Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 19Б.10-мм

Пяйве Олег Андреевич

Редактор графов рабочих процессов для Airflow

Отчёт по учебной практике

Научный руководитель:
доц. кафедры СП, к. т. н. Т. А. Брыксин

Консультант:
инженер ключевых проектов Huawei И. А. Тимуров

Санкт-Петербург
2021

Оглавление

Введение

В настоящее время автоматизация выполнения рутинных процессов — важный элемент для любой компании. Автоматизация не только позволяет выполнять простые задачи быстро и безошибочно, но и даёт возможность людям заниматься творческими задачами вместо того, чтобы делать одну и ту же работу изо дня в день. Для автоматизации процессов часто применяется Airflow.

Airflow[1] – это платформа для управления рабочими процессами. С её помощью можно создавать операторы – программы, каждая из которых выполняет одну конкретную задачу (например, преобразовать файл из одного формата в другой), и составлять из них графы. Граф задаёт последовательность выполнения операторов, а также условия, при которых должна выполниться та или иная программа. Для того, чтобы автоматизировать выполнение рабочего процесса, необходимо создать его описание в виде графа. Далее он может запускаться как по расписанию, так и по команде пользователя.

Airflow позволяет переиспользовать код: одна программа может быть применена для решения подзадач нескольких комплексных задач одновременно. Частый сценарий использования Airflow – создание новых графов из уже созданных операторов.

Airflow руководствуется принципом "Configuration as code". Это означает, что задание графа происходит с помощью написания кода на Python. С одной стороны, это даёт очень большую гибкость. С другой стороны, такой подход сильно усложняет создание графов. При этом большая часть графов выглядит очень похоже и запускается с одними и теми же настройками: их код может быть сгенерирован автоматически по набору операторов и связей между ними.

Наличие дополнительного графического пользовательского интерфейса делает инструмент более удобным. В Airflow его нет, поэтому код даже простых графов, который может быть сгенерирован автоматически, необходимо писать вручную. Это делает работу с сервисом

[1]Официальная страница Airflow https://airflow.apache.org/

Airflow достаточно сложной. Таким образом, добавление в Airflow интерфейса для создания простых графов поможет улучшить удобство его использования.

В рамках данной практики было необходимо разработать сервис, позволяющий создавать операторы, рисовать графы в пользовательском интерфейсе, конвертировать их во внутреннюю нотацию описания плана работы для Airflow — DAG[2] и запускать.

[2]Описание DAG на сайте Airflow: https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html

1 Цель и задачи

1.1 Цель

Цель данной работы – создание платформы, позволяющей в графическом интерфейсе создавать графы рабочих процессов и конвертировать их для использования с Airflow.

1.2 Задачи

Для осуществления данной цели были поставлены задачи:

1. Провести обзор предметной области и изучить существующие аналоги
2. Разработать архитектуру будущего приложения и интерфейс для взаимодействия клиентской и серверной части (API), а также нотацию для записи графов
3. Разработать клиентскую часть приложения
4. Разработать серверную часть приложения
5. Протестировать полученный продукт на реальных пользователях.

2 Обзор

2.1 Инструменты для автоматизации процессов

В настоящее время существует несколько инструментов для автоматизации рабочих процессов.

Luigi[3] – платформа, разработанная в компании Spotify. Поставляется со встроенной поддержкой Hadoop[4], в частности, поддерживает MapReduce. Графический интерфейс предоставляет возможность посмотреть смотреть на существующие графы, но создавать их можно только с помощью написания кода.

Azkaban[5] – проект фонда Apache Foundation. Рассчитан в основном на работу с Hadoop и планирование простых задач, так как работа с ним происходит только в графическом интерфейсе. Не подходит для работы со сложными зависимостями или условиями.

Oozie[6] также является проектом Apache Foundation. Меньше, чем Azkaban, ориентирован на удобство использования и больше – на работу со сложными рабочими процессами. Графический интерфейс, так же, как и в Luigi, ориентирован только на просмотр существующих графов, а не на их создание.

Airflow. Эта платформа выгодно отличается от аналогов тем, что её использует большое сообщество активных разработчиков, поэтому по Airflow доступно большое количество учебных материалов, а ответы на сложные вопросы всегда можно получить, задав вопрос сообществу. Однако, графический интерфейс Airflow устроен так же, как и у большиства аналогов: в нём можно только посмотреть графы, описанные кодом на Python.

Так как было необходимо сохранить не только удобство работы с простыми рабочими процессами, но и возможность создавать описания сложных, был выбран наиболее популярный инструмент: Airflow.

[3]Luigi GitHub: https://github.com/spotify/luigi

[4]Официальная страница Apache Hadoop https://hadoop.apache.org/

[5]Официальная страница Apache Azkaban: https://azkaban.github.io/

[6]Официальная страница Oozie: https://oozie.apache.org/

2.2 Airflow: преимущества и недостатки

В Airflow есть графический интерфейс, в котором может быть отображён граф, заданный с помощью кода на Python. Однако, большинство операторов выглядят достаточно просто, поэтому часто в написании кода нет необходимости: нужно лишь заполнить стандартный шаблон: какие в графе операторы, каковы их входные и выходные данные и что нужно сделать для того, чтобы запустить оператор.

Код даёт большую гибкость, но сильно усложняет систему в целом. Приходится заботиться не только о скорости выполнения программы, но и о читаемости её кода: DAG-и нужно поддерживать и рефакторить, как и код других элементов разрабатываемого продукта. Если же используется авто-генерация кода, а сам DAG можно посмотреть графически в интерфейсе, то сложность сгенерированного кода играет намного меньшую роль, так как нет необходимости его читать. После того, как шаблон генерации кода будет отлажен и протестирован, можно быть уверенным, что весь сгенерированный код будет работать корректно. В случае же с написанием кода каждый отдельный DAG требует тестирования.

В декабре 2020 года вышло большое обновление Airflow 2.0 – появился декоратор `@task`, избавляющий от необходимости писать отдельный PythonOperator (оператор Airflow, запускающий код на Python) для каждой используемой функции. Это сильно сокращает количество лишнего кода, требующегося для создания DAG-а, но добавляет некоторые другие проблемы, в частности, появляются лишние рёбра с зависимостями (см. пример графа, на рис. 1), что не очень удобно. Таким образом, в Airflow 2.0 была сильно упрощена структура кода, но тем не менее разработка простых DAG, состоящих только из Bash и Python операторов, в графическом интерфейсе по-прежнему удобнее, чем написание кода.

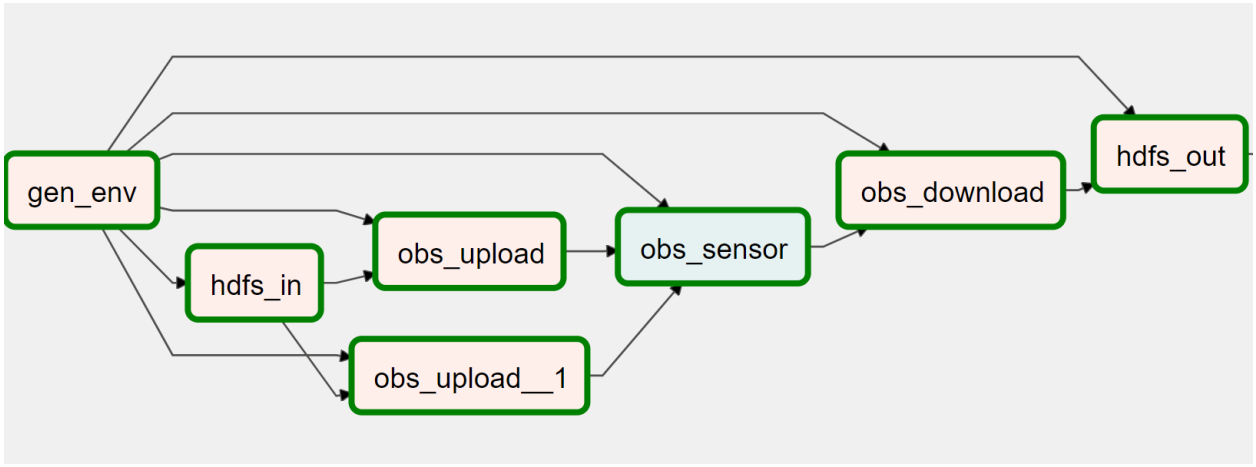

Рис. 1: DAG и зависимости между операторами

Система Airflow существует только под Linux, поэтому устройства с Windows без WSL[7] не могут с ней работать. Airflow UI, запущенный на сервере, может быть доступен с любого устройства сети, но в нём можно работать только с уже созданными графами: не поддерживается удалённая загрузка файлов на сервер.

2.3 Используемые технологии

Был выбран формат веб-приложения, потому что веб-приложение поддерживается сразу всеми платформами, включая десктопные (Windows, Linux, Mac OS), и мобильные (Andoid, iOS, менее популярные операционные системы). Кроме того, оно не требует установки.

Такой формат немного ограничивает возможности разработчика, но в данном случае JavaScript позволяет реализовать всё, что планировалось. При этом некоторые необходимые функции, такие, как заполнение форм, в браузере обрабатываются даже удобнее, чем в любом десктопном приложении: браузер запоминает введённые данные и подсказывает их при следующем вводе. Это удобно, например, для редактирования оператора.

В качестве языка программирования для серверной части был выбран **Python**: язык программирования общего назначения, предназначенный для быстрого создания прототипа будущего продукта.

[7]WSL документация: https://docs.microsoft.com/ru-ru/windows/wsl/about

Наиболее популярные инструменты для создания веб-сервисов – это Django[8] и Flask[9]. Django предоставляет множество возможностей «из коробки»: в этом пакете есть, например, собственный ORM[10] фреймворк. Flask же позволяет гибко настроить используемые инструменты. Кроме того, именно Flask в сочетании с SQLAlchemy[11] используется в самом Airflow. Так как в данном проекте важна была именно гибкость настройки компонентов, был выбран Flask [2].

В качестве системы управления базами данных выбрана PostgreSQL [3]. Данная СУБД обладает следующими преимуществами:

1. Качественная и подробная документация, позволяющая разобраться во всех тонкостях работы с базой данных

2. Большое количество активных разработчиков, отвечающих на вопросы на форуме Stack Overflow [12]

3. Удобный клиент *pgAdmin*[13]

SQLAlchemy — ORM пакет для Python. Обладает широкой функциональностью и при этом очень прост в освоении. Один из наиболее популярных инструментов в своей сфере, имеет большое сообщество разработчиков, активно развивается.

Язык программирования для клиентской части – это JavaScript. Были использованы две основные JS библиотеки.

Для того, чтобы обрабатывать добавление и удаление операторов, а также их соединение, была использована библиотека MXGraph [1]. Её используют такие крупные сервисы, как Diagrams.net[14]. В MXGraph реализованы все необходимые методы работы с графами, такие, как валидация рёбер, сериализация графа или перетаскивание узлов графа с панели управления. При этом, эта библиотека предоставляет воз-

[8]Официальный сайт Django: https://www.djangoproject.com/
[9]Документация Flask: https://flask.palletsprojects.com/en/2.0.x/
[10]Object-Relation Mapping
[11]Документация SQLAlchemy: https://docs.sqlalchemy.org/en/13/
[12]Stack Overflow: https://stackoverflow.com/
[13]https://pgadmin.org/
[14]https://app.diagrams.net

можность легко переопределить нужное поведение, например, написать свой сериализатор.

Во время работы возникла необходимость использовать списки, которые будут синхронизироваться с сервером. Для этой цели был выбран KnockoutJS [5] – фреймворк, использующий паттерн Model-View-ViewModel (структура – рис. 2).

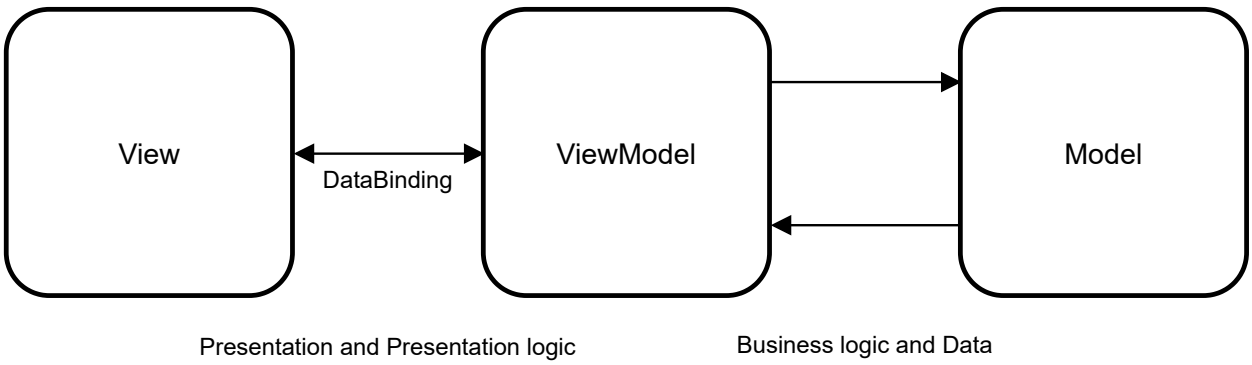

Рис. 2: Структура Model-View-ViewModel

Для того, чтобы интерфейс адаптировался для различных соотношений сторон и размеров экрана, был выбран Bootstrap [4]. Начать пользоваться им очень просто, тем не менее, это очень продвинутый и гибкий инструмент для работы с HTML-страницами. В Bootstrap все элементы сделаны в едином стиле (поэтому можно просто выбирать подходящие компоненты и использовать их: они всегда будут смотреться гармонично). Страница, созданная с помощью этого инструмента, одинаково выглядит во всех современных браузерах. Кроме того, Bootstrap позволяет гибко настраивать поведение элементов на странице, например, модальных окон.

3 Реализация

3.1 Сохранение графа и оператора в базе данных

Когда пользователь создаёт оператор, он вводит все необходимые сведения о нём, которые в дальнейшем хранятся в базе данных. Аналогично — с графом. Однако, для графа хранятся сразу две нотации.

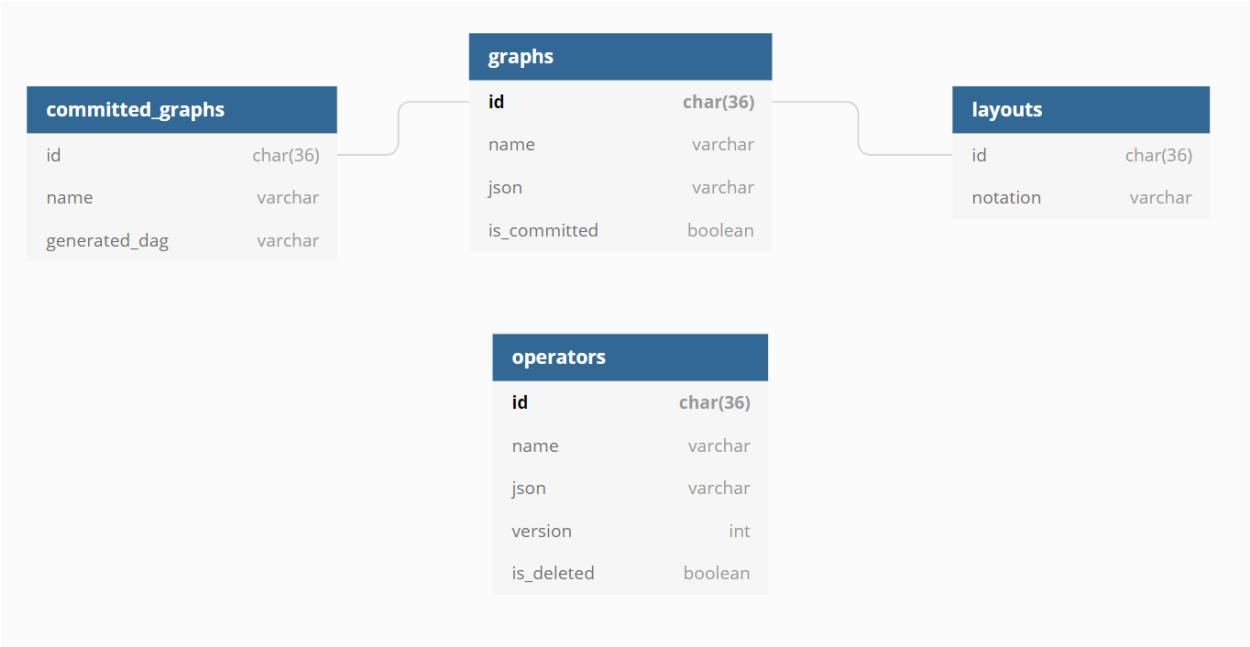

Рис. 3: Структура базы данных

Первая — стандартная для MXGraph. Она использует XML, что сильно затрудняет её чтение. Однако, она используется только для того, чтобы сохранять не только связи между операторами, но и их геометрическое расположение друг относительно друга. Иногда автоматические алгоритмы расположения операторов на плоскости дают не лучший результат: рёбра пересекаются между собой или пропадает последовательное расположение операторов слева направо и затруднено понимание относительного порядка их выполнения. Поэтому удобнее сохранять граф в точности в таком виде, в каком он был нарисован.

Вторая нотация разработана в рамках данной работы. Здесь и далее будем называть её JSON-нотацией. Она занимает намного меньше места в базе данных, чем стандартная от MXGraph. В стандартной есть

часть тех данных, которые есть в JSON-нотации, но получать их оттуда сложно: это занимает много времени и делает код очень трудно читаемым. Большая часть стандартной нотации нужна для обеспечения внутренней функциональности MXGraph, поэтому там хранится много дополнительных данных.

JSON-нотация помогает хранить несколько версий оператора. Был выбран следующий подход к обработке версий. Все операторы считаются неизменяемыми. Когда оператор сохранён в базе, удалить его оттуда или изменить уже невозможно. Для создания изменённого оператора необходимо создать новую его версию, в которой можно подправить любые необходимые данные. Сделано это для того, чтобы старые графы, которые могли использовать старую версию оператора, продолжали работать так, как раньше. Таким образом, сохраняется обратная совместимость операторов с графами. Поддержка версий расширяет возможности: если один и тот же оператор должен быть применён очень похожими, но всё же разными способами, можно создать единый оператор с тремя версиями вместо трёх разных: это упрощает понимание структуры системы.

JSON-нотация, кроме того, содержит в себе сведения о том, какие операторы есть в графе и как они зависят друг от друга: указывается тип входов и выходов всех операторов и их имена. Благодаря этому сильно упрощается валидация графа, которая происходит на клиенте в момент его сохранения.

Валидация – важный элемент работы с DAG. DAG – это способ представления зависимостей задач в виде ориентированного графа. При этом задачи должны быть такими, чтобы можно было составить из них очередь и выполнить их последовательно. Для этого необходимо и достаточно, чтобы в графе не было контуров. Поэтому, перед сохранением графа или, тем более, окончательной конвертацией его в DAG, нужно проверить, не ошибся ли пользователь, создав контур в графе. Это реализуется с помощью алгоритма обхода графа в глубину.

Однако, проверить граф на наличие контуров может и сама система Airflow. При этом проверить, что все входы и выходы операторов

имеют нужные типы данных, можно только во время исполнения DAG, которые бывают очень сложные и долго выполняющиеся. Поэтому производить отладку непосредственным запуском в данном случае неправильно.

Разработанный в рамках данной работы редактор использует заданные пользователями данные о необходимых входных и выходных данных для каждого оператора, а затем проверяет, что выходные данные первого оператора, создающего зависимость для второго оператора, имеют тот же тип, что и входные данные для второго оператора. Таким образом, исключается возможность ошибки с типами данных.

3.2 Особенности реализации

Для того, чтобы конвертировать граф из JSON-нотации в DAG, используется Jinja[15] — шаблонизатор для языка Python. За счёт того, что структура всех DAG одинаковая, есть возможность заполнить данные самих операторов и связи между ними с помощью стандартных средств Airflow.

Для представления оператора был создан AbstractOperator. Этот класс — наследник стандартного класса Airflow для исполнения команд в командной строке — BashOperator — и расширяет его возможности. BashOperator только исполняет команду и проверяет, какой код возврата был получен. AbstractOperator перед исполнением команды загружает на устройство все файлы, которые необходимы для исполнения оператора. Затем он исполняет команду и загружает все выходные файлы обратно в облачное хранилище. Для передачи сведений о том, по какому пути находятся сгенерированные файлы, используется XCom — внутренний интерфейс Airflow для коммуникации между операторами.

[15]Официальный сайт Jinja: https://jinja.palletsprojects.com/en/3.0.x/

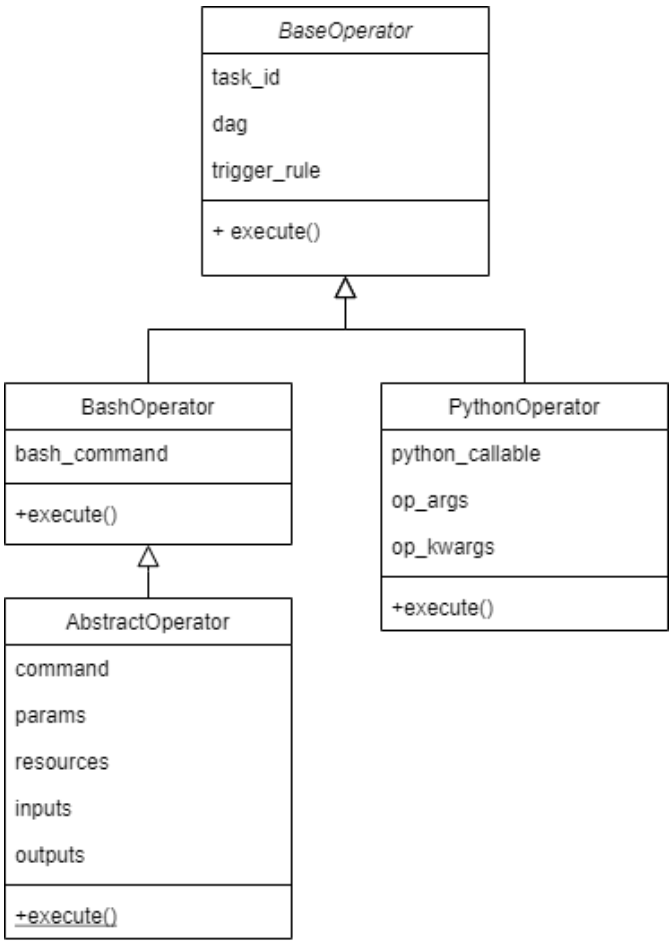

Рис. 4: Диаграмма классов для AbstractOperator

Таким образом, сохраняется возможность запускать операторы параллельно на разных устройствах: синхронизация между ними осуществляется с помощью общедоступного хранилища файлов.

3.3 Интерфейс

В пользовательском интерфейсе редактора графов поддерживаются базовые сочетания клавиш, такие, как Ctrl+z и Ctrl+shift+z: отмена и возврат отмены действия. Реализовано это стандартными средствами MXGraph: эта библиотека позволяет назначать любые действия на нажатия соответствующих клавиш. Отмена действия реализована с помощью средства MXGraph для этого — MXUndoableEdit.

С заданной периодичностью система пользовательского интерфейса посылает запросы на сервер, который присылает актуальное состояние

DAG в Airflow, отсортировав его по типу запускаемого DAG, и загружает эти данные во ViewModel. Далее KnockoutJS отрисовывает все элементы с помощью заданной HTML-разметкой структуры.

3.4 Обзор возможностей редактора

При запуске веб-приложения открывается главная страница.

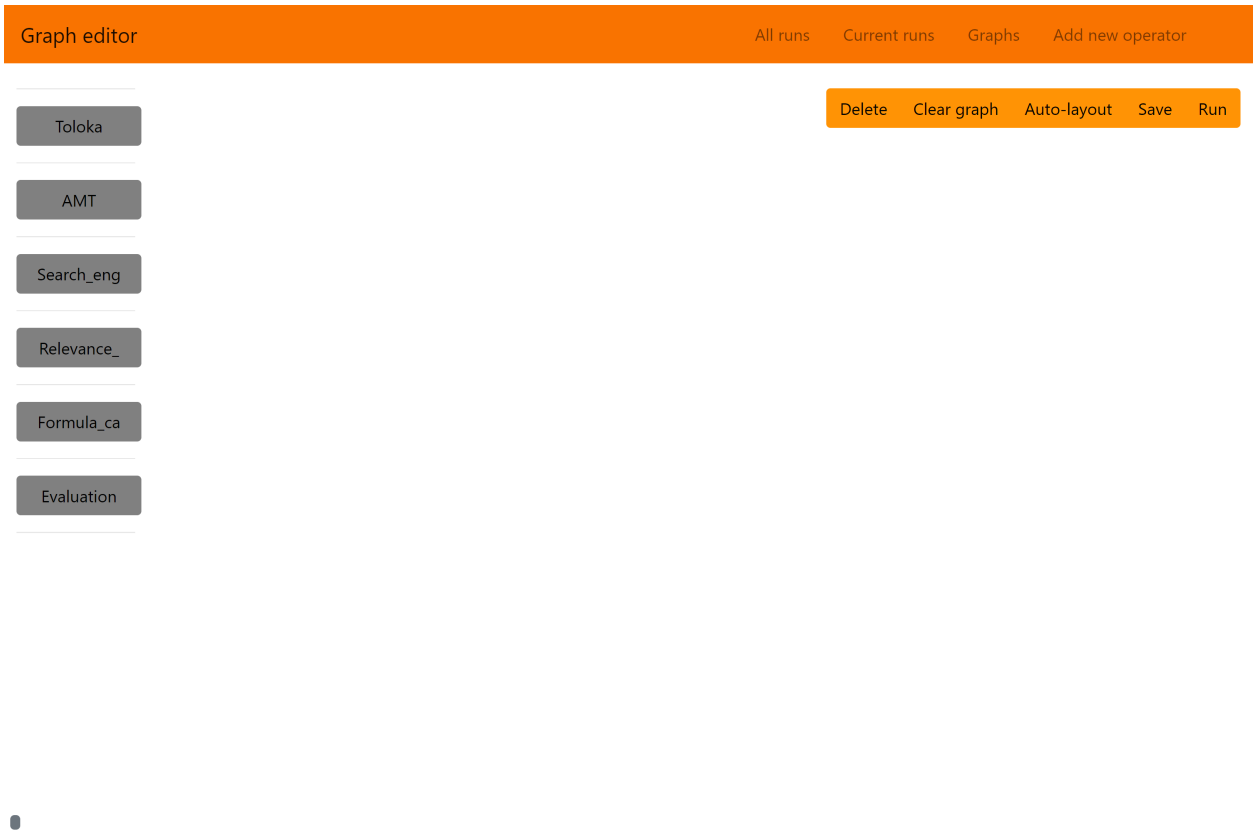

Рис. 5: Главная страница

Слева находится список всех доступных операторов. Из них можно составить граф, например, такой, как показано на иллюстрации ниже. В данном случае к нему был применён Auto-layout: операторы выровнены таким образом, чтобы линейный порядок был хорошо виден.

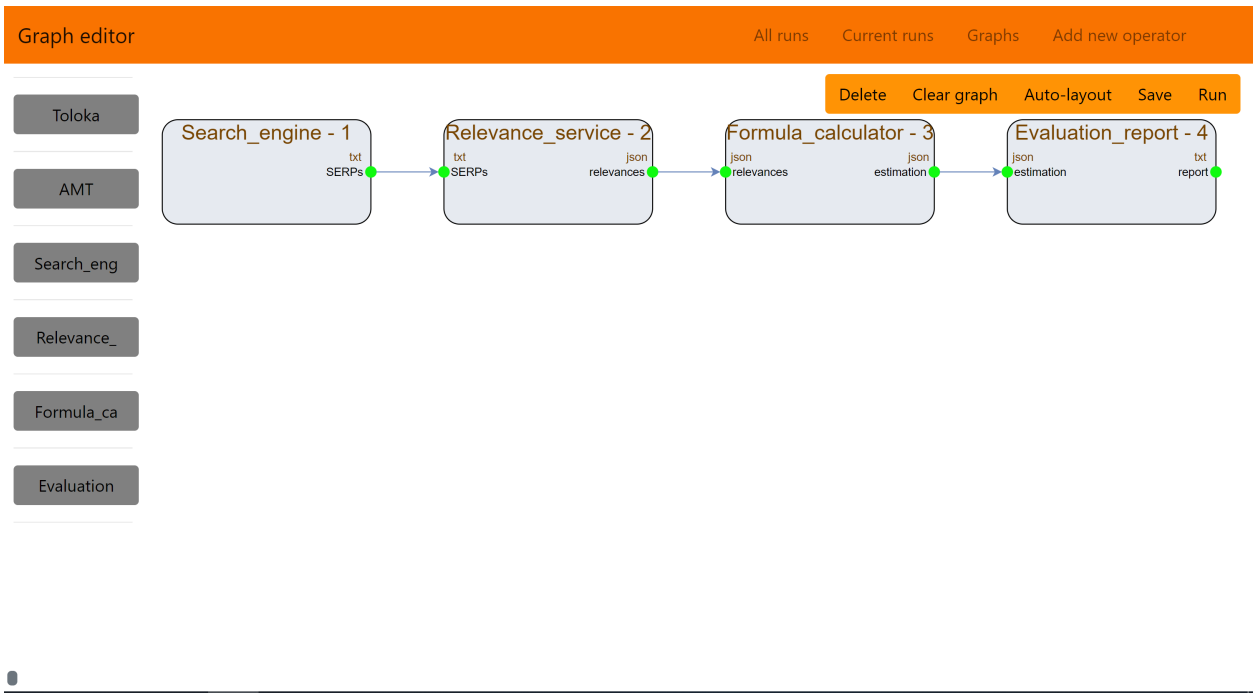

Рис. 6: Пример графа

Далее представлены основные окна, доступные в интерфейсе редактора графов.

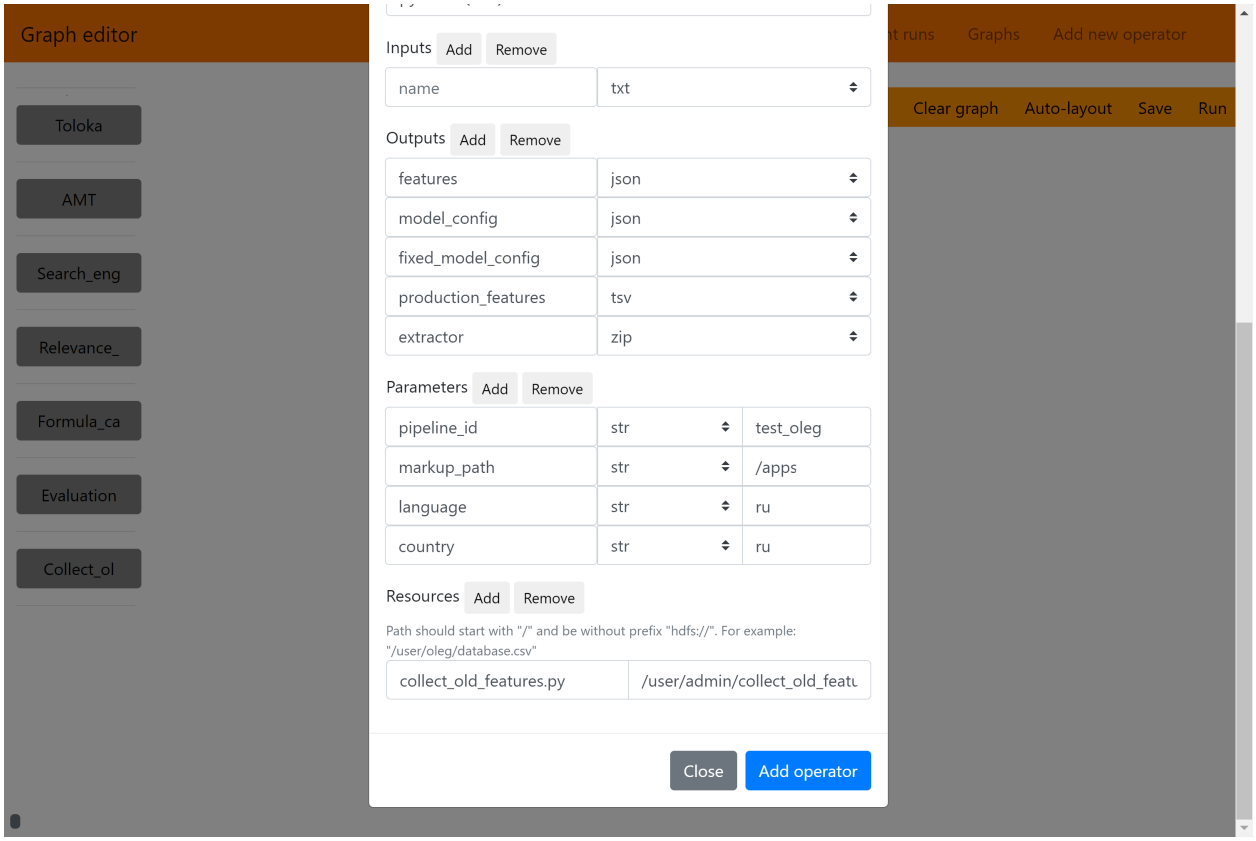

Рис. 7: Окно добавления оператора

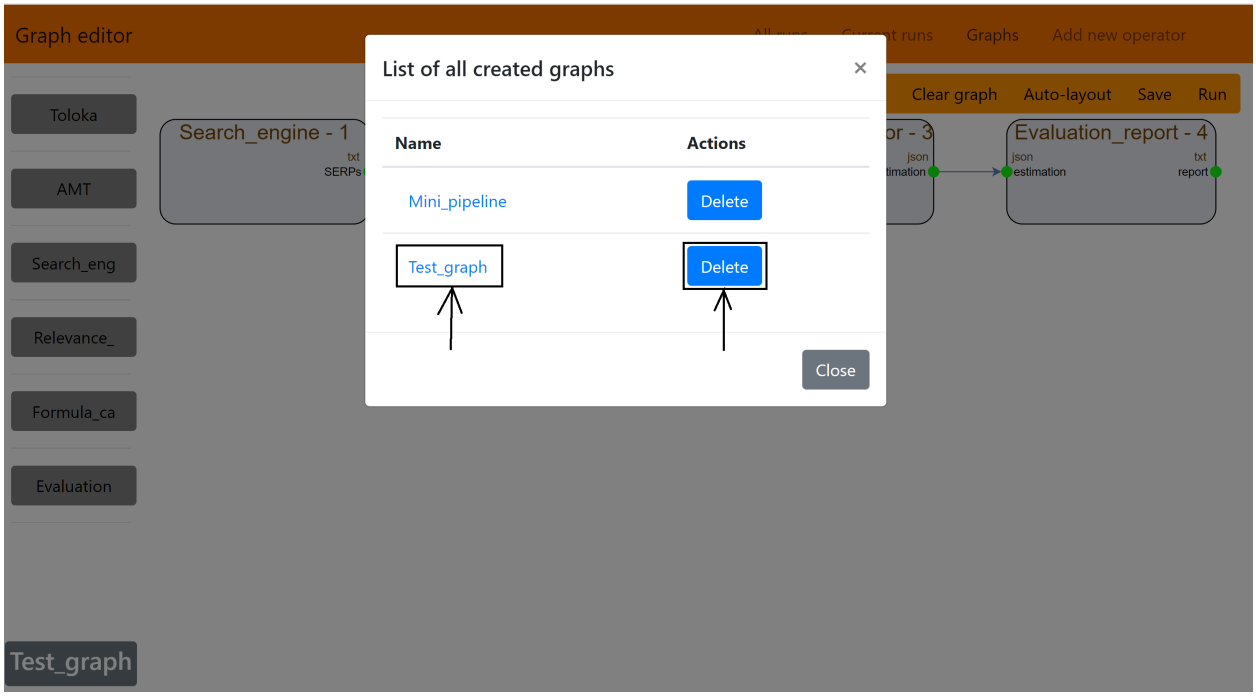

Рис. 8: Список графов

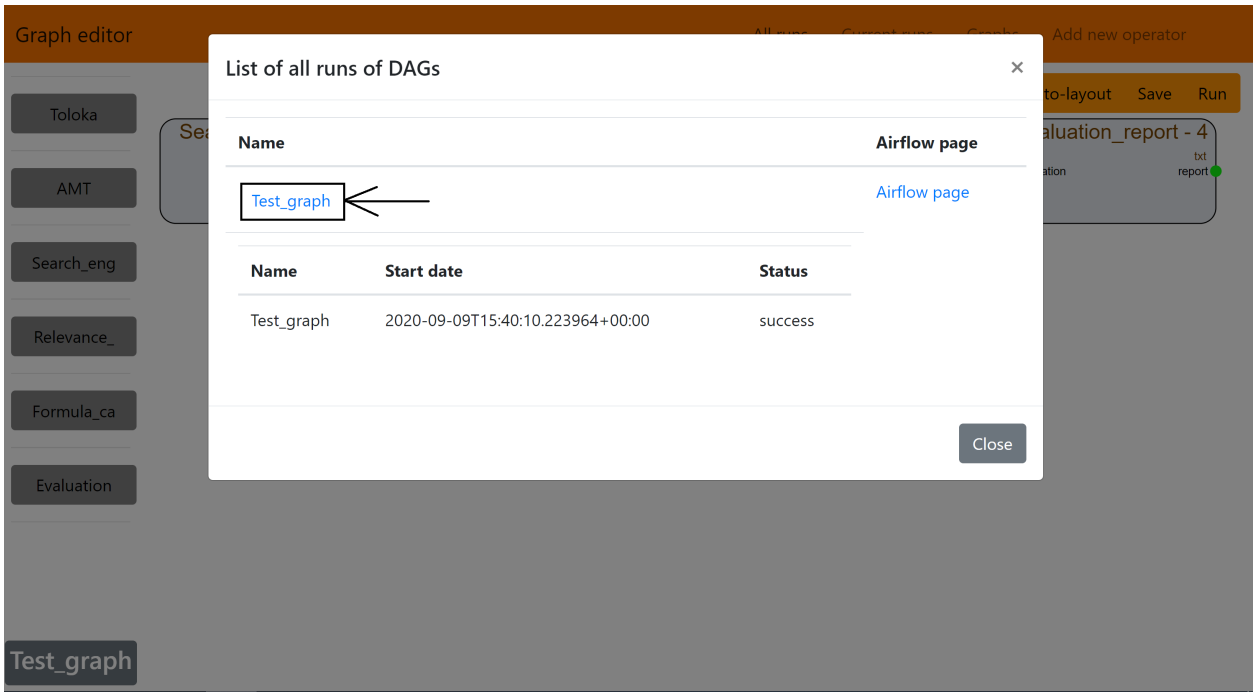

Рис. 9: Окно всех запусков

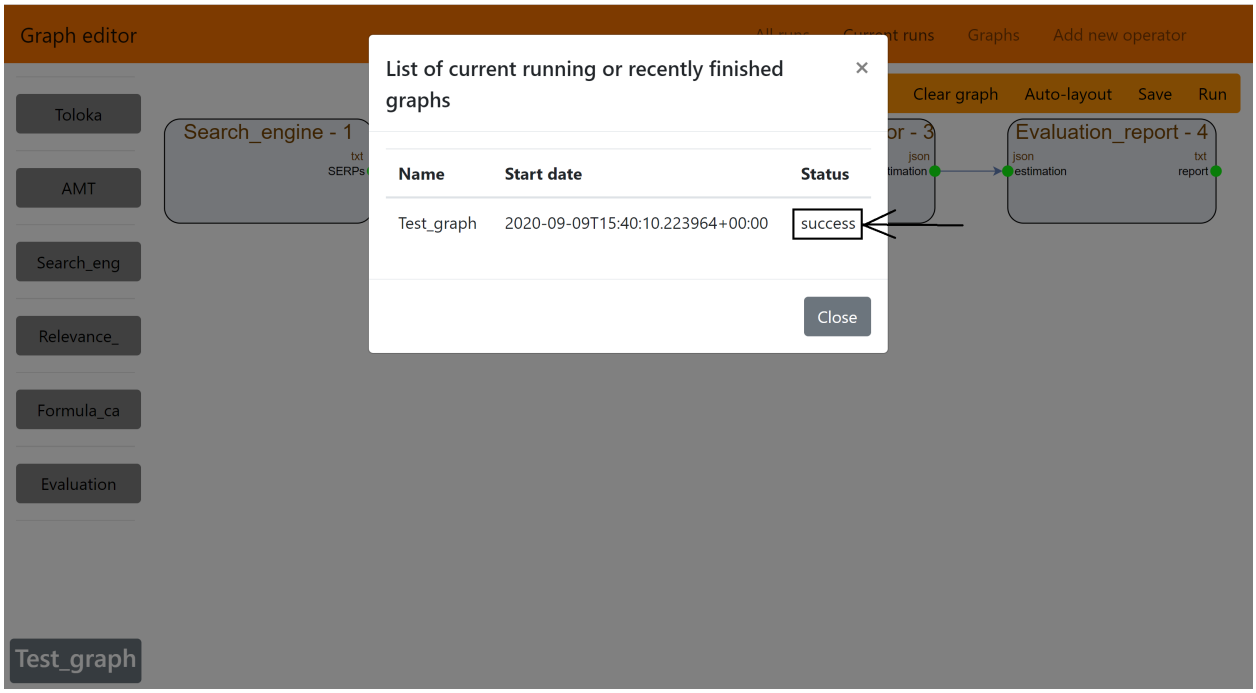

Рис. 10: Недавние запуски

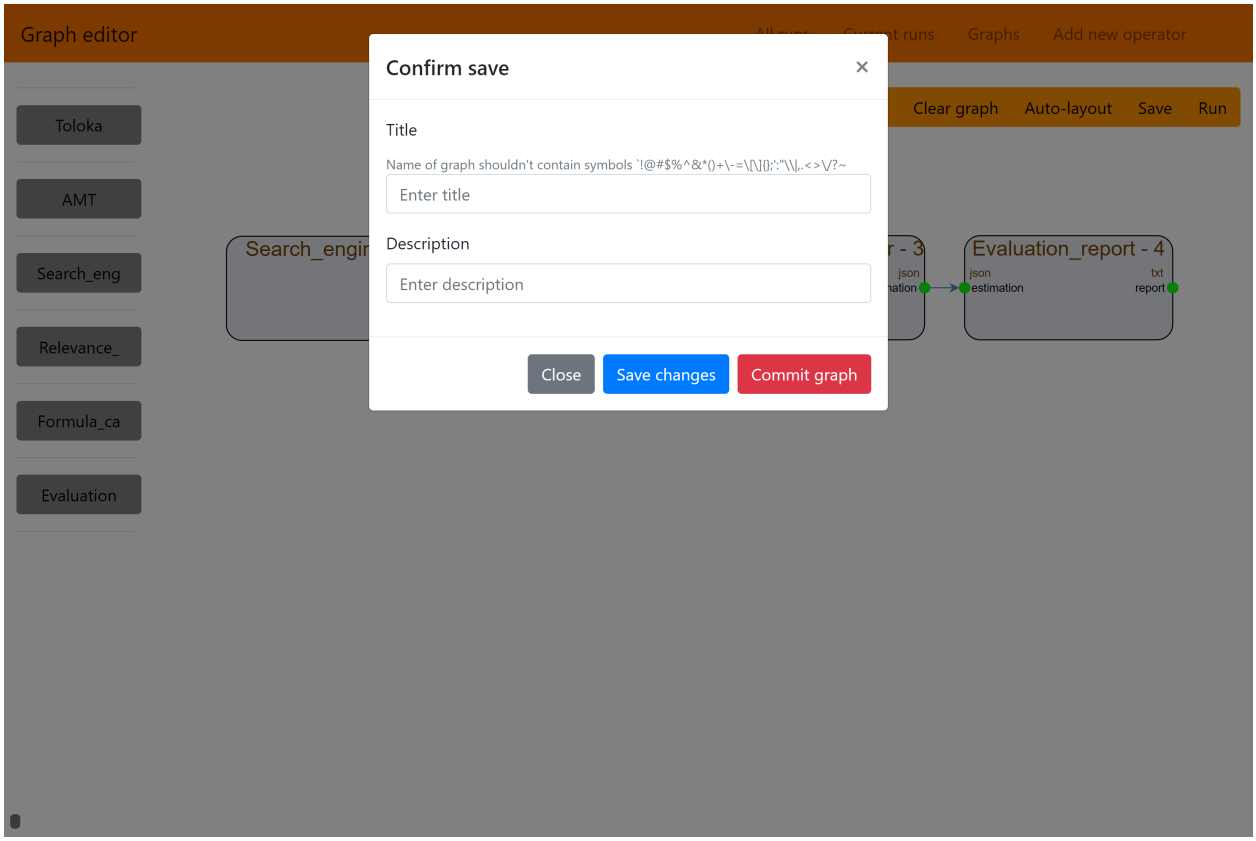

Рис. 11: Окно сохранения графа

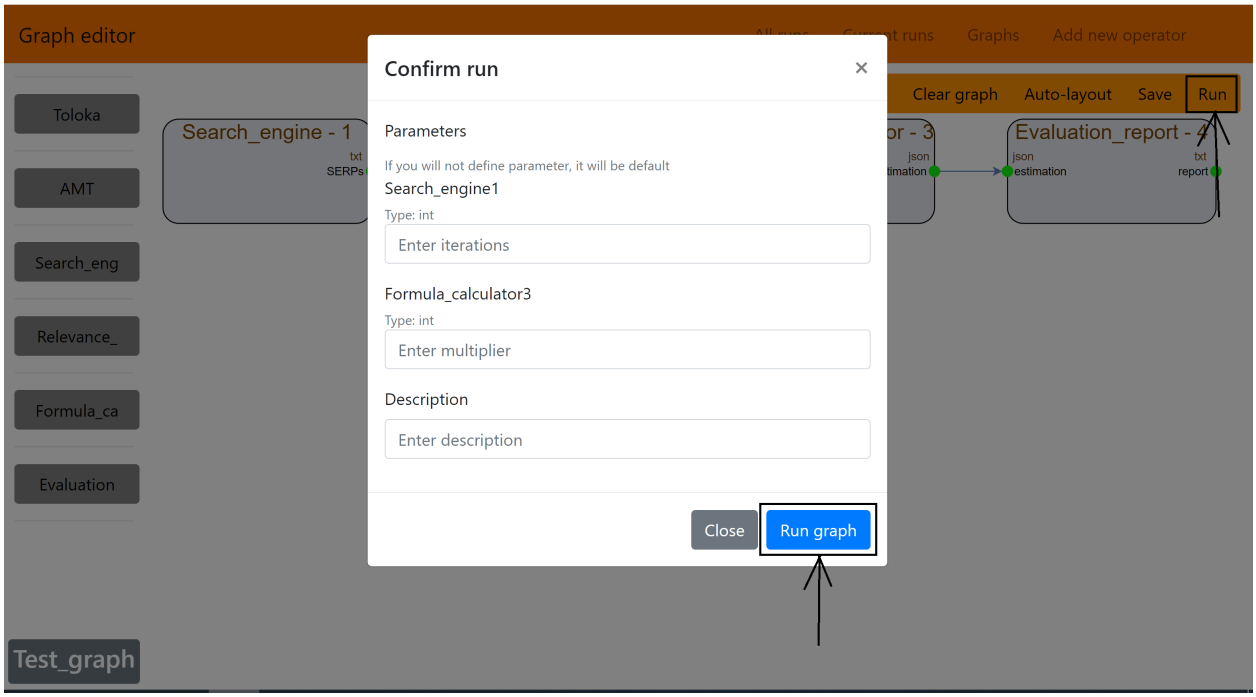

Рис. 12: Окно запуска графа

3.5 Тестирование

Тестирование редактора проводили реальные пользователи, члены команды оценки качества поиска компании Huawei, — каждый со своего устройства. Таким образом, было проверено, что веб-интерфейс корректно отображается на устройствах с различным разрешением и размером экрана.

Для тестирования самой функциональности каждым пользователем было проведено end-to-end тестирование. Были последовательно использованы следующие возможности редактора:

1. Добавление операторов и работа с ними: редактирование и удаление

2. Составление графа из операторов, добавление и удаление вершин и рёбер, отмена действия и возврат отмены, некорректные соединения рёбрами (вход и выход разных типов или ребро, не ведущее ни в какую вершину)

3. Сохранение и редактирование сохранённого графа, конвертация его в DAG

4. Запуск DAG и проверка его статуса через списки запусков

Таким образом, было проведено бета-тестирование всех частей редактора. В его ходе были выявлены ошибки в отображении статуса DAG и в валидации графа в клиентской части. Все эти ошибки были исправлены и было ещё раз проверено, что теперь они не проявляются.

Заключение

В ходе данной работы были выполнены следующие задачи:

1. Проведён обзор предметной области: аналоги Airflow и возможности его доработки
2. Разрабона архитектуру приложения, API для передачи данных между серверной и клиентской частями, а также нотация для записи графов
3. Разработана клиентская часть приложения:
   (a) Создан пользовательский интерфейс, позволяющий редактировать, сохранять и удалять графы
   (b) Добавлена возможность удаления и добавления операторов
   (c) Добавлена проверка графа перед его конвертацией в Airflow DAG
4. Разработана серверная часть приложения:
   (a) Создан модуль для обработки запросов из пользовательского интерфейса
   (b) Создан модуль конвертации графа в Airflow DAG
   (c) Развёрнута и интегрирована с серверной частью приложения система управления базами данных PostgreSQL.
5. Полученный продукт протестирован на реальных пользователях, в результате этого тестирования было выявлено и исправлено несколько проблем в синхронизации пользовательского интерфейса, сервера и Airflow.

Данный проект разрабатывался в рамках практики в компании Huawei. По требованию руководства компании, код проекта находится в приватном репозитории.

Перспективы развития

В данный момент, если редактором пользуется несколько человек сразу, каждый из них может запустить DAG, и этот запуск появится в общем списке. Таким образом, будет невозможно понять, какой запуск кем совершён. В большинстве случаев это и не требуется, так как важно только увидеть итоговый результат: DAG отработал успешно и сформировал необходимые файлы.

Однако, в некоторых случаях пользователю может быть удобнее оставить среди запусков только те, которые создал он сам: так проще найти нужный статус и конфигурацию, с которой был запущен DAG. Таким образом, удобна будет возможность **регистрации и авторизации**.

В Airflow не показывается прогнозов насчёт того, как долго будет выполняться тот или иной DAG. Соответственно, можно добавить возможность приблизительно предсказывать оставшееся время выполнения. С очень высокой точностью это сделать не получится, так как по внешнему виду команды или кода невозможно понять, сколько времени он будет работать. Однако, если усреднить время всех предыдущих запусков и использовать полученное значение в качестве оценочного времени работы DAG, то точность будет достаточной для повседневного использования.

Список литературы

[1] API MXgraph. — Access mode: https://jgraph.github.io/mxgraph/docs/js-api/files/index-txt.html (online; accessed: 2020-05-20).

[2] Документация Flask. — Access mode: https://flask.palletsprojects.com/en/2.0.x/ (online; accessed: 2020-05-20).

[3] Документация PostgreSQL. — Access mode: https://www.postgresql.org/docs/ (online; accessed: 2020-05-20).

[4] Официальный сайт Bootstrap. — Access mode: https://getbootstrap.com/ (online; accessed: 2020-05-20).

[5] Официальный сайт KnockoutJS. — Access mode: https://knockoutjs.com/ (online; accessed: 2020-05-20).