Санкт–Петербургский государственный университет

ТЕРЕЩЕНКО Дмитрий Владиславович

Выпускная квалификационная работа

Проектирование сервиса управления данными в специализированной распределенной вычислительной системе SPD Online filter

Уровень образования: магистратура
Направление 02.04.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа «Распределенные вычислительные технологии» №21/5827/1

Научный руководитель:
доцент, Кафедра компьютерного моделирования и многопроцессорных систем,
к.ф. – м.н. Корхов Владимир Владиславович

Руководитель от организации:
ведущий программист, МЛИТ, ОИЯИ,
к.т.н. Олейник Данила Анатольевич

Рецензент:
ведущий программист, МЛИТ, ОИЯИ,
к.т.н. Петросян Артём Шмавонович

Санкт-Петербург
2023

Saint–Petersburg State University

TERESHCHENKO Dmitrii Vladislavovich

Graduation project

Designing a Data Management Service in a Specialized Distributed Computing System SPD Online Filter

Level of education: Master level
Main field of study 02.04.02 «Fundamental Informatics and Information Technology»
Academic programme title «Distributed Computational Technologies»
№21/5827/1

Scientific supervisor:
PhD in Computer Science
V.V.Korkhov

Supervisor from organisation:
lead software developer, MLIT, JINR
PhD in Technical science
D.A.Oleynik

Reviewer:
lead software developer, MLIT, JINR
PhD in Technical science
A.Sh.Petrosyan

Saint–Petersburg
2023

Содержание

Глоссарий 4

Введение 7

Обзор существующих решений 14

Постановка задачи 18

Глава 1. Анализ данных 19

1.1. Организация исходных данных 19

1.2. Организация промежуточных и выходных данных 20

1.3. Потоки данных 21

Глава 2. Общая архитектура 23

2.1. Функциональные требования 23

2.2. Концептуальная архитектура 24

Глава 3. Разработка требований 26

3.1. Требование к БД 26

3.1.1 Концептуальная модель данных 26

3.1.2 Дополнительные механизмы 30

3.2. Требования к сервисам 31

3.2.1 Сервис dsm-register 31

3.2.2 Сервис dsm-manager 33

3.2.3 Сервис dsm-inspector 34

Глава 4. Прототипирование 37

4.1. Выбор стека технологий 37

4.2. Прототип сервиса dsm-manager 40

4.2.1 Структура проекта 40

4.2.2 Слой работы с БД 41

4.2.3 Слой работы API 42

4.3. Прототип сервиса dsm-register 44

Глава 5. Тестирование 47

5.1. Подготовка инфраструктуры 47

5.2. Проверка работоспособности 51

5.3. Разработка тест-кейсов 53
Заключение 56
Список литературы 57
Приложения 61

Глоссарий

Термины	**Определения**
БАК (LHC)	Большой Адронный Коллайдер (Large Hadron Collider)
БД	База данных
Веб-фреймворк	Программная платформа для написания веб-приложений
Задание (task)	Единица рабочей нагрузки по обработке блока данных
Задача (job)	Единица рабочей нагрузки по обработке элемента (или нескольких элементов) блока данных
МЛИТ	Лабораторией Информационных Технологий имени Мещерякова
ОИЯИ	Объединенный Институт Ядерных Исследований
СУБД	Система управления базами данных
ФВЭ	Физика Высоких Энергий
ADC (АЦП)	Analog-to-digital converter (Аналого-цифровой преобразователь). Устройство, преобразующее входной аналоговый сигнал в дискретный код (цифровой сигнал).
ALICE	A Large Ion Collider Experiment
AMQP	Advanced Message Queuing Protocol. Протокол прикладного уровня для передачи сообщений между компонентами системы.
API	Application Programming Interface. Описание способов взаимодействия одной компьютерной программы с другими.
ASGI	Asynchronous Server Gateway Interface. Клиент-серверный протокол взаимодействия веб-сервера и приложения.

ATLAS	A Toroidal LHC Apparatus
CLI	Command Line Interface
DAO	Data access object
DAQ	Data acquisition
Dataset	Логический набор данных
DDD	Domain-Driven Design
DDL	Data Definition Language
DIP	Dependency inversion principle. Один из принципов SOLID.
DLQ	Dead letter queue
DTO	Data trasfer object
ER-диаграмма	Схема «сущность-связь», где показано, как разные «сущности» связаны между собой внутри системы.
Firewall	Программный или программно-аппаратный элемент компьютерной сети, осуществляющий контроль и фильтрацию проходящего через него сетевого трафика в соответствии с заданными правилами.
FTS	File Transfer System
GFAL	Grid File Access Library
Git	Распределённая система управления версиями.
GitLab	Веб-приложение и система управления репозиториями программного кода для Git.
GRID	Является географически распределённой инфраструктурой, объединяющей множество ресурсов разных типов, доступ к которым пользователь может получить из любой точки, независимо от места их расположения.

HTTP	HyperText Transfer Protocol. Протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.
JSON	JavaScript Object Notation. Текстовый формат обмена данными, основанный на JavaScript
LCG	LHC Computing Grid
LFC	LCG File catalogue
NICA	Nuclotron based Ion Collider fAcility
ORM	Object-Relational Mapping. Технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования.
REST	Representational State Transfer. Архитектурный стиль / набор правил взаимодействия компонентов распределённого приложения в сети.
SOLID	Аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании
SPD	Spin Physics Detector
SRM	Storage Resource Manager
SSH	Secure Shell. Сетевой протокол прикладного уровня, позволяющий производить удалённое управление операционной системой и туннелирование TCP-соединений.
URL	Uniform Resource Locator
URN	Uniform Resource Name
WLCG	Worldwide LHC Computing Grid

Введение

На базе Объединённого института ядерных исследований (ОИЯИ) (г.Дубна, Россия) [1] с целью изучения свойств плотной барионной материи создаётся новый ускорительный **комплекс NICA** (Nuclotron based Ion Collider fAcility) [2] (см. рис. 1). После того, как коллайдер NICA будет запущен, исследователи из научных организаций смогут воссоздать в лабораторных условиях особое состояние вещества, в котором пребывала наша Вселенная в первые мгновения после Большого Взрыва, – кварк-глюонную плазму.

Рис. 1: Инфраструктура комплекса NICA [3].

Одним из стратегически важных инфраструктурных проектов, с точки зрения долговременного научного плана ОИЯИ, является комплекс NICA для спиновой физики на поляризованных пучках – детектор **SPD (Spin Physics Detector)** [4] (см. рис. 2). Основная цель эксперимента – проверка основ квантовой хромодинамики путем изучения поляризованной структуры нуклона и спиновых явлений при столкновении продольно и поперечно поляризованных протонов и дейтронов с энергией центра масс до 27 ГэВ. и светимостью до 10^{32} см$^{-2}$ с$^{-1}$. Детектор SPD задуман как универсальный 4π-спектрометр, основанный на современных технологиях.

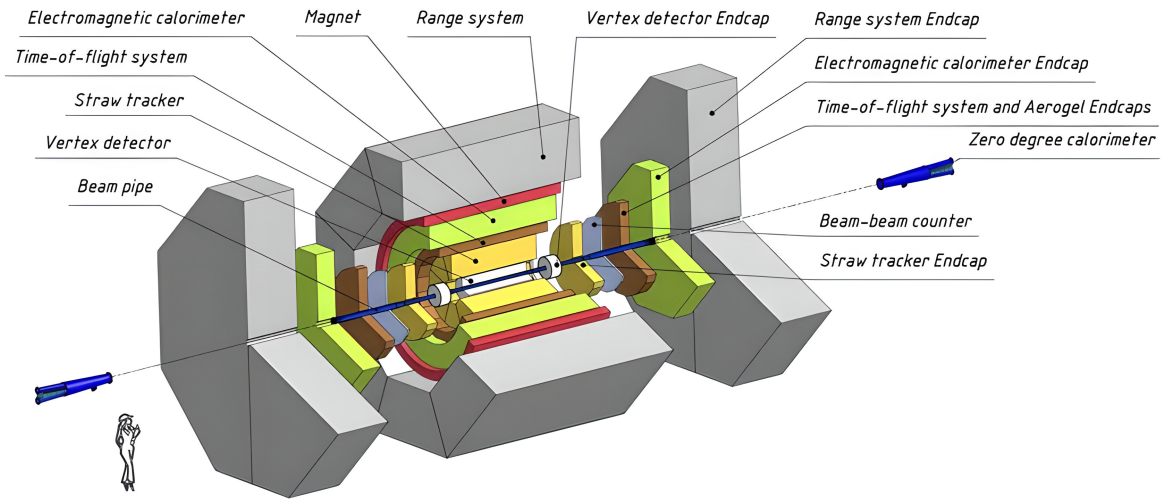

Рис. 2: Экспериментальная установка SPD [5].

Эксперименты, проводимые на ускорителях, предоставляют возможность исследования взаимодействий ускоренных частиц между собой или с веществом. Такие взаимодействия называются «событием» (см. рис. 3).

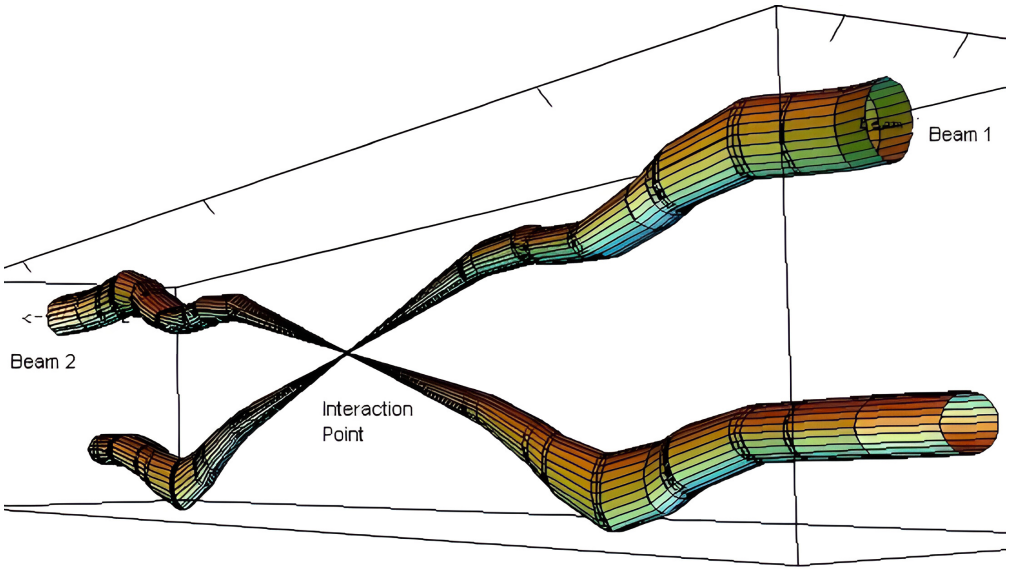

Рис. 3: Иллюстрация «события» в коллайдере [6].

Каждое событие не зависит от другого и поэтому может обрабатываться независимо. В обработке данных ФВЭ (Физики Высоких Энергий) событие является наименьшей единицей.

Регистрацию событий осуществляет чувствительный сенсор. Количество сенсоров в современной физической установке измеряется сотнями

тысяч (в SPD $\sim$ 500000). Сигналы с сенсоров собираются **DAQ (Data acquisition)** [7] (см. на рис. 4 «классическую» схему). Ключевая задача классического DAQ состоит в том, что бы по триггерному сигналу (получаемому со специального сенсора) зафиксировать все текущие сигналы с других сенсоров (иными словами – сделать «снимок»). Такой снимок и есть первоначальный набор данных о событии.

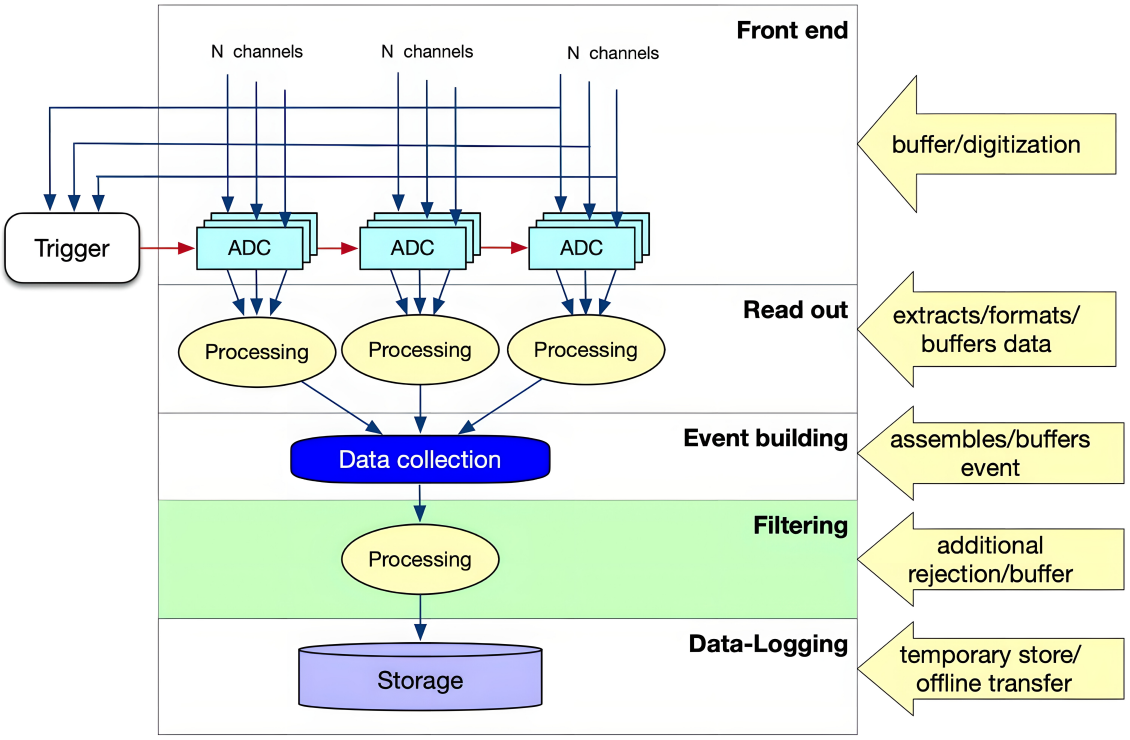

Рис. 4: Схема «классического» DAQ в эпоху БАК [8].

Таким образом, важной частью «классической» схемы DAQ является триггер. В общем смысле, он нужен для указания на то, когда наступает «подходящий» по некоторыми критериям момент для отбора данных, что позволяет не сохранять весь поток данных с сенсоров.

Ввиду сложности и широты изучаемых процессов невозможно построить критерии отбора данных на аппаратном уровне, поэтому для детектора SPD реализовать аппаратную триггерную систему также невозможно. Общее количество каналов регистрации в установке SPD, как уже было сказано выше, составляет около 500000. В связи с этим, с учетом ожидаемой максимальной частоты столкновений частиц в коллайдере около

3 МГц, суммарный поток данных с детектора можно оценить как 20 ГБ/с, что эквивалентно 200 ПБ/год (для эксперимента предполагается выделить 30% пучкового времени коллайдера). Это представляет серьезную проблему для вычислительной инфраструктуры эксперимента и требует разработки новых методов и подходов для подготовки и обработки экспериментальных данных.

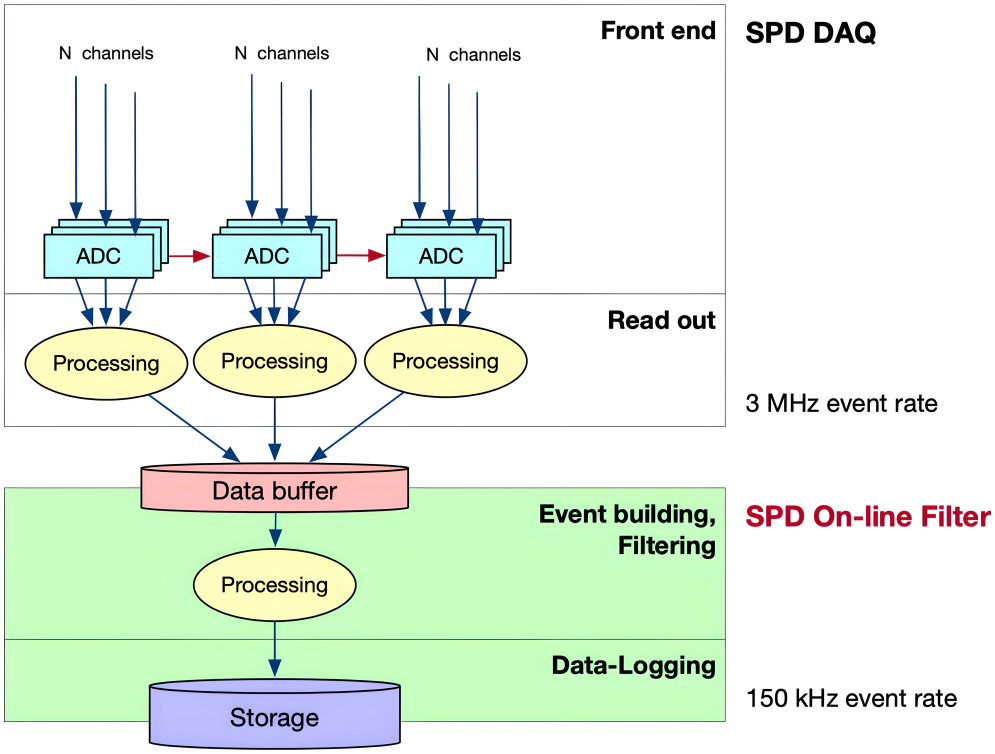

Рис. 5: Схема «triggerless» DAQ [8].

Схема нового подхода «triggerless» DAQ изображена на рис. 5. И в том и в другом случае на выходе из DAQ получаются наборы сигналов с сенсоров, организованных во временные блоки, превосходящие по времени период между событиями (в одном таком блоке предполагается несколько событий). Но, в случае с триггерной системой, эти сигналы сразу же ассоциированы с каким либо событием, а в случае с безтриггерной системой из полученного набора агрегированных с разных сенсоров сигналов нужно сначала выявить события, а затем уже отобрать нужные. Для реализации безтриггерной системы потребуется специальная вычислительная установка под названием **«SPD On-line Filter»** [8].

Данная вычислительная система должна решать следующие задачи:

- Раскодировать данные, полученные от DAQ (каждый детектор предоставляет блоки данных в собственном формате)
- Выделить события;
- Отфильтровать «скучные» события и оставить только «интересные»;

- Упорядочить выходные данные, объединив события в файлы, а файлы в наборы данных для дальнейшей обработки;
- Подготовить данные для системы контроля качества данных и т.п.

Перечисленные выше функции не являются законченным списком, т.к. нельзя изначально определить все те преобразования, которые могут быть необходимы над полным потоком данных. Именно поэтому в разрабатываемой системе отделено управление обработкой непосредственно от самой обработки.

С этой целью предполагается следующая архитектура системы (см. рис. 6):

- Высокоскоростное хранилище входных данных, записанных DAQ;
- Буфер для промежуточных данных и данных, подготовленных к передаче в долгосрочное хранилище и будущей обработке;
- Комплекс промежуточного программного обеспечения для автоматизации рабочего процесса. В него будет входить следующие компоненты:
  - *Система управления данными* (регистрация новых данных, каталогизация/структуризация, контроль целостности);
  - *Система управления процессами обработки* (формирование и контроль исполнения этапов обработки данных);
  - *Система управления нагрузкой* (реализация этапов обработки, посредством формирования и выполнения необходимого количества задач для обработки набора данных);

* *Pilot* (агентское приложение, работающее на вычислительном узле и исполняющее задачи, поставляемые от системы управления нагрузкой).

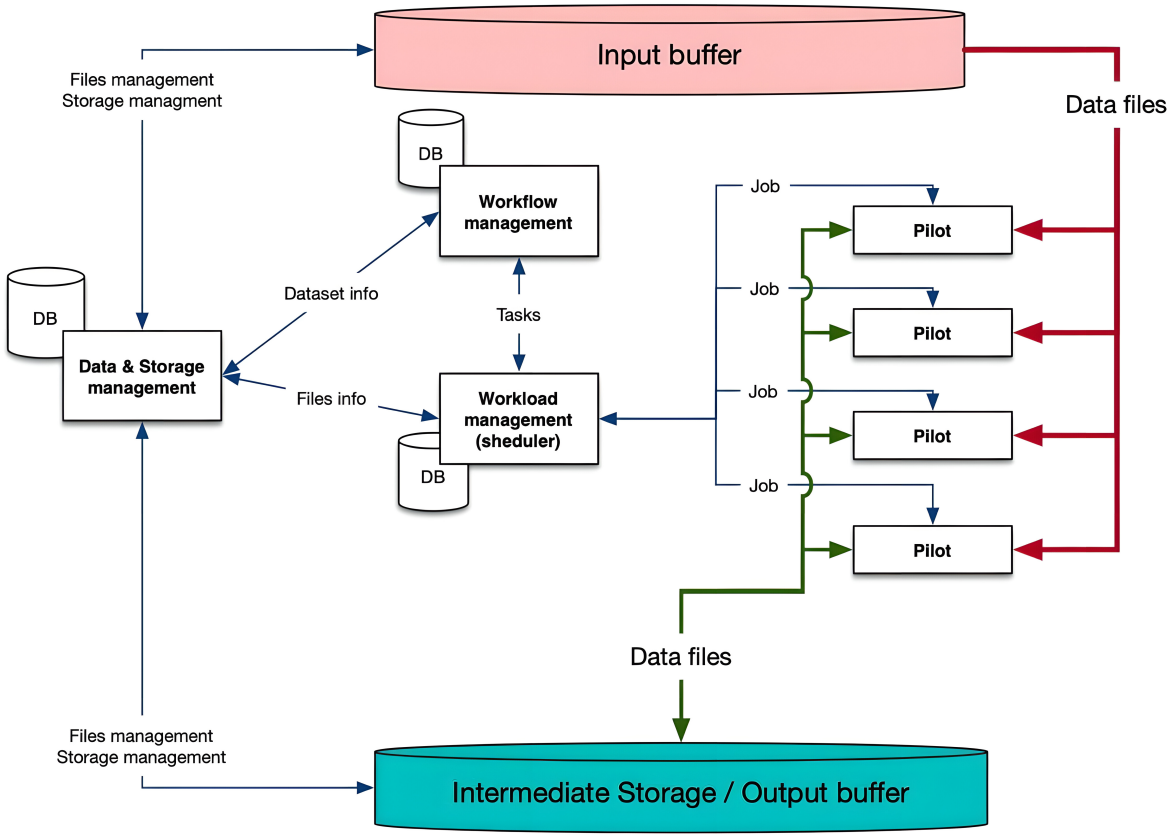

Рис. 6: Архитектура SPD On-line Filter.

Система управления данными. Исходя из поставленной задачи для «SPD On-line Filter», в системе видится довольно интенсивная циркуляции следующих данных:

- SPD DAQ после разбивки по временным блокам сигналов с сенсоров передаёт данные на входной буфер SPD On-Line Filter в виде файлов согласованного размера;
- На каждом шаге обработки данных Pilot-ы будут создавать вторичные данные;

- Система управления процессом обработки создает и удаляет промежуточные и конечные наборы данных;
- Pilot-ы читают и записывают файлы на хранилище;
- Система управления нагрузкой «заполняет» наборы информацией о результирующих файлах.

Таким образом, необходима специализированная система осуществляющая функции каталогизации данных и реализующая управление данными на хранилищах. При этом требуется заложить в стратегию управления данными следующее:

- Абстрагирование от формата данных источника DAQ;
- Возможность логической группировки данных (не релевантное к уровню физического хранения);
- Отсутствие избыточности при организации наборов (контроль лишних реплик);
- Отделение метаданных от физического хранения данных (каталогизация);
- Учёт состояния данных с точки зрения процесса обработки;
- Контроль согласованности информации в каталоге по отношению к входному и выходному хранилищу.

Требования по скорости работы на текущем этапе невозможно задать. Параллельно сейчас строится имитационная модель установки, которая должна позволить определить предварительные производственные характеристики всей системы.

Обзор существующих решений

На текущий день задача управления данными в области распределенных вычислений решается либо для GRID, либо для облачных инфраструктур. Во втором случае в основном преобладают коммерческие решения, которые не могут быть применены в силу отличительной специфики рассматриваемой предметной области. GRID-решения проходили путь эволюции в основном в рамках проекта WLCG (Worldwide LHC Computing Grid), и реализации систем распределенной обработки данных для экспериментов на БАК (Большой Адронный Коллайдер).

Одним из первых рабочих решений по управлению данными в GRID стала система **LFC** (LCG File catalogue) [10], разработанная для проекта WLCG. В сущности это многокомпонентная система, предоставляющая услуги хранилища метаданных файлов и расположения реплик. Доступ к данным осуществляется через иерархическое представление файлов с семантикой, подобной UNIX. В LFC представлены:

- Сопоставление логического имени файла (LFN) с URL-адресом хранилища (SURL);
- Авторизация в своем логическом пространстве имен на основе режимов файлов UNIX, списков контроля доступа POSIX и различных видов строгой аутентификации, включая X509;
- Наборы утилит и клиентов, предоставляющих возможность работы с каталогом и хранилищами.

Но этого функционала явно недостаточно для системы управления данными в «SPD Online Filter» (например, нет функций управления наборами данных). В силу того, что данное решение на данным момент практически не поддерживается, отсутствует возможность его расширения.

Решение LFC шло наряду с компонентами **SRM** (Storage Resource Manager) [11], как универсальный интерфейс к системам хранения в GRID, и **GFAL** (Grid File Access Library) [12] – клиент для передачи файлов и работы с хранилищем. Но они сейчас также не поддерживаются.

В рамках эксперимента ALICE (A Large Ion Collider Experiment) на БАК был разработан компонент **O2** [13] для поддержки функции максимального уменьшения объема потока данных, считываемых с детектора, на как можно более раннем этапе (см. рис. 7). Такая цель была достигнута за счет полной многоэтапной реконструкции данных, происходящей синхронно со сбором данных. Это в свою очередь привело к очень тесной интеграции с DAQ. Исходя из этого, данное решение не подходит для проекта SPD On-Line filter, т.к. в нём система управления данными задумывается как более универсальная и легковесная.

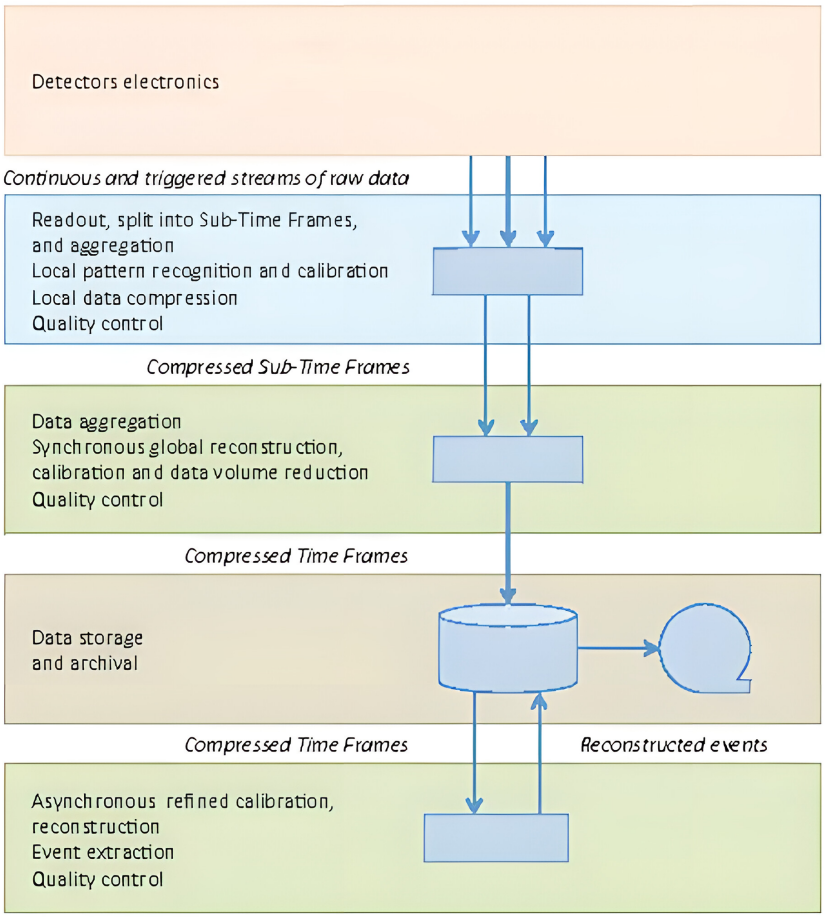

Рис. 7: Вычислительная модель ALICE / O2 [13].

По похожему принципу, исходя из последних публикаций, в рамках эксперимента **LHCb** [14] была разработана система управления данными (см. рис. 8), которая также основана на тесной интеграции с DAQ и полной реконструкции событий.

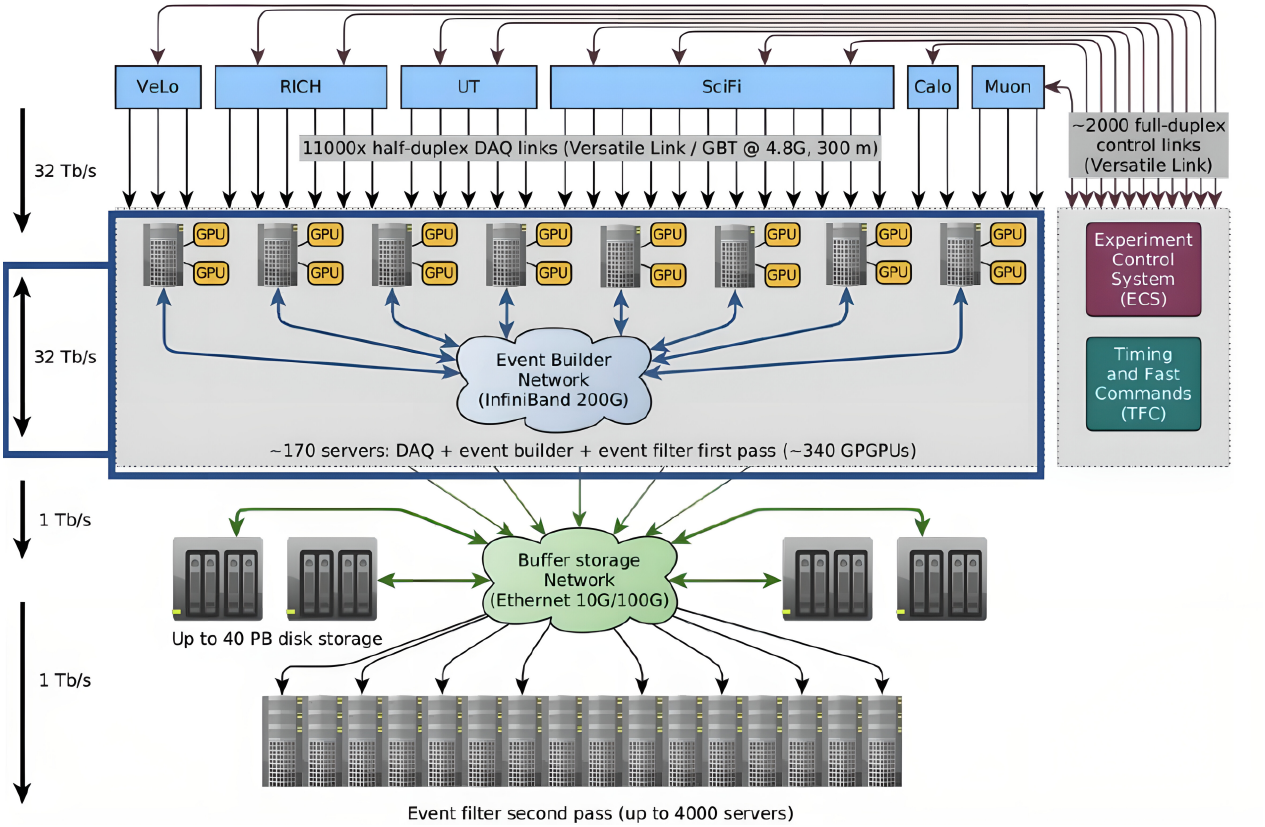

Рис. 8: Система считывания данных LHCb [15].

Самым известным и актуальным на сей день решением для управления данными в распределенной вычислительной среде является программная платформа **Rucio** [9]. Изначально она была разработана в соответствии с требованиями эксперимента ATLAS (A Toroidal LHC Apparatus) и по сей день постоянно расширяется для поддержки экспериментов на БАК и за его пределами. Rucio предоставляет функциональные возможности для организации, управления и доступа к большим объемам данных с использованием настраиваемых политик. В задачи данной платформы входит (см. рис. 9):

- Хранение всех экспериментальных данных в распределенной среде;
- Логическая группировка данных с помощью вложенных наборов файлов;
- Хранение мета-информации о данных, связанной с физическим расположением, с их состоянием, отношением к задаче/заданию, а также с репликацией;

- Поддержка различных протоколов передачи данных и интерфейсов к системам хранения при помощи подключаемых модулей;
- Авторизация и аутентификация пользователей, взаимодействующих с системой;
- Восстановление данных;
- Адаптивная репликация.

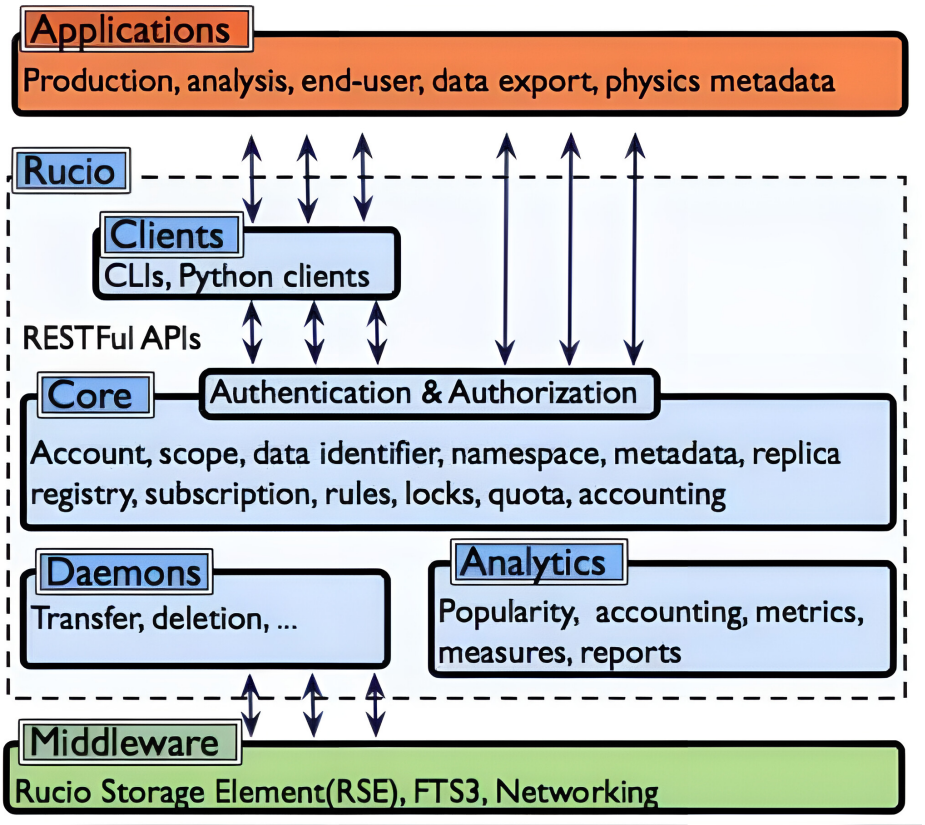

Рис. 9: Основные компоненты Rucio [9].

Исходя из этого может показаться, что Rucio отлично подходит под задачи проекта. Но это не совсем так. Во-первых, в силу большой избыточности, что потребует отключение более чем половины его функционала. Во-вторых, оставшиеся полезная функциональность обладает своей спецификой, а это означает сложность её встраивания в требуемый процесс обработки данных.

Остальные решения из этой группы не подлежат рассмотрению, т.к. имеют малое кол-во публикаций и/или слабую документацию.

Постановка задачи

Новая специализированная вычислительная система «SPD On-Line filter» разрабатывается командой Лаборатории информационных технологий имени Мещерякова (МЛИТ) совместно со студентами кафедры «Компьютерного моделирования и многопроцессорных систем» факультета «Прикладной математики и процессов управления» Санкт-Петербургского государственного университета. Целью данной дипломной работы послужило проектирование системы управления данными, куда вошли следующие задачи:

- Изучение организации многоступенчатой обработки потока данных в системе «SPD On-Line filter»;
- Проработка требуемой функциональности и разработка концептуальной архитектуры системы;
- Проектирование внутренних составляющих каждой из компонент системы управления данными;
- Проектирование внутренних и внешних интерфейсов взаимодействия;

- Выбор стека технологий и частичное прототипирование;
- Подготовка к тестированию работоспособности полученной системы на инфраструктуре МЛИТ ОИЯИ (настройка окружения, разработка тест-кейсов).

Глава 1. Анализ данных

1.1 Организация исходных данных

Для дальнейшей структуризации информации о хранящихся данных в системе в первую очередь требуется разобраться с тем, какие исходные данные планируется хранить и как они организованы.

Система сбора данных DAQ группирует данные с сенсоров следующим образом (см. рис. 10):

- *Период набора* – ассоциирован с физической задачей. Состоит из набора ранов (run). Измеряется неделями. Имеет дату начала, дату окончания;
- *Ран* – интервал, когда условия эксперимента неизменны (например, калибровки). Измеряется часами. Состоит из фреймов;
- *Фрейм* – нумерованный блок данных с сенсоров. Может содержать информацию о множестве событий. Продолжительность – секунды. По объему делится на чанки (файлы);
- *Файл* – количество данных для обработки одной задачей.

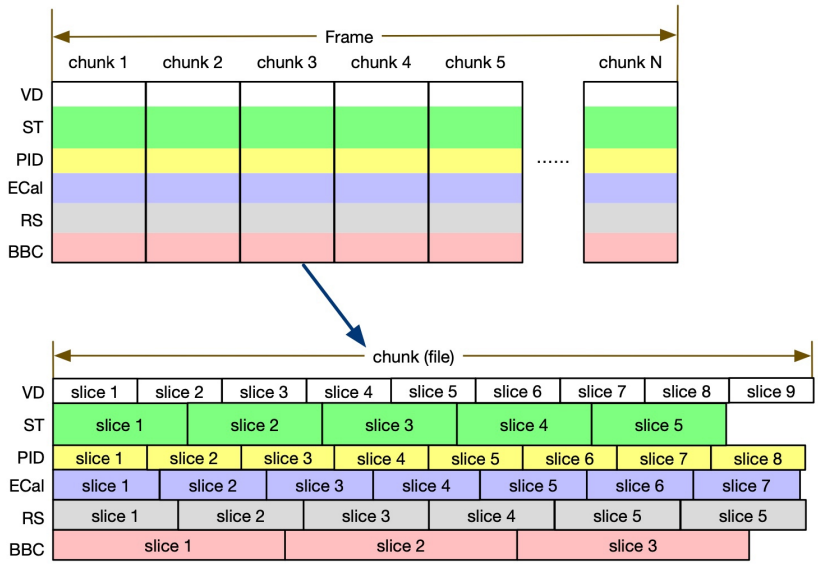

Рис. 10: Организация входных данных с DAQ [8].

Каждый из файлов содержит данные за достаточно длительный период и будет содержать информацию о множестве событий. Каждый файл в наборе может быть обработан независимо. Нет необходимости обмениваться какой-либо информацией во время обработки каждого исходного файла, но результаты могут быть использованы в качестве входных данных для следующего этапа обработки.

1.2 Организация промежуточных и выходных данных

Промежуточные данные – это данные, образующиеся в процессе обработки, но еще не находящиеся в состоянии готовности для использования в системах оффлайн обработки (GRID). В их организацию входит:

- Привязка к фрейму (соответственно рану и периоду);
- Привязка к набору для последующего шага в процессе обработки.

Промежуточные данные живут только в рамках системы, не подразумевается их долговременное хранение. Должны регулярно удаляться.

И, наконец, **выходные данные** – служат уже для последующей обработки в GRID или использования вне функционала фильтра. К их организации относится:

- Привязка к периоду набора и рану;
- Логическая группировка в наборы, с учетом того, что один и тот же файл может быть в разных группах (минимум в одной).

1.3 Потоки данных

К потоку данных относится то, как они циркулируют в системе. Рассмотрим основной поток данных (см. рис. 11):

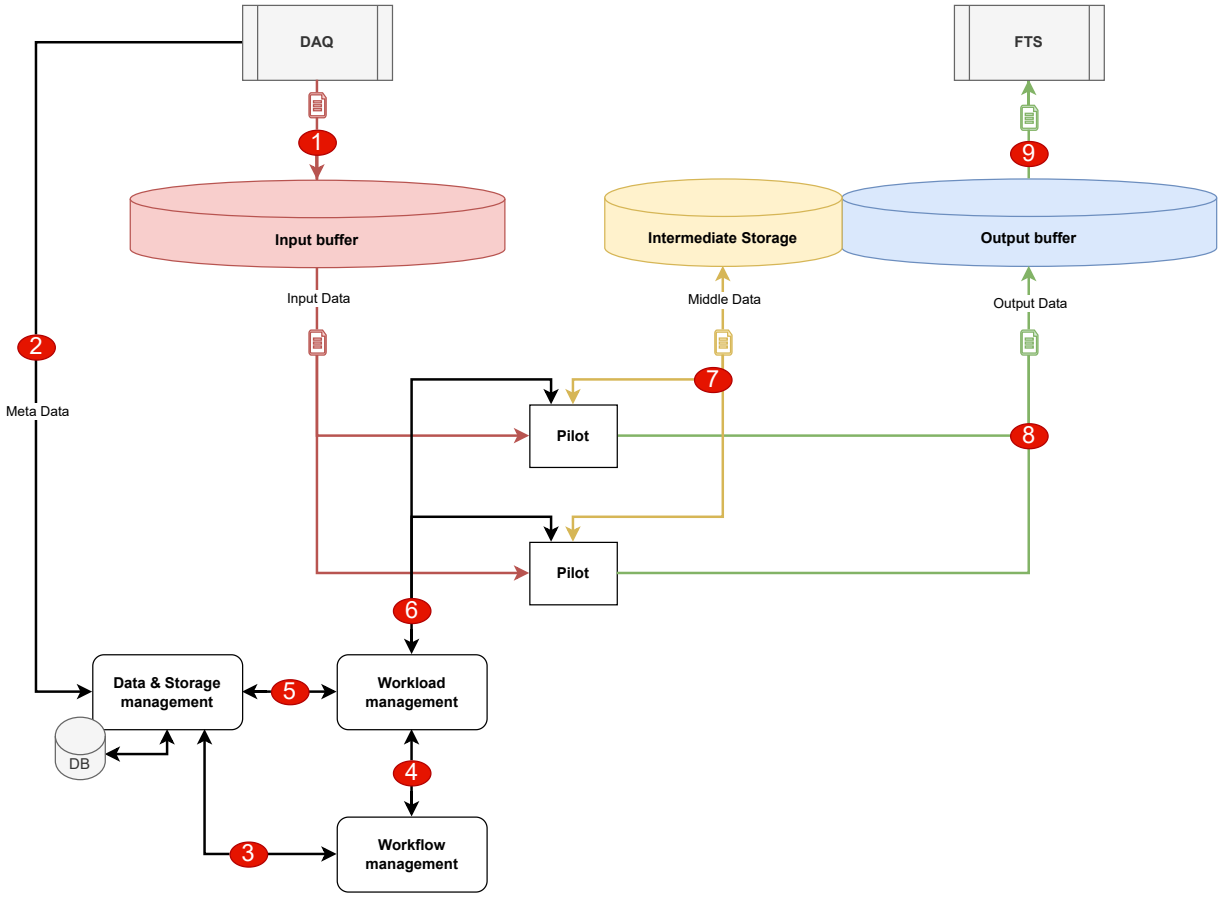

Рис. 11: Потоки данных в SPD On-line filter.

1-2) DAQ загружает набор однородных файлов на входной буфер, и информация о записанном наборе поступает в систему управления данными для первичной регистрации.

3-4) Система управления процессом обработки запрашивает данные о каком либо наборе данных без детальной информации о каждом файле. После чего эта информация поступает в систему управления нагрузкой в составе описания задания (task).

5) Система управления нагрузкой запрашивает информацию о составе набора (о конкретных файлах входящих в набор). Формирует задачи (jobs).

6-7-8) Пилоты вместе с описанием задач получают ссылки на входные файлы и имена выходных файлов. Пилоты записывают выходные файлы в хранилище и информируют систему управления нагрузкой.

5) Система управления нагрузкой регистрирует выходные файлы в выходном наборе и по окончанию обработки входного набора закрывает выходной набор.

3) Система управления процессом контролирует, какие промежуточные наборы нужно удалить, а какие передать для долговременного хранения.

9) FTS (File Transfer System) по итогу всего рабочего процесса забирает выходные данные с выходного буфера.

Глава 2. Общая архитектура

2.1 Функциональные требования

Основное предназначение системы управления данными заключается в самом названии. Т.е. система должна обеспечивать контроль над хранением, организацией, а также целостностью данных.

Для того чтобы система могла узнать о существовании файла на хранилище, должен быть интерфейс для **регистрации файлов**. Иными словами, приём информации о размещении файла (имя, физический путь к нему и т.п.). После чего эта информация может быть размещения во внутреннем каталоге данных.

Из функциональности выше следует то, что система должна осуществлять **каталогизацию файлов**. Это означает наличие интерфейса к каталогу данных, через который можно размещать информацию о файлах, запрашивать информацию из каталога (с различными фильтрами), удалять информацию в каталоге.

Как было сказано во введении, каждый шаг обработки данных оперирует не самим файлом, а набором из них. Таким образом, система управления данными должна предоставлять возможность **управления наборами** (dataset-ми). Сюда входят следующие функции: создать dataset, добавить файл в dataset, закрыть dataset, удалить dataset, дать информацию о содержимом dataset (файлах в датасете).

В конечном итоге, для корректного функционирования всей системы требуется иметь **фоновые сервисы**, которые бы осуществляли удаление файлов на хранилищах, проверку целостности файлов и общий контроль использования хранилища (например, мониторинг «темных» данных – наличие на хранилище не зарегистрированных файлов).

2.2 Концептуальная архитектура

Для разработки системы управления данными требуется, в первую очередь, определиться с её внутренней архитектурой. Была предложена концептуальная архитектура (см. рис. 12) с учётом той функциональности, которой должна обладать система.

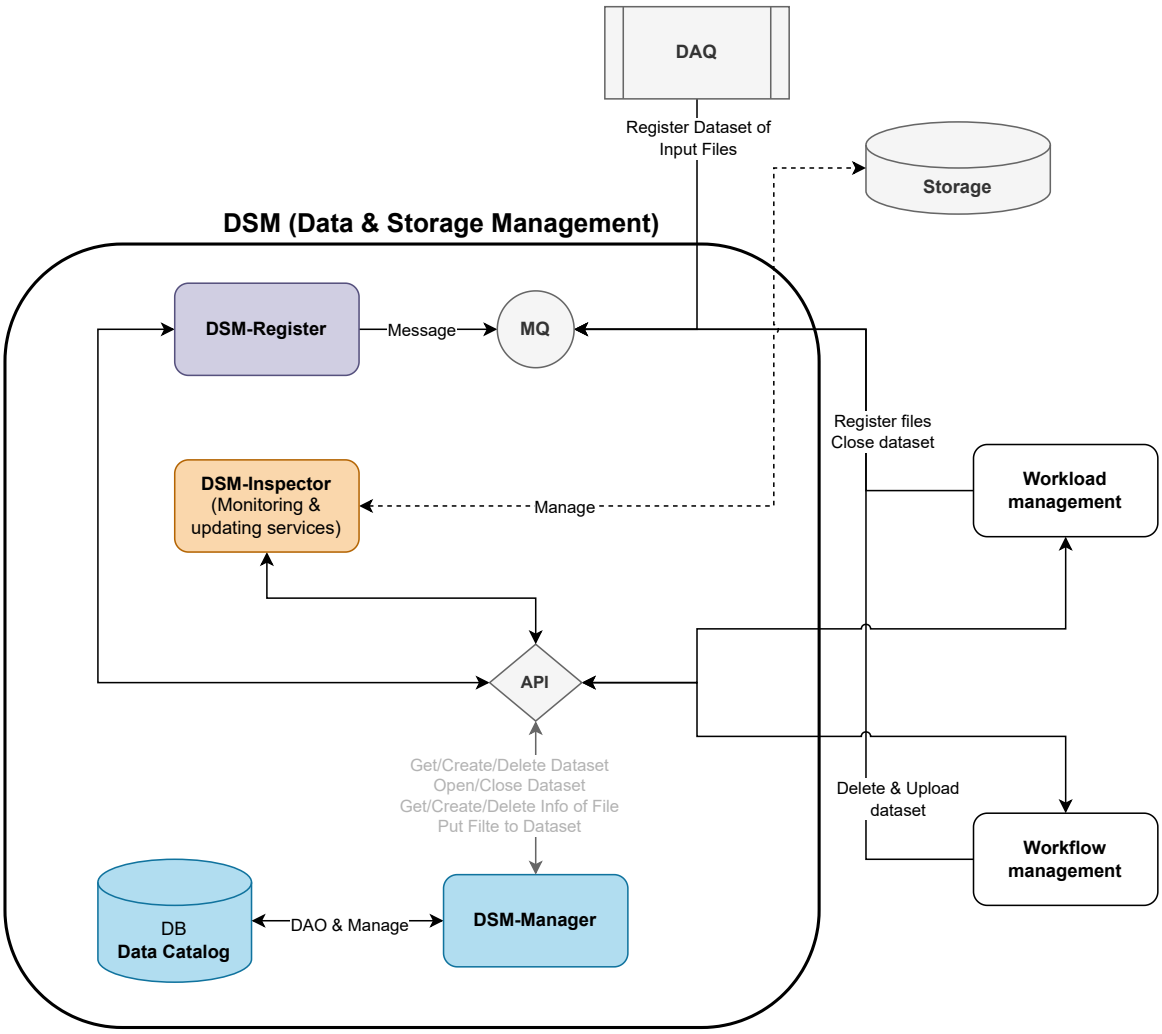

Рис. 12: Концептуальная архитектура системы управления данными.

В первую очередь система декомпозировалась на набор сервисов, следуя паттерну «микросервисной архитектуры» [16]. Причём в качестве стратегии разбиения был выбран шаблон «разбиение по поддоменам». Он основан на концепциях предметно-ориентированного проектирования (Domain-Driven Design, DDD) [17].

DDD разбивает всю модель предметной области (домен) на поддоме-

ны. У каждого поддомена своя модель данных, область действия которой принято называть ограниченным контекстом (Bounded Context). Каждый микросервис разрабатывается внутри этого ограниченного контекста. Основная задача при использовании DDD-подхода – подобрать поддомены и границы между ними так, чтобы они были максимально независимы друг от друга.

Помимо декомпозиции также были определены принципы межсервисного взаимодействия. В тех случаях, где требуется мгновенный ответ от сервиса, используется HTTP протокол обмена данными. В противном же случае, данный протокол может стать «бутылочным горлышком» для общей работоспособности системы, поэтому был выбран AMQP протокол для асинхронной передачи данных в виде сообщений.

Таким образом, были выделены следующие сервисы:

- **dsm-register**. Сервис, принимающий в асинхронном режиме (через очередь сообщений) заявки на добавление/удаление данных в системе. При обработке заявок сервис вносит изменения в каталог данных через API сервиса dsm-manager;
- **dsm-manager**. Сервис, предоставляющий REST API к каталогу данных (размещение данных в каталоге, обращение к каталогу, изменение данных в каталоге);
- **dsm-inspector**. Набор фоновых сервисов для мониторинга и контроля состояния данных в хранилище (проверка целостности файлов, контроль использования хранилища, удаление файлов на хранилищах).

Глава 3. Разработка требований

3.1 Требование к БД

3.1.1 Концептуальная модель данных

По ряду критериев: качество поддержки, функциональность, широкий круг пользователей, доступность в качестве СУБД был выбран PostgreSQL 12 [18]. Таким образом, дальнейшее проектирование БД будет строиться с учётом её особенностей и возможной функциональности.

В первую очередь, была проработана концептуальная модель данных / ER-диаграмма (см. рис. 13) с учётом того описания данных, которое было проведено в разделе анализа данных. Сюда входит множеством понятий и связей между ними, определяющих смысловую структуру рассматриваемой предметной области.

При построении схемы таблиц важно было соблюсти правила третьей нормальной формы [19]. Это необходимый уровень нормализации баз данных, при котором организация данных ориентирована на устранение избыточности и несогласованных зависимостей.

С учётом этого в данную модель вошёл следующий набор таблиц:

- *DAT_FILE* – каталог файлов, обрабатываемых системой;
- *DAT_DATASET* – каталог наборов файлов, созданных системой при построении вычислений;
- *DAT_FILE_HISTORY* и *DAT_DATASET_HISTORY* – архивные таблицы по файлам и наборам;
- *DIC_FILE_STATUS* и *DIC_DATASET_STATUS* – справочники, хранящие возможные статусы по файлам и наборам соответственно;
- *DAT_STORAGE* – информация о хранилищах.

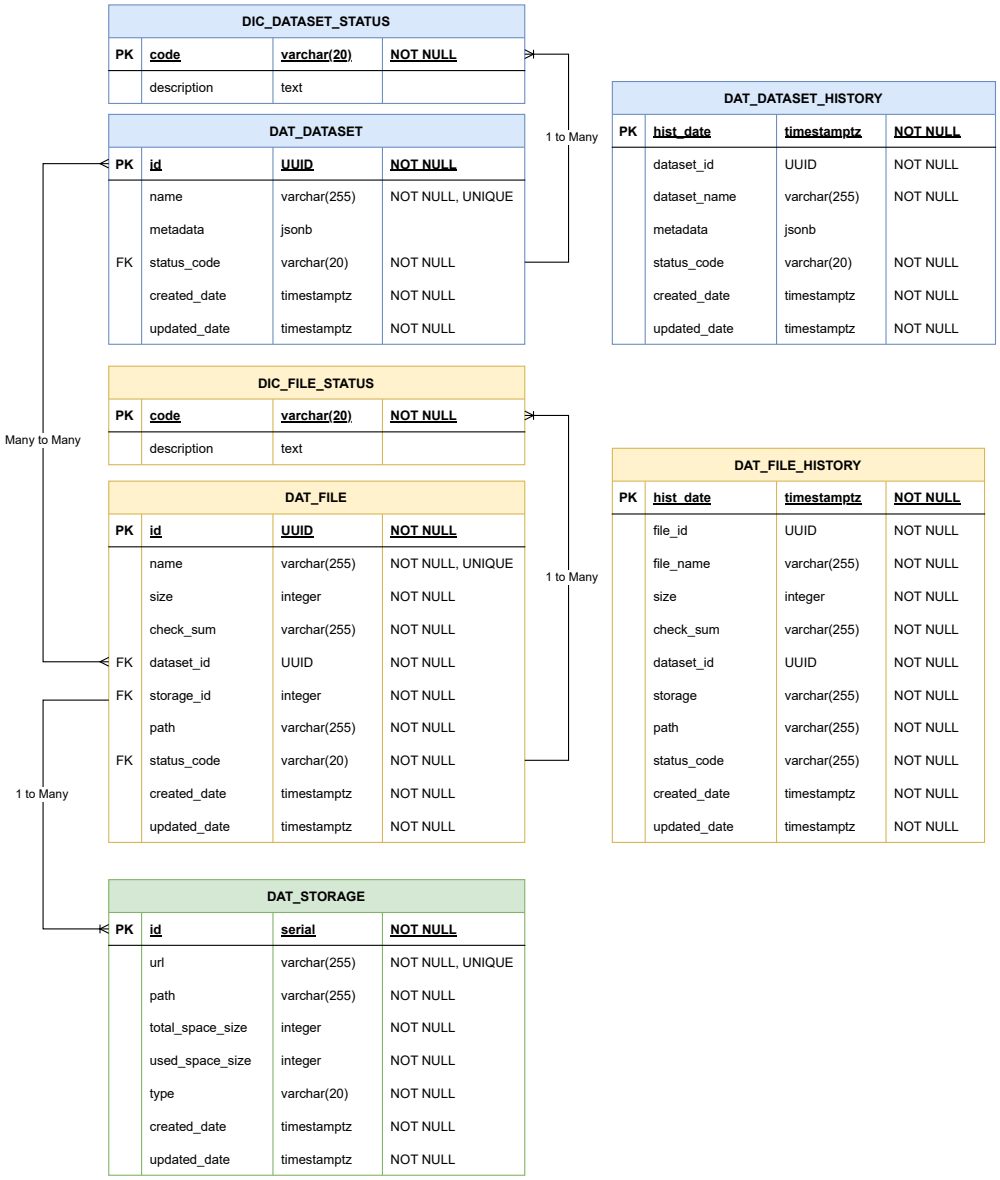

Рис. 13: Концептуальная модель данных.

Каталог файлов является ключёвой таблицей в базе данных. В её состав атрибутов вошла та информация, которой достаточно для покрытия следующих целей:

1. Идентификация файла на хранилище (имя файла, путь на хранилище, идентификатор хранилища);
2. Контроль целостности файла (размер файла и его контрольная сумма);
3. Фиксация жизненного цикла файла (статус файла).

Помимо этого предусмотрена возможность принадлежности файла к одному или нескольким наборам. Это осуществляется с помощью **ассоциативной таблицы**, реализующей отношение «многие ко многим» [20].

Каталог наборов служит для хранения логических группировок файлов, не релевантных к физическому расположению. Каждый набор имеет уникальное наименование, определяемое логикой группировки. Также в данном каталоге может быть размещена дополнительная неструктурированная мета-информация о наборе в JSON поле (например, в случае входного набора файлов, поступающих с DAQ, это может быть информация о временном срезе данных: номер фрейма, номер ран-а и т.п.). Набор, как и файл, имеет свой жизненный цикл, поэтому для его фиксации также имеется поле со статусом.

Хранилища помимо адреса могут характеризоваться также дополнительной важной информацией. Сюда входит, во-первых, его тип (например, входное, промежуточное, выходное). Во-вторых, полный путь до него: внешний и внутренний адрес. И, в-третьих, его состояние: размер всего дискового пространства и уже занятого.

Т.к. изменения в каталоге файлов и наборов может происходить достаточно часто, потребовалось ввести дополнительные **history таблицы** для фиксации предыдущих состояний данных. Это полезно для последующего анализа корректности работы системы. Заполнение такого рода таблиц предполагается осуществлять автоматически при помощи одного из механизмов баз данных – триггер. Подробнее см. раздел «3.1.2 Дополнительные механизмы».

Для строгого контроля возможных статусов файлов и наборов было принято решение хранить их в отдельных **справочных таблицах** (см. таблицу 1 и 2).

Таблица 1: Справочник возможных статусов файла (DIC_FILE_STATUS)

Код статуса	**Описание**
CREATED	Файл добавлен в систему
DAMAGED	Файл повреждён
TO_DELETE	Файл с пометкой «На удалении»
UPLOADING	Файл выгружается
DELETED	Файл удалён из системы

Таблица 2: Справочник возможных статусов набора (DIC_DATASET_STATUS)

Код статуса	**Описание**
OPEN	Набор открыт
CLOSED	Набор закрыт
FROZEN	Набор временно «заморожен»
TO_UPLOAD	Набор с пометкой «На выгрузку»
UPLOADING	Набор выгружается
TO_DELETE	Набор с пометкой «На удалении»
DELETED	Набор удалён из системы

3.1.2 Дополнительные механизмы

Для работоспособности БД требуется также предусмотреть дополнительные механизмы функционирования. Один из них уже упоминался в предыдущим разделе – **триггер** [21]. По своей сути это является некоторым сценарием обработки возникающего события в базе данных. В нашем случае он должен срабатывать при любой операции на запись (insert, update, delete) в таблицы DAT_FILE и DAT_DATASET. Логика сценария такова – перед внесением изменения сохранить предыдущее состояние в одну из history таблиц.

Очевидно, что с течением времени history таблицы будут становиться всё больше, поэтому операции на чтение и запись станут выполняться медленно. Таким образом, следует предусмотреть **партицирование таблиц** [22] с ключём партицирования hist_date. Размер партиции стоит задать равным месяцу. Это означает, что таблицы разобьются на под-таблицы по заданному ключу партицирования с учётом размера партиции. Вследствие чего, при указании значения ключа партицирования операции чтения и записи будут происходить в рамках одной партиции, что гораздо эффективнее.

Говоря про оптимизацию запросов, также немаловажно предусмотреть минимальный набор **индексов** [23]. Сюда вошли B-Tree индексы [24] для оптимизации поисковых запросов по следующим таблицам:

- DAT_DATASET_HISTORY по полям dataset_id, dataset_name, status_code;
- DAT_FILE_HISTORY по полям file_id, file_name, status_code.

Исходя из концептуальной модели данных, требуется ввести **SQL ограничения** (правила для данных в таблице) на некоторые поля:

- *dat_dataset_name_unique* – ограничение уникальности на имя набора;
- *dat_file_name_unique* – ограничение уникальности на имя файла;
- *dat_storage_url_unique* – ограничение уникальности на URL хранилища.

3.2 Требования к сервисам

3.2.1 Сервис dsm-register

Сервис должен слушать очередь сообщений по протоколу AMQP и обрабатывать заявки на добавление/удаление данных в системе. В таблице 3 представлены предполагаемые шлюзы приёма сообщений. В качестве AMQP-брокера, который осуществляет маршрутизацию и подписку на нужные типы сообщений, выбран RabbitMQ [25].

Таблица 3: Перечень шлюзов приёма сообщений сервиса dsm-register

Exchange	**Routing Key**	**Назначение**
dsm.register (direct)	file.input	Приём информации о поступивших файлах на входной буфер.
	file.process	Приём информации о новых файлах, полученных в процессе обработки.
	dataset.close	Приём заявки на закрытие набора файлов.
	dataset.upload	Приём заявки на выгрузку файлов в наборе во внешнее хранилище.
	dataset.delete	Приём заявки на удаление файлов в наборе на внутреннем хранилище.

После, следует определить контракт сообщений и алгоритм его обработки.

Шлюз **dsm.register.file.input** принимает сообщения вида: *путь до хранилища, путь к файлу на хранилище, имя файла, его размер и контрольная сумма.* Далее, выполняются следующие шаги:

1. Заводится набор по номеру frame-а со статусом OPEN;
2. Регистрируются файлы с привязкой к этому набору. Устанавливается первичный статус CREATE.

Шлюз **dsm.register.file.process** принимает сообщения вида: *идентификатор хранилища, путь к файлу на хранилище, имя файла, его размер, контрольная сумма и идентификатор набора.* Далее, происходит регистрация новых файлов (в статусе CREATE) с привязкой к указанному набору.

Шлюз **dsm.register.dataset.close** принимает сообщения вида: *идентификатор набора и контрольный список файлов (имена файлов)*. Далее, выполняются следующие шаги:

1. Запрашивается список зарегистрированных файлов в указанном наборе;
2. Если данный список совпадает с контрольным списком, то у набора устанавливается статус CLOSE. Иначе сообщение возвращается обратно в очередь для отложенного исполнения.

Шлюзы **dsm.register.dataset.upload** и **dsm.register.dataset.delete** принимают только *идентификатор набора*. Далее, выполняются следующие шаги:

1. Запрашивается текущий статус набора;
2. Если набор находится в статусе OPEN, то сообщение возвращается обратно в очередь для отложенного исполнения;
3. Иначе устанавливается статус TO_UPLOAD и TO_DELETE соответственно.

3.2.2 Сервис dsm-manager

В таблице 4 представлен перечень REST API к каталогу данных в системе, который должен предоставлять сервис dsm-manager. Часть из них нужны для внутреннего функционирования системы управления данными, остальная часть для внешнего взаимодействия. Подробнее о информации, входящей в состав ответа, см. раздел «3.1.1 Концептуальная модель данных».

Таблица 4: Описание REST API сервиса dsm-manager

Функциональность	**URN**	**Контракт запроса**	**Контракт ответа**
	Внутреннее API		
Управление информацией о наборах в каталоге			
Создать набор	POST /dataset	Информация о наборе	ID набора
Получить набор	GET /dataset/<id>	ID набора	Информация о наборе
Изменить набор	PUT /dataset/<id>	ID набора и изменения	Обновлённая информация о наборе
Удалить набор	DELETE /dataset/<id>	ID набора	–
Управление информацией о файлах в каталоге			
Добавить файл	POST /file	Информация о файле и принадлежность к набору	ID файла
Получить файл	GET /file/<id>	ID файла	Информация о файле
Изменить файл	PUT /file/<id>	ID файла и изменения	Обновлённая информация о файле
Удалить файл	DELETE /file/<id>	ID файла	–
Управление информацией о хранилище			
Добавить хранилище	POST /storage	Информация о хранилище	ID хранилища
Получить хранилище	GET /storage/<id>	ID хранилища	Информация о файле
Изменить хранилище	PUT /storage/<id>	ID хранилища и изменения	Обновлённая информация о хранилище
Удалить хранилище	DELETE /storage/<id>	ID хранилища	–
Получение информации для мониторинга			
Получение списка наборов, к которым принадлежит файл	GET /dataset?fileId=<id>	ID файла	Информация о наборах
Список файлов в определённом статусе	GET /file?status=<>	Статус файла	Информация о файлах
Поиск информации о файле по имени	GET /file?filename=<>	Имя файла	Информация о файлах
	Внешнее API		
Взаимодействие с системой управления нагрузкой			
Содержание набора	GET /dataset/<id>/file	ID набора	Общее кол-во файлов в наборе и список с краткой информацией по файлам
Взаимодействие с системой управления процессом			
Список наборов в определенном статусе (с отметкой времени)	GET /dataset?status=<>&timestamp=<>	Статус набора и Отметка времени	Информация о наборах
Некоторые API управления наборами (создание выходного набора, получение статуса выходного набора)			

3.2.3 Сервис dsm-inspector

Сервис, состоящий из набора фоновых задач для мониторинга состояния системы, выполняющихся по CRON расписанию [26] (см. листинг 1):

- Удаление файлов на хранилищах;
- Контроль выгрузки данных;
- Проверка целостности файлов;
- Контроль использования хранилища.

```
* * * * * *
- - - - - -
| | | | | |
| | | | | ----- день недели (0-7). Воскресенье это 0 или 7.
| | | | ------- месяц (1-12)
| | | --------- день месяца (1-31)
| | ----------- час (0-23)
| -------------- минута (0-59)
---------------- секунда (0-59)
```

Листинг 1: Схема CRON расписания.

Удаление файлов на хранилище является сервисом исполнения заявок на удаление наборов и выполняется при помощи двух параллельных процессов:

- Постановка файлов на удаление (CRON: «0 * * * *»):
  1. Получаем список наборов со статусом TO_DELETE;
  2. По каждому набору получаем его содержимое (информация о файлах);
  3. По каждому файлу, если файл находится не в статусе DELETED, то устанавливаем для него статус TO_DELETE;
  4. Если в датасете оказались все файлы в статусе DELETED, то устанавливаем аналогичный статус для набора.

- Попытка удаления файла (CRON: «0 * * * *»):
  1. Получаем файл со статусом TO_DELETE;

2. Запрашиваем все его наборы;
3. Проверяем: все его наборы в статусе TO_DELETE и файл существует на хранилище;
4. В случае выполнения всех условий, удаляем файл на хранилище и устанавливает статус DELETED.

Контроль выгрузки данных является сервисом исполнения заявок на выгрузку файлов в наборе во внешнюю систему и так же выполняется при помощи двух параллельных процессов:

- Постановка задач на выгрузку (CRON: «0 * * * *»):

  1. Получаем список наборов со статусом TO_UPLOAD;
  2. По каждому набору получаем его содержимое (информация о файлах);
  3. Создаём задачу во внешней системе на выгрузку набора (по ID/имени набора);
  4. Устанавливаем для каждого файла в наборе статус UPLOADING;
  5. Устанавливаем статус набору UPLOADING.

- Мониторинг выгрузки (CRON: «*/5 * * * *»):

  1. Получаем список наборов со статусом UPLOADING;
  2. По ID/имени набора получаем статус выполнения задачи на выгрузку;
  3. Если статус «УСПЕШНО», то устанавливаем статус набору TO_DELETE.

Проверка целостности файлов является сервисом мониторинга (CRON: «0 */5 * * *»). Алгоритм работы следующий:

1. Получаем список наборов в статусе CLOSED;
2. По каждому набору получаем его содержимое (информация о файлах);

3. Формируем список уникальных файлов (по ID);
4. По каждому файлу:
   (a) Проверяем наличие файла в хранилище;
   (b) Проверяем размер файла;
   (c) Проверяем контрольную сумму файла;
   (d) Если какая-то проверка не пройдена, то устанавливаем статус файлу DAMAGED.
5. Если какой-либо файл в наборе оказался повреждённым, то устанавливаем статус набору FROZEN.

Контроль использования хранилища является сервисом мониторинга (CRON: «0 */5 * * *»). Имеет две сценария проверки:

- Мониторинг входного и выходного хранилища на предмет «тёмных» файлов:
  1. Получаем список файлов в хранилище;
  2. Осуществляем поиск файла в каталоге по его имени;
  3. Если файл не найден, то добавляем его в набор со статусом TO_DELETE;
  4. Если набор не пустой, то создаём его со статусом TO_DELETE.
- Мониторинг заполненности хранилища:
  1. Получаем информацию по хранилищам (url, path);
  2. Проверяем размер занятого пространства на хранилищах и обновляем эту информацию в каталоге.

Глава 4. Прототипирование

Разработка сервисов ведётся в защищенном контуре МЛИТ ОИЯИ. Репозитории с исходном кодом расположены на внутреннем GitLab организации. Их организация представлена на рис. 14.

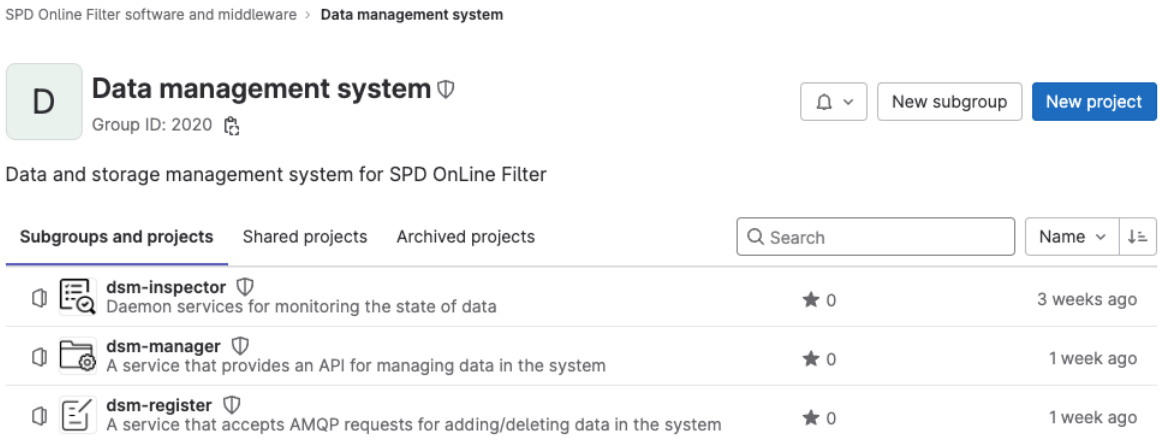

Рис. 14: Организация репозиториев с исходном кодом.

4.1 Выбор стека технологий

Все сервисы разрабатываются на базе современных, надёжных и высокопроизводительных инструментов. В таблице 5 представлен подробнее выбранный стек технологий.

Таблица 5: Выбранный стек технологий

Инструмент	**Причина использования**
Python 3.11	Для проекта SPD определён следующий стек языков программирования – C++ и Python. Первый служит для проведения вычислений, второй для разработки сервисов. Исходя из чего система управления данными разрабатывается на Python. Выбрана последняя стабильная версия 3.11, которая в себе содержит множество исправлений, оптимизаций, а также новые полезные инструменты.

FastAPI + Unicorn	Для построения API выбран асинхронный фреймворк FastAPI [27]. Данный фреймворк является некоторым балансом между пропускной способностью сервиса (см. открытые бенчмарки [28]), а также удобством разработки и сопровождения. Т.к. FastAPI следует ASGI формату, был выбран соответствующий HTTP-сервер – Unicorn, который является де факто эталонным (используется во многих коммерческих продуктах).
Pydantic Schemas + Swagger UI	Описание контрактов API принято делать из кода. Для этой цели выбран инструмент Pydantic (поставляется целостно с FastAPI), использующий аннотацию типов Python. Для отображения контрактов API используется Swagger UI, который придерживается спецификации OpenAPI 3.0 [29].
SQLAlchemy	Для следования подхода «Code-First» [30] требуется определять схему Базы Данных из кода. Это означает реализацию ORM (Object-Relational Mapping) моделей таблиц. С этой целью выбран инструмент SQLAlchemy.
Alembic (Migration)	Для генерации скриптов миграции и развётки их в БД используется инструмент Alembic [32]. Имеет нативную интеграцию с SQLAlchemy.
Aiopg	Инструмент для асинхронной работы с БД. Также увеличивает пропускную способность.
Pika	Для взаимодействия с брокером сообщений RabbitMQ требуется библиотека, реализующая протокол AMQP. Классическим вариантом является Pika [36].

Celery Cron Jobs	На проекте потребуется инструмент для запуска фоновых задач по расписанию. Одним из таких инструментов является Celery [31], основанный на асинхронной очереди задач. Данное решение поддерживает Cron расписание, а также имеет возможность в масштабировании.
Aiofiles	При выполнении мониторинга хранилища потребуется обращение к файловой системе. Для устранение блокирующего эффекта при чтении файлов было выбрано решение Aiofiles.
Dependency Injector	Большая проблема многих Python проектов – это ациклические зависимости. Данная проблема решается, если следовать последнему принципу SOLID [34] – DIP. Для имплементации данного принципа в проекте был выбран инструмент Dependency Injector. Это позволяет не только правильно организовывать зависимости в коде, а также даёт возможность удобного конфигурирования, расширения и сопровождения проекта.
Poetry	В качестве пакетного менеджера был выбран Poetry [38]. Данный инструмент в отличии от классического в Python пакетного менеджера pip умеет лучше разрешать конфликты зависимостей. Также имеется возможность автоматической фиксации списка зависимостей для воспроизводимости сборки.
Docker	Для автоматизации сборки и поставки решения на стенд принято любой сервис упаковывать в самодостаточный Docker образ [35].

4.2 Прототип сервиса dsm-manager

4.2.1 Структура проекта

Структура проекта dsm-manager-а разработана с учётом удобства дальнейшего его сопровождения. Текущее её описание представлено в таблице 6.

Таблица 6: Описание структуры проекта dsm-manager

```
|--- README.md                          <-- Файл верхнего уровня для разработчиков.
|
|--- app
|   |--- migrations                     <-- Модуль миграций для SQL Базы данных.
|   |   |--- versions                   <-- Скрипты миграции.
|   |   |   |--- ...
|   |   |--- ...
|   |--- api                            <-- Главный модуль приложения.
|   |   |--- main.py                    <-- Скрипт построения приложения.
|   |   |--- config                     <-- Модуль настройки компонент приложения.
|   |   |--- dao                        <-- Модуль работы с данными.
|   |   |   |--- models                 <-- Модуль с ORM моделями Базы данных.
|   |   |   |--- repositories           <-- Модуль с репозиториями к таблицам БД.
|   |   |   |--- database.py            <-- Объект доступа к Базе данных.
|   |   |--- domain                     <-- Модуль с доменными моделями данных.
|   |   |--- dto                        <-- Модуль с моделями данных для клиента.
|   |   |--- mappers                    <-- Модуль с сопоставлениями моделей данных.
|   |   |--- services                   <-- Модуль с сервисами бизнес логики.
|   |   |--- endpoints                  <-- Модуль с точками входа в приложение.
|   |   |   |--- v1                     <-- Функциональные точки входа в приложение.
|   |   |   |--- info_endpoints.py      <-- Служебные точки входа в приложение.
|   |   |   |--- exception_handlers.py  <-- Обработчики исключений.
|   |   |--- exceptions                 <-- Модуль с исключениями приложения.
|   |   |--- constants                  <-- Модуль с константами приложения.
|   |--- cli
|   |   |--- docs.py                    <-- CLI команды для генерации файла API.
|   |--- settings                       <-- Модуль с настройками приложения.
|
|--- Dockerfile                         <-- Файл с настройками Docker образа.
|
|--- docker-compose.yml                 <-- Настройка инфраструктуры.
|
|--- .gitlab-ci.yml                     <-- GitLab CI конфиг. для публикации API.
|
|--- pyproject.toml                     <-- Poetry файл с зависимостями приложения.
```

Исходя из функциональных требований, в сервисе были реализованы две ключевых составляющих: 1) Слой работы с базой данных каталога, 2) Слой работы с клиентским API, предоставляющий функции управления данными.

4.2.2 Слой работы с БД

Слой работы с базой данных является ключевой составляющей данного сервиса. В первую очередь, требовалось решить, как будет задаваться схема таблиц. Опираясь на общепринятый подход «Code-First» [30], было принято решение задавать её из кода на основе ORM (Object-Relational Mapping) моделей библиотеки SQLAlchemy. Пример такой модели см. в приложении листинг 8.

Для того, чтобы составить скрипты построение таблиц на основе DDL (Data Definition Language) команд, используется инструмент построения миграций Alembic [32]. Скрипты миграции – это сценарий построения текущей версии таблиц приложения относительной предыдущей версии. Их создание происходит аналогично, как происходит создание GIT коммита – достаточно вызывать CLI (Command Line Interface) команду *«migrate»* и указать комментарий. После чего скрипт миграции создаётся в директории *migrations/versions*. Пример такого скрипта миграции см. в приложении листинг 9.

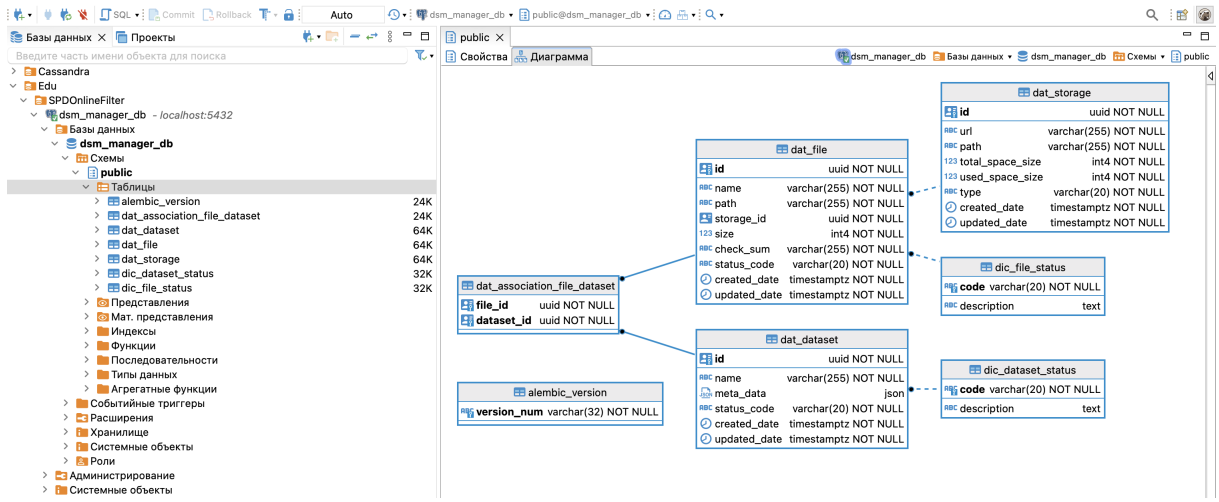

Рис. 15: Развёрнутая схема таблиц в БД сервиса dsm-manager (интерфейс Dbeaver).

Далее, требуется их применить. Для этого используется CLI команда Alembic – *«update»*. После чего схема таблиц разворачивается в Базе данных (см. рис. 15).

Как известно, в сервисе также предусмотрены несколько справочников. Их схема составляется аналогично, но наполнение идёт автоматически при старте приложения на основе объекта перечисления возможных значений (см. пример в приложении листинг 10).

Доступ к данными осуществляется при помощи паттерна «Репозиторий» [33]. Наиболее важным отличием репозиториев является то, что они представляют собой коллекцию объектов. Они не описывают хранение в базах данных, или кэширование, или решение любой другой технической проблемы. Коллекция – это некоторый контракт работы с данными, абстрагирующий от деталей механизма их хранения. Пример репозитория см. в приложении листинг 11.

4.2.3 Слой работы API

Для реализации API, в первую очередь, важно выбрать веб-фреймворк, который является каркасом для построения веб-приложений. На сервисе dsm-manager было решено взять асинхронный фреймворк FastAPI [27] в силу его высокой пропускной способности (см. открытые бенчмарки [28]), а также удобства разработки и сопровождения.

Любой асинхронный веб-фреймворк в Python следует протоколу ASGI взаимодействия между HTTP-сервером и самим приложением. Таким образом, для работы приложения требуется сконфигурировать ASGI приложение, чтобы далее оно могло вызываться из HTTP-сервера (в нашем случае выбран сервер Unicorn). Функция построения приложения лежит в файле main.py.

Под конфигурированием имеется ввиду построение контейнера зависимостей и настройку с помощью него всех компонентов приложения: слой работы с базой данных, сервисный слой (бизнес логика), а также точки входа в приложение. Данная процедура выполняется на основе одного из принципов SOLID [34] – DIP (dependency inversion principle) и реализуется

при помощи инструмента Dependency Injector. Тем самым обеспечивается гибкость в управлении зависимостями, а также удобство расширения функциональности приложения.

В силу того, что на проекте присутствует несколько компонентов, было предусмотрено строгое разделение моделей данных между ними:

- *DAO-модели* – модели уровня источника данных;
- *Domain-модели* – модели самого приложения;
- *DTO-модели* – модели отправки данных клиенту.

Преобразованием из одних моделей в другие происходит при помощи отдельного механизма сопоставления – automapper.

Для удобства использования API, реализована автогенерация его описания по спецификации OpenAPI 3.0 [29], после чего оно может быть доступно в интерфейсе Swagger UI по адресу *<адрес сервера>/docs* (см. рис. 16).

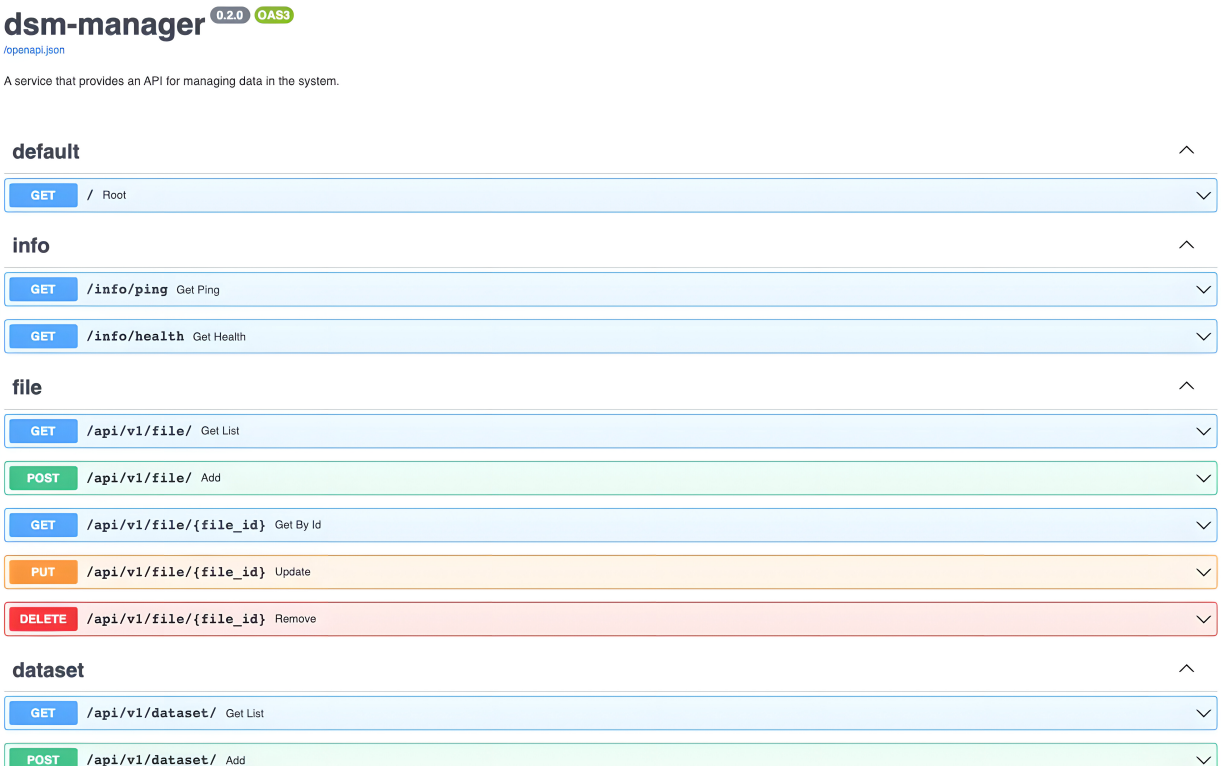

Рис. 16: Swagger UI с описанием API сервиса dsm-manager.

4.3 Прототип сервиса dsm-register

Структура проекта dsm-register-а разработана аналогичным образом. Текущее её описание представлено в таблице 6.

Таблица 7: Описание структуры проекта dsm-manager

```
|--- README.md                          <-- Файл верхнего уровня для разработчиков.
|
|--- app
|   |--- executor                       <-- Главный модуль приложения.
|   |   |--- main.py                    <-- Скрипт построения приложения.
|   |   |--- config                     <-- Модуль настройки компонент приложения.
|   |   |--- dto                        <-- Модуль с моделями данных для клиента.
|   |   |   |--- rabbit                 <-- Модуль с моделями данных RabbitMQ.
|   |   |--- rabbit                     <-- Модуль работы с RabbitMQ.
|   |   |   |--- consumers              <-- Модуль приёма сообщений RabbitMQ.
|   |   |   |--- rabbit_server.py       <-- Объект доступа к серверу RabbitMQ.
|   |   |   |--- rabbit_start.py        <-- Старт компоненты работы с RabbitMQ.
|   |   |--- utils                      <-- Модуль с ютильными методами.
|   |--- settings                       <-- Модуль с настройками приложения.
|
|--- Dockerfile                         <-- Файл с настройками Docker образа.
|
|--- docker-compose.yml                 <-- Настройка инфраструктуры.
|
|--- pyproject.toml                     <-- Poetry файл с зависимостями приложения.
```

Точкой старта приложения также является файл main.py, в рамках которого идёт конфигурирование приложения через такой же механизм DIP.

Отличительной особенностью сервиса является работа с RabbitMQ. На сервисе потребовалось предусмотреть настройку очередей брокера сообщений, а также их обработчиков.

Определение необходимых очередей на сервисе осуществляется при помощи декларативной нотации инструмента Pika [36]. Пример см. в приложении листинг 12. Далее на старте приложения идёт их развёртка. Результат см. на рисунке 17.

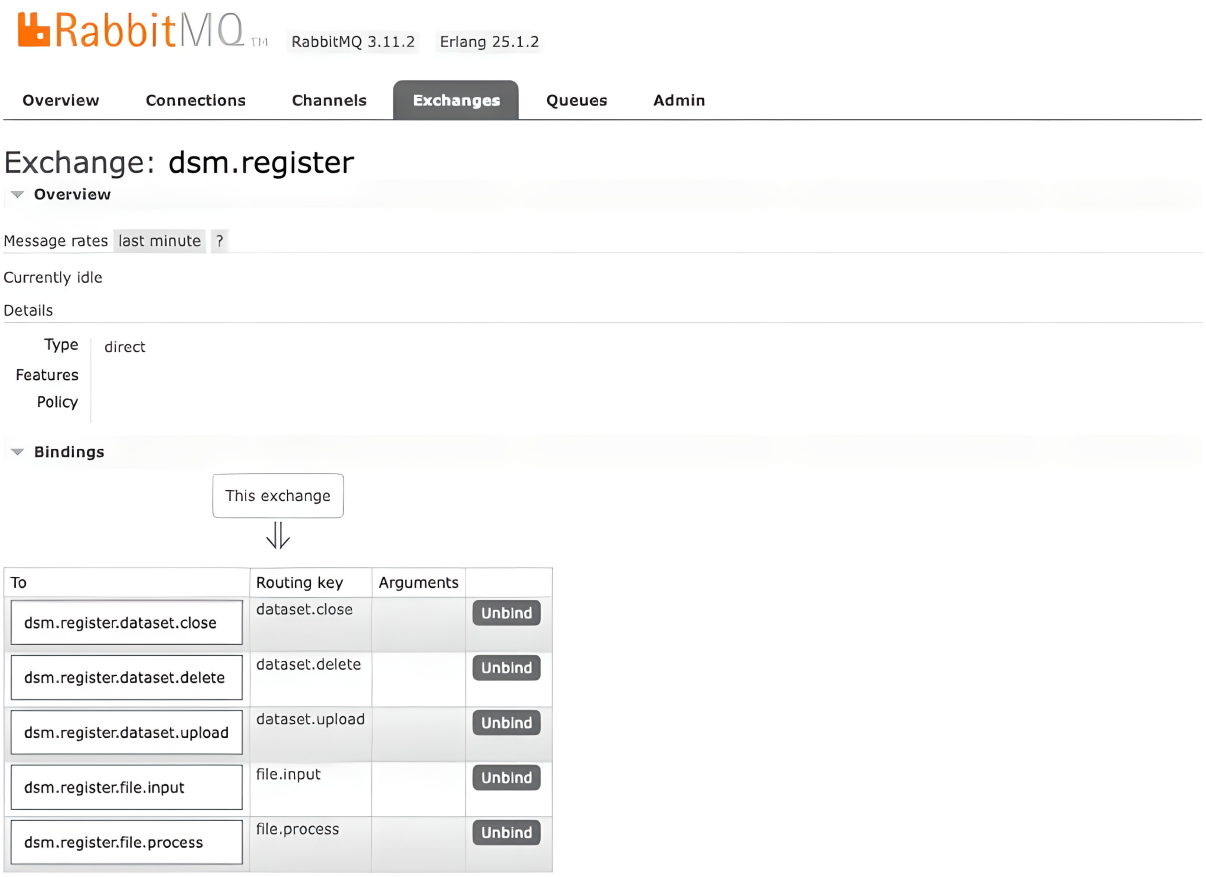

Рис. 17: Сконфигурированные очереди RabbitMQ.

Также на сервисе была предусмотрена возможность отклонения некорректных сообщений. Т.е. в случае, если при обработке сообщения возникает исключение, срабатывает обработчик исключения, в котором вызывается команда basic reject. При этом, чтобы такого рода сообщения не были полностью утеряны, для каждой очереди реализована доп. «очередь мёртвых сообщений» (DLQ/Dead letter queue) [37]. Их построение происходит аналогичным образом (см. рис. 18).

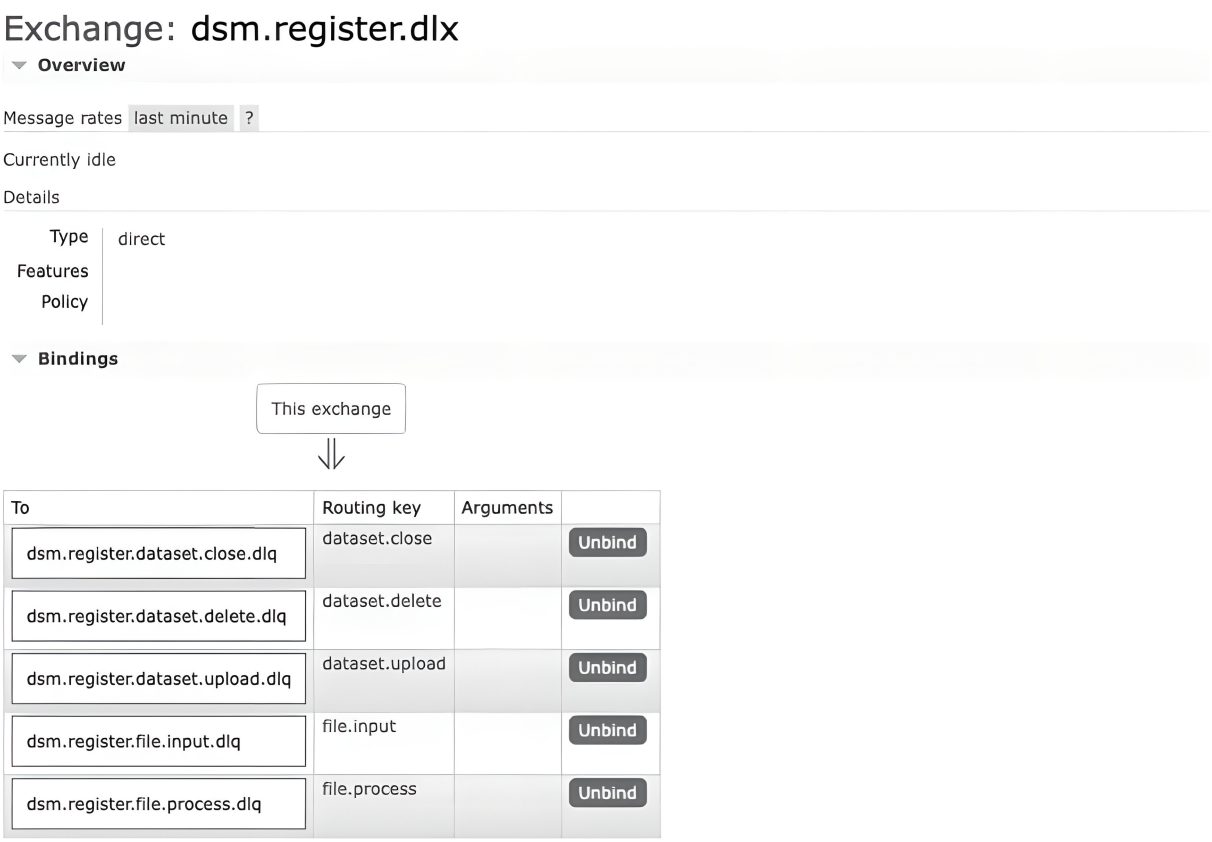

Рис. 18: Сконфигурированные «очереди мёртвых сообщений» RabbitMQ.

Глава 5. Тестирование

5.1 Подготовка инфраструктуры

Гибкость в сборке и запуске сервисов является также немаловажной. В первую очередь было предусмотрена возможность задания необходимых **настроек**. Сами настройки могут задаваться либо через переменные окружения, либо через .env файл. Схема файлов настроек по сервисам-прототипам представлена на листинге 2 и 3.

```
# general FastAPI settings
FASTAPI_DEBUG=false
FASTAPI_VERSION=0.2.0

# Cross-Origin Resource Sharing settings
FASTAPI_CORS_ALLOW_ORIGINS='["*"]'
FASTAPI_CORS_ALLOW_CREDENTIALS=true
FASTAPI_CORS_ALLOW_METHODS='["*"]'
FASTAPI_CORS_ALLOW_HEADERS='["*"]'

# Datasource settings
DATASOURCE_DB=dsm_manager_db
DATASOURCE_HOST=db
DATASOURCE_PORT=5432
DATASOURCE_USER=postgres
DATASOURCE_PASS=postgres
```

Листинг 2: Схема .env файла настроек сервиса dsm-manager.

```
# general App settings
APP_VERSION=0.0.1
LOG_DEBUG=false

# Datasource settings
RABBIT_HOST=rabbit-local
RABBIT_PORT=5672
RABBIT_USERNAME=rabbitmq
RABBIT_PASSWORD=rabbitmq
```

Листинг 3: Схема .env файла настроек сервиса dsm-register.

Далее для автоматизации и воспроизводимости сборки и запуска были разработаны **Docker образы сервисов** [35] при помощи Dockerfile-а. В нём прописаны: настройка среды (например, операционной системы),

настройка зависимостей сервиса, а также настройка компонентов сервиса. Сама сборка сервиса идёт при помощи пакетного менеджера Poetry [38].

Также составлена **схема запуска необходимой инфраструктуры** через Docker compose файл – база данных PostgreSQL, брокер сообщений RabbitMQ и сами сервисы (см. рис. 19 и 20).

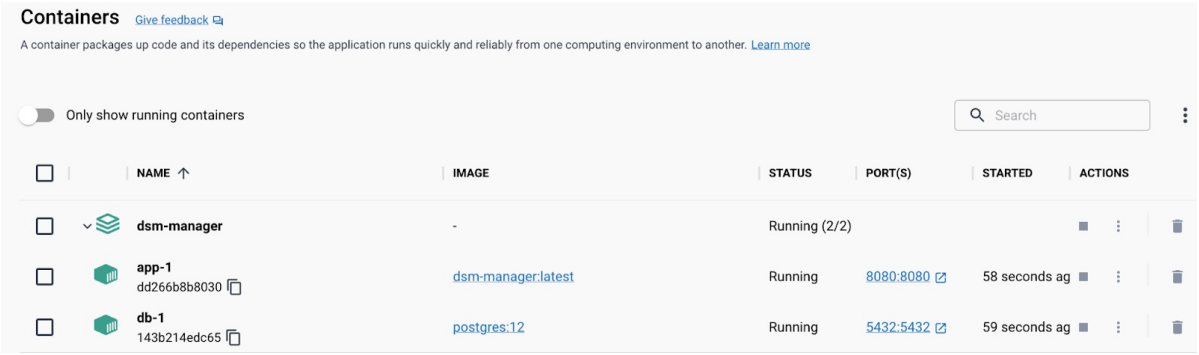

Рис. 19: Запущенный сервис dsm-manager через Docker compose.

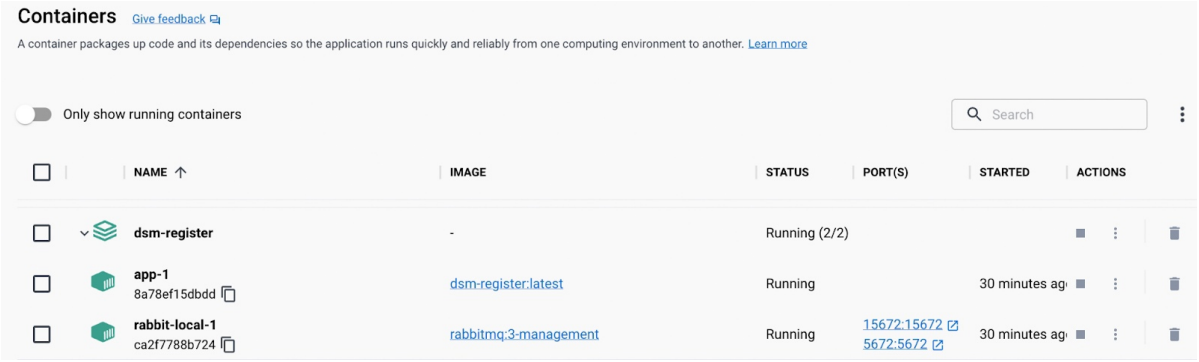

Рис. 20: Запущенный сервис dsm-register через Docker compose.

Все сервисы системы управления данными размещены в общей локальной сети. Если вернуться к файлам настроек, то можно обратить внимание, что в качестве HOST-параметра объектов инфраструктуры указаны их имена Docker-контейнеров.

При запуске Docker-контейнеров сервисов идёт окончательная настройка инфраструктуры (см. логи на листинге 4 и 5).

- *Для сервиса dsm-manager*: миграция схемы БД, заполнение справочных таблиц и запуск Gunicorn сервера с передачей ему экземпляра ASGI приложения;

- *Для сервиса dsm-register*: настройка очередей RabbitMQ и запуск служб приёма сообщений.

```
2023-05-10 01:55:13 INFO  [alembic.runtime.migration] Context impl
   PostgresqlImpl.
2023-05-10 01:55:13 INFO  [alembic.runtime.migration] Will assume
   transactional DDL.
2023-05-10 01:55:13 INFO:     Started server process [9]
2023-05-10 01:55:13 INFO:     Waiting for application startup.
2023-05-10 01:55:13 INFO:     Application startup complete.
2023-05-10 01:55:13 INFO:     Uvicorn running on http://0.0.0.0:8080 (Press
    CTRL+C to quit)
```

Листинг 4: Логи запуска приложения dsm-manager.

```
2023-05-10 16:42:23 INFO: Pika version 1.3.1 connecting to ('172.19.0.2',
   5672)
2023-05-10 16:42:23 INFO: Socket connected: <socket.socket fd=6, family=2,
    type=1, proto=6, laddr=('172.19.0.3', 36388), raddr=('172.19.0.2',
   5672)>
2023-05-10 16:42:23 INFO: Streaming transport linked up: (<pika.adapters.
   utils.io_services_utils._AsyncPlaintextTransport>,
   _StreamingProtocolShim: <SelectConnection PROTOCOL transport=<pika.
   adapters.utils.io_services_utils._AsyncPlaintextTransport> params=<
   ConnectionParameters host=rabbit-local port=5672 virtual_host=/ ssl=
   False>>).
2023-05-10 16:42:23 INFO: AMQPConnector - reporting success: <
   SelectConnection OPEN transport=<pika.adapters.utils.io_services_utils.
   _AsyncPlaintextTransport> params=<ConnectionParameters host=rabbit-
   local port=5672 virtual_host=/ ssl=False>>
2023-05-10 16:42:23 INFO: AMQPConnectionWorkflow - reporting success: <
   SelectConnection OPEN transport=<pika.adapters.utils.io_services_utils.
   _AsyncPlaintextTransport> params=<ConnectionParameters host=rabbit-
   local port=5672 virtual_host=/ ssl=False>>
2023-05-10 16:42:23 INFO: Connection workflow succeeded: <SelectConnection
   OPEN transport=<pika.adapters.utils.io_services_utils.
   _AsyncPlaintextTransport> params=<ConnectionParameters host=rabbit-
   local port=5672 virtual_host=/ ssl=False>>
2023-05-10 16:42:23 INFO: Created channel=1
2023-05-10 16:42:23 INFO: [DSM-REGISTER-I001] [RABBIT-SERVER] Build rabbit
   components [in /src/app/executer/rabbit/rabbit_server.py:65
2023-05-10 16:42:23 INFO: [DSM-REGISTER-I002] [RABBIT-SERVER] Start rabbit
   consumers [in /src/app/executer/rabbit/rabbit_server.py:67
```

Листинг 5: Логи запуска приложения dsm-register.

В завершении подготовки стенда потребовалось настроить **доступ остальных систем к системе управления данными**. Т.к. стенды всех систем находятся в одной компьютерной сети, потребовалось только открыть нужные порты.

По умолчанию на стенде закрыты все порты (кроме SSH для входа на стенд) с помощью linux firewall – iptables [39]. Для внешнего взаимодействия нужно открыть порты, которые обслуживаются HTTP-сервером dsm-manager и брокером сообщений dsm-register. Это 8080 и 5672 соответственно (см. рис. 21).

```
dmitrytv@[redacted]->/home/dmitrytv (0)
> sudo iptables -I INPUT -p tcp -m tcp --dport 8080 -j ACCEPT
dmitrytv@[redacted]->/home/dmitrytv (0)
> sudo iptables -I INPUT -p tcp -m tcp --dport 5672 -j ACCEPT
dmitrytv@[redacted]->/home/dmitrytv (0)
> sudo service iptables save
iptables: Saving firewall rules to /etc/sysconfig/iptables:[  OK  ]
dmitrytv@[redacted]->/home/dmitrytv (0)
> sudo iptables -L INPUT -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:5672
ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:8080
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     2    --  0.0.0.0/0            0.0.0.0/0
ACCEPT     tcp  --  [redacted]           0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
ACCEPT     tcp  --                       0.0.0.0/0           state NEW tcp dpt:22
REJECT     all  --  0.0.0.0/0            0.0.0.0/0           reject-with icmp-host-prohibited
```

Рис. 21: Настройка firewall на стенде системы управления данными.

5.2 Проверка работоспособности

Для того, чтобы убедиться в корректности работы настроенной инфраструктуры, достаточно проверить доступность сервисов, а также их базовые наборы функций. С этой целью, проверку будем выполнять со стенда системы управления рабочим процессом.

Проверка работоспособности сервиса dsm-manager. Попробуем открыть новый набор. Убедимся, что сервис не возвращает ошибку, а также то, что данные действительно разместились в каталоге, запросив список наборов (см. рис. 22).

Рис. 22: Проверка открытия набора через dsm-manager.

Проверка работоспособности сервиса dsm-register.

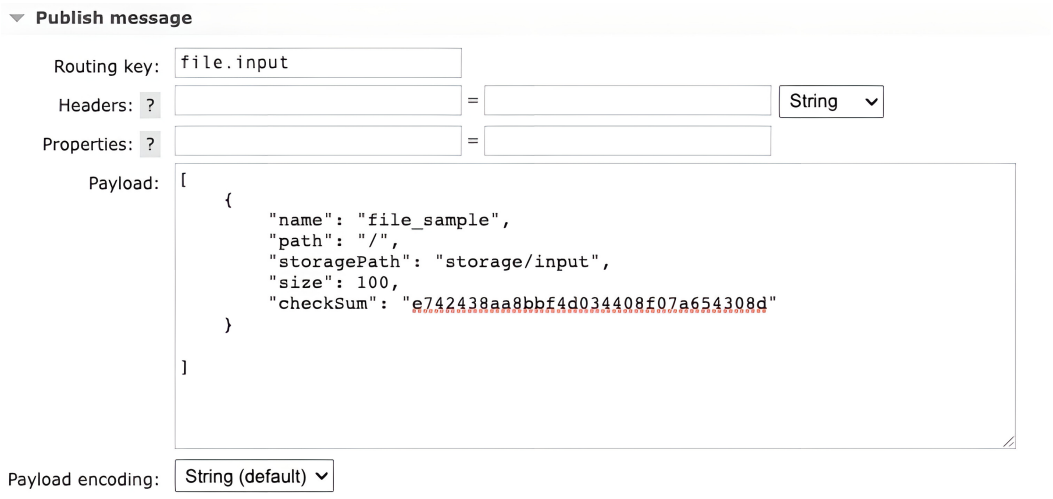

Рис. 23: Проверка отправки сообщения в очередь входных файлов dsm-register.

Для начала попробуем через панель администрирования RabbitMQ

отправить корректное сообщение с информацией о входном файле в очередь dsm.register.file.input (см. рис. 23). На листинге 6 можно увидеть лог приёма сообщения.

```
2023-05-10 17:23:28 INFO: [DSM-REGISTER-I004] [CONSUMER-HANDLER] Received
   message [FileInputDto(name='file_sample', path='/', storage_path='
   storage/input', size=100, check_sum='e742438aa8bbf4d034408f07a654308d')
   ] from queue 'dsm.register.file.input' with delivery tag 1 [in /src/app
   /executer/rabbit/consumers/base/consumer_handler.py:48
```

Листинг 6: Лог приёма сообщения из очереди входных файлов.

Далее, также проверим механизм отклонения некорректных сообщений. В очередь dsm.register.file.input отправим сообщение «bad msg». На листинге 7 теперь отображается лог ошибки приёма сообщения, а на рис. 24 видим, что сообщение попало в «очередь мёртвых сообщений».

```
2023-05-10 17:35:24 ERROR: [DSM-REGISTER-E001] [CONSUMER-HANDLER] Error on
   consume msg b'bad msg': Expecting value: line 1 column 1 (char 0) [in /
   src/app/executer/rabbit/consumers/base/consumer_handler.py:30
```

Листинг 7: Лог ошибки приёма сообщения из очереди входных файлов.

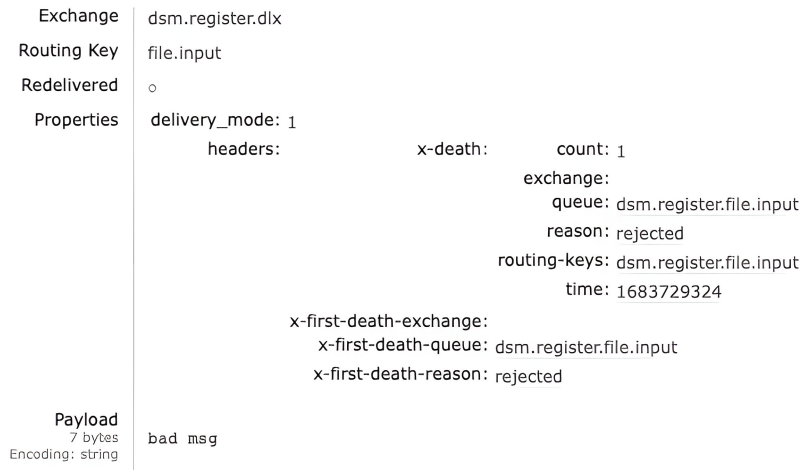

Рис. 24: Сообщение, попавшее в «очередь мёртвых сообщений».

5.3 Разработка тест-кейсов

Далее, для проверки работоспособности бизнес-сценариев, в первую очередь, требуется разработать набор тест-кейсов (см. таблицу 8).

Таблица 8: Набор тест-кейсов для проверки бизнес-сценариев

Название	**Действие**	**Ожидаемый результат**
Приём входных файлов с DAQ	Вручную разместить файлы во входной директории. Отправить в очередь dsm.register.file.input информацию о входных файлах.	Сервис dsm-register принял сообщение успешно. Данные разместились в каталоге dsm-manager (провериить через REST API): информация о входном наборе, а также список всех фалов.
Начало работы	Запустить сервис «начало работы» у системы управления процессом.	В панели администрирования системы управления процессом имеется информация о полученном входном наборе файлов. Записана временная метка забора данных.
Открытие новых наборов	Запустить сервис конфигурирования заданий у системы управления процессом.	В REST API сервиса dsm-manager имеется информация о новых открытых наборов для размещения результата выполнения задания.

Регистрация новых файлов	Запустить сервисы выполнения заданий у системы управления нагрузкой, а также сами пилотные приложения.	В логах сервиса dsm-register имеется информация об успешном приёме сообщений в очереди dsm.register.file.process регистрации файлов. В панели администрирования RabbitMQ очереди с некорректными сообщениями пустые. В REST API сервиса dsm-manager имеется информация о зарегистрированных файлах. Файлы прикреплены к открытым наборам.
Закрытие набора	Дождаться завершения выполнения задания системой управления нагрузкой.	В логах сервиса dsm-register имеется информация об успешном приёме сообщения в очереди dsm.register.dataset.close закрытия набора. Корректно отрабатывает механизм отложенного приёма сообщения в том случае, если контрольный список файлов не совпадает со списком зарегистрированных файлов. В панели администрирования RabbitMQ очереди с некорректными сообщениями пустые. В REST API сервиса dsm-manager имеется информация о закрытие набора.

Удаление набора	Дождаться завершения обработки выполненного задания системой управления процессом.	В логах сервиса dsm-register имеется информация об успешном приёме заявки в очереди dsm.register.dataset.delete по удалению наборов. В панели администрирования RabbitMQ очереди с некорректными сообщениями пустые. Статус набора сменился на TO_DELETE. В логах сервиса dsm-inspector есть информация о запуске процесса удаления набора. По завершению файлы по набору отсутствуют на хранилище, а статус у набора и файлов DELETED.
Выгрузка набора	Дождаться завершения всего процесса.	В логах сервиса dsm-register имеется информация об успешном приёме заявки в очереди dsm.register.dataset.upload по выгрузке наборов. В панели администрирования RabbitMQ очереди с некорректными сообщениями пустые. Статус набора сменился на TO_UPLOAD. В логах сервиса dsm-inspector есть информация о запуске процесса выгрузки набора. Статус у набора и файлов UPLOADING.

Также должны быть проведён ряд экспериментов по исследованию масштабируемости, отказоустойчивости и производительности системы.

Заключение

В данной дипломной работе была спроектирована система управления данными для специализированной вычислительной установки «SPD On-Line Filter». Все поставленные задачи были выполнены. Во-первых, подготовлены все необходимые материалы для разработки системы управления данными: описание данных и их организации, схема основного потока данных, требуемая функциональность от системы управления данными, а также концептуальная архитектура системы. Во-вторых, спроектированы внутренние составляющие каждой из компонент системы управления данными, а также интерфейсы их внутреннего и внешнего взаимодействия.

Помимо этого, были проведены работы по прототипированию части сервисов системы управления данными. Выбран стек технологий, проработана внутренняя архитектура, разработан базовый каркас приложения, а также часть его функциональности.

Для дальнейшего апробирования решения, было настроено окружение на инфраструктуре МЛИТ ОИЯИ и разработаны тест-кейсы для проверки всех бизнес-сценариев.

Далее планируется завершить прототипирование системы управления данными «SPD On-Line filter», доработать взаимодействие с остальными системами, провести полноценное тестирование на эмулированных данных и в конечном счёте начать интегрировать систему с прикладным ПО.

Список литературы

[1] Seven-Year Plan for the Development of JINR for 2024–2030 [Электронный ресурс] URL: https://indico-hlit.jinr.ru/event/329/contributions/1995/attachments/578/1035/01_Director_next_7YP_for_LIT.pdf (дата обращения: 20.11.2022)

[2] NICA [Электронный ресурс] URL: https://nica.jinr.ru/ru/ (дата обращения: 20.11.2022)

[3] Инфраструктура комплекса NICA [Электронный ресурс] URL: https://nica.jinr.ru/ru/complex.php (дата обращения: 20.11.2022)

[4] Эксперимент SPD [Электронный ресурс] URL: http://spd.jinr.ru/wp-content/uploads/2021/04/SPD_Korzenev_DIS2021.pdf (дата обращения: 21.11.2022)

[5] Экспериментальная установка SPD [Электронный ресурс] URL: https://nica.jinr.ru/ru/projects/spd.php (дата обращения: 21.11.2022)

[6] Иллюстрация «события» в коллайдере [Электронный ресурс] URL: http://lhc-machine-outreach.web.cern.ch/collisions.htm (дата обращения: 02.12.2022)

[7] DAQ. Сбор и управление данными NICA [Электронный ресурс] URL: https://dewesoft.com/blog/what-is-data-acquisition (дата обращения: 02.12.2022)

[8] Oleynik D. Data processing in HEP experiments [Электронный ресурс] // URL: https://indico-hlit.jinr.ru/event/329/contributions/1972/attachments/575/1032/HEPNICA_computing_2022.pdf (дата обращения: 10.12.2022)

[9] Barisits, M., Beermann, T., Berghaus, F. et al. Rucio: Scientific Data Management. Comput Softw Big Sci 3, 11 (2019). https://doi.org/10.1007/s41781-019-0026-3

[10] LFC – data management server of LCG [Электронный ресурс] // URL: https://lcgdm.web.cern.ch/lfc (дата обращения: 10.12.2022)

[11] SRM Documentation [Электронный ресурс] // URL: https://dmc-docs.web.cern.ch/dmc-docs/srm-ifce.html (дата обращения: 10.02.2023)

[12] GFAL Documentation [Электронный ресурс] // URL: https://dmc-docs.web.cern.ch/dmc-docs/gfal2/gfal2.html (дата обращения: 12.02.2023)

[13] ALICE O2 Project [Электронный ресурс] // URL: https://alice-o2-project.web.cern.ch/ (дата обращения: 15.12.2022)

[14] Alessio F., Barandela C., Brarda L. et al. LHCb Online event processing and filtering // Journal of Physics: Conference Series, Volume 119, ONLINE COMPUTING (2008). https://doi.org/10.1088/1742-6596/119/2/022003

[15] LHCb DAQ & event filter in 2021-2024 [Электронный ресурс] URL: https://doi.org/10.1088/1742-6596/119/2/022003https://indico.cern.ch/event/1037447/contributions/4379026/attachments/2256520/3829068/daq-muon-collider-workshop-0621.pdf (дата обращения: 20.12.2022)

[16] 26 основных паттернов микросервисной разработки [Электронный ресурс] URL: https://mcs.mail.ru/blog/26-osnovnyh-patternov-mikroservisnoj-razrabotki (дата обращения: 20.01.2023)

[17] Domain Driven Design: давайте не будем усложнять [Электронный ресурс] // URL: https://tproger.ru/articles/domain-driven-design-davajte-ne-budem-uslozhnyat/ (дата обращения: 10.02.2023)

[18] PostgreSQL 12.15 Documentation [Электронный ресурс] // URL: https://www.postgresql.org/docs/12/index.html (дата обращения: 17.02.2023)

[19] Нормализация отношений. Шесть нормальных форм [Электронный ресурс] // URL: https://habr.com/ru/articles/254773/ (дата обращения: 18.02.2023)

[20] Связи между таблицами базы данных [Электронный ресурс] // URL: https://habr.com/ru/articles/488054/ (дата обращения: 20.02.2023)

[21] Тема 13: Триггеры в SQL на примере базы данных SQLite [Электронный ресурс] // URL: https://zametkinapolyah.ru/zametki-o-mysql/tema-13-triggery-v-sql-na-primere-bazy-dannyx-sqlite.html (дата обращения: 10.03.2023)

[22] Партиционирование в PostgreSQL – Что? Зачем? Как? [Электронный ресурс] // URL: https://habr.com/ru/articles/273933/ (дата обращения: 15.03.2023)

[23] Индексы в PostgreSQL [Электронный ресурс] // URL: https://tproger.ru/articles/indeksy-v-postgresql/ (дата обращения: 17.03.2023)

[24] B-Tree индекс и его производные в PostgreSQL [Электронный ресурс] // URL: https://habr.com/ru/companies/quadcode/articles/696498/ (дата обращения: 20.03.2023)

[25] RabbitMQ Documentation [Электронный ресурс] // URL: https://www.rabbitmq.com/documentation.html (дата обращения: 11.02.2023)

[26] Планировщик CRON – Запуск программ по расписанию [Электронный ресурс] // URL: https://wiki.merionet.ru/servernye-resheniya/39/planirovshhik-cron-zapusk-programm-po-raspisaniyu/ (дата обращения: 19.02.2023)

[27] FastAPI Documentation [Электронный ресурс] // URL: https://fastapi.tiangolo.com/ (дата обращения: 10.03.2023)

[28] Web Framework Benchmarks [Электронный ресурс] // URL: https://www.techempower.com/benchmarks/#section=data-r21 (дата обращения: 10.03.2023)

[29] Swagger (OpenAPI 3.0) [Электронный ресурс] // URL: https://habr.com/ru/articles/541592/ (дата обращения: 15.03.2023)

[30] What is Code-First? [Электронный ресурс] // URL: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx (дата обращения: 15.03.2023)

[31] Celery. Periodic Tasks [Электронный ресурс] // URL: https://docs.celeryq.dev/en/stable/userguide/periodic-tasks.html (дата обращения: 22.03.2023)

[32] Alembic migrations [Электронный ресурс] // URL: https://blog.volodichev.com/alembic (дата обращения: 27.03.2023)

[33] Паттерн «Репозиторий». Основы и разъяснения [Электронный ресурс] // URL: https://habr.com/ru/articles/248505/ (дата обращения: 28.03.2023)

[34] SOLID – принципы объектно-ориентированного программирования [Электронный ресурс] // URL: https://web-creator.ru/articles/solid (дата обращения: 01.04.2023)

[35] Docker Documentation [Электронный ресурс] // URL: https://docs.docker.com/get-started/ (дата обращения: 20.01.2023)

[36] RabbitMQ. Pika Python client [Электронный ресурс] // URL: https://www.rabbitmq.com/tutorials/tutorial-one-python.html (дата обращения: 10.04.2023)

[37] Очереди недоставленных сообщений в Apache Kafka и RabbitMQ [Электронный ресурс] // URL: https://bigdataschool.ru/blog/dead-letter-queues-in-kafka-and-rabbitmq.html (дата обращения: 15.04.2023)

[38] Poetry Documentation [Электронный ресурс] // URL: https://python-poetry.org/docs/ (дата обращения: 02.02.2023)

[39] Iptables [Электронный ресурс] // URL: https://ru.wikibooks.org/wiki/Iptables (дата обращения: 05.05.2023)

```
class DatFile(DatBase):
    __tablename__ = "dat_file"


    id = Column(
        UUIDType(binary=False), primary_key=True, index=True, default=uuid.
    uuid4
    )
    name = Column(String(length=255), unique=True, nullable=False)
    path = Column(String(length=255), nullable=False)
    storage_id = Column(
        UUIDType(binary=False), ForeignKey("dat_storage.id"), nullable=
    False
    )
    size = Column(Integer, nullable=False)
    check_sum = Column(String(length=255), nullable=False)
    datasets: Mapped[list[DatDataset]] = relationship(
        secondary=dat_association_file_dataset
    )
    status_code = Column(
        String(length=20), ForeignKey("dic_file_status.code"), nullable=
    False
    )
    created_date: Mapped[datetime] = Column(
        DateTime(timezone=True), server_default=func.now(), nullable=False
    )
    updated_date: Mapped[datetime] = Column(
        DateTime(timezone=True),
        server_default=func.now(),
        onupdate=func.now(),
        nullable=False,
    )
```

Листинг 8: Пример ORM модели каталога файлов.

```
op.create_table(
    "dat_file",
    sa.Column(
        "id", sqlalchemy_utils.types.uuid.UUIDType(binary=False), nullable=
    False
    ),
    sa.Column("name", sa.String(length=255), nullable=False),
    sa.Column("path", sa.String(length=255), nullable=False),
    sa.Column(
        "storage_id",
        sqlalchemy_utils.types.uuid.UUIDType(binary=False),
        nullable=False,
    ),
    sa.Column("size", sa.Integer(), nullable=False),
    sa.Column("check_sum", sa.String(length=255), nullable=False),
    sa.Column("status_code", sa.String(length=20), nullable=False),
    sa.Column(
        "created_date",
        sa.DateTime(timezone=True),
        server_default=sa.text("now()"),
        nullable=False,
    ),
    sa.Column(
        "updated_date",
        sa.DateTime(timezone=True),
        server_default=sa.text("now()"),
        nullable=False,
    ),
    sa.ForeignKeyConstraint(
        ["status_code"],
        ["dic_file_status.code"],
    ),
    sa.ForeignKeyConstraint(
        ["storage_id"],
        ["dat_storage.id"],
    ),
    sa.PrimaryKeyConstraint("id"),
    sa.UniqueConstraint("name"),
)
```

Листинг 9: Скрипт миграции для таблицы с каталогом файлов.

```
class FileStatus(Enum):
    """
    Enum with statuses of file.
    """

    CREATED = ("CREATED", "Файл добавлен в систему")
    DAMAGED = ("DAMAGED", "Файл повреждён")
    TO_DELETE = ("TO_DELETE", "Файл с пометкой "На удалении"")
    UPLOADING = ("UPLOADING", "Файл выгружается")
    DELETED = ("DELETED", "Файл удалён из системы")

    def __init__(self, code, description):
        self.code = code
        self.description = description
```

Листинг 10: Класс перечисления значений для справочника статусов файлов.

```
class BaseSqlRepository:
    """Basic sql repository."""


    def __init__(
        self,
        session_factory: Callable[..., AbstractContextManager[Session]],
        entity_class: object,
    ) -> None:
        self.session_factory = session_factory
        self.entity_class = entity_class


    def add(self, entity: object) -> object:
        """Add entity."""
        with self.session_factory() as session:
            session.add(entity)
            session.commit()
            session.refresh(entity)
            return entity
    ...

class DatFileRepository(BaseSqlRepository):
    """Repository of dat_file."""


    def __init__(
        self, session_factory: Callable[..., AbstractContextManager[Session
    ]]
    ) -> None:
        super(DatFileRepository, self).__init__(session_factory, DatFile)
        self.session_factory = session_factory

created_dat_file = file_repository.add(dat_file)
```

Листинг 11: Пример репозитория для каталога файлов.

```
def __rabbit_bootstrap(channel: Channel, settings: RabbitSettings) -> None
    :
    channel.exchange_declare(
        exchange=settings.CONSUMING_EXCHANGE, exchange_type="direct"
    )
    channel.queue_declare(
        queue=settings.queue_file_input(),
        durable=True,
    )
    channel.queue_bind(
        exchange=settings.CONSUMING_EXCHANGE,
        queue=settings.queue_file_input(),
        routing_key=settings.CONSUMING_ROUTING_KEY_FILE_INPUT,
    )
```

Листинг 12: Построение очереди RabbitMQ для входных файлов.