ТОПОЛОГИЧЕСКИЙ AST-СЕНСОР ДЛЯ ИНФРАСТРУКТУРНЫХ ИИ -АГЕНТОВ, ПРЕДОТВРАЩЕНИЕ КАСКАДНЫХ СБОЕВ CI/CD В ПАРАДИГМЕ AI-TO-AI ВЗАИМОДЕЙСТВИЯ

Мухамадиева Кибриё Баходировна

Узбекский государственный университет мировых языков
mkb78@mail.ru
DOI: 10.61587/mmit.tiue.uz.v1i1.432

Аннотация: В данной работе доказывается, что для эффективного AI-to-AI взаимодействия инфраструктурным агентам требуется специализированный "сенсорный" слой. Мы предлагаем архитектуру автономного перехватчика с модулем топологической фильтрации на базе абстрактных синтаксических деревьев (AST), интегрируемого непосредственно в shell-оболочку CI/CD раннера. Наш инструмент не заменяет агентов конвейера развертывания, а действует как микрохирургическая оптика для них. Алгоритм детерминированно извлекает строгий семантический подграф G , сжимая контекст с $O(N)$ до $O(K)$ токенов.

Ключевые слова: *AI-DevOps, Мультиагентные системы, CI/CD, RAG с поддержкой AST, Топологическая фильтрация, Галлюцинации LLM, MTTR, Zero-Human-in-the-Loop.*

TOPOLOGICAL AST SENSOR FOR INFRASTRUCTURE AI AGENTS, PREVENTING CASCADING CI/CD FAILURES IN THE AI-TO-AI INTERACTION PARADIGM

Mukhamadieva Kibriyo Bahodirovna

Lecturer of the department of "Modern Information Technologies and artificial intelligence". Uzbek State University of World Languages
mkb78@mail.ru

Abstract: This paper proves that for effective AI-to-AI interaction, infrastructure agents require a specialized "sensory" layer. We propose an architecture for an autonomous interceptor with a topological filtering module based on abstract syntax trees (AST), which can be integrated directly into the CI/CD runner shell. Our tool does not replace the agents of the deployment pipeline, but acts as microsurgical optics for them. The algorithm deterministically extracts a strict semantic subgraph G , compressing the context from $O(N)$ to $O(K)$ tokens.

Keywords: *AI-DevOps, Multi-Agent Systems, CI/CD, AST-Aware RAG, Topological Filtering, LLM Hallucinations, MTTR, Zero-Human-in-the-Loop.*

ВВЕДЕНИЕ

Индустрия программной инженерии завершила исторический переход от классической ручной разработки к парадигме фабрик кодогенерации, управляемых многоагентными когнитивными системами. Современные высоконагруженные пайплайны проектируются как сквозные AI-to-AI конвейеры. На этапе до фиксации изменений автономные ИИ-разработчики генерируют бизнес-логику и покрывают ее тестами. На этапе развертывания в дело вступают высокоуровневые инфраструктурные агенты, отвечающие за оркестрацию контейнеров, управление манифестами и доставку артефактов.

Мы представляем архитектуру автономного перехватчика с интегрированным модулем топологической фильтрации (AST-Aware Context Extraction). Данный модуль не является самостоятельной заменой AI-DevOpsy конвейера развертывания; напротив, он

выступает в роли его точного "оптического сенсора". Внедряясь на низкоуровневый слой shell-оболочки раннера, сенсор активируется в момент аварийного прерывания процесса. Вместо того чтобы позволить CI/CD системе уничтожить контейнер и отправить сырой текстовый лог внешнему оркестратору, наш инструмент замораживает окно восстановления $(T_{recovery})$ и детерминированно транслирует упавший код из плоского текста в строгий направленный граф (Абстрактное синтаксическое дерево).

В рамках данного исследования мы представляем следующие ключевые научно-практические вклады, формирующие архитектурный базис для надежной эксплуатации сквозных AI-to-AI конвейеров:

ОБЗОР ЛИТЕРАТУРЫ

Эволюция методов автоматического устранения дефектов (Automated Program Repair, APR) исторически протекала в рамках парадигмы «человеко-центричного» ассистирования. Ранние фундаментальные работы базировались на поисково-эвристических стратегиях и генетических алгоритмах. Наиболее репрезентативным примером является система GenProg [1], использующая генетическое программирование для стохастического поиска патчей. Параллельно развивались методы, основанные на заранее определенных синтаксических шаблонах, такие как TBar [2] и Prophet [3], которые демонстрировали высокую эффективность в изолированных лабораторных условиях.

Традиционная оценка данных подходов проводилась на синтетических бенчмарках для монолитных языков, в частности Defects4J [4]. В рамках этих тестов программные сбои интерпретировались как строго локализованные логические ошибки внутри конкретных функций. Однако, как отмечают исследователи [5], подобная изоляция не учитывает динамику современных распределенных систем, что ставит под сомнение масштабируемость классических методов в реальных производственных средах. Массовое внедрение AI-фабрик и многоагентных систем генерации кода коренным образом изменило характер программных сбоев. В современных конвейерах типа «AI-to-AI» исходный код, синтезируемый пре-коммит агентами, характеризуется высокой логической консистентностью благодаря встроенным механизмам саморефлексии и итеративного тестирования [6]. Вследствие этого центр тяжести ошибок сместился на инфраструктурный уровень.

Современные исследования указывают на то, что падения пайплайнов теперь детерминированы дрейфом конфигураций сред исполнения, несовместимостью API-контрактов и конфликтами транзитивных зависимостей внутри эфемерных контейнеров [7]. В этой новой реальности классический математический аппарат APR оказывается концептуально несостоятельным. Основная причина заключается в комбинаторном взрыве пространства поиска, когда эвристические алгоритмы пытаются мутировать абстрактное синтаксическое дерево (AST) в условиях динамичного облачного CI/CD. Более того, классический APR жестко ограничен требованием исчерпывающего тестового покрытия для оценки функции приспособленности, что технически недостижимо на этапах инициализации среды, таких как pip install или npm run build [8]. Аблационные исследования подтверждают, что при инфраструктурном сбое векторный поиск успешно извлекает фрагмент с местом падения, но детерминированно «отсекает» блоки глобальных импортов и родительских классов из-за отсутствия текстового сходства со строками трассировки стека [9]. Получая семантически разорванный контекст, ИИ-агент впадает в состояние «информационной слепоты», что ведет к генерации галлюцинаторных вызовов и синтаксически корректных, но семантически невалидных патчей.

Таким образом, текущее состояние литературы подтверждает необходимость разработки специализированных «сенсоров», способных детерминированно сохранять топологию графа исходного кода для обеспечения жизнеспособности автономных AI-to-AI конвейеров.

МЕТОДОЛОГИЯ

В данном разделе описывается фундаментальная архитектура интеграции топологического сенсора в сборочный конвейер. Традиционные системы непрерывной интеграции (такие как инстансы GitHub Actions, Jenkins или GitLab CI) исторически спроектированы в рамках жесткой реактивной парадигмы. Они рассматривают пайплайн как дискретный "черный ящик" с бинарным исходом: при возникновении любой критической ошибки (характеризующейся системным прерыванием с кодом $exitcode \neq 0$) система немедленно прерывает выполнение фазы, сбрасывает буфер стандартного вывода в статичный текстовый лог-файл и безвозвратно уничтожает эфемерную среду выполнения.

Для инфраструктурных AI-DevOps агентов, оперирующих во внешнем контуре управления AI-фабрики, такое деструктивное поведение является архитектурным тупиком. Получая лишь "мертвый" текстовый слепок системы (лог), внешний агент полностью лишается доступа к живому состоянию файловой системы, транзитным артефактам и динамически сформированным конфигурациям, которые физически спровоцировали сбой. Попытка восстановить картину сбоя по тексту приводит к галлюцинациям[10].

В момент детекции аварийного прерывания Error Interceptor императивно блокирует протокол автоматического уничтожения контейнера. Алгоритм формирует строго детерминированное "окно восстановления" ($T_{recovery}$). В течение этого временного окна инфраструктура переводится в состояние гибернации: выполнение шагов CI/CD приостанавливается, но файловая система, иерархия локальных директорий, загруженные модули и дерево исходного кода сохраняют свое точное физическое состояние на миллисекунду падения.

Создание защитного окна ($T_{recovery}$).является математически необходимым условием для инициализации AST-сенсора. Именно это физическое удержание среды позволяет сенсору приступить к построению абстрактного синтаксического дерева на основе реальных, неповрежденных файлов внутри контейнера, а не их абстрактных лог-копий. Обеспечивая удержание среды, Error Interceptor позволяет внешнему AI-DevOps агенту проводить микрохирургическое вмешательство непосредственно "на операционном столе" (внутри контейнера), аппаратно предотвращая инициацию дорогостоящей петли $O(n^2)$ полного перезапуска конвейера развертывания.

После того как демон-перехватчик успешно блокирует процесс уничтожения контейнера и формирует окно гибернации ($T_{recovery}$), топологический сенсор инициирует фазу первичной координатной привязки. В этот момент физическое состояние файловой системы зафиксировано, однако сенсору требуется определить точную алгоритмическую точку входа для построения графа. Отправной точкой служит сырой текстовый лог выполнения упавшего пайплайна, который захватывается из потоков стандартного вывода и ошибок (stdout/stderr). Математически этот дамп представляется как многомерный массив последовательных строк $L = \backslash\{l_1, l_2, \dots, l_n\backslash\}$.

В реалиях AI-to-AI конвейеров, где AI-DevOps оркестраторы непрерывно компилируют и тестируют гигантские объемы кода, генерируемого внешними агентами, сырой лог L характеризуется экстремально высокой информационной энтропией. Он перенасыщен шумом: отладочными сообщениями сторонних сервисов, варнингами

линтеров и логами загрузки транзитивных зависимостей. Наивная передача всего массива L в контекстное окно инфраструктурного ИИ-агента - это классический антипаттерн, приводящий к вычислительному параличу и астрономическим затратам на токены ($TokenCost$). Для предотвращения этого сенсор реализует аппаратный модуль лексической локализации, который работает строго детерминированно, без привлечения генеративных нейросетей [10].

Основой этого модуля является высокоскоростной алгоритм обратного анализа трассировки стека. В отличие от поиска регулярными выражениями (Regex) по всему тексту лога, алгоритм сканирует фреймы вызовов снизу вверх, начиная анализ с последнего кадра, физически зафиксировавшего фатальное исключение. Фундаментальным предохранителем на этом этапе является жесткий эвристический фильтр системных путей. Сенсор автоматически отсекает все ветви стека, ведущие в системные директории раннера. Эта пространственная изоляция гарантирует, что "оптика" будет сфокусирована исключительно на пользовательском пространстве - то есть на исходном коде, сгенерированном Pre-commit ИИ-агентами.

Результатом работы этапа лексической локализации является трансформация

В данной формуле:

•τ (Target File) - абсолютный системный путь к сбойному файлу внутри заблокированного контейнера.

•ρ (Target Line) - точный целочисленный номер строки исходного кода, на которой прервался вычислительный процесс.

•ϵ (Error Semantics) - инкапсулированный тип инфраструктурной или логической ошибки с сопутствующим текстовым сообщением среды исполнения.

Сформированный кортеж F выступает в роли "прицела" для AST-сенсора. Он передается на следующий этап конвейера, где позволит детерминированно спроецировать ошибку из текстовой плоскости на многомерную структуру абстрактного синтаксического дерева, полностью исключив "слепые" зоны в работе AI-DevOpsa. Получив от модуля лексической локализации точные пространственные координаты сбоя в виде кортежа $F = (\tau, \rho, \epsilon)$, инфраструктурный AI-DevOps сталкивается с фундаментальной алгоритмической дилеммой извлечения контекста. Исходный файл τ может содержать десятки тысяч строк. Наивная передача всего объема файла в языковую модель неминуемо провоцирует эффект рассеивания внимания и экспоненциально сжигает вычислительные бюджеты на инференс [5-7].

Для сжатия входного окна индустрия стандартизировала применение плотного векторного поиска (Vector RAG / Dense Retrieval). В этой парадигме исходный код механически фрагментируется на перекрывающиеся лексематические окна (чанки, $token\_512$), которые затем извлекаются на основе максимизации косинусного сходства их эмбеддингов с текстом системной ошибки ϵ Однако, применяя методологию First Principles к физике программного обеспечения, мы констатируем: исходный код не является линейным текстом на естественном языке. Математически он представляет собой строгий направленный граф зависимостей и областей видимости. Для предотвращения каскадного коллапса предложенная архитектура вводит теорию топологической фильтрации. Этот подход требует полного отказа от векторного извлечения в контуре CI/CD. Вместо стохастического поиска, AST-сенсор детерминированно транслирует сбойный файл в абстрактное синтаксическое дерево $G = (V, E)$. Передавая AI-DevOpsy не обрывки текста с высокой косинусной близостью, а строго математически связанные узлы графа зависимостей, сенсор аппаратно

гарантирует сохранение контекста исполнения, делая галлюцинации переменных структурно невозможными.

Пусть сбойный файл τ, локализованный на предыдущем этапе конвейера, транслируется парсером в направленный ациклический граф (Абстрактное синтаксическое дерево). Данный граф строго определяется как $G = (V, E)$, где V - множество синтаксических узлов (определения классов, функций, глобальных импортов), а E - множество направленных ребер, физически отражающих иерархическую вложенность областей видимости (scope containment).

Каждый синтаксический узел $v \in V$ аппаратно наделен метаданными

$$v_{target} = \underset{v \in V}{argmin} \backslash\{end_v - start_v \backslash\}$$

Процесс выполняется при строгом соблюдении пространственного граничного

Передача всего дерева G во внешний контур управления AI-DevOpsy алгоритмически нецелесообразна из-за квадратичной сложности вычислений механизма внимания в архитектуре трансформеров. Поэтому сенсор применяет строгую функцию топологической фильтрации $f_{filter}: G \to G$, формируя усеченный семантический подграф $G = (V, E)$. Множество сохраняемых узлов V формируется как детерминированное объединение критических компонентов:

$$V = \backslash\{v_{target} \backslash\} \cup Anc(v_{target}) \cup V_{imports} \cup Sig(v_{target})$$

В данном уравнении $Anc(v_{target})$ представляет множество всех предков целевого узла вплоть до корня дерева, что аппаратно гарантирует AI-DevOpsy абсолютное понимание контекста родительских классов. Элемент $V_{imports}$ - это множество узлов глобальных импортов. Его принудительное включение в подграф навсегда устраняет фундаментальный изъян векторного поиска, предотвращая риск каскадных галлюцинаций и ошибок типизации при генерации патчей. Множество $Sig(v_{target})$ включает исключительно сигнатуры сторонних функций, вызываемых внутри целевого узла, отсекая их внутреннюю реализацию для агрессивной экономии контекстного окна. На финальном шаге выделенный подграф G транслируется обратно в текстовый формат с помощью реверсивного парсера (UnparseToSource), формируя компактную строку C . Алгоритмическая сложность экстракции строго ограничена линейным временем $O(N)$, где N - количество токенов исходного кода. В результате объем входных данных, передаваемых в "мозг" инфраструктурного ИИ-агента, детерминированно сжимается с $O(N)$ до $O(K)$, где $K \ll N$. Это наделяет агента конвейера идеальной, математически выверенной оптикой для синтеза патча. Сформированный на предыдущем этапе топологически отфильтрованный подграф исходного кода C , совместно с кортежем пространственных координат сбоя $F = (\tau, \rho, \epsilon)$, передается в генеративное ядро инфраструктурного агента (AI-DevOpsa). В условиях полностью автономного AI-to-AI конвейера критически важным архитектурным требованием является строгая алгоритмическая регламентация "свободы воли" большой языковой модели. Любое отклонение от ожидаемого формата вывода неизбежно разрушает пайплайн автоматического применения исправлений.

Для минимизации стохастичности генеративных сетей применяются строго зафиксированные гиперпараметры стратегии декодирования (Decoding Strategy). Температурный параметр устанавливается на экстремально низком уровне ($T \in [0.02,0.1]$), что смещает распределение вероятностей в сторону наиболее математически ожидаемых токенов. Дополнительно применяется динамическое отсечение м

Синтезированный AI-DevOpsом патч категорически запрещено применять к основной ветке репозитория "вслепую", так как это нарушает фундаментальные принципы изоляции отказов (Fault Isolation) в облачных инфраструктурах. В предложенной архитектуре реализуется итеративный цикл изолированной микро-валидации (Sandbox Feedback Loop). Полученный diff-файл накладывается на код внутри гибернированного контейнера (внутри окна $T_{recovery}$), после чего перехватчик локально перезапускает упавший шаг пайплайна (например, pytest или npm run build). Если процесс завершается успешно ($exitcode = 0$), AI-DevOps получает подтверждение, автономно формирует git commit и разблокирует CI/CD конвейер.

В случае, если патч провоцирует новое каскадное исключение, обновленный лог трассировки захватывается сенсором и передается обратно в модель в качестве негативного подкрепления (Negative Prompting) для повторной итерации. Для предотвращения возникновения бесконечных циклов восстановления и контроля экономических затрат на вычисления, в систему внедрен жесткий математический лимит попыток: $M_{retries} \leq 3$. По истечении данного лимита агент детерминированно выполняет безопасный откат (rollback) изменений и эскалирует инцидент, сохраняя стабильность системы.

Для обеспечения безупречной научной воспроизводимости архитектура сенсора и генеративного контура тестировалась в строго детерминированной среде. Каждый инцидент был инкапсулирован в воспроизводимый Docker-контейнер, эмулирующий стандартный эфемерный раннер (аналог GitHub Actions, Ubuntu 22.04 LTS, 2 vCPU, 7 ГБ RAM). Ключевой инновацией экспериментального стенда стала аппаратная фиксация состояния файловой системы контейнера за миллисекунду до аварийного прерывания процесса (сразу после детекции $exitcode \neq 0$). Это позволило создать абсолютно идентичное окно гибернации $T_{recovery}$ для каждого тестового прогона. Внешний инфраструктурный AI-DevOps подключался к этому "замороженному" окружению, что полностью исключило влияние сетевых флуктуаций при сравнительной оценке оптических сенсоров. Для объективной оценки вклада предложенного топологического AST-сенсора в решение кризиса масштабируемости сквозных AI-to-AI конвейеров, был проведен строгий сравнительный анализ. В качестве контрольных архитектурных подходов (baselines) рассматривались:

Использование детерминированных встроенных механизмов автоисправления современных статических анализаторов (например, ruff --fix или black). Данный бейзлайн представляет классический не-нейросетевой подход, лишенный ИИ-агентов.

Подход, при котором инфраструктурный AI-DevOps получает сырой текстовый лог L и целиком содержимое сбойного файла τ без какой-либо фильтрации графа.

Индустриальный стандарт наделения внешних агентов контекстом. Исходный код механически нарезается на перекрывающиеся лексематические чанки (размером 512 токенов с перекрытием 64), а $Top\text{-}K$ фрагментов извлекаются на основе косинусного с

В качестве единого генеративного ядра, исполняющего роль AI-DevOpsa во всех нейросетевых сценариях, использовалась открытая модель Meta-Llama-3-70B-Instruct. Выбор архитектуры с открытыми весами фундаментально обусловлен строгими

корпоративными стандартами безопасности. Реальные CI/CD пайплайны оперируют проприетарным исходным кодом, передача которого через публичные API закрытых моделей категорически запрещена. Инференс AI-DevOpsа осуществлялся на выделенном GPU-кластере (2 × NVIDIA A100 80GB VRAM) с использованием фреймворка vLLM для максимизации пропускной способности. Для предотвращения генерации невалидного синтаксиса, свойственного языковым моделям при избыточной креативности, гиперпараметры стратегии декодирования были жестко зафиксированы. Температура генерации T установлена на уровне 0.1, что критично для задач автоматического восстановления кода, требующих математической детерминированности. Параметр $Top\text{-}p$) зафиксирован на 0.95. Штрафы за лексические повторения (Frequency / Presence Penalty) установлены на 0.0, поскольку валидный программный код часто требует повторов при инициализации и вызове методов. Максимальное окно генерации эмпирически ограничено 1024 токенами - объемом, достаточным для вывода diff-файлов без преждевременного обрыва потока.

Эффективность топологической оптики оценивалась по трем ортогональным метрикам, отражающим жизнеспособность системы самовосстановления в условиях массовой генерации кода:

•Pass@K. Строгая метрика успешности исправления, где K - максимальное количество циклов верификации (попыток коммита внутри окна $T_{recovery}$). Мы замеряем $Pass@1$ (успех с первого "выстрела") и $Pass@3$ (успех после получения обратной связи от песочницы). Патч считается валидным ($Pass = 1$), только если повторный запуск возвращает $exitcode = 0$ и проходят все unit-тесты.

•

.

Критическая метрика пропускной способности инфраструктуры, рассчитываемая как время в секундах от момента перехвата сигнала завершения до успешного применения валидного патча (MTTR = $t_{commit} - t_{crash}$).

•Token Cost Efficiency (Вычислительная эффективность). Суммарный объем токенов C_{token} (prompt + completion), затраченный на один успешный патч. Эта метрика доказывает экономическую целесообразность использования AST-сенсора для снижения нагрузки на бюджет AI-фабрики.

ОБСУЖДЕНИЕ И РЕЗУЛЬТАТЫ

В рамках сравнительного анализа детерминированный подход (Static Linter Auto-fix) продемонстрировал свою полную фундаментальную несостоятельность в роли автономной "иммунной с

беспрецедентный показатель $Pass@1$ равный 94.2% для ошибок синтаксиса и типов, 82.5% для конфликтов зависимостей и 71.3% для дрейфа конфигураций.

Интеграция замкнутого цикла изолированной верификации внутри окна гибернации $T_{recovery}$ с алгоритмическим лимитом $M_{retries} \leq 3$ позволила AI-DevOpsy безопасно обучаться на собственных ошибках. Это подняло итоговую успешность автономного исправления ($Pass@3$) до 98.1% (синтаксис), 88.4% (зависимости) и 85.0% (конфигурации) соответственно. В абсолютном выражении, обеспечение инфраструктурных агентов правильной топологической оптикой позволяет им разрешать до 91.4% инцидентов без эскалации на человека, доказывая критическую необходимость AST-сенсора для масштабирования конвейеров производства кода.

Механика алгоритмического провала векторного подхода обусловлена самой математической природой поиска по косинусному сходству. Рассмотрим типичный сценарий - эфемерный контейнер падает с ошибкой несоответствия API-контракта на условной строке 452. Стандартный векторный RAG извлекает текстовые чанки, эмбеддинги которых максимально близки к тексту системного исключения ϵ. В результате "мозг" AI-DevOpsa получает окно контекста, содержащее исключительно окрестности строки 452. Однако блок глобальных импортов, традиционно располагающийся в начале исходного файла (строки 1–15), обладает околонулевой семантической векторной близостью к тексту низкоуровневой ошибки. Следовательно, векторный алгоритм детерминированно «слепнет» и отсекает критически важную информацию о загруженных модулях, их алиасах и версиях зависимостей. Получив такой семантически разорванный контекст, инфраструктурный AI-DevOps попадает в ловушку "информационной слепоты". Пытаясь устранить конфликт типов, модель инициирует вызовы сторонних библиотек, опираясь на галлюцинированные зависимости, которых физически нет в пространстве имен целевого файла τ. Это запускает ресурсоемкий цикл верификации внутри окна $T_{recovery}$, который вхолостую сжигает вычислительные квоты, провоцирует вторичные каскадные сбои и неизбежно упирается в установленный лимит итераций $M_{retries}$, так и не разрешив инцидент.

В противовес этому, предложенный нами топологический AST-сенсор решает данную проблему на уровне физики направленных графов. Согласно математической формализации подграфа G , множество узлов импортов $V_{imports}$ и предков $Anc(v_{target})$ аппаратно включается в итоговый промпт независимо от их текстового расстояния до места сбоя. Данная детерминированная привязка гарантирует инфраструктурному агенту абсолютное сохранение лексической области видимости. Количественные результаты аблационного исследования безоговорочно подтверждают эту архитектурную гипотезу. В то время как Vector RAG смог обеспечить AI-DevOps успешное разрешение лишь 38.5% конфликтов зависимостей с первой попытки ($Pass@1$), снабжение агента AST-сенсором продемонстрировало показатель на уровне 82.5%. Этот колоссальный разрыв в 44 процентных пункта строго доказывает, что для автономного восстановления инфраструктуры сохранение иерархических топологических связей кода несравнимо важнее текстовой плотности информации. Таким образом, AST-сенсор является единственно жизнеспособной "оптикой" для ИИ-агентов в runtime-средах непрерывной доставки. В классической парадигме делегирования сбоев внешним инфраструктурным ИИ-агентам, метрика MTTR неизбежно включает в себя затраты на физическое уничтожение эфемерного контейнера, асинхронную переброску массивных текстовых логов в оркестратор, генерацию "слепого" патча и полный перезапуск сборочной линии. Эмпирические замеры показывают, что этот цикл занимает от 15 до 40 минут вычислительного времени облака. В противовес этому, связка AI-DevOpsa со встроенным перехватчиком, оперирующим внутри замороженного окна $T_{recovery}$, радикально меняет физику процесса. Медианное время автономного восстановления (от

перехвата POSIX-сигнала до успешного git push изнутри песочницы) составило всего 24 секунды.

Результаты нашего исследования ставят под сомнение доминирующую парадигму «грубой силы» в облачных вычислениях. Традиционный подход - увеличение контекстного окна LLM до миллионов токенов - не устраняет проблему "информационного шума" в логах CI/CD. Напротив, наши данные подтверждают наличие эффекта «Lost-in-the-Middle», т.е. при подаче полного текста сбойного микросервиса, точность инфраструктурного ИИ-агента падает до 38.5% на сложных конфликтах зависимостей. Это доказывает, что для надежного AI-to-AI взаимодействия первичным является не объем данных, а их топологическая чистота. Фундаментальный успех предложенной архитектуры (91.4% успешных патчей) объясняется тем, что мы перевели задачу из плоскости «угадывания по тексту» в плоскость «навигации по графу». Интеграция AST-сенсора решает проблему семантической слепоты ИИ-агентов. В то время как Vector RAG оперирует статистической вероятностью соседства слов, наш метод оперирует физикой программной инженерии - иерархией вызовов и импортов. Таким образом, AST-сенсор выполняет роль «очков» для AI-DevOpsa, позволяя ему видеть не просто текст, а структуру связей внутри заблокированного контейнера. Снижение Token Cost в 22 раза имеет критическое значение для промышленного внедрения ИИ-фабрик. В условиях, когда количество коммитов, генерируемых нейросетями, растет экспоненциально, стоимость каждой ошибки в CI/CD становится блокирующим фактором. Предложенная методология не просто исправляет код - она защищает инфраструктуру от «галлюциногенного резонанса», когда одна неверная правка ИИ-агента порождает серию NameError исключений, парализуя работу всей команды.

Мы констатируем, что ролевых моделей самих по себе недостаточно для обеспечения стабильности. Без низкоуровневых инструментов локализации сбоев, эти агенты остаются «генералами без разведданных». Наше исследование доказывает, что будущее автономных облачных сред лежит в гибридизации стохастических нейросетей и детерминированных анализаторов кода (AST). Только такая симбиотическая архитектура способна превратить CI/CD из реактивного конвейера в самозалечивающуюся экосистему.

ЗАКЛЮЧЕНИЕ

В данной работе мы представили архитектуру топологического AST-сенсора - критически важного компонента для обеспечения стабильности и экономической эффективности современных AI-to-AI конвейеров производства программного обеспечения. Исследование подтвердило, что фундаментальным барьером на пути к полной автономности инфраструктурных ИИ-агентных систем является «семантическая слепота» традиционных методов извлечения контекста, таких как Vector RAG.

Доказано, что использование детерминированного подграфа G , включающего иерархию наследования и глобальные импорты, устраняет риск каскадных галлюцинаций (NameError), повышая успешность автономного восстановления до 91.4% $Pass@3$). Внедрение демона-перехватчика и создание окна гибернации $T_{recovery}$ позволило сократить медианное время восстановления инфраструктуры с десятков минут до 24 секунд. Применение топологической фильтрации контекста обеспечивает 22-кратное снижение затрат на инференс больших языковых моделей, делая масштабирование ИИ-фабрик кодогенерации финансово устойчивым.

СПИСОК ЛИТЕРАТУР:

1. Liu, K., et al. (2019). TBar: Revisiting Template-based Automated Program Repair. International Symposium on Software Testing and Analysis (ISSTA). https://dl.acm.org/doi/10.1145/3293882.3330577
2. Long, F., & Rinard, M. (2016). Automatic synthesis of code forces for software repair. Principles of Programming Languages (POPL).
https://people.csail.mit.edu/rinard/paper/popl16.pdf
3. Just, R., et al. (2014). Defects4J: A database of existing faults to enable controlled studies for Java programs. ISSTA. https://dl.acm.org/doi/10.1145/2610384.2628055
4. Gazzola, L., et al. (2019). Automatic Software Repair: A Survey. IEEE Transactions on Software Engineering.
https://www.computer.org/csdl/journal/ts/2019/01/08089448/17D45WaTknO
5. Madaan, A., et al. (2023). Self-Refine: Iterative Refinement with Self-Feedback. arXiv preprint. https://arxiv.org/abs/2303.17651
6. Ajay Varma, (2025). Infrastructure as code: A paradigm shifts in cloud resource management and deployment automation. https://doi.org/10.30574/wjaets.2025.15.1.0478
7. Renicius Pagotto. (2025). The Importance of CI/CD in Modern Software Development. https://dev.to/reniciuspagotto/the-importance-of-cicd-in-modern-software-development-3d9k
8. Yang, J., et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Software Engineering Agents. Research Report. https://arxiv.org/abs/2405.15793
9. Jimenez, C. E., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR. https://arxiv.org/abs/2310.06770
10. Allamanis, M., et al. (2018). Learning to Represent Programs with Graphs. ICLR. https://arxiv.org/abs/1711.00740