МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

09.03.03 Прикладная информатика
(код и наименование направления подготовки / специальности)

Разработка социальных и экономических информационных систем
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка системы мониторинга и управления данными чат-бота

Обучающийся Н.А. Третьяков _____
(Инициалы Фамилия) (личная подпись)

Руководитель канд.пед.наук, доцент, О.М. Гущина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант канд.пед.наук, доцент, А.В. Егорова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2024

Аннотация

Тема бакалаврской работы – «Разработка системы мониторинга и управления данными чат-бота».

Актуальность работы обусловлена необходимостью автоматизации обработки и мониторинга данных.

Объектом исследования бакалаврской работы является проект внедрения системы мониторинга и управления данными.

Предметом исследования бакалаврской работы является проект использования инструментов разработки для внедрения системы мониторинга и управления данными.

Цель выпускной квалификационной работы – применение современных инструментов разработки, обеспечивающих высокую эффективность внедрения системы мониторинга и управления данными.

Методы исследования – методологии и технологии внедрения и проектирования информационных систем.

Данная работа состоит из введения, двух глав, заключения и списка используемой литературы. Практическая значимость бакалаврской работы заключается в разработке проекта использования инструментов разработки для внедрения системы мониторинга и управления данными, обеспечивающей повышение эффективности управления деятельностью компании.

Результаты бакалаврской работы представляют практический интерес и могут быть рекомендованы для разработчиков, работающими над проектами внедрения системы мониторинга и управления данными в деятельность предприятий и компаний.

Бакалаврская работа состоит из 58 страницы текста, 86 рисунков и 31 источников.

Annotation

The topic of the bachelor's thesis is "Development of a data monitoring and management system in a chatbot".

The relevance of the work is due to the need to automate data processing and monitoring.

The object of research of the bachelor's work is a project for the implementation of a data monitoring and management system.

The subject of the research of the bachelor's work is a project of using development tools to implement a data monitoring and management system.

The purpose of the final qualification work is the use of modern development tools that ensure high efficiency of the implementation of a data monitoring and management system.

Research methods – methodologies and technologies for the implementation and design of information systems.

The practical significance of the bachelor's work lies in the development of a project for using development tools to implement a data monitoring and management system that improves the efficiency of managing the company's activities.

This work consists of an introduction, two chapters, a conclusion and a list of the literature used.

The results of the bachelor's work are of practical interest and can be recommended for developers working on projects for the implementation of a data monitoring and management system in the activities of enterprises and companies.

The bachelor's thesis consists of 58 pages of text, 86 figures and 31 sources.

Оглавление

Введение

В современном информационном обществе автоматизация процессов играет ключевую роль в улучшении эффективности работы различных систем.

Реализация данного проекта представляет актуальность и научно-практический интерес.

Объектом исследования бакалаврской работы является проект внедрения системы мониторинга и управления данными.

Предметом исследования бакалаврской работы является проект использования инструментов разработки для внедрения системы мониторинга и управления данными.

Целью работы являлось разработка решения, направленного на улучшение эффективности и функциональности системы. Для достижения этой цели были поставлены следующие задачи: анализ существующей системы и выявление проблемных моментов, разработка методов и инструментов для модернизации системы, реализация и тестирование предложенного решения.

Практическая значимость бакалаврской работы заключается в разработке проекта использования инструментов разработки для внедрения системы мониторинга и управления данными, обеспечивающей повышение эффективности управления деятельностью компании.

Данная работа состоит из введения, двух глав, заключения и списка используемой литературы. В первой главе дано описание предметной области. Вторая глава посвящена разработке для внедрения системы мониторинга и управления данными. В данной главе произведено моделирование, программная реализация, реализация сбора данных, а также реализация дашбордов.

Бакалаврская работа состоит из 58 страниц текста, 86 рисунков и 31 источников.

Глава 1 Описание предметной области

1.1 Описание задачи мониторинга и управления данными в чат-боте

В современном мире цифровизация и автоматизация процессов становятся все более важными компонентами успешного функционирования предприятий. Одним из ключевых элементов этой трансформации является сбор, обработка и использование данных.

С ростом числа клиентов возникает необходимость эффективного управления данными и мониторинга работы. Несмотря на то, что многие компании уже используют аналитику данных в своей деятельности, существует несколько проблем, которые необходимо решить для обеспечения более эффективного функционирования таких систем:

- Недостаточная производительность: Существующие системы управления данными и мониторинга могут не обеспечивать достаточной производительности при обработке большого объема запросов. Это может приводить к задержкам в реакции на происходящие события.
- Ограниченный функционал: Многие системы мониторинга и управления данными имеют ограниченный функционал и не предоставляют инструменты для настройки достаточно глубокого анализа данных для принятия обоснованных решений по оптимизации работы системы.

Нам поступил заказ от центра цифровых компетенций Тольяттинского Государственного Университета разработать и внедрить к ним в существующего чат-бота систему мониторинга и управления данными. Который был создан для выполнения их функций и задач.

Назначением центра цифровых компетенций является:

- Разработка и актуализация закрепленных за Центром образовательных программ.
- Занятие значимой доли на рынке «скилловых» образовательных программ высшего образования по IT-направлениям.
- Коммерциализация проектов учебно-проектной лаборатории «IT Student».

Центр цифровых компетенций ТГУ подчиняется проректору по цифровизации и представляет из себя небольшую структуру в составе ТГУ. Чат-бот был создан по заказу центра маркетинга. Необходимость в чат-боте была обоснованна тем, что в колл-центр от абитуриентов и студентов поступает большое количество типовых звонков, особенно в период поступления студентов и первые месяцы учёбы. В связи с этим у колл-центра появляется большая сезонная нагрузка, которую потенциально можно значительно снизить, отсеивая большую часть типовых вопросов.

1.2 Описание чат-бота

Чат-бот, отвечающий на вопросы, был реализован на языке программирования python на библиотеке aiogram и имел событийно-ориентированную архитектуру.

Событийно-ориентированная архитектура (EDA) – это парадигма программной архитектуры, способствующая порождению, обнаружению, потреблению событий и реакции на них. В нашем случае после написания сообщения определённому боту в телеграмме, телеграмм уведомлял нашего бота об этих изменениях за счёт так называемой системы webhook, бот обрабатывал его и отвечал на сообщение продолжая ожидать от телеграмма новое событие.

Чат бот работал с СУБД postgres который разворачивался совместно с чат-ботом. Удобство в поднятии чат-бота и postgres, а также их бесперебойное и без

проблемное взаимодействие друг с другом обеспечивает контейнеризация через Docker(рисунок 1).

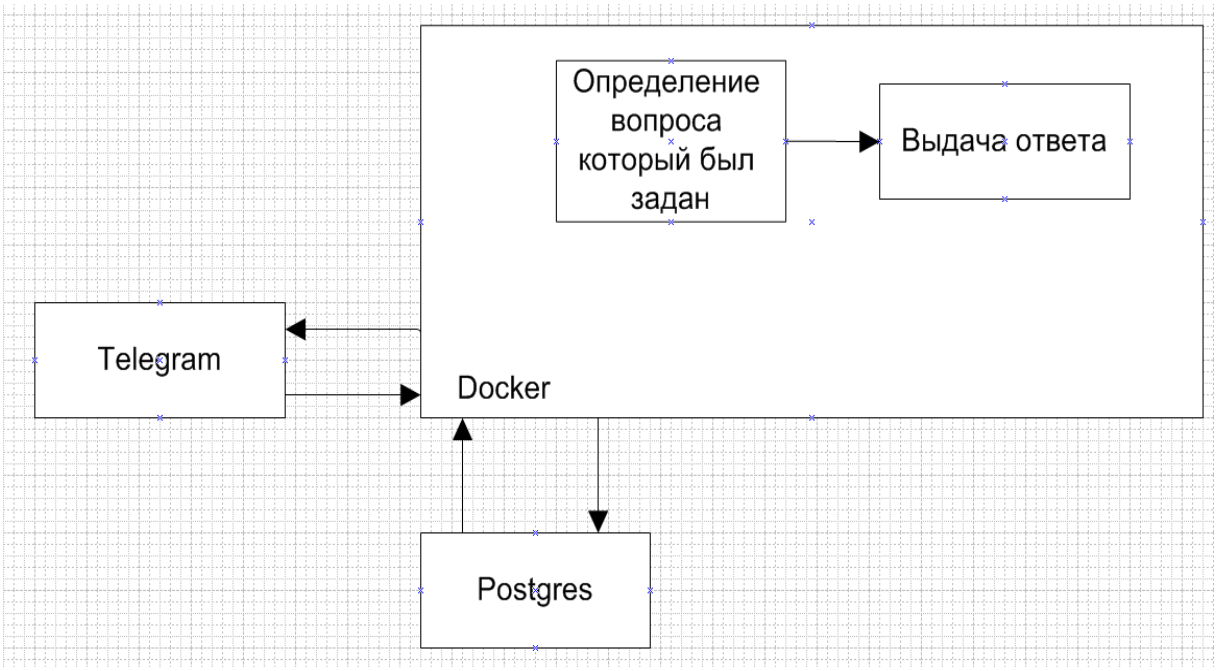


Рисунок 1 - Моделирование работы бота как было

Помимо удобства взаимодействия контейнеров между собой, docker обеспечил возможность без каких-либо проблем разворачивать данный проект на любом сервере, а также переносить его с сервера на сервер. Помимо этого, он снизил порог входа при разработке как за счёт удобства в поднятии проекта на любой виртуальной машине, исполняющей роль сервера за счёт только одной команды docker-compose up --build.

Так же изоляция контейнеров между собой позволила значительно снизить риски при дальнейшей разработке, за счёт того, что можно было не беспокоиться о сохранности базы данных, так как даже в случае катастрофических проблем при сборке в контейнере чат-бота, это бы не повлияло на базу данных.

Вывод

В рамках главы 1 была рассмотрена предметная область, связанная с цифровизацией и автоматизацией процессов на предприятиях. Были выявлены основные проблемы, с которыми сталкиваются организации при управлении данными и мониторинге работы, такие как недостаточная производительность существующих систем и ограниченный функционал.

Заказчиком был представлен запрос на разработку и внедрение чат-бота для центра цифровых компетенций Тольяттинского Государственного Университета с целью облегчения работы колл-центра в периоды повышенной нагрузки, особенно во время поступления абитуриентов и первых месяцев учебы. Чат-бот предназначен для ответа на типовые вопросы и направления пользователей к нужным ресурсам.

Описание реализации чат-бота включает в себя информацию о выбранной архитектуре, используемых технологиях (Python, aiogram, PostgreSQL), а также методах контейнеризации с использованием Docker. Этот подход обеспечивает удобство в развертывании и обслуживании приложения, а также повышает его масштабируемость и безопасность.

В целом, разработка и внедрение чат-бота для центра цифровых компетенций ТГУ является актуальной задачей, направленной на оптимизацию работы колл-центра и улучшение обслуживания пользователей.

Глава 2 Разработка нововведений для бота

2.1 Моделирование нововведений для бота

Изучив принцип работы данного бота, было выявлено, что взаимодействие пользователя с ботом происходило по следующему принципу. Пользователь, найдя бота задавал ему вопрос. Бот, получив вопрос, определял, что это за вопрос, тем самым находя ответ на него, после бот выдавал ответ пользователю. Основываясь на данном принципе, работы была составлена схема как было (рисунок 2).

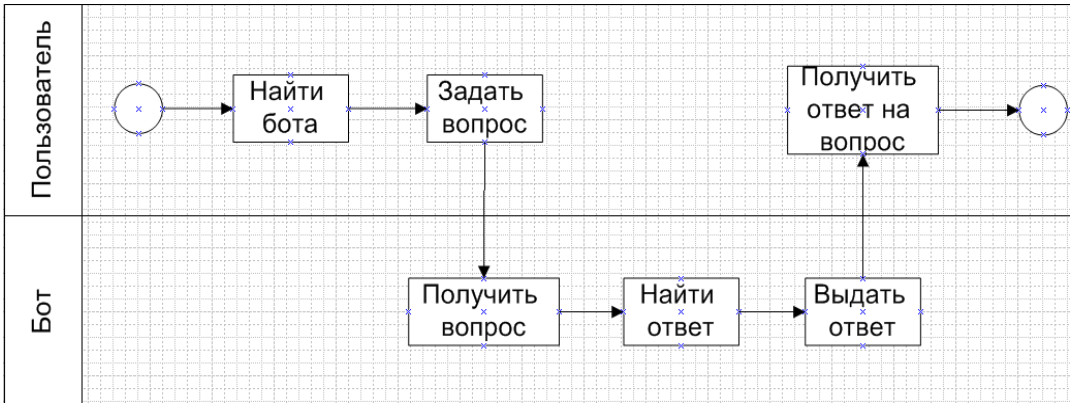


Рисунок 2 - Схема работы бота как было

Так же была составлена диаграмма последовательности (рисунок 3). Согласно которой пользователь задаёт вопрос, пот запрашивает их из базы данных, база данных выдаёт список вопросов, бот обработав их определяет какой был вопрос и выдаёт ответ.

Однако недостаток данного принципа работы заключается в том, что бот не учитывается то какие вопросы чаще задают и нет системы мониторинга для управления данными.

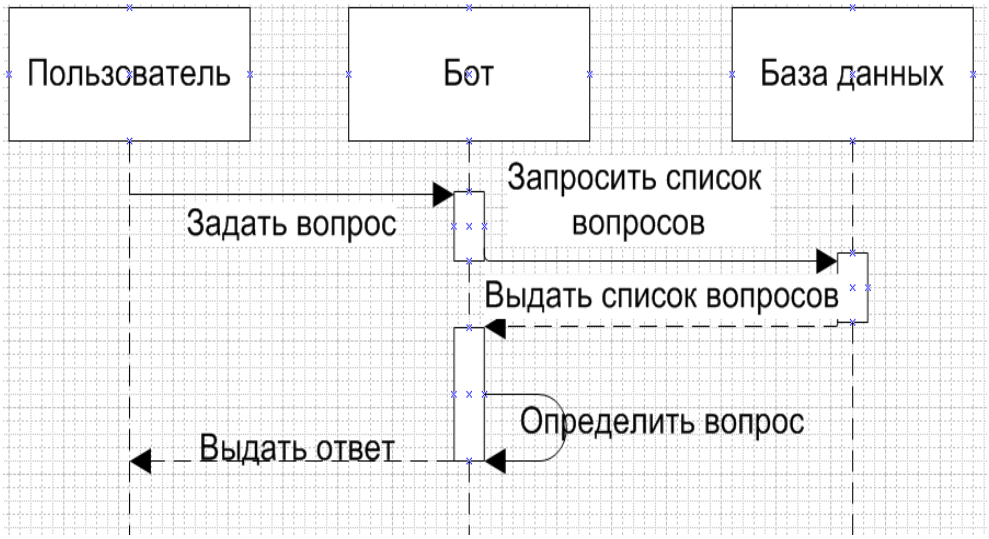


Рисунок 3 - Диаграмма последовательности

Для будущего внедрения в данного бота отдельной системы мониторинга и управления данными было решено, что взаимодействие с ботом должно происходить следующим образом (рисунок 4).

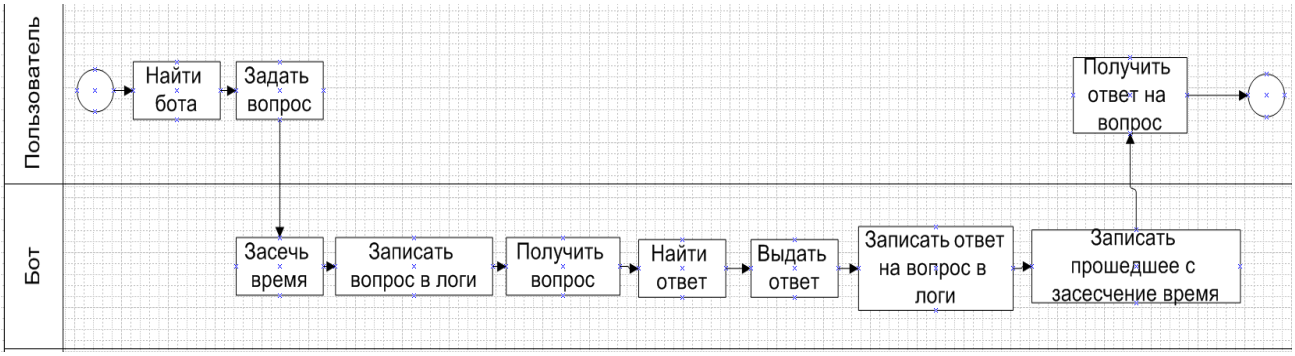


Рисунок 4 - Схема работы бота как должно быть

Пользователь задает вопрос, прежде чем искать ответ, бот засекает время, записывает заданный вопрос в логи, как и ранее ищет ответ на заданный вопрос, выдает ответ, записывает ответ на вопрос, а также засеченное время в логи.

Для того чтобы собирать и анализировать данные необходимо в первую очередь добавить в существующий принцип работы хранение собранных данных и систему для их анализа. Бот работает в Docker контейнере, взаимодействует с базой данных, в роли которой выступает postgres, так же взаимодействует с telegram, выдавая ответ пользователю. Внутри docker контейнера бот сначала определяет какой вопрос был задан и выдает на него ответ. [2], [10], [13], [28], [29], [31]

После модернизации планируется, что проект будет выглядеть как указанно на рисунке 5.

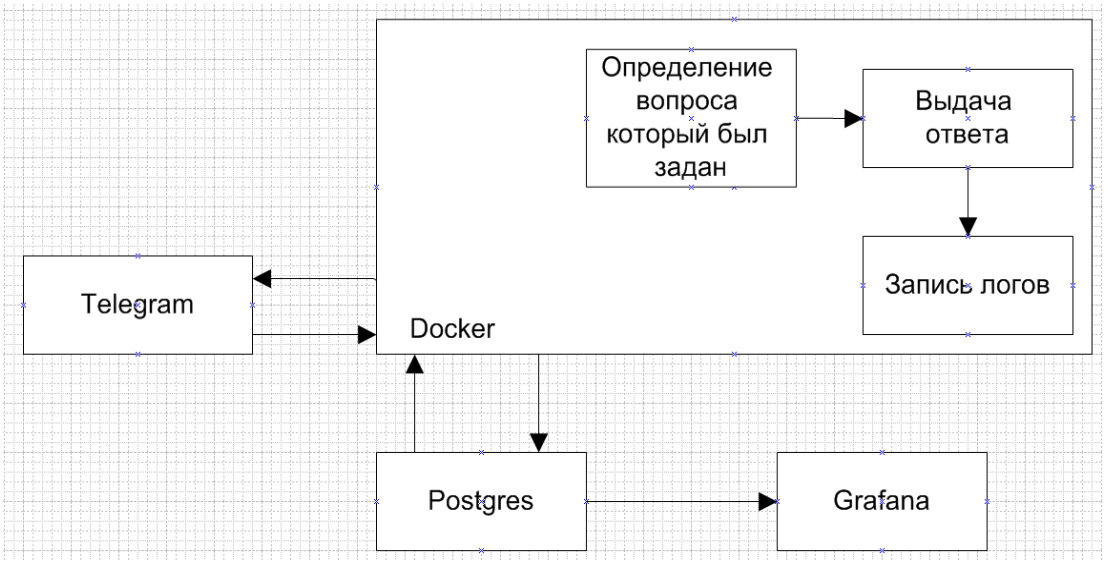


Рисунок 5 - Моделирование работы бота как должно быть

В качестве базы данных предполагается использовать реляционную базу данных, в которой имеется опыт и желательна которая уже использовалась в системе, под эти условия в данном случае подходит postgresql (однако это не значит, что при необходимости невозможно заменить postgres на любую другую реляционную базу данных).[4], [14] Ранее эта база данных была использована для хранения ответов, которые надо выдавать на вопросы, задаваемые

пользователями. А для реализации поставленной задачи, было решено добавить систему построения графиков grafana. [1], [3], [5], [7], [8], [17], [20], [27], [30] Связанно это с тем, что в отличие от его основного конкурента kibana, у него источником данных может являться что угодно, в том числе и используемая нами база данных postgres, в то время как для kibana в качестве источника данных может выступать только Elasticsearch из стека ELK. [6], [18], [19], [20], [24], [26] Так же в саму систему ответа на вопросы необходимо добавить запись логов. На основании этих логов и будут строиться графики.

Однако перед реализацией данных изменений, было необходимо построить диаграмму классов. Изначально в чат боте был класс вопросов с ответом. В данном классе хранился вопрос, ответ на него, а также cron id для обновления динамических ответов (рисунок 6).

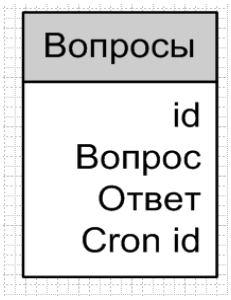


Рисунок 6 – Диаграмма классов как было

Однако для работы системы мониторинга и управления данными, нам необходимо собирать всю имеющуюся информацию о процессе ответа на вопрос. Поэтому на диаграмму необходимо добавить так же класс лога сессии, который будет ссылаться на вопросы который задает пользователь, а также на пользователя, который задает вопрос. Помимо этого, необходимо записывать лог всех сообщений пользователя (рисунок 7).

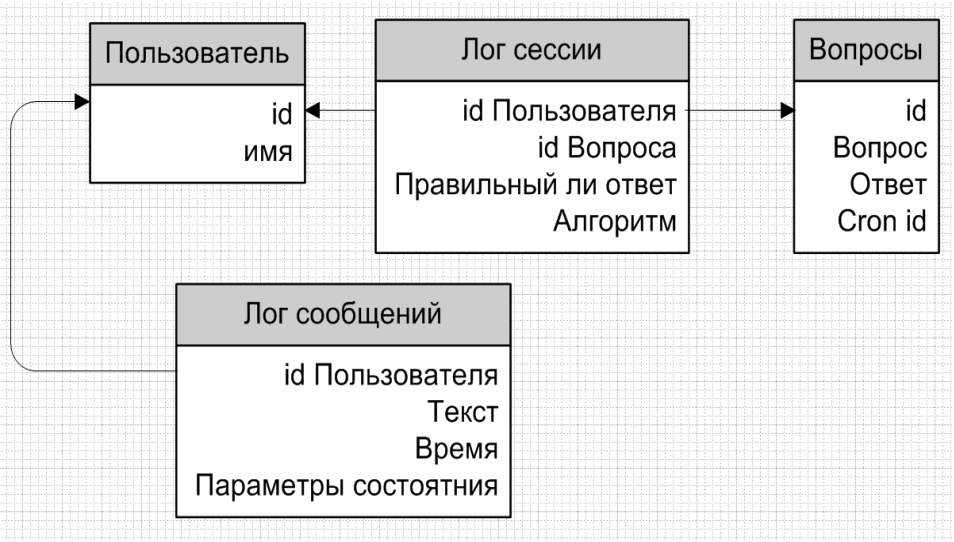


Рисунок 7 – Модель данных

В результате была спроектирована модель данных, которая за счёт того, что удовлетворяет требованиям 3 нормальной формы, позволяет удобно взаимодействовать с данными, а также потенциально добавить новые без существенны изменений.

2.2 Программная реализация нововведений для бота

При реализации работа с базой данных происходила через библиотеку для python psycopg2. [13], [15], [21], [25] Для удобства работы был использован паттерн DAO, согласно которому создается класс, содержащий CRUD методы для конкретной сущности. Для удобства работы и не нарушения паттерна DRY был создан базовый класс DAO, единый для всех будущих сущностей, в нем была прописана внутренняя функция для считывания данных, данная функция позволяет не просто считать данные из конкретного класса, но и поддерживает join с другими классами и фильтры (рисунок 8).

```python
class BaseDAO:
    async def _generate_select(self, joins: Optional[List[Tuple]] = (), **filters) -> sql.Select:
        """
        :param filters: fields to filter

        :return: SqlAlchemy Select object
        """
        to_select_models = {self.model, *chain(*[join[:-1] for join in joins])}
        to_select = []
        for model in to_select_models:
            to_select.extend(
                [
                    field.label('_'.join((model.__tablename__, field.description)))
                    for field in list(model.__table__.columns)
                ]
            )
        fields_to_filter = []
        for field_name, field_value in filters.items():
            if isinstance(field_value, tuple):
                model, value = field_value
                fields_to_filter.append(getattr(model, field_name) == value)
            else:
                fields_to_filter.append(getattr(self.model, field_name) == field_value)
        query = sql.select(to_select)
        if joins:
            to_join = sql
            for join_args in joins:
                to_join = to_join.join(*join_args)
            query = query.select_from(to_join)
        return query.where(sql.and_(*fields_to_filter))
```

Рисунок 8 – Базовый класс DAO для чтения данных

Так же были написаны единые и универсальные функции в базовом классе DAO для создания записей в классах в условиях транзакции и без транзакции (рисунок 9).

Для считывания данных был прописан ряд функций в базовом классе DAO, для различных ситуаций, которые в будущем могут потребоваться. Функция, которая ищет запись по заданным параметрам и выдает ее если ее нет, так называемый get or create. И так же 2 функции для считывания одиночной записи и множества записей (рисунок 10).

```python
async def create(self, **fields) -> Record:
    query = sql.insert(self.model).returning(
        *[
            column.label('_'.join((self.model.__tablename__, column.description)))
            for column in self.model.__table__.columns
        ]
    ).values(**fields)
    return await pg.fetchrow(query)

async def trans_create(self, conn, **fields) -> Record:
    query = sql.insert(self.model).returning(
        *[
            column.label('_'.join((self.model.__tablename__, column.description)))
            for column in self.model.__table__.columns
        ]
    ).values(**fields)
    return await conn.fetchrow(query)
```

Рисунок 9 - Функции в базовом классе DAO для записи данных в отдельной транзакции и без нее

```python
async def get_or_create(self, joins: Optional[List[Tuple]] = (), **fields) -> Record:
    record = await self.get(joins, **fields)
    if not record:
        record = await self.create(**fields)

    return record

async def get(self, joins: Optional[List] = (), **fields) -> Record:
    """
    :param joins: list of joins
    :param fields: fields to filter

    :return: Record object
    """
    query = await self._generate_select(joins=joins, **fields)
    return await pg.fetchrow(query)

async def get_many(self, joins: Optional[List] = (), **fields) -> List[Record]:
    """
    :param joins: list of joins
    :param fields: fields to filter

    :return List of Record objects
    """
    query = await self._generate_select(joins=joins, **fields)
    return await pg.fetch(query)
```

Рисунок 10 - Функции в базовом классе DAO получения данных

Так же были прописаны функции в базовом классе DAO, на обновление данных по id и удалению их (рисунок 11).

```python
async def update_by_id(self, record_id, **fields) -> Record:
    """
    :param record_id: database record ID
    :param fields: Fields to update

    :return SqlAlchemy model record
    """

    query = sql.update(self.model).returning(self.model.id).where(self.model.id == record_id).values(fields)
    return await pg.fetchrow(query)

async def delete_by_id(self, record_id: int) -> None:
    """
    :param record_id: database record ID
    """
    query = sql.delete(self.model).where(self.model.id == record_id)
    return await pg.fetchrow(query)
```

Рисунок 11 - Функции в базовом классе DAO для обновления и удаления данных по id

Для работы ряда функций базового класса DAO требуется уникальный id, в связи с этим было решено так же создать базовый класс, в котором будет прописан id данной записи и время, когда данная запись была создана(рисунок 12).

```python
@as_declarative()
class Base:
    """
    The Base class is used as a base for other classes.
    Contains:
        record id
        record creation time
    """
    @declared_attr
    def __tablename__(self):
        return self.__name__

    id = Column(Integer, primary_key=True)
    created = Column(DateTime, default=datetime.now, server_default='NOW()')
```

Рисунок 12 - Базовый класс

В классе пользователя от наследованном от базового класса было прописано только имя пользователя и его id в телеграмме(рисунок 13).

```python
class User(Base):
    """
    User class for recording all users of the bot.
    Contains:
        tg_id - user id in the telegram
        name - how the bot will refer to the user
    """
    tg_id = Column(Integer, nullable=False)
    name = Column(String, nullable=False)
```

Рисунок 13 - Класс пользователя

В классе DAO пользователя помимо уже прописанных функций была написана функция для информации о пользователе основываясь на его telegram id. Поскольку мы заранее знаем, что этот id является уникальным для каждого пользователя и не может повторяться нам будет удобнее обновлять записи основываясь на нем, нежели по id записи, это сократит количество обращений к базе данных(рисунок 14).

```python
class UserDAO(BaseDAO):
    def __init__(self):
        self.model = User

    async def update_by_tg_id(self, tg_id: int, **fields) -> Record:
        """
        :param tg_id: Telegram ID of the user
        :param fields: Fields to update

        :return User record
        """
        query = sql.update(self.model).returning(self.model.id).where(self.model.tg_id == tg_id).values(fields)
        return await pg.fetchrow(query)
```

Рисунок 14 – Классе DAO пользователя с уникальной функцией обновления данных по tg_id

В свою очередь класс ответов на вопросы пользователей можно обобщить до структуры, где прописан вопрос, который задается, ответ на него и код по которому производится автоматическое обновление ответа по расписанию(рисунок 15).

```
class QuestionAnswer(Base):
    question = Column(String, nullable=False)
    answer = Column(String, nullable=False)
    cron_id = Column(String, unique=True)
```

Рисунок 15 - Класс вопросов с ответом

Далее в боте необходимо добавить место куда в будущем будет производиться запись всех сообщений пользователей, их ответы на контрольные вопросы, а также логи ответов на вопросы. Первым делом было решено добавить ответы на контрольные вопросы. Заказчик указал что в качестве контрольных вопросов должен быть вопрос про то прошел ли пользователь регистрацию благодаря боту, а также его оценка сервиса. Поэтому в классе пользователя были добавлены колонки под ответ на соответствующий вопрос, а также дата, когда был дан ответ на этот вопрос(рисунок 16).

```
class User(Base):
    tg_id = Column(Integer, nullable=False)
    name = Column(String, nullable=False)
    group_id = Column(Integer, ForeignKey('GroupITs.id'), nullable=True)

    rate_service = Column(Integer, nullable=True)
    rate_service_created = Column(DateTime, nullable=True)
    able_to_register = Column(Boolean, nullable=True)
    able_to_register_created = Column(DateTime, nullable=True)
```

Рисунок 16 – Дополненный класс пользователя

Класс для сохранения всех сообщений помимо информации о том, кто написал сообщение и какое сообщение было написано, хранится так же и время, за которое пользователь получил ответ на свое сообщение, параметр состояния и название функции. Время ответа мы сохраняем для анализа скорости работы наших функций и в случае проблем со скоростью ответа лучшего понимания того привело к ним, скорость интернета или сами функции. Название функции предполагается сохранять на случай необходимости проанализировать чем пользователи чаще всего пользуются, а также для отладки работы функций в случае неполадок. Для отладки будут сохраняться параметры состояния, это уникальный параметр для функции ответа на вопросы и сохраняется он для понимания того, что могло привести к проблемам и какие внутренние параметры были на момент ответа на сообщения(рисунок 17).

```
class LogAllMessage(Base):
    user_id = Column(Integer, ForeignKey('User.id'), nullable=False)
    text = Column(String, nullable=False)
    time_answer = Column(Float, nullable=False, default=-1.0)
    params_state = Column(String, nullable=True)
    func_name = Column(String, nullable=False)
```

Рисунок 17 - Класс логов всех сообщений

В свою очередь класс для сохранения ответов на заданные пользователем вопросы предполагается, что должна сохранять id пользователя, которому был дан ответ, алгоритм, который дал ответ (в нашем случае их несколько), был ли дан ответ на правильный вопрос и ответ на какой вопрос был дан.(рисунок 18)

```python
class SessionLog(Base):
    user_id = Column(Integer, ForeignKey('User.id'), nullable=False)
    algorithm = Column(String, nullable=False)
    successful = Column(Boolean, nullable=True)
    questions_id = Column(Integer, ForeignKey("QuestionAnswer.id"), nullable=False)
```

Рисунок 18 – Класс лога сессии

Для данного класса помимо стандартного DAO была прописана уникальная функция, которая позволяет на основании ранее данных алгоритмом ответов вычислить его "рейтинг" достоверности его ответа на вопрос. Данный рейтинг рассчитывается как сумма нелинейных рейтингов для каждого пользователя чтобы избежать умышленного занижения рейтинга каким-то пользователем. А также функция подсчета числа вопросов, на который был дан ответ конкретному пользователю(рисунок 19).

```python
class SessionLogDAO(BaseDAO):
    def __init__(self):
        self.model = SessionLog

    async def get_count_questions(self, user_id) -> Record:
        query = sql.select([func.count(self.model.id)]).where(self.model.user_id == user_id)
        return await pg.fetchrow(query)

    async def get_rating(self, questions_id, algorithm_id) -> Record:
        return await pg.fetchrow(text(f"""select r.algorithm,  sum(3/(r.row_number)) as rating  from
                            (SELECT s.algorithm,
                            ROW_NUMBER() OVER(PARTITION BY s.user_id, s.successful ORDER BY s.created)*CASE
                            WHEN s.successful THEN 1.0 ELSE -1.0 END AS row_number
                            FROM public."SessionLog" as s
                            where s.questions_id={questions_id} and s.successful is not null and s.algorithm='{algorithm_id}') as r
                            group by r.algorithm
                            order by rating desc"""))
```

Рисунок 19 - DAO лога всех сообщений с уникальными функциями на число вопросов и рейтинг вопросов

После добавления в боте возможности на хранение данных необходимо добавить возможность для дальнейшего анализа собранных данных. В качестве

системы для анализа поступающих данных было решено использовать grafana, по той причине, что в отличие от его основного конкурента kibana у него источником данных может являться что угодно, в том числе и используемая нами база данных postgres, в то время как для kibana в качестве источника данных может выступать только Elasticsearch из стека ELK(рисунок 20).

```yaml
grafana:
  image: grafana/grafana
  user: root
  ports:
    - 3000:3000
  volumes:
    - ./grafana:/var/lib/grafana
    - ./grafana/provisioning/:/etc/grafana/provisioning/
  container_name: grafana
  hostname: grafana
  depends_on:
    - database
  restart: always
```

Рисунок 20 - Добавление в docker-compose grafana

Таким образом в docker-compose файл мы добавили grafana основываясь на одноимённом образе. Для её корректной работы необходимо указать порт соединённый с 3000 портом внутри, так же для того чтобы происходило сохранение файлом и было возможно передать построенные графики не передавая образ, был соединён volume grafana с папками внутри проекта и для того чтобы grafana не падала от того, что не может получить ответ от базы данных до того как она поднялась, мы указали depends_on.

2.3 Реализация сбора данных

В созданные классы нам необходимо записывать данные для дальнейшего анализа из telegram. [6], [15], [16], [21], [23], [25] Первым делом было решено сохранять логи всех сообщений. Для сохранения логов был написан декоратор. Данный декоратор берет telegram id пользователя, написавшего сообщение. Если по данному id есть пользователь, то мы отмечаем, что этот пользователь активен(если было указано обратное), записываем время для дальнейшего расчета времени работы и запускаем функцию, которая была обернута декоратором. После того как функция отрабатывает рассчитываем сколько времени прошло и записываем в класс все полученные данные включая то какая функция была вызвана(рисунок 21).

```
def log_decorator(func):
    async def wrapper(message, state, *args, **kwargs):
        id_user = message.from_user.id
        user = await UserDAO().get(tg_id=id_user)
        if user:
            if not user['User_is_active']:
                await UserDAO().update_by_tg_id(tg_id=id_user, is_active=True)
            time_start = time()
            result = await func(message, state, *args, **kwargs)
            async with state.proxy() as data:
                params_state = str(data)
            await LogAllMessageDAO().create(user_id=user['User_id'],
                                            text=message.text,
                                            time_answer=time()-time_start,
                                            params_state=params_state,
                                            func_name=func.__name__)
            return result
    return wrapper
```

Рисунок 21 - Декоратор для записи логов

С определенным шансом при ответе на вопросы, после ответа, задается вопрос правильно ли ответили и запускается функция выставления рейтинга. При этом пользователю будет предложено ответить вариантами да/нет на то правильно ли поняли вопрос. При соответствующих ответах происходит запись в бд, иначе если данный пользователем ответ не является ожидаемым нами, мы предполагаем, что пользователь проигнорировал наш вопрос и хочет продолжить задать свои вопросы, в таком случае мы ничего не записываем, а переходим снова в состояние определение вопрос и ответа на него(рисунок 22, 23).

```python
@rate_limit(limit=1)
@mat_filter
@log_decorator
async def wait_for_rating(message: Message, state: FSMContext, raw_state: Optional[str] = None):
    id_user = message.from_user.id
    user = await UserDAO().get(tg_id=id_user)
    if message.text.lower() == 'да':
        async with state.proxy() as data:
            answer_to_rate = data['answer_to_rate']
        await SessionLogDAO().create(user_id=user['User_id'],
                                     algorithm=answer_to_rate.algorithm_key,
                                     successful=True,
                                     questions_id=answer_to_rate.question_id)
        await message.reply(random_answer.get_rand_val('guess_answer', int(time())),
                            reply_markup=ReplyKeyboardRemove())
        await state.finish()
    elif message.text.lower() == 'нет':
        async with state.proxy() as data:
            answer_to_rate = data['answer_to_rate']
        await SessionLogDAO().create(user_id=user['User_id'],
                                     algorithm=answer_to_rate.algorithm_key,
                                     successful=False,
                                     questions_id=answer_to_rate.question_id)
        await message.reply(random_answer.get_rand_val('couldnt_guess_answer', int(time())),
                            reply_markup=ReplyKeyboardRemove())
        await state.finish()
    else:
        await state.finish()
        await wait_for_question(message, state, raw_state)
```

Рисунок 22 - Функция оценивания ответа на вопрос

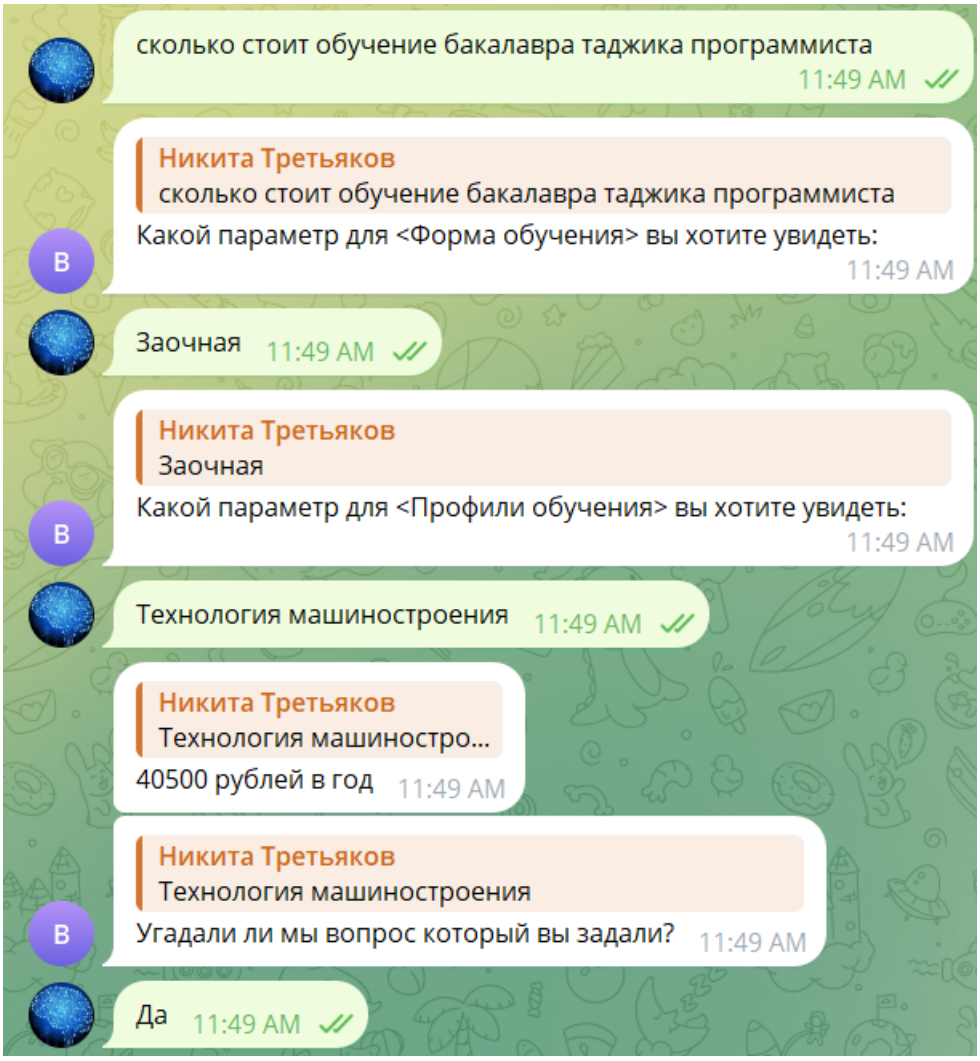


Рисунок 23 - Работа функции выдачи рандомного вопроса при оценивании ответа на вопрос

Аналогичным образом(рисунок 24) с каким-то шансом выдавались вопросы про то готов ли пользователь порекомендовать данного бота(рисунок 26, 27) и смог ли он пройти регистрацию(рисунок 25), с различием, что перед вопросом о прохождении регистрации он должен задать как минимум 5 вопросов.

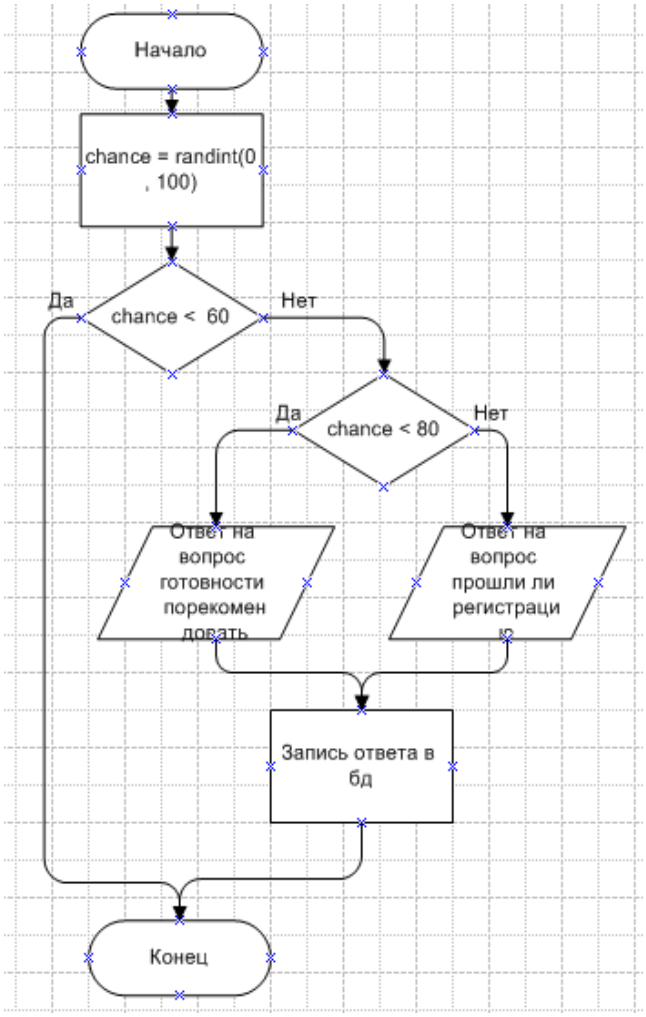


Рисунок 24 - Блок-схема рандомизации выдачи вопроса об оценке ответа

```python
@rate_limit(limit=1)
@mat_filter
@log_decorator
async def wait_for_able_to_register(message: Message, state: FSMContext, raw_state: Optional[str] = None):
    answers = {'да': True, 'нет': False}
    text = filter_punctuation(message.text)
    if text in answers.keys():
        await UserDAO().update_by_tg_id(tg_id=message.from_user.id,
                                        able_to_register=answers[text],
                                        able_to_register_created=datetime.now())
        await message.reply(random_answer.get_rand_val('thanks_for_rate', int(time())))
        await state.finish()
    else:
        await message.reply(random_answer.get_rand_val('answer_only_yes_no', int(time())))
```

Рисунок 25 - Функция ответа на вопрос “удалось ли зарегистрироваться”

```python
@rate_limit(limit=1)
@mat_filter
@log_decorator
async def wait_for_rate_service(message: Message, state: FSMContext, raw_state: Optional[str] = None):
    if message.text.isdigit():
        rate = int(message.text)
        if 1 <= rate <= 10:
            await UserDAO().update_by_tg_id(tg_id=message.from_user.id,
                                            rate_service=rate,
                                            rate_service_created=datetime.now())
            await message.reply(random_answer.get_rand_val('thanks_for_rate', int(time())))
            await state.finish()
        else:
            await message.reply(random_answer.get_rand_val('answer_only_1_10', int(time())))
    else:
        await message.reply(random_answer.get_rand_val('answer_only_1_10', int(time())))
```

Рисунок 26 - Функция оценки сервиса

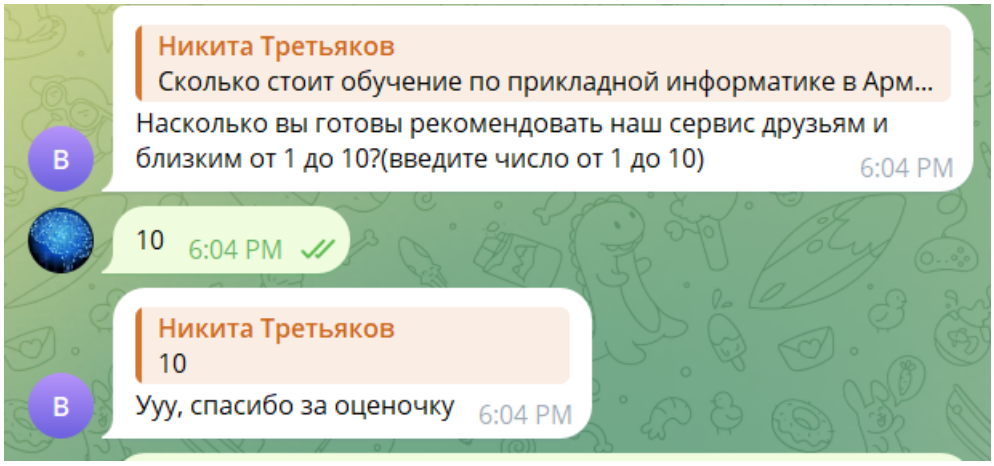


Рисунок 27 - Работа функции по оценке сервиса

Таким образом была реализована логика частичного опроса пользователей на значимые для заказчика вопросы. Данная логика позволит не только безболезненно добавить новые опросы или заменить какие-то из текущих, но также и потенциально поменять процент опрашиваемых на каждом из опросов не прерывая работу бота.

2.4 Реализация дашбордов

Как только нами были собраны данные, в течение нескольких месяцев, было решено приступить к анализу этих данных и построению по ним графиков и дашбордов, которые бы выводили нужную нам информацию.

Перед построением графиков нам необходимо подготовиться к особенностям среды grafana. Первым делом стоит учесть, что стандартные возможности, предоставляемые в правом верхнем углу, лишь урезают предоставляемые данные для графиков, однако они не дают группировать или использовать границы для своих запросов вне графиков, например в таблицах(рисунок 28).

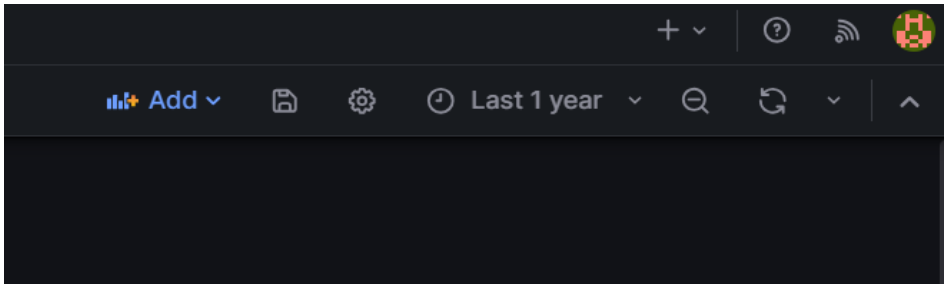


Рисунок 28 - Встроенный выбор среза данных в grafana

Таким образом, если использовать только их, то мы не будем иметь возможность выбирать промежуток для группировки данных, а также в будущем при необходимости выставления урезанных таблиц мы не сможем ничего сделать. В связи с этим было решено добавить параметры для пользовательского ввода, в виде выбора промежутка, за который идет группировка данных, а также даты начала и окончания(рисунок 29).

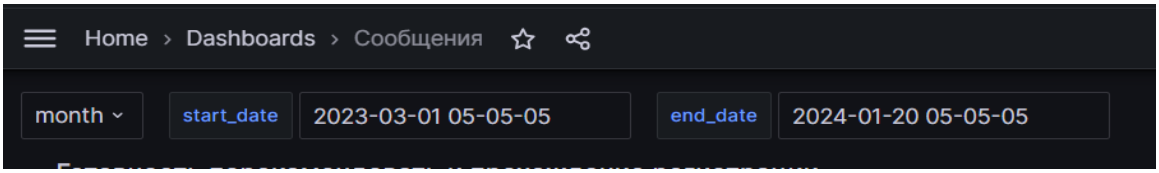


Рисунок 29 - Добавленный выбор среза данных и группировки в grafana

Данные параметры добавляются в настройках всего дашборда в переменных(рисунок 30).

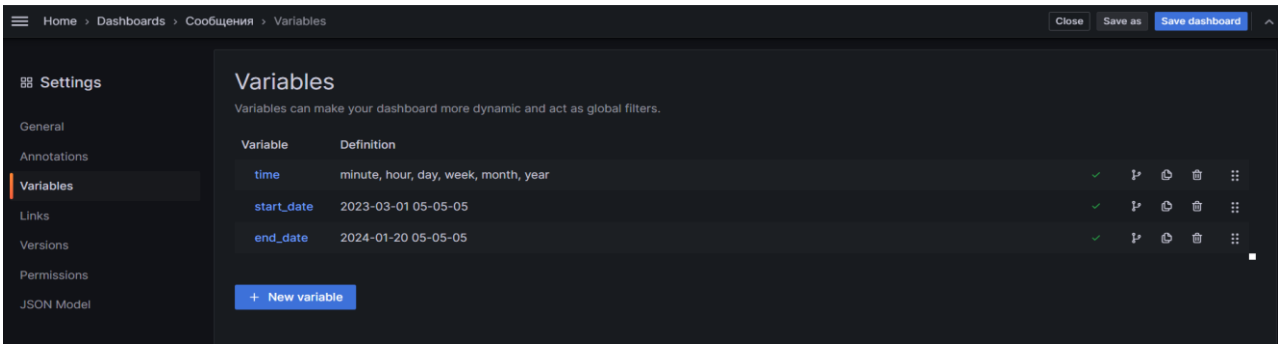


Рисунок 30 - Реализация выбора среза данных и группировки

В связи с большим количеством классов, а как следствие и большим количеством графиков, было решено разделить их по группам, данные о которых они выдают. А именно Готовность порекомендовать и прохождение регистрации (для выведение статистической информации о параметрах, которые попросил добавить и спрашивать заказчик), время ответа (для контроля за скоростью работы бота и контролем за нештатными ситуациями), верность ответа (для контроля за тем как отвечает каждый алгоритм на заданные вопросы) и пользователи (для сбора общей информации о количестве пользователей и сообщений от них)(рисунок 31).

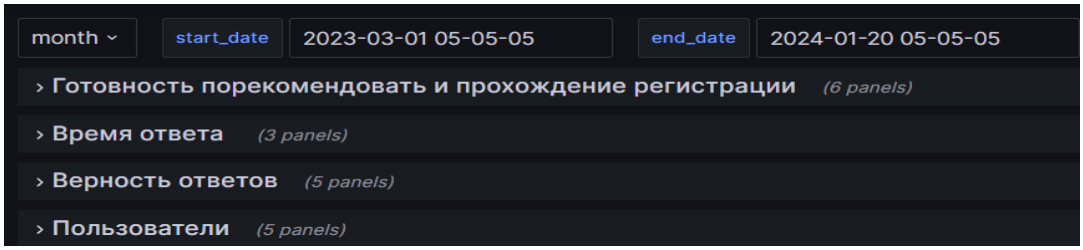


Рисунок 31 - Группы дашбордов

По собранным данным о готовности порекомендовать, было решено построить 4 графика.

Средний рейтинг готовности порекомендовать за промежуток времени и за все время. Данные графики необходим для понимания того, как в каждом из промежутков отвечали пользователи о готовности порекомендовать данный сервис или как менялся средний ответ о готовности порекомендовать сервис(рисунок 32).

Рисунок 32 - Дашборд рейтинга готовности порекомендовать за промежуток времени

Данные для этого графика мы получаем, указав sql для нашей postgresql базы данных. [11], [12], [16], [22] В данном запросе мы берем из класса

пользователей, где указан рейтинг среднее значение рейтинга, преобразуем дату к указанному ранее селектору для группировки, а все полученные данные группируем по дате(рисунок 33, 34).

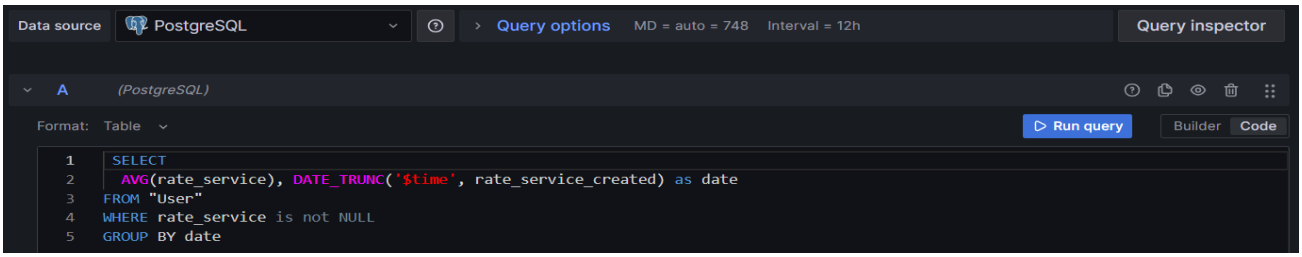


Рисунок 33 - Реализация дашборда рейтинга готовности порекомендовать за промежуток времени

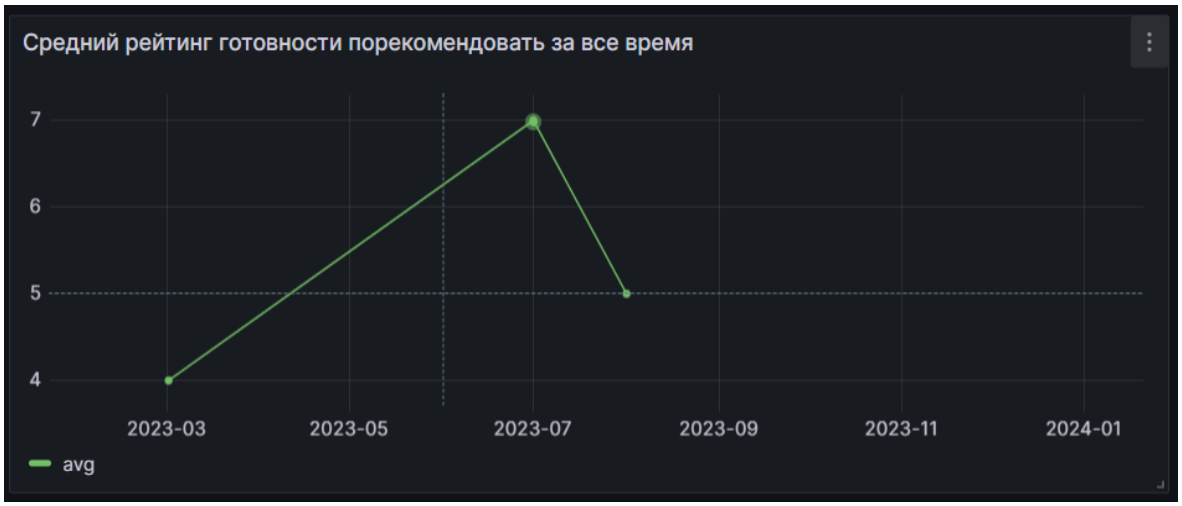


Рисунок 34 - Дашборд среднего рейтинга готовности порекомендовать за все время

В данном запросе мы берем из класса пользователей, где указан рейтинг среднее значение рейтинга, а также сортируем данные по дате предварительно преобразовав дату к указанному ранее селектору для группировки(рисунок 35).

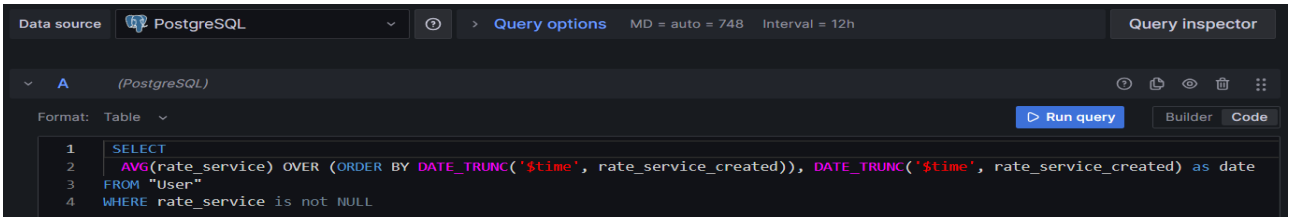


Рисунок 35 - Реализация дашборда среднего рейтинга готовности порекомендовать за все время

Так же было решено сделать свечи рейтинга готовности порекомендовать. График свечей уникален тем, что позволяет на одном графике отобразить, где находится минимум данных, максимум, а также границы 25 и 75 процентов данных. Они необходимы нам для понимания распределения ответов пользователей и потенциального выявления завышения/занижения оценок(рисунок 36).

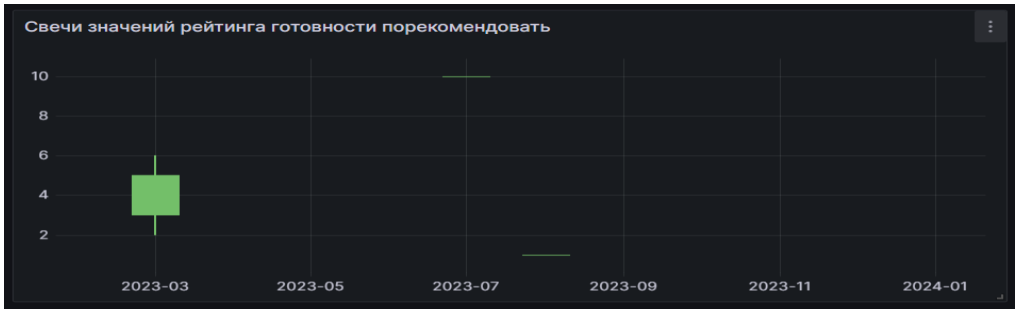


Рисунок 36 - Дашборд свечей значений рейтинга готовности порекомендовать

В запросе для данного графика мы из пользователей, где пользователь указал рейтинг, берем дату, приведенную к селектору даты, а также каждую из необходимых нам для свечей точку, а именно минимум, максимум, 1 и 3 перцентиль, сгруппировав по дате(рисунок 37).

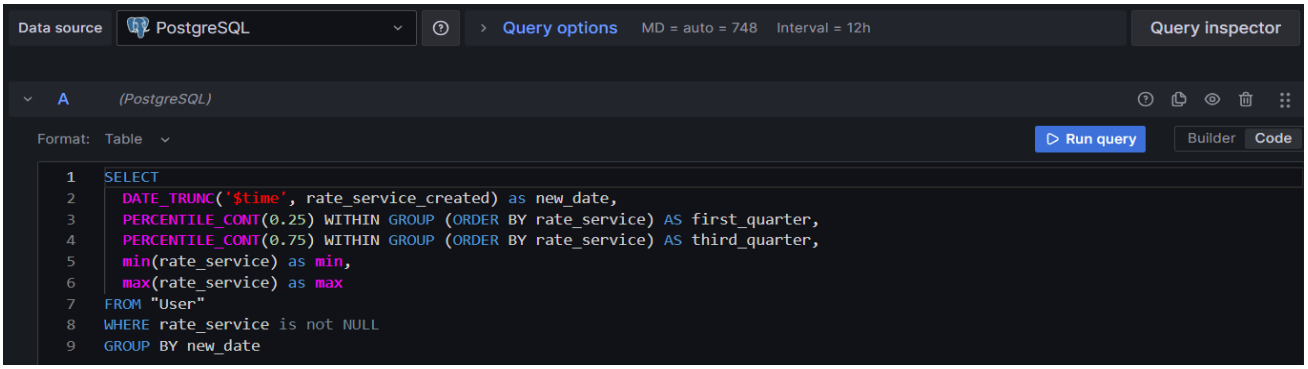


Рисунок 37 - Реализация дашборда свечей значений рейтинга готовности порекомендовать

А также процент пользователей, которые ответили на вопрос о готовности порекомендовать в процентах(рисунок 38).

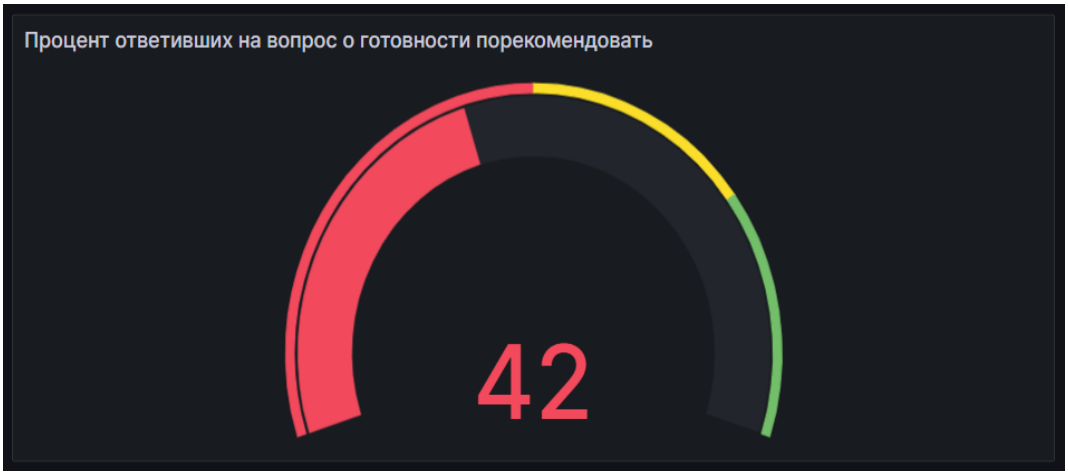


Рисунок 38 - Дашборд процента ответивших на вопрос о готовности порекомендовать

Данный рейтинг составляется просто количество не ответивших на общее количество пользователей, а чтобы при пустой базе данных рейтинг не сломался, мы добавили проверку на то есть ли вообще ответы(рисунок 39).

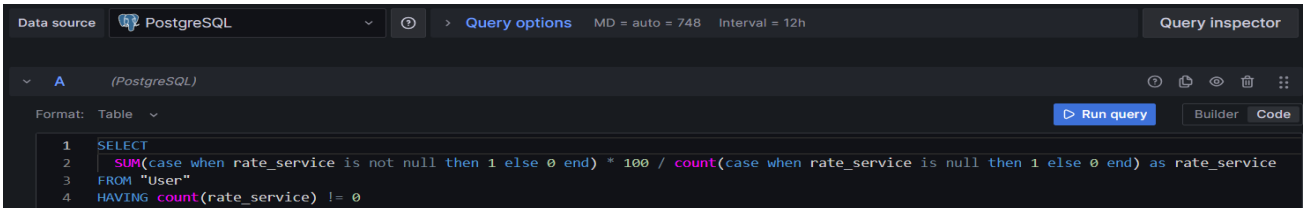


Рисунок 39 - Реализация дашборда процента ответивших на вопрос о готовности порекомендовать

Так же необходимо добавить графики для пользователей, прошедших регистрацию. Из данных ответов нас интересует скорее только то, как пользователи ответили на вопрос о том смогли ли они пройти регистрацию. Поэтому было решено сделать 2 дашборда.

Процент пользователей, прошедших регистрацию, будет показывать то, как изменялся процент пользователей, прошедших регистрацию при каждом ответе. Падение данного графика будет показывать, что к нам присоединилось больше пользователей чем количество ответивших на вопрос о прохождении регистрации, а рост наоборот(рисунок 40).

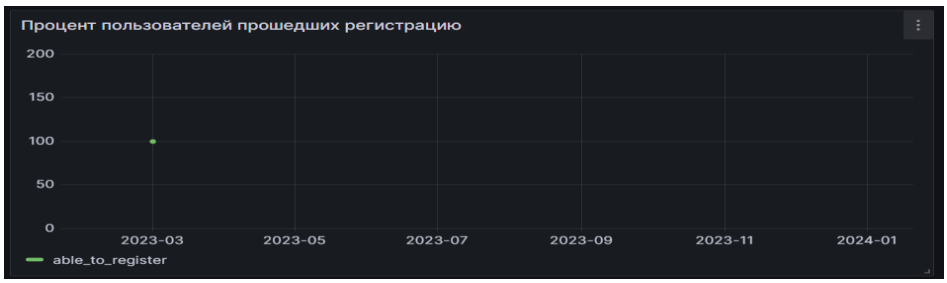


Рисунок 40 - Дашборд процента пользователей, прошедших регистрацию

Аналогично с дашбордом о готовности порекомендовать, мы составляем sql запрос, который группирует по дате и выводит суммарный процент ответивших на вопрос за время до текущего(рисунок 41, 42).

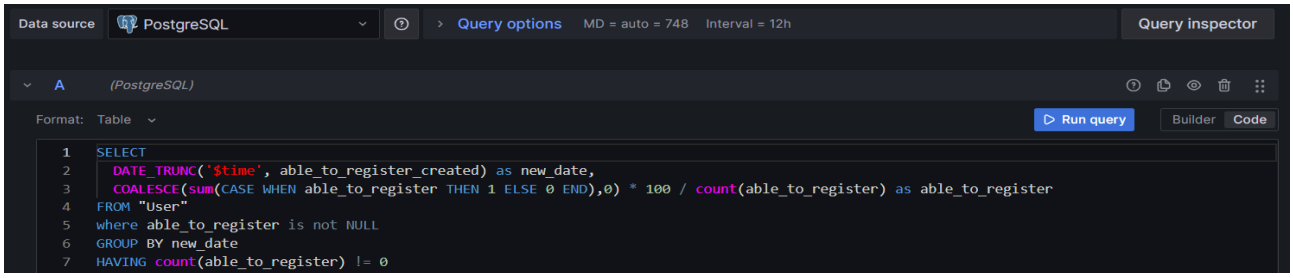


Рисунок 41 - Реализация дашборда процента пользователей, прошедших регистрацию

Рисунок 42 - Дашборд процента прошедших регистрацию

Поскольку данные о прохождении регистрации у нас в формате bool, мы преобразуем их к int и делим на общее число ответивших. Таким образом мы получаем дашборд, который показывает, как ответили люди о прохождении регистрации(рисунок 43).

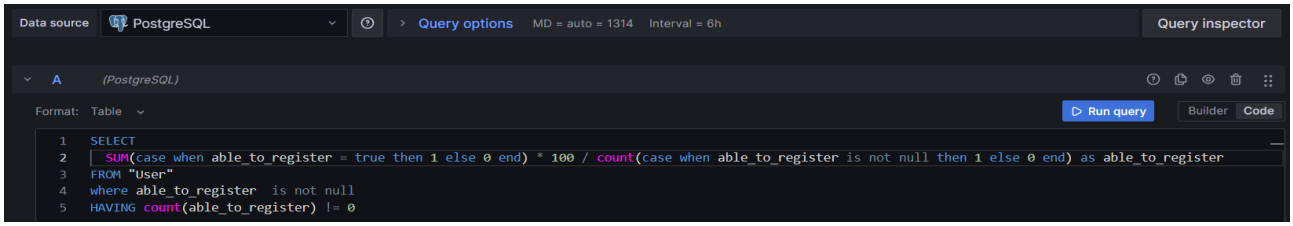


Рисунок 43 - Реализация дашборда процента прошедших регистрацию

Для того чтобы отслеживать скорость ответа бота на сообщения пользователя было решено добавить дашборд свечей времени ответа. По данному графику видно явно завышенные экстремумы, которые показывают на время, когда производилась отладка бота и она долго находилась между вопросом и ответом, однако так же видно, что третий квартиль – примерно 1 секунда. А значит, как минимум 75% ответов производилось за 1 секунду(рисунок 44, 45).

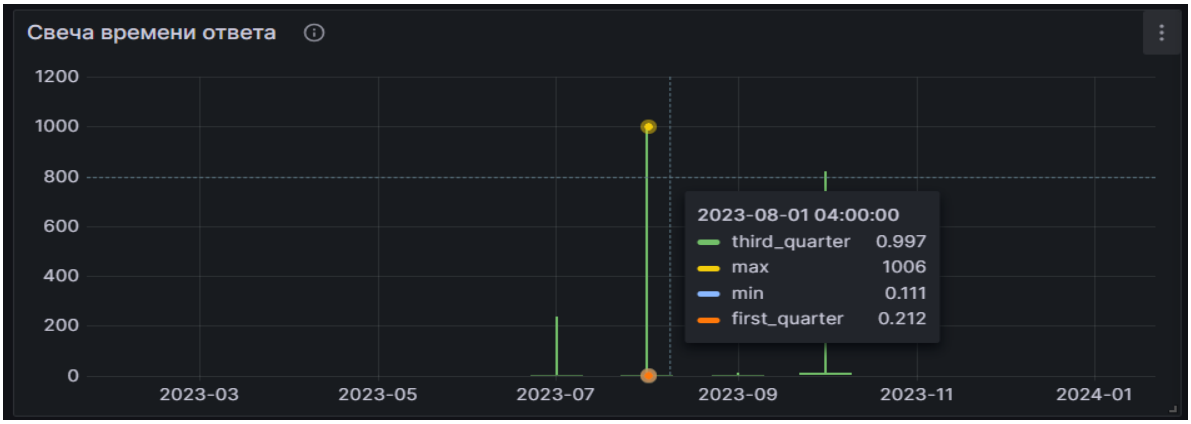


Рисунок 44 - Дашборд свечей времени ответа

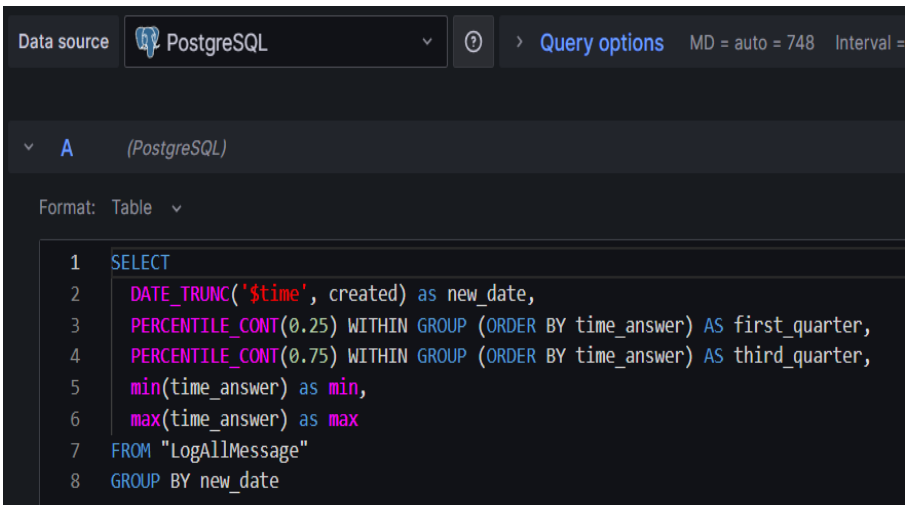


Рисунок 45 - Реализация дашборда свечей времени ответа

Для большей наглядности было решено так же добавить график среднего и медианного значения, по которому видно, что медианное время ответа как правило не превышает 4 секунд, при этом среднее время может доходить до 1.5 минут, что указывает на то, что имеется достаточно много экстремумов свойственных отладке бота(рисунок 46).

Рисунок 46 - Дашборд медианы и среднего времени ответа на вопрос

Для построения данного дашборда мы группируем данные по дате, берем среднее, а чтобы взять медиану берем 50 перцентиль сгруппировав по дате(рисунок 47).

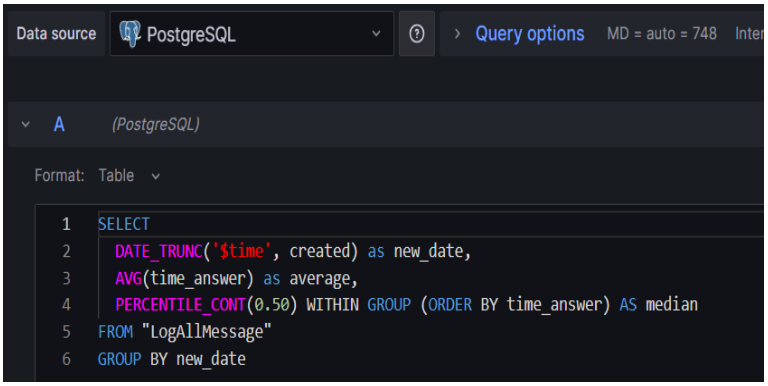
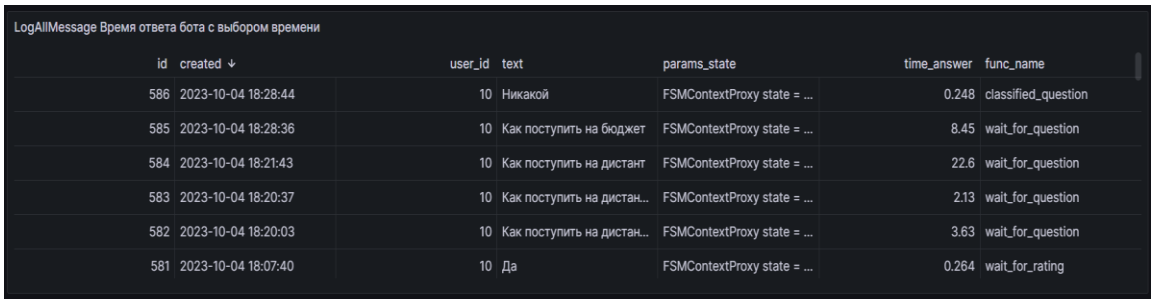


Рисунок 47 - Реализация дашборда медианы и среднего времени ответа на вопрос

Так же для более удобного отслеживания того какая скорость ответа на вопросы, от кого поступают вопросы и какие это вопросы, было решено добавить таблицу, в которой было бы отражены все необходимые данные(рисунок 48).



LogAllMessage Время ответа бота с выбором времени

id	created ↓	user_id	text	params_state	time_answer	func_name
586	2023-10-04 18:28:44	10	Никакой	FSMContextProxy state = ...	0.248	classified_question
585	2023-10-04 18:28:36	10	Как поступить на бюджет	FSMContextProxy state = ...	8.45	wait_for_question
584	2023-10-04 18:21:43	10	Как поступить на дистант	FSMContextProxy state = ...	22.6	wait_for_question
583	2023-10-04 18:20:37	10	Как поступить на дистан...	FSMContextProxy state = ...	2.13	wait_for_question
582	2023-10-04 18:20:03	10	Как поступить на дистан...	FSMContextProxy state = ...	3.63	wait_for_question
581	2023-10-04 18:07:40	10	Да	FSMContextProxy state = ...	0.264	wait_for_rating

Рисунок 48 - Таблица времени ответов в логах

Для построения данной таблицы мы просто фильтруем данные по введенным ранее в поля датам(рисунок 49).

Data source PostgreSQL | Query options MD = auto = 748 Interval = 12h | Query inspector

A (PostgreSQL)

Format: Table | Run query | Builder | Code

```
SET TIME ZONE 'UTC-4';
SELECT *
FROM "LogAllMessage"

where
  to_timestamp('$start_date', 'yyyy-mm-dd hh24:mi:ss') < to_timestamp(to_char(created ,'yyyy-mm-dd'), 'yyyy-mm-dd')  and
  to_timestamp(to_char(created ,'yyyy-mm-dd'), 'yyyy-mm-dd') < to_timestamp('$end_date', 'yyyy-mm-dd hh24:mi:ss')
```

Рисунок 49 - Реализация таблицы времени ответов в логах

Для понимания того, как работают алгоритмы бота, было решено построить графики, которые бы показывали то, как отвечают алгоритмы, а также процент их правильных ответов.

Дашборд количества ответов призван показать визуально сколько какой алгоритм отвечал(рисунок 50).

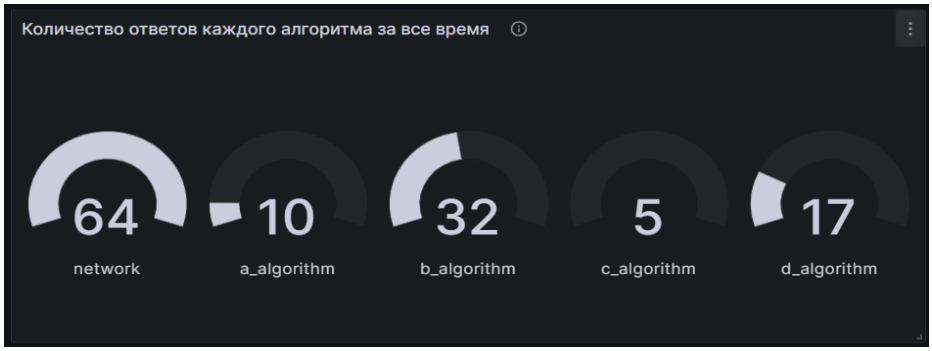


Рисунок 50 - Дашборд количества ответов каждого алгоритма за все время

Для его построения было сделано 5 запросов, каждый из которых показывает количество ответов, сделанных каждым алгоритмом(рисунок 51, 52, 53, 54, 55).

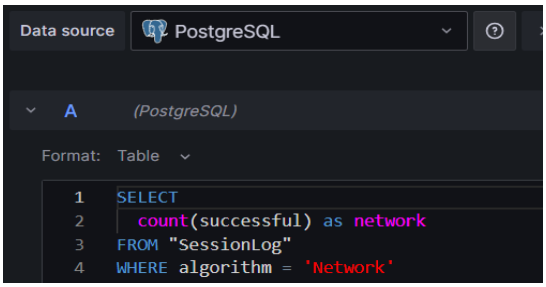


Рисунок 51 - Реализация дашборда количества ответов нейросети за все время

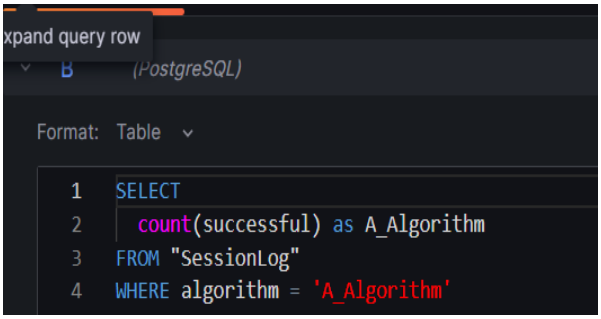


Рисунок 52 - Реализация дашборда количества ответов алгоритма А за все время

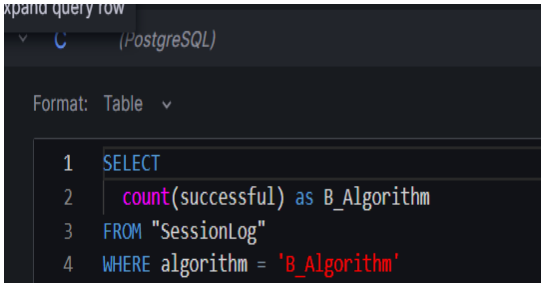


Рисунок 53 - Реализация дашборда количества ответов алгоритма B за все время

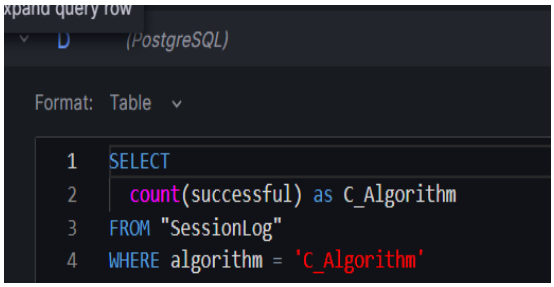


Рисунок 54 - Реализация дашборда количества ответов алгоритма C за все время

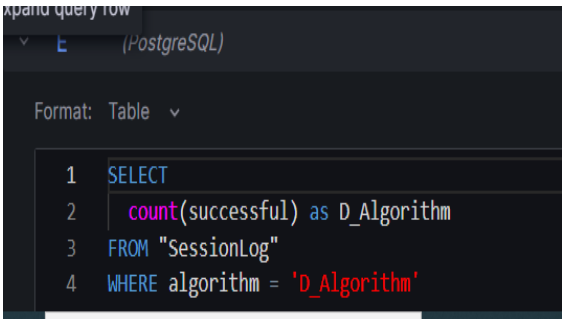


Рисунок 55 - Реализация дашборда количества ответов алгоритма D за все время

Нас так же интересует тенденция общего процента правильных ответов за каждый промежуток времени, поскольку он будет показывать, как менялась оценка пользователями в связи и позволит точнее определить, когда она начала падать или расти, а значит и определить причины(рисунок 56).

Рисунок 56 - Дашборд процента правильных ответов

Данный график строить группировкой по выбранному промежутку времени и выводом числа успешных ответов поделенных на общее количество ответов(рисунок 57).

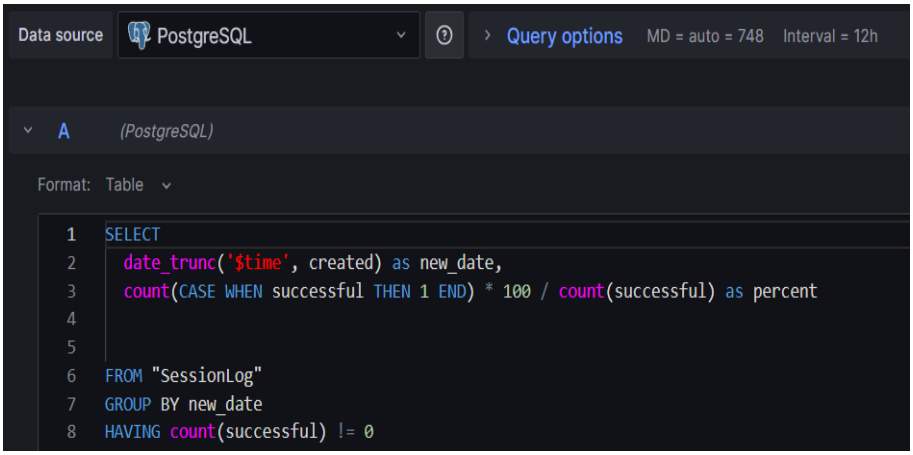


Рисунок 57 - Реализация дашборда процента правильных ответов

Для отслеживания того, когда был дан ответ каждым алгоритмом, было решено добавить график, показывающий это. В случае необходимости данный

график поможет найти, когда произошли проблемы с распределением алгоритмов или с каким-то алгоритмом(рисунок 58).

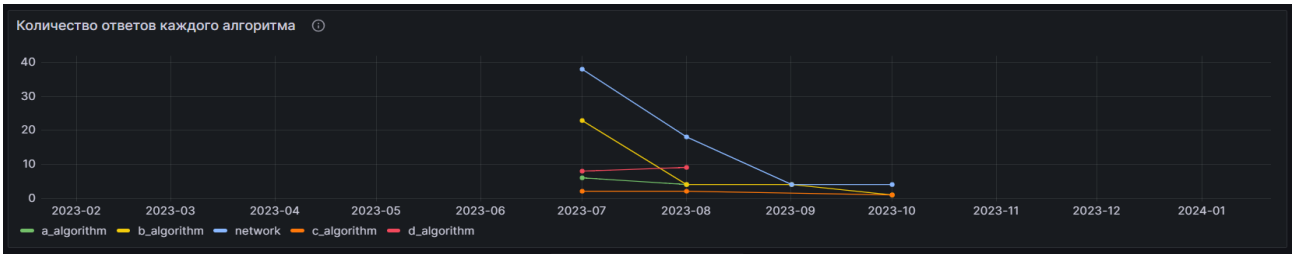


Рисунок 58 - Дашборд количества ответов каждого алгоритма

Для того чтобы не делать большой запрос было решено разделить его на 5 отдельных запросов, где в каждом из них будет производиться группировка по выбранному промежутку и фильтрация количества ответов от конкретного алгоритма(рисунок 59, 60, 61, 62, 63).

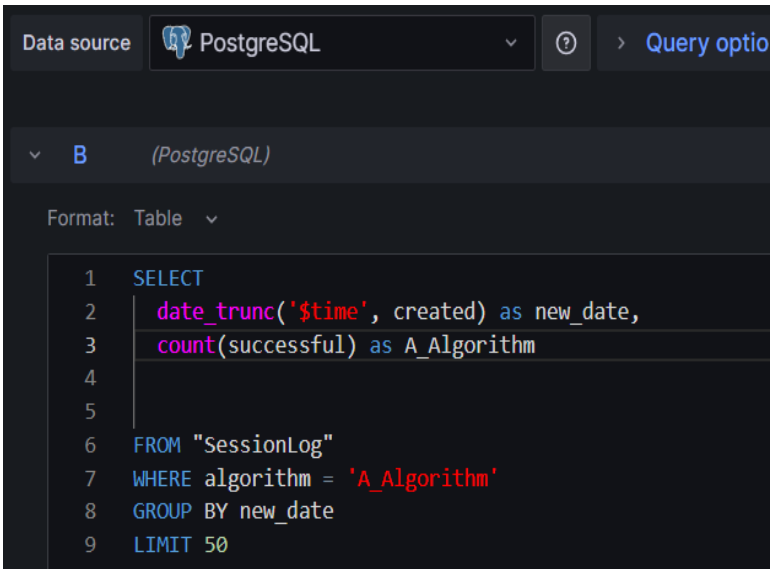


Рисунок 59 - Реализация дашборда количества ответов алгоритма A

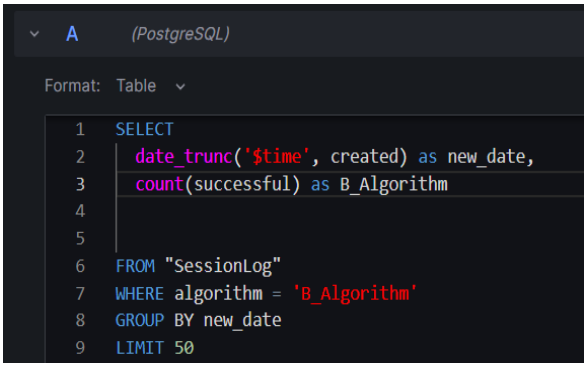


Рисунок 60 - Реализация дашборда количества ответов алгоритма B

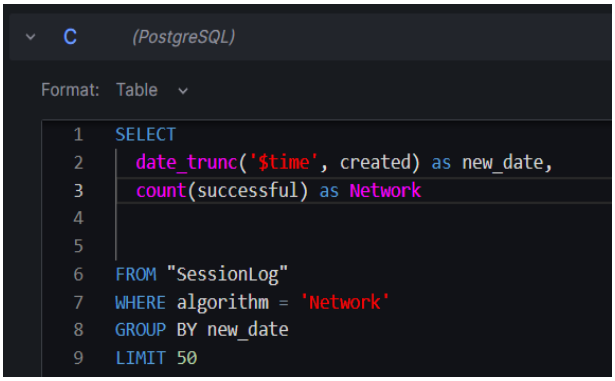


Рисунок 61 - Реализация дашборда количества ответов нейросети

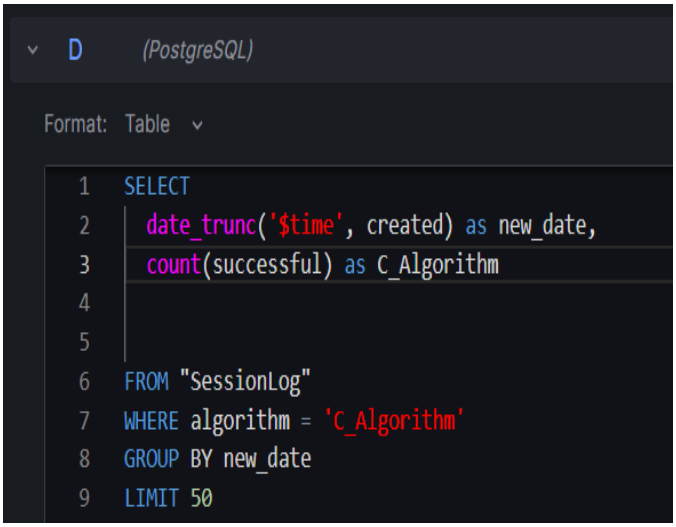


Рисунок 62 - Реализация дашборда количества ответов алгоритма C

```
E (PostgreSQL)
Format: Table
1  SELECT
2    date_trunc('$time', created) as new_date,
3    count(successful) as D_Algorithm
4
5
6  FROM "SessionLog"
7  WHERE algorithm = 'D_Algorithm'
8  GROUP BY new_date
9  LIMIT 50
```

Рисунок 63 - Реализация дашборда количества ответов алгоритма D

Так же для отслеживания качества ответов каждого из алгоритмов был построен график, который показывал бы как каждый из алгоритмов в каждый момент времени отвечал(рисунок 64).

Рисунок 64 - Дашборд процента правильных ответов по алгоритмам

Аналогично с предыдущим подходом запросы были разделены по алгоритмам, где в каждом запросе высчитывался рейтинг алгоритма за определенный промежуток времени, когда у него был хотя бы один ответ(рисунок 65, 66, 67, 68, 69).

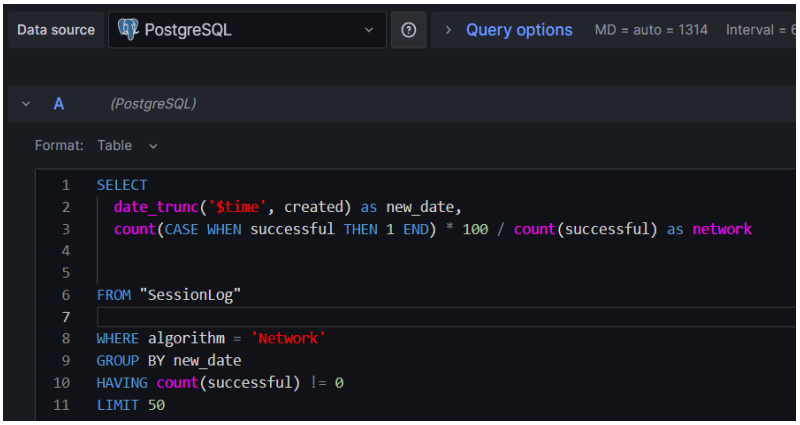


Рисунок 65 - Реализация дашборда процента правильных ответов по нейросети

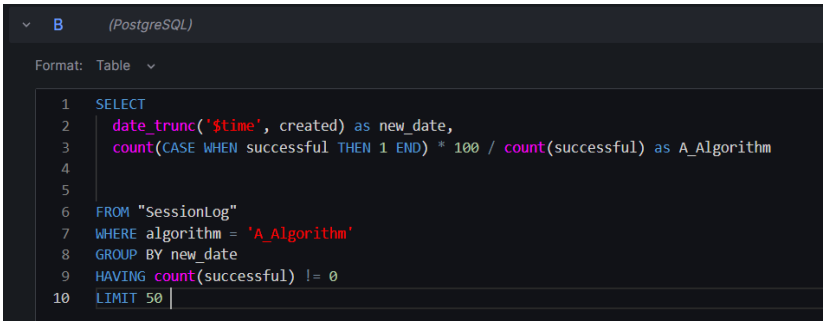


Рисунок 66 - Реализация дашборда процента правильных ответов по алгоритму A

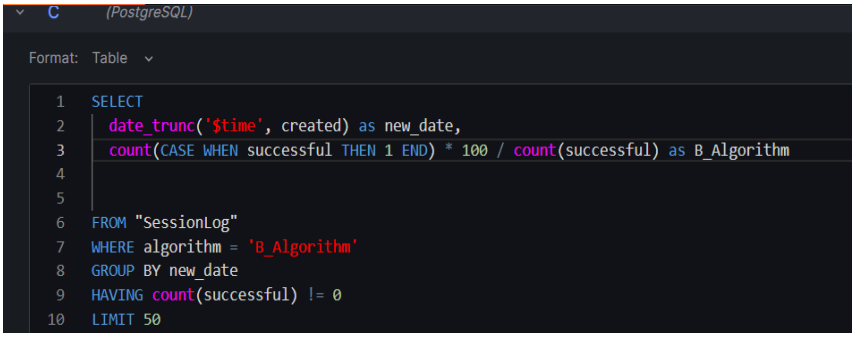


Рисунок 67 - Реализация дашборда процента правильных ответов по алгоритму B

Рисунок 68 - Реализация дашборда процента правильных ответов по алгоритму C

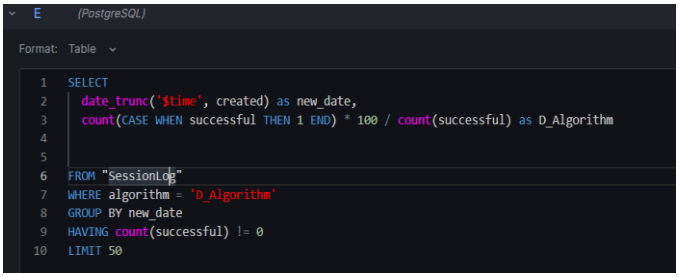


Рисунок 69 - Реализация дашборда процента правильных ответов по алгоритму D

Так же для отслеживания того, как в общем работают алгоритмы, был сделан дашборд общей средней оценки их работы(рисунок 70).

Рисунок 70 - Дашборд процента правильных ответов за все время

Для его построения так же было составлено 6 запросов на каждый алгоритм и так же 1 запрос на общий процент правильных ответов(рисунок 71, 72, 73, 74, 75, 76).

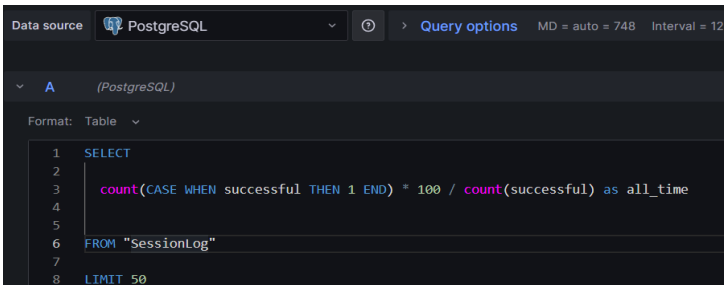


Рисунок 71 - Реализация дашборда процента правильных ответов всего за все время

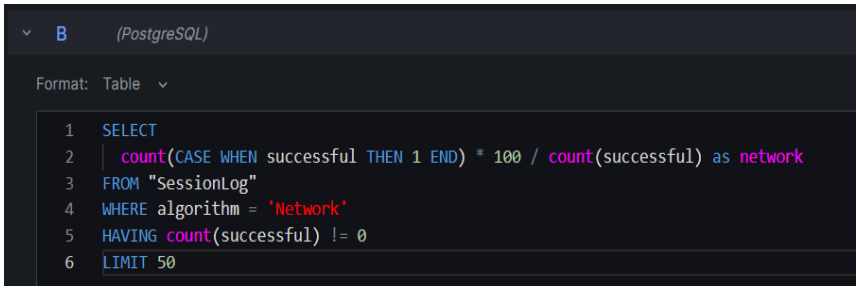


Рисунок 72 - Реализация дашборда процента правильных ответов нейросети за все время

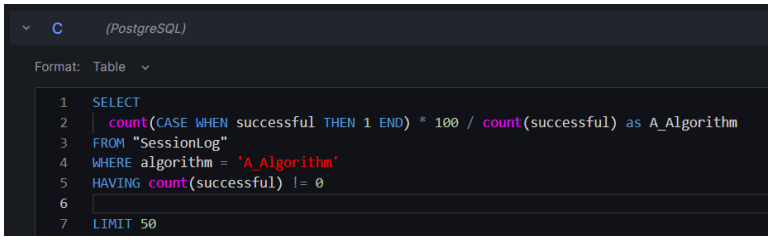


Рисунок 73 - Реализация дашборда процента правильных ответов алгоритма А за все время

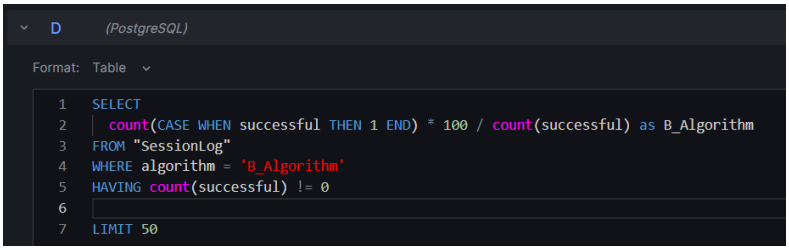


Рисунок 74 - Реализация дашборда процента правильных ответов алгоритма B за все время

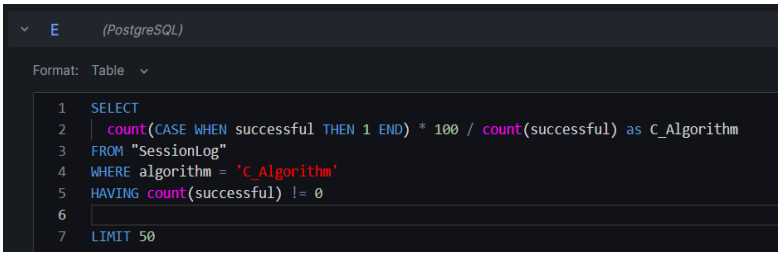


Рисунок 75 - Реализация дашборда процента правильных ответов алгоритма C за все время

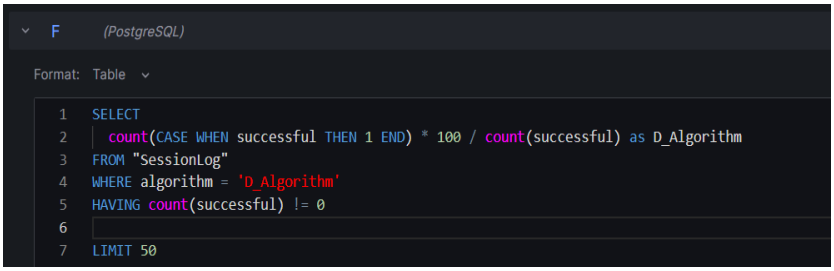


Рисунок 76 - Реализация дашборда процента правильных ответов алгоритма D за все время

В группе пользователей нас интересует сколько новых пользователей пришло в каждый момент времени(новым считается пользователь, который до этого ни разу не писал)(рисунок 77).

Рисунок 77 - Дашборд новых пользователей

Для этого мы, группируя по выбранному промежутку времени, смотрим сколько пользователей присоединились (создали аккаунт) в указанный промежуток времени(рисунок 78).

Рисунок 78 - Реализация дашборда новых пользователей

Так же нас интересует сколько людей пользовались сервисом(рисунок 79).

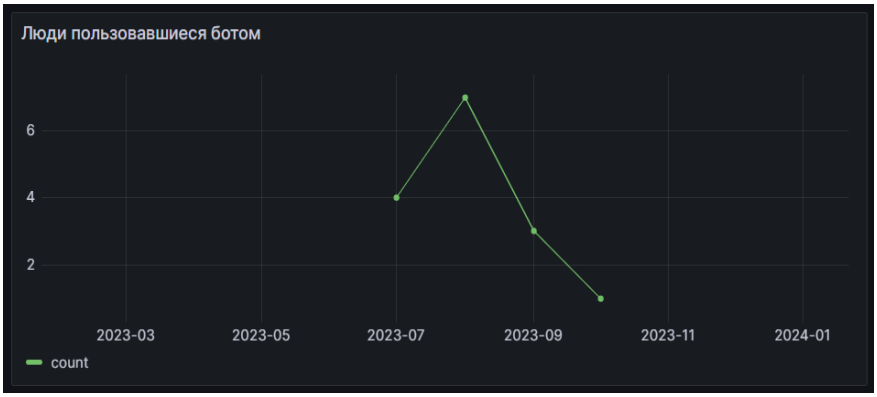


Рисунок 79 - Дашборд людей, пользовавшихся ботом

Для этого мы аналогично группируем по промежутку времени и смотрим число пользователей, но данные мы берем не из класса пользователей, а из логов(рисунок 80).

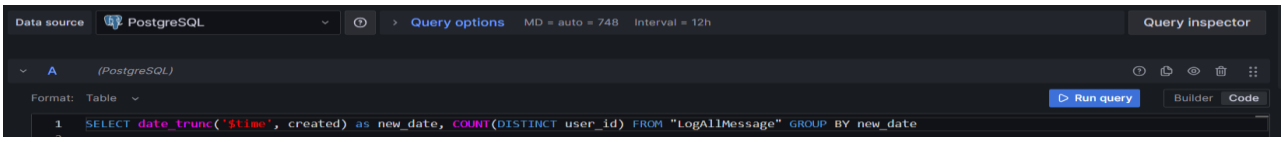


Рисунок 80 - Реализация дашборда людей, пользовавшихся ботом

Так же, заказчик попросил отслеживать количество вернувшихся людей, вернувшимися считаются люди, которые не пользовались ботом больше 30 дней(рисунок 81).

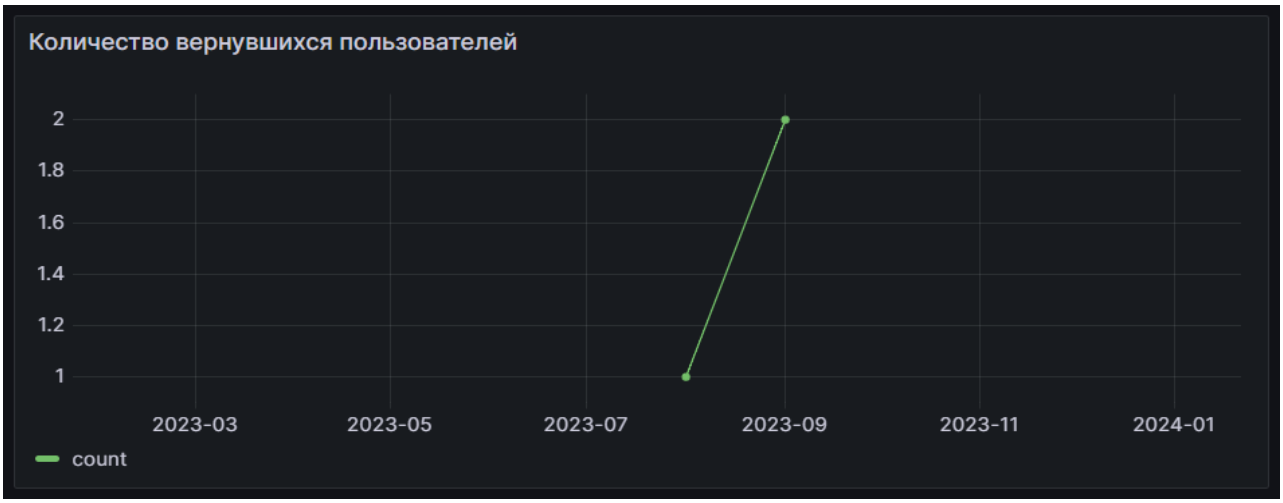


Рисунок 81 - Дашборд количества вернувшихся пользователей

Для построения этого графика пришлось использовать множественные вложенные запросы. Мы берем логи пользователей, сортируем их по дате, для каждой записи каждого пользователя берем отбираем те, которые были от пользователя до текущей по графику даты, смотрим была ли последняя запись

дольше чем за 30 дней и только в этом условии добавляем ее на график(рисунок 82).

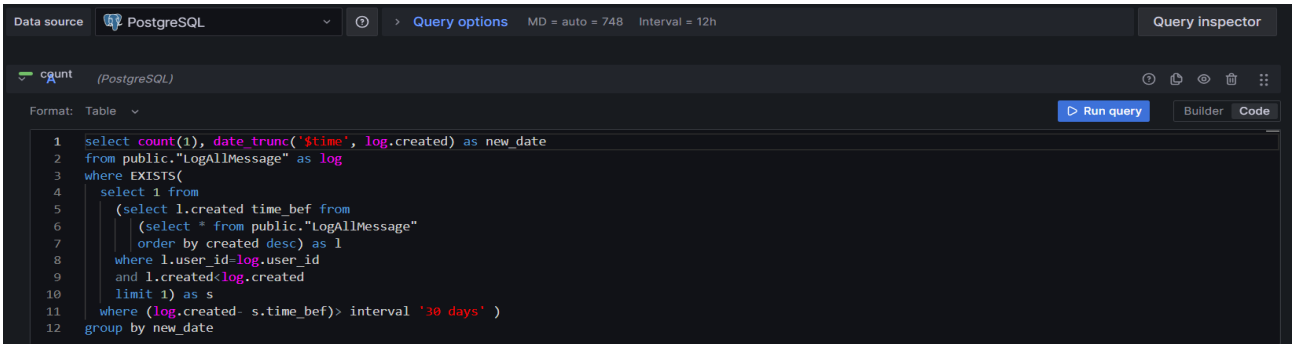


Рисунок 82 - Реализация дашборда количества вернувшихся пользователей

Так же нам интересно видеть на каждый момент времени сколько у нас пользователей, это полезно, например для отслеживания того, как сработала рекламная компания(рисунок 83).

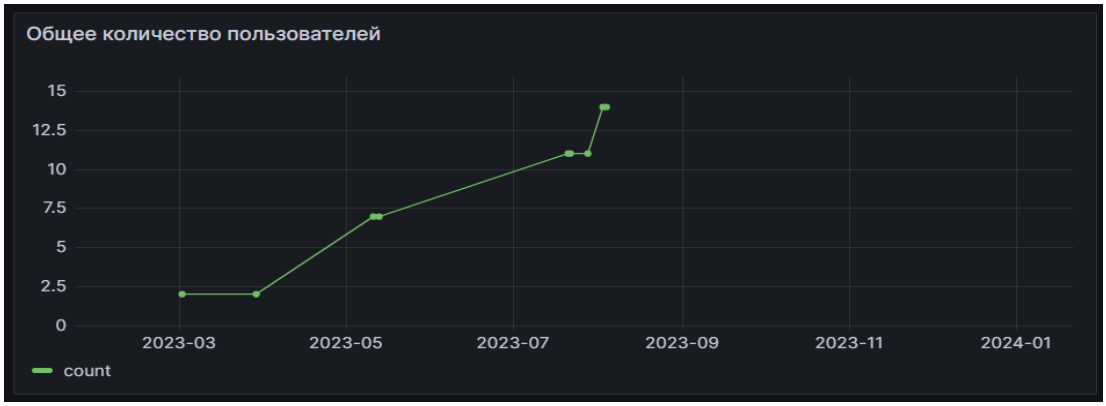


Рисунок 83 - Дашборд общего количества пользователей

Для построения данного графика мы просто берем число пользователей до определенной даты, а дату группируем по указанному промежутку(рисунок 84).

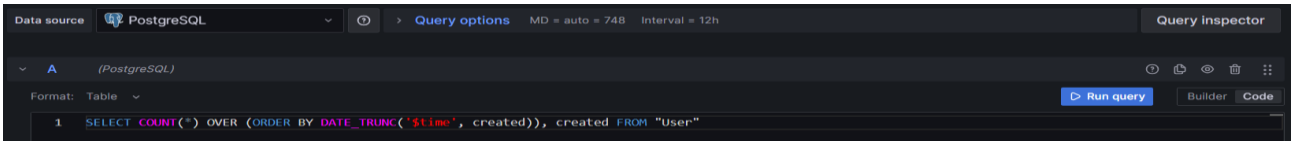


Рисунок 84 - Реализация дашборда общего количества пользователей

А для понимания того, как много пользователи пишут и детектирование ddos атак, было решено добавить график общего числа сообщений от пользователей(рисунок 85).

Рисунок 85 - Дашборд числа сообщений пользователей

Для его построения нам нужно только сгруппировать сообщения по интервалу и взять число сообщений за данный интервал(рисунок 86).

Рисунок 86 - Реализация дашборда числа сообщений пользователей

Таким образом были реализованы дашборды позволяющие анализировать все основные показатели как необходимые заказчику, так и потенциально необходимые. Так же их реализация поможет снизить количество затрачиваемого времени на реализацию иных дашбордов за счёт готовой основы в уже реализованных.

Вывод

В главе 2 были рассмотрены различные аспекты разработки нововведений для чат-бота. На начальном этапе была проведена моделирование изменений в принципе работы бота, что позволило определить необходимость внедрения системы мониторинга и управления данными. Затем была выполнена программная реализация нововведений, включая создание классов для работы с базой данных, реализацию логики сбора данных и оценки ответов пользователей, а также разработку дашбордов для анализа данных.

Был использован подход DAO для работы с базой данных, что обеспечило удобство и структурированность взаимодействия с данными. Реализованная логика сбора данных позволяет эффективно анализировать активность пользователей, их удовлетворенность сервисом и скорость ответов на вопросы. Дашборды, построенные на основе собранных данных, предоставляют заказчику возможность наглядно отслеживать ключевые метрики и принимать обоснованные решения по оптимизации работы бота.

Таким образом, вторая глава охватывает весь цикл разработки нововведений для чат-бота, начиная с проектирования изменений и заканчивая построением аналитических инструментов для мониторинга и анализа работы бота.

Заключение

В рамках данной бакалаврской работы был проведен анализ и модернизация чат-бота для автоматизированных ответов на вопросы пользователей с учетом проблем недостаточной производительности и ограниченного функционала. Результаты данного исследования позволили предложить, а также реализовать решение, которое было направленно на повышение эффективности и функциональности бота.

Модернизация чат-бота включала в себя использование библиотеки psycopg2 для взаимодействия с объектно-реляционной системой управления базой данных PostgreSQL которое позволило быстро и удобно взаимодействовать с собираемыми данными, а также разработку универсальных функций для работы с данными, позволившее сократить избыточное дублирование кода тем самым повысив гибкость системы. Внедрение паттерна DAO (Data Access Object) обеспечило единый интерфейс для работы с различными классами и сократило дублирование кода. Основными компонентами модернизированной бота стали система мониторинга и управления данными, использование Grafana для визуализации данных и разработка дополнительных функций для анализа и обновления данных. Это позволило заказчику эффективнее управлять данными, повысить производительность бота и предоставить дополнительные возможности для анализа и оптимизации ее работы.

Использование Grafana для визуализации данных стало ключевым инструментом анализа работы бота. Построение графиков и таблиц позволило оперативно реагировать на изменения в работе бота и принимать меры для улучшения ее функционирования.

Результаты данной бакалаврской работы позволяют сделать вывод о создании эффективной системы управления данными и анализа работы автоматизированного ответа на вопросы пользователей.

Список используемой литературы

1. Арсланов С.К. Визуализация метрических данных при помощи платформы GrafanA / Арсланов С.К., Зорин О.А. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=50770409 (дата обращения: 12.03.2024).
2. Афанасьева Ж.О. Применение технологии docker и средства управления docker-приложениями / Афанасьева Ж.О. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=43173940 (дата обращения:12.01.2024).
3. Безрукова О.А. Интеграция Spring Boot приложения с Prometheus и Grafana / Безрукова О.А. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=46128028 (дата обращения: 08.01.2024).
4. Богатов И.В. Эффективная оптимизация запросов в субд postgres / Богатов И.В. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=48509203 (дата обращения: 15.01.2024).
5. Галлини Н.И. Использование программной системы визуализации Grafana для создания витрин данных / Галлини Н.И., Гуляев С.М. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=54606304 (дата обращения: 15.02.2024).
6. Гришин И.Ю. Применение сервисов kibana и beats для обнаружения и анализа инцидентов информационной безопасности в системах критической информационной инфраструктуры / Гришин И.Ю., Тимиргалеева Р.Р., Чертоганов К.А. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=43880655 (дата обращения: 23.03.2024).
7. Дегтярь Р.С. Плагин для платформы визуализации grafana – индикационная панель с несколькими источниками данных / Дегтярь Р.С. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=46602321 (дата обращения: 12.01.2024).

8. Еровлева Р.В. Мониторинг с помощью micrometer, prometheus и grafana / Еровлева Р.В., Еровлев П.А. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=47245998 (дата обращения: 15.02.2024).

9. Зарубин М.Е. Разработка telegram бота для помощи абитуриентам сибгути / Зарубин М.Е., Андреев А.В. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=48675724 (дата обращения: 18.02.2024).

10. Захарчёнок В.Ф. Контейниризация и развертывание приложений с помощью docker и docker-compose / Захарчёнок В.Ф., Бизюк А.Н. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=54666783 (дата обращения: 28.04.2024).

11. Зинкевич А.В. Исследование sql-инъекций / Зинкевич А.В., Суриков Д.Н. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=47686481 (дата обращения: 25.04.2024).

12. Кириллов Д.С. Базовые SQL команды и структуры в SQL Server / Кириллов Д.С., Насиров Э.Ф., Мертинс Г.Р., Молостов Д.Д. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=47496395 (дата обращения: 25.02.2024).

13. Кузнецов М.С. Реализация распределённых вычислений на языке python с использованием технологии docker / Кузнецов М.С., Андрианов И.А. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=42764322 (дата обращения: 18.02.2024).

14. Куницын В.И. Особенности написания баз данных postgresql / Куницын В.И., Новикова Т.П. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=54513149(дата обращения: 08.01.2024).

15. Лежнин М.С. Разработка telegram бота на python / Лежнин М.С. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=50213869 (дата обращения: 15.04.2024).

16. Малашихин Д.М. Создание telegram-ботов для встраиваемой базы данных sql / Малашихин Д.М. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=54497089 (дата обращения: 05.01.2024).

17. Марковской В.В. Визуализация базы данных / Марковской В.В. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=54296100 (дата обращения: 04.01.2024).

18. Мастеров А.С. Способы реализации индекс менеджмента в системе elasticsearch / Мастеров А.С. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=42943154 (дата обращения: 16.03.2024).

19. Мастеров А.С. Сравнение метрик поисковых движков elasticsearch и splunk / Мастеров А.С. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=42978371 (дата обращения: 19.02.2024).

20. Никифоров И.В. Программные инструменты обработки и визуализации данных. Elasticsearch, logstash, kibana, grafana, prometheus / Никифоров И.В., Юсупова О.А., Воинов Н.В., Ковалев А.Д., Ткачук А.С., Варламов Д.А., Гераськин Е.В. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=54313280 (дата обращения: 18.02.2024).

21. Окулов М.В. Создание web-приложения на django для программирования асинхронных telegram ботов на языке python 3 с использованием библиотеки aiogram / Окулов М.В., Федорова Н.Е. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=47347678 (дата обращения: 05.01.2024).

22. Пменьшикова Л.В. Обзор изменений языка структурированных запросов sql к реляционным базам данных: от sql-2003 до sql-2023 / Пменьшикова Л.В., Найденова Д.М. // elibrary 2024. URL: https://www.elibrary.ru/item.asp?id=64144101 (дата обращения: 08.01.2024).

23. Погуляй Г.С. Реализация системы контроля посещаемости занятий на основе социальной сети telegram / Погуляй Г.С., Гончаренко Ю.Ю. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=47462834 (дата обращения: 15.04.2024).

24. Пьянзин С.А. Корпоративное решение с использованием поисковой системы elasticsearch / Пьянзин С.А., Григорьев Ю.А., Щеглов Д.С. // elibrary 2020. URL: https://www.elibrary.ru/item.asp?id=42731876 (дата обращения: 28.04.2024).

25. Толмачев В.Е. Разработка информационного telegram-bot`a в сфере инвестиций / Толмачев В.Е. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=48486645 (дата обращения: 02.03.2024).

26. Шелепина О.Д. Сравнительный анализ инструментов управления журналами на примере elk и graylog / Шелепина О.Д., Хадорич Д.Д. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=50138372 (дата обращения: 16.03.2024).

27. Asgher M.N. Development of a low-cost, open-source lora-based scada system for remote monitoring of a hybrid power system for an offshore aquaculture site in Newfoundland / Asgher M.N., Iqbal M.T. // elibrary 2023. URL: https://elibrary.ru/item.asp?id=64655694 (дата обращения: 16.01.2024).

28. Blumen R. Postgres server developer bruce momjian discusses multiversion concurrency control / Blumen R. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=59226549 (дата обращения: 22.01.2024).

29. Nikolaeva I. Minimizing images of docker container root file systems / Nikolaeva I., Gankevich I. // elibrary 2021. URL: https://www.elibrary.ru/item.asp?id=47545224 (дата обращения: 16.02.2024).

30. Noprianto N. Monitoring development board based on influxdb and grafana / Noprianto N., Wijayaningrum V.N., Wakhidah R. // elibrary 2023. URL: https://www.elibrary.ru/item.asp?id=61879079 (дата обращения: 05.01.2024).

31. Reis D. Developing docker and docker-compose specifications: a developers’ survey / Reis D., Piedade B., Correia F.F., Dias J.P., Aguiar A. // elibrary 2022. URL: https://www.elibrary.ru/item.asp?id=58539882 (дата обращения: 26.04.2024).