基于大模型的测试断言生成技术

房春荣 南京大学

演讲嘉宾

房春荣

南京大学副教授

博士，南京大学软件学院准聘副教授、特聘研究员，博士生导师，紫金学者，CCF高级会员，系统软件/容错计算专委会委员，主要从事智能软件工程研究(BigCode & AITesting)。主持国家自然科学基金项目3项，国家重点研发项目骨干2项，教育部产学合作协同育人项目3项，横向科研项目若干。在CCF-A会议/期刊发表论文40余篇，获得国际会议最佳论文1项，申请发明专利10余项，部分成果在华为、百度等知名企业应用。曾担任AST、AIST等国际会议程序委员会共同主席，多次担任国际顶级会议程序委员会委员及顶级期刊审稿人，并多次获得杰出审稿人。参编多项软件工程和工业APP相关国家、省、团体标准。获2022年国家级教学成果奖，CCF TCFTC 2021年软件测试青年创新奖，2020国家级一流本科课程、2018国家精品在线开放课程《软件测试》。

目 录
CONTENTS

PART 01

研究背景大语言模型和单元测试基础

研究背景-单元测试与测试断言

单元测试：一种被广泛接受的甚至是强制性的开发实践

断言问题：预期的行为或测试断言应该是什么

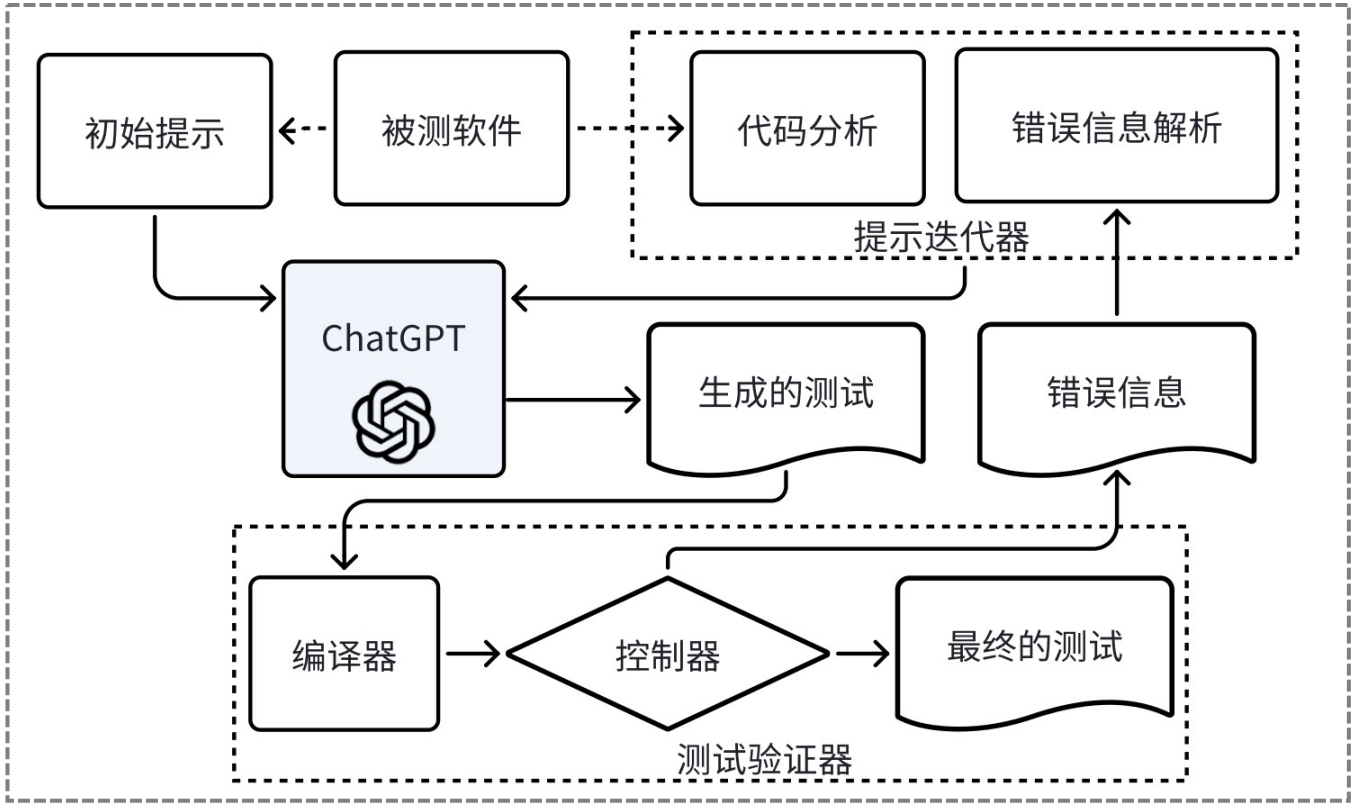
测试前缀

测试断言

```
1 @Test
2 public void test01() throws Throwable  {
3     CategoryAxis3D categoryAxis3D0 = new CategoryAxis3D();
4     categoryAxis3D0.setVisible(false);
5     CategoryAxis3D categoryAxis3D1 = new CategoryAxis3D();
6     boolean boolean0 = categoryAxis3D1.equals(categoryAxis3D0);
7     assertFalse(categoryAxis3D0.isVisible());
8     assertFalse(boolean0);
9 }
```

单元测试的例子

研究背景-单元测试生成方法存在的主要问题



大模型生成单元测试的流程图

- 以大模型会话迭代为主要框架的代表性方法充分利用了大模型的**语义理解**和**代码生成能力**，
- 以 CODAMOSA 为代表的算法，计算资源的开销主要取决于选取的**传统测试方法**，大模型作为辅助工具来改进传统测试方法

研究背景-单元测试生成方法存在的主要问题

生成无效测试用例：

- 大型语言模型（LLMs）可能在上下文不足的情况下生成**无效的测试用例**，导致编译错误。这意味着生成的测试用例无法成功运行，影响测试过程的有效性。

缺乏测试和覆盖反馈：

- 缺乏详细的测试反馈和覆盖率信息可能导致生成的测试用例在运行时出现错误，并且覆盖率低。这意味着生成的测试用例可能未能充分测试目标代码，从而**降低了测试的全面性和有效性。**

重复抑制问题：

- 现有方法中，LLMs可能会陷入自我修复或再生成尝试的重复循环，导致生成过程低效。即模型在面对生成错误时，可能会不断尝试修复同样的错误而无法前进，从而**浪费计算资源和时间。**

研究背景-单元测试生成方法存在的主要问题

■ 基于LLM的单元测试生成的局限性：

- EvoSuite缺乏深入理解源代码的能力，因此复杂的前提条件缩小了基于搜索的测试生成方法的适用范围。
- 尽管LLM在理解语义信息和推理生成能力方面表现出色，但生成的测试用例中不可避免地存在编译错误和运行时错误。如果这些错误能够得到修复，LLM生成的测试用例质量将大大提高。

研究背景-自动断言生成方法存在的主要问题

手动编写单元测试的劳动密集性：

- 编写测试断言通常非常**耗时且劳动密集**，需要测试专家手动插入测试输入（如前缀）和测试输出（如断言）。

现有方法的局限性：

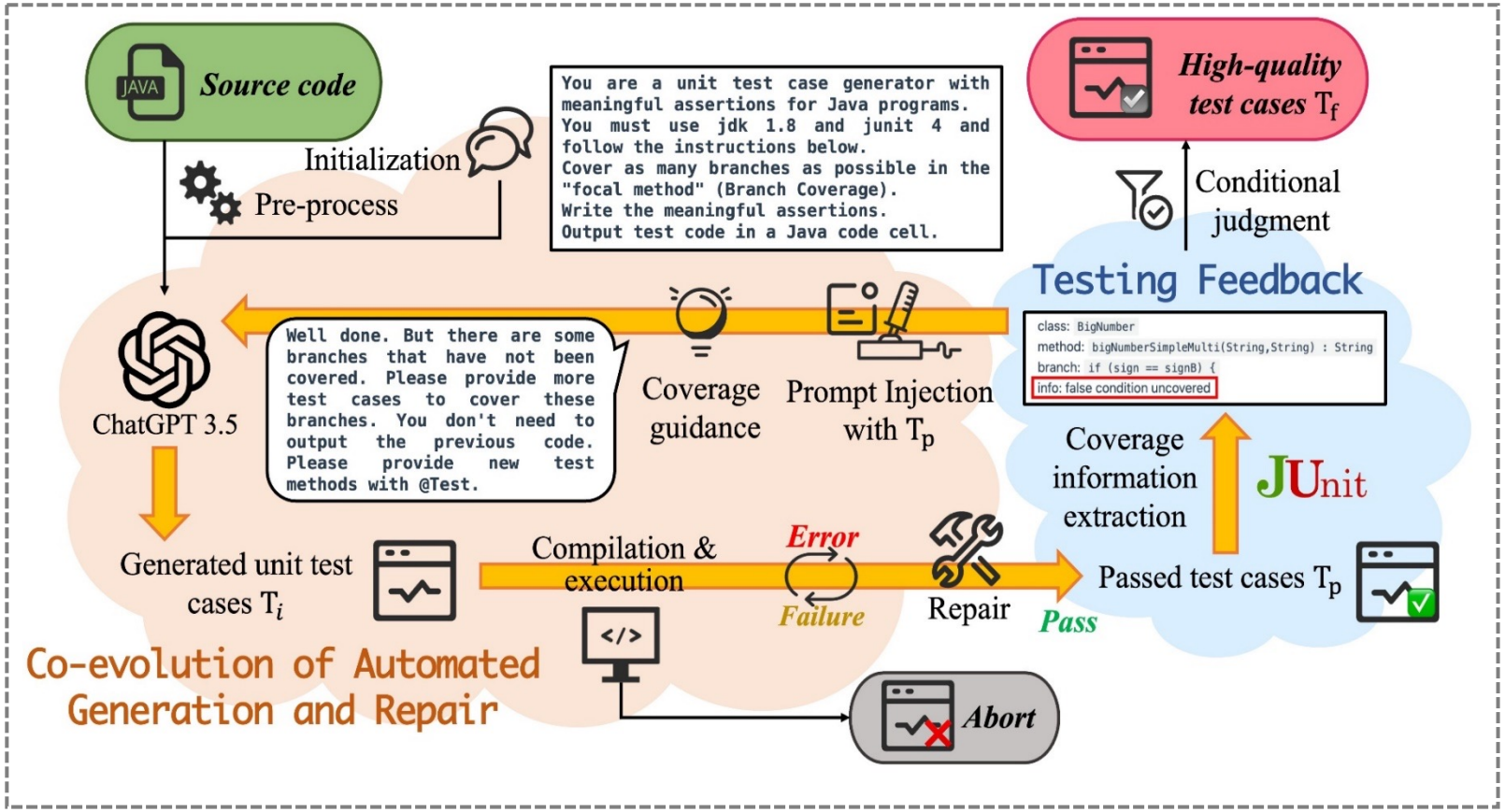
- 基于深度学习（DL）的方法通常需要**大量的训练数据**，并且可能在处理特定任务时**表现不佳**。
- 依赖于从现有数据中检索相关的测试断言，可能会受到词汇匹配的**限制**，难以生成语义上有意义的断言。
- 许多现有方法没有采用联合训练策略，导致检索器和生成器之间**缺乏协同优化**，无法充分利用各自的优势来生成更准确的断言。
- 许多现有方法在有限的训练数据上表现不佳，难以生成高质量的断言。
- 现有使用LLM进行单元测试生成的技术的有效性**并没有系统的比较研究**。

PART 02

单元测试生成和修复

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

- **生成阶段：**是TestART方法的核心步骤，旨在利用大型语言模型（LLM）生成初始的单元测试用例。通过这一阶段，TestART可以自动化生成测试用例，从而减少人工测试的工作量，并提高代码覆盖率。生成阶段的成功执行依赖于有效的上下文预处理和模型提示设计，以充分发挥LLM的潜力。



- **修复阶段：**修复部分介绍了一种通过联合训练来改进检索增强的深度断言生成方法（AG-RAG），AG-RAG结合外部代码库和预训练语言模型，解决了以前方法中的技术局限，在所有基准和指标上显著优于现有的断言生成方法，大幅提高了准确性和生成唯一断言的能力。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

■ 生成阶段步骤

1、 被测代码预处理

■ 代码清理与压缩：在生成阶段之前，源代码需要经过预处理，以确保LLM聚焦于代码的关键部分。

清理步骤:

- 去除注释： 移除所有代码注释，以防止注释内容与代码逻辑的不一致导致LLM产生幻觉。
- 删除多余空行：精简代码结构，提升模型处理效率。

上下文压缩:

- 仅保留焦点方法的完整方法体，其他非关键方法仅保留其方法签名。
- 保留必要的类变量和常量以维持代码的结构和功能完整性。

```
-   public static float toFloat(final String str){
        return toFloat(str, 0.0f);
-   }

+   public static toFloat(String): float
```

■ 变量和方法信息提取:

- 提取所需的变量和方法信息，包括方法签名、起始和结束行号、变量名和数据类型。
- 提取的信息用于构建模型提示，指导生成阶段的测试用例设计。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

生成阶段步骤

2、调用大模型生成初始测试用例

■ 提示设计与模型调用:

- 设计特定的提示以激发LLM生成测试用例的能力。
- 提示内容包括代码上下文、预期行为描述以及已知测试条件。
- 利用OpenAI的ChatGPT-3.5，通过API接口生成初始测试用例。

■ 生成的初始测试用例（ T_i ）:

- ChatGPT-3.5生成的初始测试用例被称为 T_i 。
- T_i 包含对目标代码的多样化测试，包括基本功能测试和边界条件测试。

3、 处理生成的初始测试用例

■ 初步检查和修正:

- 对生成的测试用例进行初步检查，识别并标记编译错误和语法错误。
- 修正简单的语法问题，确保测试用例在编译器中通过基础验证。

■ 生成结果记录与分析:

- 记录生成测试用例的数量、类型和初步结果。
- 分析生成的测试用例覆盖范围及其在真实场景下的有效性。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

■ 生成阶段步骤

4、准备进入修复阶段

■ 错误标记与修复策略准备:

- 标记测试用例中的编译和运行时错误，准备进入修复阶段进行详细修复。
- 应用模板修复策略，自动修复简单错误，减少修复阶段的工作量。

■ 生成-修复协同进化的准备:

- 在生成阶段结束时，确保所有初步生成的测试用例已准备好进入修复阶段。
- 修复后的代码作为新一轮生成的基础，进行迭代优化。

✓ 利用提示注入优化生成

■ 提示注入策略:

- 使用修复后的测试用例作为提示输入，防止LLM在后续生成中出现相同错误。
- 提示注入能有效减少模型的重复抑制问题，提高生成用例质量。

■ 提示优化的效果:

- 修复后的提示能够引导LLM在生成新用例时避免已知错误，并优化测试覆盖率。
- 实验证明，经过提示优化的生成阶段能显著提高生成用例的通过率和覆盖率。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

修复阶段步骤

1、编译与执行

- 把测试代码的编译和执行分开为两个步骤进行，这样做的目的是为了更准确地定位问题所在，针对不同类型的错误进行分类处理，并采用对应的修复模板。

2、错误代码定位与信息提取

- 通过分析 Maven 编译时的日志来定位问题。日志包含了详细的错误信息、错误发生的代码位置以及相关的警告信息。对于运行错误，本方法通过分析报错时输出的堆栈追踪信息来定位问题。
- 堆栈追踪信息详细包含了异常的类型、发生位置(包括类名、方法名和行号等)以及导致异常的方法调用链。

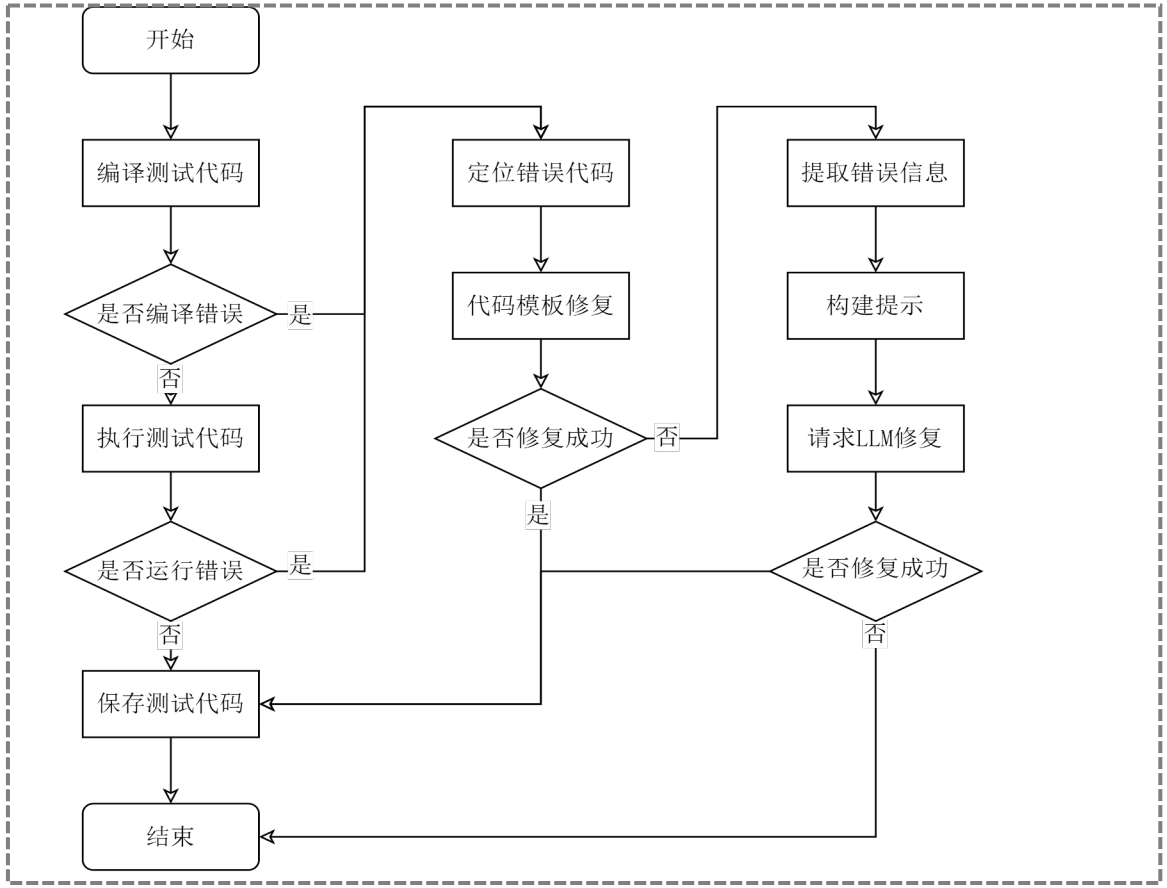


图 修复流程

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

修复阶段步骤

3、模板修复

- 本发明专注于修复生成的测试用例中出现的编译错误、断言错误以及运行时错误。本发明的修复过程更注重测试用例的内部逻辑和预期行为，确保代码的每个部分都能正确运行。

模板一、导包错误模版

```
+ import packageReference.className;
```

- 发生编译错误后，如果错误原因为未找到符号，且在详细错误信息中是由 于找不到类符号引起的，那么就认为是导包错误导致的编译错误。
- 首先索引测试项目、JDK 和所有第三方依赖的 JAR 包，以获取在测试期间所有可访问的 Java 类的完全限定类名(例如，java.util.HashMap)。
- 从详细错误原因中提取缺失的类名，并从索引中找到该类的完全限定类名，然后进行导入。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

修复阶段步骤

模板二、布尔类断言修复

```
-   Assert.assertNull(param);
+   Assert.assertNotNull(param);
or
-   Assert.assertNotNull(param);
+   Assert.assertNull(param);
```

```
-   Assert.assertTrue(param);
+   Assert.assertFalse(param);
or
-   Assert.assertFalse(param);
+   Assert.assertTrue(param);
```

- 当在测试用例中使用 assertNull 方法或者assertTrue方法进行断言时，如果测试失败，意味着被检查的对象与预期矛盾。在这种情况下，将 assertNull 更正为 assertNotNull ， assertTrue更正为assertFalse是一种快速修复策略。相反，如果在断言中使用 assertNotNull 或assertFalse测试失败，则将assertNotNull更正为 assertNull， assertFalse更正为 assertTrue 。

模板三、相等类断言修复

```
- Assert.assertEquals(expectedValue, expression);
+ Assert.assertEquals(actualValue, expression);
```

- 在测试用例中使用 assertEquals 方法进行断言时，通常需要比较两个值是否相等。
- 使用正则表达式来提取测试报告中的预期值和实际值。
- 对提取出的实际值进行处理，去除多余的括号和特殊字符，并根据实际值的类型和格式，构造相应的替换字符串。
- 使用替换字符换将 assertEquals 中的第一个参数替换为实际值。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

修复阶段步骤

➢ 模板四、插入异常处理

```
-   obj.method1();
+   try{
+       obj.method1();
+   }
+   catch(ExceptionType e){
+   // Expected
+   }
```

- 当测试用例的某行代码抛出异常时，通过插入异常处理语句，可以捕获这个异常，达到修复的目的通过使用 try-catch 语句包裹可能抛出异常的代码行，并在 catch 块中处理预期的异常类型。

➢ 模板五、增加捕获语句

```
    try{
        obj.method1();
    }
    catch(ExceptionType1 e){
        //mismatched or insufficient
    }
+   catch(ExceptionType2 e){
        // Expected
+   }
```

- 在某些情况下，现有的异常处理语句可能无法完全覆盖代码执行过程中实 际抛出的所有异常类型。为了进一步提高测试用例的健壮性，添加catch语句可以确保原有的异常处理逻辑 保持完整，同时避免修改引入新的错误、扩大了异常处理的覆盖范围，使测试用例能够应对更多的错误场景。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

修复阶段步骤

4、大模型修复

```
Well done, But there may be assertion errors. / Your unit test has encountered
    an error.
The code snippet and error details for the thrown exception are shown below.
error code and error message:
{error_message}
Modify your test code accordingly and output completed test code in a java
    code cell.
```

图 大模型修复错误反馈模板

- 在模板修复失败的情况下，本方法采用大模型修复作为补充手段，以进一步提高测试代码的修复成功率。

大模型修复的基本思路:将模板修复失败的错误信息反馈给大模型，让其分析错误原因，并尝试生成修正后的测试代码。

提示模板主要由三部分**组成**：

- 错误提示:告知大模型先前生成的测试代码存在断言错误或者运行时异 常，需要进行修正。
- 错误信息:将出错的代码行以及相关错误信息填充到提 示中，为大模型提供错误定位和分析的线索。
- 修复要求:明确要求大模型修改测试代码，并以 Java 代码块的形式输出 完整的测试代码。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

■ 修复阶段步骤

5、覆盖信息反馈

- 该环节的主要目的是计算第三个环节中输出的测试用例 *Tp* 在源代码焦点方法上的覆盖率，并将测试用例的覆盖率信息反馈给大型语言模型，引导其生成下一轮的增量测试用例。

① 计算覆盖率

- 使用 OpenClover 工具的 Maven 插件来计算覆盖率。
- 清理项目以确保分析的准确性
- 利用 OpenClover 插件的 setup 目标初始化覆盖率数据收集环境。
- 执行项目的测试用例集，收集测试过程中被执行的代码行和分支的信息。
- aggregate 目标用于整合所 有单元测试的覆盖率信息。
- clover 目标生成最终的覆盖率报告。

② 构建反馈提示

```
Well done. But there are some branches that have not been covered.
```java
{covered_result}
```
Please provide more test cases to cover these branches.
You don't need to output the previous code. Please provide new test methods
    with @Test.
```

class: `BigNumber`
method: `bigNumberSimpleMulti(String,String) : String`
branch: `if (sign == signB) {`
info: false condition uncovered

自动断言生成-②基于混合检索增强的单元测试断言生成

■ 研究问题:

RQ1: TestART生成的测试用例在正确性方面与基线方法进行比较

RQ2: TestART生成的测试用例的充分性与基线方法进行比较

RQ3: 不同部分的组合如何影响TestART的鲁棒性

■ 数据集:

Defects4J : 该数据集包含可复现的软件缺陷和支持基础设施，以推动软件工程研究。数据集包括来自五个Java项目的公共和非抽象类，研究人员从中提取了作为焦点方法的公共方法，总计8192个方法用于评估。

■ 基线方法：*EvoSuite，A3Test，ChatGPT，ChatUniTest*

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

RQ1：TestART生成的测试用例在正确性方面与基线方法进行比较

表 Defects4J 数据集所选用的项目

Project	Abbr.	Version	Number of Focal methods
Gson	Gson	2.10.1	378
Commons-Lang	Lang	3.1.0	1728
Commons-Cli	Cli	1.6.0	177
Commons-Csv	Csv	1.10.0	137
JFreeChart	Chart	1.5.4	5772
Total			8192

通过比较不同方法对8192个焦点方法生成的测试用例结果：

- 相比ChatGPT提升了**10%编译通过率，16%的运行通过率，30%的整体通过率**；
- 相比GPT-4.0提升了**4%编译通过率，17%的运行通过率， 20%的整体通过率**；

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

RQ1：TestART生成的测试用例在正确性方面与基线方法进行比较

表 TestART与不同基线的正确性性能比较

Method	Projects	Focal methods	Fail	SyntaxError↓	CompileError↓	RuntimeError↓	Pass↑
TestART(Ours)	Gson	378	1.06%	0.79%	23.54%	11.11%	63.49%
	Lang	1728	0.00%	0.12%	4.17%	7.87%	87.85%
	Cli	177	0.00%	1.69%	12.99%	9.60%	75.71%
	Csv	137	0.73%	0.00%	10.22%	12.41%	76.64%
	Chart	5772	0.02%	0.50%	14.21%	8.39%	76.89%
A3Test	Total	8192	0.00%	24.43%	34.77%	25.54%	15.26%
ChatGPT-3.5			0.07%	0.45%	25.09%	24.48%	49.91%
ChatGPT-4.0			0.01%	**0.05%**	19.43%	20.75%	59.75%
ChatUnitTest			0.70%	0.60%	23.35%	16.47%	60.05%
TestART(Ours)			0.07%	0.45%	**12.43%**	**8.50%**	**78.55%**

💡 **Finding：实验结果表明，使用固定的修复模板更有效，能将通过率提高18.50%。**

生成的测试用例常常有相对一致的错误，而大语言模型难以运行这些测试用例以获取错误反馈，因此使用大语言模型进行调试和修复往往会陷入困境。

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

RQ2：TestART生成的测试用例的充分性与基线方法进行比较

表 基线的总覆盖率比较

Method	Project	Branch Coverage↑	Line Coverage↑
A3Test	Total	15.04%	14.63%
ChatGPT-3.5		43.10%	42.58%
ChatUnitTest		48.68%	47.39%
ChatGPT-4.0		51.86%	50.88%
TestART(Ours)		**69.40%**	**68.17%**

💡 **Finding：在相同的基于ChatGPT-3.5的实验方法中，ChatUniTest的覆盖率虽然不如ChatGPT-4.0，但仍高于ChatGPT-3.5。TestART的平均覆盖率比ChatGPT-4.0高出约17%，这主要得益于测试反馈和提示注入。通过覆盖信息的指导，增量迭代的测试用例可以大大覆盖原测试用例中遗漏的部分**

单元测试生成-基于提示引导和动态反馈的大模型测试用例生成和修复

RQ3：不同组件的组合如何影响TestART的鲁棒性

表 消融实验对比结果

Method	Branch Coverage↑	Line Coverage↑	Pass↑
ChatGPT-3.5	43.10%	42.58%	49.91%
+ Repair	62.13%	62.24%	**78.55%**
+ Repair + Iteration	66.48%	64.49%	**78.55%**
TestART	**69.40%**	**68.17%**	**78.55%**

💡 **Finding：我们展示了四种不同TestART变种的总分支覆盖率、总行覆盖率和通过率：仅使用ChatGPT-3.5、加上修复模块（+ Repair）、加上修复和迭代模块（+ Repair + Iteration）、以及完整的TestART（包括修复、迭代和测试反馈）。数据表明，修复模块起着至关重要的作用，将通过率提高了28.64%，分支覆盖率提高了19.03%，行覆盖率提高了19.66%。迭代和测试反馈分别贡献了约3%的覆盖率提升。**

PART 03

单元测试的断言问题

单元测试核心问题-测试断言生成

自动测试生成工具的局限性

Randoop
Automatic unit test generation for Java

EVOSUITE
Automatic Test Suite Generation for Java

- 缺乏语义理解能力
- 难以检测真实缺陷

单元测试示例

```
1 class Stack( ) {
2   public void pop( ) {
3     // NO_OP
4   }
5
6   …
7
8 }
```

(a) Buggy implementation

```
1 public void testPop() {
2   Stack <int> s = new Stack <int> ();
3   int a = 2
4   s.push (a);
5   s. pop ( );
6   bool empty = s.isEmpty ( );
7   assertTrue (empty);
8 }
```

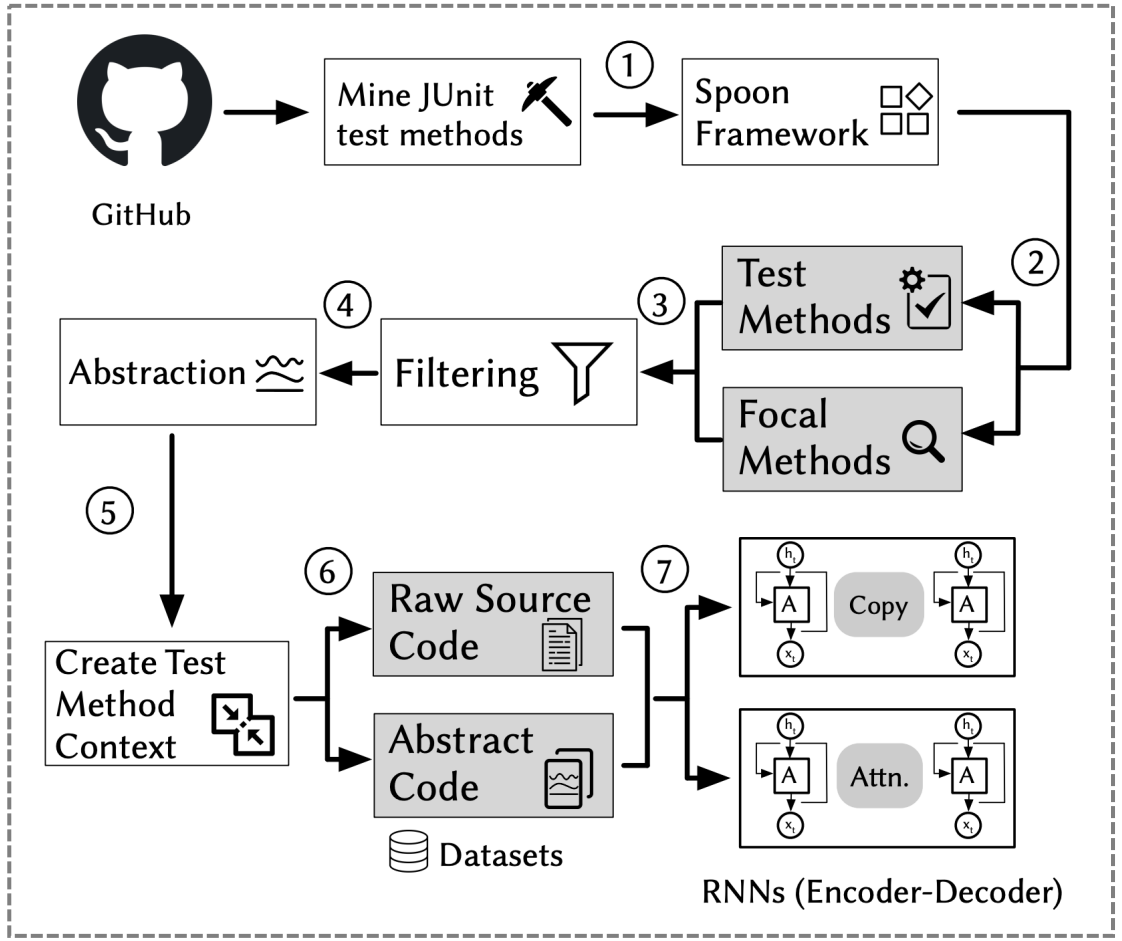
(b) Unit test for pop

```
1 public void testPop() {
2   Stack <int> s = new Stack <int> ();
3   int a = 2
4   s.push (a);
5   s. pop ( );
6   bool empty = s.isEmpty ( );
7   assertFalse (empty);
8 }
```

(c) Generated by Evosuite

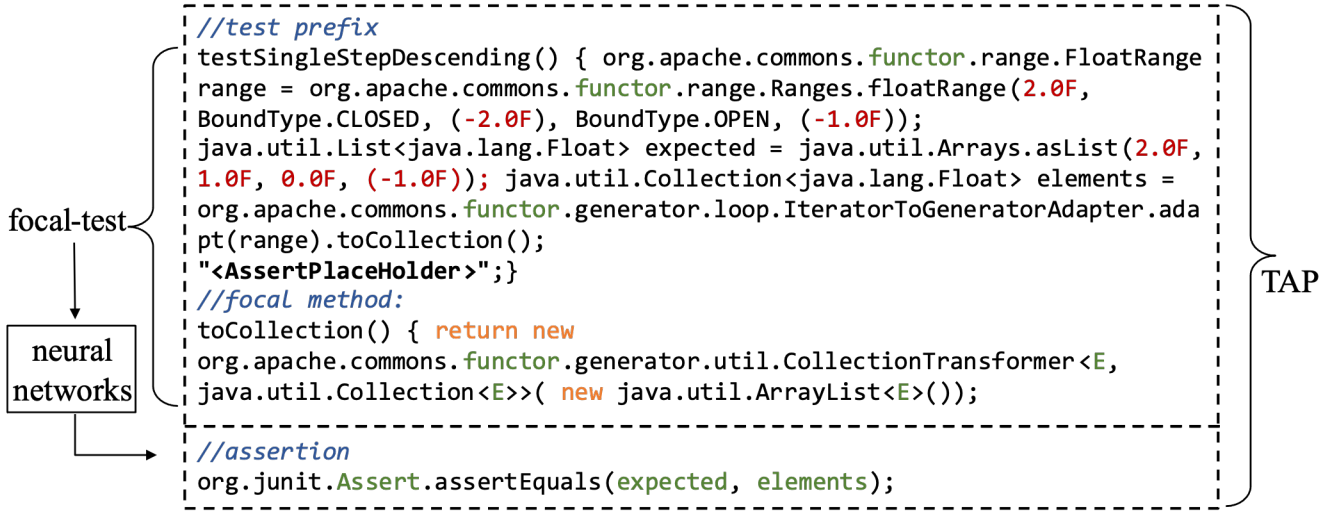
Example of a unit test case generated by Evosuite

自动断言生成-之前的工作



基于深度学习的断言生成 : ATLAS

- 训练语料有限：无法充分理解单元测试语义语义信息
- 模型架构局限性：RNN难以学习长序列数据和上下文关联信息



- ATLAS，通过训练RNN模型来从现有代码中提取相关断言对的基于检索的断言生成方法。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

💡是否可以利用更先进的大语言模型完成断言生成任务?

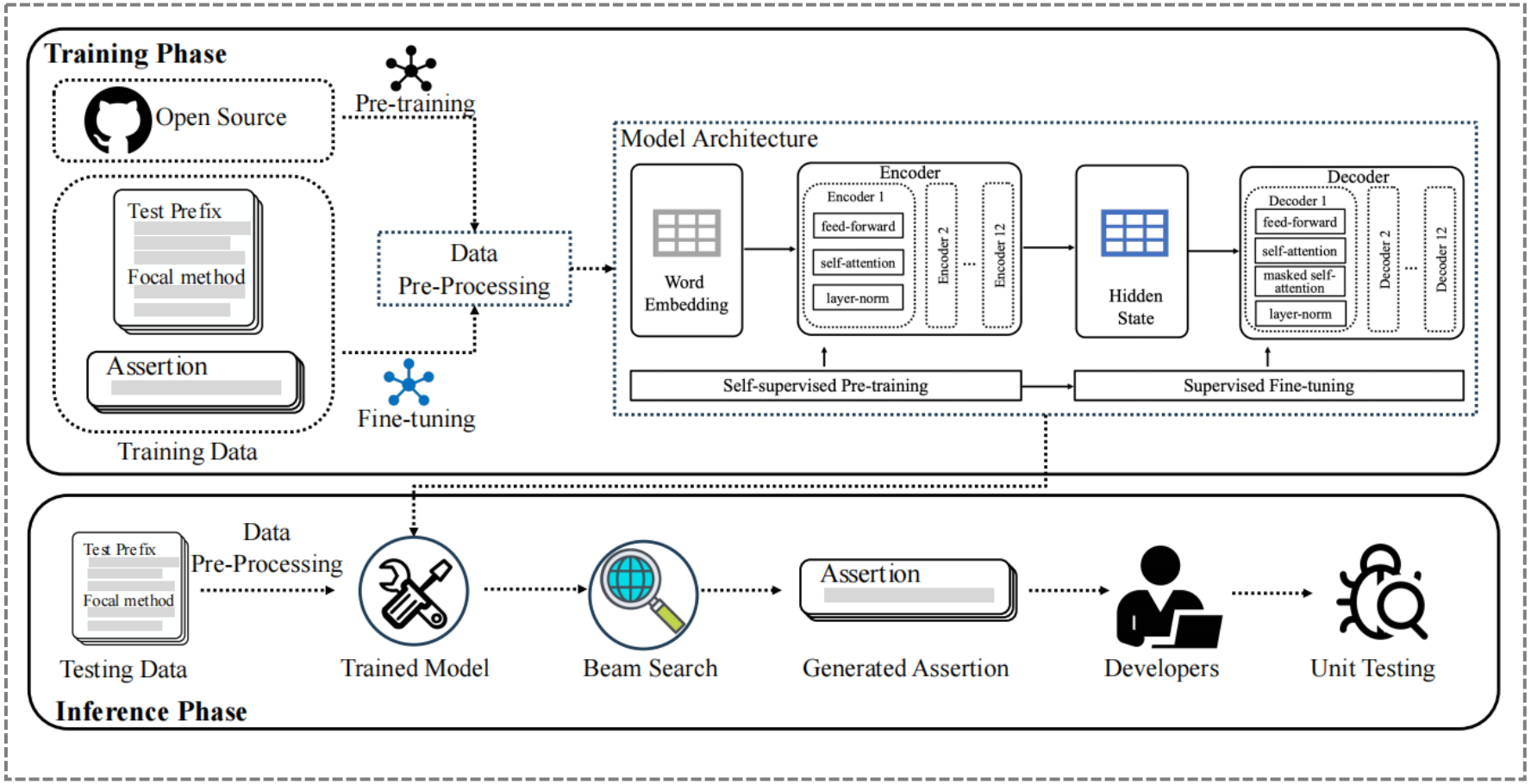


图1 基于LLM的断言生成的架构概述

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

- **系统性比较现有LLMs的效果：**
  - 评估不同LLMs在自动断言生成中的表现：研究中评估了多个大语言模型，包括CodeBERT、GraphCodeBERT、UniXcoder、CodeT5和CodeGPT
- **探索这些模型的优缺点：**
  - 分析每个模型的优缺点，找出其在生成准确性、覆盖率等方面的差异。

RQ1

- LLM在生成单元断言方面的推断准确性如何?

RQ2

- LLM生成的断言在实际软件缺陷检测方面的性能如何?

RQ3

- 能否通过信息检索增强LLM的断言生成能力?

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

数据集

AssertType	Total	Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
$Data_{old}$	15,676	7,866 (50%)	2,783 (18%)	1,441 (9%)	1,162 (7%)	1,006 (6%)	798 (5%)	307 (2%)	311 (2%)	2 (0%)
$Data_{new}$	26,542	12,557 (47%)	3,652 (14%)	3,532 (13%)	1,284 (5%)	1,071 (4%)	735 (3%)	362 (1%)	319 (1%)	3,030 (11%)

表 数据集对比

数据集1: $Data_{old}$

- 来源: GitHub的250万测试方法
- 内容: 包含测试前缀和相应的断言
- 预处理: 去除长度超过1K的测试方法，过滤掉包含未知令牌的断言，最终样本数量为156,760个

训练集: 80%

验证集: 10%

测试集: 10%

数据集2: $Data_{new}$

- 扩展: 在$Data_{old}$基础上增加了之前被过滤的样本，共计265,420个样本

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

模型选择

- 选取的LLM包括: CodeBERT, Graph CodeBERT, UniXcoder, CodeT5, CodeGPT
- 这些模型在编码器-解码器架构上进行预训练和微调

评价指标

- 准确率（Accuracy）：生成断言与参考断言完全匹配的比例
- 错误检测能力（BugFound）：生成断言能够检测到的软件错误数量

Model		CodeBERT	GCodeBERT	UniXcoder	CodeT5	CodeGPT
Open Source		✔	✔	✔	✔	✔
Programming		✔	✔	✔	✔	✔
Archite	Encode-Only	✔	✔			
	Decoder-Only					✔
	Encoder-Decoder			✔	✔	
Organization		HIU	SYU	SYU	Salesforce	Microsoft
Model Size		125M	125M	125M	220M	124M
Pre-training		CSearchNet	CSearchNet	CSearchNet	CSearchNet BigQuery	CSearchNet

表：我们实验中选取的开源LLMs的详细信息。

基线方法

- 现有的AG技术包括: ATLAS、 IR_{ar} 、 RA^{H}_{adapt} 、 RA^{NN}_{adapt}、Integration,、EditAS

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

RQ1：LLMs在断言生成中的推理准确性如何?

$Data_{old}$数据集

- LLM的准确率范围: 51.82%~58.71%
- ATLAS的准确率: 31.42%
- 其他AG技术的准确率: 36.26%~46.54%
- 最高准确率: 58.71%
- 比ATLAS提高: 86.86%
- 比Integration提高: 26.15%

准确率对比

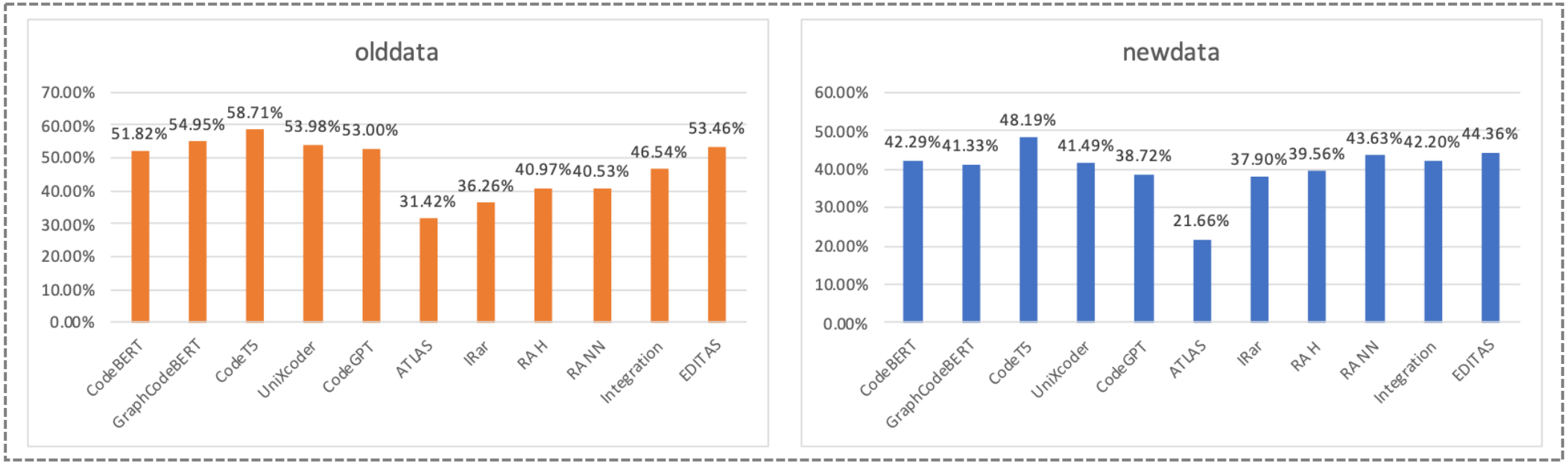
CodeT5表现

$Data_{new}$数据集

- LLM的准确率范围: 38.72%~48.19%
- ATLAS的准确率: 21.66%
- 其他AG技术的准确率: 37.90%~44.36%
- 最高准确率: 48.19%
- 比ATLAS提高: 122.48%
- 比Integration提高: 14.19%

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

RQ1：LLMs在断言生成中的推理准确性如何?

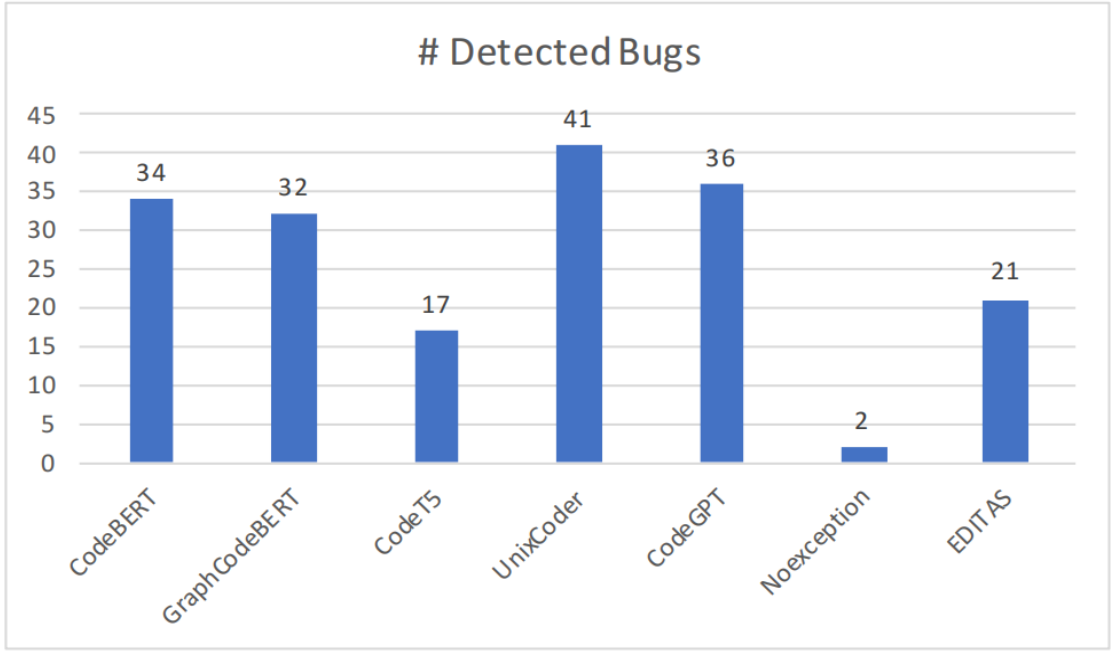


(a) $Data_{old}$ dataset　　(b) $Data_{new}$ dataset

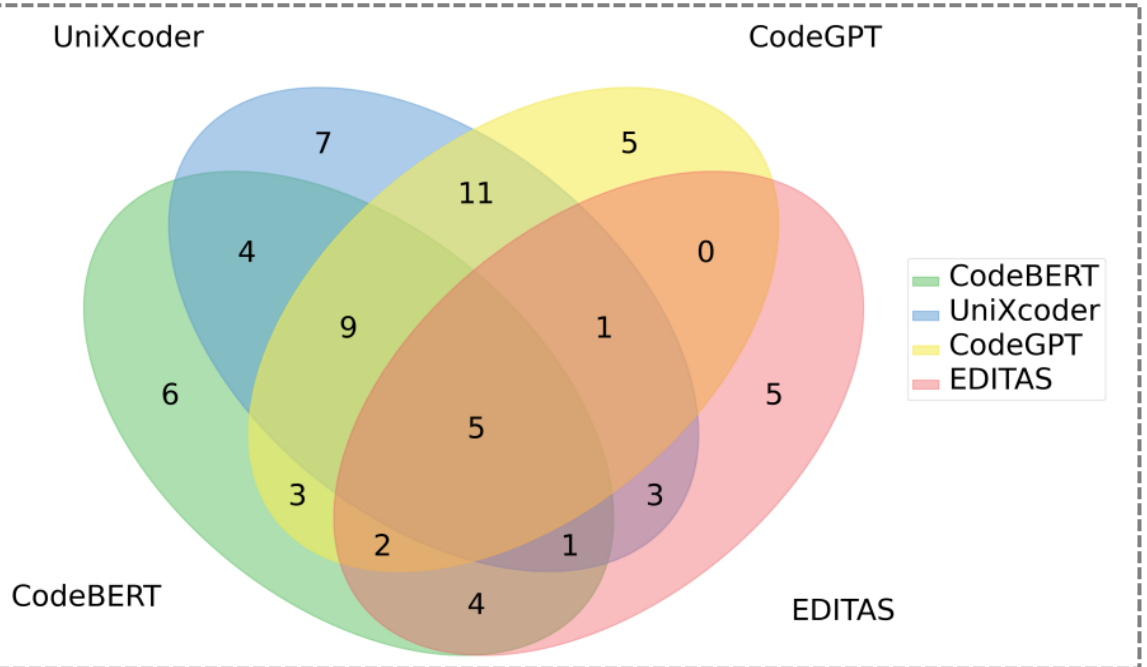
Finding： （1）LLMs在自动化断言生成中能够达到显著的表现，其在$Data_{old}$和$Data_{new}$上的预测准确率分别为51.82%、58.71%和38.72%~48.19%；（2）与最先进的AG技术相比，LLMs在预测性能上有所提高，平均提高了29.60%和12.47%；（3）CodeT5在所有LLMs和所有基线中表现最出色，其在两个基准测试上的预测准确率平均高出13.85%和39.65%。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

RQ2：LLMs生成的断言能有效检测真实漏洞吗?



图A. LLMs 和断言生成领域最新技术的错误检测性能



图B. 不同方法检测到的错误的重叠率

Finding： **（1）五个选定的LLMs能够平均检测到32个真实漏洞，比最先进的技术EDTAS高出52.38%；**

（2）三个表现最好的LLMs平均能检测到26个漏洞，比最先进的技术EDTAS提高了136.36%；

（3）这三个LLMs的结合能够检测到61个漏洞，证明了LLMs在单元测试中检测现实世界漏洞的潜力。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

观察：RQ1中直接检索的断言准确率达到36%，我们是否可以利用检索的结果进一步引导大模型生成?

EASE方法的概念

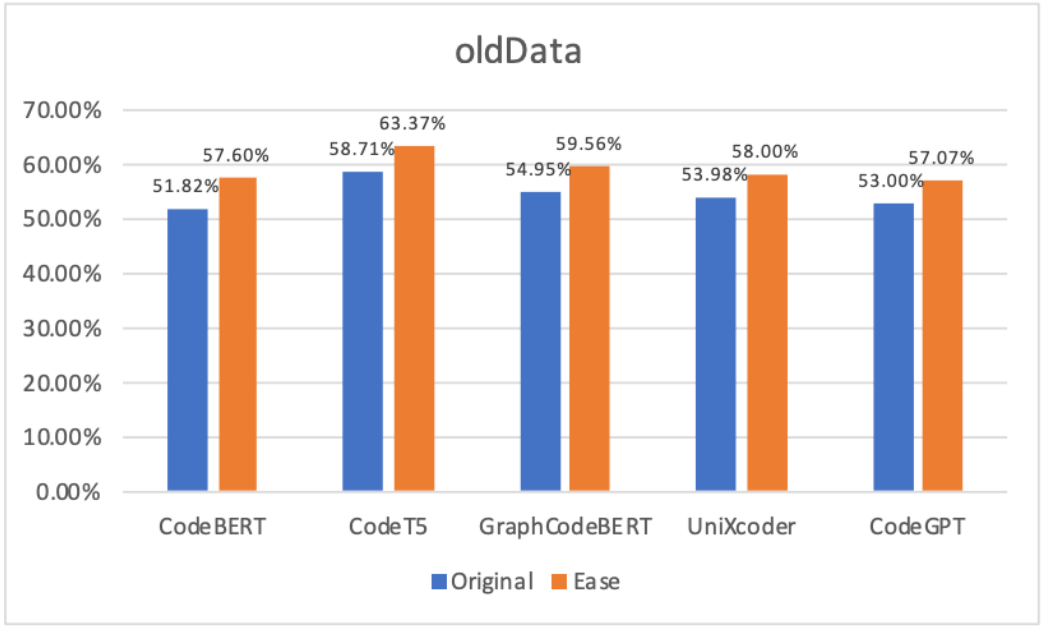
- 结合检索到的断言和LLMs生成新的断言：类似程序修复任务中的“整容手术”假设，通过少量编辑操作生成新的断言。
- 不需要从头生成断言，仅进行少量编辑操作：提高生成效率和准确性，减少生成新断言的复杂度。
- 提供实际开发场景中的额外信息：开发人员经常参考相似的代码片段来生成新代码，通过结合检索到的断言，提供更多上下文信息。

EASE方法实验结果

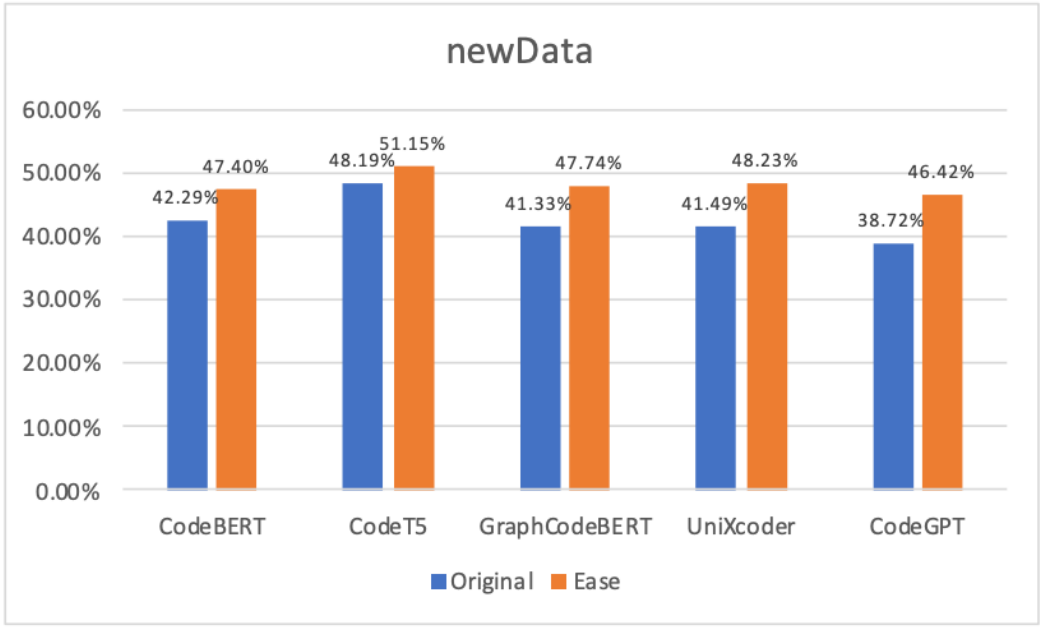
- EASE方法进一步提升了LLMs的断言生成准确率：CodeT5的准确率从原来的58.71%提升到63.37%。
- 其他LLMs在EASE方法的辅助下也表现出色：例如，CodeBERT的准确率从51.82%提升到57.60%，显示出EASE方法的广泛适用性和有效性。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

RQ3：通过检索增强的LLM方法EASE能否提高现有LLM技术的效果?



(a) Data_{old} dataset



(b) Data_{new} dataset

💡 **Finding：我们的比较结果表明，通过结合LLMs和检索到的断言，EASE能进一步提升现有断言生成方法的性能，为所有LLMs带来了预测准确率的新记录，例如CodeT5在$Data_{new}$上的预测准确率达到51.15%。**

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

■ 讨论1：候选断言数量的影响

①设计：

（1）研究了不同候选断言数量（beam size）对LLMs预测准确性的影响。

（2）现有研究（如CIRCLE）表明，生成更多候选断言可以提高预测性能。

（3）探讨在不同实际场景中，LLMs在生成Top-1和Top-5候选断言时的预测准确性。

Model	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=10$	$k=30$	$k=50$
CodeT5	48.19% -	55.50% (↑ 15.17)	58.33% (↑ 21.04)	59.97% (↑ 24.44)	61.02% (↑ 26.62)	63.36% (↑ 31.48)	64.34% (↑ 33.51)	64.72% (↑ 34.30)
CodeGPT	38.72% -	45.32% (↑ 17.05)	48.61% (↑ 25.54)	50.57% (↑ 30.60)	52.09% (↑ 34.53)	55.94% (↑ 44.47)	60.28% (↑ 55.68%)	61.87% (↑ 59.79%)
CodeBERT	42.99% -	48.65% (↑ 13.17)	51.21% (↑ 19.12)	52.80% (↑ 22.82)	53.90% (↑ 25.38)	56.84% (↑ 32.22)	60.23% (↑ 40.10)	61.65% (↑ 43.41)
GraphCodeBERT	41.33% -	47.78% (↑ 15.61)	50.36% (↑ 21.85)	51.81% (↑ 25.36)	53.07% (↑ 28.41)	56.10% (↑ 35.74)	59.75% (↑ 44.57)	61.66% (↑ 49.19)
UniXcoder	41.49% -	47.74% (↑ 15.06)	50.32% (↑ 21.28)	51.95% (↑ 25.21)	53.11% (↑ 28.01)	56.21% (↑ 35.48)	59.84% (↑ 44.23)	61.20% (↑ 47.51)

↑ denotes performance improvement of LLMs at various k values against the performance at $k=1$.

表：不同候选断言数量下LLM在断言生成中的实验结果

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

Model	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=10$	$k=30$	$k=50$
CodeT5	48.19% -	55.50% (↑ 15.17)	58.33% (↑ 21.04)	59.97% (↑ 24.44)	61.02% (↑ 26.62)	63.36% (↑ 31.48)	64.34% (↑ 33.51)	64.72% (↑ 34.30)
CodeGPT	38.72% -	45.32% (↑ 17.05)	48.61% (↑ 25.54)	50.57% (↑ 30.60)	52.09% (↑ 34.53)	55.94% (↑ 44.47)	60.28% (↑ 55.68%)	61.87% (↑ 59.79%)
CodeBERT	42.99% -	48.65% (↑ 13.17)	51.21% (↑ 19.12)	52.80% (↑ 22.82)	53.90% (↑ 25.38)	56.84% (↑ 32.22)	60.23% (↑ 40.10)	61.65% (↑ 43.41)
GraphCodeBERT	41.33% -	47.78% (↑ 15.61)	50.36% (↑ 21.85)	51.81% (↑ 25.36)	53.07% (↑ 28.41)	56.10% (↑ 35.74)	59.75% (↑ 44.57)	61.66% (↑ 49.19)
UniXcoder	41.49% -	47.74% (↑ 15.06)	50.32% (↑ 21.28)	51.95% (↑ 25.21)	53.11% (↑ 28.01)	56.21% (↑ 35.48)	59.84% (↑ 44.23)	61.20% (↑ 47.51)

↑ denotes performance improvement of LLMs at various k values against the performance at $k=1$.

表：不同候选断言数量下LLM在断言生成中的实验结果

②结果：

（1）生成Top-1候选断言时表现出色，候选数量增加至5时，预测准确性显著提高。

（2）UniXcoder的预测准确性从Top-5的53.11%提高到Top-50的61.20%。

③总结：更多候选断言显著提升性能，未来研究可探索多断言过滤方法，与现有AG方法结合。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

■ 讨论2：断言类型的影响

①设计：

（1）研究了不同类型断言对LLMs性能的影响。

（2）使用JUnit测试框架中的常见断言类型进行实验。

Approach	Total	AssertType								
		Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
CodeT5	**12,791 (48%)**	**5,916 (47%)**	**1,692 (46%)**	**1,532 (43%)**	**816 (64%)**	**580 (54%)**	**404 (55%)**	**178 (49%)**	**169 (53%)**	**1,504 (50%)**
CodeGPT	10,276 (39%)	4,718 (38%)	1,344 (37%)	1,185 (34%)	653 (51%)	466 (44%)	348 (47%)	130 (36%)	125 (39%)	1,307 (43%)
CodeBERT	11,224 (42%)	5,076 (40%)	1,538 (42%)	1,266 (36%)	769 (60%)	517 (48%)	376 (51%)	155 (43%)	142 (45%)	1,385 (46%)
GraphCodeBERT	10,969 (41%)	4,999 (40%)	1,472 (40%)	1,244 (35%)	762 (59%)	496 (46%)	390 (53%)	138 (38%)	146 (46%)	1,322 (44%)
UniXcoder	11,011 (41%)	5,032 (40%)	1,508 (41%)	1,230 (35%)	756 (59%)	515 (48%)	345 (47%)	145 (40%)	146 (46%)	1,334 (44%)
Average	42.40%	41.00%	41.37%	36.56%	58.51%	48.07%	50.70%	41.22%	45.64%	45.23%

表：不同LLM在每种断言类型下的详细统计数据

②结果：

（1）CodeT5在各种断言类型上表现最佳，特别是在NotNull（58.51%）、False（48.07%）和Null（50.70%）类型上。

（2）对于Equals（41.00%）和ArrayEquals（41.22%）类型的断言，生成准确性较低。

自动断言生成-①面向单元测试场景的大模型断言生成能力探索

■ 讨论3：测试长度和断言长度的影响

①设计： （1）研究了输入测试长度和输出断言长度对LLMs预测准确性的影响。

（2）探讨了LLMs在生成不同长度测试和断言时的性能。

#Tokens		CodeT5	CodeGPT	CodeBERT	GraphCodeBERT	UniXcoder	Average
≤ 500	0-100	43.94%	35.16%	38.67%	37.80%	37.50%	38.61%
	100-200	49.98%	40.05%	43.44%	42.43%	42.80%	43.74%
	200-300	51.35%	41.62%	46.20%	44.40%	45.02%	45.72%
	300-400	54.21%	43.75%	48.19%	47.37%	48.60%	48.42%
	400-500	50.74%	42.43%	45.62%	45.16%	43.91%	45.57%
	Average	50.04%	40.60%	44.42%	43.43%	43.57%	44.41%
> 500		49.29%	38.98%	40.76%	41.13%	41.57%	42.35%

表：五个LLM在生成不同长度断言时的比较结果

②结果：

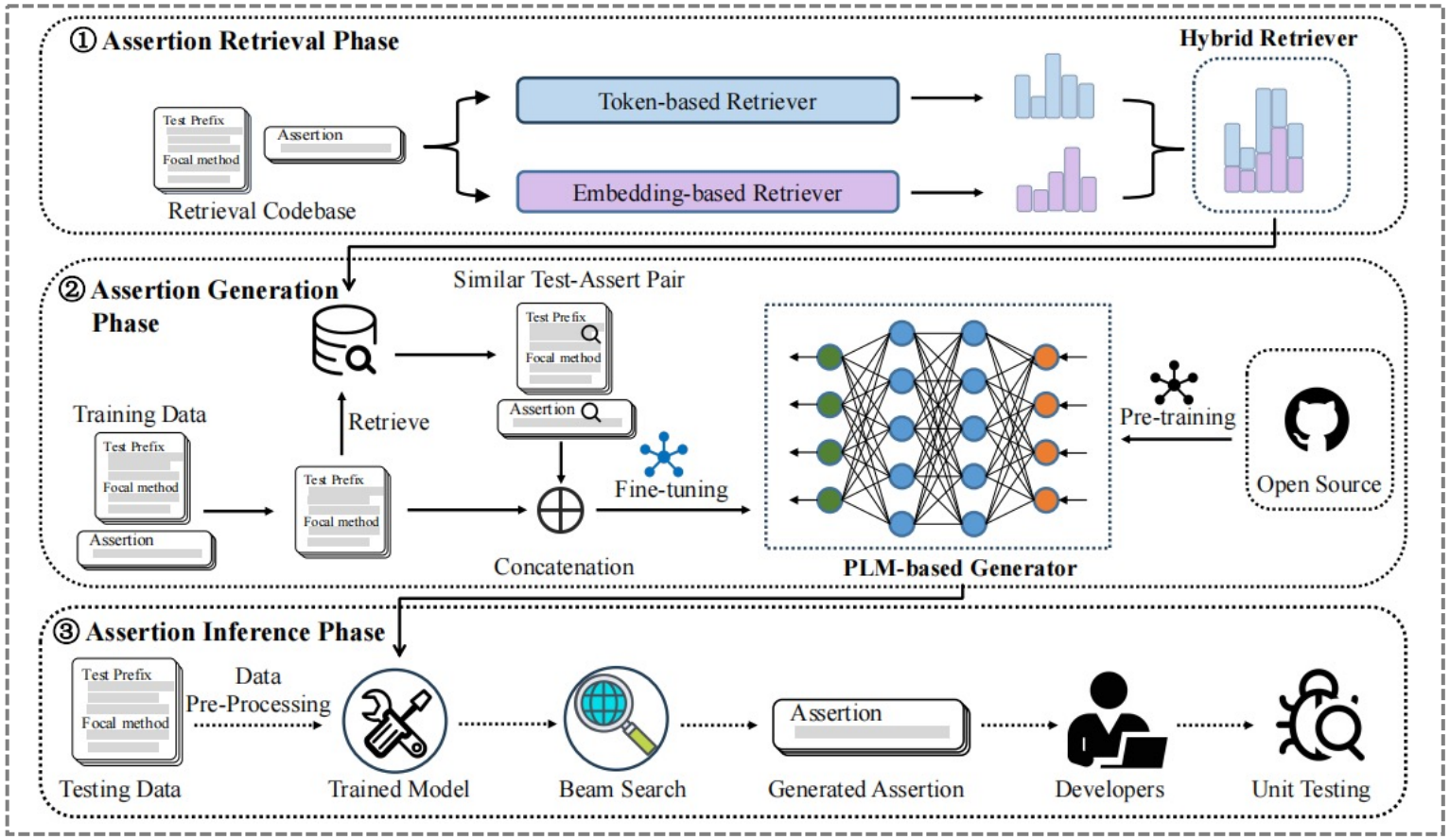
（1）输入测试长度在300-400 tokens时，LLMs的表现最佳。

（2）输出断言长度小于100 tokens时，LLMs的生成准确性较高，如CodeT5在0-20 tokens长度下的预测准确性为61.33%。

自动断言生成-②基于混合检索增强的单元测试断言生成

RetriGen从检索优化角度，将传统信息检索和大模型的代码理解能力结合，设计了一种基于代码字符-嵌入向量混合检索的断言生成框架

RetriGen首先构建一个混合断言检索器，从外部代码库中搜索最相关的测试-断言对。检索过程既考虑了基于token和基于Embedding的断言检索，既有词汇相似性也有语义相似性。然后，RetriGen将断言生成作为序列到序列任务，并设计了一个基于预训练语言模型的断言生成器，以预测与检索到的断言相一致的断言。



该方法解决之前方法无法同时考虑代码语法和语义相似性的问题。

自动断言生成-②基于混合检索增强的单元测试断言生成

定义1：深度断言生成

- 给定一个包含m个代码token的焦点测试输入 $FT_i = [ft_1, \cdots, f_m]$和包含n个代码token的断言输出$A_i=[a_1, \cdots, a_n]$，断言生成问题形式化为最大化条件概率，即$A_i$作为正确断言的可能性：

$$P_\theta(A_i|FT_i) = \prod_{j=1}^{n} P_\theta(a_j|a_1, \cdots, a_{j-1}; ft_1, \cdots, ft_m)$$

其中： FT_i是测试前缀，A_i是断言，P_θ是模型参数，a_j是断言的第j个标记 。

定义2：检索增强的深度断言生成

- 假设C = $(FT'_j, A'_j)_{j=1}^{|C|}$ 是一个包含大量历史测试-断言对的外部代码库，其中 FT'_j 和 A'_j 分别表示第j个历史焦点测试及其对应的断言。
- 给定数据集D中的一个焦点测试 FT_i，检索器从代码库C中搜索最相关的焦点测试 FT'_j及其断言A'_j 。
- 原始焦点测试输入 FT_i与检索到的断言连接形成新输入序列 $\hat{FT}_i = FT_i \oplus A'_j$ ，其中⊕表示连接操作。
- 最终，断言生成器尝试通过学习以下概率从 $\hat{FT}_i$ 生成A_i ：

$$P_\theta(A_i|\hat{FT}_i) = \prod_{j=1}^{n} P_\theta(a_j|a_1, \cdots, a_{j-1}; \underbrace{FT_i}_{Original}; \underbrace{a'_1, \cdots, a'_z}_{Augmented})$$

其中：P_θ是检索模型参数， FT_i是测试前缀， A_i 是断言

自动断言生成-②基于混合检索增强的单元测试断言生成

基于token的检索器：

- RetriGen的混合断言检索组件包括基于token的检索器和基于Embedding的检索器，以同时考虑词汇和语义相似性。
- 基于token的检索器使用稀疏策略IR来搜索与查询focal-test在词汇上相似的断言。它通过分词数据集和代码库中的所有focal-test，并删除重复token，以提高检索效率。
- 然后，使用Jaccard系数计算词汇相似性。Jaccard系数通过计算两个向量的重叠和唯一token的数量，来测量它们之间的相似性。

$$Jac(FT_i, FT_j) = \frac{|S(FT_i) \cap S(FT_j)|}{|S(FT_i) \cup S(FT_j)|}$$

公式定义了Jaccard相似性的计算方法，用于确定两个focal-test之间的相似程度。

(其中$S(FT_i)$和$S(FT_j)$分别是两个 focal-test FT_i和FT_j的代码token集。)

自动断言生成-②基于混合检索增强的单元测试断言生成

基于Embedding的检索器：

- RetriGen的基于Embedding的检索器使用了预训练的**CodeLlama模型**，直接利用其提供的语义Embedding来进行断言检索。
- 与传统方法从零开始训练模型不同，RetriGen**直接使用预训练模型**，利用其对大量代码的训练经验来获取有意义的语义Embedding。
- RetriGen通过将focal-test代码分词并**转化为向量**，然后计算这些向量之间的余弦相似性来评估断言的语义相关性。余弦相似性是一种常用的相似性度量方法，通过计算向量之间的角度余弦值来确定它们的相似程度。

$$Cos(FT_i, FT_j) = \frac{\mathbf{FT_i} \cdot \mathbf{FT_j}}{\|\mathbf{FT_i}\| \|\mathbf{FT_j}\|}$$

公式定义了余弦相似性的计算方法

(其中FT_i和FT_j分别是两个 focal-test FT_i和FT_j的Embedding。)

自动断言生成-②基于混合检索增强的单元测试断言生成

混合检索器：

- 混合检索器**结合**了基于token的检索器和基于Embedding的检索器的优点。
- 基于token的检索器侧重于代码的**词汇相似性。**
- 基于Embedding的检索器侧重于代码的**语义相似性**。
- 通过将这两种方法结合起来，混合检索器能够更全面地评估两个焦点测试之间的相似性。
- 公式3展示了组合相似度分数的计算方法，其中Jaccard相似度用于词汇相似性，余弦相似度（cos）用于语义相似性，λ是平衡这两种相似度的权重参数。最终，选择相似度分数**最高的测试断言**对来指导断言生成器生成新的断言。

$$Sim(FT_i, FTj) = Jac(FT_i, FTj) + \lambda Cos(FT_i, FTj)$$

自动断言生成-②基于混合检索增强的单元测试断言生成

断言生成组件

- 基础模型：RetriGen 使用 CodeT5 作为基础模型，根据 focal-test FT_i和检索到的断言 A'_j 生成断言。CodeT5 是针对源代码优化的编码-解码预训练模型。
- 输入表示：输入为 $\widehat{FT}_i = FT_i \oplus A'_j$ ，生成器通过序列到序列学习从 $\widehat{FT}_i$ 生成输出 A_i 。
- 模型架构：包括编码器和解码器，最后通过 softmax 激活的线性层输出。代码通过 BPE 标记器处理，生成词嵌入向量，并经过编码器和解码器生成断言概率分布。
- 损失函数

使用交叉熵最小化预测断言和真实断言之间的差异

$$\mathcal{L} = -\sum_{i=1}^{|\mathcal{D}|} \log(P_\theta(A_i|\hat{FT}_i))$$

断言推理

- 在模型推理阶段，一旦生成模型训练完成，RetriGen 使用集束搜索策略，根据词汇表的概率分布生成候选断言的排序列表。
- 束搜索选择概率最高的候选断言。
- 生成的断言可以自动或手动评估其正确性，供单元测试使用。

自动断言生成-②基于混合检索增强的单元测试断言生成

■ 研究问题:

RQ1: RetriGen与最先进的断言生成方法相比

RQ2: 不同的组件对RetriGen的整体有效性的影响分析

RQ3: 当使用其他先进的LLMs时分析RetriGen的可扩展性

■ 数据集:

$Data_{old}$: 包含250万个测试方法，经过预处理后，最终保留了156,760个样本。

$Data_{new}$: 是$Data_{old}$的扩展数据集，总共包含265,420个数据项，按8:1:1的比例划分为训练集、验证集和测试集。

表 $Data_{new}$和$Data_{old}$数据集

AssertType	Total	Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
$Data_{old}$	15,676	7,866 (50%)	2,783 (18%)	1,441 (9%)	1,162 (7%)	1,006 (6%)	798 (5%)	307 (2%)	311 (2%)	2 (0%)
$Data_{new}$	26,542	12,557 (47%)	3,652 (14%)	3,532 (13%)	1,284 (5%)	1,071 (4%)	735 (3%)	362 (1%)	319 (1%)	3,030 (11%)

■ 基线方法:

- 现有的AG技术包括: ATLAS、 IR_{ar} 、 RA^{H}_{adapt} 、 RA^{NN}_{adapt}、Integration,、EditAS

自动断言生成-②基于混合检索增强的单元测试断言生成

RQ1：RetriGen与最先进的断言生成方法相比

表 RetriGen与最先进的自动生成（AG）方法的比较

Appraoch	$Data_{old}$		$Data_{new}$	
	Accuracy	**CodeBLEU**	**Accuracy**	**CodeBLEU**
ATLAS	31.42% (↑103.98%)	63.60% (↑25.46%)	21.66% (↑136.47%)	37.91% (↑75.89%)
IR_{ar}	36.26% (↑76.75%)	71.03% (↑12.33%)	37.90% (↑35.15%)	62.67% (↑6.40%)
RA^{H}_{adapt}	40.97% (↑56.43%)	72.46% (↑10.12%)	39.65% (↑29.18%)	63.66% (↑4.74%)
RA^{NN}_{adapt}	43.63% (↑58.13%)	72.12% (↑10.64%)	40.53% (↑17.40%)	63.19% (↑5.52%)
Integration	46.54% (↑37.71%)	73.29% (↑8.87%)	42.20% (↑21.37%)	63.00% (↑5.84%)
EDITAS	53.46% (↑19.88%)	77.00% (↑3.62%)	44.36% (↑15.46%)	64.40% (↑3.54%)
RetriGen	**64.09%**	**79.79%**	**51.22%**	**66.68%**

- RetriGen 在两个数据集上的平均预测**准确性为 57.66%**，CodeBLEU **得分为 73.24%**，相比所有基线分别**提高了 50.66% 和 14.14%**。
- 与 ATLAS 比较： 在 $Data_{old}$ 和 $Data_{new}$ 数据集上的预测准确性分别达到了 **64.09% 和 51.22%**，分别显著提高了 **103.98% 和 136.47%**；CodeBLEU 提升了 **25.46% 和 75.89%**。
- 与基于检索的方法比较： 在两个数据集上平均提高了 **55.95%、42.81% 和 37.76%** 的准确性，相比 IR_{ar} 、RA^{H}_{adapt}和RA^{NN}_{adapt}。
- 与集成方法比较： 在 $Data_{old}$ 上的预测准确性分别提高了 **37.71% 和 19.88%**，在 $Data_{new}$ 上分别提高了 **21.37% 和 15.46%**。

自动断言生成-②基于混合检索增强的单元测试断言生成

表 RetriGen和基线在每种断言类型下的详细统计数据

Dataset	Approach	Total	AssertType								
			Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
$Data_{old}$	*ATLAS*	4,925 (31%)	2,501 (32%)	966 (35%)	248 (17%)	598 (51%)	229 (23%)	236 (30%)	100 (33%)	47 (15%)	0 (0%)
	IR_{ar}	5,684 (36%)	2,957 (38%)	1,039 (37%)	449 (31%)	439 (38%)	314 (31%)	285 (36%)	111 (36%)	89 (29%)	**1 (50%)**
	RA^{H}_{adapt}	6,423 (41%)	3,300 (42%)	1,151 (41%)	536 (37%)	553 (48%)	335 (33%)	316 (40%)	120 (39%)	111 (36%)	**1 (50%)**
	RA^{NN}_{adapt}	6,839 (44%)	3,509 (45%)	1,225 (44%)	551 (38%)	610 (52%)	342 (34%)	341 (43%)	134 (44%)	126 (41%)	**1 (50%)**
	Integration	7,295 (47%)	3,714 (47%)	1,333 (48%)	546 (38%)	724 (62%)	348 (35%)	352 (44%)	148 (48%)	129 (41%)	**1 (50%)**
	EDITAS	8,380 (53%)	4,131 (53%)	1,581 (57%)	526 (36%)	807 (69%)	577 (57%)	469 (59%)	167 (54%)	122 (39%)	0 (0%)
	RetriGen	**10042 (64%)**	**5027 (64%)**	**1791 (64%)**	**804 (56%)**	**820 (71%)**	**654 (65%)**	**568 (71%)**	**184 (60%)**	**194 (62%)**	0(0%)
$Data_{new}$	*ATLAS*	5,749 (22%)	2,900 (23%)	619 (17%)	537 (15%)	388 (30%)	126 (12%)	85 (12%)	47 (13%)	37 (12%)	1,010 (33%)
	IR_{ar}	10,059 (38%)	4,664 (37%)	1,436 (39%)	1,070 (30%)	600 (47%)	394 (37%)	286 (39%)	147 (41%)	113 (35%)	1,349 (45%)
	RA^{H}_{adapt}	10,525 (40%)	4,882 (39%)	1,487 (41%)	1,142 (32%)	651 (51%)	403 (38%)	297 (40%)	154 (43%)	121 (38%)	1,388 (46%)
	RA^{NN}_{adapt}	10,758 (41%)	4,988 (40%)	1,526 (42%)	1,161 (33%)	691 (54%)	401 (37%)	308 (42%)	162 (45%)	126 (39%)	1,395 (46%)
	Integration	11,201 (42%)	5,248 (42%)	1,566 (43%)	1,196 (34%)	711 (55%)	401 (37%)	313 (43%)	162 (45%)	128 (40%)	1,476 (49%)
	EDITAS	11,773 (44%)	5,339 (42%)	1,702 (47%)	1,304 (37%)	800 (62%)	523 (49%)	376 (51%)	172 (47%)	139 (44%)	1,418 (47%)
	RetriGen	**13590 (51%)**	**6294 (50%)**	**1860 (51%)**	**1588 (45%)**	**840 (65%)**	**609 (57%)**	**433 (59%)**	**195 (54%)**	**176 (55%)**	**1595 (53%)**

- 不同断言类型的有效性：
- RetriGen 在两个数据集上的所有标准 JUnit 断言类型上均**优于所有基线**。
- 对于最常见的 Equals 类型，RetriGen 在两个数据集上的预测准确率分别为 64% 和 50%，相比表现最好的 EDITAS 分别**提高了 20.75% 和 19.05%**。
- 在 Other 断言类型中，RetriGen 在 $Data_{new}$ 上生成了 1595 个正确的断言，相比表现最好的 Integration **提高了 8.16%**。

自动断言生成-②基于混合检索增强的单元测试断言生成

- 案例研究：展示了两个实际项目中的断言示例，分别说明 RetriGen 的**检索和生成能力**。相比现有方法，RetriGen 能够**捕捉到语义差异**，并应用相应的编辑操作**生成正确的断言**。

Input Focal-Test

```
//test prefix
testGettersNotNull() {
    org.semanticweb.owlapi.change.OWLOntology
ChangeRecord record = new org.semanticweb.owl
api.change.OWLOntologyChangeRecord(mockOntolo
gyID, mockChangeData);
    "<AssertPlaceHolder>";
}
//focal method:
getOntologyID() {
    return ontology.getOntologyID();
}
//assertion (ground truth)
org.junit.Assert.assertNotNull(record.getOnto
logyID())
```

Focal-Test Retrieved by AG-RAG

```
//test prefix
testGettersNotNull() {
    org.semanticweb.owlapi.change.RemoveAxiom
Data data = new org.semanticweb.owlapi.change.
RemoveAxiomData(mockAxiom);
    "" < AssertPlaceHolder > "";
}

//focal Method:
getAxiom() {
    return verifyNotNull(axiom);
}
//assertion
org.junit.Assert.assertNotNull(data.getAxiom
())
```

Focal-Test Retrieved by Baselines

```
//test prefix
testEquals() {
org.semanticweb.owlapi.change.OWLOntologyChangeRecord record1 = new org.semanticweb.owlapi.c
hange.OWLOntologyChangeRecord(mockOntologyID, mockChangeData);
org.semanticweb.owlapi.change.OWLOntologyChangeRecord record2 = new org.semanticweb.owlapi.c
hange.o(mockOntologyID, mockChangeData);"<AssertPlaceHolder>";}
//focal method: …
//assertion
org.junit.Assert.assertEquals(record1, record2)
```

IR_{ar}, RA^{NN}_{adapt}: `org.junit.Assert.assertEquals(record1, record2)`

RA^{H}_{adapt}: `org.junit.Assert.assertEquals(record, ontology)`

Intergation, ATLAS: `org.junit.Assert.assertEquals(mockChangeData.ontology, record.getOntology ID())`

EDITAS: `org.junit.Assert.assertNull(record.getOntologyID())`

RetriGen: `org.junit.Assert.assertNotNull(record.getOntologyID())`

Input Focal-Test

```
//test prefix
test25() {
    byte[] expected = new byte[] {
        ((byte)(0)), ((byte)(0))
    };
    "<AssertPlaceHolder>";
}
//focal method:
build_filler(int) {
    return com.openddal.server.mysql.p
roto.Proto.build_filler(len, ((byte)
(0)));
}
//assertion (ground truth)
org.junit.Assert.assertArrayEquals(exp
ected, com.openddal.server.mysql.proto.
Proto.build_filler(2))
```

Focal-Test by AG-RAG and Baselines

```
//test prefix
test24() {
    byte[] expected = new byte[] {
        ((byte)(0))
    };
    "<AssertPlaceHolder>";
}
//focal method:
build_filler(int) {
    return com.openddal.server.mysql.p
roto.Proto.build_filler(len, ((byte)
(0)));
}
//assertion
org.junit.Assert.assertArrayEquals(exp
ected, com.openddal.server.mysql.proto.
Proto.build_filler(1))
```

IR_{ar}, RA^{H}_{adapt}, RA^{NN}_{adapt}, *Intergation*, EDITAS:

```
org.junit.Assert.assertArrayEquals(expected, com.openddal.server.mysql.proto.
Proto.build_filler(1))
```

RetriGen：

```
org.junit.Assert.assertArrayEquals(expected, com.openddal.server.mysql.proto.
Proto.build_filler(2))
```

自动断言生成-②基于混合检索增强的单元测试断言生成

💡 **Finding：**

（1）RetriGen在准确性和CodeBLEU方面显著超过了所有基线，跨两个数据集的平均准确性提高了58.81%；

（2）RetriGen在所有标准JUnit断言类型上一贯超过所有基线，例如，在生成Equals类型的断言上，相比于ED-ITAs，分别提高了20.75%和19.05%；

（3）RetriGen在两个数据集上生成了大量独特断言，这些断言是所有基线均未能生成的。

自动断言生成-②基于混合检索增强的单元测试断言生成

RQ2：不同的组件对RetriGen的整体有效性的影响分析

表 不同检索器选择下RetriGen的有效性

Appraoch	$Data_{old}$		$Data_{new}$	
	Accuracy	CodeBLEU	Accuracy	CodeBLEU
$RetriGen_{none}$	58.71% (↑9.16%)	74.91% (↑6.51%)	48.19% (↑6.29%)	63.40% (↑5.17%)
$RetriGen_{token}$	63.37% (↑1.14%)	78.54% (↑1.59%)	51.15% (↑0.14%)	66.54% (↑0.21%)
$RetriGen_{embed}$	63.11% (↑1.55%)	79.01% (↑0.99%)	50.74% (↑0.95%)	66.09% (↑0.89%)
RetriGen	**64.09%**	**79.79%**	**51.22%**	**66.68%**

- 在$Data_{old}$数据集上，RetriGen的混合检索器**提高了预测准确率**在准确率和CodeBLEU指标上，RetriGen在两个数据集（$Data_{old}$和$Data_{new}$）上**均优于所有变体**，显示出每个组件的重要性。 即使没有任何检索器，RetriGen在两个数据集上的预测准确率分别为58.71%和48.19%，比基准ATLAS分别**提高了86.86%和122.48%**。

> 💡 **Finding：所有组件（例如，基于Token的和基于Embedding的检索器）对RetriGen的性能都有积极的贡献，根据两个评价指标度量，在两个广泛采用的数据集上创下了新纪录。**

自动断言生成-②基于混合检索增强的单元测试断言生成

- **RQ3：当使用其他先进的LLMs时，RetriGen的可扩展性如何?**

表 不同LLM作为断言生成器的有效性

Appraoch	$Data_{old}$		$Data_{new}$	
	Accuracy	CodeBLEU	Accuracy	CodeBLEU
GraphCodebert	60.25% (↑6.37%)	77.42% (↑3.06%)	46.96% (↑9.07%)	63.05% (↑5.76%)
Unixcoder	59.17% (↑8.32%)	75.77% (↑5.31%)	48.58% (↑5.43%)	63.67% (↑4.73%)
CodeBERT	58.84% (↑8.92%)	77.14% (↑3.44%)	47.92% (↑6.89%)	63.46% (↑5.07%)
CodeGPT	57.69% (↑11.09%)	77.70% (↑2.69%)	48.13% (↑6.42%)	64.08% (↑4.06%)
CodeT5 (RetriGen)	**64.09%**	**79.79%**	**51.22%**	**66.68%**

💡 **Finding：**

（1）RetriGen适用于不同的LLMs，并且可持续地达到最先进的性能，例如，五种涉及的LLMs在两个数据集上的平均准确率为54.29%，平均超过最新的EDITAS 10.86%；

（2）默认的断言生成器（即CodeT5）是自然且在单元断言生成场景中相当有效的，例如，平均在两个数据集上提高了其他LLMs的准确率7.81%。

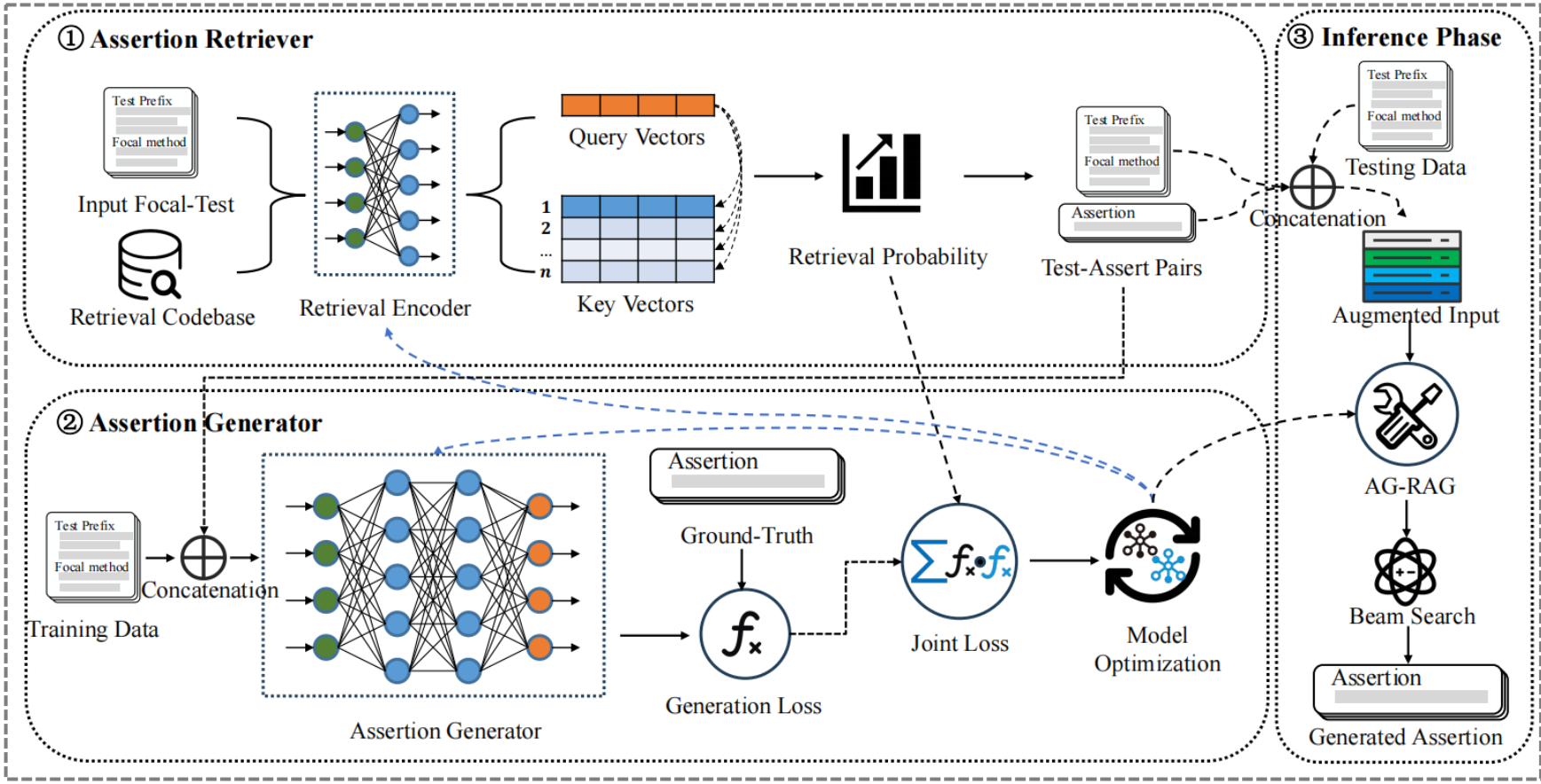
自动断言生成-③基于检索生成协同增强的单元测试断言生成

1.基于密集连接的断言检索器：设计基于编码器的密级检索器，通过嵌入向量扑捉测试代码语义信息，从外部代码库从检索相似断言。

2.基于检索增强的断言生成器：利用检索断言作为提示引导大模型断言生成的过程，以解决大模型无法及时更新、缺乏最新知识和幻觉问题，引导生成语法和语义正确测试代码。

3.基于检索生成协同优化策略：从概率角度将断言生成概率分解为检索的边缘分布和生成的条件概率，从而设计统一的损失函数同时优化两个组件。使得检索器能够返回最能指导生成过程的断言，同时生成器根据对应生成结果反馈来优化检索器。

AG-RAG从训练优化角度，设计了一种检索-生成协同优化的断言生成框架，分别利用大模型的代码理解和生成能力设计断言检索器和生成器。



该方法解决之前方法检索和生成组件独立训练对代码语义理解不充分的问题。

自动断言生成-③基于检索生成协同增强的单元测试断言生成

定义1：深度断言生成

- **输入：**焦点测试 $FT_i = [ft_1, \cdots, ft_m]$：包含单元测试的前缀代码。
- **输出：**断言 $A_i = [a_1, \cdots, a_n]$：指定测试期望结果的语句。
- **目标：**通过学习模型参数θ，最大化输入焦点测试生成正确断言的条件概率：

$$P_\theta(A_i|FT_i) = \prod_{j=1}^{n} P_\theta(a_j|a_1, \cdots, a_{j-1}; ft_1, \cdots, ft_m)$$

定义2：检索增强的深度断言生成

- **输入增强：**检索到的 TAPs（FT'_j，A'_j）：从外部代码库中检索到的与当前焦点测试语义相似的历史测试-断言对。检索机制考虑代码的语义相似性，而不仅仅是表面词汇匹配。
- **增强输入过程：**原始焦点测试 FT_i 通过与检索到的 FT'_j 和其断言 A'_j 进行组合，形成新的输入序列：$\hat{FT}_i = FT_i \oplus FT'_j \oplus A'_j$。$\oplus$表示连接操作，将检索到的相关信息与原始测试结合，提供更多上下文。
- **生成目标：**在增强后的输入上进行生成任务，提高断言的准确性

$$P_\theta(A_i|\hat{FT}_i) = \prod_{j=1}^{n} P_\theta(a_j|a_1, \cdots, a_{j-1}; \underbrace{FT_i}_{Original}; \underbrace{FT'_j; A'_j}_{Augmented})$$

自动断言生成-③基于检索生成协同增强的单元测试断言生成

1.基于密集连接的断言检索器:

目标: 从外部代码库中检索相关的测试断言模式（TAP）。

- **使用密集检索器来搜索相关的TAPs:** 检索器利用一个基于Transformer的编码器，将每个focal-test编码为固定大小的密集向量。

- **采用BPE分词器:** 将源代码划分为多个子词，基于它们的频率分布进行分割。

$$CLS'_{FT} = \frac{CLS_{FT}}{\sqrt{\sum_{j=1}^{d} CLS_{FT}^2}}$$

- **对每个焦点测试的[CLS]标记进行L2正则化:** 在编码器的最终隐藏层状态中为每个测试用例添加了一个特殊的[CLS]标记，并用这个标记的状态作为上下文嵌入。

- **评估两个焦点测试的语义相关性:** 计算两个焦点测试间的余弦相似度（即通过它们的标准化嵌入向量的内积）。

$$f_\phi\left(FT_i, FT_j\right) = \left[CLS'_{FT_i}\right]^T \left[CLS'_{FT'_j}\right]$$

自动断言生成-③基于检索生成协同增强的单元测试断言生成

2.基于检索增强的断言生成器:

目标: 根据检索到的测试-断言对（TAP）生成准确的断言。

- **编码:** 使用CodeT5编码器对输入序列进行编码，生成上下文表示。

- **组合输入（输入增强）:** 将检索到的TAP与原始测试用例组合，形成新的输入序列。

$$\hat{FT}_i = \texttt{[CLS]} \cdot X_i \cdot \backslash n \cdot // \cdot FT_j \cdot \backslash n \cdot A_j$$

- **解码:** 使用CodeT5解码器生成断言序列，通过softmax层指导的断言生成，最大化断言生成的条件概率。

- **训练:** 通过teacher forcing方法最小化交叉熵损失，优化从检索增强输入到断言生成的过程。

$$\mathcal{L}_{ce} = -\sum_{i=1}^{|\mathcal{D}|} \log(P_\theta((A_i|\hat{FT}_i))$$

自动断言生成-③基于检索生成协同增强的单元测试断言生成

3.基于检索生成协同优化策略：

目标: 优化检索器和生成器，使其更好地协同工作。

- **检索概率计算:** 将检索视为从外部代码库中检索与给定测试用例语义相似的TAP。

$$P(A_i|FT_i) = P_\phi(\mathcal{C}|FT_i) \cdot P_\theta(A_i|FT_i, \mathcal{C})$$
$$= \sum_{j=1}^{|\mathcal{C}|} \underbrace{P_\phi(TAP_j|FT_i)}_{\text{Retriever}} \cdot \underbrace{P_\theta(A_i|FT_i, TAP_j)}_{\text{Generator}}$$

$P_\emptyset$：检索模型参数
P_J：生成模型参数
C：外部代码库

- **生成概率分解:** 将生成概率分解为检索概率和检索增强概率。

$$P(A_i|FT_i) \approx \sum_{j=1}^{k} P_\phi(TAP_j|FT_i) \cdot P_\theta(A_i|FT_i, TAP_j)$$

- **概率分布:** 使用softmax函数将语义相关性转换为概率分布。

$$P_\phi(TAP_j|FT_i) = \frac{f_\phi(FT_i, FT_j)}{\sum_{k=1}^{|\mathcal{C}|} f_\phi(FT_i, FT_k)}$$

$f_\emptyset$：语义相似度
TAP_J：检索到的测试断言对
FT_I：测试前缀

- **训练目标近似:** 利用前k个检索到的TAP来近似训练目标，通过联合训练策略优化模型性能。使用交叉熵损失。

$$\mathcal{L} = \sum_{j=1}^{k} \mathcal{L}_{ce} \cdot P_\phi(TAP_j|FT_i)$$

自动断言生成-③基于检索生成协同增强的单元测试断言生成

■ 研究问题:

RQ1: AG-RAG在断言生成任务中的表现如何?

RQ2: 联合训练策略对模型性能有何影响?

RQ3: AG-RAG对不同预训练语言模型的泛化能力如何?

■ 数据集:

$Data_{old}$: 包含9000多个开源项目中的250万个开发者编写的测试方法。

$Data_{new}$: 是$Data_{old}$的扩展数据集，包含额外的108,660个样本，以更好地反映真实数据分布。

表 $Data_{new}$和$Data_{old}$数据集

AssertType	Total	Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
$Data_{old}$	15,676	7,866 (50%)	2,783 (18%)	1,441 (9%)	1,162 (7%)	1,006 (6%)	798 (5%)	307 (2%)	311 (2%)	2 (0%)
$Data_{new}$	26,542	12,557 (47%)	3,652 (14%)	3,532 (13%)	1,284 (5%)	1,071 (4%)	735 (3%)	362 (1%)	319 (1%)	3,030 (11%)

■ 基线方法:

- 现有的AG技术包括: ATLAS、 IR_{ar} 、 RA^{H}_{adapt} 、 RA^{NN}_{adapt}、Integration,、EditAS

自动断言生成-③基于检索生成协同增强的单元测试断言生成

RQ1：与现有方法的比较

表 AG-RAG与基线方法对比

Appraoch	$Data_{old}$			$Data_{new}$		
	Accuracy	**CodeBLEU**	**BLEU**	**Accuracy**	**CodeBLEU**	**BLEU**
ATLAS	31.42% (↑105.57%)	63.60% (↑27.14%)	68.51% (↑23.76%)	21.66% (↑160.06%)	37.91% (↑79.66%)	37.91% (↑92.19%)
IR_{ar}	36.26% (↑78.13%)	71.03% (↑13.84%)	71.49% (↑18.60%)	37.90% (↑48.63%)	62.67% (↑8.68%)	57.98% (↑25.66%)
RA^{H}_{adapt}	40.97% (↑57.65%)	72.46% (↑11.59%)	73.28% (↑15.71%)	39.65% (↑42.07%)	63.66% (↑6.99%)	59.81% (↑21.82%)
RA^{NN}_{adapt}	43.63% (↑59.36%)	72.12% (↑12.12%)	73.95% (↑14.66%)	40.53% (↑29.11%)	63.19% (↑7.79%)	59.81% (↑21.82%)
Integration	46.54% (↑38.78%)	73.29% (↑10.33%)	78.86% (↑7.52%)	42.20% (↑33.48%)	63.00% (↑8.11%)	60.92% (↑19.60%)
EDITAS	53.46% (↑20.82%)	77.00% (↑5.01%)	80.77% (↑4.98%)	44.36% (↑26.98%)	64.40% (↑5.76%)	63.46% (↑14.81%)
AG-RAG	**64.59%**	**80.86%**	**84.79%**	**56.33%**	**68.11%**	**72.86%**

↑ denotes performance improvement of AG-RAG against state-of-the-art baselines

- 表展示了AG-RAG与基线方法在三种指标上的比较结果。
- 总体而言，AG-RAG在两个基准上的准确率为56.33%-64.59%，CodeBLEU得分为68.11%-80.86%，BLEU得分为72.86%-84.79%，分别比所有基线高出20.82%-160.06%、5.01%-79.66%和4.98%-92.19%。

自动断言生成-③基于检索生成协同增强的单元测试断言生成

表 AG-RAG与基线在不同类型断言上的表现

Dataset	Approach	Total	AssertType								
			Equals	TRUE	That	NotNull	FALSE	Null	ArrayEquals	Same	Other
$Data_{old}$	*ATLAS*	4925(31%)	2501(32%)	966(35%)	248(17%)	598(51%)	229(23%)	236(30%)	100(33%)	47(15%)	0(0%)
	IR_{ar}	5684(36%)	2957(38%)	1039(37%)	449(31%)	439(38%)	314(31%)	285(36%)	111(36%)	89(29%)	**1(50%)**
	RA^{H}_{adapt}	6423(41%)	3300(42%)	1151(41%)	536(37%)	553(48%)	335(33%)	316(40%)	120(39%)	111(36%)	**1(50%)**
	RA^{NN}_{adapt}	6839(44%)	3509(45%)	1225(44%)	551(38%)	610(52%)	342(34%)	341(43%)	134(44%)	126(41%)	**1(50%)**
	Integration	7295(47%)	3714(47%)	1333(48%)	546(38%)	724(62%)	348(35%)	352(44%)	148(48%)	129(41%)	**1(50%)**
	EDITAS	8380(53%)	4131(53%)	1581(57%)	526(36%)	807(69%)	577(57%)	469(59%)	167(54%)	122(39%)	0(0%)
	AG-RAG	**10125(65%)**	**4993(63%)**	**1790(64%)**	**831(58%)**	**853(73%)**	**691(69%)**	**563(71%)**	**204(66%)**	**199(64%)**	**1(50%)**
$Data_{new}$	*ATLAS*	5749(22%)	2900(23%)	619(17%)	537(15%)	388(30%)	126(12%)	85(12%)	47(13%)	37(12%)	1010(33%)
	IR_{ar}	10059(38%)	4664(37%)	1436(39%)	1070(30%)	600(47%)	394(37%)	286(39%)	147(41%)	113(35%)	1349(45%)
	RA^{H}_{adapt}	10525(40%)	4882(39%)	1487(41%)	1142(32%)	651(51%)	403(38%)	297(40%)	154(43%)	121(38%)	1388(46%)
	RA^{NN}_{adapt}	10758(41%)	4988(40%)	1526(42%)	1161(33%)	691(54%)	401(37%)	308(42%)	162(45%)	126(39%)	1395(46%)
	Integration	11201(42%)	5248(42%)	1566(43%)	1196(34%)	711(55%)	401(37%)	313(43%)	162(45%)	128(40%)	1476(49%)
	EDITAS	11773(44%)	5339(42%)	1702(47%)	1304(37%)	800(62%)	523(49%)	376(51%)	172(47%)	139(44%)	1418(47%)
	AG-RAG	**14950(56%)**	**6938(55%)**	**2055(56%)**	**1832(52%)**	**884(69%)**	**676(63%)**	**447(61%)**	**203(56%)**	**188(59%)**	**1727(58%)**

💡 **Finding：AG-RAG在三个指标上显著优于所有以前的AG方法，预测准确率为56.33%-64.59%，并在两个数据集上生成了1739-2866个独特断言。**

自动断言生成-③基于检索生成协同增强的单元测试断言生成

Input Focal-Test	TAP Retrieved by AG-RAG

//test prefix
testGetConfigClass() {org.apache.eagle.app.storm.MockStormApplication mockStormApplication = new org.apache.eagle.app.storm.MockStormApplication();"<AssertPlaceHolder>";}
//focal method:
getEnvironmentType() {return org.apache.eagle.app.environment.impl.StormEnvironment.class;}
//assertion
org.junit.Assert.assertEquals(org.apache.eagle.app.environment.impl.StormEnvironment.class, mockStormApplication.getEnvironmentType())
	//test prefix
	testGetType() {notExpr = new com.huawei.streaming.expression.NotExpression(new com.huawei.streaming.expression.ConstExpression(false));"<AssertPlaceHolder>";}
	//focal method:
	getType() {return com.huawei.streaming.expression.Boolean.class;}
	//assertion
	org.junit.Assert.assertEquals(com.huawei.streaming.expression.Boolean.class, notExpr.getType())

Retrieved Assertion: org.junit.Assert.assertNotNull(org.apache.eagle.common.Version.str())	
IR_{ar}: org.junit.Assert.assertNotNull(org.apache.eagle.common.Version.str())	
RA^{H}_{adapt}: org.junit.Assert.assertNotNull(org.apache.eagle.common.Version.getEnvironmentType())	
RA^{NN}_{adapt}: org.junit.Assert.assertNotNull(org.apache.eagle.common.Version.str())	
Intergation: org.junit.Assert.assertEquals(org.apache.IDENT_0.types. IDENT_4.IDENT_5.class,IDENT_3.METHOD_2())	
EDITAS: org.junit.Assert.assertNotNull(mockStormApplication.getEnvironmentType())	
ATLAS: : org.junit.Assert.assertEquals(org.apache.IDENT_0.types. IDENT_4.IDENT_5.class,IDENT_3.METHOD_2())	
AG-RAG : org.junit.Assert.assertEquals(org.apache.eagle.app.environment.impl.StormEnvironment.class, mockStormApplication.getEnvironmentType())	

- **案例研究：**展示了来自Apache Eagle项目的一个断言示例，只有AG-RAG能够正确生成，而所有基线都未能生成。
- AG-RAG使用IR_{ar}基于词汇匹配来检索相似的断言，即使检索到的断言和查询焦点测试在功能上不完全相似。
- 其他方法的不足:RA^{H}_{adapt}、RA^{NN}_{adapt}、和EDITAS未能生成正确的断言，主要是因为它们选用了错误的断言类型（如assertNotNull）并对断言进行了错误的修改（如函数调用或参数替换）。

自动断言生成-③基于检索生成协同增强的单元测试断言生成

Input Focal-Test	Retrieved TAP

//test prefix
test_reduce_char_sequence() {char[] a = new char[] {'a','b','c'}; int result = server.reduce_char_sequence(a);"<AssertPlaceHolder>";}
//focal method:
reduce_char_sequence(char[]) {return seq.length;}
//assertion
org.junit.Assert.assertEquals(3, result)
	//test prefix
	test_reduce_empty_char_sequence() {char[] a = new char[] {}; int result = server.reduce_char_sequence(a); "<AssertPlaceHolder>";}
	//focal method:
	reduce_char_sequence(char[]) {return seq.length;}
	//assertion
	org.junit.Assert.assertEquals(0, result)

Retrieved Assertion: org.junit.Assert.assertEquals(0, result)	
IR_{ar}: org.junit.Assert.assertEquals(0, result)	
RA^{H}_{adapt}: org.junit.Assert.assertEquals(0, result)	
RA^{NN}_{adapt}: org.junit.Assert.assertEquals(0, result)	
Intergation: org.junit.Assert.assertEquals(0, result)	
EDITAS: org.junit.Assert.assertEquals(0, result)	
AG-RAG： org.junit.Assert.assertEquals(3, result)	

- **案例研究：**所有方法包括AG-RAG都检索到了相同的断言。检索到的断言在词汇上几乎正确，但AG-RAG在参数上进行细化（从“0”修改为“3”）以更精确地反映测试的实际方法。
- AG-RAG成功识别出检索到的**焦点测试与输入焦点测试之间的语义差异**，并应用了相应的编辑操作，从而生成了正确的断言。
- 其他方法的不足: 现有基线方法直接返回了检索到的断言，假设它已经是正确的。

自动断言生成-③基于检索生成协同增强的单元测试断言生成

RQ2: 联合训练策略对模型性能有何影响?

表：在AG-RAG中进行联合训练的影响

Appraoch	$Data_{old}$			$Data_{new}$		
	Accuracy	CodeBLEU	BLEU	Accuracy	CodeBLEU	BLEU
No Retriever	61.06% (↑5.78%)	77.36% (↑4.52%)	78.23% (↑8.39%)	46.00% (↑22.45%)	60.60% (↑12.39%)	62.02% (↑17.48%)
Random Retriever	59.38% (↑8.77%)	77.22% (↑4.71%)	76.74% (↑10.49%)	44.12% (↑27.67%)	59.86% (↑13.78%)	58.86% (↑23.79%)
IR Retriever	63.37% (↑1.93%)	78.54% (↑2.95%)	79.72% (↑6.36%)	51.15% (↑10.13%)	66.54% (↑2.36%)	67.41% (↑8.08%)
Pre-trained Retriever	63.15% (↑2.27%)	78.99% (↑2.37%)	79.45% (↑6.72%)	52.02% (↑8.29%)	66.44% (↑2.51%)	67.44% (↑8.04%)
Fine-tuned Retriever	64.19% (↑0.62%)	79.02% (↑2.33%)	80.19% (↑5.74%)	53.40% (↑5.48%)	67.96% (↑0.22%)	68.84% (↑5.84%)
Joint Retriever (AG-RAG)	**64.59%**	**80.86%**	**84.79%**	**56.33%**	**68.11%**	**72.86%**

表展示了我们默认联合检索器与基线方法的比较结果。总体而言，**默认的联合检索器**在所有指标和数据集上**表现最佳。**

💡 **Finding：我们的联合训练策略在三个指标上对AG-RAG的性能有积极贡献，例如，在$Data_{old}$数据集上，相比于无检索器和预训练检索器，分别提高了22.45%和8.28%的准确率。**

自动断言生成-③基于检索生成协同增强的单元测试断言生成

- **RQ3: AG-RAG对不同预训练语言模型的泛化能力如何?**

表：不同LLMs在AG-RAG中的有效性。

PLMs	$Data_{old}$	$Data_{new}$	Average
GraphCodeBERT	54.58% (↑2.09%)	50.11% (↑12.96%)	52.34%
Unixcoder	59.54% (↑11.37%)	51.41% (↑15.89%)	55.47%
CodeBERT	55.22% (↑3.29%)	52.73% (↑18.86%)	53.97%
CodeT5	**64.59% (↑20.82%)**	**56.33% (↑26.97%)**	**60.46%**
Average	58.48% (↑9.39%)	52.64% (↑18.67%)	55.56%

💡 **Finding：AG-RAG对不同的LLM具有通用性，在两个数据集上的平均准确率分别为58.48%和52.64%，而CodeT5在促进断言检索和生成方面表现显著，准确率分别为64.59%和56.33%。**

PART 04

应用验证

TestART应用验证

待测工程说明：

- **开源项目包括**：Defects4J中的五个常用项目：

  - Commons-Lang：1728待测方法
  - Gson：378待测方法
  - Commons-Csv：137待测方法
  - Commons-Cli：177待测方法
  - jfreechart：5772待测方法

  总计一共提取了 8192 个待测方法

- **UTBench包括（华为内部数据集）**：

  - JavaBasicDemo：82个待测方法，均是较为简单的方法
  - mall：332个待测方法
  - MicroServiceDemo：36个待测方法
  - UTService：218个待测方法

  总计668个待测方法

内部部署模型：

- **Pangu-Coder**

TestART应用验证-实验结果

数据集（待测工程）	生成模型	修复模型	工具	编译通过率	总行覆盖率	总分支覆盖率	通过测试的行覆盖率	通过测试的分支覆盖率
开源项目	Pangu L1.9	/	/	34.6%	29.55%	28.93%	74.72%	73.16%
开源项目	Pangu L1.9	Pangu L1.9	TestART	61.92%	48.82%	47.95%	95.67%	93.96%
UTBench	Pangu L1.9	/	/	49.97%	45.90%	45.18%	74.63%	73.75%
UTBench	Pangu L1.9	Pangu L1.9	TestART	65.33%	56.69%	56.00%	95.85%	94.94%

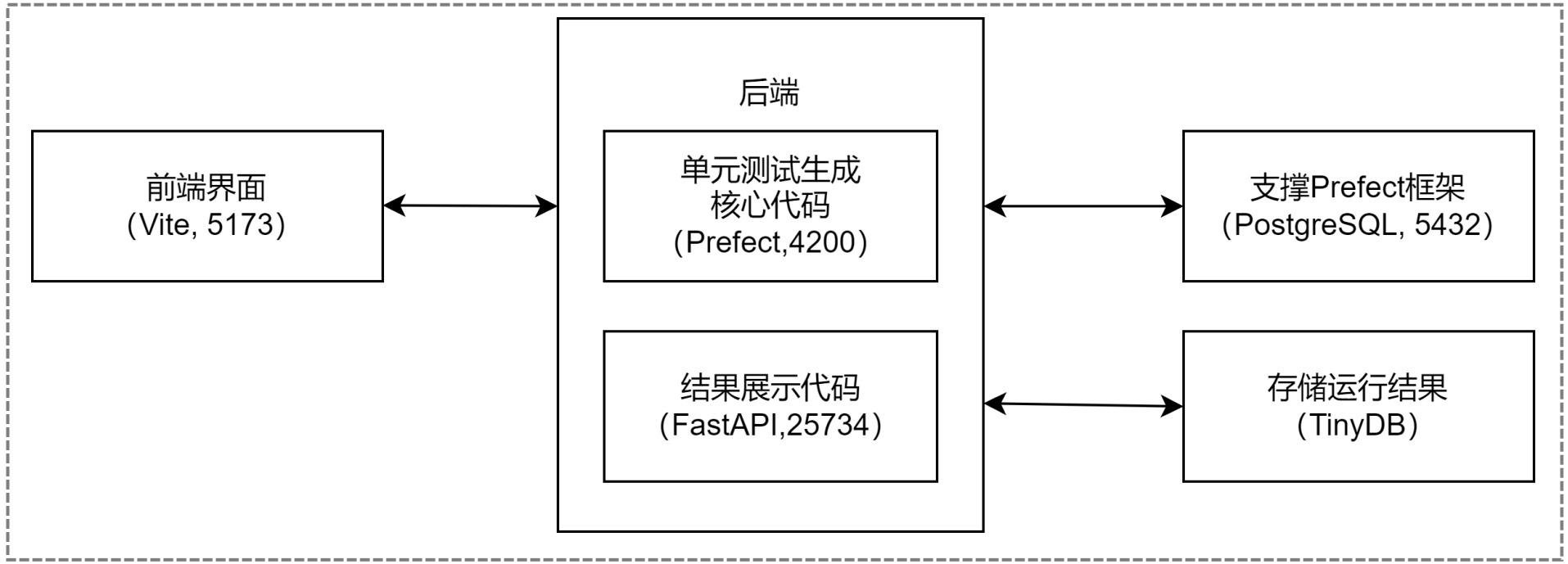
表 TestART应用验证

- 待测工程为开源项目时，使用TestART后，覆盖方法数、覆盖语句数、通过测试的覆盖率和分支覆盖率均有显著提升。
  例如，覆盖方法数从34.6%提升到61.92%，通过测试的覆盖率从74.72%提升到95.67%。
- 待测工程为为UTBench时，使用TestART同样显著提升了所有指标。
  例如，覆盖方法数从49.97%提升到65.33%，通过测试的覆盖率从74.63%提升到95.85%。

Finding：TestART实际运用于Pangu L1.9 （Pangu-Coder）模型进行修复后修复结果效果良好。这些结果表明，TestART工具在两种测试配置中均显著提高了代码覆盖率和测试质量。在实际应用中，这表明TestART能有效增强现有的测试框架。

TestART应用验证

- **工具说明：该工具用于Java单元测试的生成，以docker镜像的方式给出，包含以下几部分组成。**



前端界面。基于Prefect-UI（https://github.com/PrefectHQ/ui），为工具提供可视化展示。
单元测试生成核心代码。基于Prefect框架，为工具提供基本功能。
结果展示代码，用于展示任务运行结果，使用FastAPI提供接口。

TestART应用验证

- **工具使用：**我们的工具得到了部署，也可以通过前端进行交互使用。前端界面还提供了丰富的功能选项，允许用户根据自己的需求进行定制和调整，进一步提升了工具的实用性

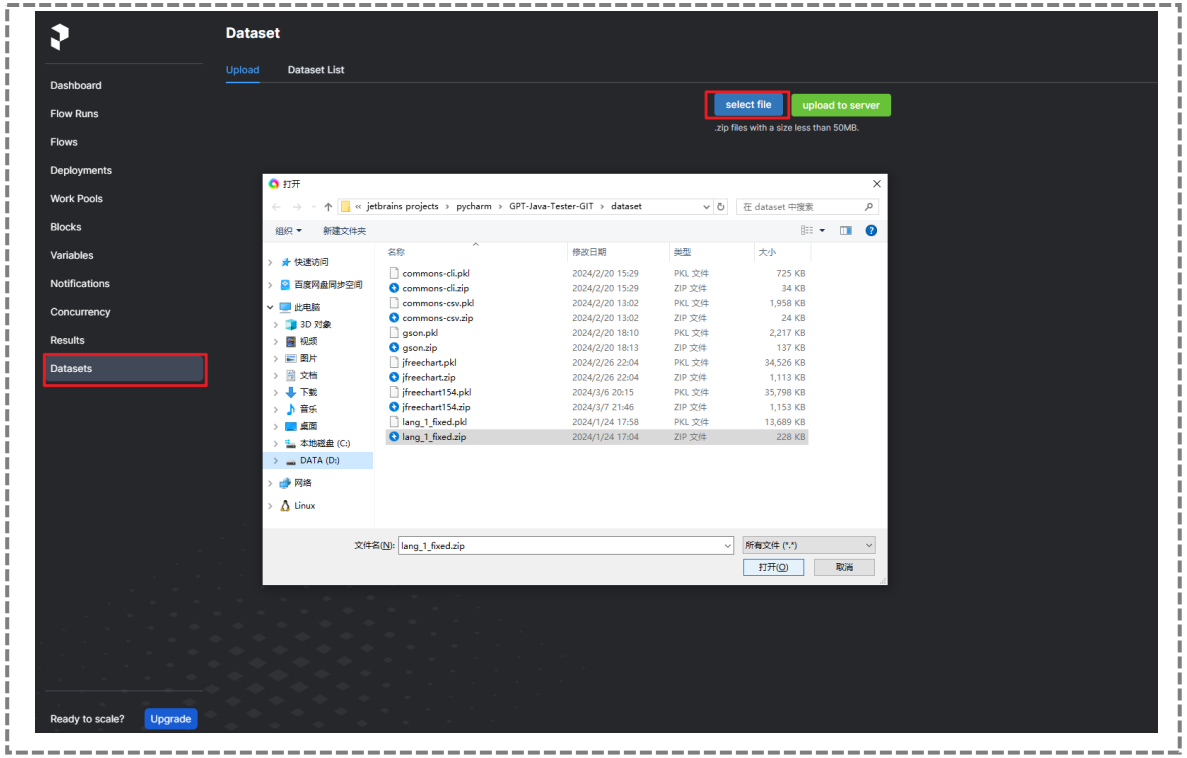


图 上传数据集

```
set_name test --dataset_start_index 0 --dataset_end_index 0
20:40:00.867 | INFO    | prefect.engine - Created flow run 'chirpy-sawfish' for flow 'UnitTest-Dataset'
20:40:00.869 | INFO    | Flow run 'chirpy-sawfish' - View at http://192.168.203.20:4200/flow-runs/flow-run/4c88a026-30a2-49bd-bd0c-11c67310068a
20:40:00.988 | INFO    | Flow run 'chirpy-sawfish' - flow count: 1
20:40:01.054 | INFO    | Flow run 'chirpy-sawfish' - Created subflow run 'stoic-salmon' for flow 'UnitTest'
20:40:01.055 | INFO    | Flow run 'stoic-salmon' - View at http://192.168.203.20:4200/flow-runs/flow-run/e10a4e25-684c-497d-9c09-5eb5498c2cf6
20:40:01.096 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - Running Dataset Id: 0
20:40:01.097 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - e10a4e25-684c-497d-9c09-5eb5498c2cf6 waiting for resource allocation... <asyncio.locks.Semaphore object at 0x789bfda96010 [unlocked, value:3]>
20:40:01.098 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - resource allocated : /root/GPT-Java-Tester/project/TestJavaCode, 2 left
20:40:01.099 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - 资源分配: /root/GPT-Java-Tester/project/TestJavaCode
20:40:01.126 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - Created task run 'init_clover_executor_TASK-0' for task 'init_clover_executor_TASK'
20:40:01.127 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - Executing 'init_clover_executor_TASK-0' immediately...
20:40:01.167 | INFO    | Task run 'Init Executor' - cleaning...
20:40:02.622 | INFO    | Task run 'Init Executor' - b"[INFO] Scanning for projects...\n[WARNING] \n[WARNING] Some problems were encountered while building the effective model for org.example:TestJavaCode:jar:1.0-SNAPSHOT\n[WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-surefire-plugin is missing. @ line 89, column 21\n[WARNING] \n[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.\n[WARNING] \n[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.\n[WARNING] \n[INFO] \n[INFO] ----------------------< org.example:TestJavaCode >----------------------\n[INFO] Building TestJavaCode 1.0-SNAPSHOT\n[INFO]    from pom.xml\n[INFO] --------------------------------[ jar ]---------------------------------\n[INFO] \n[INFO] --- clean:3.2.0:clean (default-clean) @ TestJavaCode ---\n[INFO] Deleting /root/GPT-Java-Tester/project/TestJavaCode/target\n[INFO] ------------------------------------------------------------------------\n[INFO] BUILD SUCCESS\n[INFO] ------------------------------------------------------------------------\n[INFO] Total time:  0.223 s\n[INFO] Finished at: 2024-05-05T20:40:02+08:00\n[INFO] ------------------------------------------------------------------------\n"
20:40:02.624 | INFO    | Task run 'Init Executor' - config initializing...
20:40:02.626 | WARNING | Task run 'Init Executor' - test folder /root/GPT-Java-Tester/project/TestJavaCode/src/test/java/example not exists, created
20:40:02.626 | WARNING | Task run 'Init Executor' - source folder /root/GPT-Java-Tester/project/TestJavaCode/src/main/java/example not exists, created
20:40:02.627 | INFO    | Task run 'Init Executor' - config init done
20:40:02.628 | INFO    | Task run 'Init Executor' - code writing...
20:40:02.646 | INFO    | Task run 'Init Executor' - Finished in state Completed()
20:40:02.664 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - Created task run 'unzip_dataset_TASK-0' for task 'unzip_dataset_TASK'
20:40:02.664 | INFO    | Flow run 'BigNumber#bigNumberSimpleMulti(String, String) : String' - Executing 'unzip_dataset_TASK-0' immediately...
```

图 运行单元测试任务

TestART应用验证

工具使用：

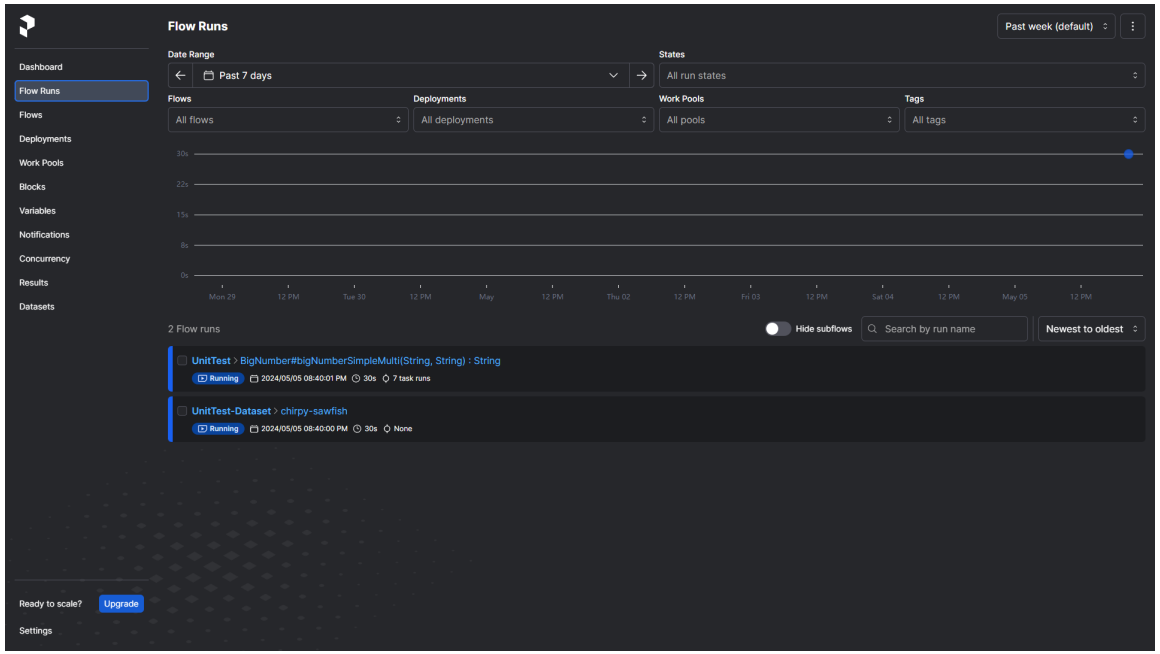


图 查看任务执行过程

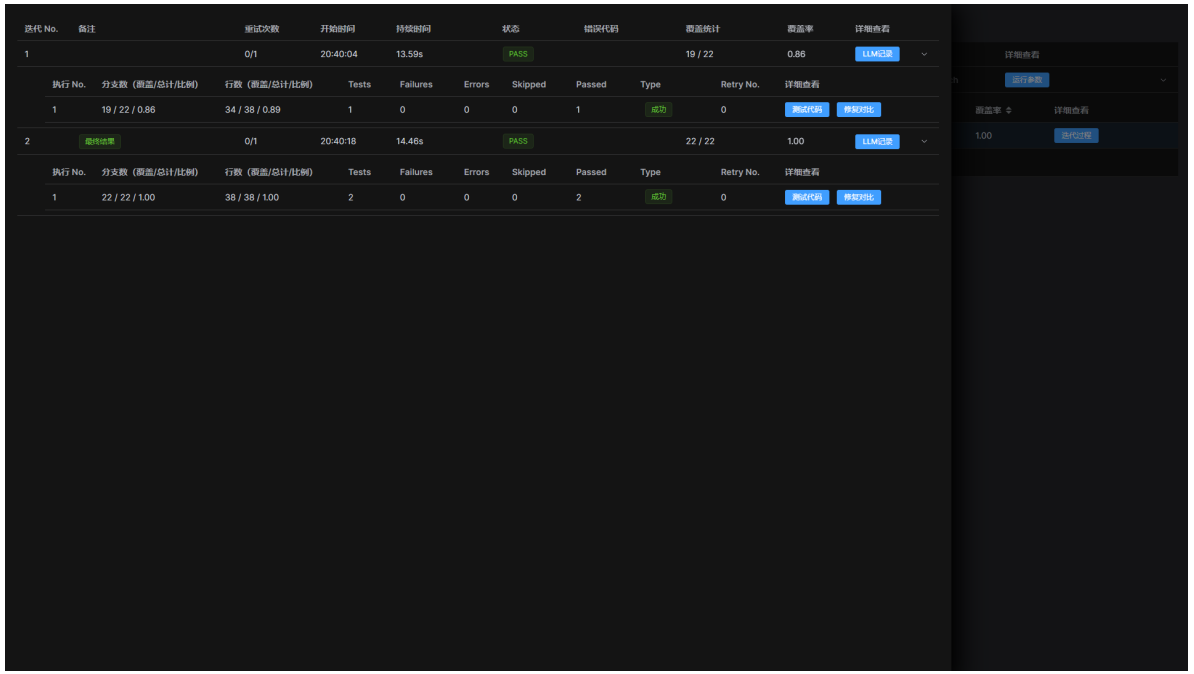


图 查看任务结果

PART 05

总结与展望

单元测试生成-总结与展望

总结：

- 大模型（如LLMs）在自动生成单元测试方面展现了显著的潜力。通过研究如TestART等最新方法，我们可以看到这些模型在生成能力和自动修复方面的结合可以**显著提高生成的测试用例的通过率和覆盖率。**
- 具体来说，TestART通过模板修复和迭代生成策略，成功地克服了LLMs在生成单元测试时常见的错误和不足。实验结果表明，TestART在通过率和覆盖率上都明显优于现有方法，如ChatGPT和EvoSuite，证明了其在单元测试生成中的**有效性和实用性。**

单元测试生成-总结与展望

展望：

- **模型优化：**进一步优化和微调LLMs，使其更适合单元测试生成任务，并能够处理更多样化和复杂的代码场景。
- **集成增强：**探索更多的集成方法，将不同的检索和生成技术结合起来，以进一步提高测试用例的质量和覆盖率。
- **领域适应性：**针对不同的编程语言和框架开发专门的单元测试生成模型，使得这些技术在各种开发环境中都能得到广泛应用。
- **自动化反馈循环：**构建更智能的反馈循环系统，利用运行时信息和测试结果不断优化生成策略，提升生成测试用例的准确性和有效性。
- **大规模应用：**在实际的软件开发流程中大规模应用这些技术，通过实践验证其可行性和有效性，并不断改进和优化。

自动断言生成-总结与展望

Approaches	CodeT5	CodeLLama	Mastropao *et al.* [36]	Mastropao *et al.* [35]	CEDAR [37]
Accuracy	66.76%	72.50%	47.00%	68.93%	75.55%

表 基于LLM的断言生成方法的比较

总结

- LLMs在自动断言生成中表现优异，平均准确率为51.82%~58.71%：显示出LLMs在该任务中的潜力，证明其在断言生成中的高效性。
- CodeT5在多个数据集上表现最佳：证明了其在断言生成任务中的强大性能，是当前最优的断言生成模型之一。

RetriGen

- **混合检索器：**结合基于令牌的检索器和基于嵌入的检索器，既考虑了词汇相似性也考虑了语义相似性。
- **预训练语言模型（LLM）生成器**：利用预训练语言模型的代码理解和生成潜力，以解决现有方法无法理解代码语义差异的问题。

AG-RAG

- **联合训练策略：**通过联合训练检索器和生成器，提升断言生成的质量和准确性。
- **多模态信息整合：**结合代码、文档和评论等多模态信息，提升断言生成的全面性和准确性。

自动断言生成-总结与展望

展望：

- **集成更多先进的LLMs**：将最新的、更大规模的预训练语言模型（如CodeLlama和StarCoder）应用于断言生成任务，进一步提升生成断言的准确性和覆盖范围。
- **支持多种编程语言**：扩展研究以支持更多的编程语言，不仅限于Java，从而验证方法在不同编程语言上的通用性。
- **动态评估指标**：引入更多动态执行指标，以更准确地评估生成的断言在实际场景中的有效性，而不仅仅依赖静态匹配指标。
- **自动化错误修复**：结合自动程序修复技术，解决大模型生成的测试用例中存在的编译和运行时错误，进一步提高生成测试用例的质量。

自动断言生成-总结与展望

展望：

- **探索联合训练策略：**进一步优化检索器和生成器的联合训练策略，以提高断言生成的准确性和覆盖率。
- **多候选断言改进：**引入多候选断言生成策略，进一步提升预测准确率和生成质量。
- **检索增强生成策略：**探索和优化检索增强生成策略，使得检索到的断言能够更好地指导生成过程，提升生成断言的准确性和实用性。
- **研究断言生成的实际应用场景，进一步提高模型性能：**在实际开发环境中测试和验证模型，确保其在不同场景中的适用性和有效性。

AiDD

AiDD AI+研发数字峰会
AI+ Development Digital summit

THANKS