УДК 004.415.2:004.65

Низамиев А. Р., магистрант, 2 курс, кафедра программной инженерии,
Казанский (Приволжский) федеральный университет, г. Казань

ГИБРИДНЫЙ МЕТОД ОБНАРУЖЕНИЯ АНТИПАТТЕРНОВ ПРОИЗВОДИТЕЛЬНОСТИ В SQLALCHEMY ORM-ПРИЛОЖЕНИЯХ

Аннотация

Рассматривается задача обнаружения антипаттернов производительности в Python-приложениях на базе SQLAlchemy ORM и FastAPI. Актуальность исследования обусловлена тем, что ORM-абстракция ускоряет разработку, но одновременно скрывает реальное число SQL-запросов, особенности загрузки связанных данных и транзакционные издержки, из-за чего проблемы производительности часто выявляются уже на поздних стадиях тестирования. Предложен гибридный метод, объединяющий статический анализ исходного кода и трассировку SQLAlchemy-событий во время выполнения существующих pytest- и интеграционных тестов. Статический слой анализирует Python AST, выделяет контекст маршрутов FastAPI и локализует потенциально проблемные ORM-конструкции. Слой анализа во время выполнения фиксирует повторяющиеся SQL-паттерны, события `commit` и `flush`, а также связывает фактическую SQL-нагрузку с конкретным HTTP-маршрутом. Реализован программный инструмент `fastapi_sqlalchemy_detector`, формирующий объяснимый гибридный отчет со статусами подтверждения находок. Экспериментальная проверка включала статический анализ 21 open-source проекта на стеке FastAPI + SQLAlchemy и гибридную проверку 4 проектов с

воспроизводимыми тестами. В статическом корпусе признаки потенциально неэффективного ORM-кода выявлены в 13 проектах из 21, причем 37 из 76 срабатываний были локализованы до уровня конкретного маршрута API. В гибридной выборке выполнено 168 тестов, собрано 230 трасс и 2242 SQL-запроса; дополнительно зафиксировано 116 runtime-находок, а для 16 из 21 статической находки гибридный слой дал уточняющую интерпретацию через подтверждение во время выполнения или указание на отсутствие тестового покрытия. Полученные результаты демонстрируют практическую применимость предложенного метода для ранней локализации, объяснения и приоритизации ORM-проблем производительности.

Ключевые слова: SQLAlchemy, FastAPI, ORM, производительность, антипаттерны, статический анализ, динамический анализ, гибридный анализ, pytest.

Abstract

The paper addresses the problem of detecting performance anti-patterns in Python applications built with SQLAlchemy ORM and FastAPI. The relevance of the study stems from the fact that ORM abstractions improve developer productivity while concealing the actual number of SQL queries, relationship loading behavior, and transaction overhead, which often causes performance issues to surface only at late testing stages. A hybrid method is proposed that combines static source-code analysis with runtime tracing of SQLAlchemy events during existing pytest and integration tests. The static layer analyzes Python AST, identifies FastAPI endpoint contexts, and localizes potentially problematic ORM constructs. The runtime layer captures repeated SQL patterns, `commit` and `flush` events, and links observed database load to concrete HTTP routes. The approach is implemented in the

`fastapi_sqlalchemy_detector` tool, which produces an explainable hybrid report with confirmation statuses. The empirical evaluation includes static analysis of 21 open-source FastAPI + SQLAlchemy projects and hybrid analysis of 4 projects with reproducible tests. In the static corpus, signs of potentially inefficient ORM code were found in 13 out of 21 projects, and 37 of 76 findings were localized to specific API routes. In the hybrid subset, 168 tests were executed, 230 traces and 2242 SQL queries were collected; 116 runtime findings were additionally captured, and for 16 of 21 static findings the hybrid layer provided stronger interpretation through runtime confirmation or explicit absence of test coverage. The results demonstrate the practical value of the proposed method for early localization, explanation, and prioritization of ORM-related performance issues.

Keywords: SQLAlchemy, FastAPI, ORM, performance, anti-patterns, static analysis, runtime analysis, hybrid analysis, pytest.

Введение

SQLAlchemy ORM занимает важное место в современном Python-backend стеке, поскольку позволяет ускорить разработку, повысить выразительность бизнес-логики и работать с объектной моделью вместо прямого низкоуровневого SQL [1, 2, 6, 9]. Вместе с тем удобство ORM-абстракции сопровождается потерей прозрачности: разработчик не всегда видит, в какой момент происходит обращение к базе данных, сколько SQL-запросов фактически выполняется и какова стоимость материализации данных.

Именно в этой скрытой части поведения и возникают устойчивые антипаттерны производительности: N+1 запросы, ленивые загрузки связанных объектов при сериализации ответа, вызовы `all()` без ограничения

результата, загрузка полной ORM-сущности при использовании лишь части полей, а также `commit()` и `flush()` внутри циклов [1, 2, 10–12, 14]. Для веб-приложений такие конструкции особенно критичны, поскольку даже локальная неэффективность одного маршрута API может масштабироваться в заметный рост задержек, нагрузки на базу данных и потребления памяти.

Существующие средства SQL-логирования, APM и профилирования помогают увидеть уже выполненные запросы, но часто требуют ручной интерпретации и не всегда позволяют уверенно локализовать конкретную строку ORM-кода, породившую проблему. Общие статические анализаторы, напротив, хорошо работают с исходным кодом, но обычно не учитывают семантику SQLAlchemy, стратегии загрузки связей и специфику FastAPI-маршрутов [3–5, 8, 13]. Поэтому возникает потребность в специализированном, объяснимом и воспроизводимом методе анализа ORM-кода.

Целью работы является разработка и апробация гибридного метода обнаружения антипаттернов производительности в SQLAlchemy ORM-приложениях. Научная новизна работы состоит в объединении AST-анализа Python-кода, выделения контекста FastAPI-маршрутов и трассировки SQLAlchemy-событий во время выполнения существующих тестов с последующим сопоставлением статических и динамических свидетельств. Практическая значимость определяется возможностью использовать результаты анализа для раннего поиска проблемных участков, их объяснения и приоритизации в процессе ревью кода и инженерной доработки.

Постановка задачи

Объектом исследования являются Python-приложения, использующие SQLAlchemy ORM в веб-контексте. Предметом исследования выступают методы автоматического обнаружения антипаттернов производительности ORM-кода.

Для достижения цели решались следующие задачи:

1. Сформировать прикладной каталог SQLAlchemy ORM-антипаттернов, релевантных для FastAPI-приложений.
2. Разработать статический анализатор, способный находить потенциально проблемные ORM-конструкции в исходном коде и связывать их с маршрутом API.
3. Реализовать трассировку SQLAlchemy-событий во время выполнения существующих pytest- и интеграционных тестов.
4. Построить гибридный механизм сопоставления статических находок и фактических симптомов во время выполнения.
5. Выполнить экспериментальную проверку метода на открытых проектах.

Предлагаемый метод

Предлагаемый метод включает три взаимосвязанных слоя.

Первый слой представляет собой статический анализ Python AST. На этом этапе выполняется обход файлов проекта, выделяются маршруты FastAPI, анализируются вызовы SQLAlchemy Session API, `execute`, `scalars`, `query`, `all`, `commit`, `flush`, `refresh`, а также обращения к relationship-полям и возврат ORM-результатов через `response_model`. Итогом работы слоя становится набор статических находок с указанием правила, файла, строки, контекста функции и маршрута API.

Второй слой основан на анализе во время выполнения. Во время запуска существующих тестов инструмент собирает SQLAlchemy-события: типы SQL-операций, количество `SELECT`, `INSERT`, `UPDATE`, `DELETE`, повторяемость SQL-шаблонов, а также события `commit` и `flush`. Принципиально важной особенностью является привязка собранных событий к конкретным HTTP-маршрутам или тестовым сценариям, если такой контекст доступен. Благодаря этому данные выполнения становятся не просто журналом SQL-активности, а источником интерпретируемых свидетельств о поведении конкретного участка приложения.

Третий слой формирует гибридный отчет. Статические находки сопоставляются с трассами выполнения по маршруту API, типу симптома и признакам SQL-поведения. В результате каждая находка получает один из статусов: `runtime confirmed`, `runtime not executed` или `static only`. Такая схема позволяет не смешивать отсутствие подтверждения с отсутствием проблемы и тем самым делает выводы анализа существенно более корректными и полезными для разработчика.

Каталог обнаруживаемых антипаттернов

В основу инструмента положен прикладной каталог антипаттернов, включающий как статические, так и динамические правила. На текущем этапе наиболее значимыми являются следующие группы:

Группа	Примеры правил	Интерпретация
Повторяющиеся чтения	`query_in_loop`, `query_helper_in_loop`, `query_in_comprehension`	риск N+1 и множественных однотипных `SELECT`
Материализация данных	`unbounded_result_all`, `all_then_len`, `all_then_slice`	загрузка лишних строк и рост потребления памяти
Связанные объекты	`response_model_without_eager_loading`, `relationship_access_without_eager_loading`	риск ленивой загрузки при сериализации
Избыточная загрузка	`overfetching_full_entity`	чтение полной сущности при использовании части полей
Поэлементные записи	`session_write_in_loop`, `session_commit_in_loop`,	избыточные обращения к БД и накладные расходы транзакций

Группа	Примеры правил	Интерпретация
	`session_flush_in_loop`, `refresh_after_flush`	

Каталог формировался по трем критериям: встречаемость в FastAPI + SQLAlchemy-коде, понятный механизм влияния на производительность и возможность автоматического обнаружения. Это обеспечивает баланс между прикладной ценностью, объяснимостью и воспроизводимостью результатов.

Реализация инструмента

Разработанный инструмент `fastapi_sqlalchemy_detector` реализован как Python-пакет с командным интерфейсом. Архитектура инструмента организована в виде последовательного конвейера: сканирование проекта, статический анализ, сбор runtime-трассы, обнаружение паттернов поведения и формирование гибридного отчета.

Ключевыми модулями являются:

6. `static_analysis` — анализ AST, построение проектного индекса, выделение маршрутов API и ORM-паттернов.
7. `rules` — каталог статических правил, уровней серьезности, сообщений и рекомендаций.
8. `runtime_trace` — сериализация и чтение SQLAlchemy-трасс.

9. `runtime_detection` — детекторы повторяющихся SQL-шаблонов, поэлементных записей, повторных `commit` и `flush`.
10. `hybrid` — сопоставление статических находок со свидетельствами выполнения.
11. `reporting` и `cli` — генерация текстовых, JSON, CSV и HTML-отчетов.

Выходные данные инструмента строятся вокруг сущности `finding`, включающей идентификатор правила, уровень серьезности, путь к файлу, строку, контекст маршрута API, описание проблемы и рекомендацию. Для гибридного режима в запись также добавляются свидетельства выполнения: SQL-шаблоны, счетчики запросов и сессионных событий, а также итоговый статус подтверждения. Тем самым инструмент поддерживает не только функцию обнаружения, но и функцию объяснения результатов анализа.

Экспериментальная проверка

Эксперимент проводился в два этапа. На первом этапе был выполнен массовый статический анализ 21 open-source проекта на стеке FastAPI + SQLAlchemy. На втором этапе была выделена подвыборка из 4 проектов, для которых удалось воспроизвести существующие pytest- и интеграционные тесты и собрать данные выполнения.

Результаты статического анализа

Статический анализ выявил 76 срабатываний в 13 проектах; еще 8 проектов не содержали находок. Для 37 случаев удалось определить связь с конкретным маршрутом API. По уровню серьезности получено 70 срабатываний уровня `medium` и 6 уровня `high`.

Таким образом, признаки потенциально неэффективного ORM-кода были обнаружены в 61,9% проектов корпуса, а почти половина статических срабатываний оказалась привязана к конкретному маршруту API. Это важно с практической точки зрения, поскольку повышает пригодность результатов для ревью кода и последующей адресной оптимизации.

Наиболее частыми оказались следующие правила:

Правило	Количество
`unbounded_result_all`	47
`response_model_without_eager_loading`	9
`overfetching_full_entity`	7
`session_write_in_loop`	5
`query_in_loop`	3

Полученные данные показывают, что наиболее распространенной проблемой является неограниченная материализация результата через `all()` без `limit/offset`. Это подтверждает, что в реальных REST API типовые проблемы производительности часто связаны не столько со сложными низкоуровневыми SQL-ошибками, сколько с повторяющимися паттернами неэффективного использования ORM-абстракции.

Результаты гибридной проверки

В гибридной выборке были выполнены 168 pytest-тестов, собраны 230 трасс выполнения и 2242 SQL-запроса. В тех же проектах статический слой

обнаружил 21 находку, а слой анализа во время выполнения зафиксировал 116 поведенческих срабатываний.

Сопоставление результатов дало следующую картину:

Показатель	Значение
Проектов в гибридной выборке	4
Выполненных pytest-тестов	168
Трасс выполнения	230
SQL-запросов	2242
Статических находок	21
Runtime-находок	116
`runtime confirmed`	3
`runtime not executed`	13
`static only`	5

Важный результат состоит в том, что гибридный слой дал дополнительную интерпретацию для 16 из 21 статической находки: либо обнаружил фактическое подтверждение проблемы во время выполнения, либо явно показал, что соответствующий маршрут не был покрыт тестами. Такая постановка особенно ценна для инженерной практики, поскольку позволяет различать неподтвержденность из-за отсутствия покрытия и неподтвержденность как результат реального расхождения между статическим сигналом и поведением системы.

Показательным примером стал проект `Social-Media-Api`. Для маршрута `GET /users/{user_id}/followers` статический анализ выявил `all()` без `limit/offset`, а runtime-трасса подтвердила выполнение `SELECT` без ограничения результата. Этот случай демонстрирует практическую ценность гибридного метода: статический слой локализует подозрительную конструкцию, а динамический слой показывает, что она действительно проявляется в тестовом выполнении.

Обсуждение результатов

Полученные результаты позволяют сделать несколько выводов.

Во-первых, статический слой доказал свою полезность как инструмент раннего поиска кандидатов на оптимизацию. Особенно важным является то, что значительная часть срабатываний оказалась локализована до уровня конкретного маршрута API, то есть результат анализа сразу пригоден для инженерной интерпретации.

Во-вторых, слой анализа во время выполнения существенно усиливает ценность статических находок. Он не только подтверждает часть проблем на реальных тестовых сценариях, но и позволяет явно зафиксировать дефицит тестового покрытия. В этом смысле гибридный отчет работает не только как средство диагностики ORM-антипаттернов, но и как индикатор того, насколько полно тесты отражают производственно значимые маршруты приложения.

В-третьих, отдельно следует отметить 116 runtime-находок, зафиксированных в гибридной выборке. Они показывают, что инструмент способен выявлять фактические симптомы неэффективного SQL/session-поведения, а не ограничивается только предсказанием потенциальных рисков

по исходному коду. Это расширяет практическую ценность подхода и делает его перспективным для дальнейшего развития в сторону более полноценных runtime-aware средств анализа.

К ограничениям работы относятся эвристический характер правил, частичная поддержка сложных межслойных архитектур, неполное покрытие async SQLAlchemy-сценариев и зависимость динамического слоя от воспроизводимости тестового окружения. Тем не менее уже текущая версия метода демонстрирует, что даже при этих ограничениях возможно получить объяснимый, воспроизводимый и практически полезный результат.

Заключение

В статье предложен гибридный метод обнаружения антипаттернов производительности в SQLAlchemy ORM-приложениях, объединяющий статический анализ исходного Python-кода и анализ поведения во время выполнения существующих тестов. Метод ориентирован на FastAPI-контекст и учитывает типичные ORM-проблемы: N+1 запросы, неограниченную материализацию, риски ленивой загрузки, избыточную загрузку сущностей и поэлементные операции записи.

Практическая реализация подхода выполнена в инструменте `fastapi_sqlalchemy_detector`. Экспериментальная проверка на открытых проектах показала, что предложенный метод позволяет не только находить потенциально проблемные участки кода, но и сопоставлять их с фактическими SQL-симптомами в тестовом выполнении. Это делает результаты анализа более интерпретируемыми, полезными для разработчика и пригодными для дальнейшего встраивания в практики инженерного контроля качества.

Дальнейшее развитие работы связано с расширением поддержки async SQLAlchemy, углублением межслойного анализа `endpoint -> service -> repository`, учетом runtime-only находок в итоговом статусе и интеграцией с дополнительными механизмами оценки влияния, включая длительность запросов и `EXPLAIN/query plan`.

Список литературы

1. Chen T.-H., Shang W., Jiang Z. M., Hassan A. E., Nasser M., Flora P. Detecting Performance Anti-patterns for Applications Developed using Object-relational Mapping // Proceedings of the 36th International Conference on Software Engineering. 2014. P. 1001-1012. DOI: 10.1145/2568225.2568259.
2. Chen T.-H., Shang W., Jiang Z. M., Hassan A. E., Nasser M. N., Flora P. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks // IEEE Transactions on Software Engineering. 2016. Vol. 42, No. 12. P. 1148-1161. DOI: 10.1109/TSE.2016.2553039.
3. Huang Z., Shao Z., Fan G., Yu H., Yang K., Zhou Z. HBSniff: A static analysis tool for Java Hibernate object-relational mapping code smell detection // Science of Computer Programming. 2022. Vol. 217. Article 102778. DOI: 10.1016/j.scico.2022.102778.
4. Loli S., Teixeira L., Cartaxo B. A Catalog of Object-Relational Mapping Code Smells for Java // Brazilian Symposium on Software Engineering. 2020. P. 82-91. DOI: 10.1145/3422392.3422432.
5. Liu W., Chen T.-H. SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications // IEEE Transactions on Software

Engineering. 2023. Vol. 49, No. 6. P. 3376-3390. DOI: 10.1109/TSE.2023.3253700.

6. Torres A., Galante R., Pimenta M. S., Martins A. J. B. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design // Information and Software Technology. 2017. Vol. 82. P. 1-18. DOI: 10.1016/j.infsof.2016.09.009.

7. Yang J., Subramaniam P., Lu S., Yan C., Cheung A. How not to structure your database-backed web applications: a study of performance bugs in the wild // Proceedings of the 40th International Conference on Software Engineering. 2018. P. 800-810.

8. Python Documentation. `ast` — Abstract Syntax Trees [Электронный ресурс]. URL: https://docs.python.org/3/library/ast.html (дата обращения: 04.06.2026).

9. SQLAlchemy Documentation. ORM Querying Guide [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/queryguide/index.html (дата обращения: 04.06.2026).

10. SQLAlchemy Documentation. Performance FAQ [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/faq/performance.html (дата обращения: 04.06.2026).

11. SQLAlchemy Documentation. Relationship Loading Techniques [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/queryguide/relationships.html (дата обращения: 04.06.2026).

12. SQLAlchemy Documentation. Session Basics [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/session_basics.html (дата обращения: 04.06.2026).

13. SQLAlchemy Documentation. Core Events [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/core/event.html (дата обращения: 04.06.2026).
14. FastAPI Documentation. Response Model [Электронный ресурс]. URL: https://fastapi.tiangolo.com/tutorial/response-model/ (дата обращения: 04.06.2026).
15. FastAPI Documentation. Testing [Электронный ресурс]. URL: https://fastapi.tiangolo.com/tutorial/testing/ (дата обращения: 04.06.2026).