МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
вычислительной техники	МОиПЭВМ

Направление подготовки 09.03.02 «Информационные системы и технологии»

Направленность Информационные системы и технологии

БАКАЛАВРСКАЯ РАБОТА

на тему

Разработка серверной архитектуры и системы телеметрии для многопользовательской игры

Студент _____ Кубарев Вадим Сергеевич
(подпись, дата) (ФИО полностью)

Руководитель _____ Попова Н.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026 № 12)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____№____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____А.Ю. Козлов
«26» декабря 2025 г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

БАКАЛАВРА

1. Студент Кубарев Вадим Сергеевич гр. 22ВИ1 факультета ВТ
<Фамилия, имя, отчество полностью>

2.Тема работы Разработка серверной архитектуры и системы телеметрии для многопользовательской игры

Тема утверждена приказом ПГУ №1583 от "30" декабря 2025 г.

2. Руководитель работы

3. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Назначение: разработка серверной архитектуры, обеспечивающей поддержку многопользовательской игры, и системы телеметрии для сбора, хранения и анализа игровых событий в реальном времени

2) Среда выполнения: Windows 11

Языковые средства разработки: C#, JavaScript, HTML, CSS, SQL, UML

Инструментальные средства: Visual Studio, Visual Code, Visual Paradigm, React, Vite, ASP.NET Core, Dapper, PostgreSQL, ClickHouse, Unity, Fishnet, Valkey

Технологии разработки: ООП

3) Основные функции разрабатываемого приложения:

a) Управление жизненным циклом игровых серверов
b) Сбор и передача телеметрических событий
c) Аутентификация и управление учетными записями игроков
d) Хранения и агрегация телеметрии
e) Визуализация аналитики
f) Кэширование временных данных

4. Перечень подлежащих разработке вопросов

1) Анализ предметной области и планирование процесса разработки;
2) Анализ требований и постановка задачи на разработку;
3) Разработка;
4) Контроль качества программного обеспечения;
5) Оформление пояснительной записки.

5. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	07.04.2026 – 13.04.2026	
2	Проектирование программных средств	20%	14.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	27.04.2026 – 11.05.2026	
4	Тестирование программных средств	20%	12.05.2026 – 21.05.2026	
5	Оформление пояснительной записки	20%	22.05.2026 – 02.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы _____ 26.12.2025.
(подпись, дата)

Задание к исполнению принял студент _____ 26.12.2025г.
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 90 листов, 20 рисунков, 6 таблиц, 23 использованных источника и 2 приложения.

Серверная архитектура, система телеметрии, многопользовательская игра, ASP.NET Core, Clickhouse, PostgreSQL, RabbitMQ, Unity, Fishnet, React, Vite, Docker, C#, JavaScript, UML.

Объект исследования – серверная архитектура и система телеметрии многопользовательской игры.

Цель выпускной квалификационной работы – разработать серверную архитектуру и систему телеметрии для многопользовательской игры, обеспечивающие высокую производительность, масштабируемость и сбор аналитических данных в реальном времени.

Технологии разработки – разработка серверной части велась на языке программирования C# с использованием фреймворка ASP.NET Core, микро-ORM Dapper, брокера сообщений RabbitMQ и систем управления базами данных PostgreSQL и Clickhouse. Клиентская часть игры реализована на платформе Unity с сетевым движком Fishnet. Веб-панель телеметрии разработана на JavaScript с использованием библиотеки React и сборщика Vite. Контейнеризация части компонентов выполнена с помощью Docker. В качестве среды разработки использовались Visual Studio и Visual Code.

Среда исполнения – Windows 11, Linux, Веб-браузер.

Результаты работы – создана серверная архитектура, включающая менеджер игровых серверов, API-шлюз, конвейер сбора телеметрии, а также веб-панель визуализации аналитики.

Все требования к разработке приложения соблюдены.

Содержание

Введение 7

1 Анализ предметной области и требований к системе 9

1.1 Постановка задачи 9

1.2 Анализ архитектурных подходов к организации серверной инфраструктуры и сбора телеметрии 10

1.3 Выбор технологий проектирования системы 12

1.4 Выбор инструментальных средств разработки 14

1.5 Анализ требований к серверной архитектуре и системе телеметрии 15

1.6 Планирование процесса разработки и бюджета 16

2 Проектирование программных средств 18

2.1 Проектирование менеджера игровых серверов и жизненного цикла процессов 18

2.2 Проектирование схемы базы данных телеметрии 20

2.3 Проектирование реляционной базы данных 23

2.4 Проектирование системы кэширования 25

2.5 Проектирование API-шлюза 27

2.6 Проектирование веб-панели телеметрии 30

2.7 Архитектура развертывания системы 32

3 Реализация программных средств 34

3.1 Реализация менеджера игровых серверов 34

3.2 Реализация игрового сервера и отправки телеметрии 36

3.3 Реализация API-шлюза 38

3.4 Интеграция брокера сообщений и конвейера загрузки данных 39

3.5 Реализация хранения пользовательских данных 41
3.6 Реализация системы кэширования 41
3.7 Реализация веб-панели телеметрии 43
4. Тестирование 48
4.1 Тестирование методом черного ящика 48
4.2 Классы эквивалентности 48
4.3 Граничные условия 49
4.4 Модульное тестирование (Unit-тесты) 50
4.5 Анализ качества кода 50
Заключение 55
Список использованных источников 57
Приложение А (справочное) Глоссарий 59
Приложение Б (обязательное) Листинг программы 66

Введение

Современная игровая индустрия находится в этапе стремительного роста многопользовательских проектов, которые требуют не только увлекательного игрового процесса, но и надёжной и масштабируемой серверной архитектуры. Успех таких проектов напрямую зависит от способности серверной инфраструктуры выдерживать нагрузки, обеспечивая минимальную задержку между действиями и синхронизируя состояния всех игроков в реальном времени [1]. В условиях, когда количество одновременных пользователей может варьироваться от нескольких десятков до миллионов, традиционные монолитные решения демонстрируют свою неэффективность, уступая место распределённым системам с горизонтальным масштабированием. Кроме того, для поддержания игрового баланса, выявления аномального поведения игроков и принятия обоснованных управленческих решений критически важным инструментом становится сбор телеметрии о каждом действии игрока, состоянии серверов и эффективности игровых механик. Без централизованной системы наблюдения даже незначительные сбои в работе серверов или дисбаланс игровых параметров могут остаться незамеченными, что приведёт к уходу аудитории и экономическим потерям.

Цель выпускной квалификационной работы состоит в разработке серверной архитектуры и системы телеметрии для многопользовательской игры. Выбранное направление является актуальным, поскольку в современных многопользовательских онлайн проектах возникают задачи обеспечения высокой производительности, отказоустойчивости и масштабируемости серверной инфраструктуры, а также оперативного сбора и анализа телеметрических данных для поддержания игрового баланса и выявления аномального поведения игроков. Особую значимость этим задачам придаёт тот факт, что игровая индустрия предъявляет одни из самых жёстких требований к времени отклика и к плотности событий, генерируемых каждым подключённым клиентом. Решение этих задач позволяет повысить наблюдаемость системы,

качество игрового процесса и эффективность управления серверными ресурсами, что в конечном счёте определяет коммерческий успех проекта.

Для достижения поставленной цели решались следующие задачи:

– провести анализ предметной области, выделить требования к серверной архитектуре и системе телеметрии для многопользовательских игр;

– выполнить сравнительный анализ существующих технологий и обосновать выбор оптимального стека;

– выбрать технологию проектирования и разработать диаграмму вариантов использования, детализирующую роли участников и их взаимодействие с системой;

– сформулировать детальный набор функциональных и нефункциональных требований к разрабатываемой системе, включая требования к отказоустойчивости, согласованности данных и безопасности игровых сессий.

В результате выполнения работы спроектирована и реализована серверная архитектура, включающая в себя API–шлюз, обеспечивающий маршрутизацию и первичную аутентификацию запросов, менеджер игровых серверов, ответственный за динамическое выделение ресурсов под новые игровые сессии, распределённые игровые процессы, реляционную и колоночную базы данных, кэш-хранилище, сокращающее время отклика при доступе к часто запрашиваемым данным, а также веб-панель телеметрии с визуализацией ключевых метрик в реальном времени.

Разработанные программные средства могут быть использованы в составе коммерческого игрового проекта для мониторинга состояния серверов и сетевых соединений, анализа игрового баланса и выявления аномального поведения игроков на основе их действий. В настоящий момент создан рабочий прототип. В дальнейшем планируется развивать систему в направлении поддержки оркестрации контейнеров для автоматического масштабирования, а также расширения набора аналитических метрик.

1 Анализ предметной области и требований к системе

1.1 Постановка задачи

Современные многопользовательские проекты предъявляют высокие требования к серверной инфраструктуре. Игровые серверы должны обрабатывать десятки тысяч обновлений в секунду с минимальной задержкой, поддерживать синхронное состояние десятков или сотен игроков в реальном времени. Ключевыми характеристиками такой архитектуры являются производительность, отказоустойчивость и масштабируемость.

Важным элементом успешного многопользовательского проекта является телеметрия, сбор детализированных событий о действиях игроков, состояния серверов, баланса оружия и карт, позволяет принимать обоснованные решения как для изменения игровых параметров для обеспечения справедливого игрового процесса, так и планирования масштабирования инфраструктуры.

Предметной областью данной работы являются процессы взаимодействия клиентских приложений с серверной стороной, а также процессы сбора, передачи, хранения и визуализации телеметрических событий. Для практической реализации используется прототип многопользовательской игры на платформе Unity, однако на этапе анализа рассматриваются общие принципы, применимые к широкому классу проектов [2].

При анализе предметной области были выделены следующие ключевые проблемы, которые необходимо решить:

– динамическое управление количеством серверов в зависимости от нагрузки;

– гарантированный сбор и доставка телеметрических событий без потерь;

– эффективное долговременное хранение больших объёмов событий с возможностью быстрого выполнения агрегирующих запросов;

– визуализация аналитики в реальном времени для администраторов и аналитиков.

Исходя из этого была сформулирована цель работы: разработка серверной архитектуры и системы телеметрии, которая обеспечит автоматическое масштабирование игровых серверов, надёжный сбор телеметрии и оперативное предоставление аналитики через веб-панель.

1.2 Анализ архитектурных подходов к организации серверной инфраструктуры и сбора телеметрии

Для построения серверной архитектуры многопользовательской игры могут быть применены различные архитектурные стратегии. Выбор конкретного подхода определяет, как будет организовано сетевое взаимодействие, управление игровыми процессами и сбор телеметрии. В рамках данного раздела проводится анализ четырёх основных стратегий.

Монолитная архитектура с одним игровым сервером. Все игроки подключаются к одному серверному процессу. Данный подход прост в реализации, но имеет жёсткие ограничения по масштабируемости. Максимальное количество игроков ограничено мощностью одного сервера, а отказ такого сервера приводит к недоступности всей игры. Для современных онлайн-проектов этот вариант неприемлем.

Архитектура с фиксированным набором игровых серверов. Игроки распределяются по нескольким независимым серверам. Каждый сервер обслуживает свою группу игроков. Этот подход позволяет горизонтально масштабироваться, но не динамически. Количество серверов фиксировано или изменяется только вручную. Также отсутствует централизованный сбор телеметрии, каждый сервер хранит свои логи, что затрудняет аналитику.

Архитектура с оркестрацией игровых серверов под управлением менеджера. Выделяется компонент «менеджер серверов», который автоматически запускает и останавливает игровые процессы в зависимости от текущей нагрузки. При увеличении числа игроков менеджер создаёт новые экземпляры, при снижении – останавливает лишние. Это позволяет оптимально использовать ресурсы. Телеметрия собирается с каждого сервера и отправляется

в централизованное хранилище. Данный подход является наиболее гибким и экономичным.

Архитектура на основе контейнерной оркестрации. Крайний уровень автоматизации, когда игровые серверы упаковываются в контейнеры и управляются системой оркестрации. Этот подход обеспечивает высокую отказоустойчивость и удобство масштабирования, однако требует значительных затрат на инфраструктуру и администрирование. Для небольшого проекта может быть избыточен. На таблице 1 указаны краткие сведения по каждой архитектуре.

Таблица 1 – Сравнение архитектурных стратегий организации серверной инфраструктуры

Критерий сравнения	**Монолитная**	**Фиксированные наборы серверов**	**Менеджер серверов**	**Контейнерная оркестрация**
Масштабируемость	Отсутствует	Ручная	Автоматическая	Автоматическая
Отказоустойчивость	Низкая	Средняя	Высокая	Очень высокая
Сложность реализации	Низкая	Средняя	Средняя	Высокая
Сложность администрирования	Низкая	Средняя	Средняя	Высокая
Возможность централизованной телеметрии	Нет	Частично	Да	Да

Для данной работы выбран подход с менеджером серверов. Он обеспечивает нужную автоматизацию масштабирования и централизованный сбор телеметрии при разумной сложности реализации. Контейнерная оркестрация рассматривается как перспективное направление развития.

Для сбора телеметрии также возможны различные стратегии: синхронная отправка событий напрямую в базу данных, синхронная отправка через API-шлюз, асинхронная через брокера сообщений. Асинхронная стратегия с использованием брокера позволяет гарантировать доставку событий даже при перегрузке или временной недоступности хранилища, а также развязывает игровые серверы и систему аналитики. Поэтому она выбрана в качестве основной.

1.3 Выбор технологий проектирования системы

Проектирование архитектуры любого программного продукта начинается с создания визуальных моделей, которые позволяют структурировать логику работы системы. Для этих целей применяется графический язык UML [3]. Он является международным стандартом и позволяет с помощью набора универсальных диаграмм описать любые процессы и связи внутри системы, делая проект понятным для всех участников разработки.

UML охватывает 14 типов диаграмм, которые делятся на две крупные группы: структурные и поведенческие. Такой подход позволяет разработчику, аналитику или заказчику на любом этапе жизненного цикла программы, от анализа требований до тестирования и сопровождения, получить наглядное представление о том, как система будет функционировать и из каких модулей состоять. Без визуального моделирования сложные проекты быстро обрастают неявными зависимостями, а документация теряет актуальность, что в итоге ведёт к росту технического долга и срыву сроков.

В процессе подготовки работы был выполнен обзор инструментов для построения UML-диаграмм. Рассматривались ArgoUML, StarUML и Visual Paradigm. ArgoUML, несмотря на свою бесплатность и открытый исходный код, был признан устаревшим, он не получает обновлений с 2011 года и страдает от нестабильной работы на современных операционных системах. StarUML показал себя как лёгкое и быстрое решение с приятным интерфейсом и поддержкой плагинов, но его бесплатная версия сильно ограничена по функционалу. В качестве основного средства проектирования выбрана среда Visual Paradigm как профессиональное CASE-средство, предоставляющее полный набор функций для графического анализа и моделирования, удобный интерфейс и качественный экспорт изображений. Дополнительным выбран Draw.io для описания специфичных диаграмм.

В рамках проектирования разработана диаграмма вариантов использования, которая отображает функциональные возможности системы и

взаимодействие с пользователями. На рисунке 1 представлена диаграмма вариантов использования для серверной архитектуры и системы телеметрии.

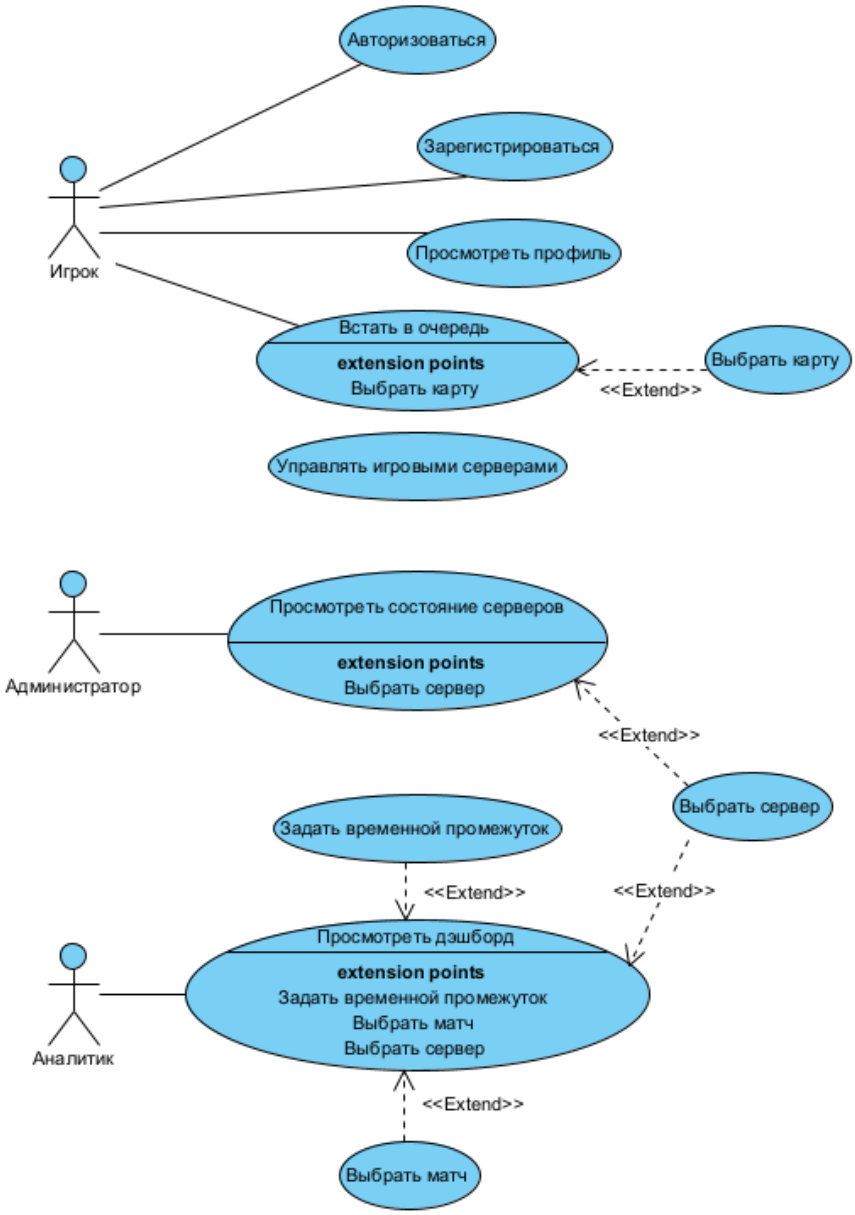


Рисунок 1 – Диаграмма вариантов использования

На диаграмме выделены следующие акторы:

Игрок – взаимодействует с игровым сервером через клиент, генерирует телеметрические события;

Администратор – управляет игровыми серверами, наблюдает за состоянием инфраструктуры через веб-панель;

Аналитик – просматривает аналитические дашборды, выявляет аномалии.

1.4 Выбор инструментальных средств разработки

Для реализации серверной архитектуры и системы телеметрии был проведён сравнительный анализ альтернативных технологий для каждого ключевого компонента. Критериями сравнения выступали производительность, масштабируемость, сложность внедрения, качество документации и наличие активного сообщества.

Сетевой движок для клиента Unity. Рассматривались FishNet, Photon Fusion и Mirror. FishNet выбран благодаря высокой производительности при синхронизации большого количества объектов, открытому исходному коду и встроенным механизмам управления интересами, что критично для снижения сетевого трафика. Photon Fusion имеет ограничения на число одновременных игроков в бесплатной версии, Mirror уступает FishNet в производительности [4].

Бэкенд-фреймворк для API-шлюза и менеджера серверов. Анализировались ASP.NET Core, Node.js и Python/Django. ASP.NET Core [5] показал лучшие результаты в независимых тестах производительности, обеспечивает встроенную поддержку асинхронного программирования и внедрения зависимостей, а также позволяет использовать общую экосистему .NET с игровым сервером, что упрощает поддержку кода.

База данных для телеметрии. Оценивались ClickHouse, TimescaleDB и InfluxDB. ClickHouse превосходит альтернативы по скорости вставки и выполнения агрегатных запросов за счёт колоночного хранения и векторизации вычислений [6]. Поддерживает материализованные представления и автоматическое управление временем жизни данных, что идеально для игровой телеметрии.

Брокер сообщений для асинхронной доставки телеметрии. Сравнивались RabbitMQ, Apache Kafka и Redis Streams. RabbitMQ выбран из-за простоты развёртывания, достаточной пропускной способности, поддержки сложной маршрутизации и подтверждений доставки [7]. Kafka избыточна по ресурсам для данного проекта, Redis Streams менее надёжен при сохранении сообщений.

Реляционная база данных для пользовательской информации и кэш. Для хранения профилей, инвентаря и других сущностей используется PostgreSQL. В качестве ORM выбран Dapper [8]. Для кэширования выбран Valkey [9], являющийся форком Redis, но полностью совместимый с его API.

Веб-панель телеметрии. Используется связка Vite и React.js, обеспечивающая быструю разработку и отличную производительность клиентской части.

Выбранный стек технологий полностью отвечает требованиям производительности, масштабируемости и скорости разработки.

1.5 Анализ требований к серверной архитектуре и системе телеметрии

Разрабатываемая серверная архитектура и система телеметрии должны обеспечивать автоматическое управление игровыми процессами, надёжный сбор событий и оперативную аналитику. На основе проведённого анализа были сформулированы следующие требования.

Функциональные требования:

– автоматическое выявление изменения нагрузки и динамический запуск и остановка игровых серверов;

– возможность для администратора вручную запускать или останавливать игровые серверы через API;

– сбор телеметрических событий движения игрока, выстрела, смерти игрока и состояния сервера;

– гарантированная доставка событий в хранилище, даже при временной недоступности конечной базы данных события не должны теряться;

– хранение телеметрии с возможностью автоматического удаления устаревших данных;

– предоставление веб-панели с дашбордами для администраторов и аналитиков;

– задержка от момента возникновения события до его отображения в дашборде не более 2 секунд.

Нефункциональные требования:

– один игровой сервер должен поддерживать не менее 4 одновременно подключённых игроков без деградации сетевых характеристик;

– система должна позволять увеличивать количество игровых серверов на одной физической машине по мере роста нагрузки;

– при падении одного из игровых серверов менеджер должен зафиксировать сбой и, при необходимости, перезапустить процесс, а остальные серверы должны продолжать работу;

– система должна обрабатывать не менее 1000 телеметрических событий в секунду без потерь;

– доступ к API-шлюзу и веб-панели должен требовать аутентификации; разграничение прав между администраторами и аналитиками.

Сформулированный набор требований позволяет создать работоспособную серверную архитектуру и систему телеметрии, пригодную для коммерческой эксплуатации.

1.6 Планирование процесса разработки и бюджета

Планирование разработки проекта было выполнено при помощи Libre Project. В результате планирования было выделено 22 задачи. Задачи охватывают все этапы жизненного цикла разработки: от анализа предметной области до оформления пояснительной записки.

Первая фаза посвящена углублённому анализу предметной области и проектированию системы, окончательному утверждению функциональных и нефункциональных требований, а также классификации телеметрических событий. Вторая фаза охватывает реализацию базовых компонентов инфраструктуры: менеджера игровых серверов, API-шлюза, схем PostgreSQL и ClickHouse. В этот же период настраивается конвейер RabbitMQ и разрабатывается начальная версия системы кэширования на Valkey, создании

самого игрового сервера на Unity с интеграцией FishNet, реализации отправки телеметрии и разработке веб-панели на React и Vite. Также включает тестирование и подготовку документации.

Задачи проекта и их длительность представлены на рисунке 2.

	Название	Продолжи...	Начало	Окончание	Предшествующие	Название ресурса
1	**Проектирование**	31 дней?	**25.12.2025, 08:00**	**05.02.2026, 17:00**		
2	Анализ предметной обла	5 дней?	25.12.2025, 08:00	31.12.2025, 17:00		Аналитик;Компьютер
3	Обзор и сравнение техно	3 дней	01.01.2026, 08:00	05.01.2026, 17:00	2	Аналитик;Компьютер
4	Анализ функциональных	3 дней	06.01.2026, 08:00	08.01.2026, 17:00	3	Аналитик;Компьютер
5	Классификация телеметр	2 дней	09.01.2026, 08:00	12.01.2026, 17:00	4	Аналитик;Компьютер
6	Планирование процесса р	2 дней	13.01.2026, 08:00	14.01.2026, 17:00	5	Аналитик;Компьютер
8	Проектирование развертк	2 дней	15.01.2026, 08:00	16.01.2026, 17:00	6	Программист;Компьютер
9	Проектирование менедже	3 дней	19.01.2026, 08:00	21.01.2026, 17:00	8	Программист;Компьютер
10	Проектирование схемы Б	2 дней	22.01.2026, 08:00	23.01.2026, 17:00	9	Программист;Компьютер
11	Проектирование реляцис	2 дней	26.01.2026, 08:00	27.01.2026, 17:00	10	Программист;Компьютер
12	Проектирование системы	2 дней	28.01.2026, 08:00	29.01.2026, 17:00	11	Программист;Компьютер
13	Проектирование API-шлю	3 дней	30.01.2026, 08:00	03.02.2026, 17:00	12	Программист;Компьютер
14	Проектирование веб-пан	1 день	04.02.2026, 08:00	05.02.2026, 17:00	13	Дизайнер;Программист;Ко...
16	**Реализация**	79 дней?	**06.02.2026, 08:00**	**27.05.2026, 17:00**	**1**	
17	Реализация менеджера и	8 дней	06.02.2026, 08:00	17.02.2026, 17:00		Программист;Компьютер
18	Реализация API-шлюза	10 дней	18.02.2026, 08:00	03.03.2026, 17:00	17	Программист;Компьютер
19	Интеграция RabbitMQ и к	7 дней	04.03.2026, 08:00	12.03.2026, 17:00	18	Программист;Компьютер
20	Реализация хранения пол	8 дней	13.03.2026, 08:00	24.03.2026, 17:00	19	Программист;Компьютер
21	Реализация системы кэш	4 дней	25.03.2026, 08:00	30.03.2026, 17:00	20	Программист;Компьютер
22	Реализация игрового сер	10 дней?	31.03.2026, 08:00	13.04.2026, 17:00	21	Программист;Компьютер
23	Разработка веб-панели т	15 дней	14.04.2026, 08:00	04.05.2026, 17:00	22	Программист;Компьютер
24	Тестирование и отладка	8 дней?	05.05.2026, 08:00	14.05.2026, 17:00	23	Программист;Компьютер
25	Исправление критически	5 дней?	15.05.2026, 08:00	21.05.2026, 17:00	24	Программист;Компьютер
26	Подготовка документаци	4 дней	22.05.2026, 08:00	27.05.2026, 17:00	25	Компьютер;Бумага;Принтер

Рисунок 2 – Список задач проекта

На основе задач и ресурсов автоматически была построена диаграмма Ганта, представленная на рисунке 3.

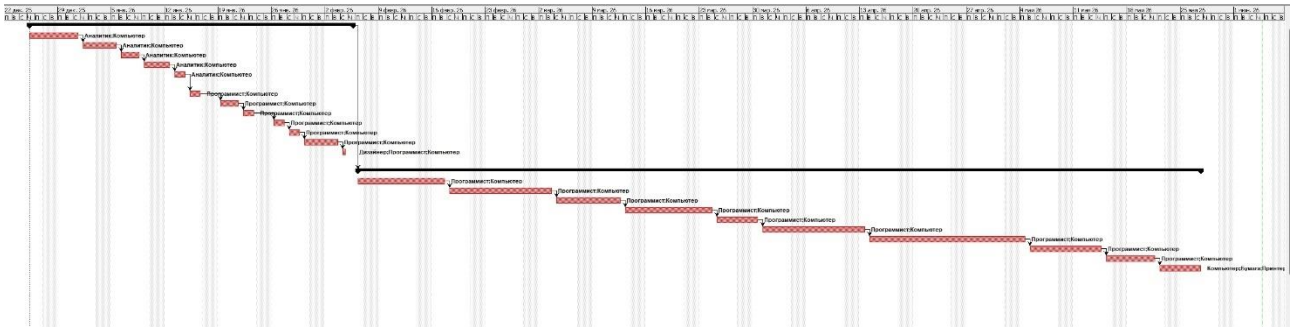


Рисунок 3 – Диаграмма Ганта для задач проекта

Разработка проекта составит около 110 полных рабочих дней.

2 Проектирование программных средств

2.1 Проектирование менеджера игровых серверов и жизненного цикла процессов

Центральным компонентом разрабатываемой серверной архитектуры является менеджер игровых серверов, отвечающий за автоматизацию запуска, мониторинга и остановки экземпляров игровых процессов. На этапе проектирования определены основные задачи этого компонента:

– создание нового игрового процесса с заданными параметрами;

– отслеживание состояния через периодические сигналы;

– корректное завершение работы, а также уведомление других подсистем об изменении статуса сервера.

На рисунке 4 представлена диаграмма последовательности, иллюстрирующая взаимодействие объектов при запуске нового игрового сервера по запросу от API-шлюза.

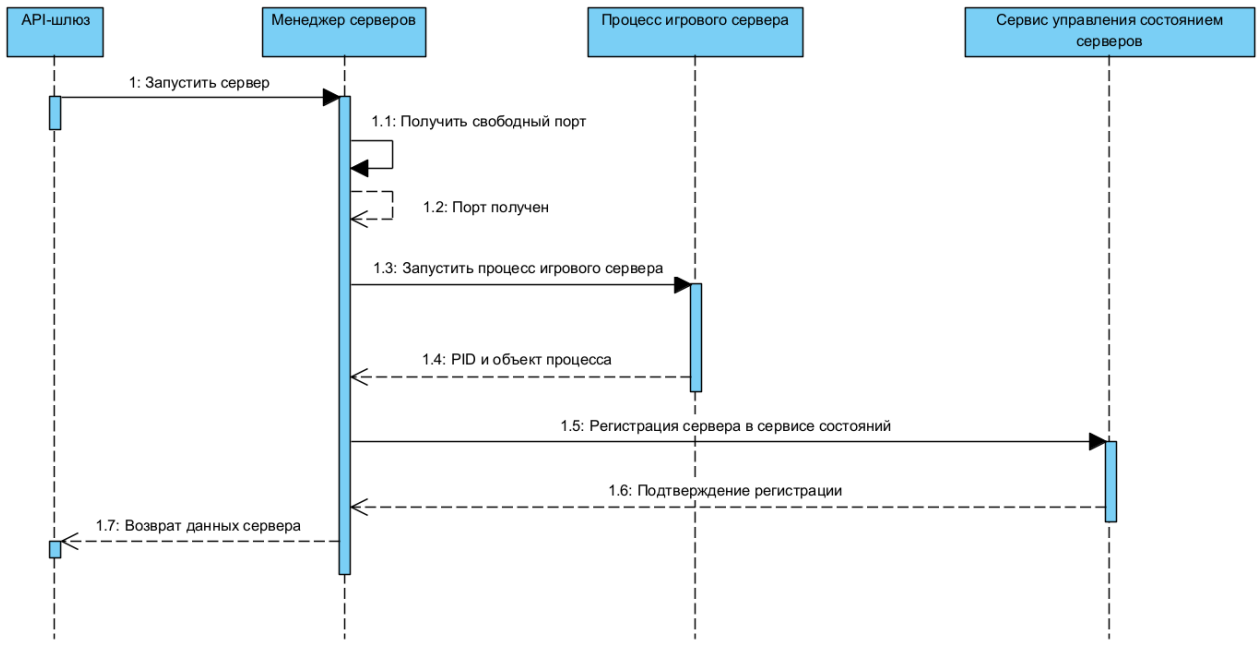


Рисунок 4 – Диаграмма последовательности запуска нового сервера

Основными участниками являются API-шлюз, экземпляр менеджера GameServerManager, процесс игрового сервера и сервис регистрации состояний IGameServerStateService. Последовательность шагов включает вызов метода

StartServerAsync с параметрами карты и максимального числа игроков, проверку доступности порта, создание и запуск процесса GameServer.exe, регистрацию сервера в сервисе состояний и возврат идентификатора сервера клиенту. Альтернативный поток описывает ситуацию, когда процесс не может быть запущен.

Для моделирования поведения менеджера и управляемых им игровых серверов была разработана диаграмма состояний, представленная на рисунке 5.

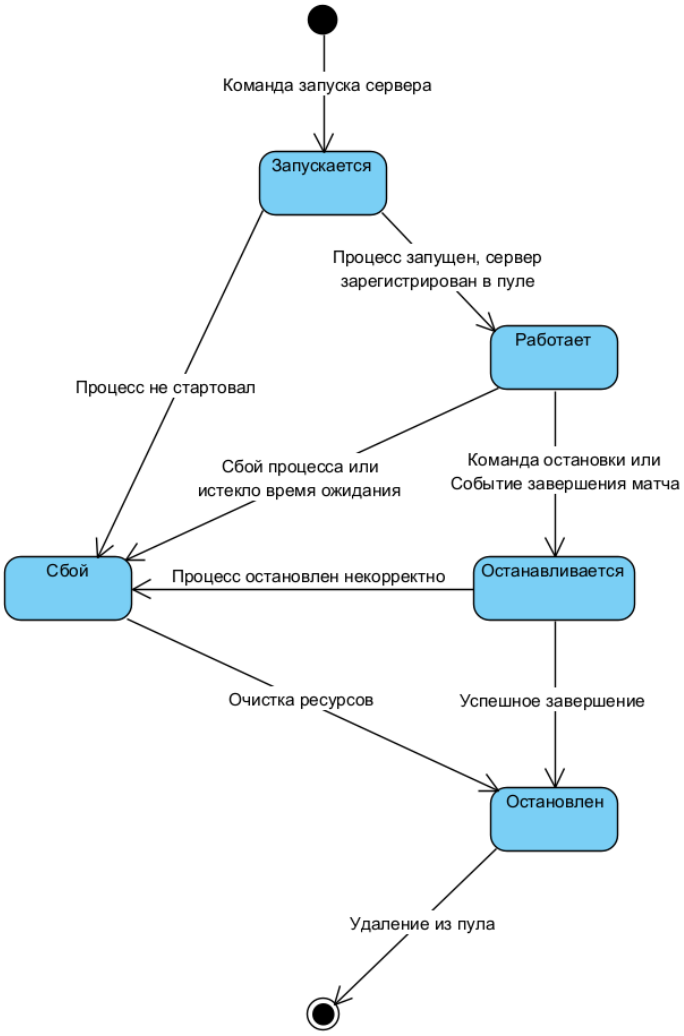


Рисунок 5 – Диаграмма состояний игрового сервера

На диаграмме выделены следующие состояния: «Остановлен», «Запускается», «Работает», «Останавливается», «Сбой». Переходы между состояниями инициируются событиями, вызванными либо сервером, либо администратором [10].

Проектирование менеджера с использованием перечисленных диаграмм позволило детализировать алгоритмы управления процессами, определить интерфейсы взаимодействия с другими подсистемами и заложить основу для последующей реализации на языке C# в среде .NET.

2.2 Проектирование схемы базы данных телеметрии

В основе сбора и хранения телеметрической информации многопользовательского проекта лежит выбор системы управления базами данных, оптимизированной для работы с высокими нагрузками на запись и аналитическими запросами по временным рядам. В рамках данной работы в качестве такой СУБД выбрана ClickHouse – колоночная аналитическая база данных, обеспечивающая высокую скорость вставки данных и эффективное сжатие за счёт колоночного хранения [11]. Основными критериями выступили: поддержка вложенных и массивных типов, низкая задержка при агрегации по большому объёму записей, механизм материализованных представлений, а также встроенные средства для управления жизненным циклом данных. На рисунке 6 приведена общая концептуальная схема потоков телеметрии от игровых серверов до конечных витрин данных, где ClickHouse выступает в роли основного хранилища исторических событий.

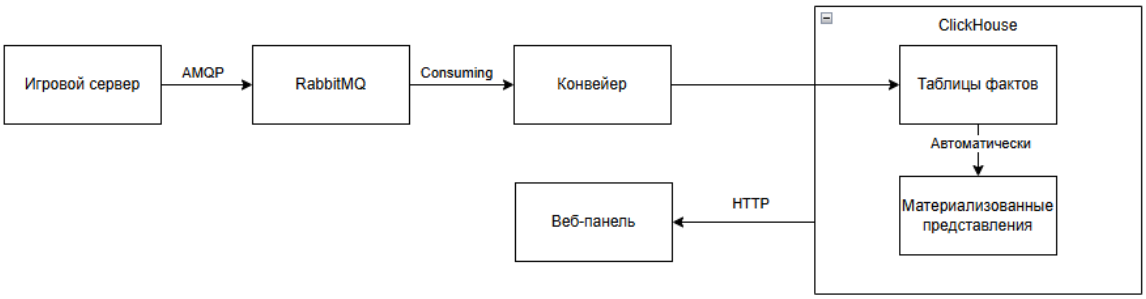


Рисунок 6 – Концептуальная схема потоков телеметрии

Проектирование логической схемы базы данных началось с классификации телеметрических событий. На основе выделенных типов событий – перемещения, выстрелы, смерти, получаемый урон, лечение, использование способностей, транзакции с внутриигровой валютой, события

транспортных средств, системные и сессионные события – были спроектированы отдельные таблицы для каждой категории. Был использован денормализованный подход: каждая таблица фактов содержит все необходимые атрибуты, дублируя такие поля, как идентификатор сервера, название карты и список активных модулей. Это соответствует рекомендациям ClickHouse, где объединения являются узким местом, поэтому их следует избегать даже ценой избыточности хранения. Такой подход выбран, так как каждый тип события обладает уникальным набором атрибутов. Объединение всех событий в одну таблицу с полем-дискриминатором привело бы к крайней разреженности данных – большая часть колонок, где в каждой строке были бы NULL, что ухудшает сжатие. Кроме того, события имеют разную гранулярность: перемещения генерируются с большей частотой, а смерти – эпизодически. Раздельные таблицы позволяют изолировать высокочастотные потоки, применять независимые политики TTL и настраивать для каждой таблицы оптимальный порядок сортировки под характерные аналитические запросы. Это даёт максимальную производительность при минимальном потреблении дискового пространства. Каждая таблица фактов начинается с первичного идентификатора события, обязательных полей идентификатора игрока, временной метки и идентификатора матча. Поля экземпляра сервера, названия карты и активных модулей экипировки вынесены в каждую таблицу для обеспечения возможности фильтрации и группировки без необходимости объединения таблиц.

Для всех фактовых таблиц использован MergeTree – базовый семейный движок ClickHouse, позволяющий задать ключ партиционирования и порядок сортировки. Партиционирование выполнено по дате, что при типичном сроке хранения телеметрии в 30 дней позволяет эффективно удалять старые данные на уровне целых каталогов на диске. Ключ сортировки подобран таким образом, чтобы наиболее частые аналитические запросы выполнялись максимально быстро – поэтому для большинства таблиц задан тройной ключ. Такое решение обеспечивает расположение записей одного матча и одного игрока на соседних участках диска, что критически снижает количество читаемых диапазонов.

При проектировании типов данных активно использовались низкокардинальные типы для полей, принимающих небольшое количество уникальных значений. Это позволило существенно сжать данные в памяти и на диске без потери читаемости. Координаты и углы поворота хранятся в типах Float32, что даёт достаточную точность для анализа траекторий и тепловых карт при экономном расходе места. Для идентификаторов выбраны целочисленные типы.

Для обеспечения высокой производительности панели телеметрии спроектирован набор материализованных представлений. В отличие от обычных представлений, в ClickHouse они хранят физически пересчитанные агрегаты, которые обновляются при вставке новых строк в базовые таблицы. Это переносит вычислительную нагрузку со времени исполнения запроса на момент записи данных, что для игровой телеметрии является оправданным компромиссом.

Итоговая схема базы данных телеметрии содержит одиннадцать фактовых таблиц и семнадцать материализованных представлений, и представлена на рисунке 7.

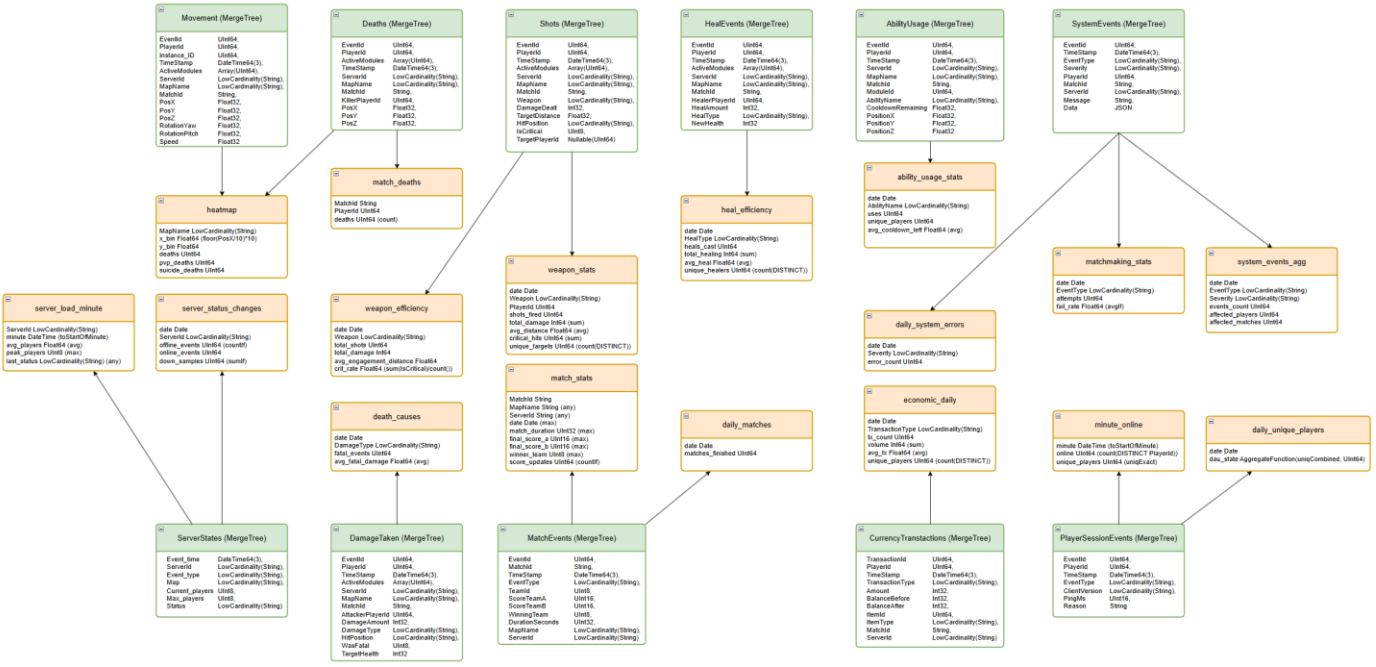


Рисунок 7 – Схема базы данных телеметрии

Для удобства фактовые таблицы были выделены зеленым, а материализованные представления – оранжевым. Такое количество обусловлено спецификой многопользовательской игры. В зависимости от проекта, наборы собираемых данных могут сильно отличаться.

2.3 Проектирование реляционной базы данных

Для хранения пользовательских данных, социальных связей, учётных записей и всей операционной информации, не связанной с высокочастотной телеметрией, спроектирована реляционная база данных на основе СУБД PostgreSQL [12, 13]. На рисунке 8 представлена ER-диаграмма базы данных системы.

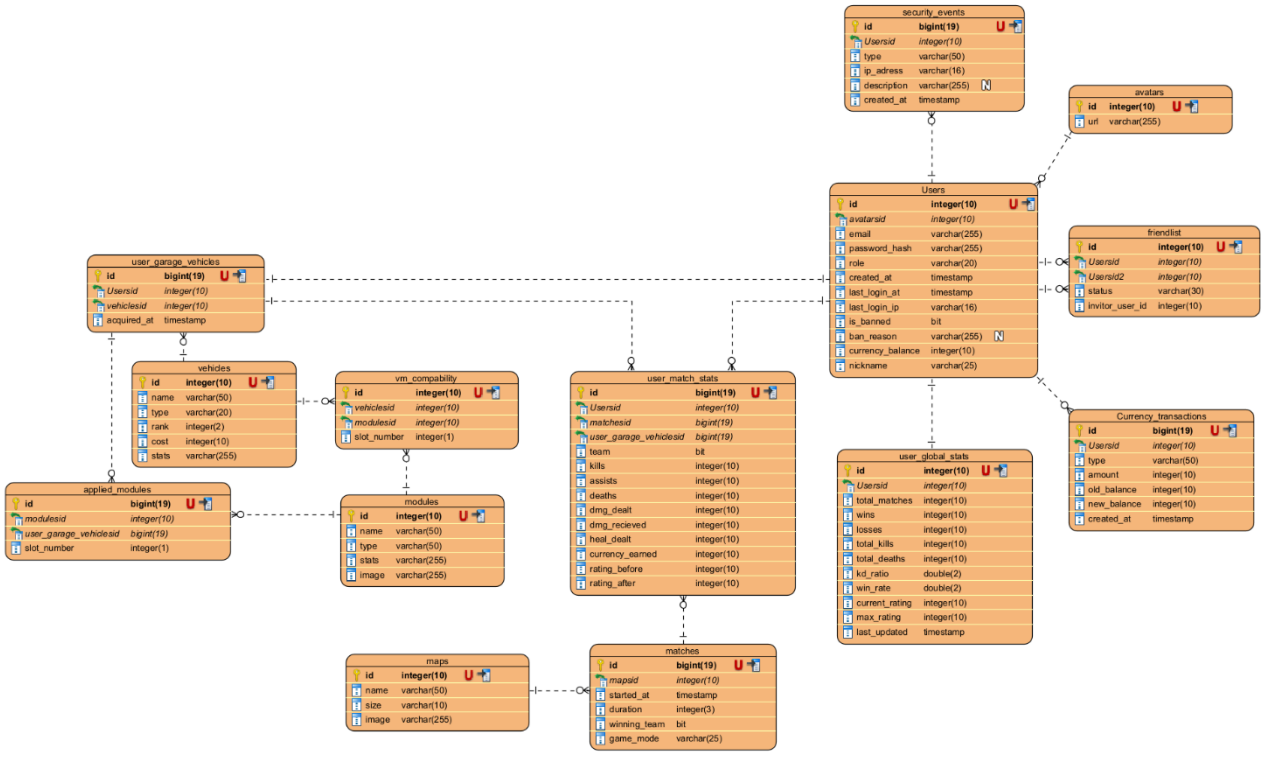


Рисунок 8 – ER-Диаграмма базы данных системы

Центральной сущностью является таблица Users, которая хранит учётные данные, статистика игроков выделена в таблицу User_global_stats с явной связью один-к-одному к Users. Такое решение принято для ускорения операций чтения – отображения профиля, рейтинга и места в таблице лидеров. Вычисляемые поля KD_Ratio и Win_rate реализованы как генерируемые, это переносит логику расчёта на уровень СУБД, гарантируя консистентность и снижая нагрузку на

прикладной код. Показатели Current_rating и Max_rating обновляются по окончании каждого матча, что позволяет строить рейтинговые системы.

Социальная подсистема реализована таблицей FriendList. Ключевое архитектурное решение – использование суррогатного первичного ключа Bind_ID и составного ограничения уникальности с проверкой. Это исключает дублирование пар друзей независимо от порядка идентификаторов. Поле Invitor_user_ID хранит инициатора запроса, а Status управляет жизненным циклом связи. Отдельные индексы на Status и составной ключ обеспечивают быструю выборку ожидающих запросов для конкретного пользователя.

Для хранения игрового прогресса спроектирована система из пяти таблиц. Vehicle описывает доступные машины, базовые характеристики вынесены в поле Base_stats с типом JSONB – это даёт гибкость, разные типы техники могут иметь разные наборы атрибутов без изменения схемы. Аналогично, таблица Modules хранит модули с их характеристиками в JSONB. Совместимость модулей с техникой определяется через таблицу VM_Compability, где пара Vehicle_ID, Module_ID является первичным ключом, а Slot_number указывает допустимую позицию установки. Эта связь многие-ко-многим с атрибутом слота – классический паттерн для настройки. Владение игрока конкретными экземплярами техники фиксируется в таблице User_garage_vehicle. Уникальный индекс на User_ID предотвращает дублирование владения одной и той же машиной одним игроком, а Acquired_at фиксирует момент покупки. Установленные модули на конкретный экземпляр хранятся в таблице Applied_modules_on_user_vehicle, где составной ключ Instance_ID, Slot_number гарантирует, что в один слот нельзя поставить два модуля.

Подсистема логирования и аудита представлена таблицами Currency_transactions и Security_events. Первая хранит все изменения баланса пользователя: тип операции, сумму, а также старый и новый баланс. Хранение обоих значений позволяет восстанавливать историю даже при последующих изменениях и выявлять аномалии. Вторая таблица фиксирует события безопасности.

При проектировании уделено внимание производительности и масштабированию. Для таблиц с потенциально высоким темпом вставки заложена возможность партиционирования по диапазону. Для снижения нагрузки на основную БД и ускорения частых операций проектом предусмотрено кэширование в Valkey, однако сама реляционная база спроектирована с запасом по производительности. Все запросы аутентификации идут по уникальным индексам, а обновление статистики использует точечные UPDATE по первичному ключу без блокировок смежных таблиц. Спроектированная база данных PostgreSQL полностью покрывает функциональные требования к хранению пользовательской информации и соответствует нефункциональным требованиям по надёжности и масштабируемости [14].

2.4 Проектирование системы кэширования

Для обеспечения высокой производительности и минимизации задержек при обработке запросов клиентов в проектируемой системе телеметрии и управления игровыми серверами необходимо внедрить слой распределённого кэша. В качестве основного хранилища ключ-значение выбран Valkey – высокопроизводительное in-memory хранилище, совместимое с протоколом Redis, поддерживающее все основные структуры данных и обеспечивающее низкую задержку операций. В данной архитектуре, Valkey используется для кэширования трёх категорий данных: сессий аутентификации и токенов пользователей, профилей игроков с глобальной статистикой, а также состояния игровых серверов в реальном времени. Такой подход позволяет разгрузить реляционную базу данных PostgreSQL и аналитическое хранилище ClickHouse, сократить время ответа API-шлюза и обеспечить горизонтальную масштабируемость кэш-слоя.

При проектировании ключей и структур данных принята иерархическая схема с префиксами, обеспечивающая логическую группировку и предотвращение коллизий. Для хранения информации о текущих сессиях

аутентификации используется хэш-таблица с ключом вида User:{userId}:Tokens:{tokenId}, где в поле хранится сам refresh-токен. Такая структура позволяет быстро проверять принадлежность токена пользователю и управлять временем жизни каждой записи независимо. Время жизни для такого ключа устанавливается равным времени действия refresh-токена. Для отзыва токенов вводится отдельное хранилище чёрного списка с ключом token_blacklist:{tokenId}, значение которого не имеет смысла, а наличие самого ключа означает, что токен более не действителен. TTL для чёрного списка также равен времени жизни токена, что гарантирует автоматическую очистку. Такой механизм позволяет API-шлюзу проверять валидность токена, выполняя лишь одну операцию в Valkey. На рисунке 9 представлена схема ключей и структур данных в Valkey.

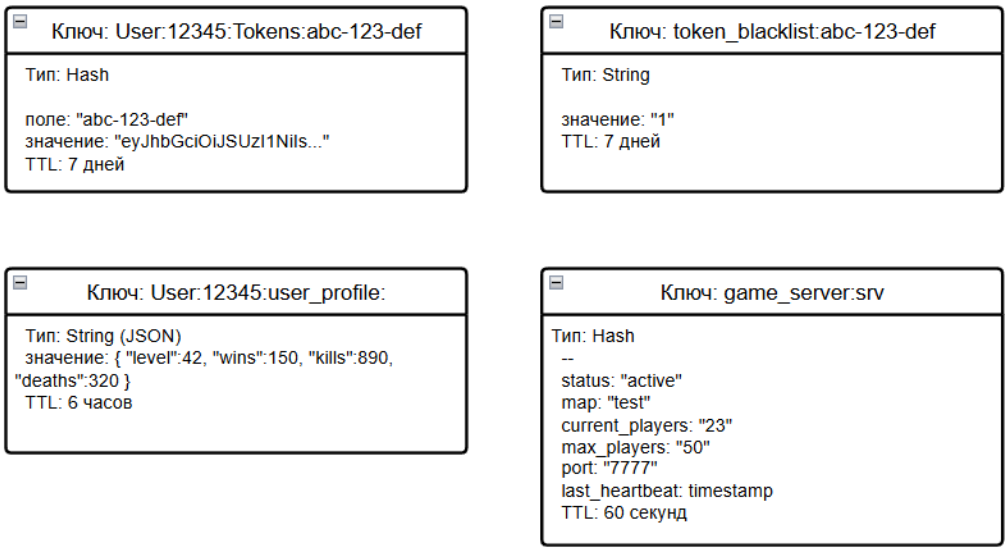


Рисунок 9 – Схема ключей и структур данных

Для кэширования профилей пользователей с глобальной статистикой применяется простая строковая структура с ключом User:{UID}:user_profile:. Значение представляет собой JSON объект, содержащий общую информацию об игроке. При первом запросе профиля из PostgreSQL данные загружаются, сериализуются и сохраняются в Valkey с TTL, равным 6 часам. Последующие запросы того же пользователя обслуживаются из кэша, что особенно важно для часто обновляемых страниц веб-панели телеметрии и игрового клиента. Для

снижения связанности с PostgreSQL реализован паттерн Cache-Aside, при отсутствии ключа выполняется чтение из базы данных, после чего кэш обновляется. Такой подход также применён при получении списка всех активных игровых серверов, где каждый сервер хранится в виде хэша с ключом game_server:{serverId}. На рисунке 10 представлена диаграмма последовательности, отображающая работу паттерна Cache-Aside.

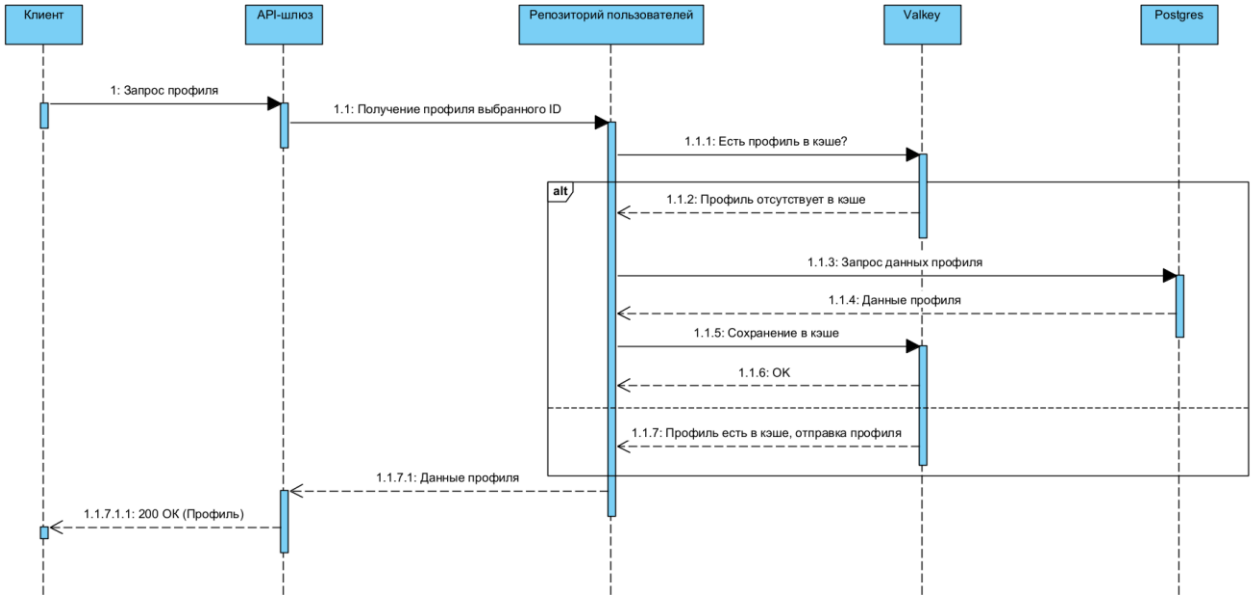


Рисунок 10 – Диаграмма последовательности получения профиля

Состояние игровых серверов – один из наиболее динамичных типов данных в системе. Каждый игровой сервер периодически отправляет запросы с информацией о текущем статусе. Эти данные принимаются API-шлюзом и передаются в сервис, который обновляет соответствующий хэш в Valkey. Время жизни ключа устанавливается в 60 секунд, что автоматически удаляет записи серверов, переставших отправлять своё состояние. Такая схема гарантирует актуальность списка доступных серверов для клиентов при подборе матча и позволяет менеджеру игровых серверов принимать решения о масштабировании на основе данных о текущей загрузке.

2.5 Проектирование API-шлюза

API-шлюз спроектирован как единая точка входа для всех клиентских приложений, включая игровые клиенты и веб-панель телеметрии. Основная

функция шлюза – маршрутизация запросов к соответствующим внутренним сервисам, обеспечение безопасности, агрегация данных из различных источников и снижение нагрузки на бэкенд-сервисы за счёт кэширования часто запрашиваемых данных. Архитектурно шлюз реализован как ASP.NET Core Web API [15], доступный клиентам по протоколам HTTP. Для обеспечения низкой задержки и высокой пропускной способности шлюз спроектирован асинхронным.

Взаимодействие с клиентами спроектировано по принципу RESTful API с использованием JSON в качестве формата обмена данными. Для игровых клиентов критически важны низкая задержка и надёжность, поэтому на этапе проектирования выделены две категории запросов: синхронные, требующие немедленного ответа, и асинхронные, где допустима отсроченная обработка. Асинхронные запросы обрабатываются шлюзом с немедленным подтверждением приёма, а фактическая запись в ClickHouse выполняется через брокер сообщений RabbitMQ. Такой подход снижает время ответа шлюза и защищает основную базу данных аналитики от пиковых нагрузок. В проекте API-шлюза учтены сценарии частичной недоступности сервисов. При отказе ClickHouse телеметрические события продолжают приниматься и складироваться в RabbitMQ, а при восстановлении системы они будут обработаны.

Проектирование маршрутов и методов API выполнено в соответствии с предметной областью и разделено на функциональные блоки. Блок аутентификации содержит методы регистрации, входа, обновления токенов и выхода, причём для защищённых точек доступа применяется JWT Bearer-аутентификация с проверкой подписи и срока действия. Блок управления игровыми серверами предоставляет клиентам возможность войти в очередь поиска матча, получить адрес и порт выделенного сервера, а серверам отправлять своё состояние для поддержания актуальности данных. Блок пользовательских данных включает точку доступа получения данных игрока с приоритетным чтением из кэша Valkey для снижения нагрузки на PostgreSQL.

Блок аналитики спроектирован как наиболее обширный, он принимает пакетные события телеметрии, а также предоставляет множество точек доступа для веб-панели телеметрии. Все точки доступа спроектированы с использованием асинхронных сигнатур, что позволяет эффективно использовать пул потоков.

Все точки доступа, кроме регистрации и входа, защищены атрибутом Authorize. Валидация JWT-токенов выполняется автоматически, при этом используется симметричное шифрование на основе ключа из конфигурации. Защита от подделки межсайтовых запросов в проекте не требуется, для веб-панели телеметрии используется CORS-политика, разрешающая запросы только с доверенных источников. Диаграмма классов, включающая в себя контроллеры, репозитории и сервисы, представлена на рисунке 11.

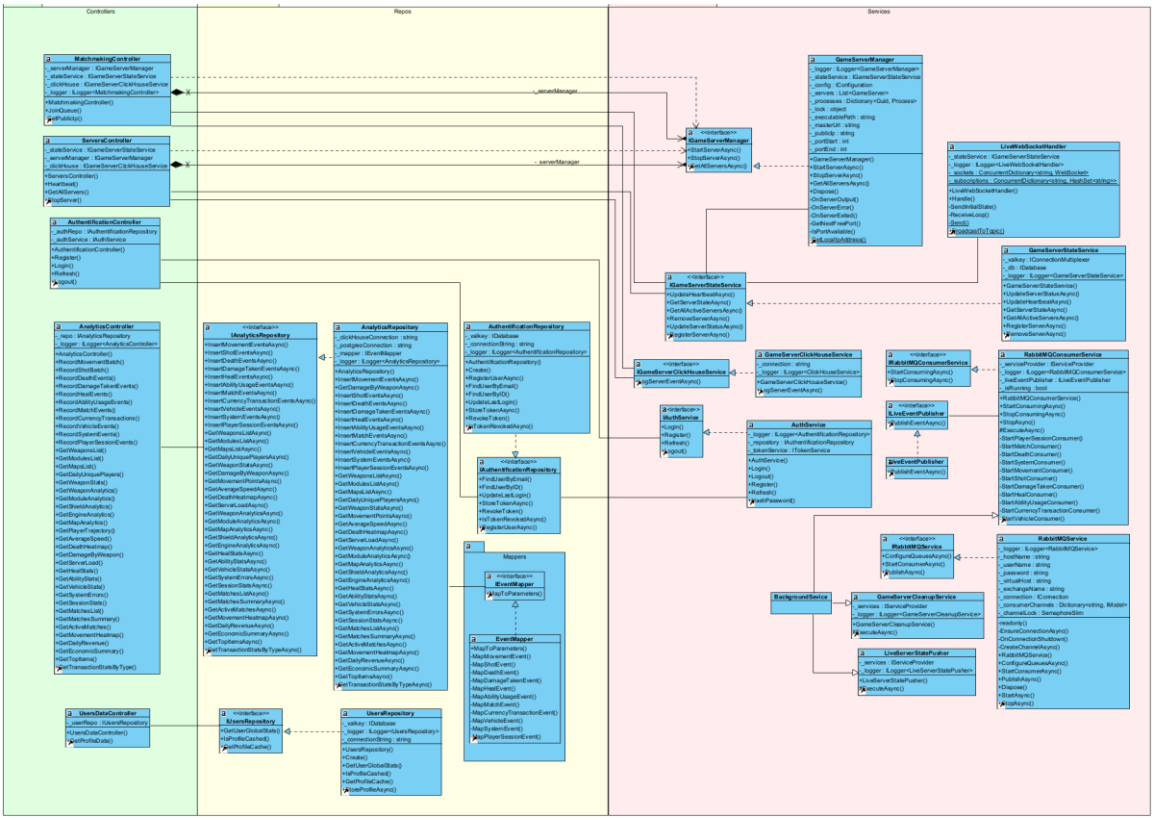


Рисунок 11 – Диаграмма классов API-шлюза

Интеграция шлюза с внешними компонентами спроектирована через явно определённые интерфейсы. Для доступа к PostgreSQL используется Dapper с хранимыми процедурами, что обеспечивает максимальную производительность и изоляцию логики данных. Для ClickHouse применяется специальный ADO.NET-клиент с поддержкой протокола HTTP. Для кэширования и хранения

сессий используется интерфейс IDatabase из библиотеки StackExchange.Redis, что позволяет абстрагироваться от конкретной реализации и при необходимости заменить Valkey на другое хранилище.

2.6 Проектирование веб-панели телеметрии

Веб-панель телеметрии проектируется как централизованный инструмент визуализации и анализа игровых событий, агрегированных в ClickHouse. Основная цель панели – предоставить командам разработки, эксплуатации и гейм-дизайна возможность в реальном времени отслеживать ключевые метрики игрового процесса, баланса, производительности серверов и экономики. Панель взаимодействует с системой через API-шлюз, который выступает единой точкой входа для всех фронтенд-запросов, выполняя аутентификацию пользователей, авторизацию по ролям и маршрутизацию запросов к соответствующим сервисам. На рисунке 12 представлена общая диаграмма последовательности загрузки данных для веб-панели.

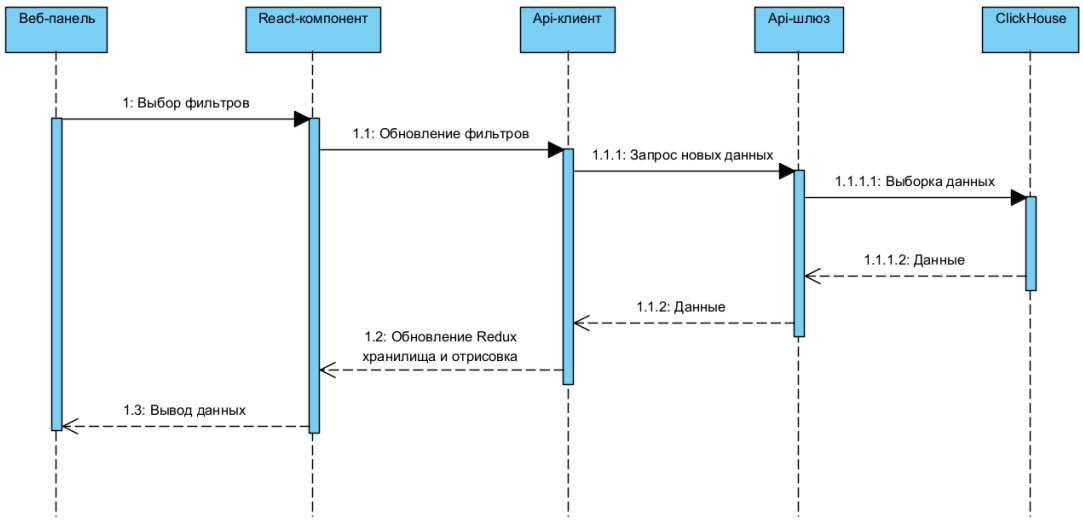


Рисунок 12 – Диаграмма последовательности загрузки данных

Проектирование интерфейса веб-панели базируется на модульной структуре React, где каждый модуль отвечает за отдельную аналитическую область. Выделяются следующие основные модули:

– dashboard – главная страница с агрегированными метриками – DAU, онлайн в реальном времени, количество активных матчей, ошибки серверов;

– matchAnalytics – детальный разбор конкретного матча: тепловая карта смертей, временная шкала событий, эффективность игроков;

– liveMonitor – состояние игровых серверов текущая загрузка, статусы, количество игроков в реальном времени;

– weaponBalance – статистика по оружию: частота использования, наносимый урон, дистанция, критический шанс;

– economy – метрики экономики: оборот валюты, популярность покупок, источники наград;

– systemHealth – события системы: ошибки античита, сбои matchmaking, нагрузка на серверы;

– heatmap – интерактивная карта уровней для анализа позиционной гибели и частых передвижений игроков.

Каждый модуль является отдельным React-компонентом, принимающим параметры временного диапазона и фильтры.

Взаимодействие React-модулей с бэкендом организовано через специализированный слой сервисов, которые инкапсулируют HTTP-запросы к точкам доступа API-шлюза [16]. Для всех аналитических данных, основанных на материализованных представлениях, используются GET-запросы с параметрами даты и фильтров. Для потоковых данных проектируется WebSocket-соединение, которое через API-шлюз подписывается на соответствующие каналы RabbitMQ.

Для визуализации данных проектируются следующие типы графиков и компоненты: временные ряды для отображения DAU, онлайн по минутам, количество матчей; тепловая карта смертей на основе данных death_heatmap с возможностью зума и фильтрации по типу смерти; столбчатые диаграммы для сравнения эффективности оружия и типов исцеления; таблицы с сортировкой и пагинацией для детального лога событий; карточки с ключевыми показателями для быстрого мониторинга состояния системы. Дополнительно проектируется панель фильтров общего действия, которая синхронизирована со всеми модулями через глобальное состояние – изменение фильтра вызывает повторную выборку данных во всех активных визуализациях.

2.7 Архитектура развертывания системы

Архитектура развертывания определяет физическое или виртуальное размещение программных компонентов на вычислительных узлах, а также сетевые протоколы и интерфейсы, обеспечивающие их взаимодействие. На рисунке 13 представлена диаграмма развертывания, отражающая распределение компонентов по узлам и основные каналы связи.

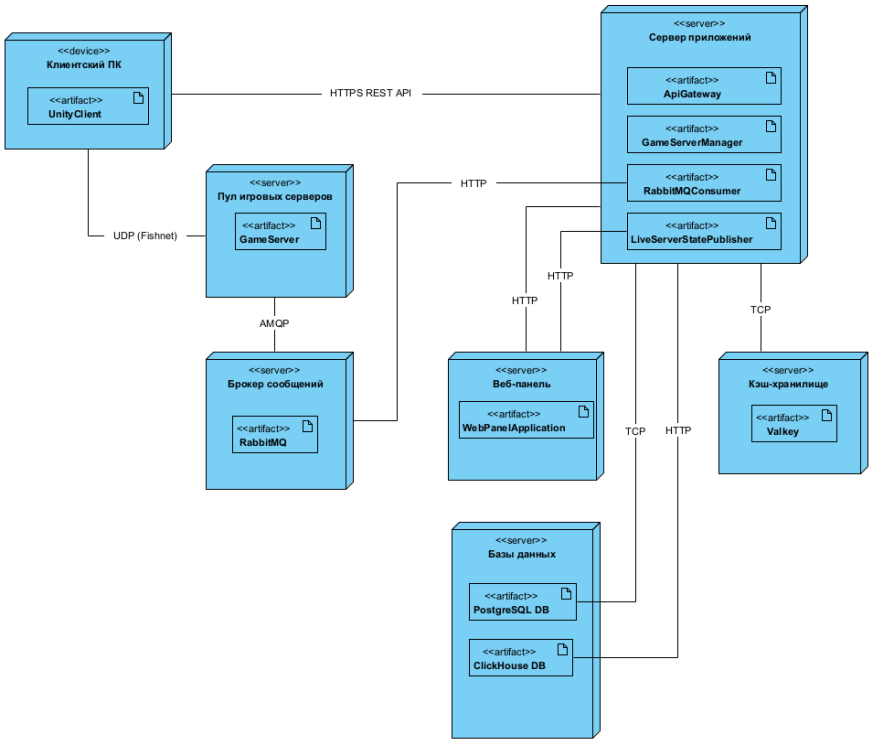


Рисунок 13 – Диаграмма развертывания

Клиентский узел представляет собой компьютер игрока, на котором функционирует игровое приложение. Клиент взаимодействует с API-шлюзом по протоколу HTTPS для аутентификации, загрузки профиля и получения данных для подключения к игровому серверу. Непосредственно игровой процесс осуществляется по протоколу UDP с установкой соединения через Fishnet с игровым сервером.

Сервер приложений развертывается на отдельной машине под управлением Windows Server или Linux. В его состав входят четыре основных компонента, реализованных на ASP.NET Core: API-шлюз, менеджер игровых серверов, приёмщик сообщений от RabbitMQ и сервис публикации данных в реальном времени для панели телеметрии. API-шлюз представляет собой

RESTful сервис для клиентов и веб-панели. Менеджер игровых серверов управляет жизненным циклом процессов GameServer.exe, запуская их через локальный вызов.

Узел баз данных представляет собой сервер с установленной СУБД PostgreSQL для реляционных данных и СУБД ClickHouse для телеметрических событий. В диаграмме развертывания обе СУБД показаны как отдельные компоненты, развернутые на одном физическом узле. Доступ к PostgreSQL осуществляется API-шлюзом через микро-ORM Dapper по протоколу TCP. Доступ к ClickHouse осуществляется через специализированный приемщик, который читает сообщения из RabbitMQ и выполняет массовую вставку по протоколу HTTP.

Узел брокера сообщений содержит запущенный контейнер RabbitMQ. Он принимает телеметрические события от игровых серверов через протокол AMQP и маршрутизирует их в соответствующие очереди.

Узел кэш-хранилища содержит запущенный контейнер Valkey, используется для хранения активных сессий пользователей, токенов и прочих чувствительных к задержкам данных. API-шлюз обращается к Valkey по протоколу TCP с использованием клиентской библиотеки StackExchange.Redis.

Веб-панель телеметрии развертывается как статичный веб-сервер, на котором размещены файлы, собранные инструментом Vite [17]. Панель реализована на React.js и взаимодействует с API-шлюзом по HTTP, запрашивая агрегированные данные телеметрии.

Пул игровых серверов состоит из одного или нескольких вычислительных узлов, на которых запускаются экземпляры процесса GameServer. Каждый игровой сервер получает при запуске уникальный идентификатор, порт, название карты, максимальное количество игроков и URL для отправки своего состояния.

Предложенная архитектура обеспечивает модульность, масштабируемость и изоляцию компонентов, что позволяет независимо развивать и наращивать мощность каждого из узлов в зависимости от нагрузки.

3 Реализация программных средств

3.1 Реализация менеджера игровых серверов

Центральным элементом серверной архитектуры является менеджер игровых серверов, реализованный в виде сервиса GameServerManager, который обеспечивает полный жизненный цикл выделенных игровых процессов. Данный компонент отвечает за запуск, мониторинг, остановку и автоматическую очистку серверов, а также за их регистрацию в системе телеметрии. Реализация построена на языке C# в ASP.NET Core с использованием паттерна «менеджер процессов» и интеграцией с Valkey для хранения актуального состояния серверов.

Ключевым классом является GameServerManager, реализующий интерфейс IGameServerManager. Он содержит коллекции запущенных серверов и соответствующих им процессов ОС, защищённые блокировкой lock для потокобезопасности. При запуске сервера метод StartServerAsync получает свободный порт из заданного диапазона, формирует аргументы командной строки для игрового процесса и создаёт новый процесс с перенаправлением стандартного вывода и ошибок. После успешного старта в словарь добавляется объект GameServer и экземпляр Process, после чего выполняется асинхронная регистрация сервера в сервисе состояния. Для проверки доступности порта используется комбинированная проверка TCP и UDP. Создание временного слушателя на заданном порту позволяет гарантировать, что порт не занят другими процессами.

Остановка сервера реализована в методе StopServerAsync с применением двухстадийного подхода: сначала процессу отправляется сигнал закрытия, что даёт игровому серверу возможность корректно завершить соединения с игроками и записать финальную телеметрию. Если в течение 10 секунд процесс не завершился, он принудительно уничтожается через Kill(). После остановки статус сервера обновляется в хранилище состояния, а запись об удалении передаётся в ClickHouse через сервис логирования. Для автоматической очистки

зависших серверов, потерявших связь с мастер-сервером, реализован фоновый сервис GameServerCleanupService, который каждые 30 секунд запрашивает список активных серверов из Valkey и проверяет время последнего сообщения о состоянии. Если с момента последнего сообщения прошло более 120 секунд, сервер считается мёртвым – менеджер принудительно останавливает соответствующий процесс и удаляет запись из Valkey. Важные фрагменты листинга представлены в приложении Б. Полный код системы расположен в репозитории [18]. Фрагмент реализации фонового сервиса остановки представлен в листинге Б.1.

Сервис состояния GameServerStateService выступает в роли абстракции над Valkey и предоставляет методы для регистрации, обновления состояний, получения информации о сервере и удаления записей. Все данные хранятся в хэш-таблицах с ключом вида game_server:{идентификатор} и временем жизни 60 секунд, что автоматически устаревает записи о не отвечающих серверах. Этот механизм обеспечивает актуальность информации для матчмейкинга без необходимости прямого опроса каждого игрового сервера.

Подбор матча реализован в отдельном контроллере MatchmakingController с методом JoinQueue, представленном на листинге Б.2.

При запросе от клиента на подключение к указанной карте система сначала сканирует активные серверы из Valkey, выбирая тот, который имеет подходящую карту, свободные слоты и статус. Если подходящий сервер найден, клиенту возвращается его публичный IP и порт. В противном случае вызывается GameServerManager.StartServerAsync для запуска нового экземпляра игрового процесса, который после регистрации в Valkey становится доступным для последующих игроков. Такой подход позволяет динамически масштабировать количество серверов в зависимости от нагрузки, избегая простоев и избыточного потребления ресурсов. Для целостности данных при неожиданном завершении процесса используется событие Exited процесса, которое асинхронно вызывает OnServerExited и удаляет сервер из всех коллекций и Valkey.

3.2 Реализация игрового сервера и отправки телеметрии

Игровой сервер представляет собой отдельное приложение и управляется менеджером серверов через командную строку. Для обеспечения полной автономности, сервер не содержит визуальных компонентов и работает в выделенном режиме, что минимизирует потребление ресурсов и упрощает развертывание.

Основным классом, отвечающим за инициализацию сервера, является ServerCommandLineArgs, который обрабатывает параметры командной строки при запуске. Сервер принимает следующие аргументы: -port, -serverId, -map и -masterUrl. Листинг Б.3 демонстрирует фрагмент ServerCommandLineArgs.

Система отправки состояния реализована в отдельном компоненте ServerHeartbeat, который после успешного старта сервера периодически отправляет POST-запрос с актуальной информацией: идентификатор сервера, текущее количество игроков, название карты и статус. Фрагмент реализации отправки состояния представлен на листинге Б.4. Эти данные позволяют API-шлюзу поддерживать актуальный реестр доступных серверов для подбора матча. При остановке сервера или потере соединения мастер-сервер по истечении времени, автоматически исключает сервер из списка доступных. Такой подход обеспечивает самовосстановление системы без необходимости ручного вмешательства.

Для управления жизненным циклом сервера реализован автоматический механизм завершения работы при отсутствии игроков. ServerCommandLineArgs подписывается на события OnServerConnectionStateChanged и OnRemoteConnectionStateChanged для отслеживания количества подключённых клиентов. При падении числа игроков до нуля запускается метод с задержкой 180 секунд, по истечении которых сервер корректно останавливает все соединения и завершает процесс через Application.Quit(). Если за это время подключается хотя бы один игрок, таймер сбрасывается. Данный механизм критически важен для экономии ресурсов, позволяя автоматически высвобождать вычислительные мощности неиспользуемых серверов.

Клиентская часть, реализованная в компоненте ClientConnector, обеспечивает подключение к подбору матчей и последующее соединение с выделенным игровым сервером [22]. Клиент отправляет POST-запрос на /api/Matchmaking/join с выбранными параметрами. В ответе получает IP-адрес и порт. После этого вызывается метод ConnectToGameServer, который программно устанавливает адрес и порт транспорта Tugboat и инициирует соединение через ClientManager.StartConnection(). FishNet автоматически обрабатывает установку соединения, репликацию объектов и синхронизацию состояния. Фрагмент ClientConnector представлен на листинге Б.5.

Ключевым элементом системы сбора телеметрии является компонент AnalyticsService, реализованный как синглтон с автоматическим подключением к RabbitMQ. При инициализации сервис создаёт соединение с брокером сообщений и настраивает автоматическое восстановление соединения. Для каждого типа игровых событий используются отдельные очереди [20]. Такая маршрутизация позволяет гибко настраивать очереди и потребителей на стороне бэкенда. На рисунке 14 представлен скриншот логов при обработке телеметрии.

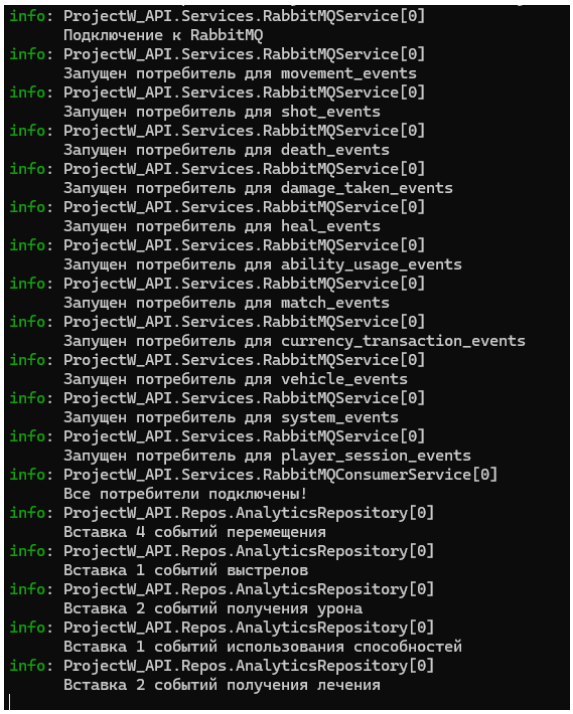


Рисунок 14 – Логи при вставке событий телеметрии

Учитывая высокую интенсивность потока событий, в системе реализован механизм пакетной отправки с использованием буферизации в памяти. Для

событий предусмотрены настраиваемые параметры: sendInterval, maxBatchSize. Отправка может быть отключена для отладки через флаги enableMovementBatching и enableShotBatching. Когда пакет заполнен или истекает таймер, события сериализуются в JSON, оборачиваются в контейнер с sessionId и временем отправки, после чего публикуются в RabbitMQ. Фрагмент реализации отправки событий представлен на листинге Б.6.

Для обеспечения целостности данных при конкурентном доступе используются блокировки lock на объектах. При выгрузке сцены или закрытии приложения вызывается метод SendAllBatchesImmediately(), который гарантирует отправку всех накопленных событий перед завершением работы, минимизируя потерю телеметрии.

3.3 Реализация API-шлюза

Центральным элементом серверной части разработанной системы является API-шлюз, реализованный на платформе ASP.NET Core. Этот компонент выполняет функции единой точки входа для всех клиентских приложений, обеспечивает аутентификацию и авторизацию запросов, маршрутизацию к соответствующим сервисам, а также служит связующим звеном между реляционной базой данных PostgreSQL, кэшем Valkey и аналитическим хранилищем ClickHouse.

Архитектура API-шлюза построена по принципу разделения ответственности. Контроллеры отвечают только за приём и валидацию HTTP-запросов, а вся бизнес-логика и работа с данными вынесены в отдельные сервисы и репозитории. Контроллеры используют интерфейсы соответствующих сервисов, что обеспечивает слабую связанность и упрощает тестирование. Для работы с реляционной базой данных реализованы репозитории, которые через Dapper выполняют как простые SQL-запросы, так и вызовы хранимых процедур. Фрагмент реализации репозитория с получением данных пользователя представлен на листинге Б.7.

Особое внимание уделено аутентификации и управлению сессиями. API-шлюз реализует механизм JWT-токенов с разделением на access-токен и refresh-токен. При успешном входе пользователя сервис аутентификации генерирует пару токенов, причём refresh-токен сохраняется в кэше Valkey с привязкой к идентификатору пользователя и уникальному идентификатору токена. При последующих запросах клиент передаёт access-токен в заголовке Authorization, а для обновления пары токенов используется точка Refresh, которая проверяет валидность refresh-токена, отсутствие его в чёрном списке и соответствие пользователю. На листинге Б.8 представлен фрагмент контроллера с точкой Refresh.

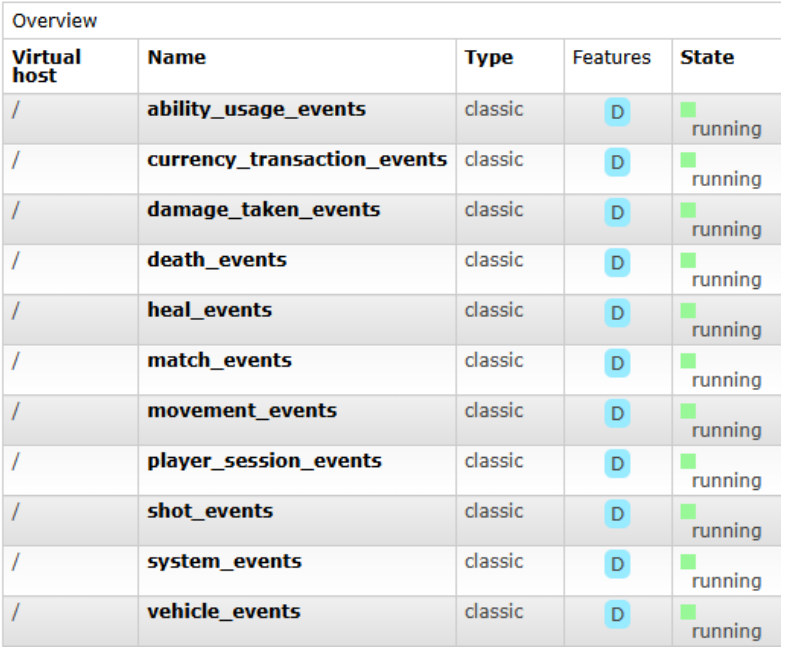
Отзыв токенов при выходе из системы или смене пароля реализован путём добавления идентификатора токена в чёрный список Valkey с временем жизни, достаточным для истечения срока самого токена. Такой подход позволяет полностью контролировать сессии без необходимости хранения состояния на стороне API-шлюза.

3.4 Интеграция брокера сообщений и конвейера загрузки данных

Разработанный конвейер обработки телеметрических данных базируется на брокере сообщений RabbitMQ [21], выступающем в роли буфера между игровыми серверами и аналитической базой ClickHouse. Такой архитектурный выбор обусловлен требованием асинхронной обработки высокого потока событий с гарантией доставки и устойчивостью к временным отказам хранилища. RabbitMQ разворачивается как отдельный сервис, обеспечивая маршрутизацию сообщений по принципу topic exchange – это позволяет гибко разделять типы телеметрических событий без изменения логики отправителя. На рисунке 15 представлен список очередей на клиенте rabbitmq.

На сервере функционирует фоновый сервис RabbitMQConsumerService, который при старте инициирует настройку очередей через IRabbitMQService.ConfigureQueuesAsync(). В рамках этой настройки создаются очереди для событий, каждая из которых привязывается к обменнику

telemetry.topic с соответствующими ключами маршрутизации [22]. Такой подход позволяет изолировать обработку разных типов событий, упрощая мониторинг и масштабирование. Фрагмент с настройкой очередей представлен на листинге Б.9.



Overview

Virtual host	Name	Type	Features	State
/	ability_usage_events	classic	D	running
/	currency_transaction_events	classic	D	running
/	damage_taken_events	classic	D	running
/	death_events	classic	D	running
/	heal_events	classic	D	running
/	match_events	classic	D	running
/	movement_events	classic	D	running
/	player_session_events	classic	D	running
/	shot_events	classic	D	running
/	system_events	classic	D	running
/	vehicle_events	classic	D	running

Рисунок 15 – Список очередей RabbitMQ

Для каждого типа событий запускается отдельный потребитель, внутри которого при получении сообщения создается отдельный scope через IServiceProvider. Полученный объект десериализуется в соответствующий класс события, после чего вызывается метод вставки в ClickHouse. Все операции логируются, а ошибки обработки перехватываются, чтобы избежать остановки всего конвейера – неподтверждённые сообщения остаются в очереди для повторной обработки. Фрагмент реализации потребителя представлен на листинге Б.10.

Запись данных в колоночную СУБД ClickHouse выполняется через реализацию IClickHouseService, использующую официальный ADO.NET провайдер ClickHouse.Client [19]. Для событий используются специализированные методы, которые формируют параметризованные вставки в таблицы. Для повышения пропускной способности применяется пакетная вставка. Фрагмент реализации вставки в базу данных телеметрии через репозиторий представлен на листинге Б.11.

Благодаря асинхронной архитектуре RabbitMQ-потребители не блокируются на операциях ввода-вывода в ClickHouse.

3.5 Реализация хранения пользовательских данных

Для хранения учётных записей создана таблица Users, содержащая идентификатор пользователя, email, хеш пароля, роль, временные метки создания и последнего входа, IP последнего входа, флаг бана с причиной, баланс игровой валюты, никнейм и ссылку на аватар. Пароли хешируются алгоритмом bcrypt – в базе хранится только хеш, что предотвращает компрометацию учётных данных при возможной утечке. Регистрация выполняется как атомарная транзакция: проверяется уникальность email и никнейма, создаётся запись в Users, затем автоматически создаётся связанная запись в User_global_stats с нулевыми значениями статистики и начальным рейтингом. Реализация запроса на создание таблицы Users представлена на листинге Б.12.

Глобальная статистика игроков хранится в таблице User_global_stats, связанной один-к-одному с Users. Эта таблица содержит накопительные показатели: общее количество матчей, побед, поражений, убийств, смертей, текущий и максимальный рейтинг. Реализация запроса на создание статистики представлена на листинге Б.13.

Логирование изменений баланса валюты вынесено в таблицу Currency_transactions, куда каждая операция записывается с указанием типа транзакции, суммы, старого и нового баланса. Это обеспечивает полный аудит игровой экономики и возможность отката при обнаружении ошибок или злоупотреблениях. Таблица Security_events фиксирует все значимые действия, связанные с безопасностью.

3.6 Реализация системы кэширования

В контексте многопользовательской игры кэш используется для трёх основных категорий данных: пользовательские токены авторизации, состояния игровых серверов, а также профильные данные игроков для ускорения загрузки

статистики. Такое разделение позволяет снизить нагрузку на основную реляционную базу данных и уменьшить время отклика API для часто запрашиваемых, редко изменяющихся данных.

Для хранения информации об активных игровых серверах используется структура хешей по ключу вида game_server:{идентификатор_сервера}. Реализация записи в кэш представлена на листинге Б.14.

Внутри хеша сохраняются поля, отражающие текущее состояние сервера: статус, название игровой карты, количество текущих игроков, максимальное количество игроков, а также временная метка последнего полученного состояния. Каждый раз при обновлении состояния или изменении статуса сервера параллельно обновляются соответствующие поля в хеше, а TTL ключа устанавливается в 60 секунд. Это гарантирует автоматическое удаление записи о сервере, который перестал присылать сигналы жизни, что критично для динамического управления подбором матчей и предотвращения направления клиентов на неработающие экземпляры.

Пользовательская аутентификация и управление токенами также используют возможности кэша. При успешном входе или обновлении токенов сгенерированный refresh-токен сохраняется по ключу User:{пользователь}:Tokens:{токен}. Значением ключа является сам токен, а TTL устанавливается равным времени жизни токена. Механизм отзыва токенов реализован двумя способами: явное удаление ключа активного токена и запись в чёрный список по ключу token_blacklist:{идентификатор_токена} с длительным TTL. При каждом запросе обновления токенов сервис проверяет наличие идентификатора токена в чёрном списке, что предотвращает возможность использования скомпрометированных или вышедших из употребления refresh-токенов. Реализация хранения токенов в кэше представлена на листинге Б.15.

Для ускорения доступа к профильной статистике игроков реализовано кэширование на уровне отдельных пользователей. Ключ User:{пользователь}:user_profile: хранит сериализованный в JSON объект UserProfileGlobalData. Время жизни этого кэша установлено в 6 часов, что

обусловлено относительно невысокой частотой запроса игровой статистики. Перед выполнением запроса к PostgreSQL сервис UsersRepository сначала проверяет наличие закэшированного профиля, при попадании данные возвращаются немедленно без обращения к базе, что значительно снижает задержки при отображении страницы профиля игрока в клиентском приложении и веб-панели. При обновлении статистики кэш инвалидируется путём удаления соответствующего ключа, что гарантирует актуальность данных при следующем запросе. Реализация загрузки профиля в кэш представлена на листинге Б.16.

Все операции с кэшем выполняются асинхронно с использованием интерфейсов IDatabase и IConnectionMultiplexer, что позволяет эффективно утилизировать потоки в среде ASP.NET Core. Логирование всех ключевых событий организовано через встроенный механизм ILogger, что облегчает отладку и мониторинг эффективности кэширования в эксплуатационной среде.

3.7 Реализация веб-панели телеметрии

Веб-панель телеметрии реализована как веб приложение с использованием библиотеки React и инструмента сборки Vite [23]. Основной клиентский код включает следующие ключевые разделы: компоненты страниц, переиспользуемые UI-компоненты, Redux-хранилище, клиенты для HTTP- и WebSocket-взаимодействия и пользовательские хуки. Применён подход функциональных компонентов с использованием хуков состояния и эффектов, что обеспечивает декларативность и предсказуемость поведения.

Основные компоненты и модули представлены на рисунке 16.

Маршрутизация реализована с помощью react-router-dom. В корневом компоненте App определены основные пути: дашборд, аналитика матчей, live-монитор, баланс оружия, экономика, здоровье системы и тепловые карты. Каждый маршрут связан с соответствующей страницей, а макет с боковым меню и верхним заголовком вынесен в общую обёртку, что обеспечивает единый стиль навигации. Боковое меню построено динамически на основе конфигурационного массива, поддерживает иконки и визуальную индикацию активного маршрута.

Управление глобальным состоянием выполнено с помощью Redux Toolkit. В хранилище определён срез filters, который хранит параметры фильтрации: диапазон дат, идентификатор сервера, название карты, идентификатор матча и идентификатор игрока. Эти значения используются для формирования запросов к API. Благодаря централизованному хранилищу, изменение дат в компоненте выбора автоматически перезапрашивает данные на всех активных графиках, что обеспечивает согласованность аналитики.

Коммуникация с бэкендом построена на двух уровнях. Для запросов исторической аналитики используется HTTP-клиент на основе axios, обёрнутый в модуль apiClient. Все функции автоматически форматируют даты в ISO-строку с помощью date-fns. Для данных в реальном времени реализовано WebSocket-соединение с поддержкой автоматического переподключения. Хук useLiveData подписывается на глобальное состояние, обновляемое при получении сообщений по протоколу, и предоставляет компонентам актуальные данные и статус соединения. Это позволило реализовать страницу LiveMonitor с отображением нагрузки серверов в реальном времени и лентой событий без необходимости периодического опроса.

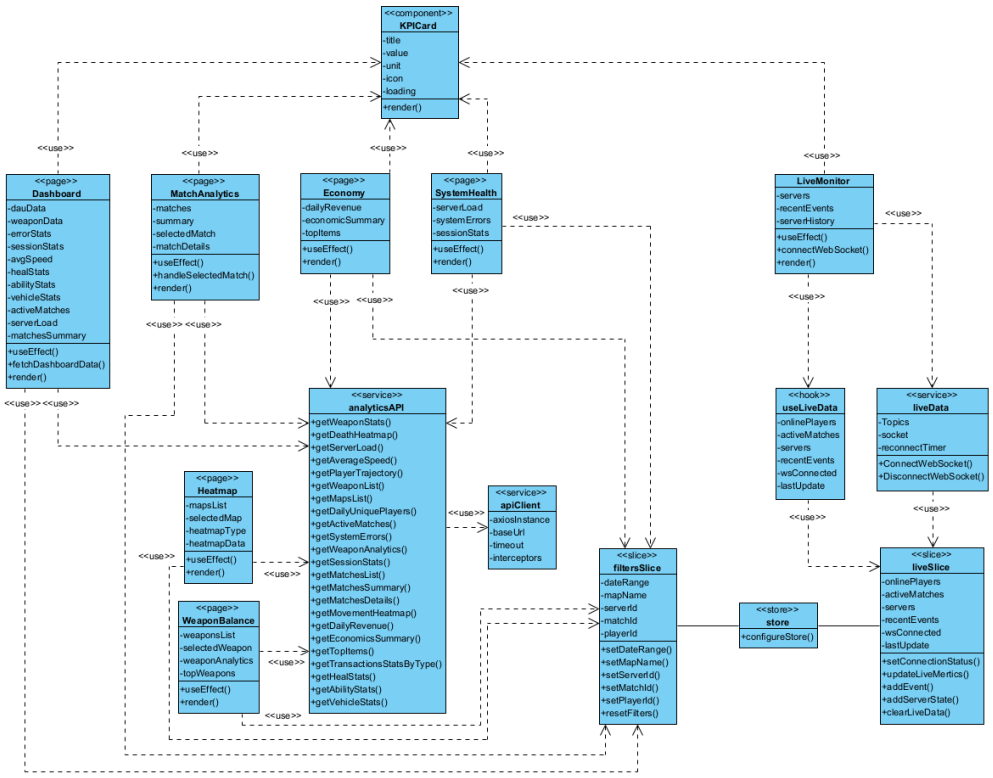


Рисунок 16 – Основные компоненты веб-панели

Интерфейс приложения с тестовыми данными представлен на рисунке 17.

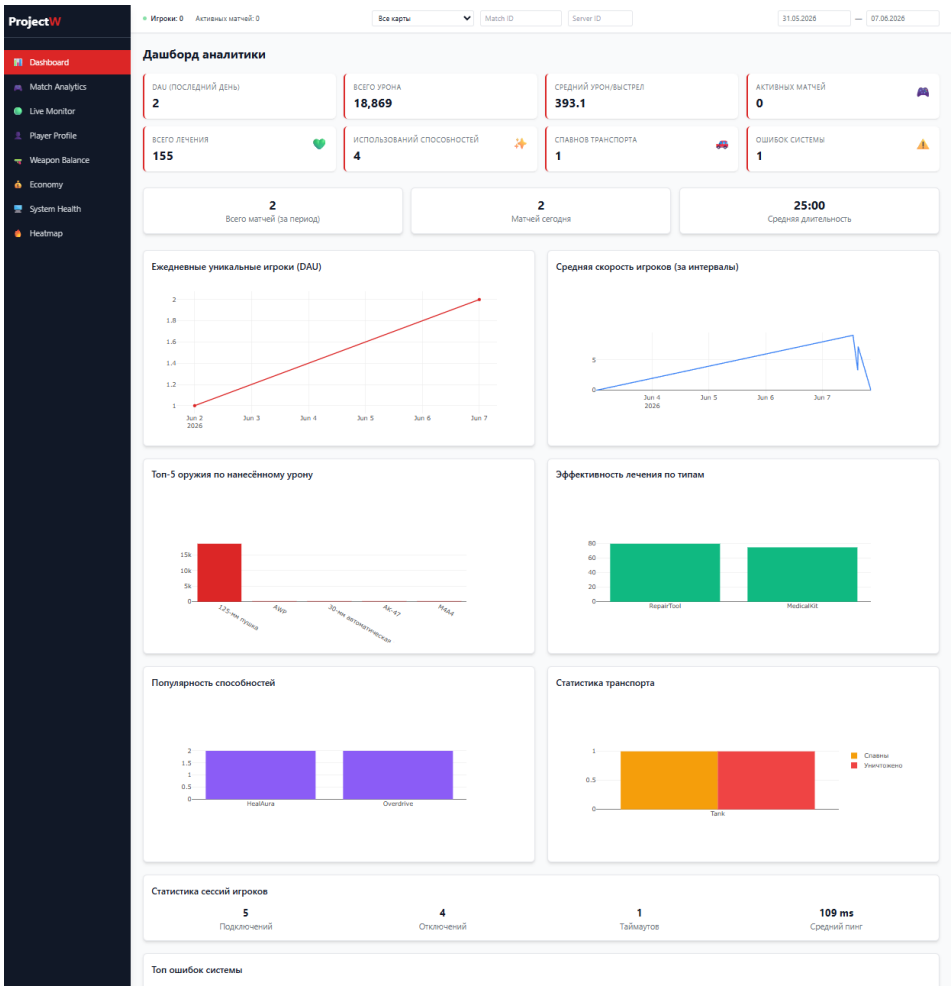


Рисунок 17 – Страница веб-панели при открытии приложения

Страница дашборда агрегирует общие метрики системы. Она загружает данные по восьми параллельным запросам. Визуальное отображение реализовано через библиотеку react-plotly.js, которая предоставляет интерактивные графики с возможностью масштабирования и просмотра значений. KPI-карточки вынесены в отдельный переиспользуемый компонент KPICard, поддерживающий заголовок, значение, единицу измерения, иконку и анимацию загрузки. Для отображения таблицы топ-ошибок системы используется HTML-таблица с условным форматированием. Пример типового функционального компонента с хуками приведён в листинге Б.17. Фрагмент страницы дашборда демонстрирует загрузку данных с помощью useEffect и параллельные запросы.

Страница аналитики матчей отображает список матчей с пагинацией и детальную информацию по выбранному матчу. Пагинация реализована на стороне сервера: при переключении страницы отправляется запрос с параметрами page, limit, а также фильтрами по датам и карте.

Страница баланса оружия предлагает двухколоночный интерфейс, изображенный на рисунке 18 . Слева список оружия с иконками и базовыми характеристиками, справа – детальная аналитика выбранного образца. При выборе оружия загружаются временные ряды нанесённого урона, количества выстрелов и точности критических попаданий за выбранный период. Эти данные отображаются на двух графиках Plotly: линейный график урона и комбинированный. Дополнительно показывается рейтинг оружия по суммарному урону за период, что помогает выявить дисбаланс.

Фрагмент компонента, отвечающий за загрузку деталей оружия при изменении выбранного элемента, приведён в листинге Б.18.

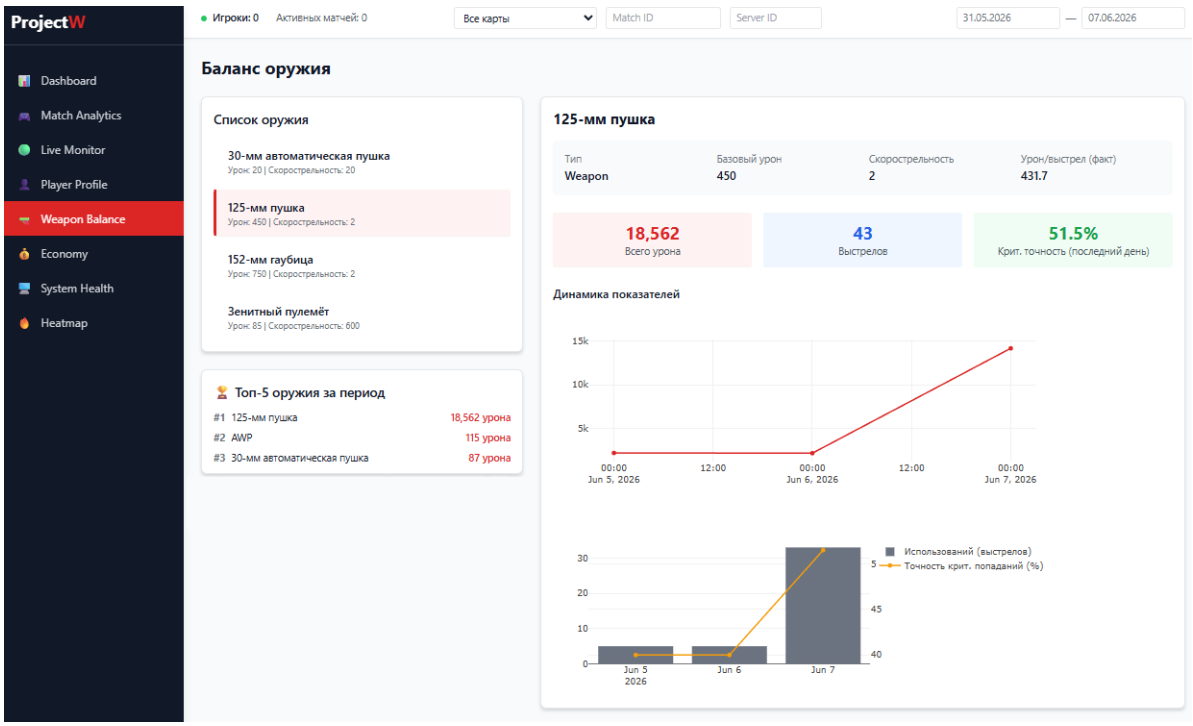


Рисунок 18 – Страница с аналитикой баланса оружия

Тепловые карты реализованы через Plotly.heatmap. Сначала загружается список карт с сервера, затем при выборе карты и типа события отправляется запрос на соответствующий контроллер. Сервер возвращает бины по осям X и Y и матрицу плотности. Компонент преобразует эти данные в формат, ожидаемый

Plotly. Пользователь может переключаться между двумя типами тепловых карт, что позволяет анализировать как горячие точки боёв, так и популярные маршруты перемещения.

Live-монитор использует хук useLiveData для получения текущего состояния серверов, количества онлайн-игроков, активных матчей и ленты событий. Интерфейс мониторинга изображен на рисунке 19 .

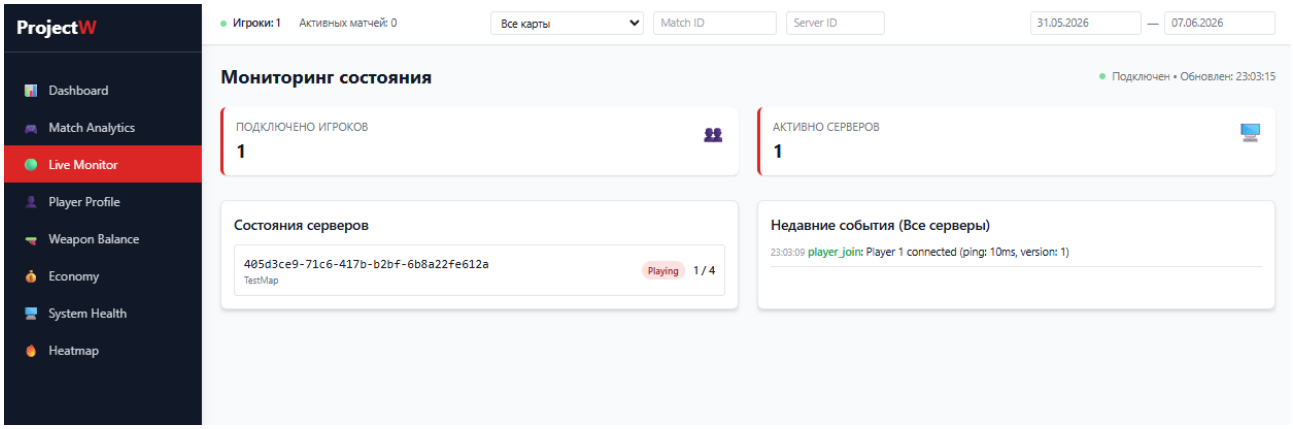


Рисунок 19 – Страница с мониторингом в реальном времени

При получении нового сообщения об изменении количества игроков на сервере компонент обновляет историю нагрузки для выбранного сервера и отображает линейный график из последних 60 измерений. WebSocket-соединение поддерживает отправку состояния для обнаружения разрывов и автоматическое переподключение. Ключевая часть логики обновления истории нагрузки представлена в листинге Б.19.

Страницы экономики, профиля игрока и здоровья системы реализованы по аналогичному принципу: загрузка данных при изменении фильтров, отображение KPI-карточек и графиков.

В ходе реализации веб-панели были решены следующие задачи: обеспечение единообразной обработки ошибок API с выводом уведомлений, синхронизация фильтров между страницами через Redux, оптимизация производительности и избежания ненужных переотрисовок. Результатом стал полнофункциональный инструмент аналитики, позволяющий администраторам и разработчикам игры в реальном времени отслеживать ключевые метрики и выявлять проблемные места в геймдизайне и инфраструктуре.

4. Тестирование

4.1 Тестирование методом черного ящика

Тестирование методом чёрного ящика определяется как методика тестирования, при которой функциональность тестируемого приложения проверяется без учёта внутренней структуры кода, деталей реализации и знания внутренних путей программного обеспечения. Этот тип тестирования полностью основан на требованиях и спецификациях программного обеспечения. В тестировании методом чёрного ящика фокусируются на входах и выходах программной системы, не заботясь о внутреннем устройстве программы. Тестируемым объектом выступает метод RecordMovementBatch контроллера AnalyticsController, отвечающий за приём и сохранение батча событий перемещения игроков.

4.2 Классы эквивалентности

При отправке запроса на сохранение батча событий требуется передать JSON-объект, содержащий поле Events – массив объектов (событий). Каждое событие должно соответствовать заданной структуре (наличие обязательных полей, корректные типы). Количество событий в одном запросе должно быть от 1 до 1000 включительно.

В таблице 2 приведены классы эквивалентности для входных параметров метода. Тест-кейсы для отправленного пакета приведены в таблице 3.

Таблица 2 – Классы эквивалентности для пакета событий

Параметр	**Правильные КЭ**	**Неправильные КЭ**
Тип данных Events	Массив объектов (1)	Не массив (2), null (3), пустой массив (4)
Количество событий	1–1000 событий (5)	0 событий (6), >1000 событий (7)
Структура события	Соответствует ожидаемой JSON-схеме (8)	Не соответствует схеме (9)

Таблица 3 – Тест-кейсы для пакета с событиями

№	КЭ	Входные данные	Выходные данные
0	1, 5, 8	{ «Events»: [{ «EventId»: 1, «PlayerId»: 100, «TimeStamp»: «2025-03-25T10:00:00Z», «PosX»: 10.5, «PosY»: 20.3, ... }] } (1 событие, валидное)	HTTP 200 OK, успешная обработка
1	2	{ «Events»: «not an array»}	HTTP 400 Bad Request
2	3	{ «Events»: null }	HTTP 400 Bad Request
3	4,6	{ «Events»: [] }	HTTP 400 Bad Request
4	7	{ «Events»: [...] } (1001 валидное событие	HTTP 400 Bad Request
5	9	{ «Events»: [{ «PlayerId»: 100 }] } (отсутствует EventId)	HTTP 400 Bad Request

Были составлены тест-кейсы, покрывающие все классы эквивалентности для метода RecordMovementBatch.

4.3 Граничные условия

Граничные условия – это набор ограничений, накладываемых на входные данные. В рамках данного тестирования были выделены границы для количества событий в пакете и для структуры запроса. Границы приведены в таблице 4, тест-кейсы по граничным условиям – в таблице 5.

Таблица 4 – Граничные условия

Границы	Описание	Пример
Физические	То, что не может существовать	Строка отрицательной длины
Логические	Не подходит по смыслу	Скидка >100%
Технологические	Перестает работать по техническим причинам	Целое число больше максимально возможного по типу данных
Произвольные	Техническое задание	Логин не больше 32 символов

Таблица 5 – Тест кейсы для граничных условий

№	Границы	Входные данные	Выходные данные
1	Физические	{}	HTTP 400
2	Логические	«PlayerId»: -1	HTTP 400
3	Технологические	1000 событий	HTTP 200
4	Технологические	1001 событие	HTTP 400
5	Произвольные	«MapName»: «Test/Map#1»	HTTP 200

Были соблюдены все граничные условия, проведено тестирование метода RecordMovementBatch контроллера AnalyticsController.

4.4 Модульное тестирование (Unit-тесты)

Для проверки корректности работы метода RecordMovementBatch были разработаны юнит-тесты на платформе .NET с использованием фреймворков xUnit, Moq и FluentAssertions. Тесты изолируют логику контроллера от реального репозитория IAnalyticsRepository с помощью mock-объектов. Написание тестов позволяет проверить:

– валидацию входных данных;

– корректную обработку успешного сценария;

– обработку исключений при ошибках репозитория.

Код тестов приведён в листинге Б.20.

Все юнит-тесты выполнены успешно. Результаты тестирования представлены на рисунке 20.

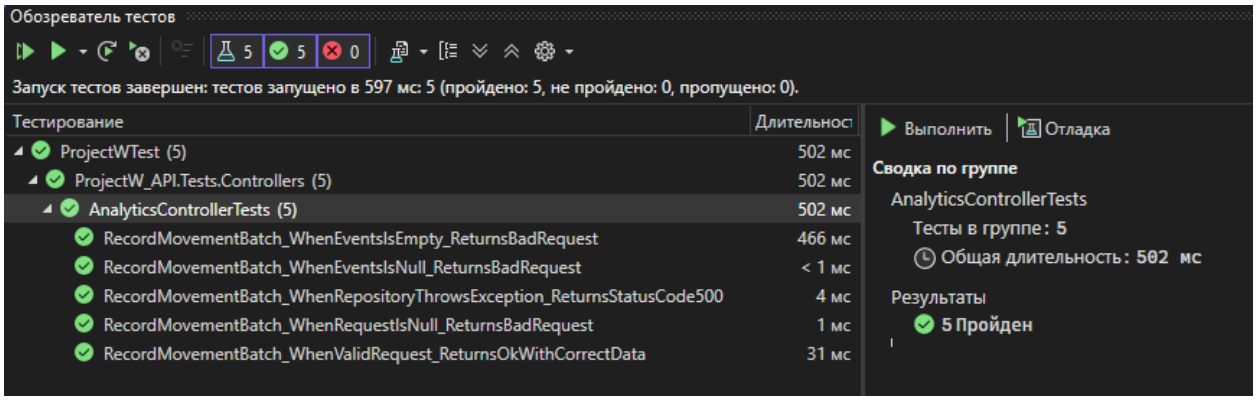


Рисунок 20 – Результаты Unit-тестирования

Было выполнено модульное тестирование метода RecordMovementBatch. Полученные данные подтверждают корректную обработку как успешных сценариев, так и ошибок, возникающих при сохранении данных.

4.5 Анализ качества кода

Для количественной оценки сложности и объёма кода метода RecordMovementBatch была применена метрика Холстеда. Основу метрики составляют четыре измеряемые характеристики программы:

n_1 – количество уникальных операторов;

n_2 – количество уникальных операндов;

N_1 – общее число всех операторов, появляющихся в программе;

N_2 – общее число всех операндов, появляющихся в программе.

Операторы и операнды, использованные в методе, представлены в таблице 6.

Таблица 6 – Операторы и операнды, использованные в методе

№	**Операнды**		**Операторы**	
1	request	4	try	1
2	Events	3	catch	1
3	null	1	if	1
4	_repo	1	return	3
5	ex	2	await	1
6	DateTime.UtcNow	1	new	3
7	Count	1	||	1
8	true	1	!	1
9	false	2	==	1
10	500	1	?.	1
11	«No events provided»	1	.	6
12	«Failed to record batch events»	1	=	9
13	«success»	3	,	6
14	«error»	2	{}	1
15	«count»	1	console.log	1
16	«processed»	1	||	1
17	«sessionId»	1	return	1
18	«details»	1		

Количество уникальных операторов:

$$n_1 = 17.$$

Количество уникальных операндов:

$$n_2 = 18.$$

Общее число операторов:

$$N_1 = 48.$$

Общее число операндов:

$$N_2 = 40.$$

Словарь программы:

$$n = n_1 + n_2 = 35.$$

Длина реализации:

$$N = N_1 + N_2 = 88.$$

Объем программы:

$$V = N \cdot \log_2 n = 88 \cdot \log_2 35 \approx 88 \cdot 5.129 \approx 451.35 \text{ бит.}$$

Сложность программы:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2} = \frac{17}{2} \cdot \frac{40}{18} = 8.5 \cdot 2.2222 \approx 18.888.$$

Усилие на понимание/написание:

$$E = D \cdot V \approx 18.888 \cdot 451.35 \approx 8525.$$

Вывод: объем V <8000, что говорит о том, что код легко читается и понимается. Усилия на поддержку минимальны.

Цикломатическая сложность метода RecordMovementBatch рассчитана по количеству линейно независимых путей и равна 4 (2 логических условия + 1 исключение + 1 базовый путь). Значение ниже рекомендуемого порога (10), что подтверждает простоту и лёгкость тестирования метода.

Метрика Чепина оценивает использование переменных в программе. Для метода RecordMovementBatch:

– вводимые данные (P): request – 1;
– модифицируемые данные (M): отсутствуют – 0;
– управляющие переменные (C): request, Events – 2;
– неиспользуемые переменные (T): отсутствуют – 0.

$$Q = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot 2 + 0.5 \cdot 0 = 1 + 0 + 6 + 0 = 7.$$

Низкое значение метрики (7) указывает на прозрачность кода и отсутствие избыточных переменных.

Разработанные юнит-тесты обеспечивают 100% покрытие операторов и всех ветвей метода, что подтверждается анализом покрытия кода в Visual Studio.

Проведённый анализ качества кода показывает, что метод RecordMovementBatch соответствует высоким стандартам сопровождаемости, тестируемости и надёжности.

Функциональная пригодность (полнота реализации требований): Высокая.

Метод полностью реализует задачу приёма батча событий, проверяет наличие данных и сохраняет их через репозиторий. Возвращает информативные ответы.

Точность (правильность реализации алгоритма): Средняя.

Логика проверки корректна, но отсутствует валидация содержимого каждого события – эта ответственность делегирована репозиторию, что может привести к исключениям.

Защищённость (защита от ошибок пользователя): Средняя.

При пустом массиве возвращается BadRequest. Однако нет проверки на тип элементов массива – клиент может прислать неверные объекты, что вызовет исключение в репозитории.

Устойчивость к дефектам: Высокая.

Метод корректно обрабатывает штатные ситуации (пустой массив) и исключения БД, возвращая соответствующие HTTP-статусы. Приложение не падает.

Понятность (удобство восприятия логики и интерфейса): Высокая.

Ответы содержат понятные сообщения на английском языке, структурированы в JSON, что удобно для клиента.

Анализируемость (простота архитектуры кода): Хорошая.

Код линейный, с минимумом ветвлений, легко читается и анализируется.

Стабильность (сцепление модулей и связность): Средняя.

Метод имеет высокую внутреннюю связность (делает одну задачу), но зависит от репозитория (_repo). Изменения в интерфейсе репозитория могут потребовать правок.

Тестируемость: Высокая.

Метод легко тестируется с использованием mock-объектов для репозитория. Все пути (успех, пустой массив, исключение) покрываются несколькими тестами.

Временная эффективность (время реакции): Высокая.

Метод асинхронный, не блокирует поток. Операции проверки легковесны.

Ресурсоёмкость (требуемый объём памяти): Высокая.

Не создаёт тяжёлых объектов, потребление памяти минимально.

Адаптируемость (зависимость от платформы): Высокая.

Код написан на C# с использованием ASP.NET Core, что обеспечивает переносимость между платформами. Зависимости от специфических API отсутствуют.

Заключение

При выполнении выпускной квалификационной работы была достигнута поставленная цель – разработана серверная архитектура и система телеметрии для многопользовательской игры, обеспечивающие высокую производительность, масштабируемость и сбор аналитических данных в реальном времени. В рамках работы проведён анализ предметной области многопользовательских игр, выявлены ключевые проблемы: необходимость динамического управления игровыми серверами в зависимости от нагрузки, гарантированный сбор телеметрических событий без потерь, эффективное долговременное хранение больших объёмов данных и визуализация аналитики в реальном времени. Выполнен сравнительный анализ архитектурных подходов, включая монолитную архитектуру, фиксированный набор серверов, управление через менеджер серверов и контейнерную оркестрацию, на основе чего обоснован выбор подхода с менеджером игровых серверов как оптимального по соотношению сложности реализации и получаемой автоматизации масштабирования.

Обоснован выбор технологического стека: для серверной части использованы ASP.NET Core, Dapper, RabbitMQ, PostgreSQL и Clickhouse; для клиентской части – Unity с сетевым движком Fishnet, для веб-панели телеметрии – React и Vite, для кэширования – Valkey. Выбранные технологии в совокупности обеспечивают требуемую производительность и масштабируемость. Спроектирована и реализована серверная архитектура, включающая менеджер игровых серверов с автоматическим запуском, мониторингом и остановкой процессов, API-шлюз как единую точку входа с аутентификацией на основе JWT-токенов, асинхронный конвейер сбора телеметрии на базе RabbitMQ и Clickhouse, систему кэширования на Valkey для разгрузки баз данных и ускорения ответов, а также веб-панель телеметрии на React с дашбордами и мониторингом в реальном времени. Разработана реляционная база данных PostgreSQL для хранения пользовательских профилей, статистики, социальных связей и игрового прогресса, а также колоночная база Clickhouse для хранения и

агрегации телеметрических событий, включающая одиннадцать фактовых таблиц и семнадцать материализованных представлений.

Проведено тестирование разработанного программного обеспечения методом чёрного ящика с выделением классов эквивалентности и граничных условий, модульное тестирование с использованием xUnit и Moq, а также выполнен анализ качества кода по метрикам Холстеда, цикломатической сложности и метрике Чепина. Результаты тестирования подтвердили корректность работы системы и соответствие заявленным требованиям. Все функциональные и нефункциональные требования, сформулированные в работе, соблюдены.

Разработанная система способна автоматически управлять игровыми серверами, обрабатывать не менее тысячи телеметрических событий в секунду без потерь, обеспечивать задержку от возникновения события до его отображения в веб-панели не более двух секунд, а также поддерживать горизонтальное масштабирование. Созданный рабочий прототип может быть использован в составе коммерческого игрового проекта для мониторинга состояния серверов, анализа игрового баланса и выявления аномального поведения игроков. Перспективными направлениями дальнейшего развития системы являются внедрение оркестрации контейнеров для автоматического масштабирования инфраструктуры, расширение набора аналитических метрик.

Список использованных источников

1. Горбатов, А. Е. С нуля в геймдев. Как создавать игры и зарабатывать. – М. : Издательство АСТ, 2024. – 288 с.

2. Unity Documentation // Unity Docs [Электронный ресурс]. URL: https://docs.unity.com/ru/ (дата обращения: 20.05.2026).

3. OMG Unified Modeling Language (OMG UML) // Object Management Group [Электронный ресурс]. URL: https://www.omg.org/spec/UML/ (дата обращения: 20.05.2026).

4. FishNet Documentation // Fish-Networking GitBook [Электронный ресурс]. URL: https://fish-networking.gitbook.io/ (дата обращения: 20.05.2026).

5. .NET Documentation // Microsoft Learn (на русском) [Электронный ресурс]. URL: https://learn.microsoft.com/ru-ru/dotnet/ (дата обращения: 20.05.2026).

6. ClickHouse for gaming analytics and telemetry // ClickHouse [Электронный ресурс]. URL: https://clickhouse.com/use-cases/gaming-analytics (дата обращения: 20.05.2026).

7. RabbitMQ Quick Guide: Architecture, Clustering, Scaling, And Deployment Strategies // ScaleGrid Blog [Электронный ресурс]. URL: https://scalegrid.io/blog/rabbitmq-quick-guide/ (дата обращения: 20.05.2026).

8. Dapper Documentation // Learn Dapper [Электронный ресурс]. URL: https://www.learndapper.com/ (дата обращения: 20.05.2026).

9. ValKey Documentation // Valkey.io [Электронный ресурс]. URL: https://valkey.io/documentation/ (дата обращения: 20.05.2026).

10. Григорьев, М. В., Григорьева, И. И. Проектирование информационных систем: учебное пособие для вузов. – М.: Юрайт, 2024. – 278 с

11. ClickHouse Docs // ClickHouse Documentation [Электронный ресурс]. URL: https://clickhouse.com/docs/ru (дата обращения: 20.05.2026).

12. PostgreSQL 18.4 Documentation // PostgreSQL Documentation [Электронный ресурс]. URL: https://www.postgresql.org/docs/current/ (дата обращения: 20.05.2026).

13. PostgreSQL : Документация : 16 // Postgres Professional [Электронный ресурс]. URL: https://postgrespro.ru/docs/ (дата обращения: 20.05.2026).

14. Евдокимов, П. В. C# для UNITY-разработчиков. Практическое руководство по созданию игр. – СПб. : Наука и техника, 2023. – 480 с.

15. ASP.NET Core Documentation // Microsoft Docs [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/aspnet/core/ (дата обращения: 20.05.2026).

16. React – JavaScript-библиотека для создания пользовательских интерфейсов // React.dev (на русском) [Электронный ресурс]. URL: https://ru.react.dev/ (дата обращения: 20.05.2026).

17. Vite Getting Started // Vite.js [Электронный ресурс]. URL: https://vitejs.dev/guide/ (дата обращения: 20.05.2026).

18. Кубарев, В. С. Серверная архитектура и система телеметрии для многопользовательской игры [Электронный ресурс]. URL: https://github.com/ReinFo1/Kubarev_VKR

19. How to Analyze Game Telemetry Data with ClickHouse // OneUptime Blog [Электронный ресурс]. URL: https://oneuptime.com/blog/game-telemetry-clickhouse (дата обращения: 20.05.2026).

20. Установка, настройка и использование RabbitMQ // Timeweb Cloud Tutorials [Электронный ресурс]. URL: https://timeweb.cloud/tutorials/rabbitmq/ustanovka-nastrojka-ispolzovanie-rabbitmq (дата обращения: 20.05.2026).

21. RabbitMQ 4.3 Documentation // VMware Docs [Электронный ресурс]. URL: https://www.rabbitmq.com/docs (дата обращения: 20.05.2026).

22. How to Build a RabbitMQ to ClickHouse Pipeline // OneUptime Blog [Электронный ресурс]. URL: https://oneuptime.com/blog/rabbitmq-clickhouse-pipeline (дата обращения: 20.05.2026).

23. React Quick Start // React.dev [Электронный ресурс]. URL: https://react.dev/learn (дата обращения: 20.05.2026).

Приложение А

(справочное)

Глоссарий

API-шлюз (API Gateway) – единая точка входа для всех клиентских приложений, обеспечивающая маршрутизацию запросов, аутентификацию, авторизацию, агрегацию данных и кэширование.

ASP.NET Core – кроссплатформенный высокопроизводительный фреймворк от Microsoft для создания веб-приложений и API на языке C#.

Актор – в UML – внешняя сущность (пользователь или другая система), взаимодействующая с разрабатываемой системой.

Брокер сообщений (Message Broker) – промежуточное программное обеспечение, обеспечивающее асинхронный обмен сообщениями между компонентами системы (например, RabbitMQ).

Веб-панель телеметрии – веб-интерфейс для визуализации аналитических данных, получаемых из системы сбора телеметрии.

Горизонтальное масштабирование – увеличение пропускной способности системы путём добавления дополнительных узлов (серверов) вместо наращивания мощности одного узла.

Денормализация – намеренное добавление избыточности в базу данных для ускорения чтения за счёт уменьшения числа объединяемых таблиц.

Диаграмма Ганта – тип столбчатой диаграммы, используемый для визуализации плана работ по проекту (задачи, длительность, зависимости).

Жизненный цикл процесса – последовательность состояний игрового сервера: «Остановлен», «Запускается», «Работает», «Останавливается», «Сбой».

Игровой сервер (Game Server) – выделенный процесс, обслуживающий одну игровую сессию, синхронизирующий состояние между игроками и собирающий телеметрию.

Класс эквивалентности – при тестировании – набор входных данных, которые должны обрабатываться системой одинаковым образом.

Контейнеризация – метод виртуализации на уровне операционной системы, позволяющий запускать приложения в изолированных средах (контейнерах) с собственными зависимостями.

Кэш (Cache) – промежуточное высокоскоростное хранилище данных (обычно in-memory), уменьшающее время доступа к часто запрашиваемой информации.

Материализованное представление (Materialized View) – объект базы данных, физически хранящий результат предварительно выполненного запроса и автоматически обновляющийся при изменении исходных данных.

Менеджер игровых серверов (Game Server Manager) – компонент, отвечающий за автоматический запуск, мониторинг и остановку игровых процессов в зависимости от нагрузки.

Метрика Холстеда – количественная мера сложности программы, основанная на подсчёте операторов и операндов.

Многопользовательская игра – компьютерная игра, в которой одновременно участвуют несколько игроков, взаимодействуя через общую серверную инфраструктуру.

Модульное тестирование (Unit testing) – проверка отдельных модулей или методов программного кода в изоляции от остальной системы.

ORM (Object-Relational Mapping) – технология связывания объектов программы с таблицами реляционной базы данных (в работе используется микро-ORM Dapper).

Отказоустойчивость – свойство системы сохранять работоспособность при выходе из строя одного или нескольких её компонентов.

Очередь сообщений – структура данных, реализующая принцип FIFO (первым пришёл – первым вышел), используемая в брокерах сообщений для надёжной доставки.

Пагинация – разбиение набора данных (например, списка матчей) на отдельные страницы для пошагового отображения.

Паттерн Cache-Aside – стратегия кэширования, при которой приложение сначала проверяет наличие данных в кэше, и только при их отсутствии обращается к базе данных.

Протокол UDP (User Datagram Protocol) – транспортный протокол без установления соединения, обеспечивающий низкую задержку и используемый в многопользовательских играх для синхронизации состояния.

React – JavaScript-библиотека для построения пользовательских интерфейсов на основе компонентов и виртуального DOM.

Телеметрия – процесс автоматического сбора, передачи и анализа данных о работе системы и действиях пользователей (в игровом контексте – события перемещений, выстрелов, смертей, транзакций и т. п.).

Цикломатическая сложность – метрика программного кода, определяющая количество линейно независимых путей; используется для оценки сложности тестирования.

ClickHouse – колоночная аналитическая СУБД с открытым исходным кодом, оптимизированная для высокоскоростной вставки и выполнения агрегирующих запросов по временным рядам.

Dapper – микро-ORM для .NET, обеспечивающая прямое выполнение SQL-запросов без автоматического маппинга сложных графов объектов.

FishNet – высокопроизводительный сетевой движок для Unity, поддерживающий управление интересами и синхронизацию большого количества объектов.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на синтаксисе JavaScript, используемый в API и телеметрии.

PostgreSQL – объектно-реляционная система управления базами данных с открытым исходным кодом, поддерживающая расширенные типы данных и целостность ссылок.

RabbitMQ – популярный брокер сообщений, реализующий протокол AMQP и обеспечивающий надёжную доставку, маршрутизацию и очереди сообщений.

UML (Unified Modeling Language) – графический язык визуального моделирования для описания архитектуры и поведения программных систем.

Unity – кроссплатформенная среда разработки игр и приложений реального времени, используемая для создания клиентской части.

Valkey – высокопроизводительное in-memory хранилище ключ-значение (форк Redis), используемое в работе для кэширования.

Vite – быстрый инструмент сборки фронтенд-приложений, использующий нативные ES-модули в режиме разработки.

AMQP (Advanced Message Queuing Protocol) – открытый протокол прикладного уровня для передачи сообщений между приложениями или компонентами, используемый RabbitMQ.

Аутентификация – процесс проверки подлинности пользователя или системы (например, по JWT-токену).

Авторизация – процесс проверки прав доступа аутентифицированного пользователя к определённым ресурсам или действиям.

Буферизация – временное накопление данных в оперативной памяти перед их отправкой или записью, используется в системе телеметрии для пакетной отправки событий.

Граничные условия – значения входных параметров, находящиеся на границе допустимых диапазонов; используются в тестировании методом чёрного ящика.

Инвалидация кэша – процесс удаления или пометки устаревших данных в кэше для обеспечения актуальности при следующем запросе.

Интерполяция – метод сглаживания движения объектов в сетевых играх путём расчёта промежуточных позиций между двумя полученными состояниями.

JWT (JSON Web Token) – компактный URL-безопасный токен, содержащий утверждения (claims), используемый для аутентификации и передачи информации между сторонами.

Кластер – группа серверов (узлов), работающих как единая система для обеспечения отказоустойчивости и масштабируемости.

Логирование – запись событий системы в специальный журнал (лог) для последующего анализа и отладки.

Матчмейкинг (Matchmaking) – процесс автоматического подбора игроков в игровую сессию на основе параметров (карта, рейтинг, уровень и т. п.).

Пакетная отправка (Batching) – группировка нескольких событий в один сетевой запрос для снижения накладных расходов и повышения пропускной способности.

Персистентность – свойство данных сохраняться после завершения работы программы (например, в PostgreSQL или ClickHouse).

Refresh-токен – долгоживущий токен, используемый для получения нового access-токена без повторной аутентификации.

Access-токен – короткоживущий токен, предъявляемый клиентом для доступа к защищённым ресурсам API.

Сигнал жизни (Heartbeat) – периодическое сообщение от компонента системы (игрового сервера) мастер-серверу, подтверждающее его работоспособность.

Синхронизация состояния – процесс поддержания актуальных данных об игровом мире на клиентах и сервере в многопользовательской игре.

Тепловая карта (Heatmap) – визуализация плотности событий на карте уровня (например, смертей или перемещений).

Чёрный список токенов (Token Blacklist) – хранилище идентификаторов токенов, которые были отозваны до истечения их срока действия.

Шардирование – горизонтальное разделение данных на независимые части (шарды), распределённые по разным узлам базы данных.

Экземпляр (Instance) – отдельный запущенный процесс (игровой сервер, контейнер) или копия данных.

TTL (Time To Live) – время жизни записи в кэше или очереди, по истечении которого данные автоматически удаляются.

DAU (Daily Active Users) – метрика количества уникальных пользователей, активно взаимодействовавших с игрой за календарный день.

RESTful API – архитектурный стиль построения веб-API, основанный на HTTP-методах (GET, POST, PUT, DELETE) и статусных кодах.

WebSocket – протокол полнодуплексной связи поверх TCP, используемый для передачи данных в реальном времени (например, в LiveMonitor веб-панели)

Приложение Б

(обязательное)

Листинг программы

Листинг Б.1 – Фрагмент реализации фонового сервиса

```
foreach (var server in activeServers)
{
    try
    {
        if ((now - server.LastHeartbeat).TotalSeconds > 120)
        {
            _logger.LogWarning("Сервер {ServerId} не отправляет
состояние, останавливаем", server.ServerId);
            await
gameServerManager.StopServerAsync(Guid.Parse(server.ServerId));
            await
stateService.RemoveServerAsync(Guid.Parse(server.ServerId));
            if (clickHouse != null)
                await
clickHouse.LogServerEventAsync(Guid.Parse(server.ServerId),
"timeout", server.Map, server.CurrentPlayers, 0,
ServerStatus.Terminated);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError("[CleanupService] Возникла ошибка при
удалении сервера: " + server.ServerId + " " +  ex.Message);
    }
}
```

Листинг Б.2 – Реализация метода для подключения

```
public async Task<IActionResult> JoinQueue([FromBody] JoinRequest
request)
{
    if (string.IsNullOrEmpty(request.Map) || request.PlayerId ==
null)
        return BadRequest("Неверные данные");

    var activeServers = await
_stateService.GetAllActiveServersAsync();
    var available = activeServers.FirstOrDefault(s =>
        s.Map == request.Map &&
        s.CurrentPlayers < s.MaxPlayers &&
        (s.Status == "Playing" || s.Status == "Waiting"));
    if (available != null)
    {
        _logger.LogInformation("Используется запущенный сервер
{ServerId} : {Port}", available.ServerId, available.Port);
        return Ok(new { ip = GetPublicIp(), port = available.Port,
gameId = available.ServerId });
    }
    var newServer = await
_serverManager.StartServerAsync(request.Map, 4);
    await _clickHouse.LogServerEventAsync(newServer.Id, "started",
newServer.Map, 0, newServer.MaxPlayers, ServerStatus.Waiting);
```

```
    _logger.LogInformation("Запущен сервер {ServerId} : {Port}",
newServer.Id, newServer.Port);
    return Ok(new { ip = GetPublicIp(), port = newServer.Port,
gameId = newServer.Id });
}
```

Листинг Б.3 – Фрагмент ServerCommandLineArgs с методом Start

```
    private void Start()
    {
        if (_serverStarted) return;
        _networkManager = GetComponent<NetworkManager>();

        var tugboat = _networkManager.TransportManager.Transport
as Tugboat;
        if (tugboat == null)
        {
            Debug.LogError("Неверный транспорт");
            return;
        }

        ushort? port = GetPortFromCommandLine();
        if (!port.HasValue)
        {
            Debug.LogError("Порт не указан");
            return;
        }

        tugboat.SetPort(port.Value);

        string publicIp = GetPublicIpFromCommandLine();
        if (!string.IsNullOrEmpty(publicIp))
        {
            SetTugboatPublicAddress(tugboat, publicIp);
            Debug.Log($"[Server] IP {publicIp}");
        }
        else
        {
            Debug.LogWarning("Неверный IP.");
        }

        ConfigureLiteNetLibForDocker(tugboat, publicIp);

        _networkManager.ServerManager.OnServerConnectionState +=
OnServerConnectionStateChanged;
        _networkManager.ServerManager.OnRemoteConnectionState +=
OnRemoteConnectionStateChanged;

        _serverStarted = true;
        _networkManager.ServerManager.StartConnection();
        StartCoroutine(CheckServerStarted());
    }
```

Листинг Б.4 – Фрагмент ServerHeartBeat отправки состояния

```
IEnumerator SendHeartbeat()
{
    var data = new HeartbeatData
    {
        ServerId = _serverId ?? "Unknown",
        CurrentPlayers =
_networkManager.ServerManager.Clients.Count,
        Map = GetArgument("-map") ?? "Unknown",
        Status = "Playing"
    };
    string json = JsonUtility.ToJson(data);
    using (var request = new UnityWebRequest(GetArgument("-
masterUrl") ?? "http://192.168.1.103:8080/api/Servers/heartbeat",
"POST"))
    {
        byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(json);
        request.uploadHandler = new UploadHandlerRaw(bodyRaw);
        request.downloadHandler = new DownloadHandlerBuffer();
        request.SetRequestHeader("Content-Type",
"application/json");
        yield return request.SendWebRequest();
```

Листинг Б.5 – Фрагмент реализации подключения в ClientConnector

```
    public void ConnectToGameServer(string serverIP, int
serverPort)
    {
        if (_tugboat == null) return;
        _tugboat.SetClientAddress(serverIP);
        _tugboat.SetPort((ushort)serverPort);
        _networkManager.ClientManager.OnClientConnectionState +=
(args) =>
        {
            Debug.Log($"Client connection state:
{args.ConnectionState}");
            if (args.ConnectionState ==
LocalConnectionState.Stopped)
                Debug.LogError($"Disconnect");
        };
        Debug.Log($"Connecting to {serverIP}:{serverPort}");
        _networkManager.ClientManager.StartConnection();
```

Листинг Б.6 – Фрагмент отправки событий в rabbitmq

```
private void PublishToRabbitMQ<T>(string routingKey, T data)
{
    if (_channel == null || _channel.IsClosed)
    {
        Debug.LogWarning($"Канал закрыт {routingKey}");
        return;
    }
    try
```

```
    {
        string json = JsonUtility.ToJson(data);
        byte[] body = Encoding.UTF8.GetBytes(json);

        var props = _channel.CreateBasicProperties();
        props.Persistent = true;
        props.ContentType = "application/json";
        _channel.BasicPublish(exchange: rabbitExchange,
                              routingKey: routingKey,
                              basicProperties: props,
                              body: body);
    }
    catch (Exception e)
    {
        Debug.LogError($"Не получилось отправить: {e.Message}");
    }
}
```

Листинг Б.7 – Фрагмент репозитория UsersRepository

```
        public async Task<UserProfileGlobalData>
GetUserGlobalStats(long UID)
        {
            using IDbConnection connection = Create();
            var parameters = new { _user_id = UID };
            _logger.LogInformation($"Пытаемся получить данные по
пользователю {UID}");
            UserProfileGlobalData UserProfile = await
connection.QueryFirstOrDefaultAsync<UserProfileGlobalData>(
                "SELECT * FROM
sp_get_user_profile_by_id(@_user_id);",
                parameters
            );
            _logger.LogInformation($"Данные получены, кэшируем
{UID}");
            await StoreProfileAsync(UID, UserProfile);
            return UserProfile;
        }
```

Листинг Б.8 – Фрагмент контроллера аутентификации

```
[Authorize]
[HttpPost("Refresh")]
public async Task<ActionResult<AuthResponse>> Refresh([FromBody]
RefreshRequest refresh)
{
    JwtSecurityTokenHandler handler = new
JwtSecurityTokenHandler();
    var token = handler.ReadJwtToken(refresh.RefreshToken);
    var TokenId = token.Claims.FirstOrDefault(c => c.Type ==
JwtRegisteredClaimNames.Jti).Value;
    var usert = HttpContext.User.Identity as ClaimsIdentity;
    long uid = long.Parse(usert.FindFirst("user_id").Value);
```

```
    AuthResponse authData = await _authService.Refresh(uid,
TokenId);
    if(authData == null)
    {
        return BadRequest();
    }
    return Ok(authData);
}
```

Листинг Б.9 – Фрагмент RabbitMQService с настройкой очередей

```
        public async Task ConfigureQueuesAsync()
        {
            await EnsureConnectionAsync();
            using var channel = await CreateChannelAsync();
            await Task.Run(() =>
channel.ExchangeDeclare(_exchangeName, "topic", durable: true,
autoDelete: false));
            foreach (var (queue, routingKey) in _queueBindings)
            {
                await Task.Run(() => channel.QueueDeclare(queue,
durable: true, exclusive: false, autoDelete: false, null));
                await Task.Run(() => channel.QueueBind(queue,
_exchangeName, routingKey));
                _logger.LogDebug("Queue {Queue} bound to exchange
{Exchange} with key {RoutingKey}", queue, _exchangeName,
routingKey);
            }
        }
```

Листинг Б.10 – Фрагмент реализации потребителя для событий перемещения

```
private async Task StartMovementConsumer(IRabbitMQService
rabbitMQService)
{
    await rabbitMQService.StartConsumerAsync("movement_events",
async (json) =>
    {
        try
        {
            var batch =
JsonConvert.DeserializeObject<MovementBatch>(json);
            if (batch?.Events == null) return;
            using var scope = _serviceProvider.CreateScope();
            var repo =
scope.ServiceProvider.GetRequiredService<IAnalyticsRepository>();
            foreach (var ev in batch.Events)
                await repo.InsertMovementEventsAsync(new
List<MovementEvent> { ev });
        }
        catch (Exception ex) { _logger.LogError(ex, "Ошибка
обработки событий перемещения"); }
    });
}
```

Листинг Б.11 – Фрагмент реализации вставки событий в ClickHouse

```
        public async Task
InsertMovementEventsAsync(List<MovementEvent> events)
        {
            if (events == null || !events.Any()) return;
            using var connection = new
ClickHouseConnection(_clickHouseConnection);
            const string sql = @"
                INSERT INTO ProjectW.Movement (
                    EventId, PlayerId, Instance_ID, TimeStamp,
ActiveModules,
                    ServerId, MapName, MatchId, PosX, PosY, PosZ,
                    RotationYaw, RotationPitch, Speed, Health
                ) VALUES (
                    @EventId, @PlayerId, @Instance_ID, @TimeStamp,
@ActiveModules,
                    @ServerId, @MapName, @MatchId, @PosX, @PosY,
@PosZ,
                    @RotationYaw, @RotationPitch, @Speed, @Health
                )";
            var parameters = events.Select(e =>
_mapper.MapToParameters(e));
            await connection.ExecuteAsync(sql, parameters);
            _logger.LogInformation("Вставка {Count} событий
перемещения", events.Count);
        }
```

Листинг Б.12 – Запрос создания таблицы Users

```
CREATE TABLE Users (
User_ID BIGINT PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
Email VARCHAR(255) NOT NULL UNIQUE,
Password_Hash VARCHAR(255) NOT NULL,
UserRole User_Role_Type NOT NULL DEFAULT 'Player',
Created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP,
Last_login_at TIMESTAMPTZ,
Last_login_IP INET,
Is_banned BOOLEAN NOT NULL DEFAULT FALSE,
Ban_reason TEXT,
Currency_balance INTEGER NOT NULL DEFAULT 0 CHECK
(Currency_balance >= 0),
Nickname VARCHAR(40) NOT NULL UNIQUE,
Avatar_ID INTEGER NOT NULL DEFAULT 1,

FOREIGN KEY (Avatar_ID) REFERENCES Avatars(Avatar_ID)
);
```

Листинг Б.13 – Запрос на создание таблицы глобальной статистики пользователя

```
CREATE TABLE User_global_stats (
User_ID BIGINT PRIMARY KEY,
Total_matches INT NOT NULL DEFAULT 0,
```

```
Wins INT NOT NULL DEFAULT 0,
Losses INT NOT NULL DEFAULT 0,
Total_kills INT NOT NULL DEFAULT 0,
Total_deaths INT NOT NULL DEFAULT 0,
KD_Ratio DECIMAL(3,2) GENERATED ALWAYS AS (
CASE Total_deaths
WHEN 0 THEN Total_kills
ELSE Total_kills/Total_deaths
END
) STORED,
Win_rate DECIMAL(3,2) GENERATED ALWAYS AS (
CASE Wins
WHEN 0 THEN 0
ELSE Wins / (Wins + Losses)
END
) STORED,
Current_rating INT NOT NULL DEFAULT 0,
Max_rating INT NOT NULL DEFAULT 0,
Last_updated TIMESTAMPTZ,

FOREIGN KEY (User_ID) REFERENCES Users(User_ID) ON DELETE CASCADE
);
```

Листинг Б.14 – Запись состояния сервера в кэш

```
public async Task UpdateHeartbeatAsync(string serverId,
HeartbeatRequest data)
{
    var key = $"game_server:{serverId}";
    var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
    var fields = new HashEntry[]
    {
    new("status", data.Status),
    new("map", data.Map ?? ""),
    new("current_players", data.CurrentPlayers),
    new("last_heartbeat", now)
    };
    await _db.HashSetAsync(key, fields);
    await _db.KeyExpireAsync(key, TimeSpan.FromSeconds(60));
    _logger.LogInformation("Heartbeat updated for server
{ServerId}", serverId);
}
```

Листинг Б.15 – Методы, использующиеся для хранения токенов

```
public async Task StoreTokenAsync(TokenGenerationResult refresh)
        {
            _logger.LogInformation($"Сохранение токена у
{refresh.UserId}, id токена: {refresh.TokenId}");
            string key =
$"User:{refresh.UserId}:Tokens:{refresh.TokenId}";
            TimeSpan ttl = refresh.Expiration - DateTime.UtcNow;
```

```
            await _valkey.HashSetAsync(key, refresh.TokenId,
refresh.Token);
            await _valkey.KeyExpireAsync(key, ttl);
        }
        public async Task RevokeToken(long userID, string tokenId)
        {
            _logger.LogInformation($"Отзыв токена у {userID}, id
токена: {tokenId}");
            await
_valkey.HashDeleteAsync($"User:{userID}:Tokens:{tokenId}",
tokenId);
            TimeSpan ttl = TimeSpan.FromDays(7);
            await
_valkey.StringSetAsync($"token_blacklist:{tokenId}", "1", ttl);
        }
        public async Task<bool> IsTokenRevokedAsync(string
tokenId)
        {
            return await
_valkey.KeyExistsAsync($"token_blacklist:{tokenId}");
        }
```

Листинг Б.16 – Метод для сохранения профиля в кэше

```
        public async Task StoreProfileAsync(long UID,
UserProfileGlobalData data)
        {
            _logger.LogInformation($"Попытка сохранить данные в
кэш у {UID}");
            string key = $"User:{UID}:user_profile:";
            TimeSpan ttl = DateTime.UtcNow.AddHours(6) -
DateTime.UtcNow;
            await _valkey.StringSetAsync(key,
JsonSerializer.Serialize(data));
            await _valkey.KeyExpireAsync(key, ttl);
        }
```

Листинг Б.17 – Фрагмент файла Dashboard

```
const fetchDashboardData = async () => {
  setLoading(true);
  try {
    const [dau, weaponStats, errors, sessions, speed] = await
Promise.all([
      analyticsApi.getDailyUniquePlayers({ from: dateRange.from,
to: dateRange.to }),
      analyticsApi.getWeaponStats({ from: dateRange.from, to:
dateRange.to }),
      analyticsApi.getSystemErrors({ from: dateRange.from, to:
dateRange.to }),
      analyticsApi.getSessionStats({ from: dateRange.from, to:
dateRange.to }),
      analyticsApi.getAverageSpeed({ from: dateRange.from, to:
dateRange.to })
```

```
    ]);
    setDauData(dau);
    setWeaponData(weaponStats);
    // ... обновление других состояний
  } catch (err) {
    console.error('Dashboard data fetch error:', err);
  } finally {
    setLoading(false);
  }
};
```

Листинг Б.18 – Загрузка аналитики при выборе оружия

```
useEffect(() => {
  if (!selectedWeapon) return;
  const fetchWeaponData = async () => {
    setLoadingDetails(true);
    try {
      const analytics = await analyticsApi.getWeaponAnalytics({
        weaponId: selectedWeapon.id,
        from: dateRange.from,
        to: dateRange.to
      });
      setWeaponAnalytics(analytics);
      const allStats = await analyticsApi.getWeaponStats({ from:
dateRange.from, to: dateRange.to, matchId });
      const thisWeaponStats = allStats.find(w => w.weapon ===
selectedWeapon.name);
      setWeaponDetails(thisWeaponStats || null);
    } catch (err) {
      console.error('Failed to load weapon analytics:', err);
    } finally {
      setLoadingDetails(false);
    }
  };
  fetchWeaponData();
}, [selectedWeapon, dateRange, matchId]);
```

Листинг Б.19 – Обновление истории нагрузки сервера

```
useEffect(() => {
  if (!selectedServer) return;
  const currentServer = servers.find(s => s.id ===
selectedServer);
  if (!currentServer || typeof currentServer.currentPlayers !==
'number') return;
  const currentLoad = currentServer.currentPlayers;
  const previousLoad = prevLoadRef.current[selectedServer];
  if (previousLoad === undefined || currentLoad !== previousLoad)
{
    setServerHistory(prev => {
      const updated = [...prev, { time: new Date(), players:
currentLoad }];
      return updated.slice(-60);
```

```
    });
    prevLoadRef.current[selectedServer] = currentLoad;
  }
}, [servers, selectedServer]);
```

Листинг Б.20 – Юнит-тесты

```
        [Fact]
        public async Task
RecordMovementBatch_WhenRequestIsNull_ReturnsBadRequest()
        {
            MovementBatch request = null;
            var result = await
_controller.RecordMovementBatch(request);
            var badRequestResult =
result.Should().BeOfType<BadRequestObjectResult>().Subject;
            badRequestResult.Value.Should().BeEquivalentTo(new
            {
                success = false,
                error = "No events provided"
            });
            _mockRepo.Verify(r =>
r.InsertMovementEventAsync(It.IsAny<List<MovementEvent>>()),
Times.Never);
        }
        [Fact]
        public async Task
RecordMovementBatch_WhenEventsIsNull_ReturnsBadRequest()
        {
            var request = new MovementBatch
            {
                Events = null,
                SessionId = "session-123"
            };

            var result = await
_controller.RecordMovementBatch(request);

            var badRequestResult =
result.Should().BeOfType<BadRequestObjectResult>().Subject;
            badRequestResult.Value.Should().BeEquivalentTo(new
            {
                success = false,
                error = "No events provided"
            });
            _mockRepo.Verify(r =>
r.InsertMovementEventAsync(It.IsAny<List<MovementEvent>>()),
Times.Never);
        }
```

Листинг Б.21 – Исходный код метода RecordMovementBatch

```
[HttpPost("MovementBatch")]
```

```
public async Task<IActionResult> RecordMovementBatch([FromBody]
MovementBatch request)
{
    try
    {
        if (request?.Events == null || !request.Events.Any())
        {
            return BadRequest(new { success = false, error = "No
events provided" });
        }

        await _repo.InsertMovementEventAsync(request.Events);

        return Ok(new
        {
            success = true,
            count = request.Events.Count,
            processed = DateTime.UtcNow,
            sessionId = request.SessionId
        });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new
        {
            success = false,
            error = "Failed to record batch events",
            details = ex.Message
        });
    }
}
```

Листинг Б.22 – Файл GameServerManager

```
using System.Diagnostics;
using System.Net;
using System.Net.Sockets;
using ProjectW_API.Data.Models;
using ProjectW_API.Services;

public interface IGameServerManager
{
    Task<GameServer> StartServerAsync(string map, int maxPlayers);
    Task StopServerAsync(Guid serverId);
    Task<IEnumerable<GameServer>> GetAllServersAsync();
}

public class GameServerManager : IGameServerManager, IDisposable
{
    private readonly ILogger<GameServerManager> _logger;
    private readonly IGameServerStateService _stateService;
    private readonly IConfiguration _config;

    private readonly List<GameServer> _servers = new();
```

```
    private readonly Dictionary<Guid, Process> _processes = new();
    private readonly object _lock = new();

    private readonly string _executablePath;
    private readonly string _masterUrl;
    private readonly string _publicIp;
    private readonly int _portStart;
    private readonly int _portEnd;

    public GameServerManager(ILogger<GameServerManager> logger,
IConfiguration config, IGameServerStateService state)
    {
        _logger = logger;
        _config = config;
        _stateService = state;

        _executablePath = _config["GameServer:ExecutablePath"] ??
"GameServer.exe";
        _masterUrl = _config["GameServer:MasterUrl"]
        _publicIp = _config["GameServer:PublicIp"] ??
GetLocalIpAddress();
        _portStart = _config.GetValue<int>("GameServer:PortStart",
7777);
        _portEnd = _config.GetValue<int>("GameServer:PortEnd",
7900);
    }

    public async Task<GameServer> StartServerAsync(string map, int
maxPlayers)
    {
        var serverId = Guid.NewGuid();
        int hostPort = GetNextFreePort();
        var args = $"-port={hostPort} -map={map} -
serverId={serverId} -masterUrl={_masterUrl} -
publicIp={_publicIp}";

        var startInfo = new ProcessStartInfo
        {
            FileName = _executablePath,
            Arguments = args,
            UseShellExecute = false,
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            CreateNoWindow = true,
            WorkingDirectory =
Path.GetDirectoryName(_executablePath) ??
Directory.GetCurrentDirectory()
        };

        Process? process = null;
        try
        {
            process = new Process { StartInfo = startInfo };
```

```
            process.EnableRaisingEvents = true;
            process.OutputDataReceived += (sender, e) =>
OnServerOutput(serverId, e.Data);
            process.ErrorDataReceived += (sender, e) =>
OnServerError(serverId, e.Data);
            process.Exited += async (sender, e) => await
OnServerExited(serverId, sender as Process);
            process.Start();
            process.BeginOutputReadLine();
            process.BeginErrorReadLine();

            var server = new GameServer
            {
                Id = serverId,
                Port = hostPort,
                Map = map,
                MaxPlayers = maxPlayers,
                CurrentPlayers = 0,
                Status = "Waiting",
                StartedAt = DateTime.UtcNow,
                LastHeartbeat = DateTime.UtcNow,
                ContainerId = null
            };

            lock (_lock)
            {
                _servers.Add(server);
                _processes[serverId] = process;
            }

            await _stateService.RegisterServerAsync(server);
            _logger.LogInformation("Запущен сервер {ServerId} на
порту: {Port} с картой: {Map} (PID: {PID})",
                server.Id, hostPort, map, process.Id);

            return server;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Ошибка запуска процесса
{ServerId}", serverId);
            process?.Kill();
            process?.Dispose();
            throw;
        }
    }

    public async Task StopServerAsync(Guid serverId)
    {
        Process? process = null;
        GameServer? server = null;

        lock (_lock)
```

```
        {
            if (_processes.TryGetValue(serverId, out process))
            {
                server = _servers.FirstOrDefault(s => s.Id ==
serverId);
                _processes.Remove(serverId);
                if (server != null) _servers.Remove(server);
            }
        }

        if (process == null || process.HasExited)
        {
            _logger.LogWarning("Сервер {ServerId} остановлен",
serverId);
            return;
        }

        try
        {
            if (!process.CloseMainWindow())
            {
                process.Kill();
            }
            using var cts = new
CancellationTokenSource(TimeSpan.FromSeconds(10));
            try
            {
                await process.WaitForExitAsync(cts.Token);
            }
            catch (OperationCanceledException)
            {
                process.Kill();
                await process.WaitForExitAsync();
            }

            _logger.LogInformation("Остановлен {ServerId}",
serverId);

            if (server != null)
            {
                server.Status = "Stopped";
                await
_stateService.UpdateServerStatusAsync(server);
            }
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Не удалось остановить
{ServerId}", serverId);
        }
        finally
        {
            process.Dispose();
```

```
        }
    }

    public Task<IEnumerable<GameServer>> GetAllServersAsync()
    {
        lock (_lock)
        {
            return
Task.FromResult(_servers.ToList().AsEnumerable());
        }
    }

    private void OnServerOutput(Guid serverId, string? data)
    {
        if (!string.IsNullOrWhiteSpace(data))
        {
            _logger.LogDebug("[Сервер {ServerId}] {Output}",
serverId, data);
        }
    }

    private void OnServerError(Guid serverId, string? data)
    {
        if (!string.IsNullOrWhiteSpace(data))
        {
            _logger.LogWarning("[Сервер {ServerId} ERR] {Error}",
serverId, data);
        }
    }

    private async Task OnServerExited(Guid serverId, Process?
process)
    {
        lock (_lock)
        {
            _processes.Remove(serverId);
            var server = _servers.FirstOrDefault(s => s.Id ==
serverId);
            if (server != null) _servers.Remove(server);
        }

        _logger.LogWarning("Сервер {ServerId} завершил работу
(ExitCode: {ExitCode})",
            serverId, process?.ExitCode);

        await _stateService.RemoveServerAsync(serverId);
    }

    private int GetNextFreePort()
    {
        for (int port = _portStart; port <= _portEnd; port++)
        {
            if (IsPortAvailable(port))
```

```
                return port;
            }
            throw new InvalidOperationException("Нет свободных портов
в диапазоне " + _portStart + "-" + _portEnd);
        }

        private bool IsPortAvailable(int port)
        {
            try
            {
                using (var tcpListener = new
TcpListener(IPAddress.Loopback, port))
                {
                    tcpListener.Start();
                    tcpListener.Stop();
                }
                using (var udpClient = new UdpClient(port))
                {
                    udpClient.Close();
                }
                return true;
            }
            catch (SocketException)
            {
                return false;
            }
        }
        public void Dispose()
        {
            List<Guid> serverIds;
            lock (_lock)
            {
                serverIds = _servers.Select(s => s.Id).ToList();
            }
            foreach (var id in serverIds)
            {
                StopServerAsync(id).GetAwaiter().GetResult();
            }
        }
}
```

Листинг Б.23 – Файл LiveWebSocketHandler

```
using ProjectW_API.Services;
using System.Collections.Concurrent;
using System.Net.WebSockets;
using System.Text;
using System.Text.Json;

namespace ProjectW_API.Services
{
    public interface ILiveEventPublisher
    {
        Task PublishEventAsync(string topic, object data);
```

```
    }

    public class LiveEventPublisher : ILiveEventPublisher
    {
        public async Task PublishEventAsync(string topic, object
data)
        {
            await LiveWebSocketHandler.BroadcastToTopic(topic,
data);
        }
    }
}
public class LiveWebSocketHandler
{
    private readonly IGameServerStateService _stateService;
    private readonly ILogger<LiveWebSocketHandler> _logger;
    private static readonly ConcurrentDictionary<string,
WebSocket> _sockets = new();
    private static readonly ConcurrentDictionary<string,
HashSet<string>> _subscriptions = new();

    public LiveWebSocketHandler(IGameServerStateService
stateService, ILogger<LiveWebSocketHandler> logger)
    {
        _stateService = stateService;
        _logger = logger;
    }

    public async Task Handle(HttpContext context)
    {
        if (!context.WebSockets.IsWebSocketRequest)
        {
            context.Response.StatusCode = 400;
            return;
        }

        var socket = await
context.WebSockets.AcceptWebSocketAsync();
        var socketId = Guid.NewGuid().ToString();
        _sockets[socketId] = socket;

        await Send(socket, new { type = "connected", socketId });
        await SendInitialState(socket);

        await ReceiveLoop(socket, socketId);
    }

    private async Task SendInitialState(WebSocket socket)
    {
        try
        {
            var servers = await
_stateService.GetAllActiveServersAsync();
```

```
            var onlinePlayers = servers.Sum(s =>
s.CurrentPlayers);
            var activeMatches = servers.Count(s => s.Status ==
"online");

            await Send(socket, new
            {
                type = "data",
                topic = "metrics:all",
                data = new
                {
                    onlinePlayers,
                    activeMatches,
                    totalServers = servers.Count(),
                    servers = servers.Select(s => new
                    {
                        id = s.ServerId,
                        name = s.ServerId,
                        currentPlayers = s.CurrentPlayers,
                        maxPlayers = s.MaxPlayers,
                        status = s.Status,
                        map = s.Map,
                        lastUpdate = s.LastHeartbeat
                    })
                }
            });
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Неудалось отправить начальное
состояние");
        }
    }

    private async Task ReceiveLoop(WebSocket socket, string
socketId)
    {
        var buffer = new byte[4096];
        while (socket.State == WebSocketState.Open)
        {
            var result = await socket.ReceiveAsync(new
ArraySegment<byte>(buffer), CancellationToken.None);
            if (result.MessageType == WebSocketMessageType.Text)
            {
                var message = Encoding.UTF8.GetString(buffer, 0,
result.Count);
                try
                {
                    var json =
JsonSerializer.Deserialize<JsonElement>(message);
                    var type =
json.GetProperty("type").GetString();
```

```
                    var topic =
json.GetProperty("topic").GetString();

                    if (type == "subscribe")
                    {
                        _subscriptions.AddOrUpdate(socketId, new
HashSet<string> { topic }, (_, set) => { set.Add(topic); return
set; });
                        await Send(socket, new { type =
"subscribed", topic });
                    }
                    else if (type == "unsubscribe")
                    {
                        if (_subscriptions.TryGetValue(socketId,
out var topics))
                            topics.Remove(topic);
                    }
                    else if (type == "ping")
                    {
                        await Send(socket, new { type = "pong" });
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Неверное сообщение
сокета");
                }
            }
            else if (result.CloseStatus.HasValue)
                break;
        }

        _sockets.TryRemove(socketId, out _);
        _subscriptions.TryRemove(socketId, out _);
    }

    private static async Task Send(WebSocket socket, object data)
    {
        if (socket.State != WebSocketState.Open) return;
        var json = JsonSerializer.Serialize(data);
        var bytes = Encoding.UTF8.GetBytes(json);
        await socket.SendAsync(new ArraySegment<byte>(bytes),
WebSocketMessageType.Text, true, CancellationToken.None);
    }

    public static async Task BroadcastToTopic(string topic, object
data)
    {
        var tasks = new List<Task>();
        foreach (var (socketId, socket) in _sockets)
        {
            if (socket.State != WebSocketState.Open) continue;
```

```
            if (_subscriptions.TryGetValue(socketId, out var
topics) && topics.Contains(topic))
            {
                tasks.Add(Send(socket, new { type = "data", topic,
data }));
            }
        }
        await Task.WhenAll(tasks);
    }
}
```

Листинг Б.24 – Файл LiveServerStatePusher

```
using ProjectW_API.Services;

public class LiveServerStatePusher : BackgroundService
{
    private readonly IServiceProvider _services;
    private readonly ILogger<LiveServerStatePusher> _logger;

    public LiveServerStatePusher(IServiceProvider services,
ILogger<LiveServerStatePusher> logger)
    {
        _services = services;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                using var scope = _services.CreateScope();
                var stateService =
scope.ServiceProvider.GetRequiredService<IGameServerStateService>(
);
                var servers = await
stateService.GetAllActiveServersAsync();
                var onlinePlayers = servers.Sum(s =>
s.CurrentPlayers);
                var activeMatches = servers.Count(s => s.Status ==
"online");

                var metricsData = new
                {
                    onlinePlayers,
                    activeMatches,
                    totalServers = servers.Count(),
                    servers = servers.Select(s => new
                    {
                        id = s.ServerId,
                        currentPlayers = s.CurrentPlayers,
```

```
                        maxPlayers = s.MaxPlayers,
                        status = s.Status,
                        map = s.Map,
                        lastUpdate = s.LastHeartbeat
                    })
                };

                await
LiveWebSocketHandler.BroadcastToTopic("metrics:all", metricsData);
                await
LiveWebSocketHandler.BroadcastToTopic("server:states", servers);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Ошибка отправки состояний");
            }

            await Task.Delay(2000, stoppingToken);
        }
    }
}
```

Листинг Б.25 – Файл LiveData веб панели

```
import { store } from '../store';
import { setConnectionStatus, addEvent, updateLiveMetrics,
addServerState } from '../store/liveSlice';

let socket = null;
let reconnectTimer = null;
const RECONNECT_DELAY = 3000;

export const TOPICS = {
  PLAYER_JOIN: 'player_join',
  PLAYER_LEAVE: 'player_leave',
  PLAYER_DEATH: 'player_death',
  MATCH_START: 'match_start',
  MATCH_END: 'match_end',
  SYSTEM_ERROR: 'system_error',
};

export function connectWebSocket() {
  if (socket && socket.readyState === WebSocket.OPEN) return;

  const protocol = window.location.protocol === 'https:' ? 'wss:'
: 'ws:';
  const wsUrl = `${protocol}//${window.location.host}/ws/live`;
  socket = new WebSocket(wsUrl);

  socket.onopen = () => {
    console.log('WebSocket connected');
    store.dispatch(setConnectionStatus(true));
```

```
    socket.send(JSON.stringify({ type: 'subscribe', topic:
'metrics:all' }));
    socket.send(JSON.stringify({ type: 'subscribe', topic:
'server:states' }));
    socket.send(JSON.stringify({ type: 'subscribe', topic:
'player:events' }));
    socket.send(JSON.stringify({ type: 'subscribe', topic:
'match:events' }));
    socket.send(JSON.stringify({ type: 'subscribe', topic:
'system:events' }));
  };

  socket.onmessage = (event) => {
    try {
      const message = JSON.parse(event.data);

      if (message.type === 'data') {
        const { topic, data } = message;

        switch (topic) {
          case 'metrics:all':
            store.dispatch(updateLiveMetrics(data));
            break;
          case 'server:states':
            if (Array.isArray(data)) {
              data.forEach(server =>
store.dispatch(addServerState(server)));
            }
            break;
          case 'player:events':
          case 'match:events':
          case 'system:events':
            store.dispatch(addEvent({
              id: data.id,
              type: data.type,
              playerId: data.playerId,
              matchId: data.matchId,
              timestamp: data.timestamp,
              data: data.data
            }));
            break;
          default:
            console.debug('Неподдержанный:', topic, data);
        }
      } else if (message.type === 'pong') {
        console.debug('HB');
      }
    } catch (err) {
      console.error('WebSocket ошибка:', err);
    }
  };

  socket.onerror = (error) => {
```

```
    console.error('WebSocket ошибка:', error);
    store.dispatch(setConnectionStatus(false));
  };

  socket.onclose = () => {
    console.log('WebSocket отключен');
    store.dispatch(setConnectionStatus(false));
    if (reconnectTimer) clearTimeout(reconnectTimer);
    reconnectTimer = setTimeout(connectWebSocket,
RECONNECT_DELAY);
  };
}

export function disconnectWebSocket() {
  if (reconnectTimer) clearTimeout(reconnectTimer);
  if (socket) {
    socket.close();
    socket = null;
  }
}
```

Листинг Б.26 – Файл FilterSlice

```
import { useEffect, useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { setMapName, setServerId, setMatchId } from
'../../store/filtersSlice'
import { analyticsApi } from '../../api/analyticsApi'

export function FilterBar() {
  const dispatch = useDispatch()
  const { mapName, serverId, matchId } = useSelector((state) =>
state.filters)
  const [maps, setMaps] = useState([])
  const [servers, setServers] = useState([])

  useEffect(() => {
    analyticsApi.getMapsList().then(setMaps).catch(console.error)
  }, [])

  return (
    <div className="flex gap-3">
      <select
        value={mapName}
        onChange={(e) => dispatch(setMapName(e.target.value))}
        className="border rounded px-2 py-1 text-sm"
      >
        <option value="">Все карты</option>
        {maps.map((m) => (
          <option key={m.id} value={m.name}>
            {m.name}
          </option>
        ))}
```

```
      </select>
      <input
        type="text"
        placeholder="Match ID"
        value={matchId}
        onChange={(e) => dispatch(setMatchId(e.target.value))}
        className="border rounded px-2 py-1 text-sm w-40"
      />
      <input
        type="text"
        placeholder="Server ID"
        value={serverId}
        onChange={(e) => dispatch(setServerId(e.target.value))}
        className="border rounded px-2 py-1 text-sm w-32"
      />
    </div>
  )
}
```