International Conference on Learning Representation (ICLR 2023)*.

[46] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-driven mobile gui testing with autonomous large language model agents. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 129–139.

[47] Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. 2023. Large Language Model Cascades with Mixture of Thought Representations for Cost-Efficient Reasoning. In *The Twelfth International Conference on Learning Representations*.

[48] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. 1592–1604.

[49] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2023), 46595–46623.