The background of the slide is a detailed, close-up photograph of a complex mechanical clockwork mechanism. It features numerous large and small gears, levers, and ornate metal components, all in a dark, aged brass or copper finish. The lighting is dramatic, highlighting the textures and metallic sheen of the parts. In the lower right foreground, an open book with cream-colored pages is placed on a wooden surface, possibly a desk or a part of the clockwork's base. The overall atmosphere is one of historical craftsmanship and intellectual pursuit.

Large Language Models

Evaluating Results

Riccardo Coppola

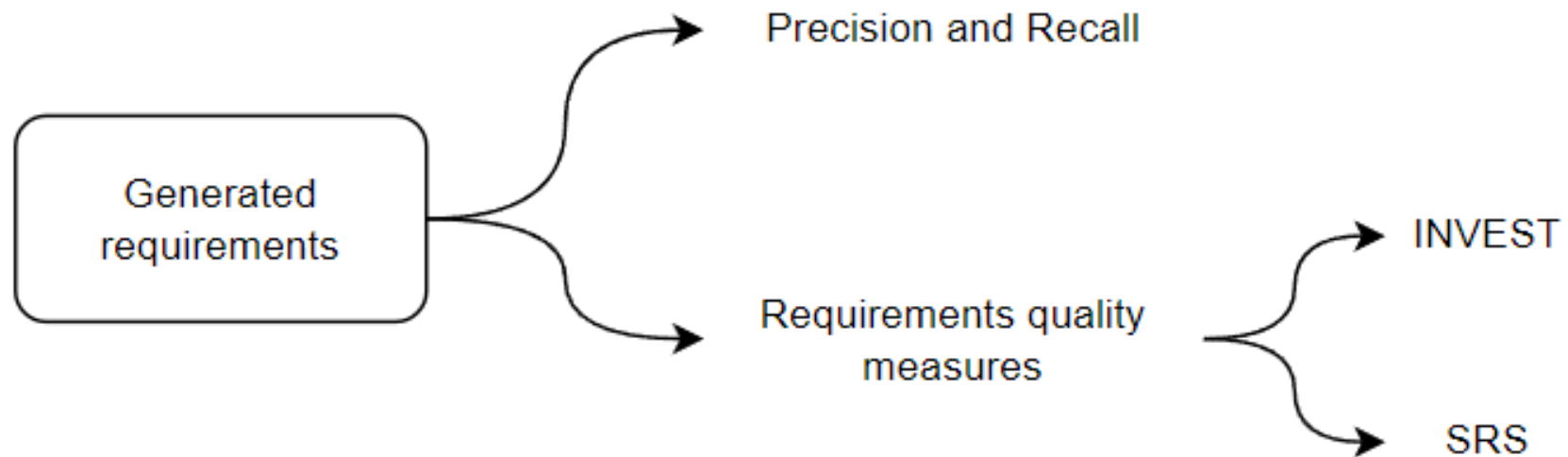
How to evaluate the results of LLMSE

- The evaluation of the results of an agent-based architecture depends:
 - On the type of deliverable that is created by the LLM (e.g., free text, structured text or tables, code, test cases...);
 - On the phase of Software Engineering that is addressed;
 - On the availability of a ground-truth to which the results can be compared.



Evaluating Requirements Extraction

Precision and Recall



Precision and Recall

- Precision and Recall are key metrics used to evaluate the quality of extracted requirements in **requirements engineering** tasks, especially when using **LLMs** to automatically generate or extract functional requirements from raw text.

Step 1: Manually or automatically review the extracted requirements

Step 2: Compare extracted requirements to the ground truth (actual reqs)

Step 3: Calculate precision, recall, and F1 score

Step 4: use the metrics to iterate on LLM model improvements

Precision and Recall

- **Precision** measures how many of the extracted requirements are actually correct (i.e., relevant and accurate).
- High precision means that the system generates fewer irrelevant or incorrect requirements.

$$\textit{Precision} = \frac{\textit{True Positive (TP)}}{\textit{True Positive (TP)} + \textit{False Positive (FP)}}$$

Precision and Recall

- **Recall** measures how many of the total relevant requirements were correctly extracted.
- High recall means that the system successfully identifies most of the relevant requirements, but may also include some irrelevant ones.

$$\text{Recall} = \frac{\text{True Positive (TP)}}{\text{True Positive (TP)} + \text{False Negative (FN)}}$$

Precision and Recall: Example

- **Scenario:** Extracting requirements for the role of a Visitor in a hiking platform.
- **Ground truth:**
 - Requirements: "Visitors can browse trails, view descriptions, and record fitness data."
- **Extracted by LLM:**
 - View trail descriptions.
 - Record fitness data.
 - Send push notifications to users.

Precision and Recall: Example

- **Scenario:** Extracting requirements for the role of a Visitor in a hiking platform.
- **Ground truth:**
 - Requirements: "Visitors can browse trails, view descriptions, and record fitness data."
- **Extracted by LLM:**
 - View trail descriptions.
 - Record fitness data.
 - Send push notifications to users.

True Positives (TP) = 2

Precision and Recall: Example

- **Scenario:** Extracting requirements for the role of a Visitor in a hiking platform.
- **Ground truth:**
 - Requirements: "Visitors can browse trails, view descriptions, and record fitness data."
- **Extracted by LLM:**
 - View trail descriptions.
 - Record fitness data.
 - Send push notifications to users.

False Positives (FP) = 1

Precision and Recall: Example

- **Scenario:** Extracting requirements for the role of a Visitor in a hiking platform.
- **Ground truth:**
 - Requirements: "Visitors can browse trails, view descriptions, and record fitness data."
- **Extracted by LLM:**
 - View trail descriptions.
 - Record fitness data.
 - Send push notifications to users.

False Negatives (FN) = 1

Precision and Recall: Example

- Calculation:

- $$Precision = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Positive\ (FP)} = \frac{2}{2+1} = 0.67$$

- $$Recall = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Negative\ (FN)} = \frac{2}{2+1} = 0.67$$

Precision and Recall: Example

- There is often a trade-off between precision and recall. Increasing one may reduce the other.
- **F1 Score:** Combines precision and recall into a single metric, providing a balance.
- A high F1 score indicates a good balance between precision and recall.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Precision and Recall: Example

- Calculation:

- $$Precision = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Positive\ (FP)} = \frac{2}{2+1} = 0.67$$

- $$Recall = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Negative\ (FN)} = \frac{2}{2+1} = 0.67$$

- $$F1Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} = 2 \times \frac{0.67 \times 0.67}{0.67 + 0.67} = 0.67$$

Precision and Recall: The challenge of variation

- **Issue:** Different phrasing, synonyms, and contextual differences in how requirements are written make automatic evaluation difficult.
 - Example: "View trail descriptions" vs. "Browse hiking trails" vs. "See details of available trails."

Precision and Recall: The challenge of variation

- **Solutions:**

- **Predefined synonym list:** Create a list of synonyms specific to your domain (e.g., "browse" = "view", "details" = "descriptions"). Use this list to map different words to the same concept.
- **Word Embeddings:** These models represent words as vectors in a high-dimensional space, where semantically similar words are closer together. You can use these embeddings to detect words with similar meanings automatically, even if they are not exactly the same.
- **Sentence Embeddings (BERT, SBERT):** These models can capture the contextual meaning of sentences and can compare if two different phrases express the same idea.

Precision and Recall: The challenge of variation

- **Solutions:**

- **Text Preprocessing:** These techniques reduce words to their base form, which can help standardize the vocabulary.
- Stemming: "Viewing" → "View", "Browsed" → "Browse"
- Lemmatization: "View" → "View", "Views" → "View"

Precision and Recall: Example with cosine similarity

- We obtain all the word embeddings of the generated requirements and we compute the cosine similarity with all the expected requirements

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^n (a_i b_i)}{\sqrt{\sum_{i=1}^n a_i^2} \sqrt{\sum_{i=1}^n b_i^2}}$$

Precision and Recall: Example with cosine similarity

- We evaluate requirement similarity as a binary classification task over all pairs (extracted requirement A, actual requirement b):
- Prediction:
 - MATCH if $cs(A,B) \geq T$
 - NO MATCH if $cs(A,B) < T$
- Ground truth:
 - MATCH if A truly corresponds to B
 - NO MATCH otherwise
- True Positive (TP): $cs(A,B) \geq T$ and A truly corresponds to B
- False Positive (FP): $cs(A,B) \geq T$ but A does NOT correspond to B
- True Negative (TN): $cs(A,B) < T$ and A does NOT correspond to B
- False Negative (FN): $cs(A,B) < T$ but A actually corresponds to B

Precision and Recall: Example with cosine similarity

	Browse trails	View descriptions	Record fitness data
View trail descriptions	0.80	0.92	0.45
Record fitness data	0.33	0.45	1.0
Send push notifications to users	0.21	0.32	0.18

T=0.9

Precision and Recall: Example with cosine similarity

Ground truth match

Ground truth no match

	Browse trails	View descriptions	Record fitness data
View trail descriptions	0.80	0.92	0.45
Record fitness data	0.33	0.45	1.0
Send push notifications to users	0.21	0.32	0.18

T=0.9

Precision and Recall: Example with cosine similarity

Ground truth match

Ground truth no match

	Browse trails	View descriptions	Record fitness data
View trail descriptions	0.80 (TN)	0.92 (TP)	0.45 (TN)
Record fitness data	0.33 (TN)	0.45 (TN)	1.0 (TP)
Send push notifications to users	0.21 (TN)	0.32 (TN)	0.18 (TN)

T=0.9

Precision and Recall: Example with cosine similarity

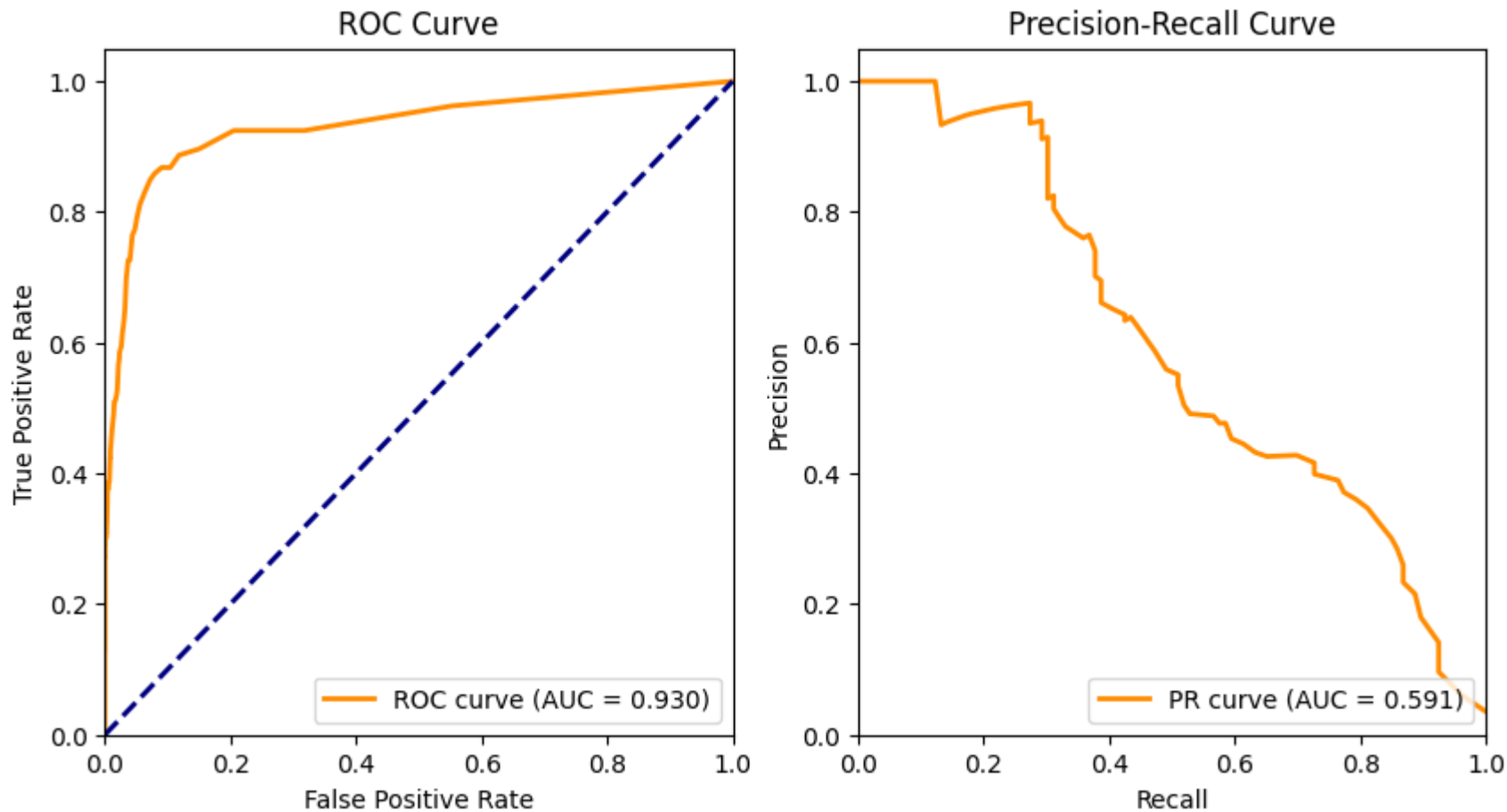
- I can study what happens at different thresholds by studying the ROC curve, which plots the following measures:
 - True Positive Rate (TPR), is the proportion of actual positives that are correctly identified.

$$TPR = \frac{TP}{TP + FN} (= Recall)$$

- False Positive Rate (FPR) is the proportion of actual negatives that are incorrectly identified as positives.

$$FPR = \frac{FP}{FP + TN}$$

Precision and Recall: Example with cosine similarity



Precision and Recall: Example with cosine similarity

- While cosine similarity can be automated, it cannot perfectly understand nuanced differences or context. In such cases:
 - **Manual review** of the results may still be necessary, especially for critical requirements or when precision is paramount. This might involve human reviewers checking the top-ranked results.
 - **Crowdsourcing** or using a group of domain experts to validate matches can help in reducing false positives.

Evaluation of the quality of requirements

- In addition of Precision and Recall, it might be important to evaluate the quality of requirements.
- Typically this is done through questionnaires by having humans in the loop (or, you can simulate the human reader with another LLM agent...)

Example: INVEST

- **INVEST**: administer a questionnaire to ask whether each generated requirement (or user story) is:
 - Independent: Check if the requirement stands alone.
 - Negotiable: Ensure the requirement is open to change and discussion.
 - Valuable: Ensure the requirement adds tangible value to users or stakeholders.
 - Estimable: Check if the requirement is clear enough for estimation.
 - Small: Verify that the requirement is small and actionable.
 - Testable: Ensure the requirement can be tested and validated.

Example: INVEST

	I	N	V	E	S	T	Score
As a visitor, I want to filter hiking trails based on difficulty so I can find trails that match my fitness level.	2	2	2	2	2	2	14/14
As a local guide, I want to add new trails to the platform so visitors can explore more hiking options.	2	1	2	1	1	2	9/14
As a platform manager, I want to broadcast weather alerts for trails so that hikers can stay informed about potential dangers.	2	1	2	2	2	2	13/14

Example: SRS quality measures

- **Per-requirement grading:**

- Unambiguous: A requirement is unambiguous if and only if it has only one possible interpretation.
- Understandable: A requirement is understandable if all classes of SRS readers can easily comprehend its meaning with a minimum of explanation.
- Correct: A requirement is deemed correct when it accurately represents a required feature or function the system must possess.
- Verifiable: A requirement is verifiable if finite, costeffective techniques exist for verifying that it is satisfied by the system as built.

Example: SRS quality measures

- **Document-wide grading:**

- Internal Consistency: An SRS is internally consistent if and only if no subsets of individual requirements conflict.
- Non-redundancy: An SRS is not redundant if no requirement is restated more than once.
- Completeness: An SRS is complete if it details all functions, describes all responses, provides organizational clarity, and avoids placeholder text.
- Conciseness: An SRS is concise when it delivers all necessary information briefly without sacrificing its quality.

Example: SRS quality measures

	Unambiguous	Understandable	Correct	Verifiable	Score
As a visitor, I want to filter hiking trails based on difficulty so I can find trails that match my fitness level.	2	5	5	4	16/20
As a local guide, I want to add new trails to the platform so visitors can explore more hiking options.	4	5	5	3	17/20
As a platform manager, I want to broadcast weather alerts for trails so that hikers can stay informed about potential dangers.	3	4	5	2	14/20

Example: SRS quality measures

SRS

As a visitor, I want to filter hiking trails based on difficulty so I can find trails that match my fitness level.

As a local guide, I want to add new trails to the platform so visitors can explore more hiking options.

As a platform manager, I want to broadcast weather alerts for trails so that hikers can stay informed about potential dangers.

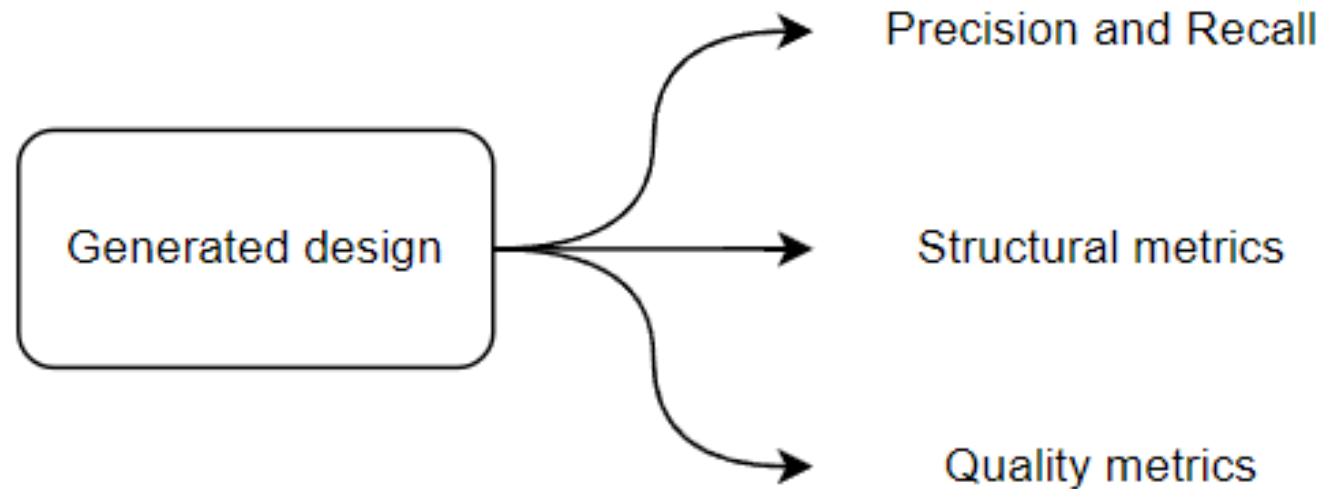
...

Parameter	Score
Internal consistency	4
Non-redundancy	5
Completeness	1
Conciseness	4
Overall score	14/20



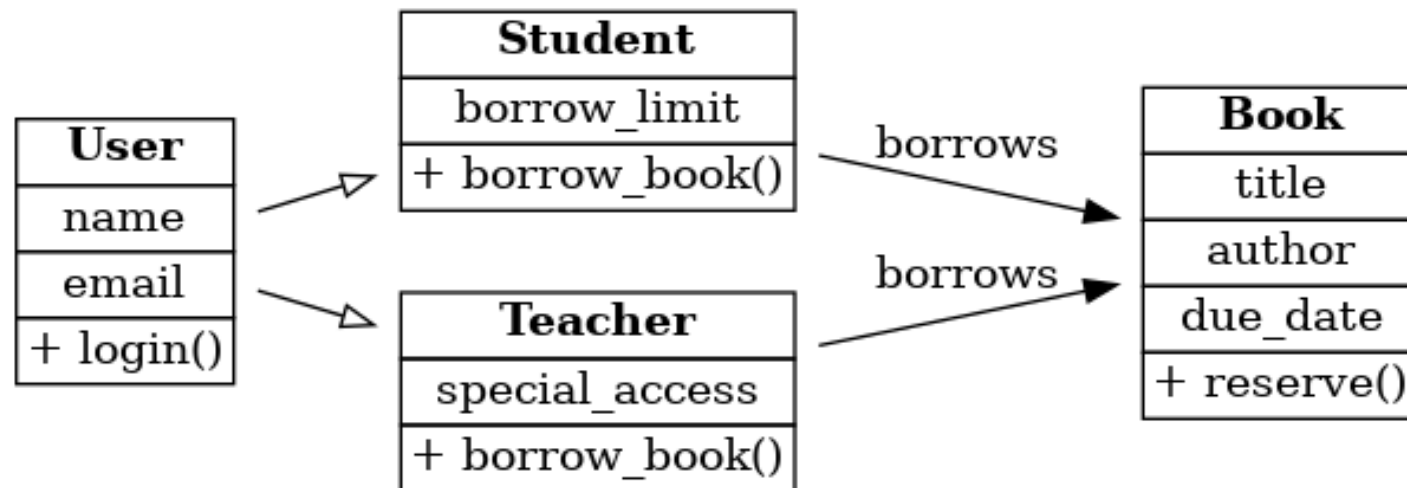
Evaluating Design

Evaluating Design



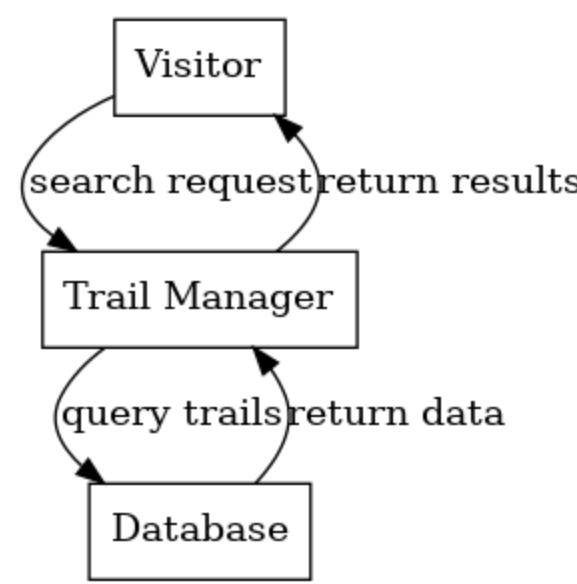
How are LLMs used in design

- **Class Diagrams:** Represent the structure of the system by showing classes, attributes, methods, and relationships.



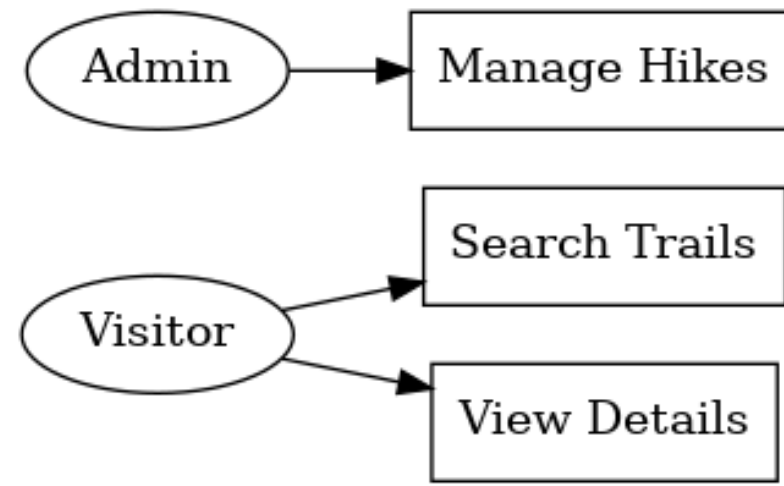
How are LLMs used in design

- **Sequence Diagrams:** Show the flow of messages or interactions between system components over time.



How are LLMs used in design

- **Use Case Diagrams:** Highlight user interactions with the system, showcasing actors and their use cases.



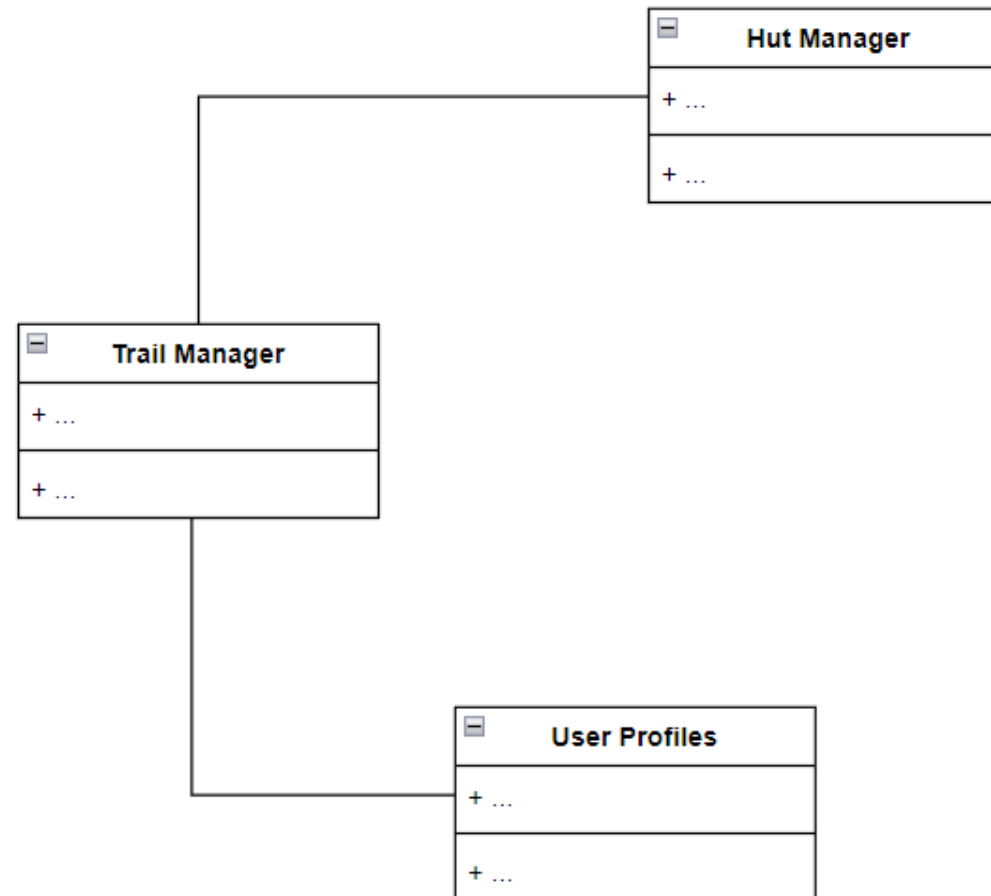
Structural metrics

- Structural metrics refer to measurements that focus on the design and architecture of a system, specifically how different components, modules, or classes are organized and interact.
- These metrics help identify design flaws early in the development process, ensuring a scalable and maintainable system.
 - **Missing Dependencies:** A design violation occurs when a component is dependent on another component, but that dependency is not properly established or is omitted in the system's design.
 - **Missing Dependency Count** = Total Expected Dependencies - Total Established Dependencies
 - **Other Violations:** Misplaced dependencies, circular dependencies, incorrect inheritance, or incorrect interface implementation.

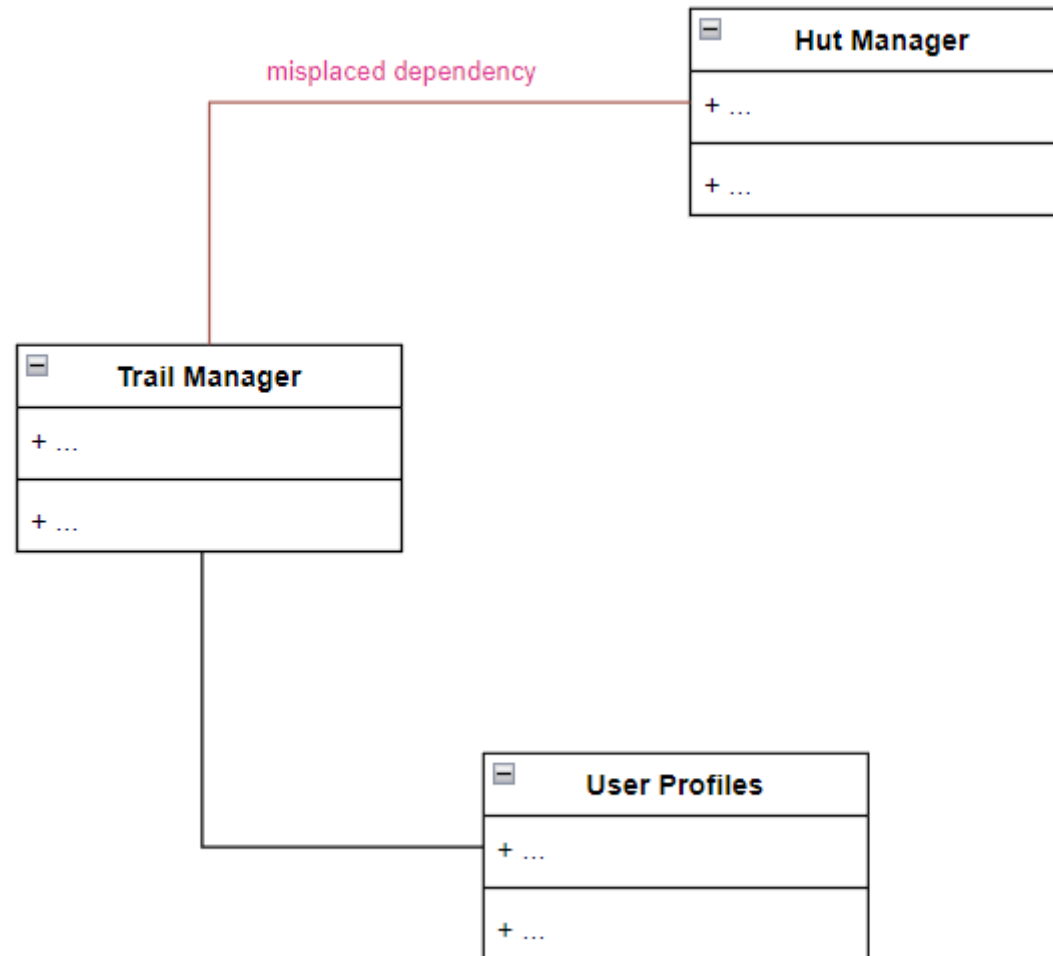
Structural metrics: Example

- **Trail Manager** depends on **Hut Manager** to get information about nearby huts.
- **Hut Manager** is expected to retrieve data from **User Profiles** to understand visitor preferences.

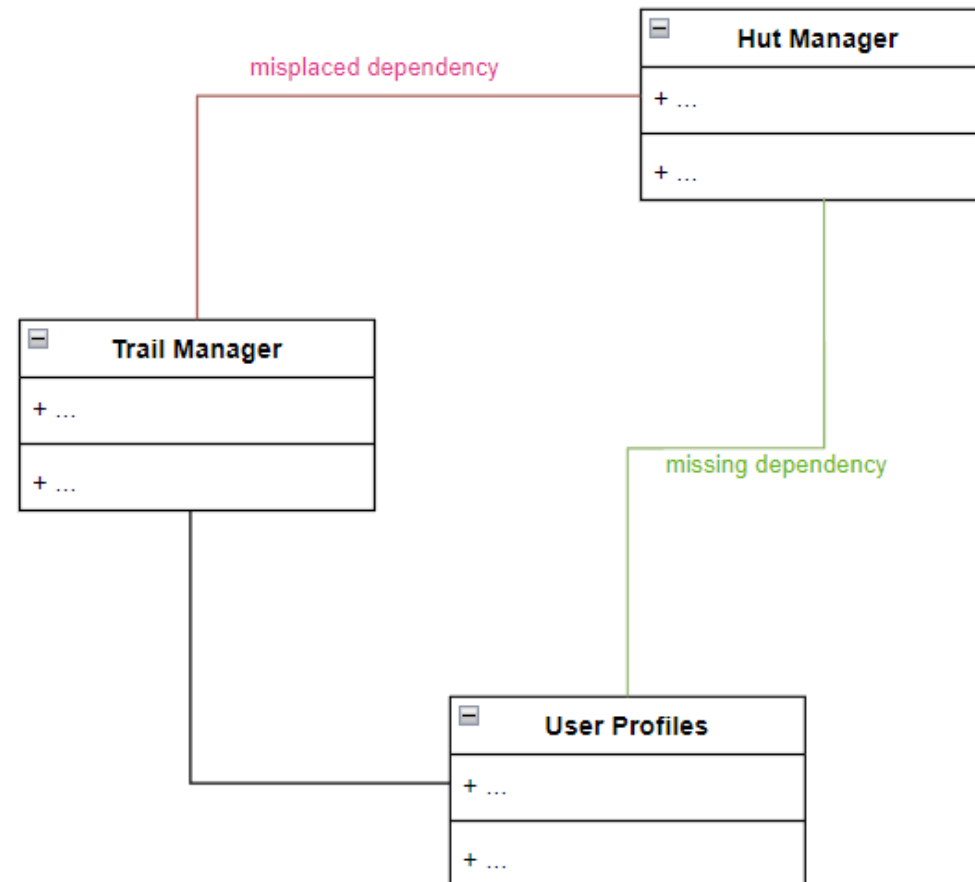
Structural metrics: Example



Structural metrics: Example



Structural metrics: Example



Precision and Recall

- In the same way we did with the requirements
- Collect all constructs generated by the LLM (it depends on the type of diagram) and compare them with a ground truth
 - (possibly implying similarity and thresholds...)

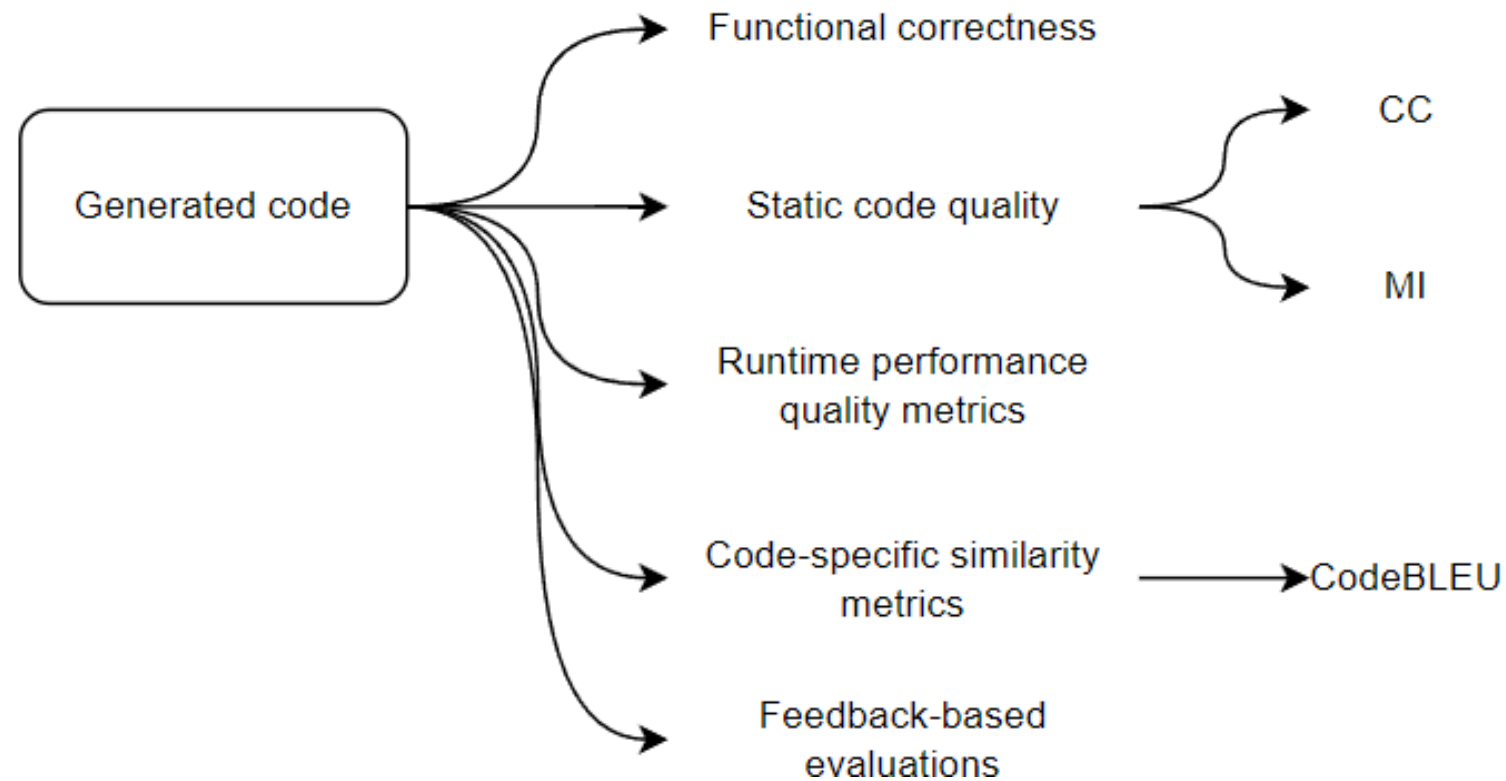
Quality issues with design (Bolloju et al.)

	Syntactic Quality	Semantic Quality	Pragmatic Quality
Use Case Models	<p>Inappropriate use case names (such as not beginning with an action verb)</p> <p>Invalid notation in use case diagram</p>	<p>Invalid relationship (include, extend, or generalization) between use cases</p> <p>Incomplete scenario description</p>	<p>Poor layout of use case diagram</p> <p>Presence of implementation details (such as user interface) in use case description</p>
Domain Models	<p>Missing cardinality details for associations</p> <p>Inappropriate naming of classes and associations</p>	<p>Incorrect cardinality specification</p> <p>Used aggregation in place of association</p>	<p>Redundant attributes and associations</p> <p>Specialization with little distinction among subclasses</p>
Dynamic Models	<p>Improper positioning of classes and/or objects along the timeline in sequence diagram</p>	<p>Incomplete specification of message parameters</p>	<p>Inappropriate delegation of responsibilities</p>



Evaluating Code generation

Evaluating Code Generation



Functional correctness

- **Objective:** Assess the accuracy of code generated by LLMs by determining how many test cases it successfully passes. This provides an empirical measure of functional correctness.
- **Warning:** I need a dependable test suite beforehand!

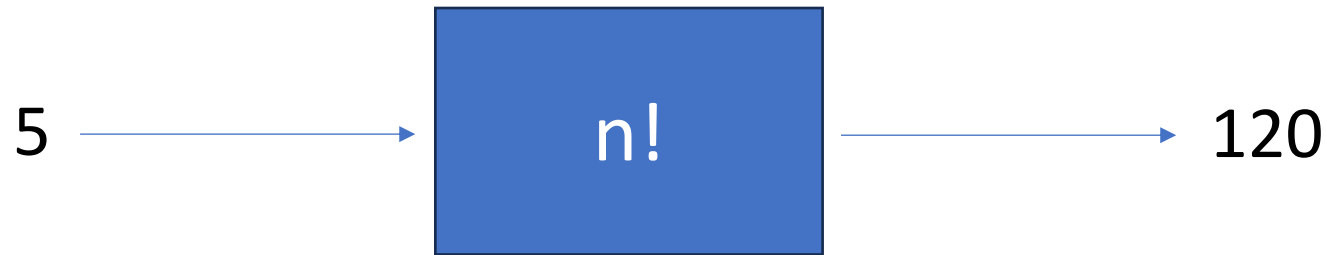
Functional correctness: Metrics

- For the whole test suite: pass rate

$$PassRate = \frac{Passed\ test\ cases}{Total\ Number\ of\ test\ cases} \times 100$$

Functional correctness: Example

- I want a function that computes a factorial of a number.



Functional correctness: Example

- Prepare a set of test cases
- Ensure tests cover a wide range of scenarios, including:
 - **Positive cases:** Valid inputs where the function should succeed.
 - **Edge cases:** Inputs that are extreme or boundary values.
 - **Negative cases:** Invalid inputs or cases expected to produce errors.
- Example Test Cases:
 - Input: 5 → Expected Output: 120
 - Input: 0 → Expected Output: 1
 - Input: -1 → Expected Output: Error

Functional correctness: Example

```
1 class TestFactorialFunction(unittest.TestCase):
2     def test_factorial_positive(self):
3         """Test factorial for a positive input."""
4         self.assertEqual(factorial(5), 120, "Factorial of 5 should be 120")
5
6     def test_factorial_zero(self):
7         """Test factorial for zero."""
8         self.assertEqual(factorial(0), 1, "Factorial of 0 should be 1")
9
10    def test_factorial_negative(self):
11        """Test factorial for a negative input."""
12        with self.assertRaises(ValueError, msg="Factorial is not defined for negative numbers"):
13            factorial(-1)
14
```

Functional correctness: Example

- Prepare a set of test cases
- Ensure tests cover a wide range of scenarios, including:
 - **Positive cases:** Valid inputs where the function should succeed.
 - **Edge cases:** Inputs that are extreme or boundary values.
 - **Negative cases:** Invalid inputs or cases expected to produce errors.
- Example Test Cases:
 - Input: 5 → Expected Output: 120
 - Input: 0 → Expected Output: 1
 - Input: -1 → Expected Output: Error

Functional correctness: Example

- Now I generate my code

```
1  def factorial(n):
2      """Incorrect implementation: does not handle negative inputs properly."""
3      if n == 0:
4          return 1
5      result = 1
6      for i in range(1, n + 1):
7          result *= i
8      return result
```

Functional correctness: Example

- Test execution results:

Test case	Input	Expected Output	Actual output	Test result
TC1	5	120	120	Pass
TC2	0	1	1	Pass
TC3	-1	Error	1	Fail

Pass rate = 66.66%

Static Code Quality Metrics

- Static code quality metrics are tools and techniques used to evaluate the quality of source code **without executing it**. They focus on analyzing the structure, complexity, and maintainability of code to identify potential issues early in the development lifecycle.
- For code generated by LLMs, static code quality metrics can provide an objective assessment of whether the generated code meets industry standards for readability, scalability, and reliability.

Cyclomatic Complexity

- **Cyclomatic Complexity (CC)** measures the complexity of a program by counting the number of linearly independent paths through its source code.
- It provides an estimate of how difficult it is to understand, test, and maintain the code.

Cyclomatic Complexity

- $CC = E - N + 2P$
 - E: Number of edges in the flow graph.
 - N: Number of nodes in the flow graph.
 - P: Number of connected components (typically, $P = 1$ for a single program).
- Alternatively, for a single function, it can be simplified to counting decision points (such as if, while, for, etc.) and using:
 - $CC = E - N + 2$

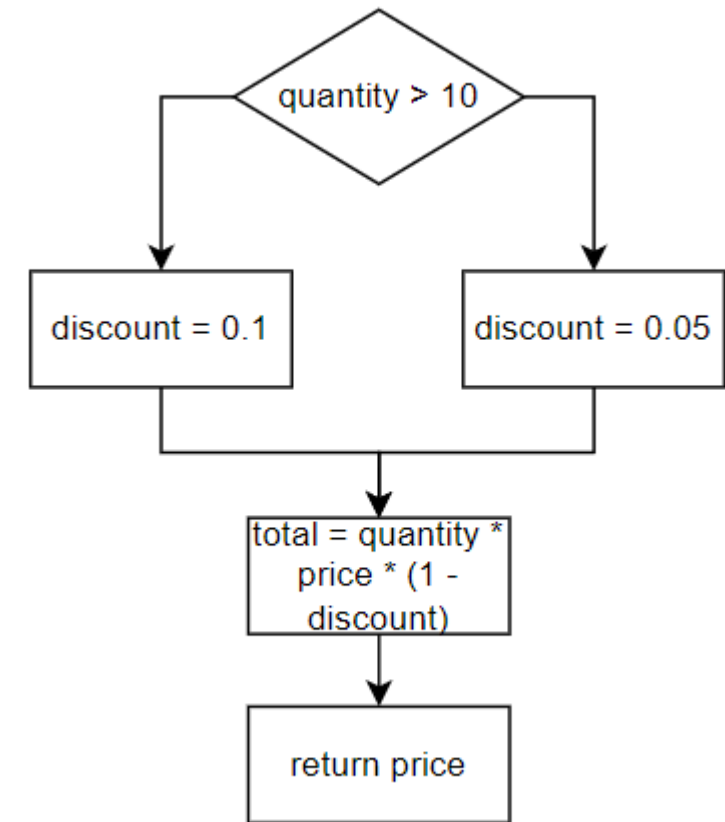
Cyclomatic Complexity

- How to interpret CC:
 - **1-10**: Simple code, easy to maintain.
 - **11-20**: Moderate complexity, requires careful review and testing.
 - **21-50**: High complexity, needs refactoring to improve maintainability.
 - **50+**: Very complex, refactor or reconsider the design.

Cyclomatic Complexity: Example

```
def calculate_total_price(quantity, price):  
    if quantity > 10:  
        discount = 0.1  
    else:  
        discount = 0.05  
    total = quantity * price * (1 - discount)  
    return total
```

CC = 2 (due to the if condition)



Maintainability Index

- **Maintainability Index (MI)** is a metric used to assess the ease with which a software system can be maintained.
- It provides an estimate of how difficult it is to understand, test, and maintain the code.
- It combines several factors (e.g., cyclomatic complexity, lines of code, and Halstead metrics) into a single score that reflects how easy it is to maintain and improve the code over time.

Maintainability Index

Interpretation:

- **between 0 and 20:** Very hard to maintain. The code is complex, hard to read, and may require frequent changes and fixes.
- **between 20 and 40:** Hard to maintain. Code may be overly complex, and developers may face difficulties when working on it.
- **between 40 and 60:** Maintainable with some effort. It's still understandable, but it may require some effort to make changes or expand functionality.
- **between 60 and 80:** Good maintainability. Code is understandable and maintainable, but might require occasional clean-up.
- **between 80 and 100:** Excellent maintainability. The code is clean, easy to maintain, and highly readable.
- **above 100:** Exceptional maintainability. Code is extremely simple, easy to modify, and well-structured.

Halstead Volume

$$HV = (N1 + N2) \times \log_2(N1 + N2)$$

- **N₁**: The number of distinct operators in the code.
- **N₂**: The number of distinct operands in the code.
- **Higher Volume** indicates more complexity and larger code, requiring more effort to understand and maintain.
- **Lower Volume** suggests simpler, more concise code.

Maintainability Index: Example

```
def calculate_total_price(quantity, price):  
    if quantity > 10:  
        discount = 0.1  
    else:  
        discount = 0.05  
    total = quantity * price * (1 - discount)  
    return total
```

$$MI = 171 - 5.2 \times \ln(CC) - 0.23 \times LOC - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):  
    if quantity > 10:  
        discount = 0.1  
    else:  
        discount = 0.05  
    total = quantity * price * (1 - discount)  
    return total
```

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times LOC - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):
```

```
    if quantity > 10:
```

```
        discount = 0.1
```

```
    else:
```

```
        discount = 0.05
```

```
    total = quantity * price * (1 - discount)
```

```
    return total
```

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times 6 - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):
```

```
    if quantity > 10:
```

```
        discount = 0.1
```

```
    else:
```

```
        discount = 0.05
```

```
    total = quantity * price * (1 - discount)
```

```
    return total
```

$$HV = (N1 + N2) * \log(N1 + N2)$$

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times 6 - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):
```

```
    if quantity > 10:
```

```
        discount = 0.1
```

```
    else:
```

```
        discount = 0.05
```

```
    total = quantity * price * (1 - discount)
```

```
    return total
```

$$HV = (5 + N2) * \log(5 + N2)$$

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times 6 - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):
```

```
    if quantity > 10:
```

```
        discount = 0.1
```

```
    else:
```

```
        discount = 0.05
```

```
    total = quantity * price * (1 - discount)
```

```
    return total
```

$$HV = (5 + 7) * \log(5 + 7)$$

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times 6 - 16.2 \times \ln(HV)$$

Maintainability Index: Example

```
def calculate_total_price(quantity, price):
```

```
    if quantity > 10:
```

```
        discount = 0.1
```

```
    else:
```

```
        discount = 0.05
```

```
    total = quantity * price * (1 - discount)
```

```
    return total
```

$$HV = (5 + 7) * \log(5 + 7)$$

$$MI = 171 - 5.2 \times \ln(2) - 0.23 \times 6 - 16.2 \times \ln(12.95) = 124.56$$

Computing CC and MI

```
from radon.complexity import cc_visit
from radon.metrics import mi_visit

# Example function code
function_code = """
def factorial(n):
    #Incorrect implementation: does not handle negative inputs properly.
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
"""

complexity, maintainability_index = analyze_function_with_radon(function_code)

print(f"Cyclomatic Complexity: {complexity}")
print(f"Maintainability Index: {maintainability_index}")
```

```
Cyclomatic Complexity: [Function(name='factorial', lineno=2, col_offset=0, endline=9, is_method=False, classname=None, closures=[], complexity=3)]
Maintainability Index: 91.76868584958225
```

Other metrics computed by Radon

- **Lines of Code (LOC):** Total number of lines of code, excluding comments and blank lines.
- **Halstead Metrics:** Includes Volume, Difficulty, and Effort; assesses the size and complexity of the code based on operators and operands.
- **Code Smells:** Flags potential code quality issues such as long methods or classes, excessive nesting, and duplication.
- **Halstead Volume (V):** Size of the code in terms of information needed to understand it, based on distinct operators and operands.
- **Code Duplication:** Detects repeated blocks of code across the codebase, indicating potential need for refactoring.
- **Comment Density:** Percentage of code that is commented, indicating the clarity and documentation level of the code.
- **Indentation Level:** Measures the depth of indentation, helping to detect deeply nested or overly complex code.

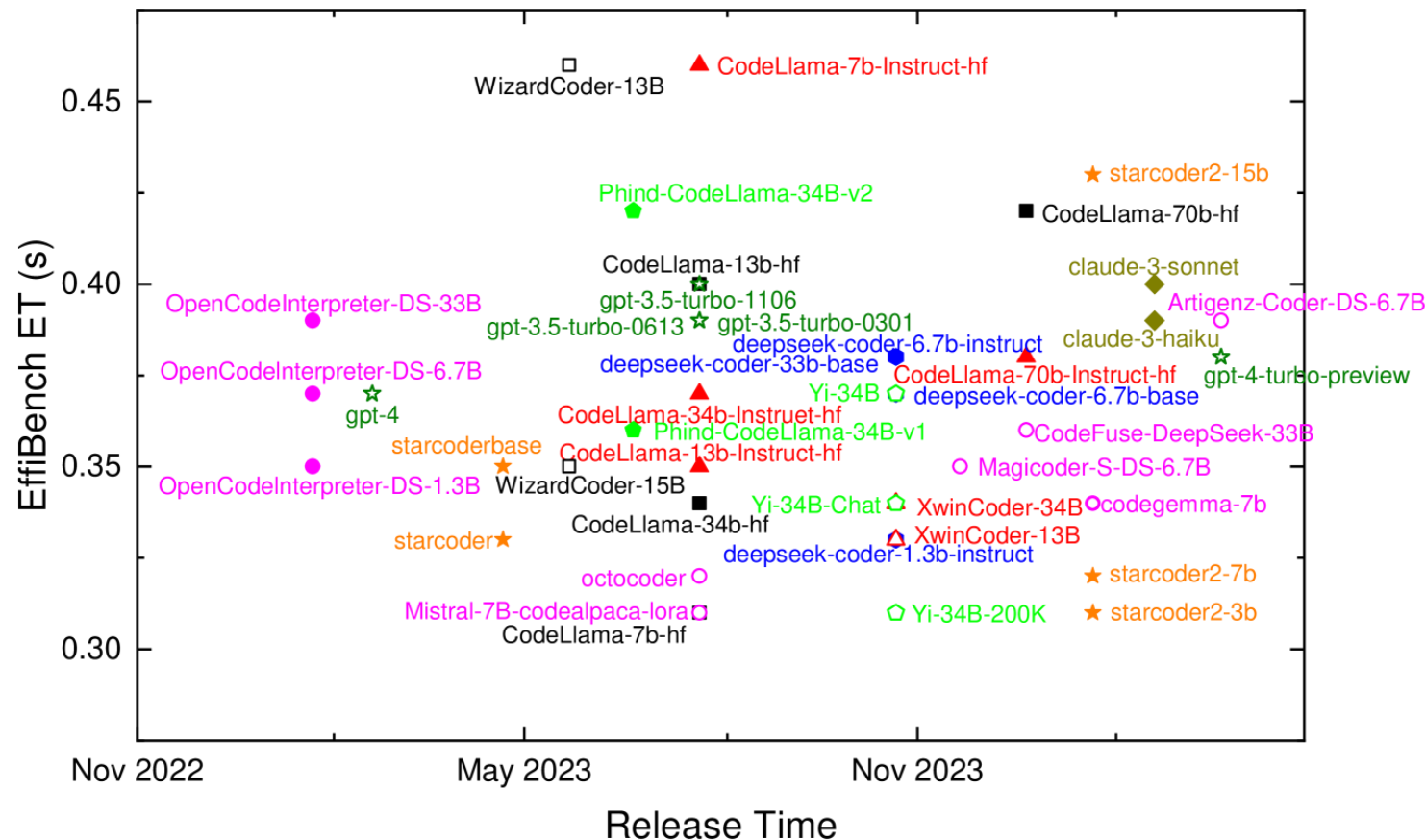
Runtime performance quality metrics

- Several key metrics can be used to assess the efficiency, scalability, and responsiveness of the generated code. These metrics focus on how the code performs during execution, providing insight into its efficiency and identifying potential performance bottlenecks.

Runtime performance quality metrics

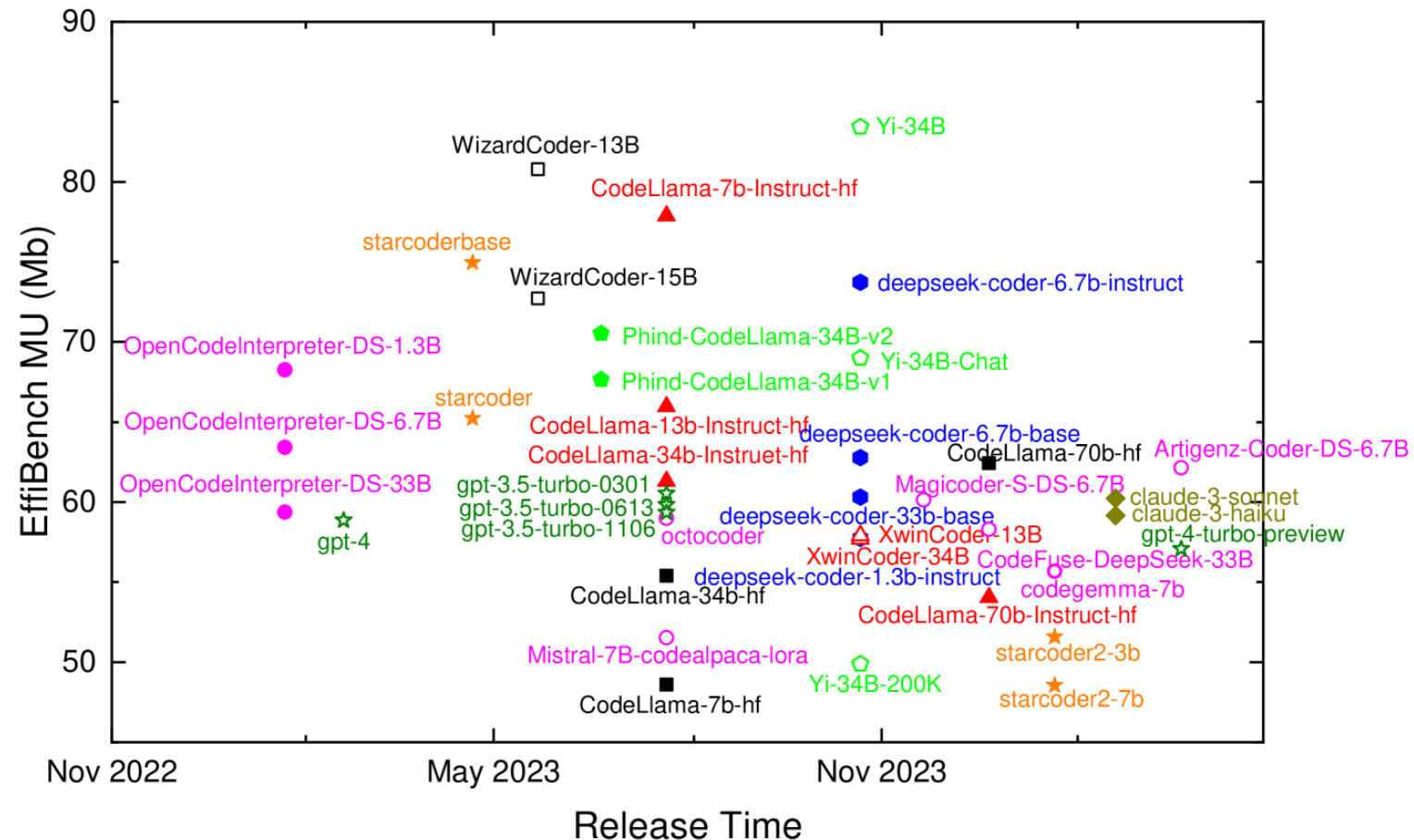
- **Execution time:** Measures the time taken for the code to complete execution for a specific input or task. This is one of the most common and essential performance metrics.
- **Throughput:** Measures the number of operations or tasks the program can perform in a given period. It is often used in systems where tasks are processed in bulk, such as batch processing or web servers.
- **Memory consumption:** Tracks how much memory the code consumes during execution. High memory usage can indicate inefficient code or the need for optimization.
- **CPU time used:** Measures how much CPU time is consumed by the code during its execution. This can help identify code that is CPU-intensive or inefficient in terms of processor utilization.
- **Error rate:** Measures the frequency of errors or exceptions during code execution. A high error rate can indicate that the code is not handling edge cases or is failing under load.

Runtime performance quality metrics



- Execution Time (ET) Evaluation of LLMs on EffiBench Over Release Time

Runtime performance quality metrics



- Memory Usage (MU) Evaluation of LLMs on EffiBench Over Release Time

Code-Specific Similarity Metrics

- The regular BLEU is not sufficient to perform the evaluation of code synthesis without considering the characteristics of the programming language.
- Code is artificially designed to produce various kinds of output, unlike the natural language that has evolved naturally among humans.

Code-Specific Similarity Metrics

- Main differences between programming and natural languages:
 - Limited keywords vs. million of words
 - Tree structure vs. sequential structure
 - Unique instructions vs. ambiguous semantics

Code-Specific Similarity Metrics

- **CodeBLEU** (Ren et al., 2021) consider different aspects in addition to the traditional bleu:
 - **Weighted N-Gram match:** As in the traditional BLEU.
 - **Syntactic AST Match:** syntactic information to consider the tree structure of code.
 - **Semantic Data-flow match:** the ordering of variables and flows in the code.

Code-Specific Similarity Metrics

Machine translation:

```
public static int Sign ( double d )
{
    return ((int)((d == 0)? 0:(d < 0)))?
    -1 : 1;
}
```

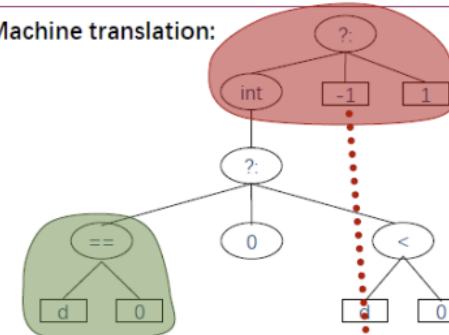
1.0 1.0 0.7 0.5

Reference (human) translation:

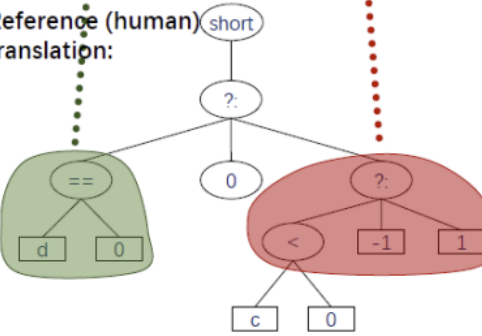
```
public static short Sign ( double d )
{
    return (short)((d == 0)? 0:(c < 0)?
    -1 : 1);
}
```

Weighted N-Gram Match

Machine translation:



Reference (human) translation:



Syntactic AST Match

[[('d', 7, 'comesFrom', [], []),
('d', 16, 'comesFrom', ['d'], [7]),
('d', 24, 'comesFrom', ['d'], [7])]

Machine translation:

```
public static int Sign ( double d )
{
    return ((int)((d == 0)? 0:(d < 0)))?
    -1 : 1;
}
```

Reference (human) translation:

```
public static short Sign ( double c )
{
    return (short)((c == 0)? 0:(c < 0)?
    -1 : 1);
}
```

Semantic Data-flow Match

$$\text{CodeBLEU} = \alpha \cdot \text{N-Gram Match (BLEU)} + \beta \cdot \text{Weighted N-Gram Match} + \gamma \cdot \text{Syntactic AST Match} + \delta \cdot \text{Semantic Data-flow Match}$$

Feedback-based evaluation

- Feedback-based evaluation methods are essential for comprehensively assessing the quality of generated code, as they incorporate human judgment and expertise to evaluate various aspects of code quality.

Feedback-based evaluation

- **Blind peer review** is a common and effective method for evaluating code quality comprehensively. In this method, reviewers assess code snippets generated by different models without knowing the identity of the models, selecting the superior code based on predetermined criteria. This approach eliminates potential biases, making the evaluation results more objective and fair.

Feedback-based evaluations

- **Real-world evaluation:** deploy the generated code in actual application environments and assess its performance in real-world tasks. This method fully evaluates the practicality and reliability of the code, reflecting its real-world effectiveness. Generated code is applied to real programming tasks, with metrics such as error rate, debugging time, and maintenance cost recorded. This approach provides valuable feedback on the code's functionality, stability, and adaptability.
- For example, generated code might perform excellently in a controlled environment but face performance bottlenecks or compatibility issues in practical applications.

Feedback-based evaluations

- **Readability evaluation:** The readability of code is crucial for understanding and maintaining it. Human evaluation methods focus on assessing the functionality, clarity, and maintainability of the code. Reviewers consider naming conventions, comments, and code logic to determine clarity and conciseness. Clear and concise code improves development efficiency and long-term sustainability.
- For example, reviewers check if variable and function names are descriptive, if appropriate comments explain the code logic, and if the code structure is easy to understand.

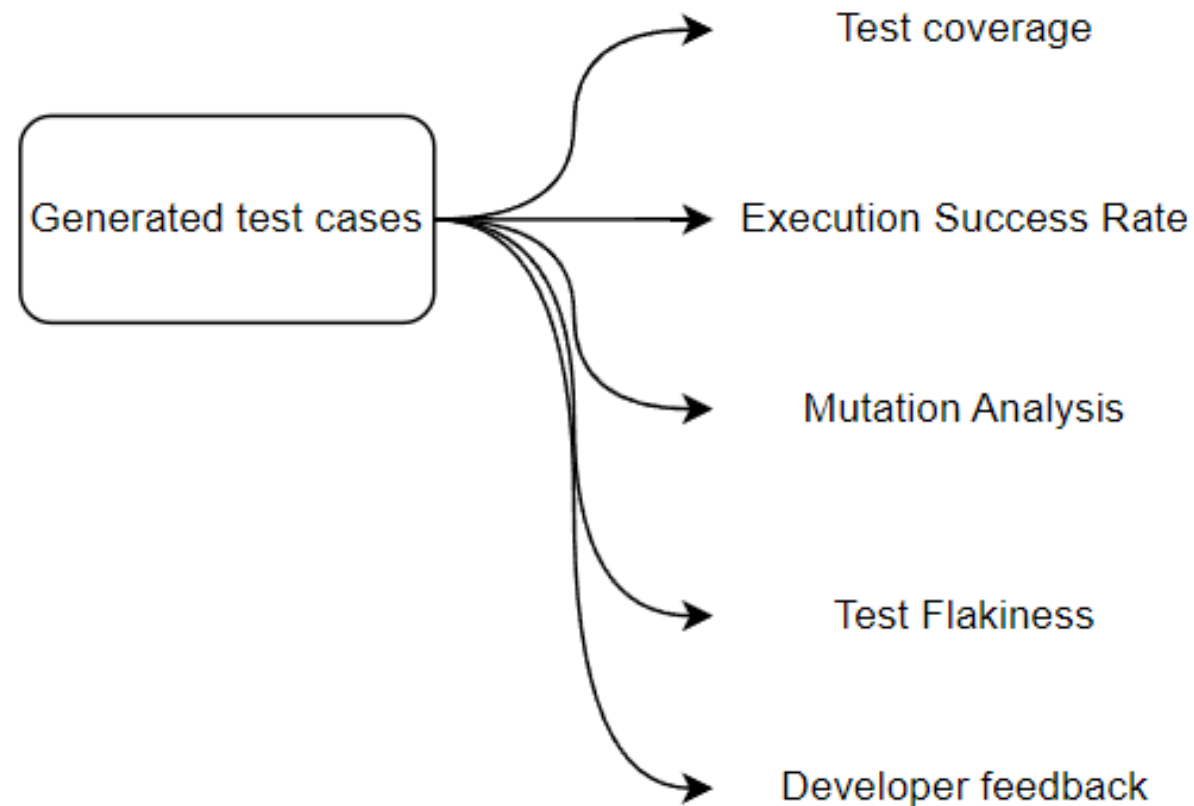
Feedback-based evaluations

- **Maintainability evaluation:** Reviewers evaluate whether the code is reasonably divided into such modules. Additionally, they check for comprehensive documentation and comments, such as descriptions of functions and classes, parameters, and return values.
- For instance, each function should have detailed comments explaining its functionality, input parameters, and return values. Good documentation and comments help current and future developers understand and maintain the code



Evaluating Test Case generation

Evaluating Test Case Generation



Test Coverage

- Coverage refers to the extent to which the codebase is exercised by a set of test cases.
- High coverage increases confidence in code correctness and reduces the risk of hidden bugs.

Test coverage: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Test coverage: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
defined for negative numbers."
```

Test coverage: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
defined for negative numbers."
```

Test coverage: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Test coverage: Example

- Consider the Factorial function

```
def factorial(n):
```

```
    if n < 0:
```

```
        raise ValueError("Factorial not defined.")
```

```
    if n == 0:
```

```
        return 1
```

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result *= i
```

```
    return result
```

T1

```
assert factorial(5) == 120
```

```
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1
```

```
# Tests base case for zero
```

T3

```
try:
```

```
    factorial(-1)
```

```
except ValueError as e:
```

```
    assert str(e) == "Factorial  
undefined."
```

Types of Coverage

- **Line Coverage:** Measures the percentage of lines of code executed during testing.
- **Branch Coverage:** Measures whether all possible branches of decision points (e.g., if statements) are tested.
- **Function Coverage:** Measures whether all functions or methods in the code have been called during testing.
- **Path Coverage:** Measures all potential execution paths through the code. More exhaustive but computationally expensive.

Types of Coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	48.26	100	46.9	48.26	
features	49.36	100	53.7	49.36	
commands.ts	100	100	100	100	
document-symbol-provider.ts	11.4	100	0	11.4	14-114
formatting-provider.ts	83.45	100	71.42	83.45	24-39,108-114
linter-provider.ts	48.35	100	60	48.35	...,186-244,247-272,275-302,364-399,436-472,475-508,511-548,551-593,698-705,717-758
lib	54.57	100	41.02	54.57	
functions.ts	41.44	100	28.57	41.44	27-30,32-45,47-60,63-64,67-68,70-97,104
glob-paths.ts	95.23	100	75	95.23	25-27
helper.ts	32.23	100	15.38	32.23	19-55,57-70,73-79,83-85,87-89,91-93,95-97,99-101,103-105,107-109,111-113
tools.ts	61.73	100	69.23	61.73	62-64,67,78-89,107-134,160,168,198-253,286-289,291-292,294-295,301-309
variables.ts	32.14	100	0	32.14	8-21,24-28
lsp	26.78	100	25	26.78	
client.ts	26.78	100	25	26.78	45-56,62-69,82-158,165-250,263-296,306-339,345-350
services	72.72	100	50	72.72	
logging.ts	72.72	100	50	72.72	4-10,19-20,23-24,47-52,75-76,79-81,95-99
ERROR: Coverage for lines (48.26%) does not meet global threshold (90%)					

Execution success rate

- Execution success rate measures the proportion of test cases that pass when applied to the generated or existing code. This metric helps evaluate the quality of test cases generated by LLMs and their alignment with the expected behavior of the code.
- A wrong test case occurs when the test case itself has an incorrect or unrealistic expected output, leading to test failures that aren't the fault of the code.

$$\text{Execution Success Rate} = \frac{\text{Number of Passing Test Cases}}{\text{Total Number of Test Cases}} \times 100$$

Execution success rate

- In this case we start from the perspective that the **reference code is correct**.
- Typical when we create test cases with the purpose of **regression testing**
 - *Regression testing is a software testing practice that ensures recent changes to the codebase, such as bug fixes, feature updates, or refactoring, do not introduce new defects into previously tested and functioning areas of the software.*

Execution success rate

- Common Causes for Wrong Test Cases:
 - Misinterpretation of the function's logic during test case generation.
 - Mistakes in mathematical or logical reasoning by the LLM.

Execution Success Rate: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2'

```
assert factorial(0) == 0  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Execution Success Rate: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2'

```
assert factorial(0) == 0  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial  
undefined."
```

Execution Success Rate: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n+1):  
        result *= i  
    return result
```

In this case, the code is correct, but T2' is **wrong** -> the expected result is not the right one.

Success Rate = 66.6%

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2'

```
assert factorial(0) == 0  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial  
undefined."
```

Mutation Analysis

- **Mutation Analysis** is a software testing technique that evaluates the effectiveness of a test suite by introducing small changes (mutations) to the code and checking if the existing test cases can detect these changes.
- Mutants represent potential faults in the code, and the goal is to determine whether the test suite can "kill" these mutants by failing when encountering them.

$$\text{Mutation Score} = \frac{\text{Number of Mutants Killed}}{\text{Total Number of Mutants}} \times 100$$

Mutation Analysis

- Mutant Generation:
 - Small changes (mutations) are introduced into the original code.
 - Mutants simulate defects by altering operators, conditions, or statements in the code.
- Test Execution:
 - The test suite is executed against the original code and its mutants.
 - A test "kills" a mutant if the test case detects the mutated fault (i.e., the test fails).
- Surviving Mutants:
 - If the test suite does not detect a mutant (i.e., the test passes despite the mutation), the mutant is said to "survive."

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):  
    if n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 1  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2'

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
defined for negative numbers."
```

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):  
    if -10 < n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 0  
    result = 1  
    for i in range(1, n + 1):  
        result += i  
    return result
```

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```





T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):  
    if -10 < n < 0:  
        raise ValueError("Factorial undefined.")  
    if n == 0:  
        return 0  
    result = 1  
    for i in range(1, n + 1):  
        result += i  
    return result
```



KILLED

T1

```
assert factorial(5) == 120  
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1  
# Tests base case for zero
```

T3

```
try:  
    factorial(-1)  
except ValueError as e:  
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):
```

```
    if -10 < n < 0:
```



```
        raise ValueError("Factorial undefined.")
```

```
    if n == 0:
```

```
        return 0
```



KILLED

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result += i
```



KILLED

```
    return result
```

T1

```
assert factorial(5) == 120
```

```
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1
```

```
# Tests base case for zero
```

T3

```
try:
```

```
    factorial(-1)
```

```
except ValueError as e:
```

```
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):
```

```
    if -10 < n < 0:
```



SURVIVED

```
        raise ValueError("Factorial undefined.")
```

```
    if n == 0:
```

```
        return 0
```



KILLED

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result += i
```



KILLED

```
    return result
```

T1

```
assert factorial(5) == 120
```

```
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1
```

```
# Tests base case for zero
```

T3

```
try:
```

```
    factorial(-1)
```

```
except ValueError as e:
```

```
    assert str(e) == "Factorial is not  
    defined for negative numbers."
```

Mutation Analysis: Example

- Consider the Factorial function

```
def factorial(n):
```

```
    if -10 < n < 0:
```



SURVIVED

```
        raise ValueError("Factorial undefined.")
```

```
    if n == 0:
```

```
        return 0
```



KILLED

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result += i
```



KILLED

```
    return result
```

T1

```
assert factorial(5) == 120
```

```
# Tests iterative multiplication logic.
```

T2

```
assert factorial(0) == 1
```

```
# Base case for zero
```

In this case, T3 passes, but it does not capture the error (mutant) introduced in the code.

Mutation Score = 66.6%

```
except ValueError as e:
```

```
    raise ValueError("Factorial is not
```

```
defined for negative numbers.")
```

Test Flakiness

- **Flaky Test Cases** are tests that sometimes pass and sometimes fail, even when there is no change in the code or environment. These tests are unpredictable and unreliable, often leading to confusion and inefficiencies in the testing process.

Test Flakiness

- **Intermittent Failures:** A test case fails occasionally but not consistently, even if the code hasn't changed.
- **Inconsistent Behavior:** The same test may pass on one run and fail on another, making it difficult to trust the results.
- **False Positives/Negatives:** Flaky tests can falsely indicate that code is faulty (false positive) or that the code is working when it isn't (false negative).

Test Flakiness

- Causes of flakiness:
 - External Dependencies: Tests that rely on external systems (e.g., APIs, databases, or network services) may fail if those systems are down or unstable.
 - Timing and Concurrency Issues: Asynchronous code, race conditions, or timing-related problems can cause tests to fail unpredictably.
 - Environment-Related Issues: Variations in the test environment (e.g., operating system, hardware, or configurations) can lead to test inconsistencies.
 - Shared State: Tests that modify shared state or rely on global variables can impact the reliability of subsequent tests.
 - Resource Limitations: Limited system resources, like memory or CPU, may cause tests to behave erratically.

Test Flakiness

- How to identify Flaky tests:
 - Run Tests Multiple Times: Repeatedly run the same tests to identify those that exhibit inconsistent results.
 - Analyze Test Logs: Investigate logs to spot patterns of failure, such as specific environments, dependencies, or conditions that trigger the flakiness.

$$\textit{Flakiness Rate} = \frac{\textit{Number of Failures}}{\textit{Total Number of Test Runs}} \times 100$$

$$\textit{Stability Index} = 1 - \textit{Flakiness Rate}$$

Test Flakiness

- Threshold for flakiness: there is no universal threshold for what counts as a flaky test, but generally:
 - A test is **flaky** if it has a **flakiness rate** above a certain threshold (for example, 20%).
 - A **stable** test is one where the **pass rate** is high (say above 80-90%).
 - It depends on the context!

(Developer) Feedback

- When using LLMs (Large Language Models) to generate test cases, **developer feedback** is a critical part of evaluating the quality and relevance of the generated tests. Developers' feedback helps refine the test cases by ensuring they align with the actual functionality of the code and by identifying edge cases or missing scenarios.
- Checklist can be employed for this purpose.

(Developer) Feedback: Example

Question	Answer
Is the test case ID unique and easy to identify?	
Is the test case easily readable?	
Is the traceability of the test case with the relevant requirements checked?	
Is the test case prioritized according to the requirements?	
Is the test case type classified properly?	
Are the test case steps clearly defined and easy to understand?	
Does the test case duplicates another test case or is it redundant?	
Is the test data for the test case available with the source of data?	
Does the test case cover both positive and negative scenario?	
Are the language, spelling, and grammatical mistakes in the test case verified?	