МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГАОУ ВО «МГТУ «СТАНКИН»)

Институт
социально-технологического менеджмента

Кафедра
автоматизированных систем обработки информации и управления

Александров Александр Владимирович

Выпускная квалификационная работа
по направлению подготовки 27.03.02 «Управление качеством»

на тему:
«Повышение качества разработки программного обеспечения на основе применения архитектурных подходов»

Регистрационный номер №_____

Заведующий кафедрой «АСОИиУ»
д.т.н., доцент _____ А.В. Капитанов
подпись

Руководитель
к.т.н., доцент _____ С.А. Тясто
подпись

Обучающийся _____ А.В. Александров
подпись

Москва 2025 г.

МИНОБРНАУКИ РОССИИ
федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технологический университет «СТАНКИН»

(ФГАОУ ВО «МГТУ «СТАНКИН»)

Институт
социально-технологического
менеджмента

Кафедра
автоматизированных систем обработки
информации и управления

«УТВЕРЖДАЮ»
Заведующий кафедрой АСОИиУ
_____ А.В. Капитанов
«01» ноября 2024 г.

ЗАДАНИЕ

на выпускную квалификационную работу
по направлению подготовки 27.03.02 «Управление качеством»
Александрова Александра Владимировича

Тема: «Повышение качества разработки программного обеспечения на основе применения архитектурных подходов»

Тема утверждена приказом от «30» октября 2024 г. №893/1.
Срок сдачи законченной ВКР на кафедру «02» июня 2025 г.

Перечень вопросов, подлежащих разработке:

1. Выполнить анализ существующих методов и средств оценки качества программного обеспечения;
2. Исследовать влияние современных архитектурных подходов, применения шаблонов разработки, методов тестирования на качество программного обеспечения;
3. Разработать рекомендации по применению архитектурных подходов и инструментов в разработке программного обеспечения.

Руководитель
к.т.н., доцент кафедры АСОИиУ _____ С.А. Тясто
подпись

Обучающийся _____ А.В. Александров
подпись

Содержание

ВВЕДЕНИЕ

В условиях цифровой трансформации промышленности программное обеспечение становится ключевым элементом управления производственными процессами. Оно определяет эффективность оборудования, точность технологических операций, уровень автоматизации и гибкость производственной системы. Особенно это актуально для машиностроительных предприятий в связи с распоряжением Правительства Российской Федерации от 07.11.2023 г. № 3113-р – «Об утверждении стратегического направления в области цифровой трансформации обрабатывающих отраслей промышленности». Современные предприятия сталкиваются с необходимостью выбора архитектурного подхода, который обеспечит не только высокое качество программных решений, но и их адаптивность к изменяющимся требованиям бизнеса.

Выбор архитектуры программного обеспечения играет определяющую роль в его масштабируемости, отказоустойчивости и поддерживаемости. От того, как организована система, зависит её способность справляться с увеличивающейся нагрузкой, поддерживать стабильность работы в случае отказов отдельных компонентов и адаптироваться к изменениям в производственном процессе.

Особое внимание в этой работе уделено влиянию архитектурных решений на процесс разработки и конечное качество программного обеспечения в производственной среде.

Цель выпускной квалификационной работы: разработка моделей повышения качества разработки программного обеспечения в производственной среде.

Для выполнения поставленной цели необходимо решить следующие задачи:

1. Выполнить анализ существующих методов и средств оценки качества программного обеспечения;

2. Исследовать влияние современных архитектурных подходов, применения шаблонов разработки, методов тестирования на качество программного обеспечения;

3. Разработать рекомендации по применению архитектурных подходов и инструментов в разработке программного обеспечения.

ГЛАВА 1. КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ: КРИТЕРИИ, ПОКАЗАТЕЛИ И СТАНДАРТЫ

1.1 Критерии оценки качества программного обеспечения (качественные показатели)

Качество программного обеспечения определяется совокупностью характеристик, которые позволяют продукту удовлетворять заданные требования заказчика. Далее будут рассмотрены основные требования согласно одному из стандартов, а также краткое описание функции критерия. ГОСТ Р ИСО 25010-2015 (рис. 1.1) выделяет для оценки качества программного обеспечения следующие критерии:

- **Функциональность:** «Функциональная полнота. Функциональная корректность. Функциональная целесообразность».

Обеспечивает выполнение программным обеспечением всех предусмотренных задач в полном объеме, с корректным и целесообразным поведением.

- **Уровень производительности:** «Временные характеристики. Использование ресурсов. Потенциальные возможности».

Оценивает эффективность работы программного обеспечения в реальных условиях. Основная задача — обеспечить быстродействие, оптимальное использование ресурсов (памяти, процессора) и возможность масштабирования под нагрузкой.

- **Совместимость:** «Сосуществование. Интероперабельность»

Обеспечивает способность программного обеспечения взаимодействовать с другими системами и компонентами. Это гарантирует корректное сосуществование и обмен данными с различными программными и аппаратными средами.

- **Удобство использования:** «Определимость пригодности. Изучаемость. Управляемость. Защищённость от ошибки пользователя. Эстетика пользовательского интерфейса. Доступность»

Направлена на обеспечение комфортного и интуитивного взаимодействия пользователя с системой. Это включает простоту освоения интерфейса, предотвращение пользовательских ошибок и общую привлекательность дизайна.

- **Надёжность:** «Завершённость. Готовность. Отказоустойчивость. Восстанавливаемость»

Обеспечивает стабильную работу программного обеспечения в заданных условиях. Критерий отвечает за устойчивость к отказам, способность продолжать работу при частичных сбоях и быстрое восстановление после неисправностей.

- **Защищённость:** «Конфиденциальность. Целостность. Неподдельность. Отслеживаемость. Подлинность»

Предотвращает несанкционированный доступ к данным и нарушения их целостности. Также обеспечивает аутентификацию пользователей, отслеживаемость действий и защиту конфиденциальной информации.

- **Сопровождаемость:** «Модульность. Возможность многократного использования. Анализируемость. Модифицируемость. Тестируемость»

Делает программное обеспечение гибким для изменений и доработок. Это включает возможность повторного использования компонентов, легкость анализа и модификации кода, а также простоту тестирования.

- **Переносимость:** «Адаптируемость. Устанавливаемость. Взаимозаменяемость»

Обеспечивает способность программного обеспечения адаптироваться к новым средам, легко устанавливаться и заменять другие аналогичные системы. Этот критерий важен для обеспечения жизнеспособности продукта в различных условиях эксплуатации.

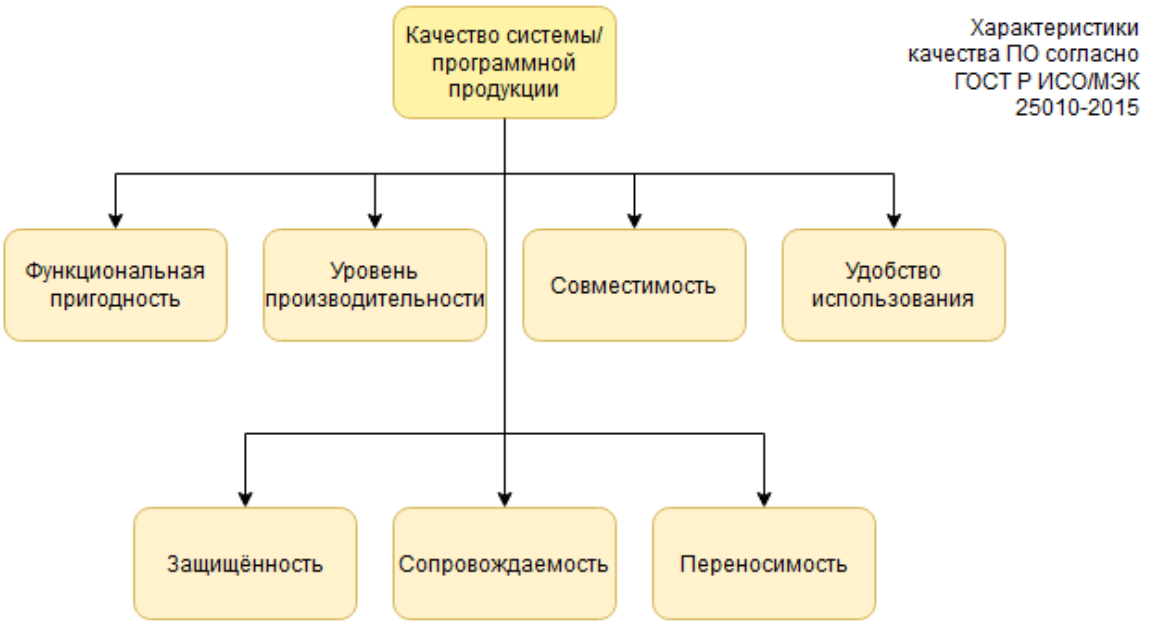

Рис. 1.1 – Характеристики качества ПО ГОСТ Р ИСО 25010-2015.

Для обеспечения высокого уровня качества программного обеспечения разработано множество стандартов, которые устанавливают принципы, критерии и подходы к разработке, тестированию и сопровождению.

1.2 Критерии оценки качества архитектуры ПО

Кроме того, отдельное внимание уделяется характеристикам архитектуры программного обеспечения. Для их оценки и описания может применяться стандарт ГОСТ Р 57100-2016/ISO/IEC/IEEE 42010:2011 «Системная и программная инженерия. Описание архитектуры. Systems and software engineering. Architecture description» Этот стандарт включает аспекты, связанные с проектированием архитектуры, и определяет характеристики, которые позволяют программному обеспечению быть функциональным, надежным и адаптируемым к изменяющимся условиям.

В разделе «4.2.3 Заинтересованные стороны и интересы» выделяются следующие возможные характеристики качества архитектуры ПО:

- **Функциональность**

Способность системы выполнять задачи в рамках предъявляемых

требований.

- **Надежность**

Обеспечение стабильной работы системы без сбоев в условиях нормального использования.

- **Безопасность**

Защита от угроз, включая несанкционированный доступ и утечку данных.

- **Гибкость**

Адаптивность системы к изменениям требований или условий эксплуатации.

- **Модульность**

Разделение системы на независимые компоненты для упрощения разработки и сопровождения.

Дополнительно: применимость, стоимость, доступность данных, динамичность, управляемость, интеграция подсистем.

1.3 Количественные показатели, характеризующие качество разработки программного обеспечения

Количественные показатели качества программного обеспечения базируются на математическом аппарате оценки, который позволяет формально измерять сложность, объём, надёжность и сопровождаемость кода. Эти метрики используются для:

- объективного сравнения версий программ;
- планирования тестирования и оценки трудоёмкости поддержки;
- выявления потенциальных проблем в реализации и архитектуре системы.

Приведённый перечень (табл. 1.1) не является исчерпывающим, и в зависимости от специфики проекта могут использоваться дополнительные показатели.

Таблица 1.1

Количественные показатели качества ПО

Метрика	**Описание**	**Формула**	**Комментарий**
Метрика SLOC (Source lines of Code) [8]	Измеряет объём исходного кода через подсчёт строк (без учета комментариев и пустых строк).	SLOC = число строк кода	Служит базовым показателем объёма программы, что косвенно влияет на трудоемкость разработки и поддержку.
Метрика «спена» (Span Metric)	Определяет количество повторных обращений к одному и тому же идентификатору в пределах программного блока.	$Span = n - 1$	Высокое значение указывает на то, что изменение идентификатора затрагивает множество участков кода, что усложняет тестирование и отладку.
Метрика Чепина (Chepin's Metric)	Оценивает информационную прочность модуля через анализ переменных ввода/вывода, разделённых на группы: V (вводимые), M (модифицируемые), U (управляющие) и L (паразитные).	CH $= \omega_1 V + \omega_2 M + \omega_3 U + \omega_4 L$	Позволяет определить устойчивость модуля к изменениям, выявляя сложность управления данными.

Метрика МакКейба (цикломатическая сложность программы) (McCabe Metric, cyclomatic complexity)	Оценивает логическую сложность программы через количество независимых путей исполнения, рассчитанных на основе графа потока управления.	$Z(G) = m - n + 2r$	Высокое значение цикломатической сложности свидетельствует о необходимости большего количества тестов для полного покрытия всех ветвей.

Метрика спена (Span Metric)

Идея заключается в том, что чем больше раз переменная повторно используется между своим первым и последним появлением, тем выше вероятность, что изменения в этой переменной затронут множество участков кода. Это может усложнять тестирование, отладку и сопровождение.

$$Span = n - 1$$

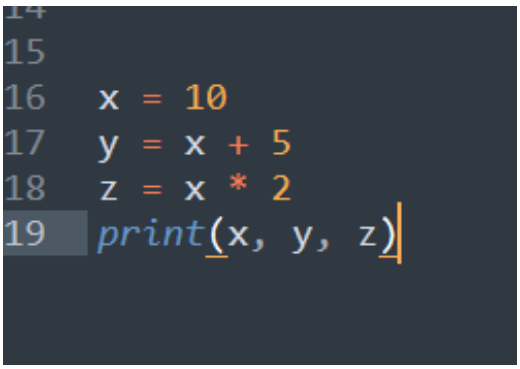

Рис. 1.2 – Фрагмент кода для расчёта метрики «спана».

Рассчитаем количество переходов от одного использования переменной x к другой:

$$Span = 3 - 1 = 2$$

Интерпретация полученного значения:

- Если большинство идентификаторов в модуле имеют небольшой спан (например, 1–2), то модуль можно считать относительно простым с точки зрения повторного использования переменных;
- Если же «спан» некоторых идентификаторов значительно выше, это

может указывать на высокую степень их задействованности и, соответственно, повышенную чувствительность к изменениям. Модуль с высоким значением «спана» может потребовать более тщательного тестирования или даже рефакторинга[1] для снижения взаимозависимости частей кода.

Метрика Чепина (Chepin's Metric)

Суть метрики состоит в оценке информационной прочности отдельно взятого

программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

$$\boldsymbol{CH} = \omega_1 V + \omega_2 M + \omega_3 U + \omega_4 L$$

Все множество переменных, составляющих список ввода-вывода, разбивается на четыре функциональные группы:

1. $\boldsymbol{V}$ – вводимые переменные для расчётов и обеспечения вывода. Примером может служить используемая в программах лексического анализатора переменная, содержащая строку исходного текста программы, т.е. сама переменная не модифицируется, а только содержит исходную информацию;

2. $\boldsymbol{M}$ – модифицируемые (создаваемые внутри программы) переменные;

3. $\boldsymbol{U}$ – переменные, участвующие в управлении работой программного модуля (управляющие переменные);

4. $\boldsymbol{L}$ – не используемые в программе (паразитные) переменные. Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе;

$\boldsymbol{\omega_i}$ – весовые коэффициенты.

Весовые коэффициенты в метрике Чепина не являются универсальными

[1] **Рефакторинг** (англ. refactoring), или **перепроектирование кода**, **переработка кода**, **равносильное преобразование алгоритмов** — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.

константами, а скорее устанавливаются эмпирическим путём на основе анализа влияния каждой функциональной группы переменных на информационную прочность определённого программного модуля.

В научной статье Антона Милютина [19] вводятся следующие коэффициенты:

- $\boldsymbol{\omega_1 = 1}$ для вводимых переменных ($\boldsymbol{V}$),
- $\boldsymbol{\omega_2 = 2}$ для модифицируемых переменных ($\boldsymbol{M}$),
- $\boldsymbol{\omega_3 = 3}$ для управляющих переменных ($\boldsymbol{U}$), поскольку они, как правило, оказывают наибольшее влияние на поток управления,
- $\boldsymbol{\omega_4 = 0,5}$ для «паразитных» переменных ($\boldsymbol{L}$), так как их влияние меньше, но всё же не должно быть нулевым, поскольку они могут затруднять понимание кода.

Рассмотрим пример на Python (рис. 1.3) и рассчитаем значение Чепина:

```
1
2  def calculate(a, b):
3      # Входные переменные (V): a, b
4      result = a + b # Модифицируемая переменная (M)
5
6      # Управляющая переменная (U): flag используется для выбора ветки
7      if result > 10:
8          flag = True # U
9      else:
10         flag = False # та же переменная (U)
11     temp = result * 2 # Модифицируемая переменная (M)
12     unused = 0 # Паразитная переменная (L) - объявлена, но не используется
13     return temp
```

Рис. 1.3 – Фрагмент кода с типами данных по метрике Чепина.

В функции **calculate** выполняются следующие действия:

- Принимается два входных параметра *a* и *b* (группа $\boldsymbol{V}$);
- Вычисляется их сумма и сохраняет в переменной *result* (группа $\boldsymbol{M}$);
- Проверяется условие: если *result* больше 10, то устанавливается управляющая переменная *flag* в значение True, иначе — в False (группа $\boldsymbol{U}$);
- Вычисляется значение *temp* как удвоенное значение *result* (еще одна модифицируемая переменная, группа $\boldsymbol{M}$);

- Объявляется переменная *unused*, которая не используется в дальнейших вычислениях (группа ***L***);
- Возвращается значение *temp*.

Подсчитаем количество переменных в каждой группе:

- ***V*** (вводимые переменные): **2** (параметры *a*, *b*);
- ***M*** (модифицируемые переменные): **2** (*result* и *temp*);
- ***U*** (управляющие переменные): **1** (*flag*);
- ***L*** (паразитные переменные): **1** (*unused*).

Подставим в формулу:

$$CH = 1 * 2 + 2 * 2 + 3 * 1 + 0{,}5 * 1 = 2 + 4 + 3 + 0{,}5 = 9{,}5$$

В дальнейшем это значение можно использоваться для сравнительного анализа кода других модулей.

Метрика МакКейба (McCabe Metric)

Предложена Томасом Дж. МакКейбом в 1976 году и применяется для оценки сложности программ, связанной с трудоемкостью тестирования. Основным показателем здесь является **цикломатическое число МакКейба** $Z(G)$, которое показывает, сколько независимых путей исполнения существует в модуле. Чем выше это значение, тем больше вариантов исполнения, что требует большего числа тестовых сценариев для полного покрытия всех путей.

$$Z(G) = m - n + 2r$$

m – количество дуг (рёбер) в графе;

n – количество вершин;

r – количество компонентов связности (узлов, имеющих точки выхода).

```
20
21  def simple_example(a, b):
22      if a > b:
23          print("a больше")
24      else:
25          print("b больше или равно")
26      print("Конец")
```

Рис. 1.4 – Фрагмент кода для расчёта цикломатической сложности.

Представим фрагмент кода (рис. 1.4) в виде *графа потока управления*[2] (рис. 1.5):

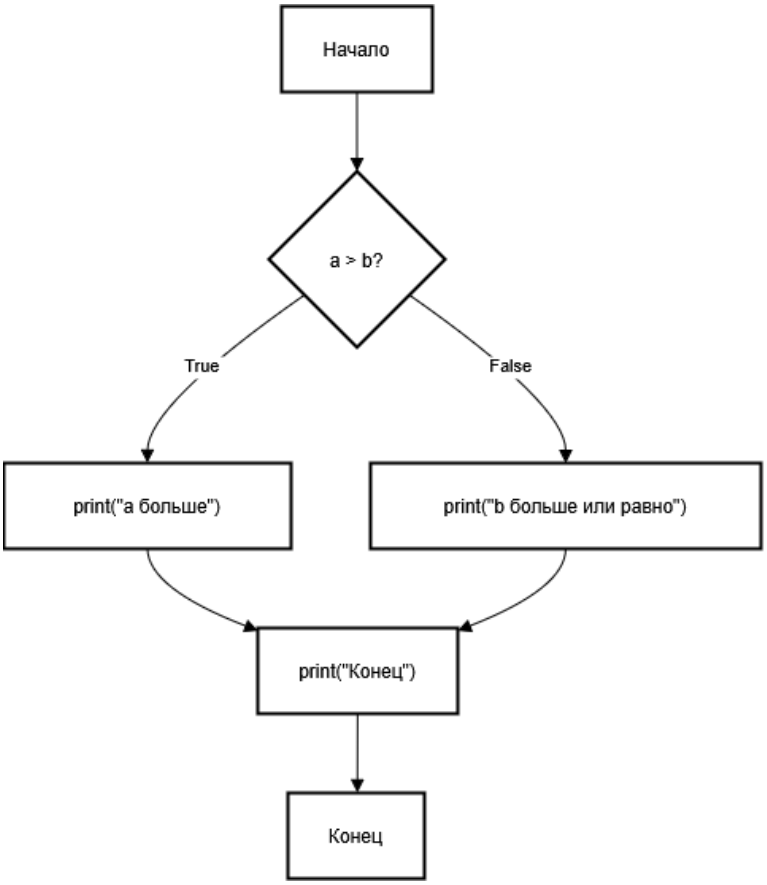

Рис. 1.5 – Граф потока управления.

Количество узлов $n = 6$;

Количество дуг $m = 6$;

В данном графе имеется одна связная компонента, то есть $r = 1$.

[2] **Граф потока управления** (англ. control flow graph, CFG) — в теории компиляции — множество всех возможных путей исполнения программы, представленное в виде графа.

Подставим значения в формулу:

$$Z(G) = 6 - 6 + 2 * 1 = 2$$

Значение цикломатической сложности $Z(G) = 2$ означает, что в функции имеется два независимых пути исполнения. Практически это говорит о том, что для полного тестирования этой функции необходимо разработать не менее двух тестовых сценариев, покрывающих каждую из ветвей (когда условие истинно и когда оно ложно).

Количественные показатели качества ПО, рассмотренные выше, представляют собой не только инструменты для анализа сложности и надёжности отдельных модулей, но и служат важными индикаторами, непосредственно влияющими на **архитектуру** программного обеспечения. Высокие значения этих метрик могут свидетельствовать о чрезмерной связности или запутанности кода, что приводит к трудностям в тестировании, сопровождении и масштабировании системы.

ГЛАВА 2. АРХИТЕКТУРНЫЕ ПОДХОДЫ В РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Разработка ПО требует применения системного подхода, который позволяет структурировать проектирование, управление компонентами и дальнейшую поддержку программных продуктов. Одним из инструментов, позволяющих формализовать и упорядочить процесс проектирования, являются архитектурные фреймворки[3].

2.1 Архитектурные фреймворки для ПО

Архитектурные фреймворки представляют собой совокупность методик, концепций и инструментов, позволяющих структурировать процесс создания, документирования и развития информационных систем. Они обеспечивают унификацию подходов к проектированию, способствуют снижению количества ошибок на ранних этапах разработки и позволяют достичь высокой гибкости, масштабируемости и отказоустойчивости конечного продукта.

4+1 View Model

Была предложена Филиппом Кручтеном из компании Rational в 1995 году. Включает пять видов представлений (логическое, процессное, физическое, уровня разработки и сценарное), что позволяет рассматривать систему с разных точек зрения (рис. 2.1). Применяется в крупных информационных системах для обеспечения четкой и эффективной документации архитектуры.

[3] **Фре́ймворк**; иногда фреймво́рк (англицизм от framework «каркас, рама; структура») — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

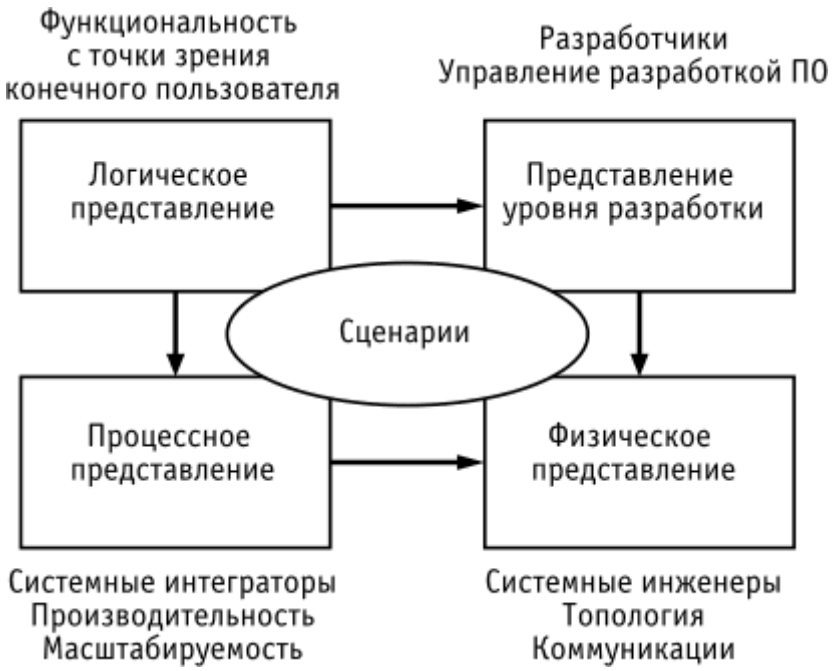

Рис. 2.1 – Модель представления архитектуры ПО «4+1»

RM-ODP (Reference Model of Open Distributed Processing)

Эталонная модель открытой распределённой обработки, разработанная ИСО и Международной электротехнической комиссией (МЭК). В Германии она была применена при описании архитектуры электронного правительства. Также телекоммуникационные компании США, швейцарский банк UBS AG использовали её для проектирования своей ИТ-архитектуры.

В её основе лежит идея, что комплексную систему можно рассматривать с пяти различных точек зрения, каждая из которых отражает специфические аспекты функционирования и разработки:

1. **Корпоративная (Enterprise)** – описывает цели, политику и стратегические вопросы бизнеса, определяя, какие процессы и функции должны быть поддержаны системой;

2. **Информационная (Information)** – фокусируется на структуре данных и информационных потоках, необходимых для обеспечения функционирования системы;

3. **Вычислительная (Computational)** – разделяет систему на отдельные функции и компоненты, определяя их взаимодействие и распределение ответственности;

4. **Инженерная (Engineering)** – описывает механизмы распределения компонентов по физическим ресурсам, обеспечивая надежную коммуникацию между ними;

5. **Технологическая (Technology)** – отражает выбор конкретных технологий, платформ и средств реализации, которые будут использованы для построения системы.

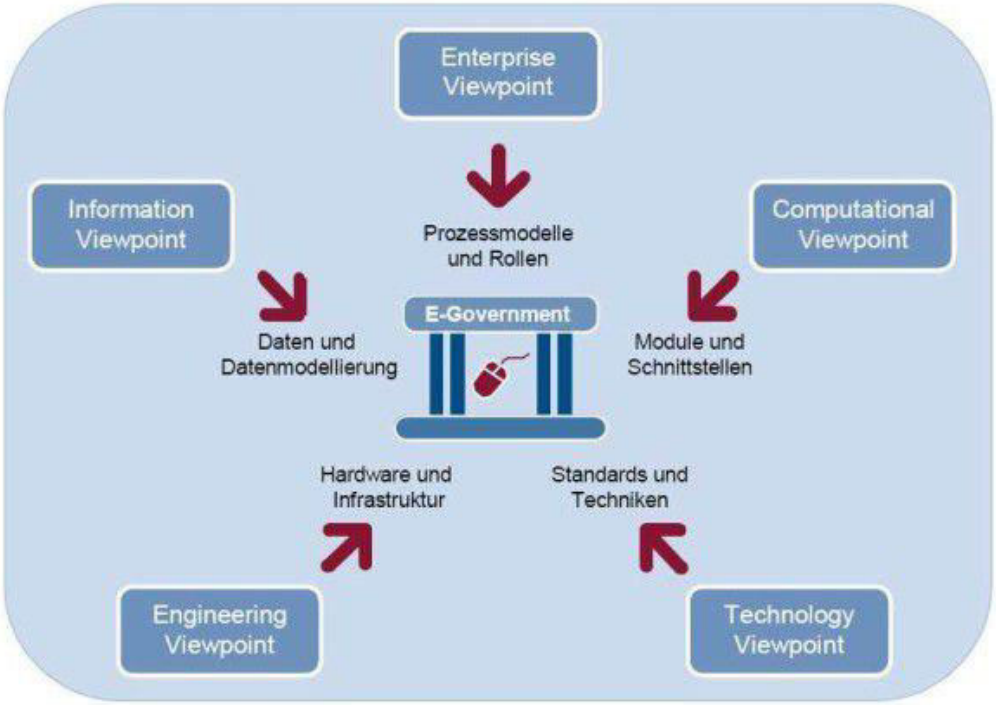

Рис. 2.2 – Точки зрения архитектуры RM-ODP

SOMF (Service-Oriented Modeling Framework)

Фреймворк, специализирующийся на проектировании архитектур сервис-ориентированных решений (SOA). На практике SOMF используется в крупных корпоративных решениях, таких как системы для финансовых учреждений, телекоммуникационных компаний и в других отраслях. Пример представления системы при помощи SOMF (рис. 2.3).

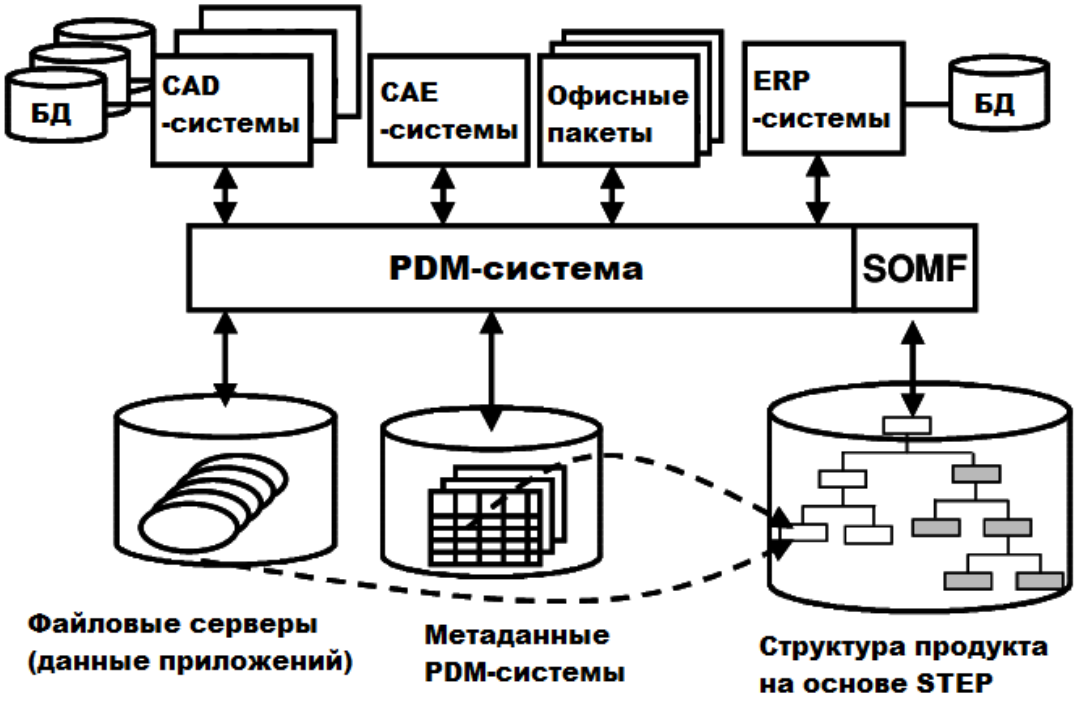

Рис. 2.3 – Пример представления архитектуры при помощи фреймворка SOMF

2.2 Архитектуры разработки ПО

- **Монолитная архитектура** (monolithic architecture) – классический шаблон к разработке ПО, когда приложение разрабатывается в формате «всё в одном». Это одна большая вычислительная сеть с единой базой кода, в которой объединены все бизнес-задачи (рис. 2.4). Применяется примерно с 1960-ых годов.

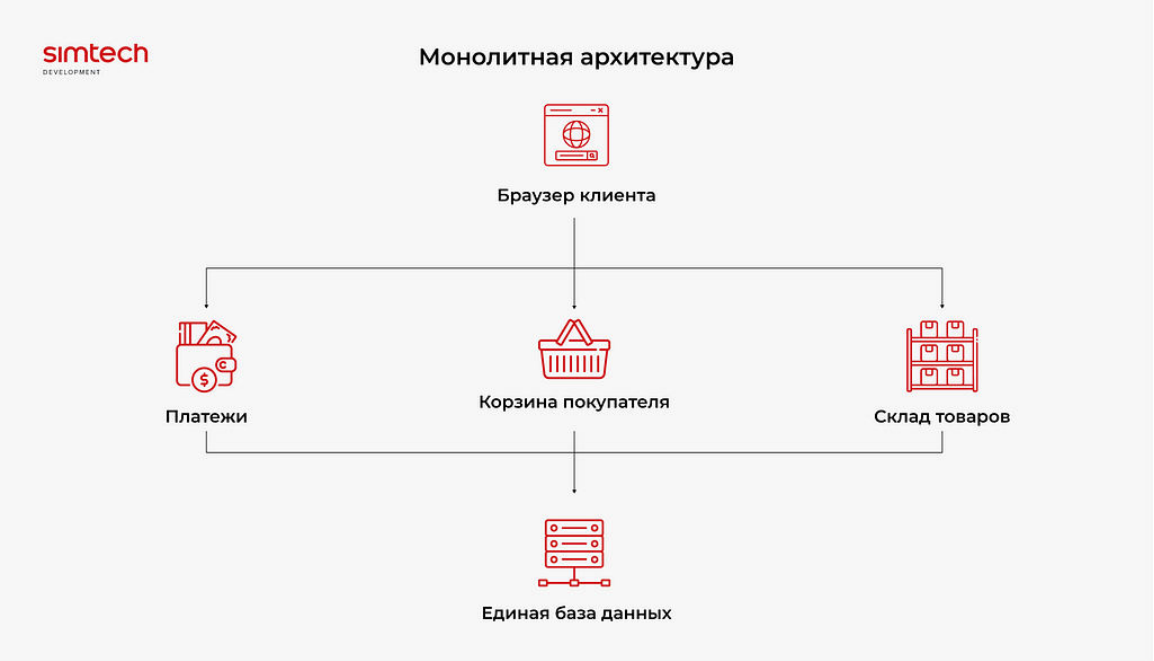

Рис. 2.4 – Модель монолитной архитектуры

- **Сервис-ориентированная архитектура (SOA)** – архитектурный стиль, при котором функции приложения предоставляются в виде независимых сервисов. Основная цель SOA — обеспечить возможность повторного использования и гибкость в разработке и интеграции программных компонентов. Ключевыми характеристиками SOA являются:
  - **Автономность сервисов**: каждый сервис является независимым и может функционировать без вмешательства других сервисов;
  - **Переиспользуемость сервисов**: сервисы могут быть использованы повторно в различных приложениях и контекстах (например, сервис аутентификации может быть задействован как в веб-приложении, так и в мобильном приложении, обеспечивая единое решения для управления доступом).

Схема SOA обычно включает в себя следующие основные компоненты: сервисы, потребители сервисов, корпоративная сервисная шина (англ. enterprise service bus, **ESB**) (рис. 2.5 и рис. 2.6), а также различные инструменты для управления и мониторинга.

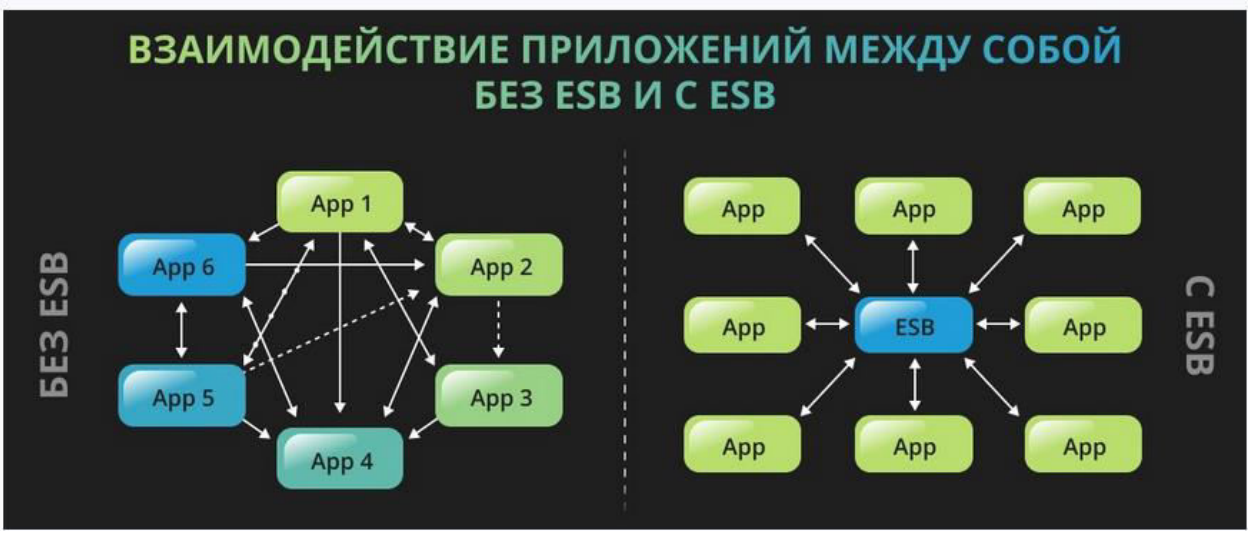

Рис. 2.5 – Взаимодействие приложений между собой без ESB и с ESB

Рис. 2.6 – Готовое решение ESB – 1С:Шина для консолидации интегрированных систем

Примеры продуктов ESB на рынке: IBM Integration Bus, Oracle Service Bus, 1С:Шина.

Основной принцип работы сервисной шины заключается в централизованном обмене сообщениями между разными системами через единый узел. Именно в этом узле выполняются такие задачи, как транзакционный контроль, преобразование данных и обеспечение сохранности сообщений. Все настройки, связанные с обработкой и маршрутизацией сообщений, сосредоточены в одной точке и задаются в конфигурации. Благодаря такому подходу при замене одной из подключенных к шине информационных систем не требуется перенастройка остальных.

Термин «сервисная шина» выбран по аналогии с компьютерной системной шиной, которая объединяет несколько устройств и обеспечивает передачу данных по общему набору проводников.

- **Микросервисная архитектура** (microservice architecture) – вариант сервис-ориентированной архитектуры программного обеспечения. SOA и микросервисная архитектура решают задачу построения распределённых систем, но подходы у них существенно различаются. Микросервисная архитектура направлена на взаимодействие насколько это возможно небольших, *слабо* связанных и легко изменяемых модулей (рис. 2.7), а также отсутствует применение ESB. Впервые архитектура была упомянута в 2000-ых годах, среди пионеров идеи: Питер Роджерс (HP Labs), Алистер Коберн (соавтор Agile манифеста). Активно применяется примерно с середины 2010-ых годов. Согласно исследованию O’Reilly в 2020 году («Microservices Adoption in 2020»), микросервисы в своей работе используют 77% компаний [5].

Один из ярких примеров перехода к микросервисной архитектуре — это опыт компании Netflix[4]. С ростом компании и увеличением числа пользователей возникли проблемы с масштабируемостью существующего решения. В 2008 году Netflix принял решение перейти на микросервисную

[4] Netflix, Inc. — американская развлекательная компания, а также стриминговый сервис фильмов и сериалов. Основана 29 августа 1997 года Ридом Хастингсом и Марком Рэндольфом. Штаб-квартира находится в Лос-Гатосе, Калифорния.

архитектуру, начав разбиение монолитного приложения на независимые сервисы. Каждый из этих сервисов можно было масштабировать и обновлять независимо от других. Например, отдельными микросервисами стали системы управления пользователями, рекомендации фильмов, обработка платежей и другие функциональные компоненты.

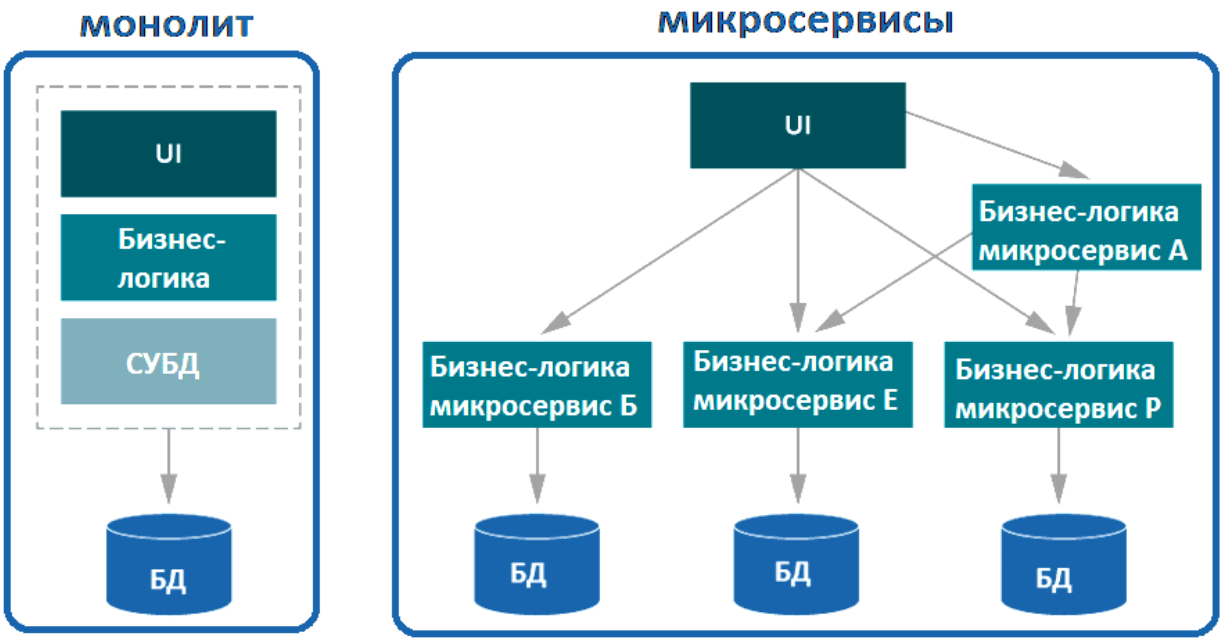

Рис. 2.7 – Модель микросервисной архитектуры в сравнении с монолитной

Ключевые различия SOA и микросервисной архитектуры:

Таблица 2.1

Различия между SOA и микросервисной архитектурой

Параметр	**Сервис-ориентированная архитектура (SOA)**	**Микросервисная архитектура**
Гранулярность сервисов	Сервисы могут быть достаточно крупными и выполнять сразу несколько бизнес-функций.	Сервисы небольшие, каждая отвечает за конкретную, узкую задачу.
Механизмы взаимодействия	Часто используются тяжеловесные протоколы и ESB для маршрутизации, трансформации и оркестрации сообщений.	Применяются легковесные протоколы (REST, gRPC) для прямого общения между сервисами без посредников

Развертывание	Сервисы могут быть взаимосвязаны, развертывание зачастую централизованное, что требует координации при обновлениях.	Каждый микросервис развёртывается и обновляется независимо, что упрощает непрерывную доставку.
Управление данными	Традиционно предполагается централизованный подход (общая база или централизованные источники данных), но это не строгое правило – в некоторых реализациях данные могут быть распределены.	Рекомендуется, чтобы каждый микросервис управлял своей собственной БД для автономности, однако на практике допускается использование общих хранилищ (если этого требует бизнес-логика).
Масштабируемость и гибкость	Хорошо подходит для крупных корпоративных систем, но масштабирование может зависеть от ESB.	Обеспечивает высокую гибкость и возможность масштабирования отдельных сервисов независимо друг от друга.
Технологическая независимость	Часто используется единый технологический стек для обеспечения совместимости между сервисами.	Позволяет разным командам выбирать наиболее подходящие технологии для каждого сервиса.

- **Бессерверная архитектура** (Serverless architecture) или бессерверные вычисления – это модель разработки и развертывания программного обеспечения, в которой разработчики фокусируются исключительно на написании кода, а управление инфраструктурой, масштабирование и обслуживание серверов полностью передаются облачному провайдеру. Хотя физические серверы остаются в распоряжении, их настройка и поддержка скрыты от разработчика. Это позволяет существенно сократить операционные затраты и ускорить выпуск продукта.

Несмотря на название «бессерверная», подход не означает, что серверов нет совсем, они по-прежнему используются для выполнения кода. Главное отличие заключается в том, что штатному сотруднику не нужно заниматься их

настройкой, обслуживанием или масштабированием – всё это автоматизировано и предоставляется как услуга. Такой подход позволяет существенно сократить операционные затраты, ускорить развертывание обновлений и сосредоточиться на бизнес-логике приложения.

Обычно применяется модель оплаты с провайдером услуг «pay-as-you-go», при которой оплачивается только фактическое время использования ресурсов, а не заранее зарезервированное мощности. Время простоя, как правило, не тарифицируется.

Однако возможны сценарии, когда компания может использоваться собственные мощности, например, из другого филиала.

Бессерверные вычисления – это «категория облачных услуг, в которой клиент может использовать различные типы облачных возможностей, не требуя от него выделения, развертывания и управления ни аппаратными, ни программными ресурсами, за исключением предоставления клиентского кода приложения или предоставления клиентских данных. Бессерверные вычисления представляют собой форму виртуализированных вычислений» (ISO/IEC 22123-2).

Бессерверные вычисления представляют собой широкую экосистему, которая включает облачного провайдера, функции как услуги [5], инструменты, фреймворки, инженеров, заинтересованные стороны и другие взаимосвязанные элементы [21].

Также существуют и гибридные архитектуры ПО, которые могут сочетать в себе несколько свойств, подходов из упомянутых выше. Безусловно, требования к архитектуре напрямую зависят от требований бизнес-логики, заказчика и заинтересованных сторон. В данной ВКР будут рассмотрены сценарии применения каждой из рассмотренных архитектур в производственной среде.

---

[5] **Функция как услуга** (англ. function-as-a-service, FaaS) — архитектурный шаблон, предполагающий возможность вызова экземпляра уффправляющего кода без необходимости управления серверами и серверным приложением; ключевой компонент бессерверных вычислений.

ГЛАВА 3. ВЛИЯНИЕ ТЕСТИРОВАНИЯ, CI/CD И DEVOPS НА КАЧЕСТВО РАЗРАБОТКИ ПО

3.1 Влияние тестирования на качество программного обеспечения

«Общеизвестно, что создать совершенное программное обеспечение невозможно. Поэтому прежде чем программное обеспечение будет передано пользователям, его необходимо протестировать, чтобы в производстве программного обеспечения снизить риск ошибок, оказывающих негативное влияние на его функционирование.»

(ГОСТ Р 56920-2016 «Программная инженерия. Тестирование программного обеспечения. Часть 1. Основные понятия и определения»)

Эта формулировка подчёркивает, что тестирование является фундаментальным элементом обеспечения качества ПО.

Сам процесс тестирования, в свою очередь, обычно подразделяется на **ручное** и **автоматизированное**.

Ручное тестирование позволяет детально изучить поведение системы, выявить тонкие моменты, которые автоматизированные системы могут пропустить, и обеспечить качественную обратную связь для разработчиков. При этом высокие показатели сложных метрик (например, по информационной прочности по метрике Чепина) указывают на потенциальные проблемы, требующие более глубокого анализа и, возможно, переработки архитектуры. Например, если тесты показывают, что отдельный модуль с высокой сложностью негативно влияет на взаимодействие с другими компонентами, это может служить основанием для применения микросервисной архитектуры или пересмотра внутренней структуры модуля.

Автоматизированное тестирование (автотестирование)

Автотестирование предполагает использование специализированных инструментов и скриптов для автоматической проверки функциональности, производительности и надежности программного продукта. Ключевые

преимущества автотестирования:

- Скорость и повторяемость: автоматизированные тесты могут выполняться быстро и регулярно, что особенно важно в условиях непрерывной интеграции и поставки (CI/CD (см. далее)).
- Уменьшение ручного труда: автотесты снижают зависимость от человеческого фактора, минимизируя вероятность пропуска ошибок.
- Непрерывная проверка качества: интеграция автотестов в процессы сборки и развертывания позволяет оперативно обнаруживать регрессии и нарушения в функциональности при внесении изменений в код.

Виды тестирования

В рамках данной ВКР основное внимание уделено наиболее распространённым и широко используемым методам тестирования программного обеспечения, поскольку они позволяют объективно оценивать качество программного продукта. Однако стоит отметить, что в современной практике разработки существует множество методов тестирования, охватывающих различные аспекты качества. Перечисленный ниже список видов тестирования не является исчерпывающим:

 Модульное тестирование (Unit Testing)

Проверка отдельных модулей или функций кода. Цель — убедиться, что каждый блок (функция, метод, класс) работает корректно в изоляции от остальных частей системы;

 Интеграционное тестирование (Integration Testing) позволяет оценить взаимодействие между модулями или компонентами системы. Здесь проверяется, корректно ли компоненты обмениваются данными и работают вместе после объединения;

 Системное тестирование (System Testing)

Тестирование полной системы как единого целого, включая проверку функциональности, производительности, безопасности и других нефункциональных требований;

 Регрессионное тестирование (Regression Testing)

Повторное тестирование системы после внесения изменений в код, чтобы убедиться, что новые изменения не вызвали непредвиденных ошибок в уже проверенных частях приложения;

 Приемочное тестирование (Acceptance Testing)

Тестирование, проводимое с участием конечных пользователей или заказчиков, для оценки соответствия продукта бизнес-требованиям и готовности к эксплуатации;

 Smoke-тестирование

Проверка основных функциональных возможностей приложения. Обычно выполняются быстро, поскольку цель таких тестов — убедиться, что основные возможности системы работают как запланировано

 Нагрузочное (Stress/Load Testing)

Этот вид тестирования направлен на оценку устойчивости системы при высоких нагрузках. Обычно нагрузочное тестирование выделяется в отдельную от автоматизированного и ручного тестирования область, так как имеет определённую специфику и требует от специалиста навыков использования других инструментов.

3.2 Влияние тестирования на архитектурные решения

- **Модульное тестирование** показывает, насколько изолирован и самодостаточен каждый компонент. Если тесты для одного модуля оказываются слишком сложными, то это может сигнализировать о необходимости разделения модуля на более мелкие компоненты. Такой подход способствует улучшению масштабируемости и сопровождаемости системы;
- **Интеграционное тестирование** выявляет проблемы взаимодействия между модулями, что может быть основанием для пересмотра архитектурного решения. Если интеграционные тесты показывают высокую сложность связи между компонентами, то целесообразно рассмотреть применение микросервисной архитектуры или использование сервисной шины (ESB) для более гибкого управления связями;

- **Автоматизация тестирования** способствует более быстрому циклу обратной связи, что позволяет оперативно реагировать на изменения в коде и архитектуре. Это особенно важно в условиях Agile и DevOps, где архитектура системы должна быть адаптивной и легко модифицируемой.

В системах, построенных по принципам микросервисов или с SOA архитектурой, автотестирование позволяет тестировать отдельные компоненты изолированно, что упрощает локализацию ошибок и способствует гибкости системы. Если, напротив, система имеет избыточную связанность между компонентами, автотестирование может выявлять сложности в изоляции функциональных блоков, что является сигналом для пересмотра архитектурного решения.

Таким образом, тестирование помогает не только поддерживать качество кода, но и служит критерием для оценки эффективности выбранной архитектуры.

И традиционные методы тестирования, и автотестирование напрямую влияют на общее качество программного обеспечения. Они позволяют:

- **Своевременно обнаруживать дефекты,** что сокращает затраты на их исправление;
- **Обеспечивать высокую надежность** системы за счёт постоянного контроля её работы;
- **Оптимизировать архитектуру** путём выявления проблемных модулей, которые можно переработать, разделить или заменить более простыми компонентами;
- **Снизить трудоемкость поддержки** и увеличить масштабируемость, благодаря возможности автоматизации проверки и быстрому реагированию на изменения.

3.3 Непрерывная интеграция, непрерывная доставка ПО (CI/CD), DevOps

DevOps и CI/CD существенно изменили способ создания и поддержки программных продуктов. Они не только позволяют автоматизировать сборку, тестирование и развертывание кода, но и создают основу для принятия архитектурных решений.

DevOps — это культурно-технический подход, объединяющий процессы разработки (Development), эксплуатации (Operations) и тестирования в единый, непрерывный цикл. Он призван обеспечить быструю, качественную и стабильную доставку программного продукта, минимизируя время между изменениями в коде и их внедрением в продуктивной среде.

Разработка (Development) и эксплуатация (Operations) продолжительное время были изолированными модулями. Код писали программисты, а системные администраторы отвечали за его развертывание и интеграцию. В рамках одного проекта специалисты работали отдельно, поскольку связь между двумя разрозненными хранилищами, серверами была ограничена. Однако такой подход был раскритикован из-за отсутствия гибкости. [20]

DevOps направлен на:

- **Ускорение разработки:** благодаря автоматизированной сборке, тестированию и развертыванию;
- **Повышение качества:** регулярное тестирование и мониторинг позволяют своевременно обнаруживать и исправлять дефекты;
- **Гибкость архитектуры:** быстрая обратная связь способствует оперативным изменениям и оптимизации структуры системы.

Рис. 3.1 – Отделы разработки ПО, которые объединяет DevOps

Рис. 3.2 – Процессы, которые объединяет DevOps

CI/CD: Непрерывная интеграция и непрерывное развертывание

CI (Continuous Integration)

Непрерывная интеграция подразумевает регулярное объединение изменений в общий репозиторий с автоматическим запуском сборки и тестов. Это помогает:

- Быстро обнаруживать дефекты;
- Снижать затраты на исправление ошибок;
- Гарантировать, что новые изменения не нарушают функциональность системы.

CD (Continuous Delivery / Continuous Deployment)

Непрерывная доставка автоматизирует подготовку к развертыванию, а непрерывное развертывание позволяет автоматически выпускать протестированные обновления в рабочую среду. Это обеспечивает:

- Быстрый выпуск новых версий продукта;
- Готовность к оперативным изменениям;
- Минимизацию времени между разработкой и эксплуатацией.

Практики CI/CD являются неотъемлемой частью культуры DevOps. Они способствуют:

- Быстрому циклу обратной связи, что позволяет оперативно реагировать на изменения,
- Автоматизации всех этапов разработки и тестирования, что уменьшает человеческий фактор,
- Поддержанию высокой гибкости и адаптивности архитектуры.

Влияние DevOps и CI/CD на качество ПО и архитектуру

Современные подходы DevOps и CI/CD напрямую влияют на архитектуру программного обеспечения и общее качество продукта:

- **Быстрая обратная связь и обнаружение дефектов:** автоматизированные тесты, интегрированные в CI/CD-процессы, позволяют своевременно выявлять ошибки, что существенно снижает затраты на их

исправление;

- **Оптимизация архитектуры:** результаты тестирования и количественные метрики помогают определить проблемные модули. Это, в свою очередь, может стать основанием для декомпозиции системы на независимые компоненты (например, переход на SOA/микросервисную архитектуру), что повышает масштабируемость и сопровождаемость;
- **Гибкость и адаптивность:** автоматизация процессов сборки, тестирования и развертывания (CI/CD) обеспечивает быструю адаптацию архитектуры под изменяющиеся требования бизнеса. Инфраструктура как код позволяет повторяемо и быстро развёртывать новые версии системы, снижая риск ошибок и увеличивая стабильность;
- **Повышение надежности:** Постоянный мониторинг и автоматическое тестирование способствуют повышению отказоустойчивости системы, что является ключевым фактором для масштабируемых и критически важных приложений.

ГЛАВА 4. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ АРХИТЕКТУРНЫХ РЕШЕНИЙ: КЕЙСЫ, МОДЕЛИ И РЕКОМЕНДАЦИИ

На основе составленной диаграммы Исикавы (рис. 4.1) в этой главе будут предложены рекомендации по обеспечению высокого качества ПО на архитектурном уровне.

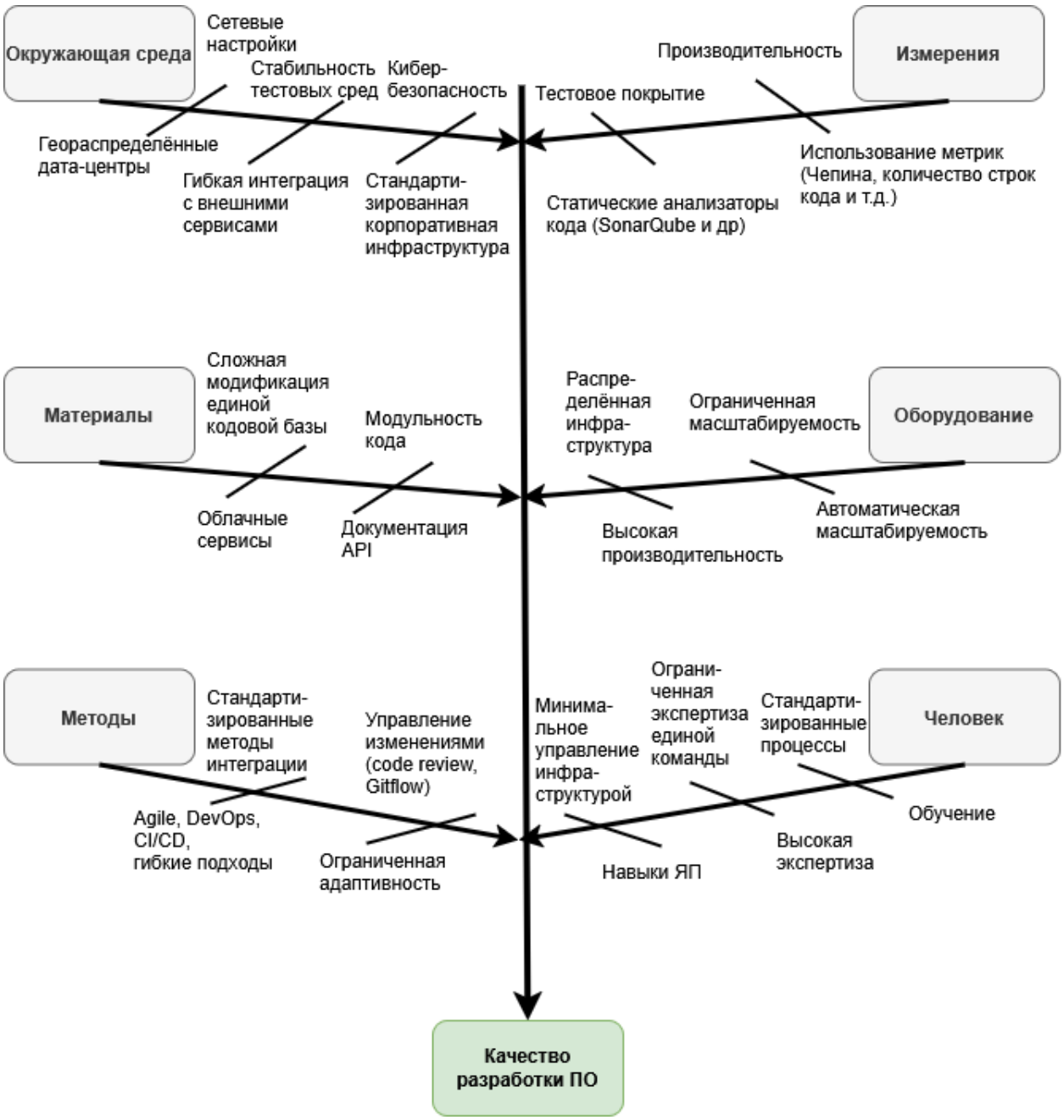

Рис. 4.1 – Диаграмма Исикава с главной проблемой – «Качество разработки ПО»

4.1 Модели использования архитектур ПО в производственной среде

Ниже мною были разработаны модели применения каждого из рассмотренных архитектурных подходов в производственной среде, выделены как преимущества каждого из них, так и недостатки, дана подробная характеристика.

4.1.1 Модель использования монолитной архитектуры

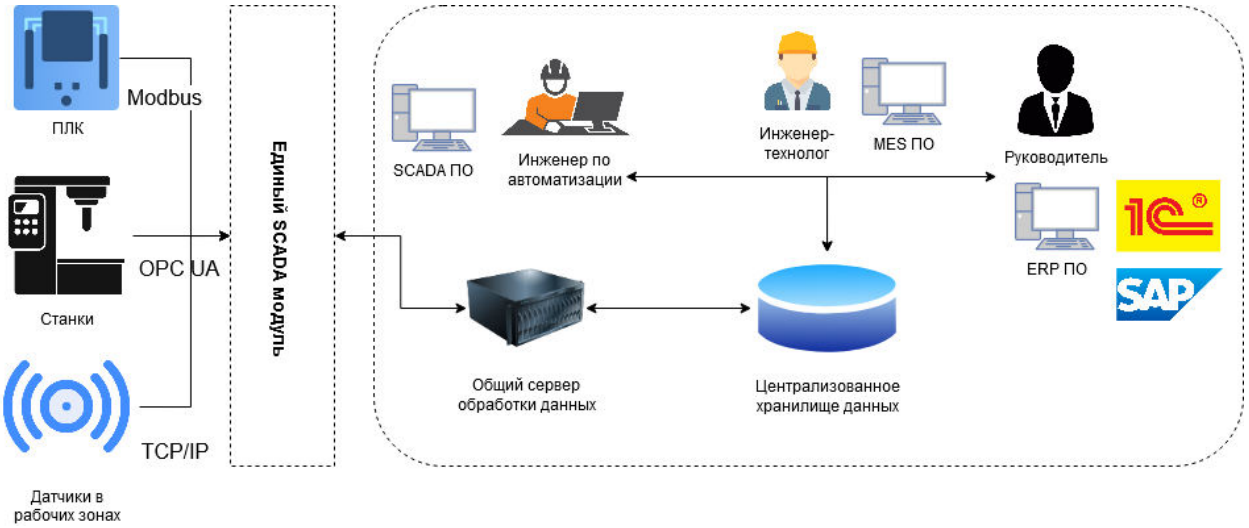

Рис. 4.2 – Модель использования монолитной архитектуры ПО на производстве

1. **ПЛК (программируемые логические контроллеры)** и станки передают данные через протоколы TCP/IP, Modbus, OPC UA в единый SCADA-модуль**;**
2. **SCADA-модуль** обрабатывает данные с датчиков в рабочих зонах и отправляет их в общий сервер обработки данных;
3. **Сервер обработки данных** взаимодействует с:

a. **Централизованным хранилищем данных** (ЦХД), где сохраняется информация о состоянии оборудования, параметрах производства и т.д;

b. **ERP ПО** для интеграции бизнес-процессов (учет заказов, логистика,

финансы);

c. **MES ПО** для управления производственными задачами.

4. **Инженер по автоматизации** и **инженер-технолог** взаимодействуют с системой через интерфейсы SCADA и MES.

5. **Руководитель** получает аналитику и отчеты через интерфейсы ERP и MES.

Производственные выгоды:

- **Простота развертывания:** монолитное ПО можно установить на один сервер или ПК оператора;
- **Низкие эксплуатационные затраты**: не требуется сложная инфраструктура, что снижает затраты на поддержку;
- **Централизованное управление**: все компоненты системы (драйверы, UI, БД) находятся в одном месте, что упрощает отладку и мониторинг.

Недостатки архитектуры:

- **Сложность масштабирования**: при увеличении числа станков или нагрузки на систему производительность может снижаться;
- **Риск каскадных сбоев**: ошибка в одном модуле (например, драйвере станка) может привести к падению БД;
- **Сложность тестирования**: тестирование отдельных компонентов затруднено из-за их тесной связанности.

Пример: для тестирования нового драйвера датчика пришлось запускать всю систему, так как модуль получения данных тесно связан с модулем составления отчётов.

4.1.2 Модель использования микросервисной архитектуры

Основная идея моей модели – это применение отдельных, «транспортных» компонентов ПО для взаимной передачи данных между более крупными MES, ERP-системами. Помимо этого, предполагается возможность их интеграции непосредственно со станками, промышленными роботами, ПЛК и др. Однако функция связующего звена между более крупными программными комплексами не является единственной, микросервисный подход также может использоваться как мера распределения нагрузки вычислительных мощностей. Более того, самостоятельный программный пакет из категории CALS-технологий может выступать и сам в качестве микросервиса при необходимой настройке.

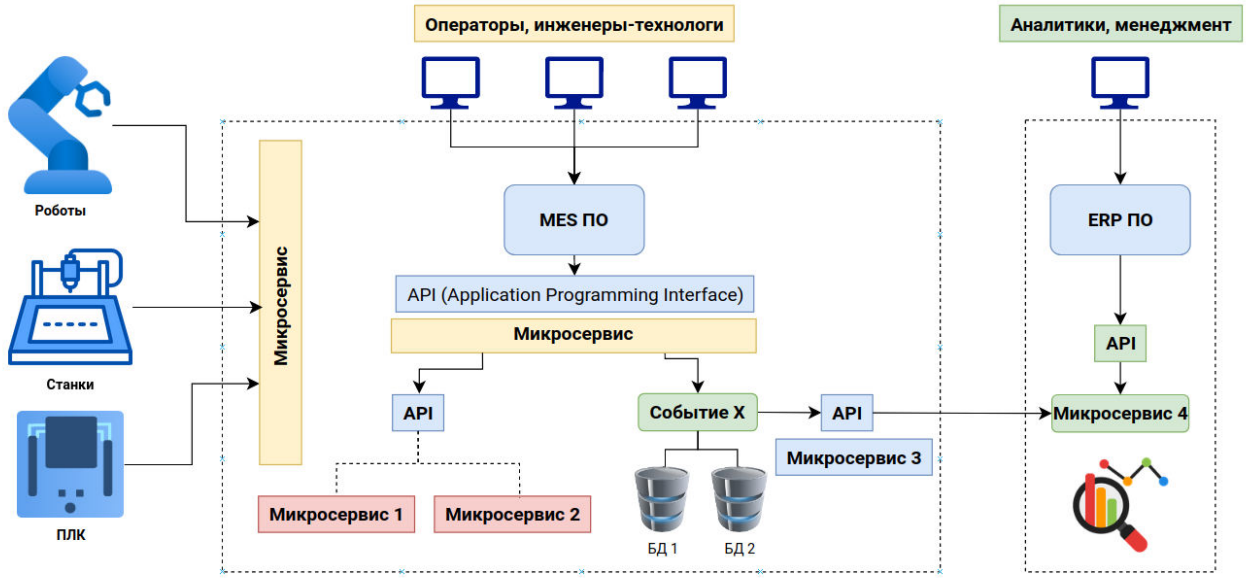

Рис. 4.3 – Модель использования микросервисной архитектуры ПО на производстве

Производственные выгоды:

- **Снижение времени простоя**: обновление одного сервиса не останавливает всю систему;
- **Возможность параллельной разработки**: команды работают над разными сервисами (например, аналитика и управление станками);

- **Изоляция сбоев**: ошибка в одном микросервисе (например, в ERP) не влияет на работу SCADA или MES;
- **Гибкость технологий**: разные сервисы могут использовать разные языки программирования и БД;

Недостатки архитектуры:

- **Высокие эксплуатационные затраты:** необходимы ресурсы для поддержки, мониторинга распределенной системы (серверы, сеть, DevOps-инженеры, инженеры внедрения и др.);
- **Сложность инфраструктуры**: требуется балансировка нагрузки и мониторинг множества сервисов.

4.1.3 Модель использования SOA (сервис-ориентированной архитектуры)

Основная идея – использование корпоративной сервисной шины (Enterprise Service Bus) для консолидации взаимодействия всех сервисов производственный среды. ESB обеспечивает маршрутизацию между различными системами (функции преобразования данных), предоставляет инструменты для мониторинга бизнес-процессов, интегрируется с базами данных.

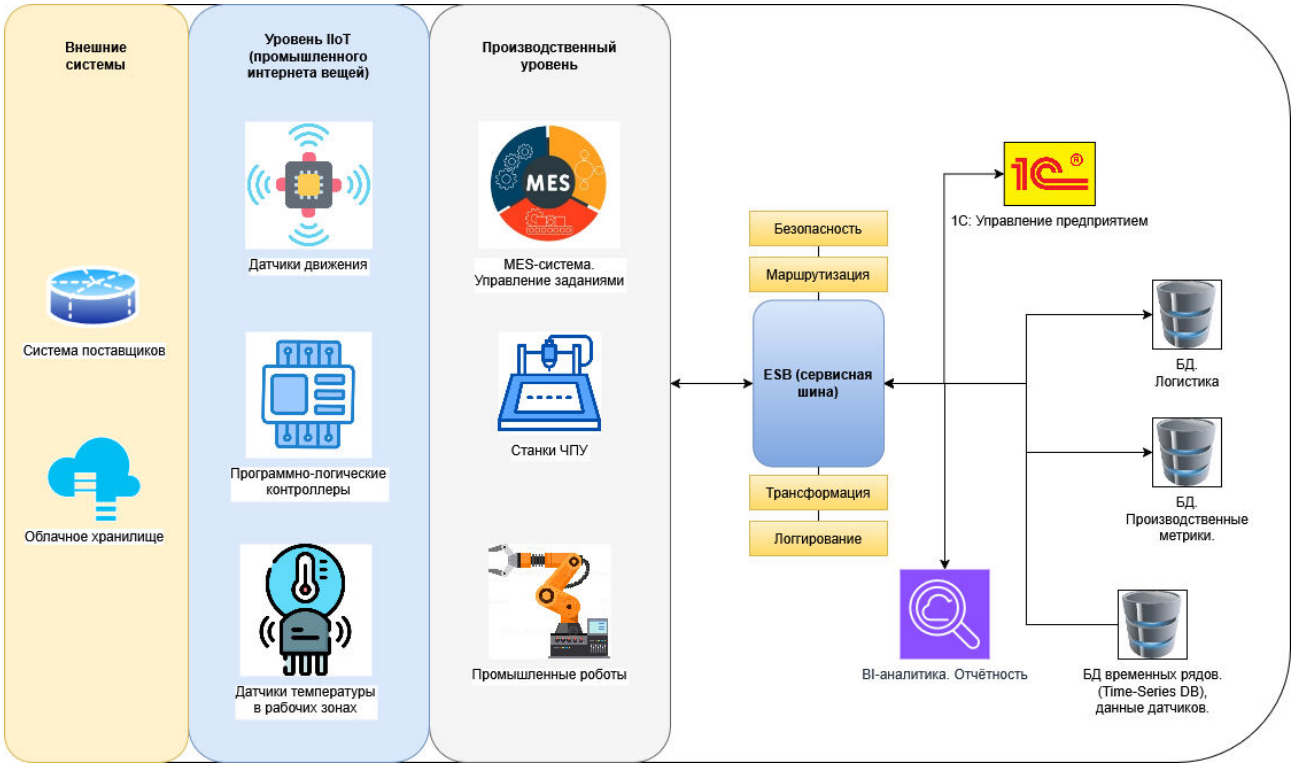

Рис. 4.4 – Модель использования SOA архитектуры ПО на производстве

Производственные выгоды:

- **Централизованное управление и мониторинг**: возможность централизованного мониторинга и контроля позволяет оперативно обнаруживать сбои и анализировать производительность на всех уровнях (бизнес, производство, IIoT и др).
- **Унификация взаимодействий между сервисами**:

централизованная маршрутизация упрощает адаптацию и трансформацию данных для обеспечения совместимости между системами с различными форматами обмена.

- **Повышение безопасности и управляемости данных**: сервисная шина позволяет внедрять комплексные механизмы безопасности (аутентификация, шифрование, контроль доступа) на единой платформе, что упрощает их администрирование

Недостатки архитектуры:

- **Ограниченная масштабируемость:** при значительном росте нагрузки на систему ESB может стать узким местом, так как все взаимодействия проходят через единую точку интеграции. Это приводит к сложности горизонтального масштабирования, что требует дополнительных инвестиций в оптимизацию и распределение вычислительных ресурсов;
- **Единая точка отказа:** ESB является критически важным элементом системы; её отказ или сбой может привести к параличу всех интегрированных процессов, что существенно увеличивает риск системных сбоев. Для снижения этого риска требуется внедрение дополнительных мер по резервированию и отказоустойчивости.

Комбинирование с бессерверной архитектурой возможно в любой из рассмотренных выше моделей, единственным различием будет частичное или полное использование вычислительных мощностей на стороне внешнего провайдера (например, Yandex Cloud).

4.2 Кейсы проблем на производстве из реального мира, связанные с ИТ-архитектурой

В этой главе будут рассмотрены случаи сбоев, инцидентов, которые непосредственно связаны с выстроенной архитектурой ПО, ИТ-инфраструктуры на производстве. Также будет сделан акцент на том, какие архитектурные решения могли бы быть применены, чтобы не допустить неблагоприятного исхода.

4.2.1 Toyota: сбой из-за архитектуры дискового пространства

28 августа 2023 года из-за переполнения дискового пространства база данных, отвечавшая за обработку заказов на детали, привела к остановке работы всех 14 заводов Toyota в Японии.

Официальный комментарий Toyota: «Сбой системы произошел из-за того, что некоторые из многочисленных серверов, обрабатывающих заказы на детали, стали недоступны. Что касается предыстории. 27 августа, за день до возникновения проблемы, проводились регулярные профилактические работы, в ходе которых мы удаляли и систематизировали данные, накопленные в базе данных. Однако из-за недостатка места на диске произошла ошибка, которая привела к остановке системы. Поскольку эти серверы работали в одной системе, аналогичный сбой также произошел в функции резервного копирования, переключение не удалось выполнить, что привело к остановке заводов. После этого 29 августа данные были перенесены на сервер с дисками большей емкости, система была восстановлена и работа заводов возобновилась».

Из этого кейса можно выделить следующие ошибки проектирования ИТ-архитектуры:

1. **Отсутствие сервисов мониторинга**, которые могли бы оповещать о критически низком свободном месте на дисках. В современных архитектурных решениях это решается через интегрированные системы мониторинга

(например, с использованием Prometheus, Zabbix, Grafana);

2. Прогнозирование и планирование ёмкости: система не была рассчитана на накопление больших объёмов данных. Возможно, стоило использовать отдельный микросервис прогнозирования для этой цели;

3. Интеграция резервного копирования и отказоустойчивость: неправильная настройка или отсутствие независимого резервного канала, что привело к тому, что отказ основного оборудования парализовал работу всей системы.

4.2.2 Toyota: использование инструмента качества «Пять почему» для выявления причин сбоя

Метод «Пять почему» — инструмент корневого анализа причин, разработанный основателем Toyota Сакити Тоёда.

Используем этот метод для выявления проблем сбоя на самих же заводах Toyota:

Шаг 1: Формулировка проблемы

Проблема: сбой дисковой системы на заводах Toyota.

Шаг 2: Задаем вопрос «Почему?»

1. **Почему произошёл сбой системы?**

– Потому что переполнилось дисковое пространство базы данных, что привело к отказу системы обработки заказов.

2. **Почему дисковое пространство оказалось переполненным?**

– Потому что данные накапливались без своевременной очистки и архивирования.

3. **Почему данные не очищались вовремя?**

– Потому что отсутствовала автоматизированная система мониторинга и процедуры управления объемом хранилища.

4. **Почему не была внедрена автоматизированная система мониторинга?**

– Потому что при проектировании системы не были учтены требования

к отказоустойчивости и прогнозированию емкости.

5. **Почему требования к отказоустойчивости и прогнозированию емкости не были учтены?**

– Потому что между ИТ-подразделением и производственными специалистами не была налажена эффективная обратная связь для анализа эксплуатационных рисков.

Таким образом, ошибка кроется не только в техническом недочёте (недостаток места), но и в **недостаточном анализе требований к масштабируемости и отказоустойчивости** на этапе проектирования архитектуры ИТ-системы. Более продуманная архитектура ПО включала бы автоматическое масштабирование хранилища, уведомление о снижении свободного пространства и независимые механизмы резервного копирования.

1. Архитектура Toyota до инцидента (28.08.2023)

Рис. 4.5 – Смоделированная ИТ-архитектура на момент сбоя Toyota в августе 2023

- **Моносистема обработки заказов**

Процесс: единая база данных, которая обрабатывала все заказы на детали. Не было разделения на независимые компоненты (например, сервисы для заказов, резервирования, аналитики).

Проблема:

Сбой в одном компоненте (например, переполнение диска) парализовал всю систему, так как все функции были жестко связаны.

- **Ручное управление данными**

Процесс: регулярные профилактические работы включали **ручное удаление данных** администраторами.

Проблема:

Человеческая ошибка: при удалении данных можно было случайно затронуть важные таблицы. Нет прогноза: не рассчитывалось, сколько места потребуется в будущем.

- **Интегрированное резервное копирование**

Процесс: резервные копии создавались на том же физическом хранилище, что и основная БД. Процесс резервирования зависел от доступности основного сервера.

Проблема: когда основной сервер вышел из строя из-за переполнения диска, резервная система тоже стала недоступна.

- **Отсутствие мониторинга**

Процесс: свободное место на дисках не отслеживалось в реальном времени. Администраторы узнавали о проблемах только после сбоя.

Проблема: нехватка места стала критической неожиданно, так как не было предупреждений.

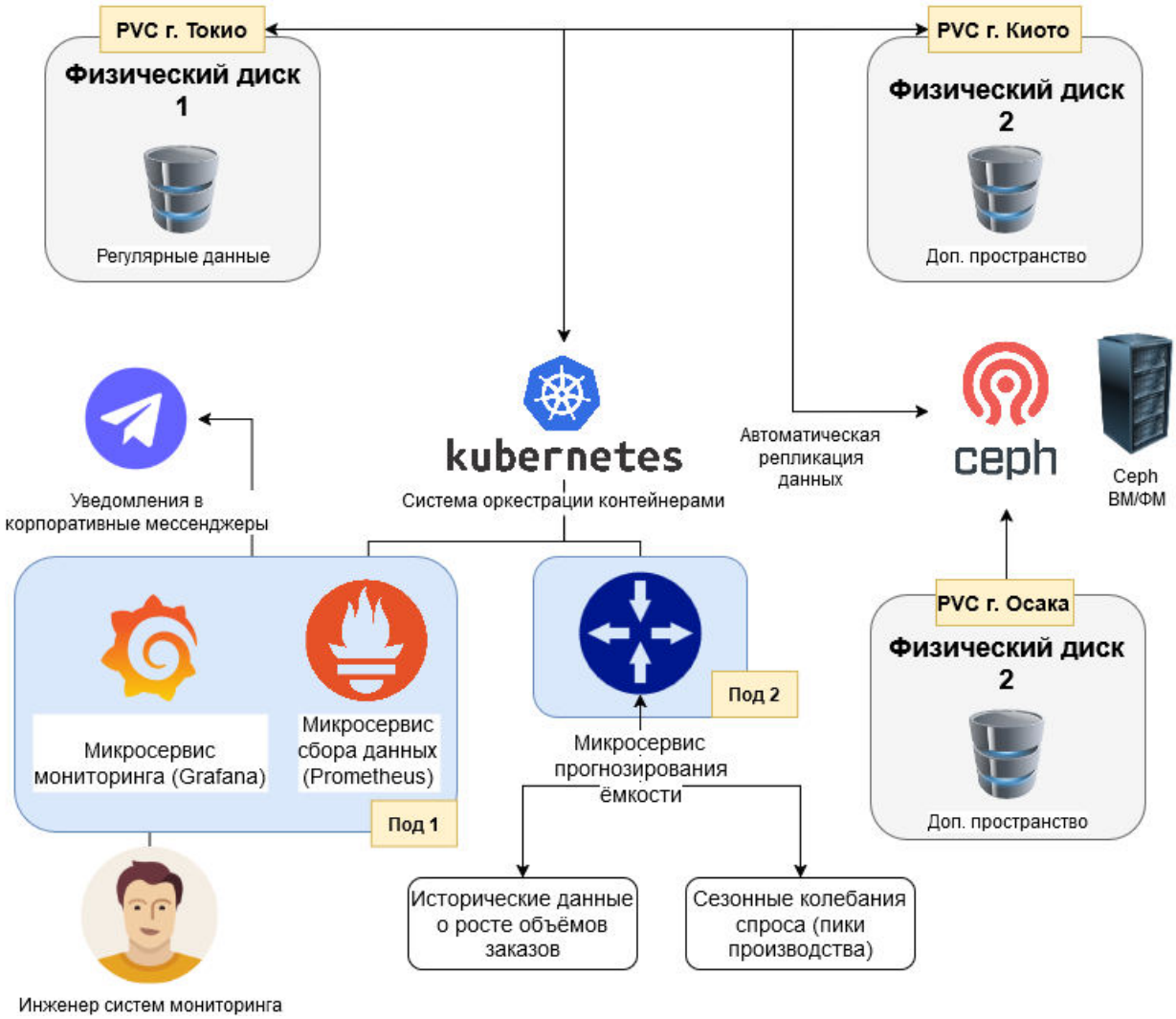

Рис. 4.6 – Предлагаемое решение, которое могло бы предотвратить сбой на заводах Toyota

2. Предложенное архитектурное решение на основе микросервисов

Автоматический мониторинг

Как работает:

1. Система (**Prometheus+Grafana**) отслеживает метрики:
2. Свободное место на дисках.
3. Нагрузку на CPU, RAM, сеть.
4. При достижении пороговых значений (например, 80% заполнения диска) отправляются алерты: на почту, в корпоративный мессенджер или любую иную систему оповещения.

Решает проблему:

Позволяет заранее увеличить дисковое пространство или очистить ненужные данные до критического сбоя.

Прогнозирование ёмкости и автомасштабирование

Как работает:

Микросервис прогнозирования анализирует:

- Исторические данные о росте объема заказов;
- Сезонные колебания спроса (например, пики производства);

На основе прогноза система автоматически выделяет дополнительные ресурсы.

Р**ешает проблему:**

Исключает ручные операции и человеческие ошибки. Система адаптируется под нагрузку.

Независимое резервное копирование

Как работает:

Резервные копии хранятся:

- В другом дата-центре (геораспределение).
- На отдельном физическом носителе, не связанном с основным сервером.

Используется **отказоустойчивый кластер**:

- Kubernetes[6] управляет подами с БД;
- Хранилище (например, **Ceph** (см. далее)**)** автоматически реплицирует данные между узлами.

Решает проблему:

Даже при сбое основного сервера резервная система остается работоспособной.

Разделение сервисов (микросервисы)

Как работает:

[6] **Kubernetes** (K8s) — открытое программное обеспечение для оркестровки контейнеризированных приложений — автоматизации их развёртывания, масштабирования и координации в условиях кластера. Поддерживает основные технологии контейнеризации, включая Docker, rkt, также возможна поддержка технологий аппаратной виртуализации.

Система разделена на независимые компоненты:

- Сервис заказов;
- Сервис резервирования деталей;
- Сервис аналитики;

Каждый сервис имеет свою БД и масштабируется отдельно.

Решает проблему:

Сбой в одном сервисе (например, переполнение диска в аналитике) не останавливает работу заказов.

Ceph – это масштабируемая и отказоустойчивая распределённая система хранения данных, которая объединяет возможности объектного, блочного и файлового хранения в единой программно-определяемой платформе. Основой архитектуры Ceph является RADOS (Reliable Autonomic Distributed Object Store), обеспечивающий автоматическое распределение, репликацию и самовосстановление данных в рамках кластера, что исключает наличие единой точки отказа. Такая архитектура позволяет системе обеспечивать непрерывный доступ к информации даже при сбоях отдельных узлов.

В предлагаемой архитектурной схеме Ceph играет ключевую роль за счёт следующих возможностей:

1. **Надёжное резервное копирование и восстановление данных:** Ceph обеспечивает создание дублированных копий информации, распределённых по различным физическим носителям, что гарантирует высокую отказоустойчивость и позволяет быстро восстановить данные при возникновении сбоев.

2. **Масштабируемость и гибкость хранения:** Благодаря децентрализованной архитектуре, Ceph способен автоматически масштабироваться по мере увеличения объёма данных. Это позволяет системе адаптироваться к росту требований бизнеса без значительных перебоев в работе.

3. **Интеграция с современными облачными и виртуализированными средами:** Ceph легко интегрируется с платформами оркестрации контейнеров (Kubernets, как выше), обеспечивает построение гибких, высокодоступных систем хранения данных. Это особенно важно для реализации резервного копирования и мониторинга производственных метрик в реальном времени.

4.2.3 НЛМК: переход с монолитной на микросервисную архитектуру

Далее будет разобран кейс российской сталелитейной компании в отрасли чёрной металлургии – НЛМК (Новолипецкий металлургический комбинат). В условиях роста объёмов обрабатываемых данных и усложнения производственных процессов компания столкнулась с проблемами, характерными для монолитных информационных систем, используемых для управления производством.

Исходно НЛМК использовала централизованное монолитное решение, построенное на базе Oracle и SQL, которое объединяло в себе все функциональные модули управления производственными процессами: от обработки заказов до интеграции с ERP-системами. Такая архитектура, несмотря на обладание преимуществами простоты развертывания и централизованного управления, в условиях расширения производства начала испытывать существенные ограничения. Растущий объём данных, увеличение количества подключаемых устройств и необходимость интеграции с современными системами приводили к ухудшению производительности, появлению задержек в обработке информации и повышению риска каскадных сбоев. В частности, отсутствие возможности горизонтального масштабирования и изоляции отказов отдельных модулей стало критическим фактором, негативно влияющим на оперативность принятия управленческих решений.

НЛМК столкнулись со следующими проблемами монолитной

архитектуры:

1. **Ограниченная масштабируемость.** Решение, построенное на базе Oracle и SQL, не способствовало эффективному распределению вычислительных ресурсов. При увеличении количества транзакций и подключаемых источников данных система не могла выдерживать возросшую нагрузку, что приводило к задержкам в обработке информации.

2. **Сложность обновления и интеграции.** Единая кодовая база монолитного решения существенно усложняла процесс внесения изменений и интеграцию новых технологий. Любые обновления требовали длительного тестирования и полного развёртывания всей системы, что замедляло адаптацию IT-системы к изменениям в требованиях производства.

3. **Неудобство независимого масштабирования функциональности.** Монолитная архитектура ограничивала возможности по независимому масштабированию отдельных функциональных модулей. По мере роста производства и расширения функционала возникала необходимость в разделении системы для оптимизации распределения ресурсов и повышения отказоустойчивости, что в рамках монолита реализовать было крайне затруднительно.

В связи с этим в НЛМК было принято решение поэтапного перехода на микросервисную архитектуру. На практике в НЛМК применяли два подхода при разделении монолитной части:

1. **Послойная миграция**: данный методологический подход (рис. 4.7) предусматривал разработку нового пользовательского интерфейса (фронтенда[7]) для определённой системы при сохранении существующей базы данных. Между новым интерфейсом и старой базой данных внедрялся промежуточный слой (middleware[8]), включающий компонент backend-for-frontend (BFF[9]), который адаптировал данные для взаимодействия с устаревшей системой. Этот подход был особенно эффективен в ситуациях, когда подразделения компании нуждались в немедленном доступе к обновлённому продукту.

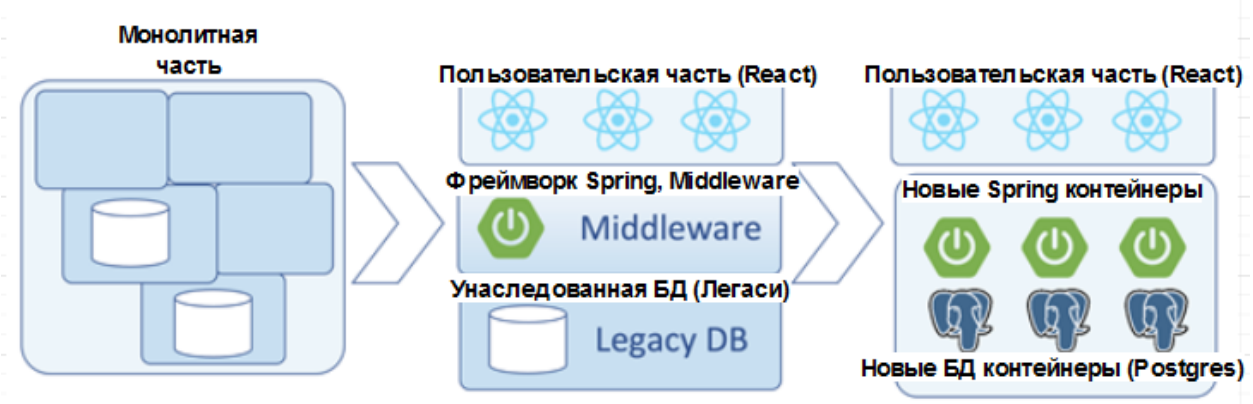

Рис. 4.7 – Подход «послойной миграции», использованный в НЛМК

2. **Функциональная миграция**: этот подход заключался в выделении конкретных функций из существующей системы и их преобразовании в автономные микросервисы. Основным вызовом при таком подходе являлась необходимость синхронизации, поскольку один и тот же процесс мог выполняться параллельно как в старой, так и в новой системах (в процессе перехода).

---

[7] **Фронтенд** (англ. front end, frontend) — презентационная часть информационной или программной системы, её пользовательский интерфейс и связанные с ним компоненты.

[8] **Мидлвар (Middleware)** — промежуточное программное обеспечение, которое обеспечивает взаимодействие между различными компонентами приложения или между различными приложениями

[9] **Backend-for-Frontend (BFF)** — архитектурный шаблон, при котором создаётся отдельный слой серверной части (бэкенда), специально предназначенный для конкретного фронтенда (клиентской части. BFF служит посредником между клиентским приложением и основными серверными сервисами, адаптируя данные и логику для нужд конкретного пользовательского интерфейса.

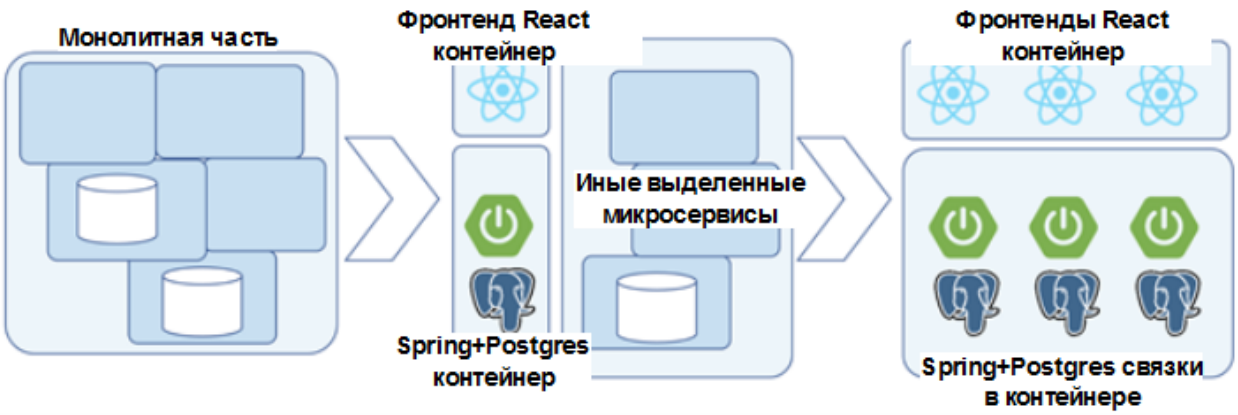

Рис. 4.8 – Подход «функциональной миграции», использованный в НЛМК

4.3 Рекомендации по применению архитектурных подходов

Монолитная архитектура

Подходит для небольших и средних компаний (от 10 до 50 разработчиков, а общее число сотрудников может быть от десятков до нескольких сотен), где бизнес-требования относительно стабильны, а система охватывает единое функциональное ядро.

Примеры целевой аудитории

- Стартапы или предприятия, где требуется быстрое развертывание и минимальные затраты на первоначальное проектирование;
- Производственные предприятия с ограниченным числом интегрируемых систем, например, локальные ERP-системы или SCADA, где сложные интеграционные задачи отсутствуют.

Обоснование

Монолитная архитектура обеспечивает быструю разработку и упрощённое тестирование, поскольку все компоненты собраны в одном приложении.

Сервис-ориентированная архитектура (SOA)

SOA целесообразна для средних и крупных предприятий, где требуется интеграция разнородных систем (ERP, MES, IoT) и повторное использование функциональных сервисов. Это решение подходит для организаций с ИТ-

командами от 50 человек, где существуют устаревшие системы и требуется централизованное управление интеграционными потоками.

Примеры целевой аудитории

Корпоративные структуры, банки, телекоммуникационные компании, крупные производственные предприятия, использующие интеграционные платформы (например, 1С:Шина или IBM Integration Bus) для обеспечения обмена данными между различными функциональными блоками.

Обоснование

SOA позволяет стандартизировать обмен данными через использование корпоративной сервисной шины (ESB), обеспечивая повторное использование сервисов и централизованный контроль над бизнес-процессами. Это подходит для систем, где критично поддержание единой политики безопасности и мониторинга. Однако требуемая настройка ESB и возможное появление единой точки отказа налагают дополнительные требования на управление и поддержку системы.

Микросервисная архитектура

Идеально подходит для крупных, динамично развивающихся организаций, где необходимо обеспечить высокую гибкость, масштабируемость и модульность системы. Такие предприятия часто имеют распределённые команды разработчиков и высокие требования к быстрому обновлению и расширению функционала.

Примеры целевой аудитории

Крупные интернет-компании (например, Netflix, Amazon), технологические компании, а также организации, которые в рамках перехода от монолита к более гибкому подходу (как в кейсе НЛМК, рассмотренном выше) уже демонстрируют рост объёма данных и число пользователей.

Обоснование

Микросервисы позволяют независимо разрабатывать, тестировать, развертывать и масштабировать отдельные компоненты, что существенно повышает отказоустойчивость и упрощает внедрение изменений. Такой подход

оправдан при динамичных бизнес-требованиях, однако требует более сложной оркестрации и инфраструктуры для управления распределённой системой.

Бессерверная архитектура (Serverless)

Подходит для малых и средних предприятий, а также для реализации отдельных функциональных модулей в рамках больших систем. Это оптимальное решение для компаний, где нагрузка на систему варьируется, а возможность автоматического масштабирования и экономия на оплате «по факту использования» являются приоритетом.

Примеры целевой аудитории

Стартапы, разработчики мобильных и веб-приложений, компании, стремящиеся быстро выйти на рынок без необходимости создавать и поддерживать инфраструктуру (например, с использованием Yandex Cloud, AWS Lambda, Google Cloud Functions). Также подходит для реализации событийно-ориентированных сценариев, например, обработки данных от IoT-устройств.

Обоснование

Бессерверный подход позволяет сосредоточиться исключительно на реализации бизнес-логики, оставляя управление инфраструктурой и масштабирование облачному провайдеру. Это существенно снижает операционные затраты и ускоряет выпуск обновлений, хотя для сложных или критически нагруженных систем могут возникать ограничения, связанные с задержками или спецификой выполнения кода в облачной среде.

4.4 Матрица выбора решения архитектурных подходов

Таблица 4.1

Матрица выбора решения рассмотренных архитектурных подходов

Критерий	**Монолитная архитектура**	**Сервис-ориентированная архитектура (SOA)**	**Микросервисная архитектура**	**Бессерверная архитектура**
Гибкость	Низкая, сложно изменять без затрагивания всей системы.	Средняя, сервисы независимы, но зависят от ESB.	Высокая, сервисы автономны и могут разрабатываться независимо.	Высокая, функции легко адаптируются под изменения.
Масштабируемость	Ограниченная, горизонтальное масштабирование затруднено	Выше, но сервисная шина может стать узким местом	Высокая, каждый сервис масштабируется отдельно	Очень высокая, масштабируемость автоматизирована провайдером
Отказоустойчивость	Низкая, сбой в одном компоненте может привести к падению всей системы	Средняя, ESB обеспечивает устойчивость, но сам может стать точкой отказа	Высокая, отказ одного сервиса не влияет на другие	Очень высокая, так как инфраструктура управляется облаком
Совместимость	Ограниченная, интеграция с внешними системами требует доработок	Высокая, ESB обеспечивает взаимодействие между разными системами	Средняя, сервисы могут использовать разные технологии	Высокая, возможность интеграции с различными API
Производительность	Высокая, но ограничена мощностью одного сервера	Средняя, наличие ESB может снижать скорость обработки	Высокая, за счет распределенной обработки	Зависит от облачного провайдера, возможны задержки

Затраты на поддержку	Высокие, сложность обновлений и исправлений	Средние, но требуют специалистов по ESB	Средние, но требуют опыта работы с распределенными системами	Низкие, провайдер берет на себя поддержку инфраструктуры

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы разработаны модели применения монолитной, микросервисной и сервис-ориентированной архитектур программного обеспечения для производственных систем.

1. Выполнен анализ существующих методов и средств оценки качества программного обеспечения. Разработанные решения соответствуют задачам цифровой трансформации обрабатывающих отраслей промышленности, обозначенным в распоряжении Правительства РФ № 3113-р от 07.11.2023. Проведённый анализ критериев систем из ГОСТ Р ИСО 25010-2015 (характеристики качества ПО) и ГОСТ Р 57100-2016 (требования к архитектуре ПО), позволил выявить ключевые зависимости между выбором архитектурного подхода и такими параметрами, как надёжность, масштабируемость и сопровождаемость систем.

2. Исследовано влияние современных архитектурных подходов, применения шаблонов разработки, методов тестирования на качество программного обеспечения. Кейсы промышленных предприятий (TOYOTA, НЛМК), связанные с проблемами ИТ-архитектуры, продемонстрировали, как переход от монолитных решений к модульным архитектурам (микросервисы, SOA) снижает риски каскадных сбоев и упрощает интеграцию новых технологий.

3. Разработаны рекомендации по применению архитектурных подходов и инструментов в разработке программного обеспечения. В заключительной части ВКР были разработаны рекомендации по применению каждой из рассмотренных архитектур ПО согласно размеру предприятия, целевой аудитории с обоснованием выбора. Была разработана матрица выбора решения для архитектур по ключевым характеристикам ГОСТ Р ИСО 25010-2015, ГОСТ Р 57100-2016.

СПИСОК ЛИТЕРАТУРЫ

Нормативные документы

1. ГОСТ Р ИСО 25010-2015. Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов; введён 01.06.2016.
2. ГОСТ Р 57100-2016/ISO/IEC/IEEE 42010:2011. Системная и программная инженерия. Описание архитектуры. Systems and software engineering. Architecture description; введён 01.09.2017.
3. ГОСТ Р 56920-2016/ISO/IEC/IEEE 29119-1:2013. Системная и программная инженерия. Тестирование программного обеспечения; введён 01.06.2017.
4. ISO/IEC 22123-2:2023. Information technology. Cloud computing. Part 2: Concepts; введён 22.09.2023.

Книги и статьи

5. Решетникова Е. В. Метрология и качество программного обеспечения. – Новокузнецк: Кемеровский государственный университет, 2020. – 80 с. URL: https://skado.khpi.ru/indicationsvkr/2614/ (дата обращения: 14.12.2024).
6. Vallecillo A. RM-ODP: The ISO Reference Model for Open Distributed Processing. – Málaga: Universidad de Málaga, 2004. – 7 с. URL: https://www2.isye.gatech.edu/~lfm/8851/Sources/RefModels/RM-ODP.pdf (дата обращения: 05.02.2025).
7. Nguyen T. V., Deeds-Rubin S. A SLOC Counting Standard. – Los Angeles: University of Southern California, 2007. – 16 с.

8. Martucci M., Dib K. Architecting Frameworks for Specific Applications with RM-ODP. – São Paulo: University of São Paulo, 2004. – с. 3–5.
9. Brisals S., Sheen S. Serverless Development on AWS: Building Enterprise-Scale Serverless Solutions. – Seattle: O'Reilly Media, 2024. – 498 с.

Интернет-ресурсы

10. «Microservices Adoption in 2020. Everyone's talking about microservices. Who's actually doing it?» // O'Reilly, 2020. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (дата обращения: 30.11.2024).
11. «Сравнение микросервисной и монолитной архитектур» // Atlassian, 2020. URL: https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith (дата обращения: 30.11.2024).
12. «26 основных паттернов микросервисной разработки» // VK Cloud, 2021. URL: https://cloud.vk.com/blog/26-osnovnyh-patternov-mikroservisnoj-razrabotki/ (дата обращения: 30.11.2024).
13. «Concerning the production order system malfunction» // Toyota. URL: https://global.toyota/en/newsroom/corporate/39732568.html (дата обращения: 02.02.2025).
14. «ESB (enterprise service bus): назначение, функционал, новые подходы к развитию» // Комсомольская правда. Новые технологии. URL: https://www.kp.ru/guide/esb.html (дата обращения: 02.02.2025).
15. «ESB: как сервисная шина предприятия упрощает обмен данными в крупных ИТ-проектах» // Корус Консалтинг. URL: https://omni.korusconsulting.ru/blog/esb-kak-servisnaya-shina-predpriyatiya-uproshchaet-obmen-dannymi-v-krupnykh-it-proektakh-/ (дата обращения: 02.02.2025).

16. «От монолита к микросервисам – как металлурги переходят с Oracle и SQL на Java-стек» // НЛМК. URL: https://career.nlmk.com/media/stati/ot-monolita-k-mikroservisam-kak-metallurgi-perekhodyat-s-oracle-i-sql-na-java-stek/ (дата обращения: 04.02.2025).
17. «Configure a Pod to Use a PersistentVolume for Storage» // Kubernetes документация, 2025. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/ (дата обращения: 06.02.2025).
18. «Знакомство с хранилищем Ceph в картинках» // Хабр, 2017. URL: https://habr.com/ru/articles/313644/ (дата обращения: 06.02.2025).
19. Милютин А. Метрики кода программного обеспечения. – 2009. – URL: https://pvs-studio.ru/ru/blog/posts/a0045/ (дата обращения: 08.02.2025).
20. Питтет С. «Различные виды тестирования ПО» // Atlassian, 2021. URL: https://www.atlassian.com/ru/continuous-delivery/software-testing/types-of-software-testing (дата обращения: 08.02.2025).
21. Габрух Н. Основы методологии DevOps // Proglib, 2021. – URL: https://proglib.io/p/osnovy-metodologii-devops-2021-02-20 (дата обращения: 08.02.2025).
22. «Искусство решения любой задачи: метод «Пять почему» // aQsi. URL: https://aqsi.ru/iskusstvo-resheniya-lyuboy-zadachi-metod-pyat-pochemu/ (дата обращения: 19.03.2025).
23. «Об утверждении стратегического направления в области цифровой трансформации обрабатывающих отраслей промышленности»: распоряжение от 07.11.2023 № 3113-р // Правительство Российской Федерации, 2023. URL: http://static.government.ru/media/files/OwFdjc3nMWk3BqAUbjqdJImPl3NxqRIS.pdf (дата обращения: 06.05.2025).