Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24.М71-мм

Сбор и использование данных времени выполнения для исправления ошибок с помощью встроенного в среду разработки AI-агента

Муравьев Илья Владимирович

Отчёт по производственной практике
в форме «Производственное задание»

Научный руководитель:
доцент факультета математики и компьютерных наук, к. ф.-м. н., Куликов Е. К.

Санкт-Петербург
2026

Оглавление

Введение

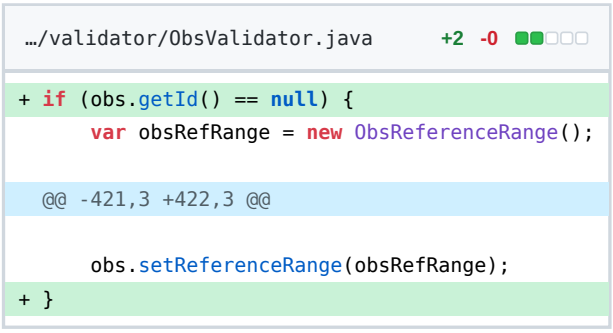
В современной индустрии разработки программного обеспечения (ПО) разработчики, тестировщики и другие участники жизненного цикла ПО всё чаще используют инструменты, основанные на технологиях искусственного интеллекта (англ. artificial intelligence, AI) [28]. Для разработчиков ПО одними из наиболее удобных в использовании инструментов являются инструменты непосредственно встраиваемые в интегрированную среду разработки (англ. integrated development environment, IDE). Сейчас на рынке сосуществует множество встраиваемых в IDE инструментов, основанных на AI, в частности предоставляемый компанией ООО «ИИТЕХ» плагин VeAI [34], который встраивается в IDE, выпускаемые компанией JetBrains. Плагин VeAI включает в себя тесно-интегрированного с IDE AI-агента, который позволяет пользователям в интерактивном режиме делегировать задачи разработки, тестирования и отладки искусственному интеллекту.

Одна из возникающих в процессе разработки ПО задач — это задача исправления ошибок (дефектов) реализации, найденных автоматизированными тестами (автотестами). До данной работы AI-агент VeAI уже во многих случаях корректно решал эту задачу, но в ряде случаев AI-агент VeAI предлагал неработающие или некорректные исправления. В частности, распространённой проблемой являлось «симптоматическое лечение» тестов, а именно: внесение изменений в код рядом с тем местом, где проявлялась ошибка реализации (где было выброшено исключение), когда настоящей причиной падения теста (англ. root cause [12]) было произошедшее существенно ранее нарушение внутренних инвариантов отлаживаемого приложения. Применение такого «симптоматического лечения» даже в небольшой доли случаев крайне опасно для долгосрочной стабильности и поддерживаемости ПО, так как «симптоматическое лечение» не устраняет нарушение инвариантов, а лишь скрывает отдельное проявление ошибки реализации.

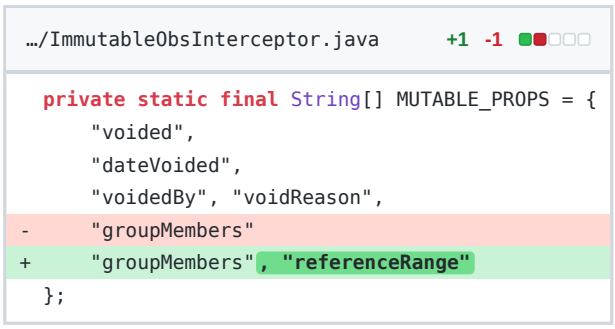
Рассмотрим пример: в проекте `openmrs-core` на момент коммита `150ee0a` тесты завершались (падали) с ошибкой «`UnchangeableObject`

`Editing the fields [referenceRange] on Obs is not allowed`», разработчики проекта исправили эту ошибку, предотвратив изменения уже сохранённых объектов типа `Obs` (см. рисунок 1a), в то время, как AI-агенты, такие как исходная версия AI-агента VeAI, «исправляют» эту ошибку путём добавления поля `referenceRange` в список полей, которые можно редактировать (см. рисунок 1b). Что же мешает AI-агентам реализовать исправление, аналогичное исправлению от разработчиков `openmrs-core`? Одна из проблем заключается в том, что неразрешенная модификация `Obs` не упоминается в трассе стека (англ. stacktrace) исключения, отделена от непрошедшего (падающего) теста 97 кадрами стека вызовов и выполняется из класса `ObsValidator`, который содержит корректную логику инициализации новых экземпляров класса `Obs`. Естественным способом решения этой проблемы является включение в контекст AI-агента некоторых данных времени выполнения, достаточных для определения фрагмента кода, где происходит нарушение инвариантов системы.

Данная работа посвящена сбору таких данных времени выполнения и использованию их для повышения точности исправления AI-агентом VeAI ошибок, найденных автотестами. Чтобы повысить сфокусированность, работа ограничивается расширением возможностей AI-агента исключительно в среде разработки IntelliJ IDEA [15], ориентированной на JVM-языки. Была выбрана именно эта среда разработки, так как на момент начала работы для этой среды разработки был реализован наиболее полный набор функциональности VeAI [35].



(a) Исправление, выполненное разработчиками `openmrs-core`.[1]



(b) Исправление, выполненное изначальной версией AI-агента VeAI при использовании модели GPT-5.2.

Рис. 1: Сравнение исправлений ошибки «`UnchangeableObject Editing the fields [referenceRange] on Obs is not allowed`» из проекта `openmrs-core`.

[1] Источник: https://github.com/openmrs/openmrs-core/compare/150ee0aed92c..d85c13706ec4.

1 Постановка задачи

Целью работы является сбор и использование данных времени выполнения для повышения точности исправления AI-агентом VeAI ошибок, найденных автотестами в проектах на JVM-языках. Для достижения этой цели были поставлены следующие задачи.

1. Выбрать подход к сбору данных времени выполнения.
2. Реализовать сборщик и постобработчики данных времени выполнения.
3. Встроить сбор и использование данных времени выполнения в VeAI.
4. Измерить, как данные времени выполнения влияют на точность исправления ошибок в реальных проектах.

2 Обзор

В данной главе проводится обзор основных возможностей AI-агента VeAI и способов представления данных времени выполнения, а также выбирается технология для сбора данных времени выполнения.

2.1 AI-агент VeAI

VeAI — это плагин, разрабатываемый для IDE, выпускаемых компанией JetBrains и их российских аналогов: IntelliJ IDEA, AndroidStudio, OpenIDE, GigaIDE, PyCharm, GoLand, Rider и WebStorm. Плагин VeAI включает в себя AI-агента и неагентскую функциональность такую как, генерация тестов по конкретному исполнению.

Рассмотрим, как работает AI-агент VeAI. Агент получает от пользователя задачу разработки, тестирования или отладки и затем с помощью большой языковой модели (LLM) генерирует вызовы инструментов взаимодействия с проектом, необходимые для сбора контекста и решения задачи. Полученные результаты работы инструментов дописываются в конец запроса к LLM, называемого «промптом», после чего LLM снова генерирует вызовы инструментов взаимодействия с проектом. Данный цикл повторяется, пока LLM не сгенерирует ответ содержащий не вызовы инструментов, а отчёт о проделанной работе или уточняющий вопрос или не сработает механизм прерывания агентского цикла. Упрощённая схема работы показана на рисунке 2.

Такая схема взаимодействия с LLM является весьма стандартной [19], особенностью VeAI является тесная интеграция агента с IDE, выпускаемыми компанией JetBrains, в частности с IntelliJ IDEA. Эта интеграция заключается в обогащении промпта контекстом и состоянием проекта и, самое главное, предоставлением агенту ряда инструментов, позволяющих использовать возможности IDE, например:

- инструмент просмотра структуры файла позволяет агенту получить данные о содержимом файла (классах, методах, XML-секциях и т. д.), не расходуя контекст на полное прочтение файла;

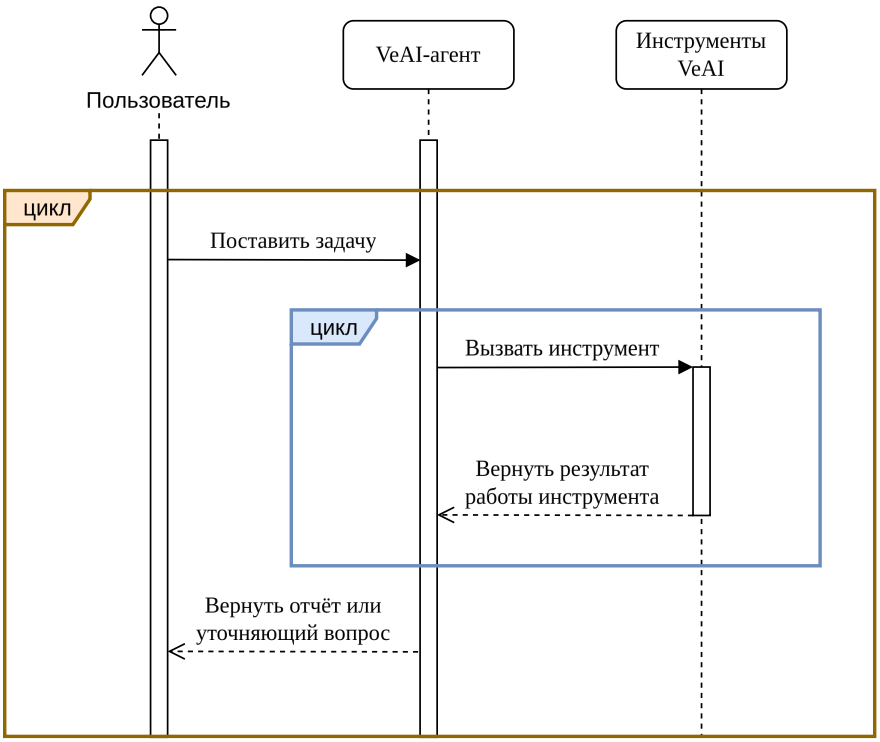


Рис. 2: Упрощённая схема работы VeAI агента.

- инструмент поиска и чтения файлов позволяет агенту работать не только с файлами самого проекта, но и с файлами библиотек, в том числе в тех случаях, когда исходный код библиотек не доступен и требуется декомпиляция;
- инструмент редактирования файла сразу же сообщает агенту о допущенных ошибках компиляции, обнаруженных IDE, без необходимости пересобирать весь проект или весь модуль;
- инструменты запуска команд и задач системы сборки, позволяют агенту выполнять команды и задачи Gradle и Maven с учётом настроек IDE, таких как настройки версии SDK;
- инструмент запуска тестов позволяет агенту запускать тесты с теми же параметрами, с которыми их запускает IDE, собирать структурированные сжатые результаты запуска тестов: данные журналирования (логи), исключения и данные о покрытии кода тестами.

Перечисленные инструменты уже позволяют AI-агенту VeAI часто

успешно решать задачу исправления ошибок реализации, когда достаточно исправить тот метод или класс, из которого было выброшено исключение. Тем не менее на момент начала данной работы оставалось затрудненным исправление ошибок в тех случаях, когда содержащий ошибку метод или класс не упоминается в трассе стека (англ. stacktrace) исключения, особенно когда содержащий ошибку метод отделён от непрошедшего (падающего) теста десятками кадров стека вызовов, в том числе в приложениях, использующих рефлексию. Данное ограничение связано с тем, что AI-агент VeAI не имеет доступа к детальным данным времени выполнения и при генерации исправлений, основывается лишь на логах падающих тестов и собранном с помощью вызова инструментов анализа проекта контексте.

2.2 Способы представления данных времени выполнения

Исправление ошибок реализации с помощью AI-агентов является популярной темой исследований [17, 33, 29, 11, 13, 24, 2]. В данном разделе проводится обзор того, какого рода данные времени выполнения пробовали передавать в AI-агентов в предыдущих работах по теме, как эти данные представлялись, и насколько такие представления могут быть полезны при исправлении ошибок, спрятанных за десятками кадров стека вызовов (аналогичных примеру рассмотренному во введении настоящей работы, см. рисунок 1).

Перед переходом к рассмотрению конкретных работ по теме стоит отметить важную особенность данной области исследований: большие языковые модели (LLM) стремительно развиваются в последние годы. Существенная доля задач в области разработки ПО, которые не решались моделями, доступными в 2024 году, успешно решаются современными языковыми моделями [26]. Научные статьи публикуются с существенной задержкой во времени, поэтому при рассмотрении даже недавних исследований надо учитывать не только их результаты, полученные на далеко не самых современных моделях, но и то, насколько

используемые в этих исследованиях подходы обобщаемы для решения задач, всё ещё вызывающих трудности у современных LLM, в частности задач, требующих анализа на много методов вглубину.

2.2.1 Данные журналирования (логи) непроходящих тестов

Простейшими данными времени выполнения, которые нельзя не упомянуть, являются данные журналирования (логи) падающих тестов. Ранние работы, посвящённые исправлению ошибок (дефектов) реализации с помощью больших языковых моделей используют логи в качестве единственных данных времени выполнения [2]. Однако, как уже было сказано в подразделе 2.1, логи падающих тестов уже доступны AI-агенту VeAI, следовательно для повышения точности исправления ошибок AI-агентом VeAI необходимо рассмотреть другие подходы.

2.2.2 Трассы выполнения

Ещё одним видом данных времени выполнения, которые могут помочь AI-агенту более точно исправлять ошибки реализации являются трассы выполнения, показывающие путь выполнения и отладочную информацию в различных точках этого пути. Данные такого вида по разному представлялись в нескольких предшествующих работах [17, 33, 29].

В работе [17] в контекст языковых моделей включается код тестируемой функции и при этом в конец каждой выполненной строки кода в этой функции добавляется комментарий, показывающий значение всех переменных в момент первого и последнего выполнений этой строки кода. Данный подход позволяет языковым моделям более точно отлаживать отдельные функции, однако в масштабах отдельных функций языковые модели уже показывают выдающиеся результаты и без использования трасс выполнения [16, 8], а при использовании данного подхода для межпроцедурной отладки теряется информация о порядке вызова функций, кроме этого включение в промпт кода каждого метода, выполненного во время выполнения теста, при отладке реаль-

ных приложений рискует переполнить эффективное контекстное окно языковой модели.[2]

В исследовании [29] предлагается подход, похожий на подход из [17], но на этот раз также исследуется подход, при котором значения переменных записываются в контекст языковой модели не в виде комментариев внутри исходного кода, а в виде отдельной секции промпта. В [29], как и в [17], трассы строятся на уровне отдельных строк кода, что также делает этот подход малоприменимым для глубокой межпроцедурной отладки: результаты исследования показывают, что успешность исправления ошибок реализации отрицательно коррелирует с длиной трассы выполнения и что наиболее стабильные результаты получаются при предварительном сжатии трассы выполнения с помощью дополнительного запроса к языковой модели [29].

Наконец, в статье [33] рассматривается включение в запрос к LLM трасс различных уровней гранулярности: трасс на уровне отдельных строк кода, трасс на уровне базовых блоков (англ. basic blocks) или трасс на уровне функций. Для каждого вида трассы наблюдается увеличение количество успешных исправлений ошибок в тестируемой программе, наилучшие результаты получаются для трасс выполнения на уровне базовых блоков. В контексте рассматриваемой в данной работе задачи глубокой межпроцедурной отладки наиболее перспективными выглядят низкие уровни гранулярности трасс выполнения, такие как гранулярность на уровне функций и, возможно, гранулярность на уровне базовых блоков. К сожалению, переиспользовать разработанный в [33] инструментарий сбора трасс в рамках данной работы не представляется возможным, так как этот инструментарий предназначен для программ на языке Python, а настоящая работа фокусируется на JVM-языках.

[2]Эффективное контекстное окно LLM — это часть контекстного окна, которую LLM может эффективно использовать для решения нетривиальных задач. Бенчмарки показывают, что у современных LLM эффективное контекстное окно существенно меньше, чем всё контекстное окно [21].

2.2.3 Интерактивно собираемые данные

Ряд исследований [11, 13, 24] вместо полного сбора трасс выполнения фокусируется на сборе данных времени выполнения в некоторых контрольных точках. В одних работах [11, 13] эти контрольные точки определяются самой большой языковой моделью в виде условных точек останова (англ. conditional breakpoint), в то время, как в [24] пользователю инструмента самому предлагается поставить точку остановки в интересном месте отлаживаемой программы и задать языковой модели вопрос про текущее состояние времени выполнения и то, как оно было достигнуто. При этом в работе [11] большая языковая модель сама определяет выражение, значение которого будет вычислено в точке остановки, а в исследовании [13] языковая модель не только может вычислять значения выражений в точке остановки, но и произвольным образом изменять состояние программы.

В контексте задачи настоящей работы, а именно в контексте расширения возможностей AI-агента VeAI необходимо учесть, что AI-агент VeAI уже в состоянии собирать данные времени выполнения интерактивно путём добавления временных инструкций логирования с помощью инструмента редактирования файлов и повторных запуска тестов после добавления этих отладочных инструкций.

2.2.4 Выбор способа представления данных времени выполнения

Итак, два из трёх рассмотренных видов данных времени выполнения уже в некоторой степени доступны AI-агенту VeAI: данные журналирования (логи) полностью доступны AI-агенту VeAI, а интерактивно собираемые данные агент может получить, воспользовавшись инструментами редактирования файлов и запуска тестов. У оставшегося подхода, а именно у сбора трасс выполнения можно выделить три уровня гранулярности: уровень строк, уровень базовых блоков (basic blocks) и уровень функций. При этом для трасс уровня отдельных строк характерен наиболее высокий риск переполнить эффективный размер контек-

Таблица 1: Результаты сравнения различных данных времени выполнения в контексте задачи повышения точности исправления ошибок реализации AI-агентом VeAI.

Вид данных времени выполнения		**Низкий риск переполнения эффективного контекстного окна**	**Может повысить качество работы VeAI**[3]
Данные журналирования (логи)		$+$[4]	$-$
Интерактивно собираемые данные		$\pm$	$\pm$
Трассы выполнения	Детальность уровня строк кода	$-$	$+$
	Детальность уровня базовых блоков	$\mp$	
	Детальность уровня функций	$\pm$	

ста. Компактно результаты сравнения рассмотренных видов данных времени выполнения представлено в таблице 1.

Учитывая, что предыдущие исследования показывают, что в различных случаях полезны разные данные времени выполнения [13] и единственным видом данных времени выполнения, полностью недоступным AI-агенту VeAI, являются трассы выполнения, в рамках настоящей работы было решено поддержать в VeAI использование именно трасс выполнения. В частности, из-за соображений рисков переполнения контекста было решено использовать трассы уровня функций (см. предпоследний столбец таблицы 1).

Стоит отметить, что то, что в предыдущих работах удалось найти перспективную идею представления данных времени выполнения в виде трассы выполнения на уровне функций вовсе не означает, что задача о представлении этой трассы в контексте языковой модели является решённой. Существенной сложностью остаётся сжатие реальных трасс выполнения, часто состоящих из десятков или сотен тысяч вызовов и уходящих на десятки методов в глубину, до разумных размеров. В рассмотренных в обзоре работах [17, 33, 29] данная проблема сжатия меж-

[3]Не покрыто исходными возможностями VeAI.

[4]При использовании логов риск переполнения контекста низкий, так как в VeAI реализован механизм сжатия логов выполнения тестов.

процедурных трасс выполнения не решалась, так как эти работы ориентировались на бенчмарки, состоящие из небольших программ, каждая из которых содержит лишь несколько функций [10, 1, 23, 31, 25, 22].

2.3 JVM-технологии сбора трасс выполнения

Как уже было сказано в подразделе 2.2, в настоящей работе не получится переиспользовать сборщик трасс выполнения уровня функций, используемый в [33], так как настоящая работа в отличии от [33] ориентирована не на язык Python, а на JVM-языки, то есть языки, которые исторически имеют наилучшую поддержку в VeAI. К счастью, в мире JVM существует множество инструментов и библиотек, позволяющих реализовать сбор трасс выполнения, таких как OpenTelemetry [7], Spring AOP [3], Java Debug Interface (JDI) [20], AspectJ [9], JacoDB [14], Soot [27], ByteBuddy [32], Javassist [6] и ASM [18]. Сравнение перечисленных технологий приведено в таблице 2.

При выборе технологии надо было учесть, что реальные приложения на JVM-языках обычно используют библиотеки и при наивном включении в трассу всех вызовов функций большая часть трассы будет описывать внутренности работы библиотек. Кроме того, сбор всех вызовов методов внутри всех библиотек рискует существенно замедлить работу приложения. В связи с этим необходимо было выбрать инструмент предоставляющий высокоуровневый программный интерфейс (API) для инструментацию только тех вызовов методов, которые происходят из кода самого приложения, а не из библиотек.

Среди рассмотренных технологий все, кроме одной, либо не поддерживают такую инструментацию либо поддерживают её лишь на низком сложном в использовании уровне. В таблице 2 единственным инструментом, предоставляющим высокоуровневый API для инструментации на стороне вызывающего метода (caller side instrumentation) является AspectJ. Именно этот инструмент и было решено использовать в данной работе.

Таблица 2: Сравнение JVM-технологий, которые можно использовать для сбора трасс выполнения.

Технология	**Применимость к вызовам любых методов**	**Высокоуровневый API для инструментации на стороне вызываемого метода**	**Высокоуровневый API для инструментации на стороне вызывающего метода**
OpenTelemetry[5]	−[6]	+	−
Spring AOP	−[7]	+	−
JDI	+	−	−
AspectJ	+	+	+
JacoDB	+	−	−
Soot	+	−	−
ByteBuddy	+	+	−
Javassist	+	+	±
ASM	+	−	−

[5]Если использовать только OTel Java-агента без дополнительной (самописной) инструментации.

[6]Только вызовы некоторых библиотечных методов.

[7]Только вызовы методов объектов, управляемых фреймворком Spring.

3 Метод

Для предоставления VeAI-агенту доступа к данным времени выполнения был доработан инструмент запуска тестов. Теперь инструмента запуска тестов выполняются следующие дополнительные шаги.

1. Во время выполнения тестов основанный на AspectJ Java-агента собирает трассу выполнения (дерево вызовов) (см раздел. 3.1).
2. Дерево вызовов постобрабатывается:
   (a) асинхронные вызовы присоединяются к основным ветвям дерева (см. подраздел 3.2.1);
   (b) удаляются части дерева, не относящиеся к упавшему тесту (см. подраздел 3.2.2);
   (c) выделяются глобально уникальные вызовы (см. подраздел 3.2.3);
   (d) удаляются локальные повторы (см. подраздел 3.2.4);
   (e) сжимаются неинформативные идентификаторы объектов (см. подраздел 3.2.5);
   (f) дерево сжимается до фиксированного размера (см. подраздел 3.2.6);
   (g) сжатое дерево вызовов суммаризируется с помощью большой языковой модели (см. подраздел 3.2.7).
3. Суммаризация дерева вызовов и путь к дереву вызовов возвращаются VeAI-агенту.

3.1 Сбор трассы выполнения (дерева вызовов)

Для сбора дерева вызовов был реализован механизм, позволяющий автоматически запустить произвольные тесты на JVM-языках со специальным разработанным в рамках данной работы Java-агентом, основанном на библиотеке AspectJ. Данный механизм основывается на

переиспользовании возможностей среды разработки IntelliJ IDEA, что позволяет ему работать с любыми тестами на JVM-языках вне зависимости от используемых фреймворка тестирования и системы сборки.

При проектировании сборщика трасс выполнения были сформулированы два требования.

1. Сбор трассы не должен требовать инструментации всех внутренних классов всех используемых библиотек. Данное требование является стандартным при разработке подобных Java-агентов (см., например, [30]). Требование мотивировано тем, что Java-приложения нередко используют крайне объёмные библиотеки, такие как Spring [4], полная инструментация которых не практична.

2. В трассе должны быть представлены вызовы библиотечных методов, выполняемые **напрямую** из кода приложения (см. пример на рисунке 3).

На рисунке 3 иллюстрируется пример, в котором метод `appMethod()` вызывает библиотечный метод `heavyLibraryMethod()`, передавая аргументы библиотечного метода в ошибочном порядке.

Также на рисунке 3 показывается, как выбор способа инструментации влияет на структуру трассы выполнения:

1. При использовании подхода, показанного в правой верхней части рисунка 3, в трассу включаются входы и выходы всех методов, включая библиотечные. Такая трасса содержит информацию, достаточную для выявления ошибки, однако её построение требует инструментации большого объёма библиотечного кода, что противоречит требованию 1 и делает данный подход непрактичным для реальных JVM-приложений.

2. При использовании подхода, показанного в правой средней части рисунка 3, инструментируются входы и выходы только методов самого приложения, но не библиотек. Трасса становится компактной и не требует трансформации кода библиотек, однако вызов

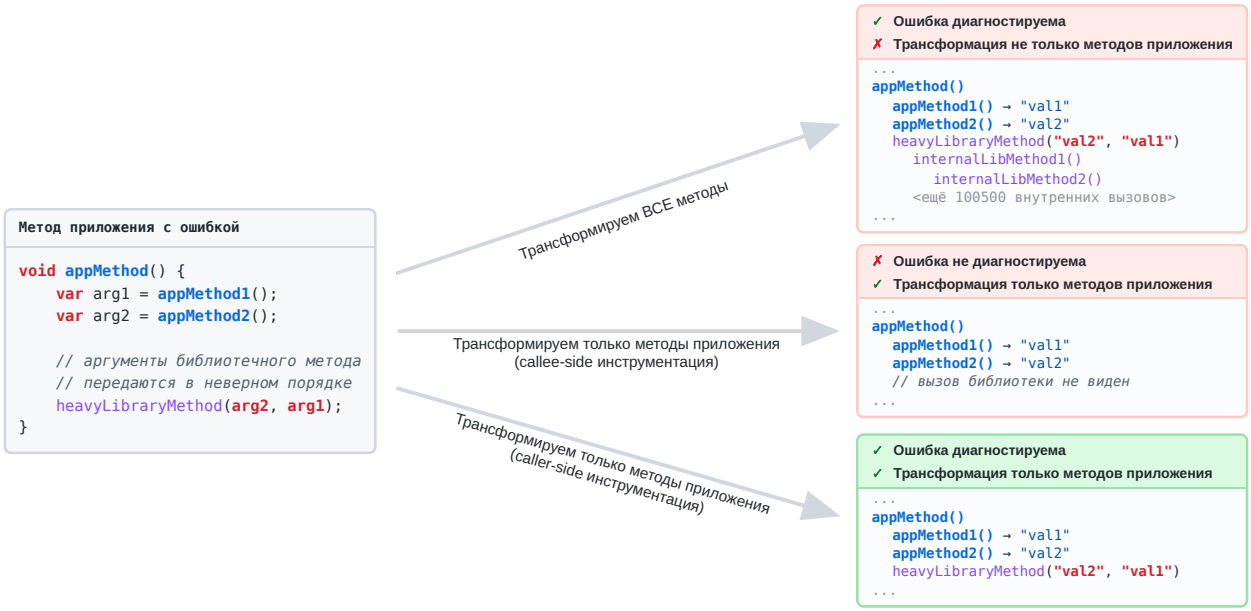


Рис. 3: Пример ошибочного вызова библиотечного метода и влияние способа инструментации на структуру трассы выполнения.

библиотечного метода в ней не отражается, из-за чего ошибка передачи аргументов не может быть локализована, несмотря на то что её причина находится в коде приложения, что нарушает требование 2.

3. При использовании подхода, показанного в правой нижней части рисунка 3, инструментируются не только входы и выходы методов самого приложения, но также и все места вызова библиотечных методов из методов приложения. Благодаря этому, в трассе явно фиксируется вызов библиотечного метода и порядок переданных аргументов и при этом трансформация внутренностей библиотек не требуется, что позволяет одновременно удовлетворить требованиям 1 и 2; именно этот подход используется в данной работе.

3.2 Постобработка дерева вызовов

«Сырое» дерево вызовов, возвращаемое Java-агентом, может существенно превосходить размер эффективного контекстного окна современных языковых моделей. Кроме того, «сырое» дерево вызовов не отражает связи между асинхронно вызываемыми методами и методами,

приведшими к асинхроным вызовам. В связи с этим после получения «сырого» дерева вызовов производится ряд шагов постобработки, описанных в данном разделе.

3.2.1 Присоединение асинхронных вызовов

Многие современные Java-приложения используют асинхронное выполнение. В «сыром» дереве вызовов асинхронные вызовы представлены отдельными верхнеуровневыми ветвями. Первый шаг постобработки дерева вызовов проиллюстрирован на рисунке 4 и заключается в присоединение асинхронных вызовов к основной ветке выполнения. Такое присоединение основывается на записанной Java-агентом информации о времени начала и конца вызовов и позволяет:

- получить дерево вызовов, более точно описывающее процесс выполнения;
- избежать обрезания содержательных частей дерева на последующих шагах постобработки (например, в случае примера, показанного на рисунке 4, присоединение асинхронного поддерева «защищает» участок, отмеченный текстом `<поддерево с первопричиной ошибки>`, от удаления на шаге, описанном в подразделе 3.2.2).

3.2.2 Фокусировка на релевантных участках дерева

Вторым шагом преобразования дерева вызовов является удаление вызовов происходящих за пределами интересуемого метода. При этом интересуемый метод определяется самим VeAI агентом. Обычно в роли интересуемого метода выступает тестовый метод, но также интересуемым методом может быть и метод, выполняющий инициализацию системы до выполнения тестового метода.

(а) Дерево вызовов до присоединения асинхронных вызовов

```
...
MyControllerTest.exampleTest() → thrown AssertionError("Expected 200...")
  WebTestClient.get() → RequestHeadersUriSpec@19af24e1(...)
    RequestHeadersUriSpec.uri("/test") → RequestHeadersUriSpec@19af24e1(...)
    RequestHeadersUriSpec.exchange() → DefaultResponseSpec@a7511c8b(status=500, ...)
      DefaultResponseSpec@a7511c8b.expectStatus() → StatusAssertions@b2e631d7
        StatusAssertions@b2e631d7.isOk() → thrown AssertionError(...)
MyController.get() → ResponseEntity@7b6fa720(status=500, ...)
  <поддерево с первопричиной ошибки>
...
```

(б) Дерево вызовов после присоединения асинхронных вызовов

```
...
MyControllerTest.exampleTest() → thrown AssertionError("Expected 200...")
  WebTestClient.get() → RequestHeadersUriSpec@19af24e1(...)
    RequestHeadersUriSpec.uri("/test") → RequestHeadersUriSpec@19af24e1(...)
    RequestHeadersUriSpec.exchange() → DefaultResponseSpec@a7511c8b(status=500, ...)
      (async in thread worker-3) MyController.get() → ResponseEntity@7b6fa720(status=500, ...)
        <поддерево с первопричиной ошибки>
    DefaultResponseSpec@a7511c8b.expectStatus() → StatusAssertions@b2e631d7
    StatusAssertions@b2e631d7.isOk() → thrown AssertionError(...)
...
```

Рис. 4: Пример присоединения асинхронного вызова метода `MyController.get()`, выполняемого из метода `RequestHeadersUriSpec.exchange()`.

3.2.3 Сохранение репрезентативных вызовов

Следующим шагом постобработки дерева вызовов является выделение представителя для каждой достаточно популярной пары вызываемого метода и результата вызов. Выделенные представители защищаются от удаления при выполнении последующих шагов сжатия дерева, что позволяет избежать полного удаления целых групп вызовов.

3.2.4 Удаление локальных повторов

Очередной шаг постобработки дерева вызовов заключается в сжатии групп смежных вызовов одних и тех же методов из одних и тех же мест вызова с одним и тем же типом результата. Такие группы часто образуются при использовании в трассируемом коде циклов, а дедупликация повторяющихся вызовов в таких группах позволяет последующим шагам сжатия сохранить больше информативных частей дерева вызовов.

3.2.5 Сжатие неинформативных идентификаторов объектов

В строковых представлениях объектов JVM часто встречаются неинформативные идентификаторы вида `Object@7b6fa720`. Идущая после символа «`@`» часть этих идентификаторов малоинформативна и часто занимает много токенов в контексте языковой модели, поэтому при постобработке дерева вызовов подобные длинные идентификаторы заменяются на более короткие идентификаторы. Например, все упоминания `Object@7b6fa720` могут замениться на `Object@42`.

3.2.6 Ограничение размера дерева с распределением бюджета

Даже после сжатие повторов дерево может оставаться слишком большим. Поэтому итоговое дерево дополнительно ограничивается по размеру с помощью алгоритма, основанного на рекурсивном распределении бюджета размера дерева между ветвями дерева.

3.2.7 Суммаризация дерева вызовов с помощью большой языковой модели

Последним шагом обработки дерева вызовов является его суммаризация с помощью большой языковой модели. Именно результат этой суммаризации и возвращается VeAI-агенту в качестве дополнительных данных, включаемых в ответ инструмента запуска тестов.

Стоит упомянуть, что помимо суммаризации VeAI-агенту также возвращается путь до файла, содержащего сжатое дерево вызовов, что позволяет агенту при необходимости выполнить поиск по дереву и прочитать релевантные части дерева.

4 Архитектура

В данной главе описываются предназначение и изменения каждого из затронутых в работе моделей, а также описывается как эти модули взаимодействуют друг с другом. Архитектура реализованного решения показана на рисунке 5 в виде диаграммы компонентов.

4.1 Роли модулей

Модуль `:veai-runtime` отвечает за мониторинг Java-процессов, запущенных VeAI. В данной работе в данном модуле были созданы два подмодуля `:call-tree-dumper` и `:call-tree-model`, отвечающие соответственно за сбор и представление «сырого» дерева выполнения. Эти модули реализованы на языке программирования Java и основываются на библиотеке AspectJ.

Модуль `:veai-test-runner` отвечает за запуск тестов в отдельном Java-процессе, в данной работе этот модуль был интегрирован с реализованным в модуле `:veai-runtime` сборщиком деревьев вызова.

Модуль `:veai-renderable-call-tree` был создан в ходе данной работы и отвечает за представление дерева вызовов в удобном для постобработки и рендеринга виде. Данное представление отличается от

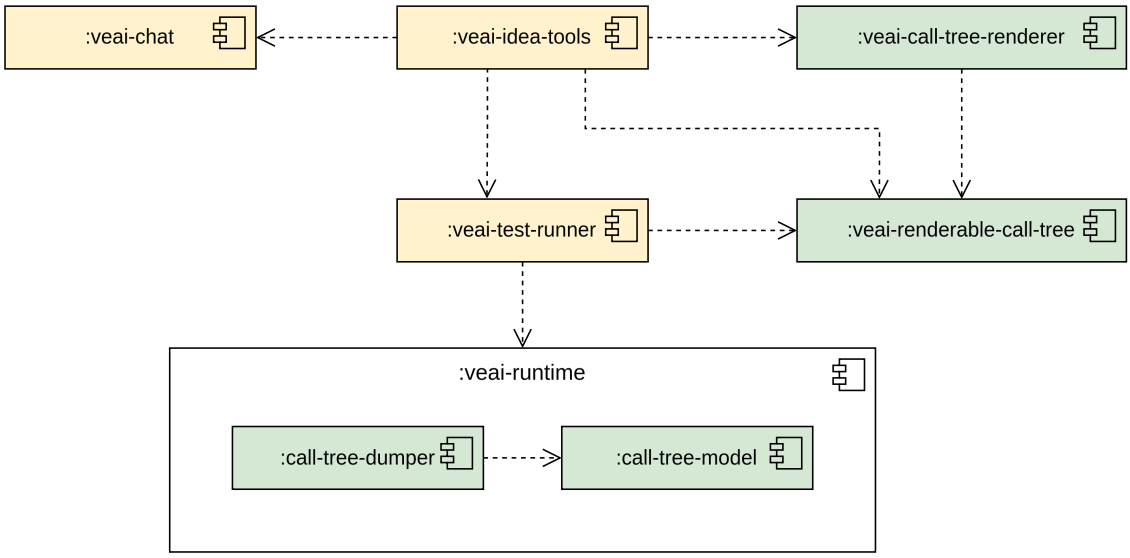


Рис. 5: Диаграмма компонентов, созданных и доработанных в ходе работы.

«сырого» представления в модуле `:veai-runtime:call-tree-model` использованием возможностей языка программирования Kotlin и наличием дополнительных свойств, используемых для передачи информации от более ранних постобработчиков дерева вызовов к более поздним, а также для передачи информации от постобработчиков к рендереру.

Модуль `:veai-call-tree-renderer` был создан в ходе данной работы и отвечает за постобработку и рендеринг дерева вызовов. Данный модуль содержит несколько постобработчиков (фильтров) реализующих единый интерфейс `CallTreeDumpFilter`, а также ренедрер дерева вызовов.

Модуль `:veai-idea-tools` отвечает за реализацию инструментов агента VeAI для взаимодействия со средой разработки IntelliJ IDEA. В данной работе реализованный в этом модуле инструмент запуска тестов был обновлён для работы со сборкой дерева вызовов.

Модуль `:veai-chat` отвечает за высокоуровневую работу чата пользователя с агентом VeAI. В данной работе был произведён существенный рефакторинг этого модуля, позволивший в том числе инструменту запуска тестов показывать пользователю генерацию суммаризации трассы выполнения в потоковом режим.

4.2 Интеграция модулей

При вызове VeAI агентом инструмента запуска тестов, реализованного в модуле `:veai-idea-tools`, происходит следующее.

1. Модуль `:veai-test-runner` поднимает отдельный JVM-процесс и подключает Java-агента, реализованного в модуле `:veai-runtime:call-tree-dumper`.

2. Java-агент, реализованный в модуле `:veai-runtime:call-tree-dumper` формирует «сырое» дерево вызовов в формате `:veai-runtime:call-tree-model`.

3. Модуль `:veai-test-runner` преобразует «сырое» дерево вызовов в абстракции модуля `:veai-renderable-call-tree`.

4. Модуль `:veai-call-tree-renderer` применяет конвейер постобработчиков и рендерит компактное представление дерева вызовов.

5. Реализация инструмента запуска тестов в модуле `:veai-idea-tools` отслеживает процесс суммаризации дерева вызовов и в потоковом режиме передает результаты модулю `:veai-chat`, который теперь позволяет инкрементально обновлять результат работы инструментов VeAI агента.

5 Экспериментальное исследование

В данной главе описывается методология и результаты экспериментального исследования, направленного на оценку влияния использования трасс выполнения (деревьев вызова) на качество генерируемых VeAI исправлений ошибок.

5.1 Исследовательские вопросы

Перед проведением замеров были поставлены следующие исследовательские вопросы.

RQ1. Как использование трасс выполнения влияет на долю генерируемых VeAI исправлений, которые проходят тесты и вносят изменения в «правильный» файл?

RQ2. Как использование трасс выполнения влияет на прохождение генерируемыми VeAI исправлениями тестов?

В RQ1 под «правильным» файлом понимается файл, который был изменён разработчиками проекта при исправлении рассматриваемой ошибки (между коммитами «до исправления» и «после исправления» (см. таблицу 3)). Проверка файла, в который вносятся изменения была добавлена, чтобы отфильтровать такие некорректные исправления, как, например, исправление, показанное на рисунке 1.

5.2 Условия эксперимента

Все замеры проводились по 5 раз для двух версий VeAI, отличающихся только использованием трассы выполнения (деревьев вызовов).

Для проведения замеров использовалась модель GPT-5.2 и рабочая станция с процессором 13th Gen Intel(R) Core(TM) i7-13700H и 32 ГБ оперативной памяти. Время на исправление одной ошибки было ограничено 10 минутами.

Замеры проводились после полной индексации средой разработки IntelliJ IDEA и плагином VeAI проекта и выполнения команды `gradle assemble` для Gradle-проектов и команды `mvn -T1C -DskipTests -Dspotbugs.skip=true -Dpmd.skip=true -Dcheckstyle.skip=true install` для Maven-проектов.

В процессе экспериментальных запусков VeAI-агенту передавалась задача в виде запроса «Run `{FAILED_TEST}` and fix the bug in the implementation», где вместо `{FAILED_TEST}` подставлялось имя непроходящего теста (см. таблицу 3).

5.3 Набор данных

Для проведения замеров использовался набор из 20 реальных дефектов в 17 проектах из набора данных BugSwarm [5]. Данные о каждом дефекте указаны в таблице 3. Каждый дефект характеризуется именем проекта, именем непроходящего теста, коммитами до и после исправления и версией Java.

5.4 Результаты

Результаты замеров показаны на рисунке 6. Экспериментальные результаты показывают, что в результате использования трасс выполнения:

- доля исправлений, прошедших тесты и сделанных в «правильном файле» (RQ1), в среднем увеличилась на 19%;
- доля исправлений, прошедших тесты (RQ2), в среднем увеличилась на 14%.

При этом во всех случаях радиус 95% доверительного не превышает 4.5%. Ограниченный прирост среди исправлений, проходящих тесты и сделанных в произвольном файле (RQ1), объясняется тем, что и до выполнения данной работы VeAI крайне часто был в состоянии сгенерировать не обязательно корректное исправление, проходящее тесты.

Таблица 3: Набор задач для оценки влияния трасс выполнения на качество исправления ошибок.

Проект	**Непроходящий тест**	**Исправление**	**Версия Java**
apache-maven	`o.a.m.c.MavenCliTest.testMVN…ViaCommandLine()`	`00ead1f..9cd64f3`	21
openpnp	`SampleJobTest.testSampleJob()`	`0143f8e..28b8070`	8
Hakky54-log-captor	`n.a.l.LogCaptorShould.capture…Exception()`	`04565e2..39b66ae`	17
weibocom-rill-flow	`c.w.r.f.o.t.InvokeMsgTest.«big flow…»()`	`061b2b8..e3f7645`	17
hedera-mirror-node	`c.h.m.m.p.TransactionPub…Test.getNodesAddressBookError()`	`06c1e61..8e38013`	17
openpnp	`SampleJobTest.testSampleJob()`	`0b2a2c1..c2cca98`	8
hyperledger-besu	`o.h.b.e.a.j.i.m.EthCreateAccessList…Test.shouldReturn…InsufficientGas()`	`0f144e8..665f9b6`	21
polaris-java	`c.t.p.c.c.i.ConfigFileLongPoll…Test.testNotReceivedPushEvent()`	`106afe7..3afea74`	8
SonarSource-sonar-java	`o.s.j.UCFGJavaVisitorTest.build_assignment_for_string()`	`136f6ba..a155e21`	8
openmrs-core	`o.o.a.h.PatientDataUnvoidHandlerTest.handle_shouldUnvoid…Patient()`	`150ee0a..d85c137`	21
iBotPeaches-Apktool	`b.a.a.AndroidOreoNotSparseTest`	`1b72ddd..eb9bade`	8
polaris-java	`c.t.p.p.c.g.ConnectionManagerTest.testSwitch…BusinessError()`	`1c288ef..2d1f132`	8
pgjdbc	`o.p.t.j.OuterJoinSyntaxTest.testOuterJoin…AndWithOj()`	`1ca3dbe..fc41717`	8
OpenTrip Planner	`o.o.e.g.LuceneIndexTest.stopLocations()`	`1f4a8d8..2ab5e85`	21
swagger-core	`i.s.SecurityDefinitionTest.createModel…Requirements()`	`2016ab0..e385803`	8
matsim-libs	`o.m.c.d.e.o.s.r.RunPrebooking…IT.testWithoutReattempts()`	`2057a6f..a314b7b`	21
ontop	`i.u.i.o.d.UnionLensTest.testUniqueNotInferred…Unique()`	`21fd326..9e31b32`	11
1c-bsl-language-server	`c.g._1c.s.b.l.s.c.LanguageServer…Test.createFromFile()`	`22dccc6..8e40ccd`	17
apache-maven	`o.a.m.l.LifecycleExecutorTest.testSetupMojoExecution()`	`22ee3a4..bdb7e8e`	21
rxjava-jdbc	`c.g.d.r.j.DatabaseSyncTest`	`244c6c0..179b962`	8

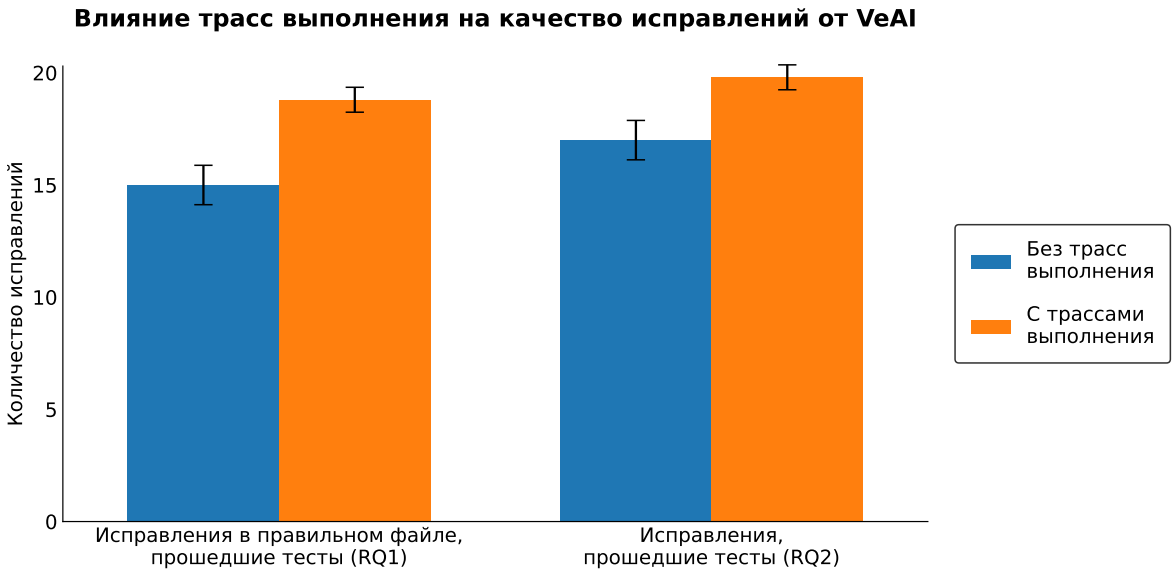


Рис. 6: Качество исправлений генерируемых VeAI без трасс выполнения и с трассами выполнения. Планки погрешности показывают 95% доверительные интервалы.

Заключение

В ходе работы были достигнуты следующие результаты.

1. Выявлено, что перспективным подходом к сбору данных времени выполнения для исправления VeAI ошибок в реальных проектах на JVM-языках является сбор трасс выполнения (деревьев вызовов) с использованием библиотеки AspectJ.

2. На основе библиотеки AspectJ реализован Java-агент для сборки деревьев вызовов и реализована цепочка фильтров для выделения наиболее существенных узлов деревьев вызовов.

3. Сбор и сжатие деревьев вызовов внедрены в инструмент запуска тестов AI-агента VeAI.

4. В эксперименте на реальных проектах измерено увеличение доли успешных исправлений агентом VeAI в правильных файлах на 19%.

Список литературы

[1] ASSERT Lab, KTH Royal Institute of Technology. HumanEval-Java. — GitHub repository. — 2023. — URL: https://github.com/ASSERT-KTH/human-eval-java (дата обращения: 5 января 2026 г.).

[2] Bouzenia Islem, Devanbu Premkumar T., Pradel Michael. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair // Proceedings of the IEEE/ACM 47th International Conference on Software Engineering. — ICSE '25. — IEEE, 2025. — P. 2188–2200. — URL: https://doi.org/10.1109/ICSE55347.2025.00157 (дата обращения: 5 января 2026 г.).

[3] Broadcom Inc. Spring AOP. — 2025. — URL: https://docs.spring.io/spring-framework/reference/core/aop.html (дата обращения: 5 января 2026 г.).

[4] Broadcom Inc. Spring Framework. — 2025. — URL: https://spring.io/ (дата обращения: 5 января 2026 г.).

[5] BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes / David A. Tomassi, Naji Dmeiri, Yichen Wang et al. // Proceedings of the 41st International Conference on Software Engineering. — ICSE '19. — Piscataway, NJ, USA : IEEE Press, 2019. — P. 339–349. — URL: https://doi.org/10.1109/ICSE.2019.00048 (дата обращения: 5 января 2026 г.).

[6] Chiba Shigeru. Javassist: Java Bytecode Engineering Toolkit. — GitHub repository. — 2023. — URL: https://www.javassist.org/ (дата обращения: 5 января 2026 г.).

[7] Cloud Native Computing Foundation. OpenTelemetry for Java. — 2025. — URL: https://opentelemetry.io/docs/instrumentation/java/ (дата обращения: 5 января 2026 г.).

[8] Quan Shanghaoran, Yang Jiaxi, Yu Bowen et al. CodeElo: Benchmarking Competition-level Code Generation of LLMs with Human-comparable Elo Ratings. — 2025. — 2501.01257.

[9] Eclipse Foundation. AspectJ. — 2025. — URL: https://eclipse.dev/aspectj/ (дата обращения: 5 января 2026 г.).

[10] Chen Mark, Tworek Jerry, Jun Heewoo et al. Evaluating Large Language Models Trained on Code. — 2021. — 2107.03374.

[11] Explainable Automated Debugging via Large Language Model-driven Scientific Debugging / Sungmin Kang, Bei Chen, Shin Yoo, Jian-Guang Lou // Empirical Software Engineering. — 2025. — Vol. 30, no. 45. — P. 1–28. — URL: https://link.springer.com/article/10.1007/s10664-024-10594-x (дата обращения: 5 января 2026 г.).

[12] Arab Maryam, Liang Jenny T., Hong Valentina, LaToza Thomas D. How Developers Choose Debugging Strategies for Challenging Web Application Defects. — 2025. — 2501.11792.

[13] Wang Yunkun, Zhang Yue, Li Guochang et al. InspectCoder: Dynamic Analysis-Enabled Self Repair through Interactive LLM-Debugger Collaboration. — 2025. — 2510.18327.

[14] Volkov Alexey, Volkov Ivan, Stepanov Daniil et al. JacoDB: Java Compilation Database. — GitHub repository. — 2024. — URL: https://github.com/UnitTestBot/jacodb (дата обращения: 5 января 2026 г.).

[15] JetBrains. IntelliJ IDEA. — 2025. — URL: https://jetbrains.com/idea/ (дата обращения: 5 января 2026 г.).

[16] LLMDB. CodeForces Benchmark. — 2025. — URL: https://llmdb.com/benchmarks/codeforces (дата обращения: 5 января 2026 г.).

[17] Ni Ansong, Allamanis Miltiadis, Cohan Arman et al. NExT: Teaching Large Language Models to Reason about Code Execution. — 2024. — 2404.14662.

[18] OW2 Consortium. ASM: A Java Bytecode Manipulation Framework. — OW2 project. — 2025. — URL: https://asm.ow2.io/ (дата обращения: 5 января 2026 г.).

[19] OpenAI. A Practical Guide to Building Agents. — 2025. — URL: https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf (дата обращения: 5 января 2026 г.).

[20] Oracle Corporation. Java Debug Interface (JDI). — 2014. — URL: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/ (дата обращения: 5 января 2026 г.).

[21] Paulsen Norman. Context Is What You Need: The Maximum Effective Context Window for Real World Limits of LLMs. — 2025. — 2509.21361.

[22] Prenner Julian Aron, Robbes Romain. RunBugRun – An Executable Dataset for Automated Program Repair. — 2023. — 2304.01102.

[23] Austin Jacob, Odena Augustus, Nye Maxwell et al. Program Synthesis with Large Language Models. — 2021. — 2108.07732.

[24] ROSE: An IDE-Based Interactive Repair Framework for Debugging / Steven P. Reiss, Xuan Wei, Jiahao Yuan, Qi Xin // ACM Transactions on Software Engineering and Methodology. — 2024. — Vol. 34, no. 4. — 39 p. — URL: https://doi.org/10.1145/3705306 (дата обращения: 5 января 2026 г.).

[25] Re-factoring based Program Repair applied to Programming Assignments / Yang Hu, Umair Z. Ahmed, Sergey Mechtaev et al. // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). — IEEE/ACM, 2019. — P. 388–398. — URL: https://doi.org/10.1109/ASE.2019.00044 (дата обращения: 5 января 2026 г.).

[26] SWE-bench: Can Language Models Resolve Real-World GitHub Issues? / Carlos E. Jimenez, John Yang, Alexander Wettig et al. // Proceedings of the 12th International Conference on Learning Representations (ICLR 2024). — 2024. — 2310.06770.

[27] Arzt Steven, Lhoták Ondřej, Benz Manuel et al. Soot: A Java Optimization Framework. — GitHub repository. — 2025. — URL: https://github.com/soot-oss/soot (дата обращения: 5 января 2026 г.).

[28] A Survey on Large Language Models for Code Generation / Juyong Jiang, Fan Wang, Jiasi Shen et al. // ACM Transactions on Software Engineering and Methodology. — 2024. — URL: https://doi.org/10.1145/3747588 (дата обращения: 5 января 2026 г.).

[29] Haque Mirazul, Babkin Petr, Farmahinifarahani Farima, Veloso Manuela. Towards Effectively Leveraging Execution Traces for Program Repair with Code LLMs. — 2025. — May. — 2505.04441.

[30] UnitTestBot. UTBotJava: Automated Unit Test Generation and Precise Code Analysis for Java. — GitHub repository. — 2025. — URL: https://github.com/UnitTestBot/UTBotJava (дата обращения: 5 января 2026 г.).

[31] Unsupervised Translation of Programming Languages / Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, Guillaume Lample // Advances in Neural Information Processing Systems 33 (NeurIPS 2020). — 2020. — 2006.03511.

[32] Winterhalter Rafael. Byte Buddy: Runtime Code Generation for the Java Virtual Machine. — GitHub repository. — 2025. — URL: https://bytebuddy.net/ (дата обращения: 5 января 2026 г.).

[33] Zhong Li, Wang Zilong, Shang Jingbo. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step // Findings of the Association for Computational Linguistics: ACL 2024 / Ed. by Lun-Wei Ku, Andre Martins, Vivek Sriku-

mar. — Bangkok, Thailand : Association for Computational Linguistics, 2024. — Aug. — P. 851–870. — URL: https://aclanthology.org/2024.findings-acl.49/ (дата обращения: 5 января 2026 г.).

[34] ООО «ИИТЕХ». VeAI. — 2025. — URL: https://veai.ru/ (дата обращения: 5 января 2026 г.).

[35] ООО «ИИТЕХ». Матрица функциональности VeAI. — 2025. — URL: https://veai.ru/docs/veai/feature-matrix (дата обращения: 5 января 2026 г.).