Условия по полисам ОСАГО у страховых компаний обновляются не чаще двух раз в год, что позволяет компании заранее подготовить и систематизировать информацию. Это включает создание базы данных с предложениями по ОСАГО, учитывающей наиболее распространенные запросы (средний возраст и водительский стаж клиентов; популярные марки автомобилей; географическое распределение клиентов по городам и регионам.)

**Архитектура решения**

### 1. Реализация osago-aggregator

* Сервисы core-app и osago-aggregator будут взаимодействовать через брокер сообщений, так как необходимо обеспечить мгновенное отображение данных в реальном времени по мере их получения. В этой схеме core-app будет подписан на событие, связанное с добавлением новых данных о страховке, а osago-aggregator будет публиковать это событие каждый раз, когда поступают данные от различных источников. В качестве брокера сообщений предлагается использовать Kafka.
* Функциональность оформления полиса ОСАГО можно реализовать с использованием технологии Long Polling (это позволит получать обновления данных в реальном времени или с минимальной задержкой). Например, статус оформления можно запрашивать с интервалом в 5-10 секунд, установив максимальное время ожидания ответа в 60 секунд. В этом случае мы зависим от API внешних сервисов, которые предоставляют данные о процессе оформления.

### 2. Хранилище данных

* Для нового сервиса **osago-aggregator** создается отдельная база данных, где будут храниться актуальные предложения по страховкам ОСАГО от партнерских компаний.
* Периодически (например раз в месяц) база данных будет обновляться, чтобы предоставлять актуальные предложения по страховкам для наиболее востребованных запросов.

### 3. Кэширование

* Для повышения скорости обработки запросов будет использоваться кэширование, например, с помощью **Redis**. Это позволит уменьшить время отклика и снизить нагрузку на основную базу данных.

### 4. Паттерны отказоустойчивости

* Для контроля нагрузки на приложение **core-app** будет применяться паттерн **Rate Limiter**, который ограничивает количество запросов от одного пользователя.
* Если данные не найдены в нашей базе, запросы будут перенаправляться к сторонним API. Учитывая требование максимальной задержки в 60 секунд, будет использоваться паттерн **Resilient Fetch** с таймаутом. Этот паттерн повторяет запросы в случае неудачи, а также поддерживает стратегию фейловера, например, переключение на резервный адрес, если он доступен.

### 5. Возможные проблемы

* Поскольку сервис будет работать на нескольких инстансах, существует риск дублирования записей в базе данных. Это может произойти, если разные инстансы обработают одинаковый запрос и сохранят его в БД. Для решения этой проблемы можно периодически запускать скрипт для оптимизации и очистки дублирующихся записей.