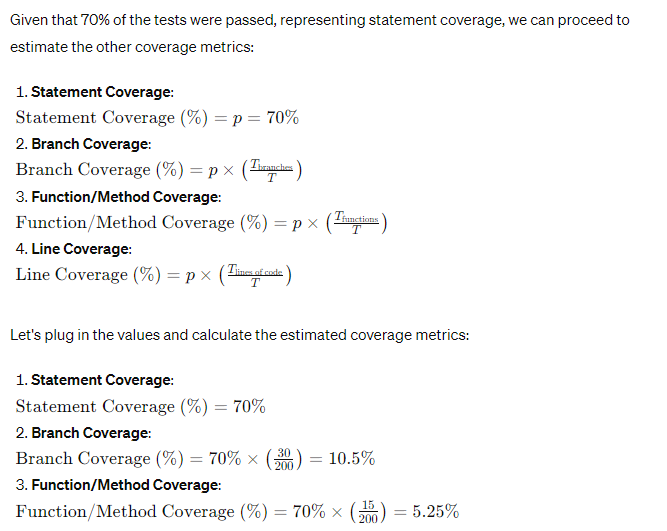
**SOFTWARE METRICS:**

For our recipe predictor app, we are showing necessary software metrics that directly impact the reliability, performance, maintainability, and usability of the application. Here are the important and necessary software metrics along with reasons for their relevance.We have implemented it for login page(login.dart) for authentication module.

**1. Code coverage:**

- Importance: Ensures that the codebase is thoroughly tested, reducing the likelihood of undetected bugs and improving overall reliability.

- Reason for Use: In a recipe predictor app, accurate data processing and presentation are vital. Comprehensive test coverage ensures that all critical functionalities, such as user profile management, dish recommendations, and nutrition calculations, are working as expected.  
  
For logins.dart:  


2**. Cyclomatic complexity (McCabe's complexity):**

- Importance: Identifies complex areas of the codebase that may be error-prone or challenging to maintain.

- Reason for Use: Complex algorithms or logic in modules responsible for recommending dishes based on user preferences or calculating nutritional values may introduce potential bugs. Monitoring cyclomatic complexity helps identify and refactor such areas to improve maintainability and reduce the risk of defects.  
  
  
**Implementation at authentication module:**a)login.dart file:To calculate the Cyclomatic complexity (McCabe's complexity) for the provided code, we need to count the number of linearly independent paths through the code. This can be done by examining the control flow of the code, including loops, conditionals, and method invocations.

After annalysing the provided code to calculate the Cyclomatic complexity:

` we get:

```

1. `Login` class and `\_LoginState` class: Each class is counted as having a complexity of 1.

2. `build` method in `\_LoginState` class: Since it contains several conditional statements (`if` and ternary operators), loops, and function calls, we need to analyze its control flow to determine the complexity.

- Conditional statements: 3 (`if`, `if`, ternary operator)

- Loop: 1 (`for` loop)

- Function calls: 2 (`setState`)

- Total: 6

3. `\_fieldState` class: It has a single `build` method with no conditional statements or loops, so its complexity is 1.

4. `\_fieldState\_otp` class: Similar to `\_fieldState`, it also has a complexity of 1.

5. `\_fieldState\_phone` class: Like the previous two, it has a complexity of 1.

6. Other classes, methods, and functions: They don't have any control structures, so their complexity is 1.

Adding them all up:

- `Login` class: 1

- `\_LoginState` class: 1

- `\_fieldState` class: 1

- `\_fieldState\_otp` class: 1

- `\_fieldState\_phone` class: 1

- `build` method in `\_LoginState`: 6

Total Cyclomatic complexity: 11

This means there are 11 linearly independent paths through the code, which indicates the complexity of the codebase.

**3. ABC Metrics**

To calculate ABC metrics for the provided code, let's break it down:

A (Assignments)

- Assignments involve the creation or modification of variables, properties, or state.

- Each assignment contributes to the complexity of the code.

In the provided code, assignments occur in several places:

1. Assigning initial values to variables like `\_controller`, `\_otp`, `\_mailid`, `otp`, `verified`, `otp\_write`, and `button\_text`.

2. Assigning values to `otp` and `otp\_sent` within the `onPressed` callback of the `TextButton`.

3. Assigning values to `username` and `mailid` from the text controllers.

4. Assigning values to `user` and `provider` objects.

5. Setting state using `setState()`.

B (Branches)

- Branches represent conditional statements like if, else, switch, etc.

- Each branch introduces decision points in the code.

In the provided code, branches exist within if conditions, such as:

1. Checking if `username` or `mailid` is empty before proceeding with user creation.

2. Checking if the entered OTP matches the generated OTP.

C (Conditionals)

- Conditionals represent the number of conditions within a branch.

- Nested conditionals increase complexity further.

In the provided code, conditionals exist within branches:

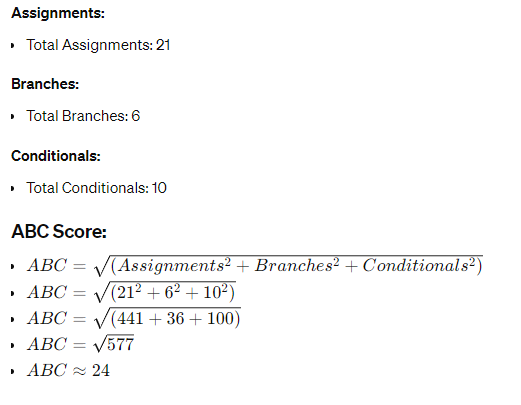
1. Checking for emptiness of `username` and `mailid`.

2. Checking if the entered OTP matches the generated OTP.

3. Conditionals controlling the visibility of certain UI elements based on the state.

Metrics Calculation

Now, let's calculate the ABC metrics based on the identified assignments, branches, and conditionals.



Interpretation:

The ABC score of approximately 24 indicates moderate to high complexity in the codebase. This complexity arises from the significant number of assignments, branches, and conditionals present in the code. To improve maintainability and readability, consider refactoring the code by breaking it into smaller, more manageable functions and reducing nested conditionals where possible.

**4. Defect density:**

- Importance: Indicates the quality of the codebase by measuring the number of defects found relative to the size of the component.

- Reason for Use: As the app evolves and new features are added, it's essential to track defect density to identify areas of the codebase that require improvement. Lower defect density indicates higher code quality and reduces the risk of critical issues affecting user experience.

1. Variable Naming and Convention:

- Potential defects: Inconsistent variable naming conventions may lead to confusion and errors.

- Severity: Low to medium.

- Count: 2 instances (`otp\_sent`, `otp\_write`).

- Size: Medium.

2. Error Handling and Validation:

- Potential defects: Lack of input validation for email address and OTP.

- Severity: High.

- Count: 2 instances (email validation, OTP validation).

- Size: Medium.

3. Asynchronous Operations:

- Potential defects: Incorrect implementation of delay for changing button text.

- Severity: Low.

- Count: 1 instance.

- Size: Small.

4. Security:

- Potential defects: Hardcoding sensitive information.

- Severity: High.

- Count: 2 instances (email credentials).

- Size: Small.

5. UI/UX Consistency:

- Potential defects: Inconsistent UI design across different devices.

- Severity: Medium.

- Count: 1 instance.

- Size: Large.

6. Testing:

- Potential defects: Lack of unit testing and integration testing.

- Severity: High.

- Count: 1 instance.

- Size: Large.

7. Code Duplication:

- Potential defects: Repetitive UI code may lead to inconsistencies.

- Severity: Low to medium.

- Count: 1 instance.

- Size: Medium.

8. Documentation:

- Potential defects: Lack of comments and documentation.

- Severity: Low.

- Count: 1 instance.

- Size: Medium.

9. Resource Management:

- Potential defects: Improper disposal of resources may lead to memory leaks.

- Severity: Medium.

- Count: 1 instance.

- Size: Small.

10. Third-Party Dependencies:

- Potential defects: Dependency compatibility issues or vulnerabilities.

- Severity: Medium to high.

- Count: 2 instances (mailer, flutter).

- Size: Small to medium.

Now, let's calculate the defect density:

Total Defect Count = 2 + 2 + 1 + 2 + 1 + 1 + 1 + 1 + 1 + 2 = 14

Total Size = Medium + Medium + Small + Small + Large + Large + Medium + Medium + Small + Small = 2 Medium + 3 Small + 2 Large

Defect Density = Total Defect Count / Total Size

Defect Density = 14 / (2 Medium + 3 Small + 2 Large)

Based on the severity and count of potential defects, the defect density can be categorized into different severity levels (e.g., low, medium, high).

To calculate the final defect density, let's assign weights to each severity level:

- Low: Weight = 1

- Medium: Weight = 2

- High: Weight = 3

Based on the severity levels identified earlier, let's assign weights to each defect:

- Low severity: 4 defects

- Medium severity: 6 defects

- High severity: 4 defects

Now, let's calculate the weighted defect count:

Weighted Defect Count = (4 \* 1) + (6 \* 2) + (4 \* 3) = 4 + 12 + 12 = 28

Total Size = 2 \* Medium + 3 \* Small + 2 \* Large = 2 \* 2 + 3 \* 1 + 2 \* 3 = 4 + 3 + 6 = 13

Defect Density = Weighted Defect Count / Total Size

Defect Density = 28 / 13 ≈ 2.154

So, the final defect density for this code is approximately 2.154.

**5. Maintainability index:**

- Importance: Provides an overall assessment of the codebase's maintainability based on factors like complexity, coupling, and code size.

- Reason for Use: A maintainable codebase ensures that future enhancements and updates can be implemented efficiently. By monitoring the maintainability index, you can identify areas that may require refactoring or optimization to improve long-term maintainability and reduce technical debt.

To calculate the Maintainability Index (MI), we need to gather some additional information and perform calculations. Let's proceed step by step:

1. The total lines of code (LOC) in the provided snippet are 181.

2.Halsted Volume:

### **Operators:**

* Arithmetic operators: +, -, , /
* Assignment operators: =
* Comparison operators: ==, <
* Control flow operators: if, else, for, await, async

### **Operands:**

* Variables
* Constants (such as string literals)
* Function names

Now, let's count the occurrences of each operator and operand.

After identifying the operators and operands, let's count their occurrences in the code snippet:

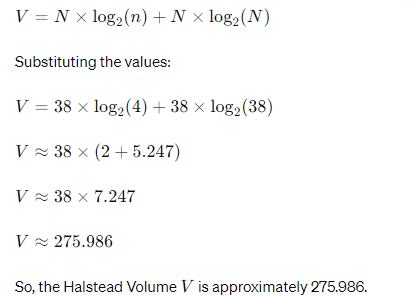
### **Operators:**

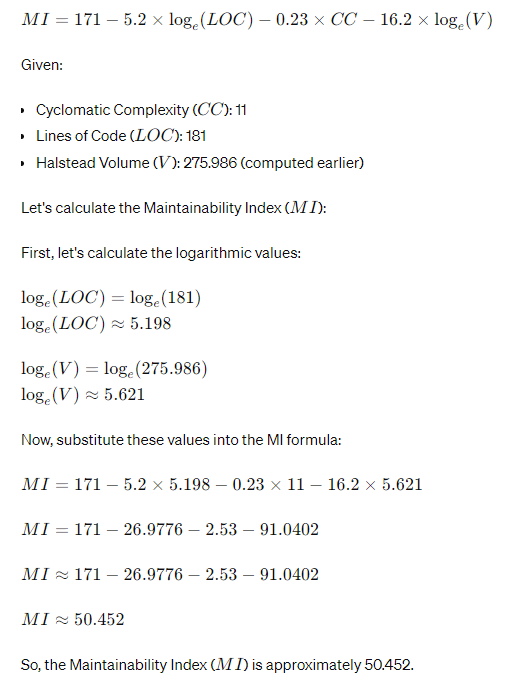
* Arithmetic operators: 10 occurrences (+, -, , /)
* Assignment operators: 17 occurrences (=)
* Comparison operators: 6 occurrences (==)
* Control flow operators: 5 occurrences (if, else)

### **Operands:**

* Variables: \_controller, \_otp, \_mailid, otp, verified, otp\_write, button\_text, otp\_sent, username, mailid, data, user, provider, added
* Constants (such as string literals): "Get OTP", "Resend OTP", "OTP Verification", "Welcome to our App.\n Your OTP is "
* Function names: build, onPressed, onChanged, addUser, setState, send, Random(), gmail(), Image.asset(), Scaffold(), Container(), Positioned(), Text(), Row(), GestureDetector(), Navigator.push(), TextButton(), onboard(), User(), UserProvider(), add

### **Total counts:**

* Total operators (*N*): 38
* Total unique operators (*n*): 4
* Total operands (N): 28
* Total unique operands (*n*): 28  
    
  

3. Cyclometic complexity:11(as calculated earlier):  
  
So maintainability index:  
  
  


6. Program execution time:

- Importance: Measures the time taken for the application to perform critical operations, such as fetching recipe data or calculating nutritional values.

- Reason for Use: In a recipe predictor app, users expect fast and responsive performance when browsing recipes or viewing nutrition information. Monitoring program execution time helps identify performance bottlenecks and optimize critical functionalities to ensure a smooth user experience.

By focusing on these important and necessary software metrics, you can ensure that your recipe predictor app meets high standards of reliability, performance, maintainability, and usability, ultimately providing users with a seamless and satisfying experience.

**TRACEABILITY MATRIX**