

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



CS423: Software Testing

Seminar Report

Testing AI Software/Solutions

MAI XUÂN BÁCH	22125005
LÊ ĐỨC PHÚ	22125074
PHAN MINH QUANG	22125083
LÊ QUỐC VĂN	22125119

HO CHI MINH CITY, DECEMBER 2025



Contents

1 What is AI Software Testing	4
1.1 AI-Powered Testing Tools	4
1.2 Testing AI Systems	5
1.3 The Convergence of Both Approaches	5
2 Differences Between Traditional and AI Testing	6
2.1 Determinism vs. Probabilism	6
2.2 Rule-Based vs. Data-Driven Behavior	6
2.3 Debugging and Root Cause Analysis	7
2.4 Coverage and Completeness	7
2.5 Validation Criteria	8
2.6 Evolution and Continuous Learning	9
2.7 Architectural Testing Considerations	9
2.8 Summary	9
3 Categories of AI Software Testing	10
3.1 Functional Testing of AI Models	10
3.2 Performance and Efficiency Testing	10
3.3 Robustness and Adversarial Testing	11
3.4 Fairness and Bias Testing	11
3.5 Data Quality and Validation Testing	12
3.6 Integration Testing	12
3.7 Edge Case and Boundary Testing	13
3.8 Safety and Security Testing	13
3.9 Agent and Multi-Step Reasoning Testing	14
3.10 Summary	14
4 OpenAI Evals: A Framework for LLM Evaluation	14
4.1 What is OpenAI Evals	15
4.2 Architecture and Components	15
4.2.1 Eval Templates	15
4.2.2 Data Registry	16
4.2.3 Completion Functions	16
4.2.4 Logging and Analysis	17
4.3 How OpenAI Evals Works	17



4.3.1	Setup and Installation	17
4.3.2	Running Evaluations	18
4.3.3	Creating Custom Evaluations	18
4.4	Practical Applications	19
4.4.1	Model Selection	20
4.4.2	Prompt Engineering Validation	20
4.4.3	Regression Testing	20
4.4.4	Fine-Tuning Validation	21
4.5	The Broader Ecosystem	21



1 What is AI Software Testing

AI software testing encompasses two distinct but complementary domains: using AI to enhance traditional software testing, and testing AI systems themselves. Understanding this distinction is essential for grasping the full scope of the field.

1.1 AI-Powered Testing Tools

The first domain involves leveraging artificial intelligence and machine learning to improve conventional software testing processes^[?]. These AI-powered tools automate and optimize quality assurance activities that traditionally required significant manual effort.

Modern AI testing tools use machine learning algorithms to address common challenges in software testing:

- **Self-Healing Tests:** One of the most significant advances in AI-powered testing is the ability of tests to automatically adapt to changes in application UI. When a button's locator changes or an element moves on the page, AI algorithms detect these modifications and automatically update test scripts. This reduces test maintenance overhead dramatically, as tests that would previously break after UI updates can now heal themselves^[?].
- **Intelligent Test Generation:** AI can analyze application behavior, user patterns, and code structure to automatically generate test cases. Rather than developers manually writing every test scenario, AI systems can suggest or create test cases based on how users actually interact with applications in production^[?].
- **Predictive Analytics:** By analyzing historical data, defect patterns, and code complexity metrics, AI can predict which parts of an application are most likely to contain bugs. This enables teams to focus testing efforts on high-risk areas before issues reach production^[?].
- **Root Cause Analysis:** When tests fail, AI-powered systems can analyze logs, stack traces, and historical data to identify the likely cause. These systems can cluster related failures, identify flaky tests, and even suggest potential fixes^[?].

As of 2025, approximately 81% of development teams incorporate AI into their testing workflows^[?], reflecting the mainstream adoption of these techniques.



1.2 Testing AI Systems

The second, more complex domain involves testing AI systems themselves. When the software under test is an AI model or an application that incorporates machine learning, traditional testing approaches prove insufficient.

AI model testing is the procedure of evaluating an AI model to ensure it functions as intended, considering aspects such as performance, accuracy, fairness, robustness, and safety^[?]. This type of testing must address fundamental characteristics that distinguish AI systems from traditional software:

- **Probabilistic Behavior:** Unlike deterministic programs, AI models produce outputs based on probabilistic patterns learned from training data. The same input can yield different outputs depending on model parameters, random seeds, or even minor variations in floating-point arithmetic.
- **Black-Box Nature:** Deep neural networks and large language models contain millions or billions of parameters whose individual contributions to behavior are nearly impossible to interpret. Testing must verify correct behavior without being able to inspect or fully understand internal logic.
- **Data Dependency:** AI model behavior is fundamentally shaped by training data. Testing must evaluate not just the model but also data quality, distribution, and potential biases that affect model decisions.
- **Context Sensitivity:** Modern AI systems, especially large language models, produce responses that depend heavily on context, making correctness a nuanced concept where multiple different outputs may all be valid.

Testing AI systems requires evaluating outputs against specific assessment criteria such as accuracy, coherence, fairness, safety, and relevance based on the intended application purpose^[?]. This evaluation often cannot rely on simple pass/fail assertions but instead requires measuring performance across distributions of test cases and quantifying acceptable ranges of behavior.

1.3 The Convergence of Both Approaches

In practice, these two domains increasingly overlap. The tools used to test AI systems often themselves employ AI techniques. For example, LLM-as-a-judge approaches use one language model to evaluate the outputs of another. Modern testing frameworks combine traditional



software testing principles for deterministic components with AI-specific evaluation methods for model inference and decision-making^[?].

This convergence represents a fundamental shift in how we approach software quality assurance, requiring test engineers to understand both classical testing methodologies and the unique characteristics of probabilistic, learning-based systems.

2 Differences Between Traditional and AI Testing

The shift from testing traditional software to testing AI systems represents a fundamental paradigm change in quality assurance. While both aim to ensure software correctness and reliability, the nature of what is being tested and how testing is performed differs substantially.

2.1 Determinism vs. Probabilism

The most fundamental difference lies in the predictability of system behavior.

Traditional Testing: In conventional software, identical inputs produce identical outputs. A function that calculates the square root of 16 will always return 4. This determinism allows testers to write assertions that check for exact expected values. If a test fails, it indicates a clear defect that can be debugged by tracing through code execution.

AI Testing: AI systems, particularly those based on machine learning, exhibit probabilistic behavior^[?]. The same prompt given to a large language model can produce different responses across multiple invocations, even with identical parameters. A classification model's confidence in a prediction may vary slightly due to floating-point arithmetic or hardware differences. This probabilistic variance breaks the assumption that repeated inputs yield repeated outputs, making traditional assertion-based testing insufficient.

Testing AI systems requires accepting that "correctness" exists within a distribution of acceptable outputs rather than as a single expected value. Testers must evaluate whether responses fall within acceptable ranges, meet quality thresholds, or satisfy semantic requirements rather than exact matches.

2.2 Rule-Based vs. Data-Driven Behavior

How systems produce their behavior differs fundamentally between traditional and AI software.

Traditional Testing: Conventional software follows explicit rules defined in source code. A login function checks if a password matches stored credentials according to logic written by



developers. Testing focuses on verifying that these rules are implemented correctly, handling edge cases, and ensuring proper error handling.

AI Testing: AI systems learn patterns from training data rather than following explicit rules^[?]. An AI-powered fraud detection system doesn't contain rules like "flag transactions over \$10,000" but instead learned from historical examples of fraudulent behavior. Testing must evaluate whether the model learned appropriate patterns, whether it generalizes to new data, and whether training data biases affect predictions inappropriately.

This shift means AI testing must examine not just the model but the entire pipeline including data collection, preprocessing, feature engineering, and training procedures. A model's behavior is inseparable from the data it learned from.

2.3 Debugging and Root Cause Analysis

When tests reveal problems, the path to understanding and fixing those problems differs dramatically.

Traditional Testing: When a traditional software test fails, developers can step through code with a debugger, inspect variable values, and trace execution flow to identify the exact line where incorrect behavior originates. Stack traces point to specific functions. The cause of failure is localized and interpretable.

AI Testing: AI models, especially deep neural networks, are largely black boxes^[?]. When a model makes an incorrect prediction, there's no single line of code to blame. The behavior emerges from millions of learned parameters interacting in complex ways. Understanding why a language model generated a particular response or why a computer vision model misclassified an image requires specialized interpretability techniques, analysis of training data, and statistical investigation rather than traditional debugging.

This opacity makes AI failures harder to diagnose and fix. Often the solution isn't to change code but to retrain with different data, adjust model architecture, or modify training procedures.

2.4 Coverage and Completeness

The concept of test coverage, fundamental to traditional testing, transforms when applied to AI systems.

Traditional Testing: Code coverage metrics indicate what percentage of code paths have been executed by tests. Achieving high branch coverage provides confidence that most code has been exercised. The input space, while potentially large, is often manageable or can be partitioned into equivalence classes.



AI Testing: For AI systems, the input space is effectively infinite^[?]. A language model can receive any text in any language. An image classifier must handle any possible image. Traditional coverage metrics become meaningless. Instead, testing focuses on:

- **Distribution Coverage:** Ensuring test data represents the distribution of real-world inputs the model will encounter.
- **Behavioral Coverage:** Verifying the model handles different types of scenarios, edge cases, adversarial inputs, and corner cases.
- **Capability Coverage:** Testing that the model possesses required capabilities like reasoning, factual knowledge, or specific task performance.

Complete testing is impossible; instead, practitioners aim for representative sampling and continuous monitoring in production.

2.5 Validation Criteria

How we judge whether a test passes differs between paradigms.

Traditional Testing: Validation typically involves exact comparisons or precise numerical checks. A test asserts that a function returns exactly "hello world" or that a calculation produces 42. Binary pass/fail outcomes are clear and objective.

AI Testing: Many AI outputs require semantic evaluation rather than exact matching^[?]. When a chatbot answers "The capital of France is Paris," responses like "Paris is France's capital," "Paris," or "France's capital city is Paris" are all correct despite being different strings. Evaluation often requires:

- Semantic similarity measures
- LLM-as-a-judge approaches where another AI evaluates correctness
- Rubrics scoring multiple dimensions like relevance, coherence, accuracy, and safety
- Human evaluation for subjective qualities

This introduces subjectivity and complexity absent from traditional testing's objective assertions.



2.6 Evolution and Continuous Learning

The temporal nature of system behavior differs significantly.

Traditional Testing: Once traditional software passes testing and deploys, its behavior remains stable unless code changes. The same version produces the same behavior over time.

AI Testing: AI systems may evolve continuously^[?]. Models can be retrained with new data, fine-tuned based on production feedback, or updated to new versions. Each change can alter behavior in unpredictable ways. Additionally, even without retraining, model behavior can drift if the distribution of production inputs changes from what the model was trained on.

This requires continuous monitoring, A/B testing of model versions, and ongoing evaluation rather than one-time validation before deployment.

2.7 Architectural Testing Considerations

Testing strategies must adapt to different architectural layers.

Traditional Testing: Testing follows established patterns like unit tests for individual functions, integration tests for component interactions, and end-to-end tests for complete workflows. Each layer is tested with similar methodologies.

AI Testing: Modern AI applications consist of multiple layers with different testing needs^[?]:

- **System Shell:** API interfaces, input validation, and output formatting can use traditional testing approaches.
- **Orchestration Layer:** Prompt engineering, context management, and tool integration require hybrid approaches combining logic validation with semantic evaluation.
- **Model Inference Core:** The AI model itself requires specialized evaluation techniques like offline batch evaluation, agent simulations, and LLM-as-a-judge scoring.

Testing complexity increases progressively from outer shells to the AI core, requiring different methodologies at each layer. Traditional and novel approaches must coexist within the same system.

2.8 Summary

The differences between traditional and AI testing are not merely technical details but reflect fundamental divergences in how systems operate. Traditional testing validates explicit, deterministic, interpretable behavior defined by code. AI testing evaluates emergent, probabilistic, opaque behavior learned from data. This requires new methodologies, tools, and mental models



for software quality assurance. The next sections explore how the field has organized these new approaches into practical categories and frameworks.

3 Categories of AI Software Testing

The field of AI software testing can be organized into several categories based on what aspect of the AI system is being tested and what techniques are employed. Understanding these categories helps teams structure comprehensive testing strategies.

3.1 Functional Testing of AI Models

Functional testing verifies that an AI model performs its intended task correctly^[?]. This is the most fundamental category, ensuring the model solves the problem it was designed to address.

For classification models, functional testing evaluates whether the model correctly categorizes inputs. A spam detection model should identify spam emails and let legitimate emails through. Testing involves running the model on labeled test datasets and measuring accuracy, precision, recall, and F1 scores.

For generative models like large language models, functional testing is more nuanced. Tests evaluate whether the model generates appropriate, relevant, and accurate responses to prompts. This might involve:

- Question-answering tests where the model must provide factually correct information
- Task completion tests where the model must follow instructions correctly
- Conversation coherence tests for chatbots
- Code generation correctness for programming assistants

Functional testing establishes baseline capability, but correctness alone is insufficient for deployment.

3.2 Performance and Efficiency Testing

AI models must meet performance requirements for production deployment^[?]. This category examines:



- **Inference Latency:** How quickly the model produces predictions. Real-time applications like autonomous vehicles require millisecond-level responses, while batch processing systems may tolerate higher latency.
- **Throughput:** How many predictions the model can process per second. This determines whether the system can handle expected production load.
- **Resource Utilization:** Memory consumption, GPU usage, and computational requirements affect deployment costs and scalability.
- **Model Size:** Parameter count and storage requirements impact deployment options, especially for edge devices or mobile applications.

Performance testing often reveals trade-offs between model accuracy and efficiency, requiring teams to optimize or select appropriate architectures.

3.3 Robustness and Adversarial Testing

Robustness testing evaluates how AI models handle unusual, corrupted, or maliciously crafted inputs^[?].

Adversarial Testing: This involves creating slightly modified inputs specifically designed to deceive the model. For image classifiers, adversarial examples might add imperceptible noise that causes misclassification. For language models, adversarial prompts attempt to elicit harmful, biased, or incorrect responses. Testing how models resist these attacks reveals vulnerabilities before malicious actors exploit them.

Noise and Corruption Testing: Real-world inputs often differ from clean training data. Images may be blurry, compressed, or partially occluded. Text may contain typos, slang, or grammatical errors. Testing with noisy inputs reveals whether models gracefully degrade or fail catastrophically.

Out-of-Distribution Testing: This evaluates model behavior when encountering inputs significantly different from training data. A model trained on clear daytime photos should be tested on nighttime or foggy conditions. Language models trained primarily on English should be tested on code-switched text or non-standard dialects.

Robustness testing is critical for safety-critical applications and adversarial environments.

3.4 Fairness and Bias Testing

AI systems can perpetuate or amplify societal biases present in training data^[?]. Fairness testing evaluates whether models make equitable decisions across different demographic groups.



For predictive models used in hiring, lending, or criminal justice, fairness testing measures whether outcomes are distributed equitably across protected attributes like race, gender, or age. Common metrics include:

- Demographic parity: Ensuring similar positive prediction rates across groups
- Equal opportunity: Ensuring similar true positive rates across groups
- Predictive parity: Ensuring similar precision across groups

For language models, bias testing examines whether responses contain stereotypes, whether the model treats mentions of different groups differently, and whether outputs could cause harm to specific communities.

Fairness testing often reveals uncomfortable truths about training data and requires difficult decisions about acceptable trade-offs between different fairness definitions.

3.5 Data Quality and Validation Testing

Since AI model behavior is fundamentally determined by training data, testing the data itself is essential^[?].

Data Integrity Testing: Verifies that data is complete, correctly formatted, and free from corruption. This includes checking for missing values, duplicate records, and data type inconsistencies.

Distribution Testing: Ensures training, validation, and test datasets have appropriate distributions and that training data represents the real-world scenarios the model will encounter.

Label Quality Testing: For supervised learning, incorrect labels in training data directly cause model errors. Testing involves auditing label quality, checking annotator agreement, and identifying mislabeled examples.

Data Drift Detection: In production, monitoring whether incoming data distribution matches what the model was trained on reveals when retraining is necessary.

Tools like Deepchecks provide comprehensive suites for data validation throughout the machine learning lifecycle.

3.6 Integration Testing

AI models rarely operate in isolation. Integration testing verifies how models interact with surrounding software components^[?].

This includes testing:



- API contracts between model serving infrastructure and applications
- Data preprocessing pipelines that prepare inputs for the model
- Postprocessing logic that interprets model outputs
- Multiple models working together (ensemble methods or multi-agent systems)
- Fallback mechanisms when models fail or produce low-confidence predictions

Integration testing is particularly important when different teams develop models and applications independently, or when multiple AI models with potentially conflicting objectives must cooperate.

3.7 Edge Case and Boundary Testing

Edge case testing examines AI model behavior in rare or extreme scenarios^[?]. Unlike traditional software where edge cases involve boundary values of input ranges, AI edge cases involve unusual combinations of features or rare situations.

For example:

- A medical diagnosis model encountering a patient with multiple rare conditions simultaneously
- A language model receiving prompts combining multiple languages and domains
- An autonomous vehicle encountering construction zones in severe weather

Since AI models learn from data, rare scenarios are underrepresented in training, often causing poor performance on edge cases. Systematic edge case testing identifies these gaps before they cause problems in production.

3.8 Safety and Security Testing

For AI systems in critical applications, safety testing ensures the model cannot cause harm even when malfunctioning^[?].

Safety Testing: Verifies that models have appropriate guardrails, that outputs cannot cause physical harm, and that the system fails safely when encountering situations beyond its capabilities. For autonomous vehicles or medical AI, this testing is mandatory and regulated.

Security Testing: Examines vulnerabilities like model inversion attacks (where attackers reconstruct training data), model extraction (stealing model functionality), prompt injection



(manipulating language model behavior), and data poisoning (corrupting training data to introduce backdoors).

The OWASP AI Testing Guide provides frameworks for systematically evaluating AI security vulnerabilities.

3.9 Agent and Multi-Step Reasoning Testing

Modern AI applications increasingly involve agents that take multiple actions, use tools, and pursue goals over extended interactions^[?]. Testing these systems requires new approaches:

Agent Simulations: Rather than testing individual responses, simulations create realistic goal-driven scenarios where agents must plan, adapt, and reason over time. Testing evaluates whether agents achieve objectives, handle errors appropriately, and make sound decisions throughout interaction sequences.

Tool Usage Testing: When agents can call external APIs, run code, or access databases, testing must verify they use tools appropriately, handle tool failures, and don't cause unintended side effects.

State and Memory Testing: For agents that maintain conversation history or world state, testing examines whether they correctly update and use state information across interactions.

This category represents the frontier of AI testing, addressing systems that are increasingly autonomous and complex.

3.10 Summary

These categories are not mutually exclusive but represent different lenses through which to examine AI systems. Comprehensive testing strategies employ multiple categories, prioritizing based on application requirements, risk profiles, and deployment contexts. The next section examines OpenAI Evals, a practical framework that implements many of these testing approaches for large language models.

4 OpenAI Evals: A Framework for LLM Evaluation

As large language models have become central to AI applications, the need for systematic evaluation frameworks has grown critical. OpenAI Evals emerged as an open-source solution to this challenge, providing both a framework for evaluating LLMs and a registry of standardized benchmarks. This section examines Evals in detail, exploring its architecture, capabilities, and practical application.



4.1 What is OpenAI Eval

OpenAI Eval is a framework for evaluating large language models or systems built using LLMs, accompanied by an open-source registry of benchmarks^[?]. It addresses a fundamental problem in LLM development: without systematic evaluation, understanding how different model versions affect specific use cases becomes difficult and time-intensive.

The framework serves multiple purposes:

- **Standardized Evaluation:** Provides consistent methodologies for measuring model performance across different dimensions
- **Benchmark Repository:** Offers pre-built evaluations that test OpenAI models on established tasks
- **Custom Evaluation Creation:** Enables developers to build evaluations tailored to their specific use cases
- **Private Testing:** Allows evaluation using proprietary data without exposing it publicly
- **Comparative Analysis:** Facilitates comparing different models, prompts, or fine-tuned versions

The philosophy behind Eval reflects a core insight: creating quality evaluations is one of the most impactful practices for LLM builders. By making evaluation systematic and reproducible, Eval helps teams make informed decisions about model selection, prompt engineering, and system design.

4.2 Architecture and Components

Eval consists of several interconnected components that work together to support comprehensive evaluation.

4.2.1 Eval Templates

Templates provide predefined patterns for common evaluation scenarios. The framework includes multiple template types:

Basic Evaluations: These use exact string matching or simple criteria to determine correctness. A basic eval might test factual knowledge by comparing model outputs against known correct answers. For example, asking "What is the capital of France?" and checking if the response contains "Paris."



Model-Graded Evaluations: More sophisticated evaluations use another LLM to judge the quality of outputs. This approach handles cases where multiple valid responses exist or where semantic understanding is required. The grading model receives both the prompt and response, then scores quality based on specified criteria.

Templates abstract common evaluation patterns, allowing developers to create new evaluations by providing data rather than implementing evaluation logic from scratch.

4.2.2 Data Registry

Evaluation data is stored in a structured registry using Git-LFS for version control and efficient storage of large datasets. Data files use JSON format with standardized schemas.

A typical evaluation dataset consists of test cases, each containing:

- Input prompts or questions
- Expected outputs or correct answers
- Metadata like difficulty level, category, or source
- Evaluation criteria or grading rubrics

The registry's version control enables tracking how model performance changes over time and sharing benchmarks across teams while maintaining data provenance.

4.2.3 Completion Functions

While basic evaluations test single LLM calls, real applications often involve complex workflows. Completion functions allow testing more sophisticated scenarios:

- **Prompt Chains:** Sequences where one model output becomes input to the next prompt
- **Agent Systems:** Autonomous agents that use tools, maintain state, and pursue multi-step goals
- **Retrieval-Augmented Generation:** Systems that search knowledge bases before generating responses
- **Multi-Model Ensembles:** Applications combining outputs from multiple models

Completion functions wrap these complex behaviors, presenting them to the evaluation framework as testable units. This enables testing real application behavior rather than just isolated model responses.



4.2.4 Logging and Analysis

Evals provides flexible logging options for evaluation results. Results can be stored locally, logged to cloud services, or integrated with Snowflake databases for large-scale analysis.

The logging system captures:

- Individual test case results with model outputs
- Aggregate metrics like accuracy, pass rate, or average score
- Performance data including latency and token usage
- Comparison data when evaluating multiple models
- Metadata about evaluation configuration and environment

This comprehensive logging enables detailed post-evaluation analysis and trend tracking across model versions.

4.3 How OpenAI Evals Works

Using Evals involves several steps, from installation through execution and analysis.

4.3.1 Setup and Installation

Evals requires Python 3.9 or higher. Installation options include:

```
1 # For running existing evals
2 pip install evals
3
4 # For development and creating custom evals
5 git clone https://github.com/openai/evals
6 cd evals
7 pip install -e .
```

Listing 4.1: Installing OpenAI Evals

Configuration requires setting OpenAI API credentials via environment variables, as model calls during evaluation use the OpenAI API and incur associated costs.



4.3.2 Running Evaluations

Evaluations are executed via command-line tools. The basic pattern is:

```
1 oai eval gpt-4 <eval-name>
```

Listing 4.2: Running an evaluation

Where `gpt-4` specifies the model to test and `<eval-name>` references an evaluation from the registry.

For example, running a logical reasoning evaluation:

```
1 oai eval gpt-4 logical-reasoning
```

Listing 4.3: Example evaluation command

The framework then:

1. Loads the specified evaluation configuration and test data
2. Iterates through test cases, sending prompts to the model
3. Collects model responses
4. Evaluates responses against expected outputs using the eval's criteria
5. Aggregates results and computes metrics
6. Logs detailed results for analysis

Dashboard integration allows configuring and executing evaluations through a graphical interface rather than command-line tools, making the system accessible to non-technical users.

4.3.3 Creating Custom Evaluations

Building custom evaluations involves defining three components:

1. Test Data: Create a JSON file containing test cases:

```
1 {
2     "samples": [
3         {
4             "input": "Translate to Spanish: Hello, world",
5             "ideal": "Hola, mundo"
6         },
7         {
8             "input": "Translate to Spanish: Good morning",
```



```

9     "ideal": "Buenos dias"
10    }
11   ]
12 }
```

Listing 4.4: Example test data format

2. Evaluation Configuration:

Define how to evaluate responses:

```

1 id: spanish-translation
2 description: Tests Spanish translation capability
3 dataset: translation_test_data.json
4 eval_type: match
5 match_type: fuzzy
```

Listing 4.5: Evaluation configuration

3. Custom Grading Logic

(if needed): For complex evaluations requiring specialized logic, implement custom evaluation classes:

```

1 class CustomEval(Eval):
2     def eval_sample(self, sample, *args):
3         prompt = sample["input"]
4         expected = sample["ideal"]
5
6         # Get model response
7         response = self.completion_fn(prompt)
8
9         # Custom evaluation logic
10        score = self.calculate_similarity(
11            response, expected
12        )
13
14        return {
15            "score": score,
16            "pass": score > 0.8
17        }
```

Listing 4.6: Custom evaluation logic

This extensibility allows tailoring evaluations to specific application requirements that standard templates don't address.

4.4 Practical Applications

Evals serves multiple practical purposes in LLM application development.



4.4.1 Model Selection

When choosing between GPT-4, GPT-3.5, or fine-tuned variants, Evals enables data-driven decisions. By running the same evaluation against multiple models, teams can compare performance on their specific use case rather than relying on general benchmarks that may not reflect their application's requirements.

For example, a customer service chatbot might be evaluated on:

- Accuracy of product information
- Appropriate tone and empathy
- Handling of edge cases and complaints
- Consistent policy application

Running these evaluations against different models reveals which performs best for the specific task, considering both quality and cost.

4.4.2 Prompt Engineering Validation

Prompt engineering significantly affects LLM performance, but without systematic evaluation, it's difficult to know if prompt changes improve or degrade behavior. Evals enables:

- A/B testing different prompt formulations
- Quantifying improvement from few-shot examples
- Detecting when prompt changes help some cases but hurt others
- Building a library of validated prompts for different tasks

This replaces subjective assessment with quantitative measurement.

4.4.3 Regression Testing

As applications evolve, changes to prompts, context, or system design can inadvertently degrade performance. Evals functions as regression testing, ensuring changes don't break existing capabilities:

1. Establish baseline evaluation results for current system
2. Make changes to prompts, retrieval, or application logic



3. Re-run evaluations
4. Compare results to detect regressions

This prevents the common problem where improving performance on new requirements accidentally breaks previously working functionality.

4.4.4 Fine-Tuning Validation

When fine-tuning models on custom data, Eval validate whether fine-tuning improves performance:

- Evaluate base model performance
- Fine-tune on training data
- Evaluate fine-tuned model on held-out test set
- Compare metrics to determine if fine-tuning was beneficial

This is especially important because fine-tuning can sometimes overfit to training data or degrade general capabilities while improving specific task performance.

4.5 The Broader Ecosystem

OpenAI Eval is part of a growing ecosystem of LLM evaluation tools. Other frameworks include:

- **LangWatch**: Focuses on agent testing with simulations
- **Braintrust**: Provides evaluation platforms with advanced analytics
- **Confident AI**: Specializes in LLM testing methodologies
- **Patronus AI**: Emphasizes enterprise LLM testing

However, Eval benefits from OpenAI's extensive use in internal model development, ensuring it addresses real-world evaluation challenges at scale. Its open-source nature and active community also provide significant advantages for adoption and extension.