Санкт–Петербургский государственный университет

Гребень Никита Владимирович

Выпускная квалификационная работа

Планировщик задач для специализированной распределенной вычислительной системы SPD On-Line filter

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

Научный руководитель:
кафедра компьютерного моделирования и многопроцессорных систем,
д.т.н. Дегтярев Александр Борисович

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

Рецензент:
заместитель начальника научно-экспериментального отдела встречных пучков, ЛЯП, ОИЯИ,
к.ф. – м.н. Жемчугов Алексей Сергеевич

Санкт-Петербург
2023 г.

Содержание

Введение

SPD - это проектируемый эксперимент по спиновой физике на одной из экспериментальных установок, входящих в мегасайнс проект NICA (Рис. 1), которая строится в ОИЯИ (г. Дубна, Российская Федерация) [1]. Основная цель эксперимента - проверка основ квантовой хромодинамики (КХД) 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$^{-1}$. Для этого будут проведены измерения, зависящие от поперечного импульса партонных распределений (TMD PDF) для глюонов в таких сложных процессах, как рождение очарованных частиц, состояний чармония и прямых фотонов. Детектор SPD задуман как универсальный 4π-спектрометр, основанный на современных технологиях [2]. Общее количество каналов регистрации в установке SPD составляет около 500000. С учетом проектной частоты возникновения интересных для эксперимент взаимодействий около 3 МГц, суммарный поток данных с детектора можно оценить как 20 ГБ/с, что эквивалентно 200 ПБ/год (для эксперимента предполагается выделить 30% времени работы коллайдера).

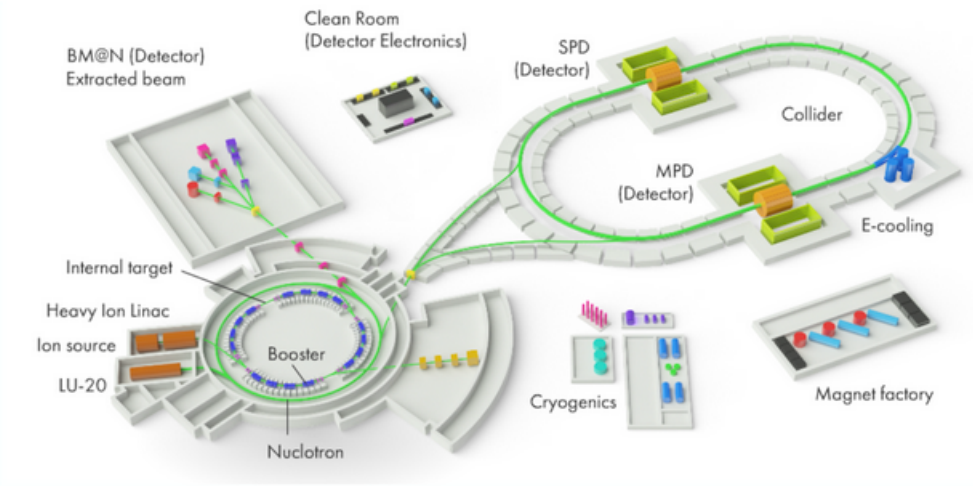


Рис. 1 Схема комплекса NICA.

Целью вычислений в SPD является получение физического результата. Перед SPD Online Filter стоит задача существенного уменьшения потока

первичных данных, путем отбора только тех событий, которые интересны в рамках текущего исследования. В то время как простые операции, такие как удаление шума, еще может выполнять DAQ, онлайн-фильтр предназначен для быстрой частичной реконструкции событий и выборки данных. Задача усложняется тем, что на установке реализуется безтриггерная система съема данных (для минимизации систематических ошибкок при изучении микроскопических эффектов), а это значит - что в полученом потоке необходимо сначала выявить события, а уже потом их отфильтровать [3; 4].

Таким образом, SPD On-Line filter будет представлять собой программно-аппаратный комплекс для высокопропускной обработки данных. Аппаратная составляющая будет включать в себя набор многоядерных вычислительных узлов, построенных на современных технологиях, высокопроизводительные системы хранения данных, некоторое количество управляющих серверов. Программная составляющая включает в себя не только специализированное прикладное программное обеспечение, но и комплекс промежуточного программного обеспечения «SPD On-Line filter» – «Visor», в задачу которого будет входить организация и реализация многоступенчатых процессов обработки данных.

Основной целью онлайн-фильтра является снижение скорости передачи данных как минимум в 20 раз, чтобы годовой прирост данных, включая смоделированные выборки, оставался в пределах 10 ПБ. Затем данные передаются на Tier-1, где происходит полная реконструкция и постоянное хранение данных.

В рамках магистерской работы разрабатывался прототип программной системы для управления большими вычислительными нагрузками в контексте высокопропускных вычислений в распределенной среде. В ходе данной работы были выработаны механизмы взаимодействия с другими сервисами в рамках проекта «SPD On-Line filter» – «Visor», разработана архитектура промежуточного программного обеспечения, обосновывается выбор программных средств, проведён анализ алгоритмов распределения заданий, а также представлены первые результаты прототипа.

Постановка задачи

Согласно установленным техническим требованиям, вычислительная система «SPD On-Line filter» выполняет следующий цикл преобразования над данными:

- Реорганизация набора сигналов с детектора к формату, где каждый сигнал привязан к событию.
- Фильтрация событий и выгрузка в хранилище данных “интересных” событий.
- Упорядочивание выходных файлов,
- Подготовка данных для последующей обработки и реконструкции событий на ресурсах Tier-1 GRID.

Для удовлетворения данным требования, система должна состоять из следующих сервисов (Рис. 2):

- Система управления и хранения данных - **DSM** (регистрация именованного набора данных о событиях, каталогизация, контроль согласованности)
- Система управления процессом - **WFMS** (управления жизненным циклом рабочего процесса, создание заданий)
- Система управления нагрузкой - **WMS** (генерация, отправка задач на Pilot, управление)
- Агентское приложение - **Pilot** (запуск задач, выгрузка выходных файлов и логов в хранилище данных)

Цель данной работы состоит в проектировании сервиса управления нагрузкой (**WMS**) на вычислительные ресурсы посредством генерирования оптимального количества задач выполняющих обработку данных. Задачи генерируются на основе поступающих в систему заданий от системы управления процессом. Система должна поддерживать одновременную обработку набора заданий.

Функциональные требования к системе управления нагрузкой

1. Регистрация: размещение информации и метаданных, необходимых для формирования задач.

2. Разбиение: генерация оптимального количества задач, для задействования всех вычислительных ресурсов и контролируемого размера очереди задач.

3. Управление: отслеживать состояния задач, выполняемые на пилотах: перезапускать, прекращать выполнение.

4. Планирование: обеспечение равномерности нагрузки и скорости прохождения заданий.

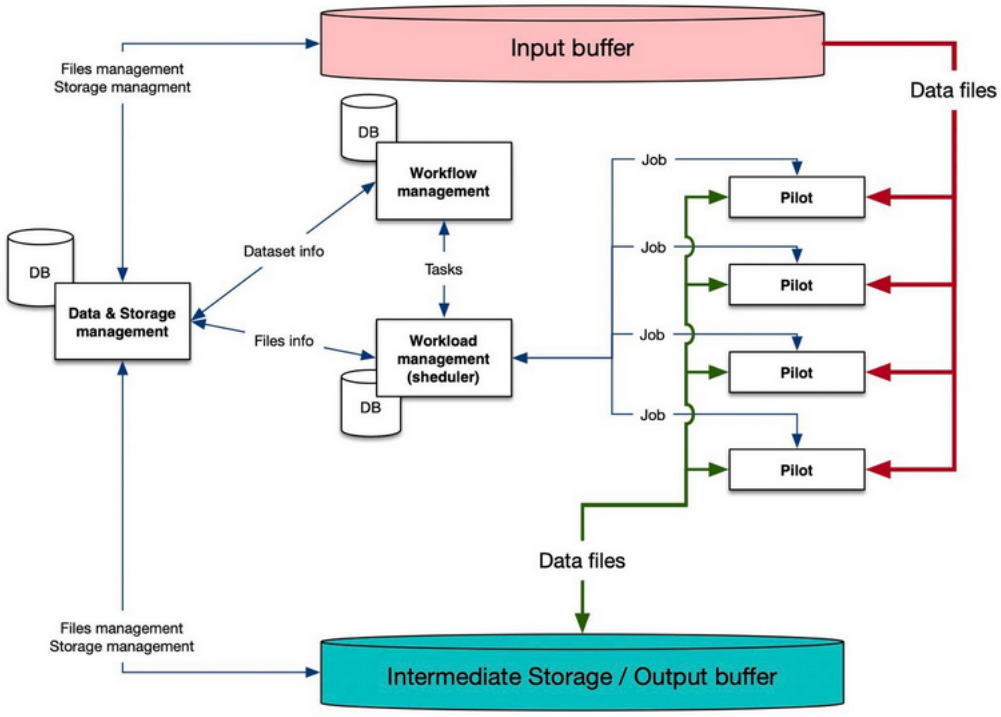


Рис. 2 Концептуальная архитектура системы SPD On-Line filter.

Обзор литературы

Существует несколько решений в области управления рабочих нагрузок высокопропускных вычислений (HTC) в распределенных системах. Высокопропускные вычисления (HTC) относится к выполнению большого количества независимых задач, каждая из которых требует относительно небольшого количества ресурсов [6].

Один из распространенных способов реализации высокопропускных систем является использование, так называемое, пилотной парадигмы. Которая включает отправку задания-заполнителя (называемого пилотным) на вычислительный ресурс, а затем использование пилотного приложения для исполнения задач на выделенном ресурсе. Таким образом, задачи HTC могут обходить очередь локальной системы управления ресурсами и получать выгоду от более быстрого и гибкого планирования.

Все они используют пилот-парадигму, которая позволяет отделить отправку рабочей нагрузки (workload) от аллокации ресурсов, а также включить многоуровневое планирование и динамическое управление ресурсами. На данный момент наиболее известны в научной среде следующие системы:

- WMS-PanDA — это система управления рабочей нагрузкой (WMS), разработанная в рамках эксперимента ATLAS в ЦЕРНе для обеспечения обработки экспериментальных данных в неоднородной географически распределенной вычислительной среде. WMS-PanDA состоит из нескольких подсистем, таких как PanDA Server, PanDA Pilot, JEDI, PanDA Monitoring и ряда других. WMS-PanDA использует пилотные приложения для выполнения задач в среде грид, добровольных вычислительных системах, облачных инфраструктурах и суперкомпьютерах и поддерживает различные типы рабочих нагрузок.

- DIRAC — представляет собой систему, обеспечивающую основу для распределенных вычислений и управления данными для различных научных сообществ (физика элементарных частиц, астрофизика, космология и т.д.). Состоит из нескольких сервисов, которые взаимодействуют друг с другом и с внешними ресурсами [7]. Одним из ее компонен-

тов является система управления рабочей нагрузкой (WMS), которая обрабатывает отправку и выполнение задач (jobs) на различных типах ресурсов, включая кластеры высокопроизводительных вычислений [8]. Также содержит таким компоненты как система управления данными (DMS), система мониторинга, система конфигурации и т. д.

- RADICAL-Pilot. — связующее программное обеспечение, обеспечивающее гибкое и масштабируемый способ выполнения множества одновременных задач в распределенных высокопроизводительные инфраструктурах [9]. RADICAL состоит из двух основных компонентов: Pilot Manager, который отправляет пилотные проекты на ресурсы HPC и управляет их жизненным циклом; и Unit Manager, который отправляет задачи пилотам и управляет их выполнением. RADICAL-Pilot можно использовать автономно, а также интегрировать с другими инструментами в качестве системы выполнения [10]. он ориентирован для адаптации использования ресурсов крупных суперкомпьютеров к выполнению относительно "небольших"задач. Однако RP является менее зрелым и стабильным, чем DIRAC и WMS-PanDA, поскольку все еще находится в стадии активной разработки и тестирования [11]. У него также есть недостаток, заключающийся в меньшем количестве функций, чем у DIRAC и WMS-PanDA.

Термины «задачи» и «задания» в этих системах используются по-разному, и они могут не иметь однозначного соответствия. Однако, в общем, их можно понимать следующим образом:

- Task — это единица работы, которая может выполняться независимо и требует определенного количества ресурсов (например, ядер CPU, памяти, дискового пространства и т. д.)

- Job — некий контейнер, который содержит одну или несколько задач и передается на вычислительный узел для выполнения.

- Pilot — это “заполнитель”, который запрашивает ресурсы и использует их для выполнения задач, минуя очередь ресурсов. Пилот может

запускать более одной задачи одновременно или одну за другой, в зависимости от того, сколько у него ресурсов.

Таблица 1 Различия в терминологии

Система	Task	Job	Pilot
RADICAL-Pilot	Unit (единица рабочей нагрузки): может работать на любом пилоте, у которого есть для этого достаточно ресурсов.	Pilot: контейнер, который запрашивает некоторые ресурсы и использует их для запуска единиц нагрузки рабочей (Units).	Pilot: то же самое, что и Job.
DIRAC	Job: часть рабочей нагрузки, которая может выполняться сама по себе и требует определенных ресурсов. Может выполняться на любом пилоте, имеющем для этого достаточно ресурсов.	Job: то же, что и Task.	Pilot: контейнер, который запрашивает некоторые ресурсы и использует их для запуска задач (Jobs).

WMS-PanDA	Job: часть рабочей нагрузки, которая может выполняться сама по себе и требует определенных ресурсов. Может выполняться на любом пилоте, имеющем для этого достаточно ресурсов [12].	Task: контейнер, содержащий одну или несколько задач, разделяющих общую научную цель.	Pilot: контейнер, который запрашивает некоторые ресурсы и использует их для запуска задач (Jobs).

Все эти решения задействуют модель пилотов для вычислений в распределенных вычислительных инфраструктурах, поддерживают различные типы рабочих нагрузок и приложений, а также представляют механизмы мониторинга и отладки выполнения задач. Однако стоит подчеркнуть существенную разницу между RADICAL-Pilot по одному сторону, и DIRAC вместе с WMS-PanDA по другую. Так, ввиду того, что RADICAL-Pilot есть программное обеспечение, предоставляющее услуги и функциональные возможности приложениям, его можно интегрировать с другими системами, тогда как DIRAC и WMS-PanDA являются частью более крупных платформ, ориентированные более на предопределенные и стандартизированные рабочие процессы для конкретных научных сообществ и приложений [13].

Глава 1. Основные компоненты системы управления нагрузкой

1.1 Применимость микросервисной архитектуры

Разрабатока системы управления нагрузкой, как и других компонентов фильтра, ведётся руководствуясь принципам микросервисной архитектуры. Микросервисная архитектура - это способ проектирования программных систем, который заключается в разбиении большого и сложного приложения на набор небольших и независимых сервисов. Каждая служба, в нашем случае это **task-manager**, **task-executor**, **job-manager** и **job-executor**, имеет свою собственную функциональность, ответственность и коммуникационный интерфейс. Ниже перечислены основные причины, определяющие выбор микросервисной архитектуры [15; 16]:

- **Контролируемая производительность**: разделение приложения на небольшие автономные блоки облегчает его разработку, развертывание и дальнейшее обслуживание.
- **Масштабируемость**: Микросервисы могут независимо увеличиваться или уменьшаться в зависимости от спроса и нагрузки каждой службы. Это позволяет лучше использовать ресурсы и оптимизировать производительность.
- **Совместимость CI/CD**: Микросервисы хорошо подходят для методов непрерывной интеграции и непрерывной доставки (CI/CD), которые направлены на более быструю и частую доставку программного обеспечения.

1.2 Функциональные компоненты и их взаимосвязь

- На стороне **Workflow Management System** идёт проверка готовности датасета в **DSM** для последующей обработки, в случае успеха создается шаблон задания, который в дальнейшем передается в очередь RabbitMQ.

- **Task-Manager** принимает из очереди заданий (direct exchange) в формате JSON, добавляет в таблицу заданий, и задает статус готовности для приема на стороне Task-Executor.

- **Task-Executor** запрашивает следующее готовое задание для обработки, после его получения, микросервис запрашивает у **dsm-manager** структуру именованного набора данных (далее датасета). Затем начинается процесс генерации задач из расчета “одна задача на один файл”, что происходит в асинхронном режиме по нескольким датасетам. Другими словами происходит “дробление” датасета на условно более мелкие единицы рабочего процесса (Map-шаг).

- Сформированные задачи передаются в очередь сообщений RabbitMQ на **Job-Manager**.

- **Job-Manager** получает описания задач из очереди, добавляет по полученным описаниями соответствующие кортеже в таблицы задач, файлов и заданий. Следит за согласованностью и текущим статусом задач. Формирует имена выходных файлов и логов.

- **Job-Executor** запрашивает следующую задачу, определяет свободные, подходящие пилоты, ресурсы и посылают в соответствующие очереди на пилоты (распределяет задачи на пилоты согласно его текущей политике). В случае простоя пилотов, определяет следующую по приоритету задачу, которая пойдет в очередь.

- Пилот присылает регулярные сообщения о статусе задачи, **Job-Executor** обновляет статус задачи, регистрирует завершившиеся задачи, закрывает датасет в случае обработке всех задач либо посредством запроса от **WFMS**.

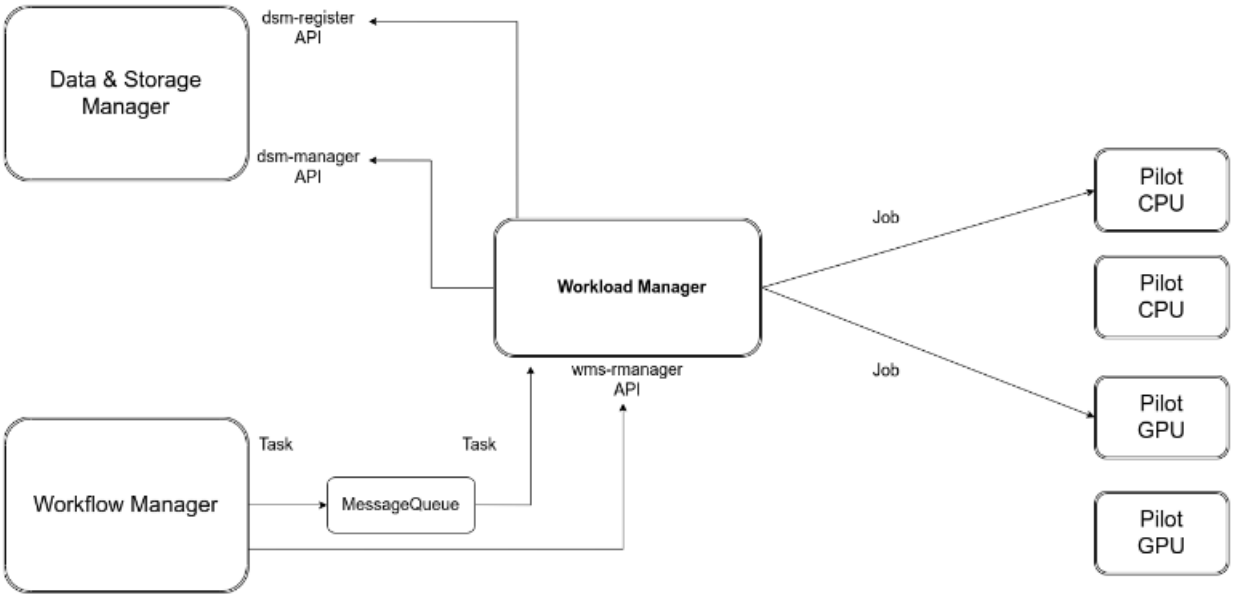


Рис. 3 Компоненты-сервисы SPD онлайн фильтра.

1.3 Задания (Tasks)

Задание представляет собой единицу рабочей нагрузки (workload) по обработке блока данных . Если же эта цель выполняется в несколько этапов, каждый шаг сопоставляется с заданием. Задание формально описывается как набор входных данных и обработчик , который произведет необходимую обработку над данными и произведет набор с выходными данными. Цель всего задания состоит в том, чтобы полностью обработать входные данные. Набор данных, которые инкапсулирует задание есть датасет (dataset).

1.3.1 Жизненный цикл заданий (Tasks)

Ниже схематично показан граф смены состояния заданий и их место нахождения в системе (Рис. 4). Синим изображены моменты нахождения в системе WMS, оранжевым — в системе WFMS.

1.3.2 Описание задания

Задание представляется в формате JSON на протяжение всего своего жизненного цикла. Ниже представлена таблица с ключами и их описание.

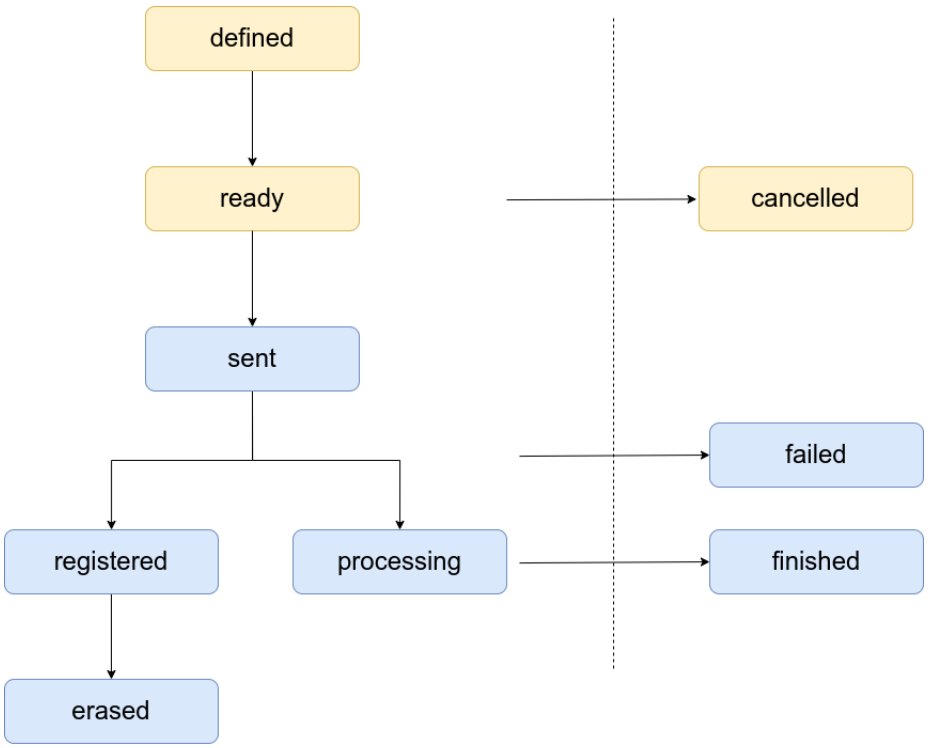


Рис. 4 Статус заданий в процессе прохождения по сервисам SPD онлайн фильтра.

Таблица 2 Конфигурация задания

Ключ	**Описание**
uid	уникальный идентификатор задания
executable	обработчик
args	входные параметры для обработчика, шаблон командной строки
rank	назначенный WFMS приоритет задания, в дальнейшем изменяющийся как функция от нескольких переменных с линейной функцией зависимости: $$rank_{i+1} = \alpha \times x_i + \beta \times y_i + \gamma \times rank_i$$ $x_i - aging, y_i - retries$

device_type	указание, какой вычислительный ресурс для выполнения задания будет задействован • CPU • GPU
mode	указание, какой тип задания будет выполняться • Map • Merge
dat_in_uid	уникальный идентификатор входного датасета
retry	максимальное количество попыток, которое будут должны выполнить соответствующие данному заданию задачи (**jobs**) в случае неудачи
Информация о выходном датасете с файлами/логами[1]	
dat_out_uid	уникальный идентификатор выходного датасета
dat_stor_url	унифицированный указатель ресурса выходного хранилища
dat_out_url	унифицированный указатель ресурса выходного датасета

[1]Путь до файла выглядит как:
<URL хранилища>/<URL датасета на хранилище>/<UID job>/<Имя файла>

1.4 Формирование задач

Задача в определенном смысле — намерено введенная/искусственная часть рабочей нагрузки, порожденная от задания. Выполнение одного задания заключается в выполнении достаточного набора однородных задач, и каждая задача выполняется на минимальном наборе вычислительных ресурсов (одна задача на один пилот). Входные датасеты каждого задания разбиваются на несколько непересекающихся подмножеств (подробнее в **Алгоритм распределения заданий**), и каждая задача получает данное подмножество для дальнейшей работы. Совокупность выходных данных в результате работы всех задач — это выходные данные всего задания.

1.4.1 Жизненный цикл задач (Jobs)

Каждая задача начинает свое существование в системе в **TaskExecutor** после получение метаданных о файле от сервиса **dsm-manager**, и заканчивает свое существование после регистрации соответствующего ей выходного файла с результатом и логом. Ниже схематично показан граф смены состояния задачи и ее места нахождения (Рис. 5). Синим изображены моменты ее нахождения в системе **WMS**, оранжевым — на пилоте.

1.4.2 Описание задач

Аналогично заданию, задачи представляется в формате JSON, некоторые поля и их значения наследуются от соответствующего задания (исполняемый файл, типа задания, флаги, приоритет). Ниже представлена таблица с ключами и их описание:

Таблица 3 Конфигурация задания

Ключ	**Описание**
uid	уникальный идентификатор задачи
executable	обработчик
args	входные параметры для обработчика, шаблон командной строки

rank	приоритет задачи
device_type	вычислительный ресурс
mode	тип задачи
task_uid	уникальный идентификатор задания, “породивший” данную задачу
task_num	количество файлов в датасете (необходимо в дальнейшем для принятия решения о закрытии датасета)
file_in_url	URL входного файла (получаем в метаинформации о файле от **dsm-manager**) • <URL хранилища><URL на хранилище><имя файла>
file_out_url	URL выходного файла — результат работы задачи: • URL файла на хранилище = <URL датасета на хранилище>/<UID job>/. . . • Имя файла = . . . • Путь файла = <URL хранилища>/<URL файла на хранилище>/<Имя файла>
file_log_url	URL выходного лога — лог файла работы прикладного ПО на пилоте

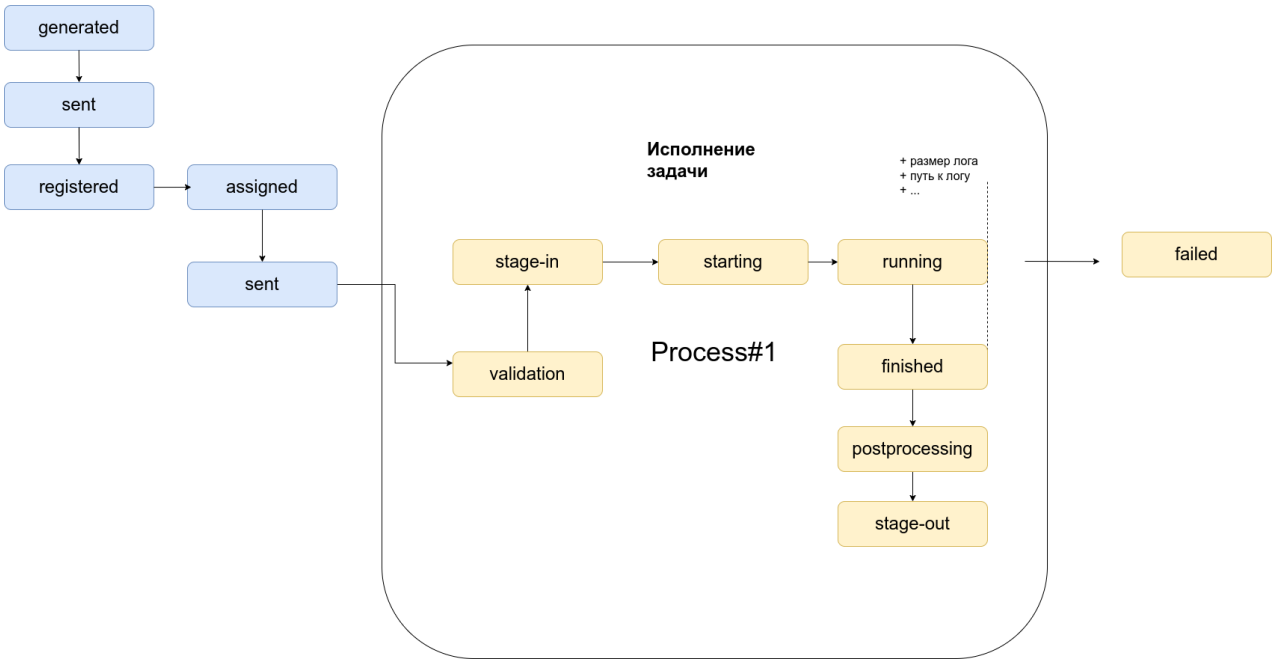


Рис. 5 Жизненный цикл задачи.

1.5 Взаимодействие WMS и DSM

DSM (Data & Storage Management) - система, обеспечивающая полный контроль над хранением, организацией, а также целостностью данных. В рамках планировщика задач существует несколько сценариев взаимодействия с системой управления данными:

1. Получение информации о содержимом датасета
   (a) Передаваемых данные:
      i. uid входного датасета
   (b) Получаемые данные:
      i. total - общее количество файлов в датасете:
      ii. data: file.info, ...,
         A. file.info:
            - <имя файла>
            - <URL хранилища>
            - <URL на хранилище>

B. доп.информация о файле (размер, check-sum)

2. Получить URL выходного хранилища по id
3. Регистрация **файлов/логов** в *datacеme*:
   (a) В качество передаваемых данных в **dsm-register** выступают:
      i. uid выходного датасета
      ii. Информация о файлах:
         A. путь до файла на хранилище (включает имя файла)
         B. ID хранилища
         C. размер файла
         D. check-sum
4. Закрытие dataset:
   (a) Передаваемых данные:
      i. uid выходного dataset
      ii. check-list - список файлов (достаточно имени файлов), которых **DSM** следует ожидать, что они присутствуют в *datacеme*, после завершения задания

1.6 Взаимодействие WMS и WFMS

В SPD онлайн фильтре **WFMS (Workflow management system)** ответственен за последовательное формирование заданий на обработку, передачу заданий в **WMS** и за контролем выполнения цепочек посредством опроса **WMS**. В качестве посредника для работы с **WMFS**, руководствуясь принципам микросервисной архитектуры, было принято решение о выделении микросервиса **Task-Manager**, представляющий как внешние, так и внутренние API (для работы с потенциально изменяющимся статусом задания и его приоритетом). Весь процесс взаимодействия между двумя системами можно представить в виде следующих сценариев:

1. Регистрация *задания* на обработку:

- **WFMS** передаёт в *RabbitMQ* сообщение с описанием задания в формате JSON (см. выше)

2. Отмена задания

   - Решение об отмене задания принимается в случае кол-ве ошибок > кол-во обработанных файлов, либо пользователем. Реализуется через *метод PUT* HTTP запроса, которые обрабатывается на стороне **Task-Manager**.

3. Запрос на статус о текущих задачах в системе (*summary*)

   - Необходим пользователю на стороне WFMS. Task-Manager вызывает внутренний API на стороне Job-Manager (для данного типа запроса задается индекс).

4. Запрос на статус о текущих выходных файлах в системе

   - *Аналогично статусам о задачах, см.выше.*

1.7 Взаимодействие WMS и Pilot

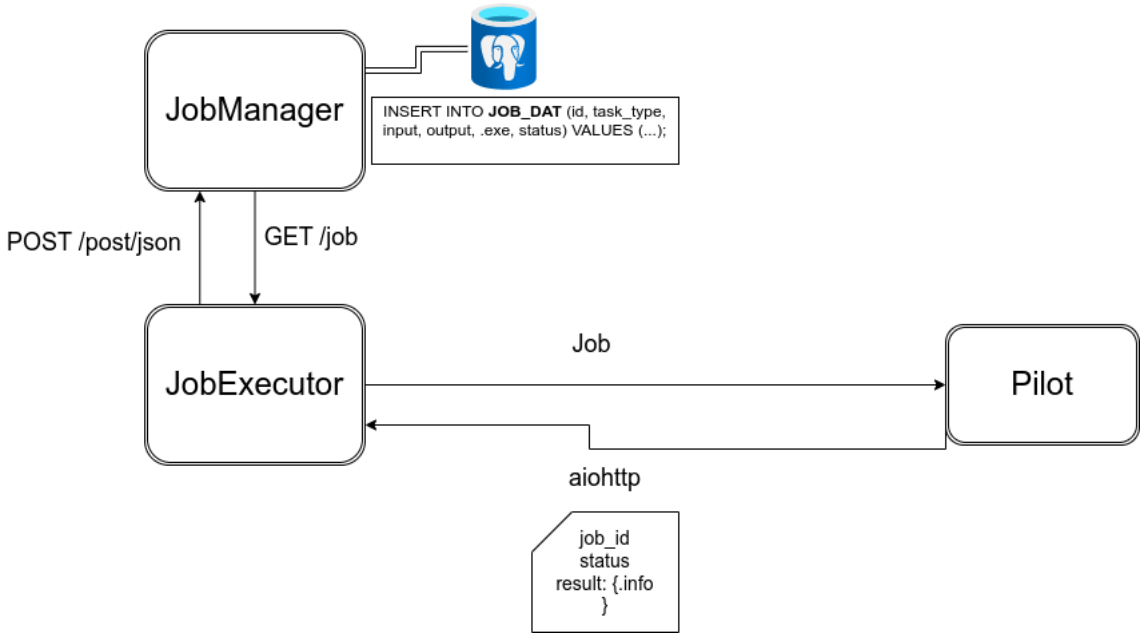


Рис. 6 Цикл жизни задачи между системами WMS и Pilot.

“Пилоты” являются универсальным подходом, позволяющим отделить выполнение единиц рабочей нагрузки от доступности ресурсов, а также осуществить планирование и динамическое использование ресурсов. Основная идея заключается в том, что вместо того, чтобы отправлять задачи непосредственно в локальные системы управления ресурсами с их собственной системой очередей и приоритетов, пилот менеджер (Pilot Manager) отправляет облегченные “заполнители” - пилоты, которые захватывают ресурсы, а затем извлекают и выполняют задачи из центральной очереди. Таким образом, пилотная система может оптимизировать использование ресурсов и выполнение задач в соответствии с различными критериями. Такой подход обеспечивают однородный интерфейс для разнородных ресурсов, скрывая сложность и разнообразие лежащего в их основе промежуточного программного обеспечения и сервисов.

В рамках данной системы пилотам доступны все ресурсы (в расчете один пилот на одну задачу), таким образом аллокация уходит на второй план, что позволяет сосредоточиться лишь на назначении задач пилотам, за что ответственен планировщик задач, решающий задачу долгосрочного планиро-

вания (**long-term scheduling**). Ниже описаны все сценарии взаимодействиях между **WMS** и пилотами:

1. Регулярно передает сообщения о статусе исполнения задачи на **Job-Executor** с заданным периодом. Передает: идентификатор пилота, параметры вычислительного узла (CPU, CPU+GPU). Возвращает статус задачи.

2. Пилот регулярно обновляет статус задачи по мере выполнения. Для этого создается ещё одна сессия aiohttp.

3. По окончанию работы передает: полный путь к выходным файлам (результат работы задачи, журнал задачи), код выхода задачи, сообщение ошибки (если есть), тип ошибки.

Глава 2. Разработка архитектуры

2.1 Описание микросервисов

Ниже расписаны функциональный характеристики основных компонентов системы **WMS**.

2.1.1 task-manager

Предоставляет следующий набор API над выполняемыми заданиями (Tasks).

Таблица 4 task-manager API

Метод	**API**	**Input**	**Output**
Получение запросов от WMFS			
Получение информации о статусе обработки задач	GET /task/<id>/summary/jobs	ID	{ $total :''$ $running :''$ $canceled :''$ $killed :''$ $failed :''$ }
Получение информации о статусе выходных файлов	GET /task/<id>/summary/files	ID	{ $total :''$ $failed :''$ }
Изменение приоритета задания	PUT /task/<id>/rank	ID задание + новый приоритет	Task info
Отмена задания	PUT /task/<id>/canceled	ID задание + новый статус	Task info
Получение запросов от TaskExecutor/Shredder			

Получить следующее неотмененное задание	GET /task/ready		Task JSON Description
Передать статус о “нарезанном” датасете	PUT /task/<id>	ID UID датасета статус	Task info
Получение запросов от JobManager			
Удаление информации о задании	DELETE /task/<id>	ID	статус

Получить статус о задании	GET
	/task/<id>
		ID	{
			$canceled :''$
			}

2.1.2 job-manager

Предоставляет следующий набор API над выполняемыми задачами.

Таблица 5 job-manager API

Метод	**API**	**Input**	**Output**
Управление информацией о задачах			
Добавить задачу в каталог	POST /job	ID status job	Job info

Получить информацию о задач	GET
	/job/<id>
		ID	{
			$retries :''$
			$status :''$
			}

Обновить статус задачи	PUT /job/<id>	ID	Job Info
Управление информацией о файлах			
Изменение статуса о файле	PUT /file/<id>	ID	response status
Удаление информации о файле	DELETE /file/<id>	ID	response status
Добавить выходной файл в каталог	POST /file	File info	response status

2.1.3 task-executor/shredder

Сервис, отвечающий за получение датасетов и информации о содержащихся в них файлах, а также за процесс генерации задач по датасету.

Таблица 6 Задачи программного модуля *task-executor*

Подзадачи	**Алгоритм**
Получить новое, готовое для обработки задание от **TaskManager**	Вызов сопрограммы (*coroutine*) на GET запрос TaskManager-a
Получить информацию о содержимом датасета	Вызов сопрограммы, реализующей GET запрос на информацию в наборе (**dsm-manager**)
Выполнить асинхронную генерацию задач по датасетам и по самим задачам внутри датасета	См. Алгоритм распределения заданий
Выгрузить новую задачу в rabbitMQ очередь	Реализована как функция обратного вызова

2.1.4 job-executor

Сервис, отвечающий за отправку задач в очереди на пилоты, регистрацию и закрытие датасета, также ответственен за обновление статуса задач.

Таблица 7 Задачи программного модуля *job-executor*

Подзадачи	**Алгоритм**
Оповещение пилота о прекращение выполнения задачи	1. Инициализация отдельной очереди оповещений, в которую будут помещаться обновленные статусы задачи: **killed/canceled**. Тип маршрутизации - прямой. 2. Установления соединения с сервером, исполование библиотеки **aio-pika** для асинхронной публикации задачи.
Отправление задач на пилот	1. Инициализация очередей двух типов: для пилотов с ресурсами *CPU* и *CPU+GPU*. 2. Определение пилота и очереди для публикации задачи, и непосредственно публикация.
Обновление статуса задачи	Вызов соответствующего API метода микросервиса **task-manager**
Получение следующей готовой задачи	Вызов соответствующего API метода микросервиса **task-manager**
Регистрация файлов/логов	1. Осуществление запроса в **task-manager**, возвращающий полное описание задачи. 2. Формирование сообщения в JSON формате и и публикация в брокер сообщения **dsm-register**.

Закрытие датасета	1. Осуществление запроса в **task-manager**, инициирующий вызов SELECT инструкции об успешно завершенных задачах в датасете. 2. Формирование сообщения в JSON формате и публикация в брокер сообщения **dsm-register**.
Приём сообщений с пилота	1. Создание **aiohttp** сессии, ожидание запроса. 2. В случае успешного завершения, обновление статуса задачи, регистрация в системе **DSM**, проверка на возможность закрытия датасета. 3. В противном случае, также, происходит обновление статуса задачи с последующей генерацией новой, счетчик попыток увеличивается на единицу и задача повторно отправляется на пилот.

2.2 Концептуальная схема Workload Management System (WSM)

Система управления нагрузкой (Рис. 7) обладает следующими характеристиками:

- Масштабируема - архитектура предполагает возможное увеличение числа микросервисов **task-executor** и **job-executor**.
- Характеризуется как распределенная, так как распределяет нагрузку по нескольким узлам сети. Относится к RESTful архитектуре.
- Микросервисы **task-executor** и **job-executor** относятся к категории не сохранающим состояния (**stateless**).

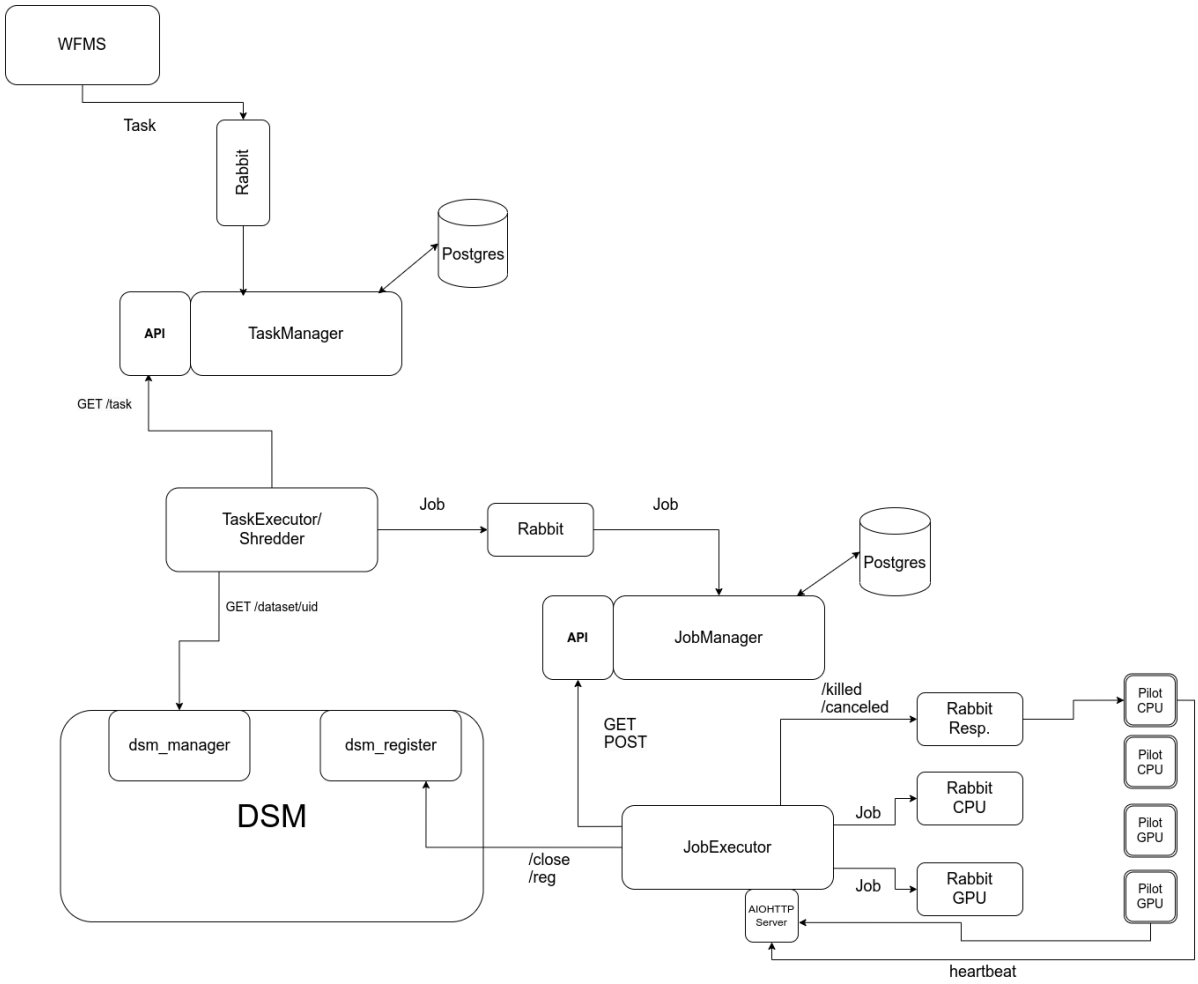


Рис. 7 Архитектура системы управления нагрузкой.

2.3 Закрытие датасета

В ходе продумывания сценариев взаимодействия между компонентами **DMS** и **WFMS**, было решено руководствоваться следующими критериями по принятию решения о закрытии датасета (завершения задания):

1. Прежде чем закрыть датасет, все задачи в датасете должны быть завершены, не важно с каким статусом (*failed/finished*)

2. По ходу своей работы **Job-Executor** регистрирует файлы в **dsm-register** по следующему критерию:

   - **on_registered** = **job.status** == **finished** OR **file.result** $\neq$ **NULL** (в случае выходных файлов)
   - В случае логов есть несколько сценариев действий, в зависимости от того, сколько было предпринято попыток (*retry*) выполнить одну и ту же задачу на пилоте:
     (a) failed $\rightarrow$ failed $\rightarrow \ldots \rightarrow$ finished — регистрировать логи, соответствующие последнему успешному статусу
     (b) failed $\rightarrow$ failed $\rightarrow \ldots \rightarrow$ failed — регистрировать всю цепочку логов

3. Если WFMS инициировал отмену задания, соответствующие задачи переводятся в статус canceled, пилоту сообщается об их приостановке. Дальнейшие действия согласованы с пунктами 1. 2.

После происходит оповещение **dsm-register** о закрытии датасета, перечисляется набор файлов. После чего следует удаление записей о задании из таблицы **TASK_DAT** после закрытия датасета.

2.4 Алгоритм распределения заданий

Процесс генерации задач должен выполняется асинхронно, учитывая возможность одновременной обработки нескольких датасетов. Общее количество возможных для обработки в любой момент времени датасетов является ограниченным и устанавливается предварительно. Внутри каждого датасета формирование задач ведётся по так называемым чанкам (chunks) — последовательным наборам файлов, размер которых заранее предопределен и зависит в дальнейшем от приоритета задания, которое может меняться со стороны **WFMS** (Рис. 8).

После обработки данных, отвечающих за конкретный *чанк* данных, сначала происходит обновления приоритета датасета, затем переход к следующему участку данных. После составления описания задачи по полученной информации о содержимом датасета (поступившим с **dsm-manager**), данная задача отправляется в очередь сообщений на **job-manager**.

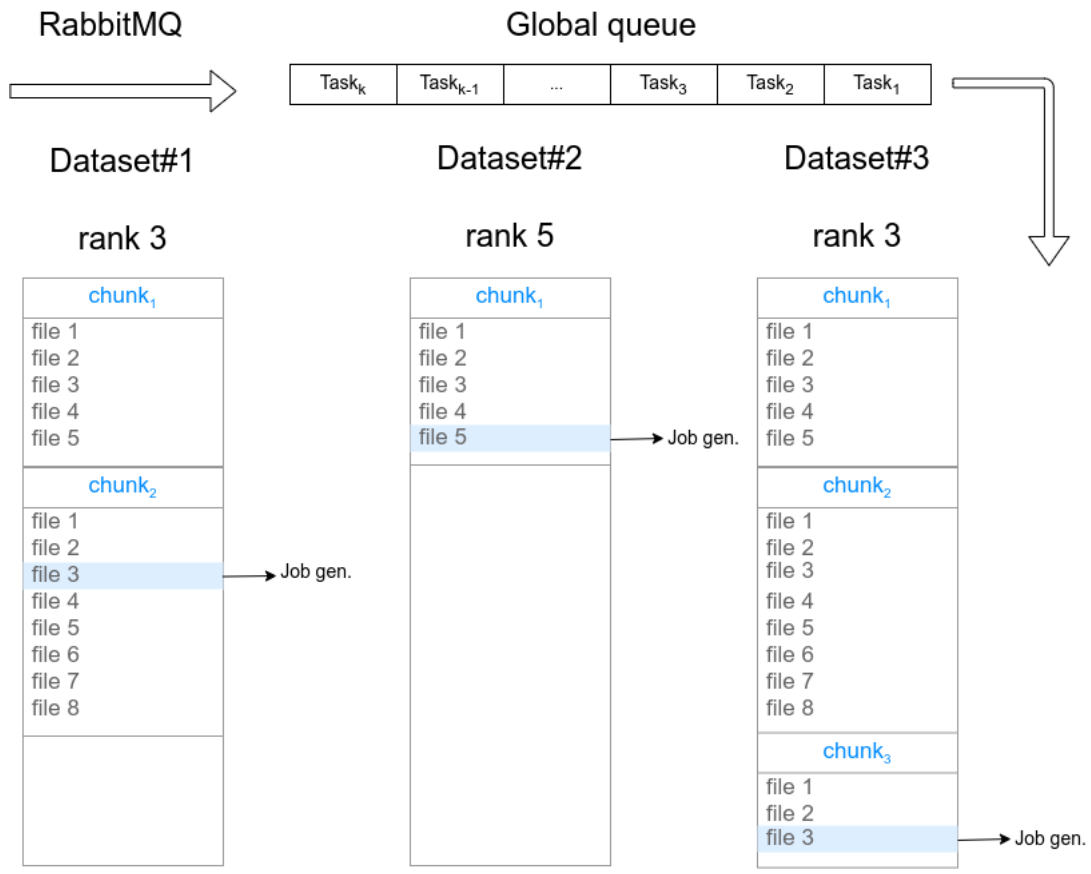


Рис. 8 Механизм асинхронной генерации задач.

Алгоритм должен гарантировать обработку каждого датасета в пропорции, которая будет зависить от *приоритетов* заданий. Таким образом, задачи, которые соответствуют датасетам с более высоким приоритетом, будут формироваться чаще, нежели чем другие, что делает данный алгоритм распределения одним из вариантов алгоритмов *долгосрочного планирования*.

2.4.1 Анализ асинхронной реализации

Для того, чтобы предступить к разработке алгоритма, удовлетворяющего наложенного поверх него требованиям, требуется проанализировать возможность его реализации стандартными методами. Реализация асинхронности во многих языках основана на общих принципах. В качестве примера рассмотрим, как это происходит в языке Python.

Асинхронное программирование в Python основано на концепции сопрограмм (*coroutines*), которые представляют собой функции, которые могут быть приостановлены и возобновлены в определенные моменты. Модуль **asyncio** в Python предоставляет инструменты для создания и запуска асинхронных программ с использованием сопрограмм.

Асинхронность в Python реализована по принципу переключение контекста. Когда выполняется асинхронная программа, она обычно состоит из одного потока управления, который выполняет несколько сопрограмм [17]. Модуль **asyncio** управляет выполнением этих сопрограмм с помощью цикла обработки событий. Цикл обработки событий отвечает за переключение между сопрограммами, то, когда они приостанавливаются и возобновляются. Когда сопрограмма, например, вызывает операцию ввода-вывода (будет использоваться дальше в качестве примера), которая блокируется, она приостанавливает ее выполнение и возвращает управление циклу обработки событий. Затем цикл обработки событий возобновляет другую сопрограмму, которая готова к выполнению.

Цикл обработки событий использует *очередь событий* для управления тем, какие сопрограммы готовы к выполнению, а какой сопрограмме необходимо дождаться завершения операции ввода-вывода. Когда операция ввода-вывода завершается, цикл обработки событий повторно планирует со-

программу, которая инициировала операцию ввода-вывода, для продолжения ее выполнения.

Ключевое слово *async* в Python используется для определения функции сопрограммы. Функции сопрограммы аналогичны обычным функциям, но их можно приостанавливать и возобновлять с помощью ключевого слова *await*. Ключевое слово await используется для приостановки выполнения сопрограммы до тех пор, пока не произойдет какое-либо событие, такое как завершение операции ввода-вывода или завершение другой сопрограммы. Когда вызывается функция сопрограммы, она возвращает ожидаемый объект, который представляет саму сопрограмму.

Ожидаемый объект может ожидаться внутри другой сопрограммы, что организует цепочку выполнения сопрограмм, что может добавить дополнительной вложенности для асинхронности, когда необходимо реализовать довольно сложную логику.

```
import asyncio
import time

async def count():
    print("Начало сна")
    await asyncio.sleep(1)
    print("Конец сна")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"Исполнение программы заняло {elapsed:0.2f} секунд.")
```

Рис. 9 Пример работы асинхронной программы в Python.

На Рис.9 преведён пример работы асинхронной программы. Создается четыре асинхронные сопрограммы: три count() и main(). Все три сопрограммы count() ’спят’ в течение одной секунды, main() создает задачи для трёх сопрограмм и ожидает их завершения.

Когда мы запускаем сопрограмму main() с помощью asyncio.run(), она выполняется синхронно до тех пор, пока не достигнет первого оператора await, который затем позволяет другим сопрограммам выполняться тем временем. Каждый вызовов сопрограммы count() представляет собой единый цикл обработки событий. Когда каждая задача достигает await asyncio.sleep(1), сопрограмма передаёт управление в цикл обработки событий и становится в очередь.

```
$ python3 async-example.py
Начало сна
Начало сна
Начало сна
Конец сна
Конец сна
Конец сна
Исполнение программы заняло 1.01 секунд
```

Рис. 10 Результат работы async-example.py.

Как только обе задачи будут выполнены, функция main() завершит работу и наша программа завершит работу. Можно видеть, что общее время работы составило одну секунду (Рис. 10), где в случае синхронного вызова сопрограмм, общее время бы составило около трех секунд. Это хорошо иллюстрирует, как асинхронное программирование можно использовать для одновременного выполнения нескольких задач. Однако, этого сделать *принципиально невозможно* при реализации алгоритма распределения заданий.

Как было сказано выше, очередь событий, отвечающая за управление сопрограммами, организована по принципу FIFO (First In First Out), что никоем образом не учитывает в какой пропорции должно отдаваться процессорное время для обработки каждого датасета. Таким образом необходим принципиально *новый подход* для реализации данного алгоритма.

2.4.2 Обобщенный алгоритм совместного использований процессов (GPS)

Обобщенный алгоритм совместного использования процессоров (GPS) – алгоритм планирования, используемый компьютерными операционными системами, пакетными сетями и телекоммуникационными сетями. Его основная цель – обеспечить справедливое и эффективное распределение ресурсов между несколькими пользователями или приложениями [18]. GPS относится к классу алгоритмов пропорциональной справедливости, которые обеспечивают пропорциональное распределение ресурсов между различными пользователями.

Основная идея GPS заключается в том, что каждому пользователю присваивается вес, и ресурсы распределяются между пользователями пропорционально их соответствующему весу. В GPS веса используются для представления относительной важности различных пользователей или приложений. Чем выше вес, присвоенный пользователю, тем больше ресурсов он получит.

В соответствии с GPS, если двум пользователям присвоен одинаковый вес, то они получат равную долю ресурсов. Однако, если одному пользователю присвоен более высокий вес, то он получит большую долю ресурсов. Такой подход обеспечивает детерминированное и справедливое распределение ресурсов между пользователями.

GPS работает путем разделения ресурсов на небольшие временные интервалы и выделения по одному интервалу каждому пользователю в заранее определенном порядке на основе их веса. В течение каждого временного интервала пользователю с наивысшим приоритетом (т.е. пользователю с наибольшим весом) предоставляется доступ к ресурсам. Если два или более пользователя имеют одинаковый вес, ресурсы делятся между ними поровну. Этот процесс повторяется для каждого временного интервала до тех пор, пока все пользователи не получат свою выделенную долю ресурсов (Рис. 11).

Рассмотрим планировщик, работающий поверх N потоков (сессий), каждому из которых назначен свой вес r_i. Тогда, GPS гарантурует, что, рассматривая на некотором временном интервале $(s, t]$ некоторый поток i, так что соответствующая ему очередь на этом интервале никогда не бывает пу-

стой, то для любого другого потока j справедливо:

$$r_j Q_i(s,t) \geq r_i Q_j(s,t) \tag{1}$$

где $Q_i(s,t)$ определяется как количество бит i-ого потока на полуинтервале $(s,t]$.

Тогда может быть показано, что каждый поток i будет иметь как минимум пропускную способность

$$R_i = \frac{r_i}{\sum_{j=1}^{N} r_j} R \tag{2}$$

где R есть пропускная способность всего сервера.

Если какой-либо поток не использует свою полосу пропускания в течение некоторого периода, эта оставшаяся пропускная способность распределяется между активными потоками [19].

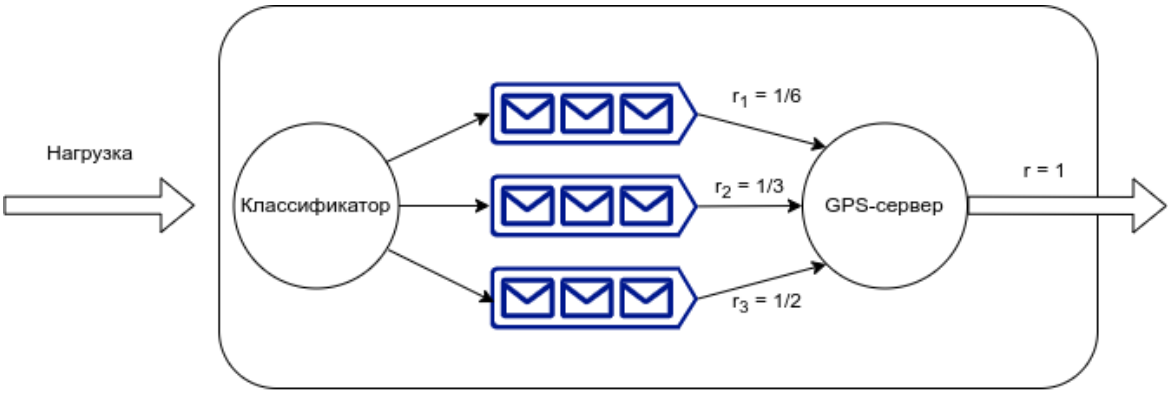


Рис. 11 Работа GPS-планировщика на примере передачи сетевых пакетов.

Одним из главных преимуществ GPS является то, что он обеспечивает гарантированную минимальную пропускную способность для каждого пользователя. Другими словами, каждому пользователю гарантируется минимальное количество ресурсов, даже если другие пользователи в данный момент используют больше ресурсов.

Существует очень занимательная двойственность, которая заключается в том, что планирование задач или процессов может быть выполнено аналогично планированию пакетов: при рассмотрении набора n активных задач, каждая из них τ_i получает w_i квант процессорного времени [20; 21].

То есть GPS также может быть расширен для поддержки концепции весов для заданий, точно так же, как пакетная версия алгоритма 1. В этом случае вес или размер задания пропорционален количеству процессорного времени, требуемого для его выполнения. Каждому заданию присваивается вес, отражающий его приоритет или потребность в ресурсах. Например, если есть две задачи с весами 2 и 3, а процессорное время составляет 10 тиков, то в *идеальном случае* [2] первая задача получит 4 единицы, а вторая задача получит 6 единиц процессорного времени.

Таким образом, можно сказать, что заданию с большим весом будет присвоен пропорционально больший виртуальный временной интервал, чем заданию с меньшим весом. Случай с алгоритмом распределения заданий аналогичный: *чанки* выступают в роли очередей, а задачи в роли пакетов.

В теории сетевой инженерии выделяют метрики “справедливости” (*Fairness measure*), для определения того, получают ли пользователи или приложения справедливую долю системных ресурсов [22]. Одной из такой метрик является метрика Джейна:

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} = \frac{\overline{\mathbf{x}}^2}{\overline{\mathbf{x}^2}} = \frac{1}{1 + \widehat{c_{\mathrm{v}}}^2}$$

Существуют несколько популярных GPS алгоритмов, имеющие хорошие метрики "справедливости": Weighted Fair Queuing (WFQ) - алгоритм взвешенной справедливой очереди и Weighted round robin (WRR) - алгоритм взвешенного циклического перебора. О них и пойдет разговор дальше.

[2] *GPS является теоретически идеальным алгоритмом планирования*

2.4.3 Взвешенная справедливая очередь (WFQ)

WFQ – алгоритм планирования, который распределяет полосу пропускания между различными потоками данных в сети пропорциональным образом на основе их весов [23]. WFQ является расширением алгоритма честных очередей (Fair queuing), который изначально разработан для совместного использования ограниченного ресурса, например, для предотвращения того, чтобы потоки с большими пакетами или процессы, генерирующие небольшие задания, потребляли больше пропускной способности или процессорного времени, чем другие потоки или процессы.

WFQ обрабатывает n потоков, каждому из которых соответствует некоторый вес w_i (Рис. 12). Тогда, i-ый поток достигнет такое же пропускной способности как в уравнении 2. Для каждого пакета будет вычислено виртуальное теоретическое время отправления, так, будто бы если планировщик был идеальным GPS-планировщиком. Затем, каждый раз, когда выходной канал простаивает, для отправки выбирается пакет с наименьшей датой.

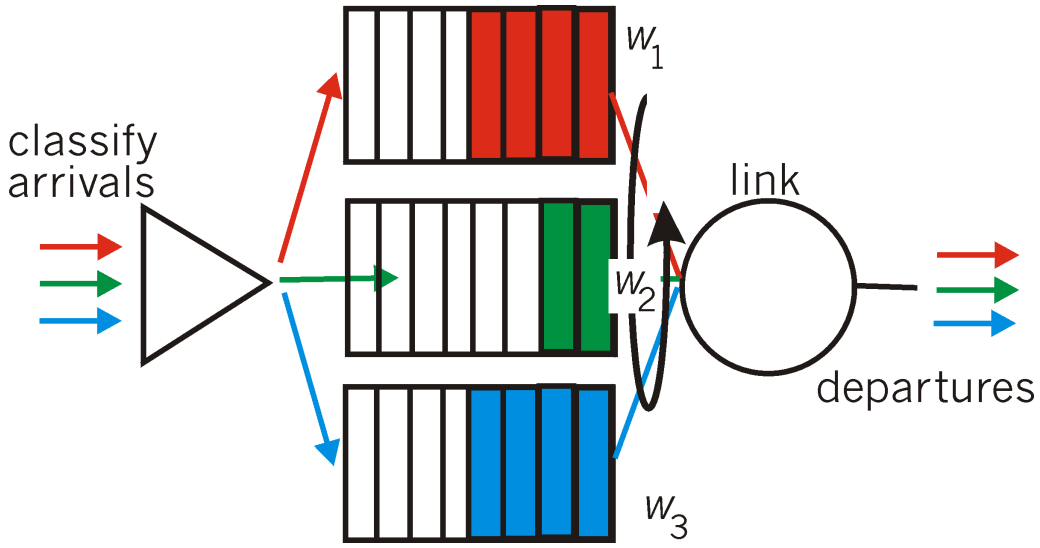


Рис. 12 Алгоритм WFQ.

Было доказано, что алгоритм WFQ отлично приближает GPS с точностью до одного пакета, независимо от их схемы прибытия [24]. Алгоритмическая сложность оценивается как $O(log(n))$, где n - количество потоков. Такова сложность берётся из-за того, что на каждом шаге выбирается поток с наименьшим виртуального временем отправки пакета, что осуществляется путем

введения структур данных с поддержкой минимума на дереве за $O(log(n))$, например, двоичная куча, AVL дерево или красно-черное дерево.

По сравнению с другими алгоритмами, стоит отметить, что WFQ имеет более сложную реализацию и конфигурацию, а также требует дополнительной памяти. И все же, несмотря на то, что данный алгоритм планирования зарекомендовал себя как один из лучших приближений GPS, он оказывается вовсе непригодным для решаемой задачи.

Причиной, по которой данный алгоритм оказывается вовсе неприменимым в контексте генерации задач внутри *чанков*, является предварительное вычисление виртуального времени для **каждой** задачи. То есть необходимо организовать предпосчет времени для задач заранее, что является абсолютно недопустимым из-за сопутствующей этому процессу дополнительной сложности в $O(n\log(n))$. Такое оценка получается из-за того, что в WFQ, на каждый принятый пакет, происходит обновление вирутального времени за $O(log(n))$.

2.4.4 Взвешенный циклический перебор (WRR)

WRR (Weighted round robin) является обобщением алгоритма *round-robin*. WRR – это сетевой алгоритм планирования, который присваивает веса каждому потоку в сети и обслуживает каждый поток в соответствии с его весом (Рис. 13). Алгоритм обслуживает каждый поток по очереди и выдает фиксированное количество данных, прежде чем переходить к следующему. Известно, что алгоритм также может быть использован для планирования заданий аналогичным образом [25].

Алгоритм состоит из следующего набора шагов:

1. Определяется общее количество очередей n. Каждой очереди q_i присваивается определенный вес w_i.

2. На каждом шаге цикла, пока очереди не пустые, определяется количество передаваемых пакетов. Для очереди q_i данное число равно равно его весу w_i.

3. Если очередь не пустая и количество переданных пакетов меньше w_i, передается следующий пакет.

```
function WRR(N, w, q)
    while True do
        dispatch = 0
        for i = 1 to N do
            while !q[i].empty() and dispatch < w[i] do
                send(q[i].head())
                q[i].dequeue()
                dispatch = dispatch + 1
            end while
        end for
    end while
end function
```

Рис. 13 Псевдокод классического алгоритма WRR.

WRR имеет простую реализацию, его легко внедрять, и, как и round robin, не приводит к ресурсному "голоданию". При условии, что все пакеты

имеют одинаковый размер, WRR является хорошим приближением обобщенного алгоритма совместного использований процессов (GPS). Так как при распределении заданий, этап с формирмированием задач аналогичен отправке сетевого пакета, WRR является хорошей с первого взгляда аппроксимацией (Рис. 14).

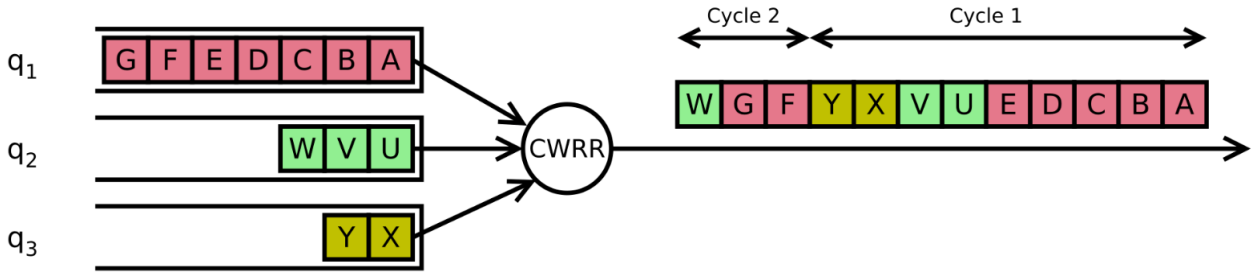


Рис. 14 Пример работы алгоритма WRR.

И как можно заключить, алгоритм в целом удовтлетворяет поставленным требованиям, однако имеет ряд существенных недостатков, из-за которых его невозможно взять за основу генерации задач:

- Процесс обработки датасетов сильно зависит от порядка их прибытия в глобальную очередь.
- В случае больших приоритетов, обработка датасетов вырождается в синхронный случай.

2.4.5 Алгоритм распределения на основе IWRR

IWRR (Interleaved Weighted round robin) – это расширение WRR, которое чередует пакеты из разных очередей в каждом цикле, а не обрабатывает все пакеты из одной очереди за раз (Рис 15) [26]. В IWRR пакеты каждого потока чередуются между несколькими циклами. Это гарантирует, что пакеты из прерывистого потока не будут доминировать в полосе пропускания в течение одного цикла, как это может происходить в WRR [27]. Последнее как раз и является причиной, по которой не является возможным выбрать WRR для распределения заданий в нашем случае. Высокие приоритеты заданий, либо приоритеты близкие к размерам *чанка*, резко уменьшают метрику справедливости (fairness measure).

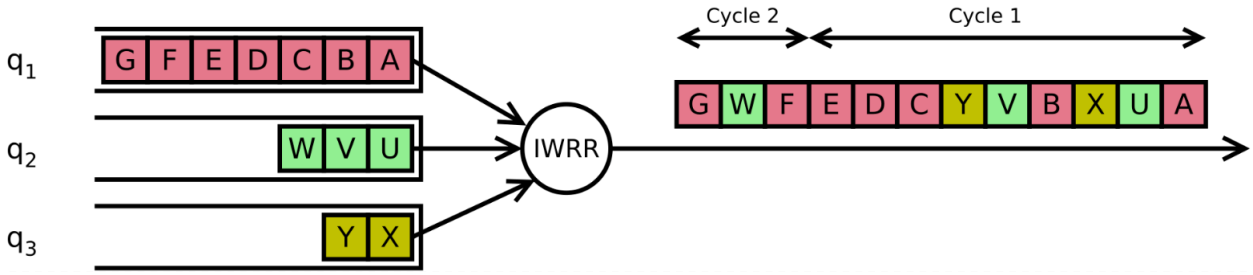


Рис. 15 Пример работы алгоритма IWRR.

Ключевое отличие от WRR состоит в следующем. Находится максимальный w_{max} среди всех очередей (потоков), затем каждый цикл разбивается на w_{max} раундов. На каждом раунде r может быть быть отправлен пакет из i-ой очереди только в том случае, если $r \leq w_i$.

По аналогии с WRR, в случае одинакового размера пакетов IWRR является приближением алгоритма совместного использований процессов (GPS). Ввиду эквивалетности формулировок пакетов и задач в алгоритмах GPS, можно заметить, что задачи (jobs) так же являются однородными по своей структуре (как и файлы). Таким образом, это позволяет воспользоваться IWRR для реализации алгоритма распределения (Рис. 16).

Переменные:
$global\_queue$ – *глобальная очередь с заданиями*
$dataset$ – *массив датасетов*
N – *количество датасетов*
$rank\_max$ – *максимальный приоритет заданий*
$heap$ – *бинарная куча, хранящая максимальные приоритеты заданий*
$rank$ – *массив с приоритетами заданий*

Алгоритм:
$initilize\_datasets(dataset)$
$build\_heap(rank)$
while $true$ **do**
  $rank\_max = heap.top()$
  for $r = 1$ to $rank\_max$ **do**
    for $i = 1$ to N **do**
      if not $dataset[i] \to chunk.empty()$ **and** $rank[i] \geq r$ **then**
        $await \quad dataset[i] \to chunk \to cur\_item$
        $update(dataset[i] \to chunk \to cur\_item)$
      else if $dataset[i] \to chunk.empty()$ **then**
        if $dataset[i] \to chunk \to cur\_item$ **then**
          $dataset[i] = global\_queue.head()$
        end if
        $update(rank[i])$
        $update(heap)$
      end if
    end for
  end for
end while

Рис. 16 Псевдокод алгоритма распределения заданий.

Перед выполнением основной части алгоритма происходит предварительная инициализация датасетов, после чего строится бинарная куча по массиву приоритетов заданий. Бинарная куча необходима для поддержки обновления приоритетов и быстрого нахождения нового максимума.

После чего начинается основной участок кода, который по структуре очень похож на IWRR. Добавляется логика итерации по *чанкам* и асинхронный вызов сопрограммы, который будет создавать задачу по файлу и публиковать в очередь сообщений.

Дойдя до конца *чанка*, происходит переход к новому участку, либо же завершение обработки всего датасета. В последнем случае необходимо из глобальной очереди заданий взять новую и обновить локальный структуры данных.

Массив датасетов представляет из себя итеративную структуру данных, где *chunk* является указателем на текущую область памяти.

Бинарная куча в таком формате является самым узким местом в алгоритме, посколько в конце каждого *чанка* происходит обновление структуры данных, что происходит за время за $O(\log(N))$. То есть, чем меньше размер *чанков*, тем чаще происходит обновление кучи (*вызов процедуры 'просеивание вниз', если быть точным*), что в худше случае вырождается в $O(M \log(N))$, где M - размер датасета, а N их количество.

Изменить это можно, если для каждого датасета/задания менять локально его текущий максимум за $O(1)$ и хранить для каждого результат в отдельном массиве. Строя бинарную кучу по этому массиву, позволит выбирать максимальный приоритет всегда за $O(\log(N))$.

2.5 Модель базы данных

2.5.1 Описание и схема БД

Было принято решение, что будет использована реляционная модель, а в качестве СУБД будет выбрана *PostgreSQL*. Также, нам необходимо следовать принципу *"одна база данных на один микросервис"*, то есть у каждой "микрослужбы"есть собственное хранилище данных, к которому другие микросервисы не могут получить прямой доступ. Таким образом избегаются "жесткие"связи между службами, ограничивается область изменений и сохраняется гибкость независимых развертываний. Одна из возникших проблем, это синхронизировать и обмениваться данными между микросервисами, которым необходим один и тот же доступ к таблице. Существуют несколько подходов:

- **Событийно-ориентированная архитектура (Event-driven architecture)**
  - При таком подходе каждый микросервис публикует события в брокере сообщений при обновлении своего хранилища данных. Другие микросервисы могут подписываться на эти события и соответствующим образом обновлять свои собственные хранилища данных. Таким образом можно достичь окончательной согласованности (**eventual consistency**), которая означает, что хранилища данных со временем придут к одному и тому же состоянию. В рамках данной задачи, такая степень согласованности между **TaskManager** и **JobManager** не является целесообразной.
- **Захват изменения данных (Change data capture)**
  - При таком подходе каждый миросервис использует инструмент, который отслеживает изменения в своем хранилище данных и передает их на центральную узел. Другие микросервисы отслеживают эти изменения и соответствующим образом обновляют свои собственные хранилища данных. Таким образом, можно добиться

синхронизации хранилищ данных практически в режиме реального времени. Однако для этого подхода требуется дополнительная инфраструктура и конфигурация для инструмента или библиотеки сбора измененных данных, а также обработка изменений схемы и конфликтов на центральной узле.

- **API composition**
  - Каждый микросервис предоставляет API, который позволяет другим запрашивать содержимое его хранилища данных. Другие микросервисы могут использовать эти API для составления ответов, включающих данные из нескольких хранилищ. В данном случае достигается согласованность по запросу (**on-demand**), что означает, что данные непротиворечивы в момент запроса. Однако этот подход требует строгой формализации API и их контрактов, а также обработки задержек и сбоев в сетевых вызовах. Для рассматриваемой задачи это решение подходит лучше всего на этапе прототипирования.

При разработке модели, учитывая требования к системе управления нагрузкой, было принято решение ввести следующие *отношения*:

- **JOB_DAT** - каталог с текущими задачами, обрабатываемые в системе
- **TASK_DAT** - каталог с заданиями, поступившие в систему из **WFMS**
- **PILOT_DAT** - пилоты и их текущие состояния
- **FILE_DAT** - каталог с указанием выходных файлов и логов, образованных на пилоте

2.5.2 Модель данных

Концептуальная схема базы данных представлена на (Рис. 17). В системе будет храниться две копии таблиц **TASK_DAT** в **TaskManager** и **JobManager**, синхронизация между которыми будет осуществляться по упомянутому

выше подходу **API-декомпозиции**. Статус задач (Jobs) фигурирует непосредственно только после момента их появления в системе **JobManager** и описывает текущее состояние выполнения задачи на пилоте, не путать с жизненным циклом задач.

Таблица 8 Принимаемые значения статуса задачи

Статус	**Описание**
Running	Задача обрабатывает входные данные
Finished	Задача была успешно завершена
Killed	Задание была намерено прервано
Failed	Задача была неуспешно завершена на пилоте во время работы исполняемого файла
Starting	Задача работает над перемещением входных данных - передача данных из хранилища на рабочий диск, подключенный к вычислительному ресурсу
Cancelled	Задача была отменена “вручную”

Таблица 9 Принимаемые значения статуса файла

Статус	**Описание**
Failed	Отсутствует выходной файл (по разным причинам)
Finished	Был получен *ненулевой* результат

2.5.3 Индексы

Ввиду того, что к таблице **JOB_DAT** происходит потенциально много запросов вида **SELECT** для создания сводки о текущим состоянии задач для **WFMS** либо получения следующей “готовой” задачи, может возникнуть потребность в создании **B-tree** индекса (в виду специфики запросов) [28]. Аналогичная ситуация возникает при принятии решения о закрытии датасета, когда перед **job-executor** стоит задача о получении списка всех завершенных задач. Также к таблице применяются операторы **INSERT** и **UPDATE**, что, как известно, занимает больше времени для таблиц, имеющих индексы. Причина в том, что при выполнении **INSERT** или **UPDATE** база данных также должна вставлять или обновлять значения индекса. *Необходимо провести предварительный анализ в необходимости введения B-tree индекса.*

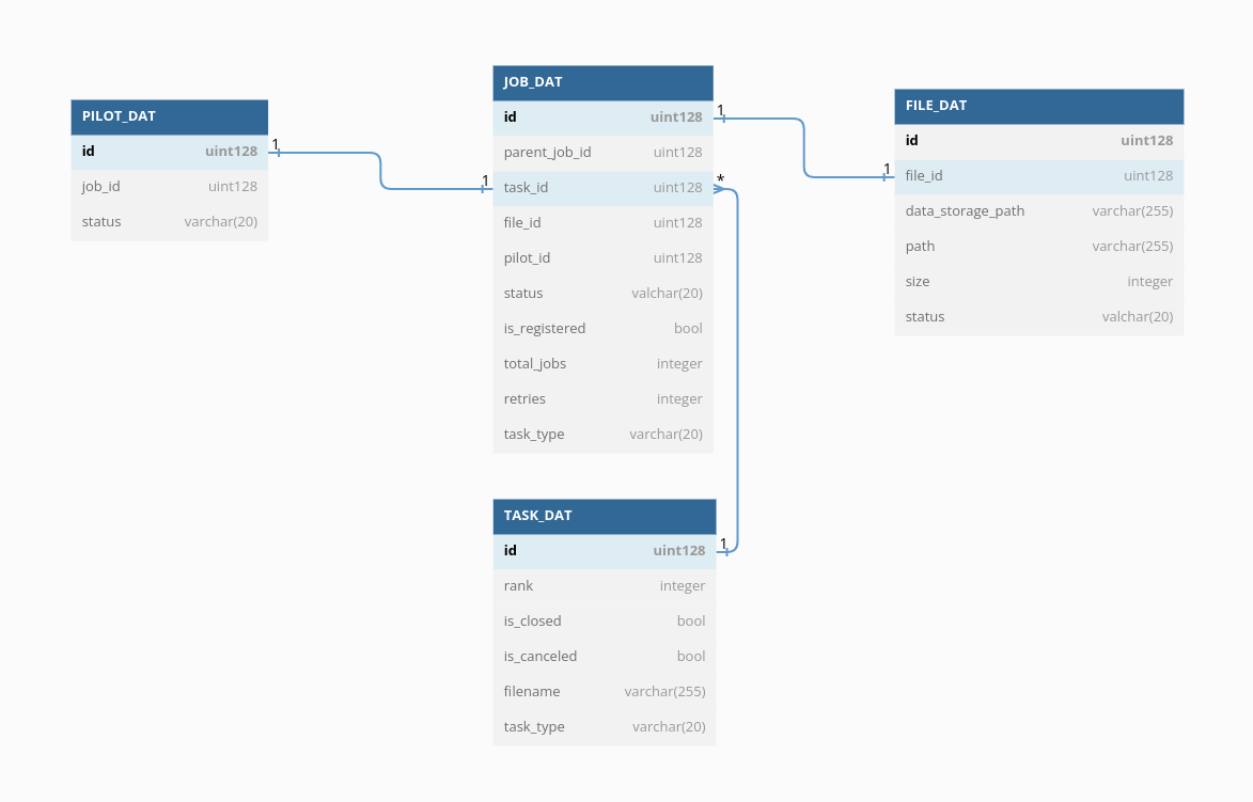


Рис. 17 Концептуальная схема базы данных.

Глава 3. Прототипирование

3.1 Разработка

Прототипирование при разработке программного обеспечения есть процесс создания ранней, неполной версии программного продукта. Она включает в себя построение упрощенной модели программной системы, которая демонстрирует основные функциональные возможности продукта. Прототипирование зарекомендовало себя как эффективный метод снижения затрат на разработку и предоставления возможностей для обратной связи и тестирования, что помогает разработчикам создавать более совершенные программные продукты, которые планируется использовать в течение долгого времени.

Прототип может быть как горизонтальным, так и вертикальным [29]. Горизонтальный прототип предназначен для демонстрации пользовательского интерфейса приложения, в то время как вертикальный прототип предназначен для имитации функциональности, а также возможностей обработки данных приложения. В данной работе, как будет видно дальше, затрагивается в определенной мере прототипирование обоих типов. Пройдены следующие этапы создания прототипа, такие как планирование, проектирование и сборка.

В качестве осного языка при разработки был выбран язык Python. В виду наличия динамической типизации и высокоуровневых абстракцих, делают его хорошим выбором для быстрого прототипирования и гибкой разработки. Также, существует большое количество доступных ресурсов, включая библиотеки, модули и фреймворки, многие из которых написанных на других языках. Python также в силе взаимодействовать со многими другими языками программирования, включая C, C++ и FORTRAN, что упрощает интеграцию системы с другими программными средствами.

Репозитории с кодом системы **«SPD On-Line filter» – «Visor»** расположены на внутреннем сервере МЛИТ ОИЯИ как **Gitlab** проект (Рис. 18), представляющий систему управления репозиториями кода для системы контроля версий **Git**.

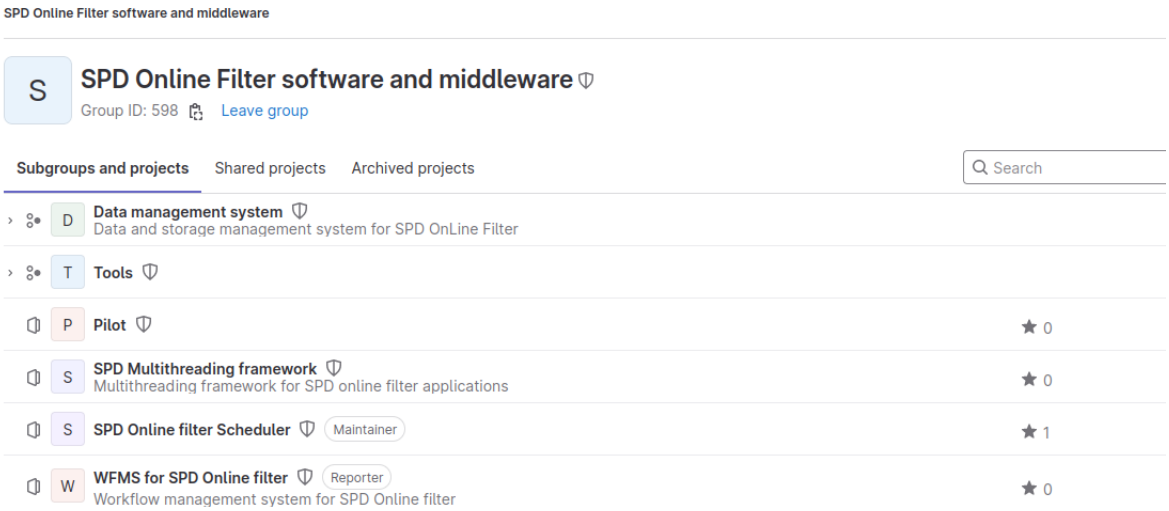


Рис. 18 Репозитории проекта с исходным кодом.

3.2 Выбор технологического стека

В основе выбора следующих фреймворков и инструментов лежало несколько ключевых принципов выбор одних их последних решений, имеющих кодовую базу и рекомендации, которые упрощают процесс разработки и обеспечивают согласованность и логичность кода (*Fast-API, aiohttp*); использовать библиотеки повторно используемого кода, которые могут ускорить разработку и уменьшить количество ошибок (*aio-pika, FastAPI-SQLAlchemy*); облегчать автоматизацию, тестирование, отладку и развертывание программных приложений (*Docker, docker-compose*); набор инструментов и методов для реализации ограничений доступа, проверки входных данных, взаимодействия с другими системами (*pydantic, alembic*). Ниже перечислены используемые инструменты:

- **Python 3.11**
  - Python 3.11 выбран как последняя стабильная версия Python, в среднем быстрее на 10-60% чем Python 3.10 благодаря новому специализированному адаптивному интерпретатору, который оптимизирует выполнение распространенных шаблонов кода.
- **Docker + Docker Compose**

  - Docker был выбран в качестве программной платформы, которая позволяет создавать, запускать приложения и предоставлять к ним общий доступ с помощью контейнеров. Docker Compose позволяет определять и запускать несколько контейнеров как согласованное приложение, предоставляет команды для управления жизненным циклом, такие как масштабирование, обновление и тестирование.

- **FastAPI + Pydantic**

  - Современный веб-фреймворк, поддерживает асинхронное программирование и работу на *ASGI* серверах (unicorn по умолчанию). FastAPI использует Pydantic для автоматической проверки, сериализации и десериализации данных, также генерирует интерактивную документацию для API с использованием OpenAPI и Swagger UI.

- **SQLAlchemy**

  - Библиотека, реализующая принцип *ORM* с реляционными базами данных, в нашем случае PostgreSQL, используя язык Python. Позволяет использовать классы и выражения Python для определения таблиц базы данных и запросов, вместо написания реальных SQL запросов.

- **FastAPI-SQLAlchemy**

  - FastAPI-SQLAlchemy - модуль, который позволяет легко использовать Fast API и SQLAlchemy вместе, предоставляет удобные утилиты, которые помогают выполнять часто встречающиеся задачи.

- **uvicorn**

  - Быстрая и легковесная реализация ASGI веб-сервера для Python, где ASGI - клиент-серверный протокол, являющийся стандартом для создания асинхронных веб-приложений на Python.

- **pika**

  - Python библиотека для работы с RabbitMQ, реализуюшая протокол AMQP 0-9-1.

- **aio-pika**

  - Библиотека для асинхронной работы с RabbitMQ, которая также оказывается очень удобнной в виду объектно-ориентированного API с поддержкой всех основных компонентов RabbitMQ.

- **psycopg**

  - Самый популярный адаптер базы данных PostgreSQL для Python. Его основными особенностями являются полная реализация спецификации Python DB API 2.0 и потокобезопасность.

- **aiohttp**

  - Легковесный асинхронный HTTP-клиент/сервер для asyncio и Python, выполняющий несколько запросов одновременно, не блокируя основной поток.

- **asyncpg**

  - Библиотека для асинхронного взаимодействия с базой данных PostgreSQL с использованием Python/asyncio.

- **alembic**

  - Инструмент миграции баз данных для Python, предназначенный для работы с SQLAlchemy, позволяет изменять структуру базы данных (*таблицы, столбцы и т.д.*) через скрипты миграции, которые описывают изменения, а также как их применять и отменять. Оказывается очень полезным в ситуации, когда необходимо автоматически генерировать сценарии миграции, сравнивая текущую схему базы данных с моделями данных, описанными с помощью SQLAlchemy.

3.3 RabbitMQ

RabbitMQ — это распределенный брокер сообщений с открытым исходным кодом, который обеспечивает эффективную доставку сообщений в сложных сценариях маршрутизации. Он называется «распределенным», потому что RabbitMQ обычно работает как кластер узлов, где очереди распределяются по узлам — реплицируются для обеспечения высокой доступности и отказоустойчивости. Разработчики используют RabbitMQ для обработки высокопроизводительных фоновых заданий (батчи, интенсивные задачи обработки и длительные процессы - workflow), а также для интеграции и взаимодействия между приложениями и внутри них.

3.3.1 Выбор брокера

Очень частый вопрос при выборе брокера сообщений заключается в том, какой является наиболее подходящим для конкретной задачи. Apache Kafka на сегодняшний день является одним из наиболее узнаваемых распределенных программных брокеров. Kafka лучше всего использовать для потоковой обработки данных, не прибегая к сложной маршрутизации, имея при том максимальную пропускную способность. Она также идеально подходит для поиска событий, потоковой обработки и моделирования изменений в системе в виде последовательности событий [30].

RabbitMQ использует модель “умный” брокер/”глупый” потребитель. Брокер последовательно доставляет сообщения потребителям и отслеживает их статус. Kafka использует модель ”глупого” брокера/”умного” потребителя, она не отслеживает сообщения, которые прочитал каждый пользователь, а сохраняет только непрочитанные сообщения. Потребители же должны следить за своими позициями в логе. Именно данная особенность, а также факт, что RabbitMQ имеет более гибкую политику маршрутизации (*Exchange Type*) и встроенный механизм для работы с приоритетом сообщений, оказалось определяющим фактором в выборе инструмента.

Таблица 10 Ключевые отличия брокеров **Apache Kafka** и **RabbitMQ**

Критерий	**RabbitMQ**	**Kafka**
Производительность	4000-10000 сообщений в секунду	1 млн сообщений в секунду
Сохранение сообщения	По принципу *Acknowledgment*	Задание *политики* хранения (30 дней)
Тип данных	Транзакционный	Операционный
Режим консьюмера	“Умный” брокер/ ”глупый” потребитель	“Глупый” брокер/ ”умный” потребитель
Топология	Тип обмена: • прямой • разветвленный • тематический • на основе заголовка	Издатель-подписчик
Использование	Большинство случаев простого межсервисного взаимодействия	Массивные данные/случаи, требующие высокой пропускной способности

3.4 Прототип task-manager

3.4.1 Структура проекта

Структура кода сервиса **task-manager** организована согласно общепринятым практикам при разработке на языке Python. В дальнейшем, по мере усложнения проекта, планируется добавить модули с исключениями, также ввести отдельный модуль ORM моделями базы данных.

```
task-manager
├── alembic ..... модуль со всеми миграциями
│   ├── versions ..... все миграции, как .py файлы
│   │   └── ⋮
│   ├── env.py ..... инструкции для конфигурации модуля
│   ├── README.md ..... базовая информация о миграционной среде
│   └── script.py.mako
├── gateway ..... модуль взаимодействия с rabbitMQ сервером
│   ├── __init__.py
│   ├── __main__.py ..... скрипт, инициирующий шлюз
│   ├── rabbitmq.py ..... инкапсуляции жизненного цикла rabbitMQ
│   └── ⋮
├── src
│   ├── __init__.py
│   ├── app.py ..... инициализация FastAPI
│   ├── db.py ..... инициализация, установка соединения с БД
│   ├── enums.py ..... инкапсуляция переменных окружения
│   ├── main.py ..... входная точка API-микросервиса
│   ├── models.py .... реализация объектно-реляционного отображения
│   ├── schema.py ..... задание Pydantic схемы
│   └── ⋮
├── .env ..... файл настроек сервиса
├── .gitignore
├── alembic.ini ..... конфигурационный файл для alembic
├── docker-compose.yml ..... конфигурационный файл Docker Compose
└── Dockerfile ..... образ микросервисов шлюза и API
```

3.4.2 Сборка и запуск

Все контейнеры связаны одной сетью Docker Compose (Рис. 19). Для привязки данных к жизненному циклу контейнера используются логические тома Docker Compose - **volumes**. Весь сервис состоит из следующего набора контейнеров:

- **rabbitmqServer**
  - Docker контейнер с брокером сообщений RabbitMQ версии 3.6, в качестве образа был взят готовое официальный образ из *dockerhub* репозитория.
- **pgadmin**
  - Преставляет из себя платформу для администрирования и разработки для PostgreSQL. Был взят самый популярный образ - dpage/pgadmin4. Для работы необходимо "прокинуть" порт 5050 системы-хоста на 80-ый порт внутри контейнера.
- **consumer**
  - Шлюз, для непосредственного принятия, обработки и валидация сообщения, пришедшего из RabbitMQ. Был примонтирован логический том к корню проекта, для тестирования и незамедлительного применения измений в контейнере без его перезапуска. Вход определяется командой *bash -c "python -m gateway"*, что выполняет модуль gateway как скрипт.

- **db**
  - Контейнер, запущенный на основе официального образа базы данных PostgreSQL. Для хранения данных, которые должны сохраняться на хосте после завершения жизненного цикла контейнера, было примонтировал логический том (*volume*) к каталогу **/var/lib/postgresql/data**, где последний есть каталог для хранения файлов данных сервера баз данных PostgreSQL.
- **web**
  - Запускает ASGI веб-сервер uvicorn, выполняет обновление базы данных до последней версии, запустив все последние сценарии Alembic миграций: *alembic upgrade head && uvicorn src.main:app –host 0.0.0.0 –port 8000 –reload*.

```
NAME                     IMAGE                          COMMAND                  SERVICE           CREATED          STATUS
pgadmin                  dpage/pgadmin4                 "/entrypoint.sh"         pgadmin           40 seconds ago   Up 37 seconds
rabbitmqServer           rabbitmq:3.6-management-alpine "docker-entrypoint.s…" rabbitmqServer    40 seconds ago   Up 39 seconds (health: starting)
 25672/tcp, 0.0.0.0:15672->15672/tcp, :::15672->15672/tcp
task-manager-consumer-1  task-manager-consumer          "bash -c 'python -m …"   consumer          40 seconds ago   Up 8 seconds
task-manager-db-1        postgres:12                    "docker-entrypoint.s…"   db                40 seconds ago   Up 39 seconds (healthy)
task-manager-web-1       task-manager-web               "bash -c 'alembic up…"   web               40 seconds ago   Up 34 seconds
```

Рис. 19 Запущенный через Docker Compose микросервисы **task-manager**.

Одной из систем интерактивной документации, которую FastAPI включает по умолчанию, является Swagger UI (Рис. 20). FastAPI автоматически генерирует схему OpenAPI на основе Pydantic схем, описанных в коде. Ниже представлен рисунок с реализованными API. Получить доступ к пользовательскому интерфейсу Swagger UI можно, посетив /docs (*например http://localhost:8000/docs*). Там можно просмотреть все детали API и опробовать их, отправляя запросы и получая ответы от сервиса **task-manager**.

Pydantic схемы в Fast API импортируются из отдельного модуля, после чего задаются аннотации для параметров и возвращаемых значений функций API. Помимо того, что Pydantic играть роль шаблона при автоматической генерации OpenAPI схемы, он также решает следующий ряд ряда задач:

- Валидация данных:
  - FastAPI проверит, соответствуют ли запросы и ответы схемам Pydantic, и выдаст ошибки, если в случае несоответствия.
- Преобразование данных:
  - Fast API автоматически преобразует типы данных запросов и ответов в соответствии со схемами Pydantic (например, из JSON в объекты Python и наоборот).

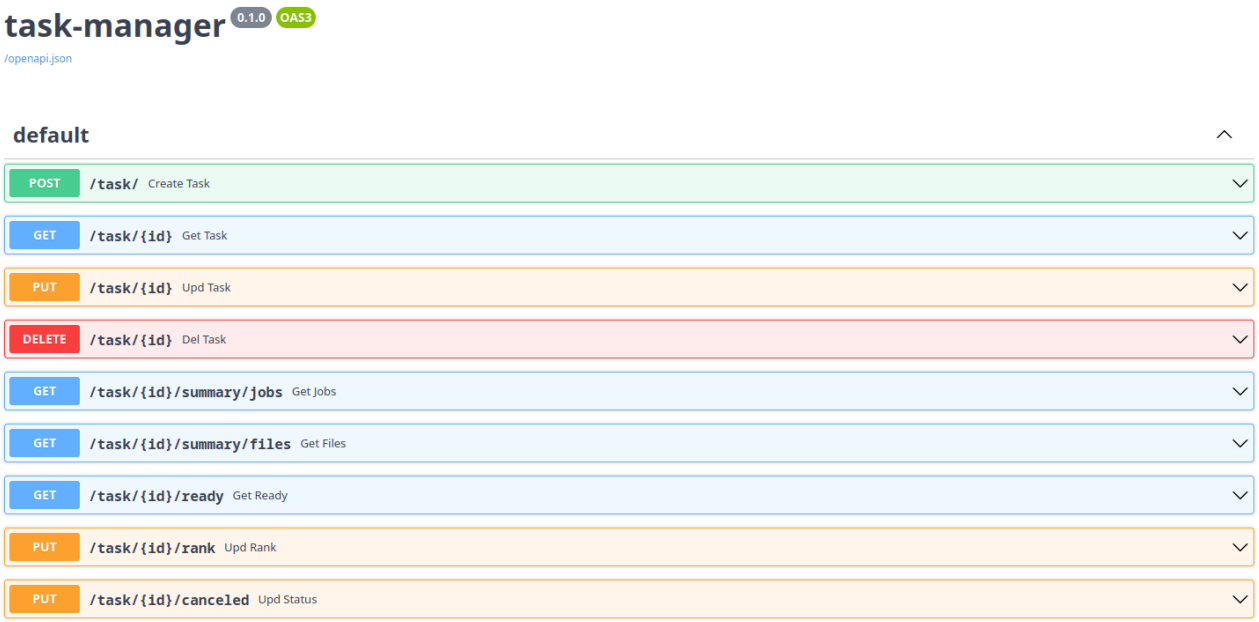


Рис. 20 API сервиса **task-manager**.

Для работы с RabbitMQ сервером на стороне контейнере **consumer** задаётся название очереди, обменник (exchange), ключ маршрутизации, которые определены в Docker Compose файле, которые затем доступны программно как переменные окружения среды. Осуществляется подключение к RabbitMQ серверу, происходит создание канала, объявление очереди и переход в режим ожидания сообщений и их асинхронная обработка при приеме, все средствами **aio-pika**. Ниже представлен пример работы с очередью через RabbitMQ UI (Рис. 21).

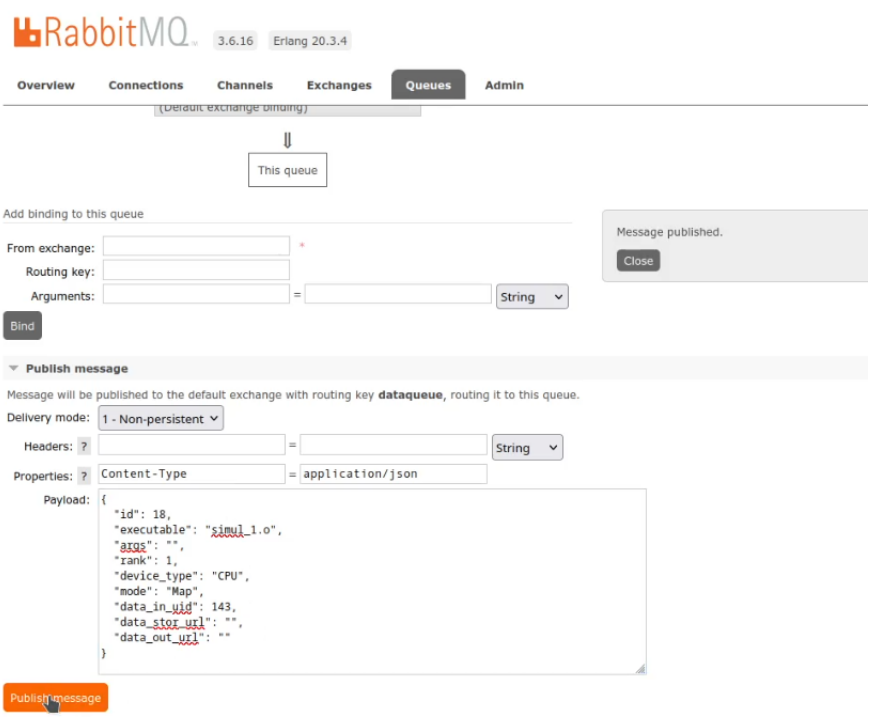


Рис. 21 Отправка сообщения с заданием в **rabbitMQ** очередь.

Внутри **web** контейнера строка подключения к базе данных будет выглядеть как **postgresql://postgres:postgres@db:5432/postgres**, а с хост-компьютера строка подключения будет выглядеть, например, **postgres://localhost:5432**, если контейнер запущен локально.

Графический инструмент администрирования pgadmin облегчает манипулирование схемой и данными в PostgreSQL, запущен в отдельном контейнере **pgadmin** (Рис. 22). Контейнер получает IP-адрес сети определенной в docker compose файле, к которой он подключается, из пула IP-адресов сети, после чего подключение к серверу PostgreSQL ведётся по его имени как Docker Compose сервиса.

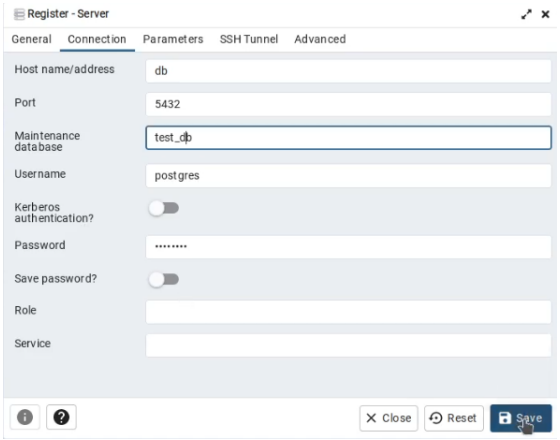
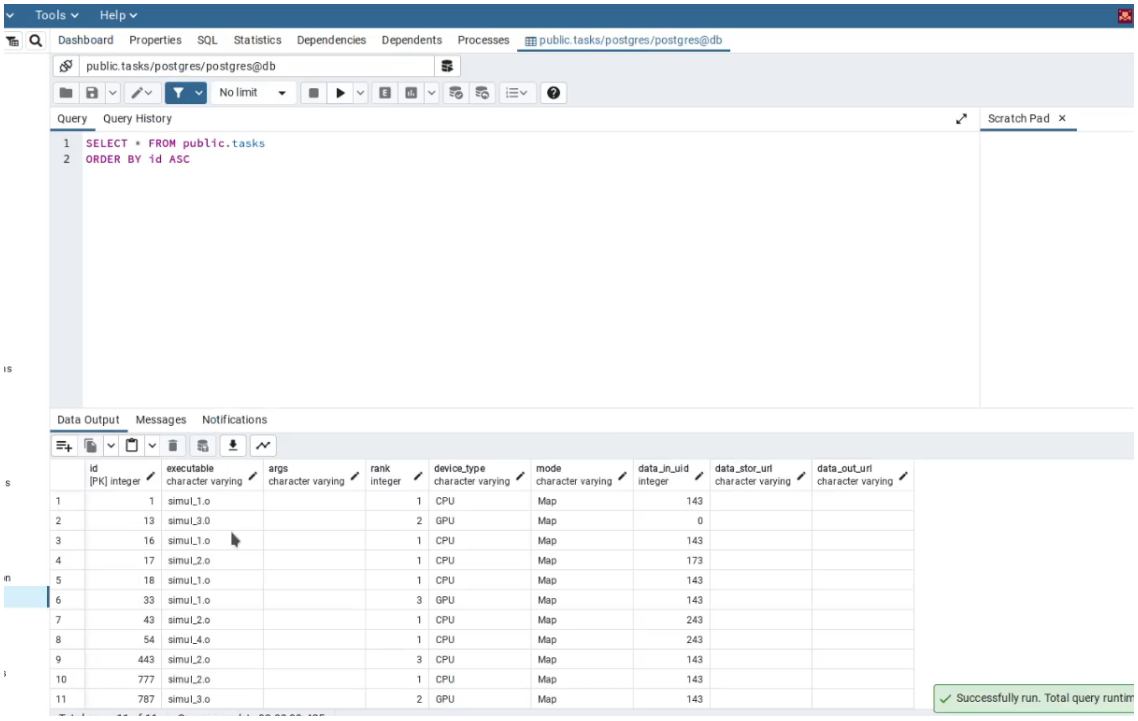


Рис. 22 Подключение к базе данных в клиенте **pgadmin**.

При старте Docker Compose, контейнер, отвечающий за ASGI сервер, обновляет базу данных до последней версии, выполняя все скрипты миграции. Ввиду монтирования логических томов Docker Compose к файлам базы данных, после установления соединения с клиентом pgadmin, можно посмотреть на все сделанные записи в таблице **TASK-DAT** (Рис. 23).



	id [PK] integer	executable character varying	args character varying	rank integer	device_type character varying	mode character varying	data_in_uid integer	data_stor_url character varying	data_out_url character varying
1	1	simul_1.o		1	CPU	Map	143		
2	13	simul_3.0		2	GPU	Map	0		
3	16	simul_1.o		1	CPU	Map	143		
4	17	simul_2.o		1	CPU	Map	173		
5	18	simul_1.o		1	CPU	Map	143		
6	33	simul_1.o		3	GPU	Map	143		
7	43	simul_2.o		1	CPU	Map	243		
8	54	simul_4.o		1	CPU	Map	243		
9	443	simul_2.o		3	CPU	Map	143		
10	777	simul_2.o		1	CPU	Map	143		
11	787	simul_3.o		2	GPU	Map	143		

Рис. 23 Записи с заданиями в таблице **TASK-DAT**.

Пример обращения к сервису со стороны **WFMS** (**Workflow Management System**) и **job-manager** для получения полной информации о задании. Определение шаблона запроса задано через Pydantic схему, в которой указаны через аннотации типы данных для каждого поля (Рис. 24).

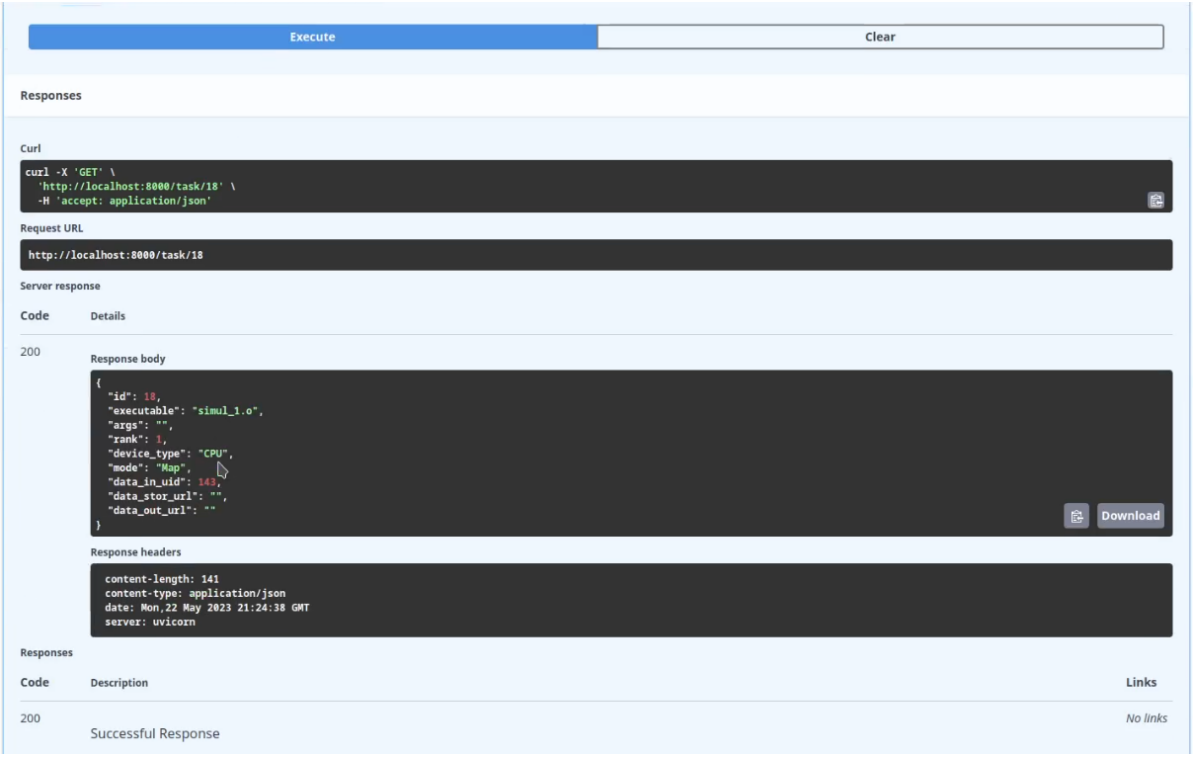


Рис. 24 Результат GET запроса сервиса **task-manager**.

Контейнер-шлюз, принимает и валидирует сообщения из **rabbitMQ**, после чего перенаправляет запрос на веб-сервер, запущенный в контейнере **web** (Рис. 25).

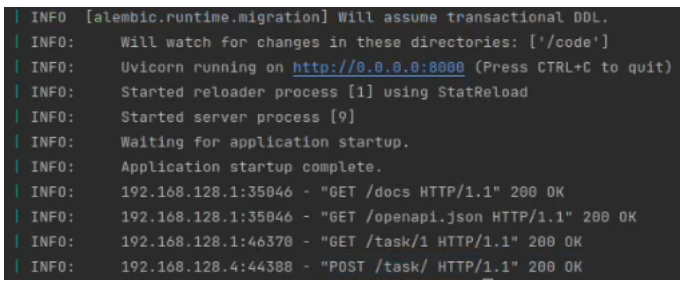


Рис. 25 Внутренние логи контейнера ASGI - сервера с обработанными запросами.

Выводы

Система управления нагрузкой (**WMS**) является центральным связующим компонентом в комплексе промежуточного программного обеспечения «SPD On-Line filter» – «Visor». **WMS** взаимодействут со всеми системами SPD On-Line filter: **WFMS**, **DMS** и агентским приложением, что сказалось на объеме как внешних, так и внутренних интерфейсов взаимодействий, и общей сложности архитектуры системы.

Количество брокеров сообщений в системе управления нагрузкой лишний раз подчеркивает сложность бизнес-логики системы. Возможность выбора более сложных режимов маршрутизации, а также отсутствие необходимости в отслеживании своей позиции в распределенном логе Apache Kafka, привели к выбору RabbitMQ в роли основного брокера. Однако, очереди задач между WMS и пилотом могут содержать сообщений примерно на порядок-два больше, чем другие очереди в системе. Поэтому, в случае если RabbitMQ будет не справляться с возникшей нагрузкой, возможен переход на Apache Kafka и реализация очереди с приоритетом на уровне сервиса job-executor и рассмотрение других алгоритмов планирования.

Приоритеты заданий играют ключевую роль в управлении процессами со стороны **WFMS**. Однако, пока задачи дойдут до сервиса job-executor, они пробывают в очереди сообщений и проходят через другие компонента системы **WMS**. Генерация задач со скоростью, пропорциональной приоритетам соответствующих заданий (tasks), позволяет уменьшить их время простоя, а также увеличить пропускную способность, что и делает представленный алгоритм распределения заданий.

Заключение

В ходе работы был проведён подробный анализ компонентов системы управления нагрузкой, каждый из которых решают одну из задач, описанных в функциональных требованиях к системе. Была произведена разработка архитектуры всей системы управления нагрузкой, учитывая особенности и бизнес-логику таким систем как **WFMS**, **DMS** и агентского приложения, также описаны интерфейсы взаимодействия между ними. Разработан первый прототип сервиса **task-manager**, дальнейшая цель состоит в том, чтобы закончить прототипирование и произвести непосредственно интеграцию с другими системами.

Также, был построен алгоритм распределения заданий и проведён его анализ, была показана невозможность его реализации, ограничиваясь только стандартными асинхронными средствами языка. За основу был взят алгоритм сетевого планирования IWRR (Interleaved Weighted Round Robin), было проведено сравнение с другими популярными GPS алгоритмами и аргументирован выбор именно IWRR.

Список литературы

[1] NICA [Электронный ресурс] // URL: https://nica.jinr.ru/ru/

[2] Эксперимент SPD [Электронный ресурс] // URL: http://spd.jinr.ru/wp-content/uploads/2021/04/SPD_Korzenev_DIS2021.pdf

[3] 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

[4] Conceptual design of the Spin Physics Detector [Электронный ресурс] // URL: https://arxiv.org/abs/2102.00442

[5] Savas, E., Koru, G. (2018). Microservices: architectural advantages and challenges. Journal of Software Engineering Research and Development, 6(1), 1-18.

[6] Oleynik, Danila & Panitkin, Sergey & Turilli, Matteo & Angius, Alessio & Oral, Sarp & De, Kaushik & Klimentov, Alexei & Wells, Jack & Jha, Shantenu. (2017). High-Throughput Computing on High-Performance Platforms: A Case Study. 295-304. 10.1109/eScience.2017.43.

[7] DIRAC [Электронный ресурс] // URL: https://dirac.readthedocs.io/en/latest/

[8] Casajus, A., Graciani, R., Luzzi, C., Marco, J., Rodriguez, M. (2010). DIRAC pilot framework and the DIRAC Workload Management System. Journal of Physics: Conference Series, 219(6), 062049.

[9] RADICAL-Pilot [Электронный ресурс] // URL: http://radical-cybertools.github.io/radical-pilot/index.html

[10] Merzky, Andre & Svirin, Pavlo & Turilli, Matteo. (2019). PanDA and RADICAL-Pilot Integration: Enabling the Pilot Paradigm on HPC Resources. EPJ Web of Conferences. 214. 03057. 10.1051/epjconf/201921403057.

[11] SAGA: A Simple API for Grid Applications. High-level application programming on the Grid [Электронный ресурс] // URL: https://www.researchgate.net/publication/228794840_SAGA_A

[12] PanDA Collaboration. (2015). PanDA: Production and Distributed Analysis System. Journal of Physics: Conference Series, 664(6), 062021. _Simple_API_for_Grid_Applications_High-level_application_programming_on _the_Grid

[13] Amundsen, M. (2019). Microservices: Yesterday, Today, and Tomorrow. IEEE Software, 36(5), 8-12.

[14] Kar, A. (2018). A Comparative Study of Monolithic and Microservices Architecture. International Journal of Engineering Science and Computing, 8(12), 16746-16757.

[15] Hjelle, G. A. (2019). Async IO in Python: A Complete Walkthrough. Real Python. https://realpython.com/async-io-python/

[16] J. S. Turner, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case,"IEEE/ACM Transactions on Networking, vol. 2, no. 2, pp. 137-150, Apr. 1994.

[17] GPS алгоритм [Электронный ресурс] // URL: https://opengl.org.ru/upravlenie-trafikom-i-kachestvo-obsluzhevaniya-v-seti/algoritm-gps.html

[18] Hirsch, P. D. (2002). Task scheduling using improved weighted round robin techniques. In Proceedings of the international conference on parallel and distributed processing techniques and applications (pp. 702-707). The Computer Science Research, Education, and Applications Press.

[19] Beaulieu, A. (2017). Real-Time Operating Systems - Scheduling and Schedulers. AMSE Journal on Modelling in Engineering, 3(1), 26-35.

[20] Demers, A., Keshav, S., Shenker, S. (1989). Weighted fair queuing and its implementation. ACM SIGCOMM Computer Communication Review, 19(4), 33-44. https

[21] Jain, R., Chiu, D. M., Hawe, W. (1984). A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. DEC Research Report TR-301, Digital Equipment Corporation.

[22] Weighted fair queueing algorithm [Электронный ресурс] // URL: https://en.wikipedia.org/wiki/Weighted_fair_queueing

[23] Parekh, A. K., Gallager, R. G. (1993). A generalized processor sharing approach to flow control in integrated services networks: The single-node case. IEEE/ACM Transactions on Networking, 1(3), 344-357.

[24] Katevenis, M., Sidgiropoulos, S., Courcoubetis, C. (1991). Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. IEEE Journal on Selected Areas in Communications, 9(9), 1490-1502.

[25] Widjaja, I., Kim, K., Yeung, D., Mikhael, W. (1995). Fast packet switching with wideband circuitry. IEEE Network, 9(1), 28-37.

[26] Urvoy-Keller, G. Nain, P. (2006). Interleaved Weighted Round-Robin: A Network Calculus Analysis. Performance Evaluation Review, 33(3), 19-30.

[27] Smith, G. (2011). Indexing PostgreSQL for High-Performance. Springer Science Business Media.

[28] Jiang, P., Gogan, J. L. (2017). A study of software prototyping and its dimensions: Horizontal and Vertical Prototypes. Journal of Software Engineering and Applications, 10(08), 816-834.

[29] RabbitMQ vs Kafka [Электронный ресурс] // URL: https://www.simplilearn.com/kafka-vs-rabbitmq-article