Санкт–Петербургский государственный университет

Строганов Никита Сергеевич

Выпускная квалификационная работа

Разработка инструмента для автоматического обнаружения архитектурных антипаттернов в микросервисных системах

Уровень образования: магистратура
Направление 01.04.02 «Прикладная математика и информатика»
Основная образовательная программа ВМ.5889.2023
«Разработка программного обеспечения и науки о данных»
Профиль Архитектура больших систем

Научный руководитель:
доцент, факультет математики и компьютерных наук СПбГУ, к.ф.-м.н., Д.С. Шалымов

Рецензент:
разработчик, ООО «Яндекс.Вертикали Технологии», А.В. Тараскин

Санкт-Петербург
2025 г.

Содержание

Введение

Современные микросервисные системы состоят из десятков и даже сотен компонентов, каждый из которых решает узкоспециализированную задачу и взаимодействует с другими через чётко определённые интерфейсы [1]. Такой архитектурный подход обеспечивает масштабируемость, отказоустойчивость и модульность, однако одновременно с этим он порождает значительные сложности. В частности, становится всё труднее поддерживать целостность системы и соблюдать архитектурные решения, принятые на этапе проектирования. В отличие от ошибок в программной логике, архитектурные нарушения, как правило, не приводят к немедленным сбоям, но накапливаются в виде технического долга. Это снижает сопровождаемость системы, тормозит развитие и в перспективе ведёт к ухудшению качества продукта [3].

Процесс проектирования архитектуры включает в себя множество артефактов: диаграммы зависимостей, схемы взаимодействия, текстовые описания и негласные договорённости между командами. Эти материалы описывают желаемую структуру системы, однако со временем неизбежно устаревают и перестают отражать её фактическое состояние. При этом визуальные схемы или текстовые спецификации по своей природе носят декларативный характер — они позволяют понять, как компоненты связаны в идеале, но не дают ответа на вопрос, какие связи действительно реализованы в коде или конфигурациях. Например, архитектор может задать, что все интеграции с внешним сервисом должны проходить через слой Anti-Corruption Layer (ACL) [5], однако у него нет надёжного способа проконтролировать соблюдение этого правила. Через некоторое время в систему может быть добавлена прямая зависимость от внешнего сервиса, нарушающая принцип изоляции, — и это останется незамеченным.

Подобные сценарии подчёркивают ключевую проблему: наличие формальных архитектурных принципов не гарантирует их соблюдение на практике. Архитектору сложно контролировать реализацию даже собственных решений, особенно когда архитектурные нарушения не приводят к очевидным ошибкам на уровне бизнес-логики. Ещё сложнее — количественно оценить архитектурный техдолг проекта. Между тем, эта метрика может быть крайне

важной как для архитекторов, так и для разработчиков. Например, если в системе насчитывается пять случаев совместного доступа нескольких сервисов к одной и той же базе данных (антипаттерн shared-database [6]), это является объективным показателем архитектурных нарушений. После проведения рефакторинга и устранения части таких связей можно говорить об «измеримом» улучшении архитектуры, опираясь на описанную метрику. Рост этой метрики, напротив, может сигнализировать о нарастающей деградации архитектуры, даже если это пока не отражается на функциональности приложения.

Один из подходов, частично решающий проблему документирования и контроля архитектуры, — это инфраструктура как код (Infrastructure-as-Code, IaC [4]). В рамках него сервисы, базы данных и другие элементы окружения описываются в виде конфигураций, хранимых в системах контроля версий и проходящих те же процедуры ревью, что и программный код. Это позволяет воспроизводимо разворачивать окружение, отслеживать изменения и минимизировать влияние человеческого фактора. Более того, IaC обеспечивает явную связь между кодом и его инфраструктурным контекстом, снижая риск несогласованности. Однако даже в условиях, когда всё окружение формально описано, смысл архитектурных решений и бизнес-ограничений по-прежнему анализируется вручную.

Существующие инструменты, такие как ArchUnit, SonarGraph и некоторые другие, в определённой степени предоставляют поддержку автоматизированной валидации архитектурных ограничений. Они позволяют описывать правила с помощью собственных DSL-языков и применять проверки на соответствие этим правилам [7]. Однако на практике эти решения нередко оказываются слишком ограниченными или сложными в интеграции. Высокий порог входа, необходимость точной настройки и ограниченная выразительность часто делают их неудобными для использования во времена быстрой эволюции микросервисных систем — особенно при наличии особенностей окружения, специфичных для конкретной организации. Более того, подавляющее большинство этих инструментов не предоставляют агрегированного представления архитектурного состояния, не поддерживают визуализацию отклонений и почти не используют числовые метрики, которые могли бы

помочь в анализе архитектурного долга и его динамики во времени.

Таким образом, существует потребность в инструменте, который бы позволял явно описывать архитектурные принципы системы и проверять их соблюдение автоматически. При этом важно, чтобы такой инструмент был удобен для использования в процессе разработки: позволял быстро формулировать правила любой сложности, удобно запускался и органично интегрировался в инфраструктуру компании.

Данная работа направлена на анализ обозначенной проблемы и разработку подхода, позволяющего автоматически выявлять архитектурные антипаттерны в микросервисных системах. Основной акцент делается на практическую применимость, гибкость расширения набора проверок и удобство использования инструмента в рамках жизненного цикла разработки. Результатом станет прототип системы, способной на основе инфраструктурных конфигураций и связей между сервисами выявлять потенциальные архитектурные нарушения и тем самым способствовать контролю качества архитектуры в условиях её неизбежной эволюции.

Практическая реализация и апробация предлагаемого в данной работе инструмента будут проводиться в инфраструктуре Яндекс.Вертикалей. Это означает, что разрабатываемое решение будет изначально ориентировано на интеграцию с существующими в компании системами управления конфигурациями и развертывания. Такой подход позволит проверить инструмент на реальных задачах и сфокусироваться на решении архитектурных проблем, наиболее актуальных для компании, с учетом специфики технологического стека.

Постановка задачи

Целью данной работы является разработка инструмента, который бы позволил автоматически проверять архитектуру микросервисной системы на соответствие заложенным в неё принципам. Для достижения цели были сформулированы следующие задачи.

- Провести обзор предметной области и подходов существующих анализаторов с целью выявления их достоинств и недостатков.
- Разработать инструмент, позволяющий описывать правила любой сложности и проверять их выполнение применимо к конкретной архитектуре.
- Придумать простой способ написания новых правил для шаблонных ситуаций.
- Предоставить правила, проверяющие соблюдение стандартных архитектурных антипаттернов, и заложить их в инструмент по умолчанию.
- Проинтегрировать решение с используемыми в Яндекс.Вертикалях системами и инструментами разработки.
- Измерить эффективность и полезность инструмента на реальных микросервисных системах, а также проанализировать полученные результаты.

1. Обзор предметной области

Одним из ключевых аспектов, влияющих на долгосрочную жизнеспособность и эффективность микросервисных систем, является качество их архитектуры. Принятые на этапе проектирования решения определяют, насколько система будет гибкой, масштабируемой, отказоустойчивой и простой в сопровождении. Однако, как было отмечено во введении, существует значительный разрыв между декларируемыми архитектурными принципами и их фактической реализацией. Этот разрыв часто приводит к появлению так называемых архитектурных антипаттернов.

1.1. Архитектурные антипаттерны

Архитектурные антипаттерны — это неэффективные или неудачные решения повторяющихся проблем проектирования программных систем. Подобно тому, как «паттерны проектирования» предлагают проверенные временем решения для типичных задач [2], антипаттерны описывают распространённые ошибки и «ловушки», в которые легко попасть при разработке. Возникновение антипаттернов часто является следствием неполного понимания проблемы, ограниченности ресурсов, давления сроков, недостаточной коммуникации между командами или постепенной эрозии первоначального замысла под влиянием изменений и доработок. Антипаттерны могут приводить к увеличению сложности, снижению производительности железа или к затруднению дальнейшего развития системы [8].

Для более глубокого понимания проблемы рассмотрим буквально два известных примера архитектурных антипаттернов.

1.1.1 Shared Database

Одним из наиболее известных и часто встречаемых антипаттернов в микросервисных системах является «Общая база данных» (Shared Database [9]). Он возникает, когда несколько независимых микросервисов напрямую обращаются к одной и той же физической базе данных, используя общие таблицы или даже всю схему данных. На первый взгляд, такое решение может

показаться удобным, особенно на ранних этапах разработки или при миграции от монолитной архитектуры, поскольку оно упрощает обмен данными и позволяет избежать дублирования. Однако в долгосрочной перспективе общая база данных становится источником множества проблем.

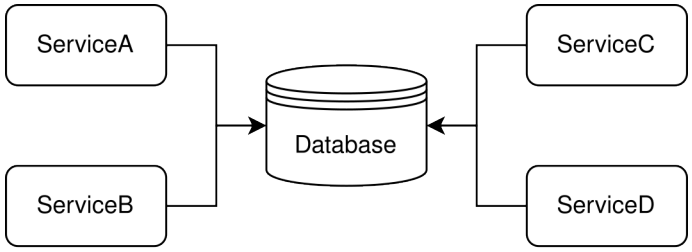

Рис. 1: Пример антипаттерна Shared Database.

Во-первых, она создаёт сильную связанность между сервисами на уровне данных. Любое изменение схемы базы данных, инициированное одним сервисом (например, добавление нового поля, изменение типа данных или удаление таблицы), может немедленно повлиять на работоспособность других сервисов, использующих эту же схему. Это приводит к тому, что команды разработки становятся зависимыми друг от друга, а процесс внесения изменений усложняется и требует тщательной координации, что противоречит идее автономности микросервисов. Во-вторых, общая база данных затрудняет независимое масштабирование сервисов. Если один из сервисов создаёт высокую нагрузку на базу данных, это неизбежно скажется на производительности всех остальных сервисов, которые от неё зависят. База данных становится «узким горлышком» и единой точкой отказа для целой группы сервисов. В-третьих, нарушается принцип инкапсуляции данных. Каждый микросервис должен владеть своими данными и предоставлять доступ к ним исключительно через чётко определённый API [10]. Прямой доступ к данным другого сервиса в обход его API нарушает этот принцип, делая систему более хрупкой и сложной для понимания.

1.1.2 Lava Flow

Следующий пример антипаттерна, иллюстрирующий сложности поддержания архитектурной целостности, – это «Поток лавы» (Lava Flow [11]).

Этот антипаттерн описывает ситуацию, когда в системе накапливаются устаревшие компоненты или код, которые никто не решается удалить из-за страха нарушить работоспособность системы. Оставаясь в системе, они затрудняют её понимание, сопровождение и развитие. «Поток лавы» может проявляться в виде мёртвого кода, неиспользуемых конфигурационных параметров или даже целых сервисов и баз данных, которые когда-то были нужны, но со временем их функциональность была перенесена или стала ненужной. Проблема заключается в том, что наличие таких артефактов увеличивает когнитивную нагрузку на разработчиков и архитекторов. Приходится тратить время на анализ этих элементов, чтобы понять, нужны ли они ещё, и как они взаимодействуют (или не взаимодействуют) с остальной системой. Это замедляет разработку новых функций и увеличивает риск внесения ошибок. Сложность борьбы с описанным антипаттерном заключается в отсутствии точной информации о реальном использовании тех или иных частей системы. Документация часто отстаёт от реального положения дел, а разработчики, которые изначально создавали эти компоненты, могли уже покинуть проект. В результате, решение об удалении чего-либо принимается с большой осторожностью, и часто проще оставить всё как есть, усугубляя проблему.

1.2. Существующие инструменты

Ключевая проблема, которую подчёркивают архитектурные антипаттерны, заключается в том, что их возникновение часто является постепенным и незаметным процессом. В больших, динамично развивающихся микросервисных системах архитектору чрезвычайно сложно отслеживать соблюдение всех заложенных принципов вручную. Архитектурные диаграммы и описания показывают желаемое состояние, но не гарантируют его реализацию. Эта ситуация породила спрос на инструменты, способные автоматически анализировать фактическую структуру системы и выявлять отклонения от заданных архитектурных правил. Рассмотрим некоторые из существующих подходов и инструментов.

1.2.1 ArchUnit

ArchUnit представляет собой бесплатную библиотеку с открытым исходным кодом для языка Java, предназначенную для проверки архитектурных правил непосредственно в коде [12]. Она позволяет разработчикам писать модульные тесты, которые анализируют байт-код приложения и проверяют его на соответствие заданным ограничениям. Правила описываются с помощью гибкого и выразительного предметно-ориентированного языка (DSL).

Например, можно легко определить правило, запрещающее классам из доменного слоя (`..domain..`) зависеть от классов слоя представления (`..web..`), или более сложные правила, описывающие слоистую архитектуру и разрешённые зависимости между ними:

```
1  class ArchitectureTest {
2      private final ClassFileImporter importer =
3              new ClassFileImporter();
4      private final JavaClasses importedClasses =
5              importer.importPackages("com.example");
6
7      @Test
8      void domainModelDoesNotDependOnUpperLayers() {
9          ArchRule rule = noClasses()
10                 .that().resideInAPackage("..domain..")
11                 .should().dependOnClassesThat()
12                 .resideInAnyPackage("..web..");
13
14         rule.check(importedClasses);
15     }
16 }
```

Сильной стороной ArchUnit является его интеграция в процесс непрерывной интеграции (CI/CD), где архитектурные тесты выполняются наряду с другими автоматизированными тестами, предотвращая попадание нарушений в основную кодовую базу. Однако ArchUnit ориентирован исключительно на анализ архитектуры внутри одного Java-приложения (монолита или отдельного микросервиса), и не предназначен для анализа межсервисных взаимодействий.

1.2.2 SonarGraph и Structure101

Инструменты SonarGraph (часть коммерческой платформы SonarQube [13]) и Structure101 [14], фокусируются на визуализации и анализе зависимостей на более высоком уровне абстракции – между модулями или компонентами системы.

SonarGraph позволяет архитекторам определять эталонную архитектуру, описывая компоненты системы и разрешённые зависимости между ними. Затем, в процессе анализа кода, SonarGraph сопоставляет фактические зависимости с этой моделью и сообщает о нарушениях. Например, архитектор может задать правило, что модуль `ServiceA` не должен напрямую обращаться к модулю `ServiceB`, а только через общий модуль `CoreServices`. При обнаружении прямого вызова SonarGraph отметит это как нарушение. Инструмент интегрируется в SonarQube и помогает отслеживать соответствие архитектурным гайдлайнам в динамике.

Structure101 предоставляет средства для визуализации архитектуры. Он строит диаграммы зависимостей, которые помогают выявлять запутанные узлы системы, циклические зависимости и чрезмерно крупные компоненты. Structure101 использует метрики для количественной оценки степени нарушения архитектурных принципов (например, принципа единой ответственности на уровне компонентов). На основе этих данных разработчики могут анализировать архитектуру и принимать решения о рефакторинге. Однако никакой автоматизации в этом месте инструмент не предоставляет.

Эти инструменты полезны для получения общего представления об архитектуре. Однако их практическое применение может быть сопряжено с трудностями. Создание и поддержка актуальной архитектурной модели для сложной, быстро меняющейся микросервисной системы требует значительных усилий. Встроенные языки для описания правил могут оказаться недостаточно гибкими для формализации всех специфических требований проекта. Кроме того, основной фокус этих инструментов – анализ кода, и им может не хватать информации о связях, определённых на уровне инфраструктуры.

1.2.3 Концепция тестов на архитектуру

Наиболее близким по идеологии к задачам данной работы является подход, представленный в докладе «Раз архитектура — "as Code", почему бы её не покрыть тестами?!» на конференции ArchDays 2023 [15].

Центральная идея этого подхода заключается в активном использовании артефактов, создаваемых в рамках практик «Infrastructure-as-Code» (IaC) и «Architecture-as-Code» (AaC). Если инфраструктура системы (сервисы, базы данных, сетевые настройки) описывается в виде кода (например, с помощью Terraform [18], Kubernetes manifests [16], Ansible [17] и так далее), а архитектурные диаграммы и модели также хранятся в текстовом, машиночитаемом формате (PlantUML [19], Mermaid [20]), то появляется возможность для их автоматического сопоставления и анализа. Примеры IaC-конфигурации и AaC-диаграмы приведены в листингах 1 и 2.

```
resource "docker_container" "service_b" {
  name  = "service-b"
  image = "myregistry/service-b:latest"
  ports {
    internal = 8081
    external = 8081
  }
  env = [
    "DATABASE_URL=postgresql://database-b:5432/mydb"
  ]
}
```

Listing 1: Пример простой Terraform-конфигурации.

Концепция «Архитектура как тесты» предлагает решать две фундаментальные проблемы, свойственные традиционному управлению архитектурой. Во-первых, это проблема устаревания архитектурной документации. В динамично развивающихся системах любые изменения в реальной конфигурации сервисов или их взаимодействий должны немедленно отражаться в архитектурных схемах. В противном случае документация быстро теряет свою актуальность и перестаёт быть надёжным источником информации. Пред-

```
@startuml C4ContainerDiagram

Container_Boundary(boundary, "System") {
    Container(serviceA, "ServiceA")
    Container(serviceB, "ServiceB")
    ContainerDb(databaseB, "DatabaseB")
}

Rel(serviceA, serviceB, "Calls", "HTTP/REST")
Rel(serviceB, databaseB, "Reads/Writes", "JDBC")

@enduml
```

Listing 2: Пример PlantUML-диаграммы в формате C4 Model.

лагаемый подход позволяет автоматически сравнивать реальную конфигурацию, извлечённую из IaC-артефактов, с задокументированной архитектурой из AaC-источников. Любое расхождение выявляется при прогоне тестов в CI/CD пайплайне, сигнализируя о необходимости либо обновить документацию, либо откатить несогласованное изменение в инфраструктуре. Таким образом, обеспечивается синхронизация реального состояния системы с её описанием.

Во-вторых, решается проблема декларативности и неявности архитектурных ограничений в схемах. Сами по себе диаграммы часто показывают лишь существующие связи, но не отвечают на вопрос, какие связи запрещены или какие архитектурные принципы должны соблюдаться. Например, если на диаграмме `ServiceA` не взаимодействует напрямую с базой данных сервиса `ServiceB`, то неясно, можно ли в будущем добавить такую связь или нельзя. Архитектура в виде картинки об этом ничего не говорит (рис. 2).

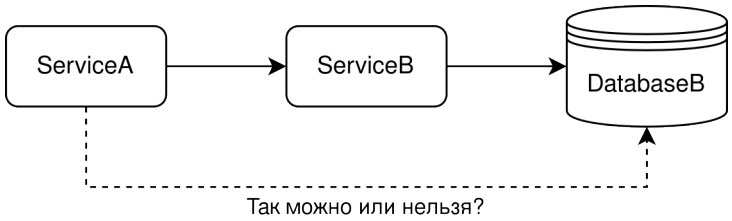

Рис. 2: Декларативность архитектурных диаграмм.

Подход «Архитектура как тесты» позволяет формализовать эти неявные правила и принципы в виде исполняемых тестов. Эти тесты оперируют графом зависимостей, построенным на основе архитектурных диаграмм AaC. Таким образом, становится возможным валидировать архитектурные решения, заложенные именно в этих диаграммах. Например, можно написать тест, проверяющий, что ни один сервис, кроме `ServiceB`, не зависит от `DatabaseB`, и этот тест будет выполняться на модели, извлечённой из PlantUML-описания.

Прототип инструмента, предлагаемый в рамках этой концепции, включает автоматический парсинг IaC-конфигураций для получения графа реальных связей и AaC-диаграмм для получения графа ожидаемых или эталонных связей. Далее, с помощью набора обычных unit-тестов, производится проверка различных свойств этих графов. Это могут быть тесты на полное соответствие двух графов, тесты на отсутствие определённых типов связей в реальной архитектуре, если они не разрешены в эталонной, или тесты, проверяющие соблюдение высокоуровневых архитектурных принципов.

Для лучшего понимания подхода рассмотрим пример теста, который проверяется, что только сервисы, помеченные тегом `"acl"`, могут иметь зависимости от внешних систем:

```
1  it("only acl can depence from external systems", () => {
2    let violationsCount = 0;
3
4    for (const container of containersFromPuml) {
5      const isAcl = container.tags?.includes("acl");
6      const externalRelations = container.relations.filter(
7        (relation) => relation.to.type === ExternalType
8      );
9
10     if (!isAcl && externalRelations.length > 0) {
11       violationsCount += externalRelations.length;
12     }
13   }
14
15   expect(violationsCount).toBe(0);
16 });
```

Интеграция таких тестов в CI/CD-конвейер позволяет превратить архитектурные принципы из деклараций в исполняемые спецификации. Любое изменение, нарушающее эти принципы или приводящее к рассинхронизации с документацией, будет автоматически обнаружено. Единственное серьёзное ограничение данного подхода — наличие внедренной в компании практики Architecture-as-Code. Также важно отметить, что рассмотренный инструмент находится на стадии прототипа, то есть скорее демонстрирует интересную идею, чем предоставляет готовое к использованию решение.

1.2.4 Выводы

Проведённый обзор существующих решений для контроля архитектуры показывает, что многие из них имеют значительные недостатки или недоработки, препятствующие их практическому применению на реальных микросервисных системах. Так ArchUnit ориентирован на анализ кода только внутри одного сервиса. SonarGraph и Structure101 не предоставляют никакой автоматизации для проверки соблюдения принципов. Концепция «Архитектура как тесты» пока что находится на стадии прототипа, а также требует наличие в проекте архитектурных диаграмм в виде кода, что на самом деле является серьезным ограничением. Также стоит отметить, что все рассмотренные подходы требуют усилий по интеграции с корпоративными решениями, инструментами разработки и инфраструктурой.

1.3. Инфраструктура как код

Концепция Infrastructure-as-Code (IaC) представляет собой подход к управлению и предоставлению вычислительной инфраструктуры (серверов, сетей, хранилищ, балансировщиков нагрузки, баз данных и других компонентов) через машиночитаемые файлы определений, а не посредством ручной настройки или интерактивных инструментов конфигурирования. Эти файлы, вместе с исходным кодом самих сервисов, хранятся в системе контроля версий, проходят через процессы ревью и автоматизированного развёртывания. Такой подход существенно упрощает внедрение и сопровождения IT-инфраструктуры, особенно в контексте облачных вычислений и микросервисных архитектур.

Основными принципами IaC являются автоматизация, воспроизводимость и идемпотентность [21]. Автоматизация достигается за счёт использования специализированных инструментов, которые интерпретируют файлы определений и приводят инфраструктуру в желаемое состояние. Воспроизводимость означает, что одна и та же конфигурация, применённая в различных окружениях, будет создавать идентичные или предсказуемо схожие инфраструктурные ландшафты. Идемпотентность гарантирует, что повторное применение одной и той же конфигурации не приведёт к нежелательным изменениям, если система уже находится в целевом состоянии.

Существует множество инструментов, реализующих подход IaC: AWS CloudFormation, Terraform, Azure Resource Manager templates, Google Cloud Deployment Manager, Ansible, Puppet, Chef и многие другие. Эти инструменты позволяют описывать желаемое состояние инфраструктуры, используя либо декларативный (описание конечного результата), либо императивный (описание последовательности шагов для достижения результата) стиль. Например, в Terraform разработчик описывает ресурсы (виртуальные машины, базы данных, сетевые правила) и их взаимосвязи в текстовых файлах `.tf`. Terraform затем вычисляет план действий для создания или изменения инфраструктуры до соответствия этому описанию. Аналогично, манифесты Kubernetes в формате YAML или JSON описывают развёртывания, сервисы, конфигурационные карты и другие сущности, необходимые для запуска приложений в

контейнерах.

Таким образом, IaC-артефакты представляют собой формализованное и машиночитаемое описание инфраструктуры, где определяются структурные зависимости и связи между компонентами системы:

- Определения взаимосвязей ресурсов. IaC-инструменты описывают не только сами компоненты, но и то, как они связаны. Например, Terraform-конфигурация может явно указывать, что определённый микросервис использует конкретный экземпляр базы данных, передавая его адрес через переменные окружения.
- Service Mesh. инструменты типа Istio или Linkerd, управляемые через IaC-манифесты, содержат детальные политики взаимодействия между сервисами [22].
- Сетевые конфигурации. Правила файрволов, группы безопасности и так далее явно разрешают или запрещают коммуникацию между определёнными IP-адресами или группами ресурсов.

Анализ этих конфигурационных файлов позволяет построить граф реальных зависимостей системы. Этот граф отражает, как сервисы связаны друг с другом, какие общие ресурсы они используют, и какие коммуникационные пути между ними фактически настроены. Полученная модель может быть использована для анализа, направленного на выявление архитектурных антипаттернов. Правда, здесь необходимо отметить, что IaC не способен составить архитектурную модель верную на 100%. В частности, динамические связи, устанавливаемые непосредственно в коде приложений во время выполнения, могут оставаться невидимыми. Однако такой подход сам по себе является в некотором роде «антипаттерном», и обычно разработчики стараются выносить зависимости между сервисами на уровень конфигурации.

1.4. Система деплоя Shiva

В контексте разработки инструмента для автоматического обнаружения архитектурных антипаттернов в Яндекс.Вертикалях, ключевым источником

информации о фактической структуре системы и связях между её компонентами выступает внутренняя система управления деплоями — Shiva. Shiva представляет собой инфраструктурный сервис, который абстрагирует разработчиков от деталей оркестрации контейнеров и механизмов service discovery, предоставляя унифицированный интерфейс для развёртывания и управления жизненным циклом микросервисов.

Центральным артефактом, с которым работает Shiva, является «карта сервиса» (service map). Это декларативное описание сервиса в формате YAML, содержащее всю необходимую метаинформацию. Карта сервиса включает в себя:

- `name`: Уникальное имя сервиса.
- `type`: Тип компонента (например, `service`).
- `owners`: Список ответственных за сервис команд.
- `provides`: Определяет интерфейсы, которые сервис предоставляет другим компонентам. Каждый такой интерфейс описывается именем, протоколом, портом и некоторыми другим опциональными параметрами.
- `depends_on`: Список сервисов, от которых зависит данный компонент. Для каждой зависимости указывается имя целевого сервиса (`service_name`) и имя используемого интерфейса (`interface_name`). На данный момент секция не является обязательной, поэтому для большинства сервисов все их реальные зависимости не будут отражены в этом списке.

А также многие другие поля, менее важные в контексте составления графа зависимостей. Ниже приведен простейший пример карты сервиса в Shiva:

```yaml
name: service-a
type: service
provides:
- name: grpc-api
  protocol: grpc
  port: 80
depends_on:
- service_name: service-b
  interface_name: http-api
```

На основе карты сервиса Shiva подготавливает инфраструктуру для контейнеров, управляет их ресурсами и жизненным циклом. Процесс деплоя также конфигурируется декларативно с помощью манифеста деплоя, который является ещё одним YAML-файлом. Манифест деплоя определяет параметры развёртывания для различных окружений (prod и test), включая:

- `datacenters`: Количество инстансов сервиса в каждом дата-центре.
- `resources`: Запросы к ресурсам (CPU, память).
- `config`: Специфичные для окружения конфигурации в виде переменных окружения, которые будут установлены в контейнере.

А также некоторые другие поля. Для лучшего понимания ниже приведен пример части манифеста деплоя сервиса в Shiva:

```yaml
prod:
  resources:
    cpu: 1000
    memory: 3072
  config:
    params:
      - SERVICE_B_URL: service-b-grpc-api.yandex.net
      - DB_URL: jdbc:postgresql://...
      - DB_USER: ...
```

Для выявления актуальных зависимостей сервиса, помимо `depends_on` в карте сервиса, нужно смотреть в переменные окружения, определяемые в манифесте деплоя. Именно через них сервисы часто получают информацию

о местонахождении других сервисов или баз данных. Например, переменная `SERVICE_B_URL: service-b-grpc-api.yandex.net` указывает, что данный сервис будет обращаться к сервису `service-b` по его grpc-api интерфейсу. Аналогично, переменные вида `DB_URL: jdbc:postgresql://...` явно указывают на используемые экземпляры баз данных.

Таким образом, карты сервисов и манифесты деплоя в системе Shiva представляют собой богатый источник структурированной информации об архитектуре системы. Анализируя эти артефакты, можно с высокой точностью восстановить граф зависимостей между микросервисами, а также их связи с базами данных и другими инфраструктурными компонентами. Именно эти данные и предполагается использовать в этой работе.

2. Описание инструмента

Целью данной работы является создание инструмента для автоматического обнаружения архитектурных антипаттернов и других нарушений заданных принципов в микросервисных системах. Основная задача инструмента — предоставить архитекторам и разработчикам удобное и гибкое средство для контроля за состоянием архитектуры и количественной оценки технического долга. Инструмент позволяет описывать правила любой сложности и органично интегрируется с существующей инфраструктурой Яндекс.Вертикалей. В данном разделе будет подробно описано его устройство, а также предоставляемое пользователям API.

2.1. Верхнеуровневая архитектура

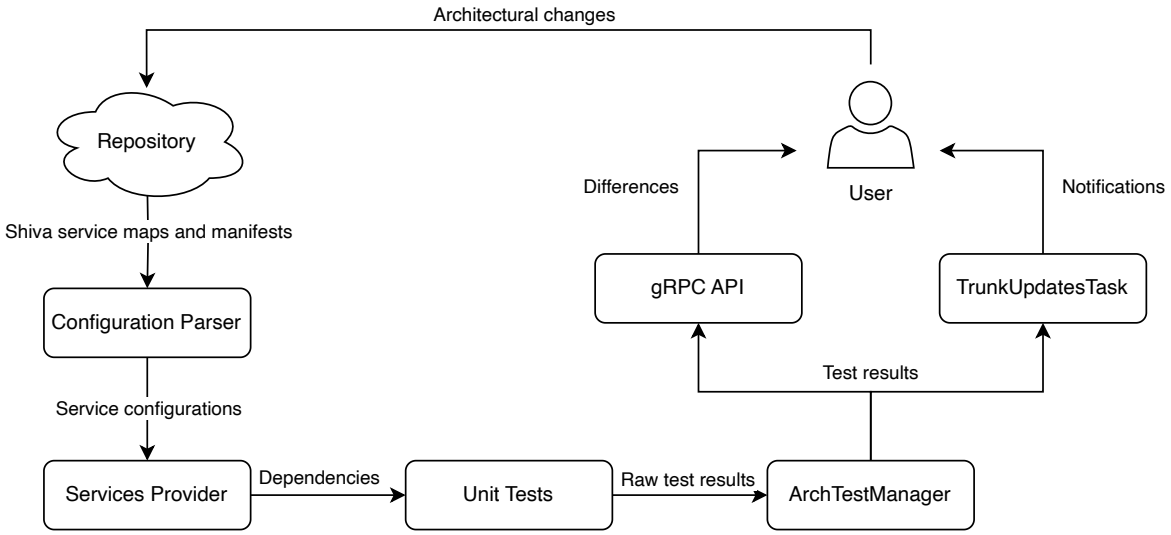

Рис. 3: Ключевые компоненты и их взаимодействие.

Основные компоненты приложения, а также их взаимодействие между собой и с пользователем представлено на рисунке 3. Таким образом, приложение состоит из нескольких модулей:

- **Модуль загрузки и парсинга конфигураций**. Отвечает за получение данных об архитектуре анализируемой системы.
- **Архитектурные правила**. Могут быть заданы двумя способами: в виде unit-тестов на языке Scala или декларативно в формате YAML.

- **Модуль выполнения тестов**. Управляет процессом запуска архитектурных проверок и выставляет программное API для получения агрегированных результатов прогона тестов.
- **gRPC API** [23]. Предоставляет удобные методы для интеграции инструмента с CI/CD и автоматической проверки изменений в пулл-реквестах.
- **TrunkUpdatesTask**. Отслеживает изменения в основной ветке разработки (trunk [24]) и отправляет уведомления в Telegram при обнаружении новых архитектурных проблем.

В следующих разделах будет подробно рассмотрен каждый из перечисленных компонентов.

2.2. Загрузка и парсинг конфигураций

Источником данных для тестов служат конфигурационные файлы системы развертывания Shiva: карты сервисов и манифесты деплоя. Инструмент умеет получать эти файлы либо из системы контроля версий Arcadia [25] для указанной ревизии (коммита или ветки), либо из локальной файловой системы. Это позволяет анализировать как любой заданный срез архитектуры в прошлом, так и текущее локальное состояние. Загруженные YAML-файлы проходят через стадию парсинга и преобразуются во внутреннюю модель системы. Эта модель представляет собой граф, узлами которого являются сущности типа `Service`, а рёбрами — зависимости между ними. Каждый объект `Service` содержит информацию об имени сервиса, его типе, предоставляемых интерфейсах и зависимостях: как явных, указанных в `depends_on`, так и неявных, содержащихся в переменных окружения из конфигурации. Помимо этих полей в `Service` содержится и другая метаинформация о сервисе, которую предоставляет Shiva.

2.3. Написание unit-тестов

Основным и наиболее гибким способом описания архитектурных правил является создание unit-тестов на языке Scala с использованием фреймворка ScalaTest [26]. Для этого разработчикам предоставляется специальный

трейт `ArchTest`, который необходимо унаследовать в классе с тестами. Этот трейт обеспечивает доступ к загруженной и обработанной модели архитектуры, а также предоставляет вспомогательные методы для написания проверок. Как уже упоминалось, при локальном запуске тестов будет использована текущая локальная конфигурация сервисов Shiva.

Вместо построения явного графа зависимостей, здесь используется несколько другой подход: `ArchTest` предоставляет список всех сервисов `services: Seq[Service]`, внутри каждого из которых содержится вся информация о его зависимостях. Чтобы проверить наличие зависимости между сервисами `A` и `B`, нужно вызвать метод `hasDependencyOn` у `A`. Метод просканирует секцию `depends_on` и все переменные окружения сервиса `A` на упоминание сервиса `B`. Если оно будет найдено, то метод возвращает `true`. Более формально это отражено в алгоритме 1.

Algorithm 1: Проверка зависимости от другого сервиса

hasDependencyOn $(this, other)$
- **Data:** $this$ — текущий сервис со своими зависимостями
- **Data:** $other$ — другой сервис для проверки
- $inDependsOn \leftarrow$ false;
- **foreach** $dependency \in this.dependsOn$ **do**
  - **if** *matches*$(other, dependency)$ **then**
    - $inDependsOn \leftarrow$ true;
    - **break**;
  - **end**
- **end**
- $inConfig \leftarrow$ false;
- **foreach** $envVariable \in this.config.envVariables$ **do**
  - **if** *matches*$(other, envVariable)$ **then**
    - $inConfig \leftarrow$ true;
    - **break**;
  - **end**
- **end**
- **return** $inDependsOn \vee inConfig$;

Алгоритм использует функцию `matches`, чтобы проверить совпадение сервиса `other` с очередным сервисом из `depends_on` или с очередной переменной окружения `envVariable`. Подробнее про внутреннее устройство этой

функции будет написано в одном из следующих разделов, а пока что можно воспринимать её работу «естественно»: сервисы могут совпасть по имени, по URL и так далее.

Помимо основного `hasDependencyOn`, библиотека предоставляет другие полезные функции для разных сценариев использования:

- `hasDependencyInServiceMap`: Проверяет только явные зависимости из `depends_on`.
- `hasDependencyInConfig`: Проверяет только зависимости, выведенные из переменных окружения (например, URL другого сервиса или строки подключения к БД).
- `hasDependencyOn(other: ServicePattern)`: Работает аналогично основному методу, но принимает не существующий в системе `Service`, а описанный пользователем `ServicePattern`. Паттерн позволяет гибко задавать критерии поиска целевого сервиса (по имени, типу, хосту, IP-адресу и так далее).

Для иллюстрации рассмотрим простой пример теста, который проверяет, что сервис `payment-service` не должен напрямую зависеть от базы данных пользователей `user-db` (допустим, что такие компоненты есть в нашей системе).

```
1 "payment-service" should "not access user-db" in {
2   val paymentService = Services.service("payment-service")
3   val userDb = Services.mysql("user-db")
4
5   paymentService.hasDependencyOn(userDb) shouldBe false
6 }
```

`Services.service("payment-service")` является удобным способом получить экземпляр `Service` для сервиса с именем "payment-service". Для базы данных `user-db` используется аналогичный метод `Services.mysql`.

Другой пример: проверка, что только `orders-proxy` может зависеть от внешней системы `orders.com`.

```
"orders.com" should "only be accessed by orders-proxy" in {
  val ordersProxy = Services.service("orders-proxy")
  val ordersCom = ServicePattern.withHost("orders.com")

  val ordersComDependents = services.filter { service =>
    service.hasDependencyOn(ordersCom)
  }

  ordersComDependents shouldBe Seq(ordersProxy)
}
```

Здесь мы итерируемся по всем сервисам и проверяем условие, что только прокси имееет прямую зависимость на внешнюю систему.

2.4. Задание проверок в YAML-конфиге

Для простых и типовых архитектурных правил инструмент позволяет описывать их в декларативном стиле с помощью YAML-файлов [27]. Это может быть удобно для для быстрого добавления стандартных проверок без написания Scala-кода.

Каждый YAML-файл может содержать список правил. Структура правила определяется следующими полями:

- `rule`: Человекочитаемое описание правила.
- `service`: Определяет сервис или группу сервисов, к которым применяется правило. Содержит обязательное поле `name`, значение которого интерпретируется как регулярное выражение для полного имени сервиса (например, `mysql/user-db` или `payment-service`). Структура задумывалась расширяемой, чтобы в дальнейшем можно было задавать сервис не только по имени, а, например, по владельцу или другой метаинформации.
- `dependencies` (опционально): Определяет ограничения на исходящие зависимости проверяемого сервиса. Может содержать либо `whitelist` (список разрешенных сервисов-зависимостей), либо `blacklist` (список запрещенных сервисов-зависимостей). Элементы в `whitelist` и

`blacklist` являются «сервисами», то есть могут быть также заданы по имени с помощью регулярного выражения.

- `dependents` (опционально): Определяет ограничения на входящие зависимости (сервисы, которые зависят от проверяемого сервиса). Структура аналогична `dependencies`.

Важно, что в рамках одного блока (`dependencies` или `dependents`) нельзя одновременно использовать `whitelist` и `blacklist`, так как это не имеет смысла.

Для иллюстрации возможностей правил в YAML, перепишем unit-тест про `payment-service` из предыдущего раздела декларативно.

```yaml
- rule: "payment-service should not access user-db"
  service:
    name: mysql/user-db
  dependents:
    blacklist:
      - name: payment-service
```

Другой пример: сервисы catalog-api и catalog-tasks не должны ни от кого зависеть.

```yaml
- rule: "catalog should not depend on other services"
  service:
    name: catalog-(api|tasks)
  dependencies:
    whitelist: []
```

Разбор и выполнение таких YAML-правил осуществляется специальным ScalaTest-тестом `YamlSpec`. Он находит все `*.yml` файлы в заданной директории, парсит их и для каждого правила из файла динамически генерирует соответствующие проверки во время выполнения тестов.

2.5. Запуск тестов в рантайме

Библиотека для написания архитектурных тестов разработана таким образом, чтобы их можно было запускать не только локально из IDE, но

и программно, например, по запросу или периодически. Эту функциональность обеспечивает `ArchTestManager`. Когда `ArchTestManager` получает запрос на запуск тестов для определённой ревизии (коммита или имени ветки), ему необходимо каким-то образом передать в тесты нужную версию инфраструктурных схем. Для этого он устанавливает системное свойство JVM `archtest.run_mode`. Значением этого свойства является JSON-строка, представляющая объект `RunMode`, например,

```
{
    "mode": "Branch",
    "revision": "my-feature-branch",
    "arc_config": {
        "token": "<ARC_TOKEN>",
        "path": "/opt/homebrew/bin/arc"
    }
}
```

Использование `System.setProperty` для передачи параметров запуска является ключевым моментом. Стандартный механизм `ConfigMap` из ScalaTest здесь не подходит, поскольку он становится доступен уже после обнаружения и инициализации тестовых классов. В нашем же случае, конфигурация (режим запуска, ревизия) нужна на самом раннем этапе – до того, как ScalaTest начнет исполнять тесты, и даже до того как он полностью определит их список. Это связано с тем, что сама структура архитектуры (и, как следствие, набор сервисов, для которых могут генерироваться тесты) зависит от выбранной ревизии.

После установки системного свойства, `ArchTestManager` запускает стандартный `scalatest.Runner`. Он обнаруживает все скомпилированные тестовые классы, и при инстанцировании каждого такого класса, благодаря подмешанному трейту `ArchTest`, происходит следующее:

1. Читается системное свойство `archtest.run_mode`.

2. На основе этого свойства выбирается и конфигурируется соответствующий провайдер данных (например, `ServicesProviderArcadia` для режима `Branch`).

3. Выбранный провайдер загружает конфигурационные файлы Shiva для заданной ревизии (из Arcadia или локально).

4. Загруженные файлы парсятся, и строится полная внутренняя модель системы в виде последовательности объектов `Seq[Service]`.

5. Эта модель становится доступной внутри тестового класса через поле `lazy val services: Seq[Service]`, и уже на ее основе выполняются проверки.

Результаты выполнения тестов (успех/неудача, сообщения об ошибках и т.д.) собираются с помощью кастомной реализации `scalatest.Reporter`. Он подписывается на события `TestSucceeded` и `TestFailed`, которые генерирует ScalaTest во время выполнения тестов, и собирает информацию о результатах прогона. В итоге они возвращаются в виде структурированных объектов `RunTestResult` в вызывающий код.

2.6. Валидация изменений в пулл реквесте

Одна из основных задач инструмента — предотвращение попадания нежелательных архитектурных изменений в основную ветку разработки (trunk). Для этого инструмент предоставляет возможность сравнить результаты архитектурных тестов между текущей веткой разработки (например, веткой пулл-реквеста) и её точкой расхождения с trunk, так называемой merge-base. Эта логика реализована в `ArchTestManager` и доступна через gRPC-метод `compareTestsWithTrunk`.

При вызове этого метода с указанием имени проверяемой ветки, gRPC-сервис определяет merge-base и дважды запускает архитектурные тесты: для указанной ветки и для merge-base. После этого результаты двух запусков сравниваются, и выделяются различия: новые упавшие тесты (потенциальные проблемы), исправленные тесты (улучшения), а также добавленные или удаленные тесты. Эта информация возвращается через gRPC и может быть использована CI/CD системой для автоматического комментирования пулл-реквеста, информируя разработчиков и ревьюеров о влиянии предлагаемых изменений на архитектуру.

На данный момент, непосредственная интеграция с интерфейсом систем управления версиями (в случая Яндекса — это Arcanum) для публикации комментариев в пулл-реквестах не реализована, так как было принято решение сначала провести апробацию инструмента и оценку его эффективности в более узком кругу разработчиков. Однако весь необходимый функционал на стороне бэкенда для такой интеграции уже существует и запущен в облаке.

2.7. Уведомления в Telegram

В качестве основного способа обратной связи на текущем этапе и для поэтапного внедрения инструмента была выбрана система уведомлений об архитектурных изменениях в основной ветке (trunk) через Telegram. Это решение является временной альтернативой полноценной интеграции проверок в процесс код-ревью, позволяя избежать немедленной раскатки новой функциональности на всех разработчиков компании (которые неизбежно столкнулись бы с инструментом при работе с пулл-реквестами). Такой подход даёт возможность узкому кругу заинтересованных лиц апробировать инструмент, собрать обратную связь и оценить его пользу в реальных условиях без широкого анонсирования. Реализацию этого непрерывного мониторинга обеспечивает фоновая задача `TrunkUpdatesTask`. Важно отметить, что даже после полного запуска валидации в пулл-реквестах, `TrunkUpdatesTask` сохранит свою актуальность: она будет служить дополнительным уровнем контроля для архитекторов, информируя их о случаях, когда архитектурные нарушения все же могли быть пропущены или сознательно внесены в основную ветку разработки.

`TrunkUpdatesTask` с настраиваемой периодичностью запрашивает историю коммитов в репозиторий с инфраструктурными схемами Shiva. Для каждой пары последовательных коммитов (старый и новый) запускается сравнение результатов архитектурных тестов. Если обнаруживаются различия (например, тест, который проходил в старом коммите, начал падать в новом, или наоборот), формируется соответствующее уведомление в Telegram.

Ключевую роль в системе уведомлений играет механизм тегирования результатов тестов и конфигурация подписок. Система уведомлений может

понимать контекст упавшего или исправленного теста и отправлять сообщение только тем, кому данный тест может быть интересен. Для этого все тесты снабжаются тегами с помощью метода `report`. Предусмотрено несколько способов вызова:

- `report(service: Service*)`: Автоматически добавляет стандартный набор тегов для указанных сервисов: их полное имя, владельцев и некоторые другие поля. Например, `report(myService)` добавит тег `"fqn": "my-service"`, `"owner": "..."` и так далее. Более того, для удобства написания тестов сделано так, что обращение к функции `Services.service` само по себе вызывает `report` на переданном сервисе.

- `report(key: String, value: String)`: Добавляет пользовательский тег, например, `report("tag", "critical")`.

Технически, метод `report` вызывает `note` из ScalaTest, передавая строку с тегами в специальном формате. Уже упомянутый ранее кастомный `scalatest.Reporter` перехватывает события `NoteProvided` и извлекает теги, прикрепляя их к результату соответствующего теста (`RunTestResult`).

Сформированные уведомления и ассоциированные с ними теги передаются в `TelegramNotifier`. Последний, на основе конфигурации подписок, определяет, каким пользователям или в какие чаты следует отправить сообщение. Конфигурация подписок загружается в рантайме из системы управления фича-флагами и представляет собой YAML-структуру следующего вида:

- `name`: Человекочитаемое описание подписки.

- `enabled`: Включена ли данная подписка или нет.

- `chat_ids`: Список числовых ID чатов Telegram, куда нужно отправлять уведомления по данной подписке.

- `conditions`: Список условий для срабатывания подписки. Сообщение будет отправлено, если результат теста удовлетворяет всем условиям из этого списка (логическое "И"). Каждое условие — это пара, где ключ

— это имя тега (например, `fqn`, `owner`, `tag` и т.д.), а значение — список допустимых строковых значений. Для того чтобы условие по ключу считалось выполненным, тег с таким ключом у результата теста должен иметь хотя бы одно из значений, перечисленных в списке (логическое "ИЛИ"). Например, условие `fqn: ["service-a", "service-b"]` сработает, если имя сервиса, связанного с тестом, равняется `service-a` или `service-b`.

Ниже приведен пример с двумя подписками на уведомления.

```yaml
subscriptions:
  - name: "Critical issues for Team A"
    enabled: true
    chat_ids:
      - 123456789
      - 987654321
    conditions:
      - tag: ["critical"]
      - owner: ["team-a"]
  - name: "All payment service issues"
    enabled: true
    chat_ids:
      - 112233445
    conditions:
      - fqn: ["payment-api", "payment-tasks"]
```

Пример уведомления в Telegram показан на рисунке 4.

3. Внутреннее устройство библиотеки написания тестов

В данном разделе будет рассмотрено внутреннее устройство библиотеки для написания архитектурных тестов. Библиотека предоставляет разработчикам высокоуровневое API, абстрагируя их от деталей парсинга конфигурационных файлов и представления архитектуры. Центральное место в этом API занимают методы объекта `Service`, позволяющие проверять наличие зависимостей. Далее будут подробно описаны основные концепции, компоненты и алгоритмы, лежащие в основе этого механизма.

3.1. Основные подходы к написанию тестов

Как было показано в разделе 2.3, написание архитектурных тестов сводится к вызову методов на экземплярах класса `Service`

- `hasDependencyOn(service: Service): Boolean`
- `hasDependencyOn(service: ServicePattern): Boolean`
- и так далее

```
1 val serviceA = Services.service("service-a")
2 val serviceB = Services.service("service-b")
3 serviceA.hasDependencyOn(serviceB) shouldBe true
4
5 val ordersCom = ServicePattern.withHost("orders.com")
6 serviceA.hasDependencyOn(ordersCom) shouldBe false
```

Эти методы инкапсулируют логику поиска зависимостей как в явно декларированной секции `depends_on` в карте сервиса, так и в переменных окружения, указанных в манифесте деплоя. Внутренний механизм, обеспечивающий эту функциональность, основан на гибкой системе сопоставления (matching), реализованной с помощью трейта `Matchable`.

3.2. Механизм сопоставления зависимостей

`Matchable[A, B]` определяет единственный метод `matches(a, b)`, который возвращает `true`, если объект `a` типа `A` «соответствует» объекту `b` типа

B по некоторым критериям.

```
1 trait Matchable[A, B] {
2   def matches(a: A, b: B): Boolean
3 }
```

Такой подход позволяет абстрагироваться от конкретных способов сравнения сервисов или их атрибутов, предоставляя возможность легко подменять и комбинировать различные стратегии сопоставления.

3.2.1 Поиск зависимости в секции depends_on

Для проверки, содержится ли сервис `other` в списке явных зависимостей `depend_on`, используется метод `hasDependencyInServiceMap`. Он пробегается по списку зависимостей сервиса и пытается найти среди них совпадение с переданным `other`.

Например, сервис, объявленный в коде как

```
val serviceA = ServicePattern
  .withName("serviceA")
  .withInterfaceName("grpc-api")
```

должен «соответствовать» следующей записи в карте сервисов:

```
depends_on:
  - service: service-a
    interface_name: grpc-api
```

При парсинге YAML-файла, эта запись преобразуется в экземпляр `ServicePattern`. Таким образом, задача сводится к сопоставлению двух `ServicePattern`: `other` и полученного из списка `dependsOn`. А это равносильно написанию `Matchable[ServicePattern, ServicePattern]`.

Библиотека предоставляет экземпляр такого `Matchable` по умолчанию. Он сравнивает сервисы по имени, типу и интерфейсам. Тем не менее, при необходимости пользователь может определить и передать в функцию свой собственный экземпляр `Matchable`.

3.2.2 Поиск зависимости в конфигурационных файлах

Аналогичный подход используется для поиска зависимостей, косвенно определённых через переменные окружения в конфигурационных файлах сервиса.

Например, сервис:

```
val serviceA = ServicePattern
  .withName("serviceB")
  .withInterfaceName("http-api")
```

может быть упомянут в конфигурации другого сервиса через различные переменные окружения:

```
SERVICE_B_HOST: ${host:service-b:http-api}
SERVICE_B_URL: service-b-http-api.yandex.net
```

Для обнаружения подобных неявных зависимостей достаточно пробежаться по списку переменных окружения сервиса с экземпляром `Matchable` для `ServicePattern` и `EnvVariable`, где `EnvVariable` — это простая структура, представляющая пару ключ-значение переменной окружения.

Аналогично поиску в depends_on, библиотека предоставляет экземпляр нужного `Matchable` по умолчанию. Он способен распознавать различные форматы указания зависимостей: URL, внутренние шаблоны Shiva, строки подключения к базам данных, указания на топики Kafka и многое другое.

3.2.3 Гибкость и композируемость Matchable

Ключевое преимущество трейта `Matchable` заключается в его гибкости и возможности композиции. Поскольку типы `A` и `B` не зафиксированы, можно создавать экземпляры `Matchable` для совершенно разных пар типов и комбинировать их для построения сложной логики сопоставления.

Существует два подхода к созданию экземпляров `Matchable`. Более прямолинейный подход заключается в том, что можно определить экземпляр `Matchable` как обычную функцию. Например, для сопоставления двух сервисов по их IP-адресам, можно написать следующее:

```
1 val matchServicesByIp: Matchable1[ServicePattern] =
2   (a: ServicePattern, b: ServicePattern) =>
3     (a.ip, b.ip) match {
4       case (Some(ipA), Some(ipB)) => ipA == ipB
5       case _ => false
6     }
```

Здесь и далее тип `type Matchable1[A] = Matchable[A, A]` определен для удобства.

С прямолинейным подходом получилось не сложнее, чем определить обычную функцию. Правда, переиспользовать такой код вряд ли получится.

Второй подход отличается от первого и активно использует композицию. Библиотека предоставляет набор функций-конструкторов и комбинаторов, позволяющих собирать сложные `Matchable` из более простых «строительных блоков».

- `byEquals[A]`: Создает `Matchable1[A]`, который проверяет равенство переданных значений.

  ```
  def byEquals[A]: Matchable1[A] =
    (a1: A, a2: A) => a1 == a2
  ```

- `lift[A, B, C, D]`: «Поднимает» `Matchable[A, B]` до `Matchable[C, D]`, если есть способы преобразовать `C` в `A` и `D` в `B`.

  ```
  def lift[A, B, C, D](
      m: Matchable[A, B]
  )(ca: C => A,
    db: D => B): Matchable[C, D] =
  m.matches(ca(c), db(d))
  ```

- `lift1[A, C]`: Сокращенная версия `lift[A, B, C, D]` для случая, когда `A == B` и `C == D`.

```
def lift1[A, C](
    m: Matchable1[A]
  )(ca: C => A): Matchable1[C] =
  lift[A, A, C, C](m)(ca, ca)
```

- `options[A, B]`: Преобразует `Matchable[A, B]` в версию с `Option`. Сопоставление происходит, только если оба `Option` содержат значения.

```
def options[A, B](
    m: Matchable[A, B]
  ): Matchable[Option[A], Option[B]] = {
  case (Some(a), Some(b)) => m.matches(a, b)
  case _ => false
}
```

Используя эти конструкторы, пример `matchServicesByIp` можно переписать по-другому.

```
1 val matchStrings: Matchable1[String] =
2   byEquals[String]
3
4 val matchIps: Matchable1[Ip] =
5   matchStrings.lift1[Ip](_.value)
6
7 val matchServicesByIp: Matchable1[ServicePattern] =
8   matchIps.options.lift1[ServicePattern](_.ip)
```

Для улучшения читаемости кода, объявления стандартных функций вроде `lift1` или `options` продублированы в implicit-классах, что позволяет использовать их через точечную нотацию: `matchIps.options.lift1(_.ip)`. Такой подход, хоть и кажется более многословным для простого случая, обеспечивает высокую степень переиспользования промежуточных `Matchable`.

Библиотека также определяет функции `or` и `and` для комбинации нескольких `Matchable`.

```
def or[A, B](inners: Matchable[A, B]*): Matchable[A, B] =
  (a: A, b: B) => inners.exists(_.matches(a, b))

def and[A, B](inners: Matchable[A, B]*): Matchable[A, B] =
  (a: A, b: B) => inners.forall(_.matches(a, b))
```

Это позволяет выражать сложные условия сопоставления очень компактно. Например, стандартный `Matchable1[ServicePattern]`, который используется по умолчанию, определен в библиотеке следующим образом.

```
1 val byIp: Matchable1[ServicePattern] =
2   matchIps.options.lift1[ServicePattern](_.ip)
3
4 // byHost, byFqn, byInterfaces аналогично byIps
5
6 val matchServices: Matchable1[ServicePattern] =
7   byIp or byHost or (byFqn and byInterfaces)
```

То есть сервисы считаются совпадающими, если у них совпадает IP-адрес, или хост, или одновременно полное имя сервиса и его интерфейсы.

3.3. Прямой доступ к конфигурационным данным

Хотя использование `Matchable` является предпочтительным и наиболее гибким способом определения критериев сопоставления, иногда может потребоваться прямой доступ к сырым данным конфигурации. Библиотека оставляет такую возможность, предоставляя доступ к списку всех переменных окружения сервиса через поле `config.all: Seq[EnvVariable]`.

Это может быть полезно для написания специфических проверок, которые сложно или нецелесообразно выражать через `Matchable`. Таким образом разработчик тестов не ограничен предопределенными механизмами `Matchable` и может реализовывать любую необходимую логику.

Рассмотрим пример теста, который проверяет, что у сервиса `api-gateway` нет внешних зависимостей.

```
"api-gateway" should "not have external dependencies" in {
  val apiGateway = Services.service("api-gateway")
  val envVariables = apiGateway.config.all
  envVariables.count(isExternalService) shouldBe 0
}
```

Тест использует функцию `isExternalService`, которая по значению переменной окружения определяет, внешняя это зависимость или нет. Конкретная реализация функции опущена для краткости.

3.4. Оптимизации для повышения производительности

При анализе больших архитектур с сотнями сервисов и тысячами конфигурационных параметров производительность проверок становится важным фактором. В библиотеке реализовано несколько оптимизаций для ускорения процесса сопоставления зависимостей.

3.4.1 Кеширование результатов сопоставления

Тесты могут быть написаны таким образом, что одна и та же пара сервисов будет проверяться на наличие зависимости между ними несколько раз. В таком случае логично кешировать результаты сопоставления внутри экземпляра класса `Service`, чтобы не вычислять их каждый раз заново.

При вызове методов типа `hasDependencyOn` результат сначала ищется в соответствующем кеше. Если для данного аргумента `ServicePattern` результат уже был вычислен ранее, он возвращается немедленно. В противном случае, выполняется честное сопоставление, и результат сохраняется в кеш для последующих вызовов.

Оптимизация валидна, так как граф зависимостей не может поменяться во время выполнения тестов. При этом она срабатывает только при передаче стандартных экземпляров `Matchable`, так как от конкретной реализации трейта зависит результат выполнения функции.

Кеширование обладает небольшими накладными расходами, но зато значительно сокращает количество повторных вычислений, особенно в сценариях с большим количеством тестов.

3.4.2 Эвристики для ускорения сопоставления

Наиболее дорогостоящая операция при выполнении тестов — это поиск зависимостей в конфигурационных файлах, так как он требует итерации по всем переменным окружения и применения к ним потенциально сложных правил `Matchable`. Для ускорения этого и других процессов в библиотеке реализован ряд эвристик и подходов.

Во-первых, это эвристика `skimConfig`, которая применяется перед полным сканированием конфигурации на наличие зависимости от сервиса `other`. Если `other` определён только по своему имени (что верно для большинства сервисов), метод быстро проверяет, содержится ли строка с именем этого сервиса в конкатенированном значении всех переменных окружения. Если имя не найдено, то детальное сопоставление с помощью `Matchable` можно пропустить, так как зависимость по имени точно отсутствует. Эвристика не влияет на корректность результатов, так как используется только тогда, когда ее вывод является гарантированно верным. Эффективность данной схемы обусловлена тем фактом, что граф зависимостей скорее всего будет разреженным, то есть функция `hasDependencyOn` гораздо чаще будет возвращать `false`, чем `true`. При этом поиск подстроки в строке выполняется гораздо быстрее честного сопоставления. В итоге, общий выигрыш в производительности достигается за счет того, что быстрое исключение большинства `false`-случаев легко компенсирует незначительные накладные расходы на выполнение этой предварительной проверки в более редких `true`-случаях.

Во-вторых, для выполнения массовых операций над всеми сервисами системы, таких как фильтрация или поиск, библиотека предоставляет параллельную коллекцию `servicesPar: ParSeq[Service]`, использование которой в тестах позволяет задействовать многопоточность. Стоит отметить, что не всегда замена `services` на `servicesPar` даёт выигрыш, однако грамотное её использование также повышает производительность.

В-третьих, повсеместно применяется стратегия «дешевых предварительных проверок» перед выполнением дорогих операций. Например, известно, что любой внутренний шаблон Shiva для задания хоста сервиса начинается со знака `'$'`. Поэтому перед применением сложного регулярного выражения,

которое бы распарсило шаблон, сначала выполняется простая проверка, что первый символ строки является `'$'`. Если нет, то применение регулярного выражения не производится, что немного ускоряет выполнение.

Эти основные оптимизации, вместе с другими, менее значительными улучшениями, позволили ускорить выполнение набора архитектурных тестов более чем в полтора раза. Если также учитывать выигрыш и от кеширования результатов сопоставления, то общая скорость выполнения тестов увеличилась почти в 2.5 раза по сравнению с вариантом без оптимизаций. Важно подчеркнуть, что точное измерение производительности не входило в задачи данной работы, поэтому приведённые значения являются оценочными и могут отличаться для разных наборов тестов и конфигураций оборудования. Тем не менее, можно утверждать, что после оптимизаций процесс анализа архитектуры заметно ускорился.

4. Применение на реальных микросервисных системах

Для оценки практической применимости и эффективности разработанного инструмента был проведен его запуск на нескольких реальных микросервисных системах, разрабатываемых в Яндекс.Вертикалях. В данном разделе будут представлены итоги этого эксперимента, включая примеры реализованных архитектурных тестов и анализ полученных результатов.

4.1. Описание целевых систем

Запуск инструмента проводился на четырех различных микросервисных системах, условно обозначенных как A, B, C и D. Все четыре системы представляют собой сложные, распределенные приложения внушительного масштаба:

- **Система A**: включает в себя около 550 отдельных компонентов (микросервисы, базы данных, очереди сообщений и т.д.).
- **Система B**: состоит из 300 компонентов.
- **Система C**: насчитывает 400 компонентов.
- **Система D**: включает 150 компонентов.

Названия систем и точное число компонентов не приводятся в целях сохранения внутренней информации компании. Во всех последующих примерах названия конкретных сервисов также изменены по тем же соображениям.

Для лучшего понимания масштаба исследуемых систем отметим, что, по открытым данным, в 2023 году известный сервис Яндекс.Еда функционировал на основе около 180 различных микросервисов, то есть имел сопоставимые системам A-D размеры [28].

4.2. Обнаруженные архитектурные антипаттерны

В ходе эксперимента был реализован набор тестов и правил, нацеленных на выявление как общеизвестных архитектурных антипаттернов, так и нарушений специфичных для конкретных систем.

4.2.1 Shared Database

Антипаттерн Shared Database (Общая база данных) возникает, когда несколько независимых сервисов совместно используют одну и ту же базу данных. Разработанный тест проверяет, что каждая база данных используется не более чем одним сервисом или одной логической группой сервисов. Под логической группой подразумевается ситуация, когда один сервис физически разделён на несколько компонентов, например, когда API сервиса и его асинхронные задачи реализованы как разные развертываемые единицы.

```
1  "Each database" should {
2    services
3      .filter(_.isDatabase)
4      .foreach { database =>
5        s"be used by only one service: ${database.fqn}" in {
6          val dependents = services
7            .filter(_.hasDependencyOn(database))
8
9          fromOneServiceGroup(dependents) shouldBe true
10       }
11     }
12 }
```

Вспомогательная функция `fromOneServiceGroup` определяет принадлежность сервисов-потребителей к одной логической группе, например, по общему префиксу имени или другим метаданным. Конкретная реализация данной функции для краткости опущена.

Данный тест обнаружил 4 общие базы данных в системе A и по одной в системах B и D.

4.2.2 Lava Flow

Антипаттерн Lava Flow (Поток лавы) описывает накопление в системе устаревших и неиспользуемых компонентов. Это могут быть базы данных, сервисы или топики Kafka, на которые ни один другой компонент системы уже не зависит.

Тест для поиска неиспользуемых баз данных:

```
"Each database" should {
  services
    .filter(_.isDatabase)
    .foreach { database =>
      s"be used: ${database.fqn}" in {
        val dependents = services
          .filter(_.hasDependencyOn(database))

        dependents should not be empty
      }
    }
}
```

Тест для поиска неиспользуемых API-сервисов выглядит аналогично. Отметим, что проверять таким образом сервисы, которые не выставляют API, не имеет смысла, так как на них в принципе нельзя зависеть в рамках построенного графа.

```
"Each API service" should {
  services
    .filter(_.isService)
    .filter(_.provides.nonEmpty) // provides some API
    .foreach { service =>
      s"be used: ${service.fqn}" in {
        val dependents = services
          .filter(_.hasDependencyOn(service))

        dependents should not be empty
      }
    }
}
```

С топиками Kafka ситуация несколько сложнее: тест должен проверить, что у каждого топика есть хотя бы один продюсер и хотя бы один консьюмер. Также необходимо учесть, что одна очередь сообщений (представленная как один компонент в графе зависимостей) может предоставлять несколько топиков, каждый из которых тест должен обработать отдельно.

```
"Each kafka topic" should {
  val topics = Services.Kafka.provides.map { topic =>
    // kafka with only one topic
    Services.Kafka.copy(provides = Seq(topic))
  }

  topics.foreach { topic =>
    val topicName = topic.provides.head.name.value

    s"have producer and consumer: $topicName" in {
      val dependents = services
        .filter(_.hasDependencyOn(topic))
      val producers = dependents
        .filter(_.isKafkaProducerFor(topicName))
      val consumers = dependents
        .filter(_.isKafkaConsumerFor(topicName))

      producers should not be empty
      consumers should not be empty
    }
  }
}
```

Вышеописанные тесты помогли обнаружить 29 неиспользуемых сервисов, 5 неиспользуемых баз данных и 28 Kafka-топиков без продюсера или консьюмера: 17 сервисов, 2 базы данных и 9 топиков в системе A; 4 сервиса и 6 топиков в системе B; 6 сервисов, 2 базы данных и 4 топика в системе C; 2 сервиса, 1 база данных и 9 топиков в системе D.

Важно отметить, что не все из обнаруженных «неиспользуемых» компонентов действительно являются таковыми. Некоторые сервисы могут быть предназначены для внешних интеграций, и поэтому внутренние зависимости на них отсутствуют. Также несколько сервисов на момент запуска тестов находились в фазе активной разработки, поэтому формальные зависимости на них еще не были установлены, что, конечно, не является архитектурной проблемой. Более подробный анализ количества ложноположительных срабатываний будет сделан в конце данного раздела.

4.2.3 Cyclic Dependency

Антипаттерн Cyclic Dependency (Циклическая зависимость [29]) между компонентами усложняет развертывание, тестирование и общее понимание системы. Тест ниже находит двусторонние циклические зависимости (циклы длины 2).

```
"Dependency graph" should {
  services.foreach { service =>
    s"not have 2-cycles with ${service.fqn}" in {
      val cycles = services.filter { other =>
        other.fqn != service.fqn &&
        service.hasDependencyOn(other) &&
        other.hasDependencyOn(service)
      }

      cycles shouldBe empty
    }
  }
}
```

Данный тест выявил 53 циклические зависимости в сумме по всем четырем исследуемым системам.

4.2.4 Inverted API Gateway Dependency

Антипаттерн Inverted API Gateway Dependency [30] возникает при некорректном использовании паттерна API Gateway, когда нижестоящие бэкенд-сервисы начинают зависеть от самого Gateway.

```
"Backend service" should {
  services
    .filter(Services.ApiGateway.hasDependencyOn)
    .foreach { dependency =>
      s"not depend on api gateway: ${dependency.fqn}" in {
        val misuseFound = dependency
          .hasDependencyOn(Services.ApiGateway)

        misuseFound shouldBe false
      }
    }
}
```

Этот тест обнаружил 12 нежелательных зависимостей на API Gateway в системе A и по 3 зависимости в системах C и D.

4.2.5 Anti-Corruption Layer Violation

Паттерн Anti-Corruption Layer (ACL) предполагает, что любая интеграция с внешней системой должна проходить через специально выделенный компонент-адаптер. Соответственно, прямая зависимость внутреннего сервиса на внешнюю систему в обход ACL будет являться нарушением этого принципа.

```
"only external-system-acl" should {
  "be allowed to depend on external-system" in {
    val externalSystem = ServicePattern
      .withHost("external.system.com")
    val dependents = services
      .filter(_.hasDependencyOn(externalSystem))

    dependents shouldBe Seq(Services.ExternalSystemAcl)
  }
}
```

Тесты такого типа не выявили нарушений в анализируемых системах.

4.2.6 Другие типовые антипаттерны

Помимо подробно рассмотренных выше антипаттернов, были также реализованы тесты на ряд других распространенных архитектурных проблем. Ниже приведено их краткое описание:

- **Distributed Monolith** [31]. Тест находит клики в графе зависимостей, то есть такие подмножества сервисов, где каждый сервис связан с каждым другим сервисом в этом подмножестве. Такая структура является явным признаком распределённого монолита.

- **Shared Cache**. Аналогично общей базе данных, тест проверяет, что любой используемый кэш (например, Redis) используется только одним сервисом или одной логической группой сервисов.

- **Heterogeneous Database Dependencies**. Тест проверяет микросервисы на наличие у них одновременных зависимостей от нескольких различных типов SQL-баз данных (например, от MySQL и PostgreSQL одновременно). Такая ситуация может быть неоправданной и несколько усложняет эксплуатацию и поддержку соответствующего сервиса.

- **API Gateway Bypass**. Тест обнаруживает прямые обращения от внешних клиентов к внутренним сервисам в обход API Gateway, что является нарушением паттерна API Gateway.

- **Improper External Dependencies**. Тест выявляет зависимости на внешние системы у сервисов, которые для этого не предназначены (например, у внутренних сервисов бизнес-логики или у самого API Gateway).

- **Lack of ownership**. Используя метаинформацию о компонентах системы, предоставляемую Shiva, тест валидирует, что каждым сервисом владеет ровно одна команда или несколько команд, но только из одного бизнес-домена.

4.2.7 Проекто-специфичные правила

Помимо общих антипаттернов, наиболее ценные и релевантные проверки часто возникают из специфических требований конкретной системы или домена. Большинство таких проверок укладываются в следующую классификацию:

- **Ограничение входящих зависимостей**. От компонента `A` могут зависеть только сервисы, удовлетворяющие определённому предикату (например, по имени, владельцу или другим метаданным из Shiva).
- **Ограничение исходящих зависимостей**. Сервис `B` может обращаться только к компонентам (сервисам, базам данных, очередям сообщений и т.д.), соответствующим заданному предикату на основе их метаданных.
- **Контроль отсутствия зависимостей**. Сервис `C` не должен обращаться к другим компонентам системы. Например, являясь прокси к внешней системе или простым сервисом с повышенными требованиями к производительности.
- **Явный запрет определённых зависимостей**. Прямой запрет на взаимодействие между двумя конкретными сервисами или даже между группами сервисов, где группы определяются предикатами, опять же, на основе метаинформации.

Разработанный инструмент позволяет гибко описывать такие проекто-специфичные правила как с помощью Scala-тестов, так и декларативно в YAML-конфигурациях. Причем заметим, что большинство шаблонных проверок могут быть выражены именно через YAML.

4.3. Анализ результатов запуска

4.3.1 Сводные результаты

Результаты запуска всех описанных выше тестов на четырех целевых системах представлены в таблице 1. Каждая строка таблицы соответству-

ет определенному типу найденного архитектурного нарушения, а столбцы отражают количество таких нарушений для каждой из систем A, B, C и D.

Архитектурное нарушение	**A**	**B**	**C**	**D**
Shared Database	4	1	0	1
Unused Database	2	0	1	2
Unused API Service	17	4	6	2
Unused Kafka Topic: No Producers	4	2	0	3
Unused Kafka Topic: No Consumers	5	4	4	6
Cyclic Dependency	15	16	17	5
Inverted API Gateway Dependency	12	0	3	3
Distributed Monolith	2	2	1	1
Shared Cache	1	0	1	3
Heterogeneous Database Dependencies	2	0	2	1
API Gateway Bypass	11	0	8	1
Improper External Dependencies	4	3	9	10
Custom check	2	0	0	0
Lack of ownership	1	0	1	1
Итого	**82**	**32**	**53**	**39**

Таблица 1: Сводные результаты запуска тестов на системах A, B, C и D

Как видно из таблицы 1, для каждой из проанализированных систем было выявлено от нескольких десятков до почти сотни потенциальных архитектурных проблем различной степени критичности.

4.3.2 Анализ ложных срабатываний

Важно отметить, что определённый процент обнаруженных нарушений (по предварительным оценкам, около 8%) может быть классифицирован как «ложноположительные» срабатывания. Причины таких срабатываний могут быть разнообразны. Основной источник — это неточности или неактуальность информации в конфигурационных файлах Shiva. Например, устаревшие записи в секции `depends_on` или переменные окружения, больше не отражающие реальные зависимости. Конкретно в этом случае исправление падающего теста заключается в актуализации самих IaC-конфигураций, что

само по себе является полезным процессом. Помимо этого ложные срабатывания могут возникать из-за не вполне корректно написанных или излишне строгих архитектурных правил, а также указывать на ограничения самого подхода, когда архитектура выводится исключительно из инфраструктурных артефактов. Таким образом, метрика ложноположительных срабатываний в данном контексте скорее отражает текущее состояние и культуру ведения IaC-артефактов в командах разработки, и в меньшей степени указывает на недостатки самого инструмента.

Стоит отдельно отметить, что инструмент намеренно спроектирован таким образом, чтобы можно было игнорировать ложные срабатывания без внесения изменений в код тестов. Например, уведомления в Telegram сообщают в первую очередь о новых или изменившихся архитектурных проблемах относительно базовой версии кода. То есть существующие упавшие тесты не будут постоянно беспокоить разработчиков.

4.3.3 Ограничения используемого подхода

Необходимо понимать, что предлагаемый подход, основанный на статическом анализе конфигурационных файлов и построении графа зависимостей, имеет свои ограничения. Не все типы архитектурных антипаттернов могут быть эффективно обнаружены таким способом. Например:

- **Проблемы производительности**. Архитектурные решения, негативно влияющие на производительность системы (например, неэффективные запросы к базам данных, отсутствие кэширования в узких местах, проблема N+1 запросов), не могут быть выявлены исключительно на основе анализа статических конфигураций зависимостей.

- **Консистентность данных**. Проблемы, связанные с обеспечением согласованности данных между различными микросервисами, особенно если они не используют общие хранилища, сложно детектировать без глубокого анализа протоколов взаимодействия и логики обработки сообщений.

- **Сложные поведенческие антипаттерны**. Антипаттерны, которые про-

являются в динамике во время выполнения системы (например, некорректная обработка ошибок, приводящая к каскадным сбоям, или неправильная реализация паттернов отказоустойчивости, таких как Circuit Breaker), требуют анализа логов, метрик выполнения или трассировок вызовов, а не только статических конфигураций.

4.3.4 Выводы

Разработанный инструмент был апробирован на четырёх реальных микросервисных системах, состоящих из сотен компонентов. В ходе этого эксперимента был реализован набор архитектурных тестов, нацеленных на выявление как общеизвестных антипаттернов, так и нарушений специфических правил конкретных систем. Применение инструмента позволило обнаружить значительное количество потенциальных архитектурных проблем в каждой из проанализированных систем. Таким образом, полученные результаты подтверждают практическую применимость реализованного инструмента к автоматизированному контролю качества архитектуры.

Заключение

Основным результатом работы является готовый к использованию инструмент, способный на основе инфраструктурных конфигураций микросервисной системы выявлять потенциальные архитектурные нарушения. В процессе достижения этой цели были выполнены следующие задачи.

- Проведён обзор предметной области и существующих решений для автоматизированной валидации архитектуры, в ходе которого были выявлены их ключевые недостатки — ограниченная область применения и сложность в интеграции с корпоративной инфраструктурой.
- Разработан инструмент, предоставляющий гибкие возможности для описания архитектурных правил любой сложности и их автоматической проверки применимо к конкретной архитектуре.
- Реализован простой способ написания новых правил для шаблонных ситуаций через YAML-конфигурации, что ускоряет процесс добавления стандартных проверок.
- В инструмент заложен набор правил по умолчанию, проверяющих соблюдение распространённых архитектурных антипаттернов.
- Обеспечена интеграция решения с используемыми в Яндекс.Вертикалях системами и инструментами разработки, включая возможность сравнения архитектурных изменений в ветках и получение уведомлений через Telegram.
- Измерена эффективность инструмента на четырёх реальных микросервисных системах. Анализ полученных результатов показал способность инструмента выявлять десятки потенциальных архитектурных нарушений, формируя числовую метрику архитектурного технического долга анализируемого проекта.

Список литературы

[1] Fowler, M., Lewis, J. Microservices: A Definition of This New Architectural Term. Martin Fowler Blog, 2014. URL: https://martinfowler.com/articles/microservices.html (дата обр. 19.04.2025).

[2] Richardson, C. Microservices Patterns. Manning Publications, 2018.

[3] Brown, N., Cai, Y., Guo, Y., Kazman, R. Managing Technical Debt in Software-Reliant Systems. In Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10), 2010, pp. 47–52.

[4] Morris, K. Infrastructure as Code: Managing Servers in the Cloud. O'Reilly Media, 2016.

[5] Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[6] Taibi, D., Lenarduzzi, V., Pahl, C. Microservices Anti-Patterns: A Taxonomy. Microservices - Science and Engineering. Springer. 2019.

[7] Fowler, M. Domain-Specific Languages. Addison-Wesley Professional, 2010.

[8] Brown, W. J., Malveau, R. C., McCormick, H. W., Mowbray, T. J. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, 1998.

[9] Richards, M. Software Architecture Patterns. O'Reilly Media, 2015.

[10] Wolff, E. Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016.

[11] Contieri, M. Lava Flow. In Proceedings of the 21st Conference on Pattern Languages of Programs (PLoP '14), 2014, Article No. 9.

[12] ArchUnit. Official Website. URL: https://www.archunit.org/ (дата обр. 20.04.2025).

[13] SonarSource. SonarGraph Documentation. URL: https://docs.sonarsource.com/sonarqube-server/latest/ (дата обр. 20.04.2025).

[14] Structure101. Official Website. URL: https://structure101.com/ (дата обр. 20.04.2025).

[15] Сафин, Р. "Раз архитектура — «as Code», почему бы её не покрыть тестами?". ArchDays Conference, 2023. URL: https://github.com/Byndyusoft/aact/ (дата обр. 25.04.2025).

[16] Objects In Kubernetes. https://kubernetes.io/docs/concepts/overview/working-with-objects/ (дата обр. 25.05.2025).

[17] Ansible community documentation. https://docs.ansible.com/ (дата обр. 25.05.2025).

[18] Terraform by HashiCorp. Official Website. URL: https://www.terraform.io/ (дата обр. 27.04.2025).

[19] PlantUML. Official Website. URL: https://plantuml.com/ (дата обр. 26.04.2025).

[20] Mermaid. Official Website. URL: https://mermaid.js.org/ (дата обр. 26.04.2025).

[21] Humble, J., Farley, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010.

[22] Istio. Official Website. URL: https://istio.io/ (дата обр. 27.04.2025).

[23] gRPC. Official Website. URL: https://grpc.io/ (дата обр. 03.05.2025).

[24] Trunk Based Development. URL: https://trunkbaseddevelopment.com/ (дата обр. 03.05.2025).

[25] Arc — система контроля версий для монорепозитория. https://habr.com/ru/companies/yandex/articles/482926/ (дата обр. 25.05.2025).

[26] ScalaTest. Official Website. URL: http://www.scalatest.org/ (дата обр. 03.05.2025).

[27] YAML Ain't Markup Language. Specification. URL: https://yaml.org/spec/1.2/spec.html (дата обр. 03.05.2025).

[28] Yandex Go Tech. От монолита до микросервисов: как менялась архитектура Яндекс Еды. Блог Yandex Go Tech, 2024. URL: https://dev.go.yandex/blog/from-monolith-to-microservices-2024-03-12 (дата обр. 17.05.2025).

[29] Farsi1, H., Allaki1, D., En-Nouaary1, A., Dahchour, M. An Iterative Approach for Detecting and Resolving Cyclic Dependencies AntiPattern in Microservices Architectures, 2025.

[30] Pattern: API Gateway. https://microservices.io/patterns/apigateway.html (дата обр. 18.05.2025).

[31] Microservices Antipattern: The Distributed Monolith. URL: https://mehmetozkaya.medium.com/microservices-antipattern-the-distributed-monolith-%EF%B8%8F-46d12281b3c2 (дата обр. 18.05.2025).

Приложение

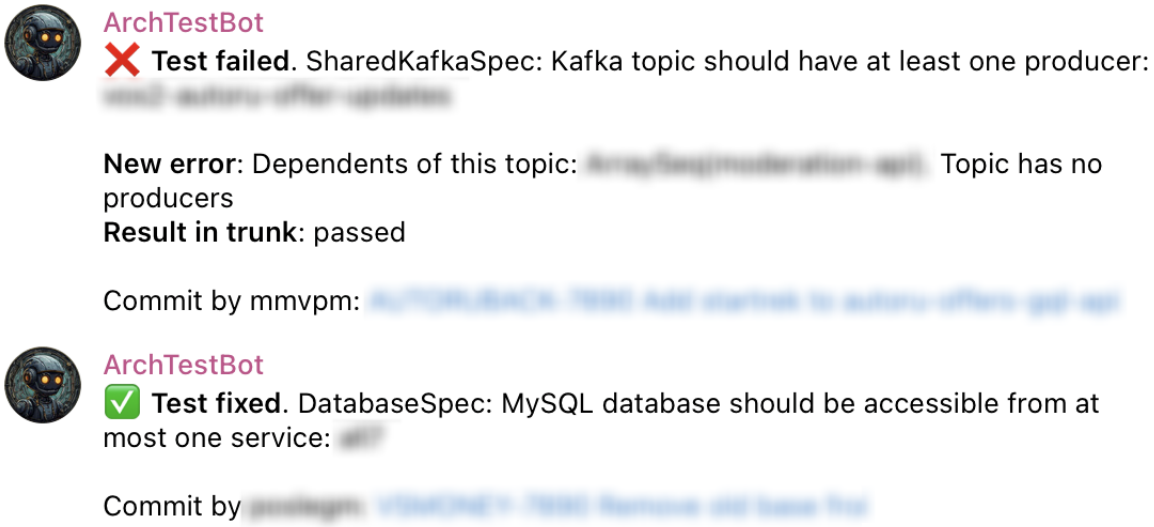

Рис. 4: Пример уведомлений в Telegram.