Дизайн сервиса по доставке еды в крупном мегаполисе

1. **Сбор требований**:

a. *Scope refinemet*:

Для начала нужно определиться с тем, что мы будем реализовывать, а что не будет входить в решение в рамках нашего сервиса. Для реализации доставки достаточно будет 4 видов пользователей: ***ресторан***, ***курьер***, ***покупатель***, ***админ***. В задачу сервиса не входит построение логистики ресторанов (закупка/доставка продуктов, производство еды). В задачу сервиса входит доставка готовой еды оптимальным образом (построение маршрутов курьеров, где один курьер может доставлять несколько заказов).

b. *Functional requirements*:

Основные функциональные требования (на мой взгляд) перечислил на слайде. За скобками оставил такие понятия, как: оставление отзывов/комментариев, выставление рейтинга услуг ресторанов или доставки). Также необходима служба поддержки для разрешения конфликтных ситуаций и контроль качества услуг.

c. *Non-Functional reuirements*:

Для сервиса по доставке еды очень важно быть **доступным** 24/7. Если данные будут не консистентные (напр. с одного устройства будет показано, что какой-то товар ресторан не производит, а с другого устройства наоборот), это не будет являться критичной проблемой. С другой стороны, если наш сервис предоставляет услуги в пределах одного мегаполиса, то и данные будут содержаться где-то поблизости в пределах одного ДЦ. Следовательно, лаг репликации не будет большим. Также **SLA** по поиску товаров/ресторанов/курьеров **должен быть не большим**.

2. **Оценка нагрузки:**

Сделаем оценку для г.Москва с населением 10млн человек:

Для ресторанов:

1тыс – кол-во ресторанов. Для каждого ресторана в среднем возьмем меню, состоящее из 100 различных позиций (товаров). Каждая позиция содержит одну картинку + описание товара. Довольно редко обновляют меню. Чаще просматривают статистику. Следовательно, трафик будет не таким большим (по сравнению с покупателями), основная нагрузка придется на storage. Размер картинки с описанием товара возьмем за 500kB.

1K (ресторанов) * 100 (товаров) * 500kB = **50 Gb занимает хранение меню магазинов**.

Для курьеров:

12тыс – курьеров всего (одновременно на смене находится ~4тыс курьеров). Курьеры в основном генерируют трафик тогда, когда отправляют свои координаты. Предположим, что они это делают каждые 10с (0.1 rps). Координаты вместе с id можно поместить в 20 байт. Следовательно, 4K * 0.1rps * 20 байт = **~64Kb/s передача координат** (что очень мало)**.**

Для пользователей:

MAU = 1 млн и DAU = 100 тыс. Каждый пользователь в среднем создает один заказ в день. Тогда в день выходит ~100 тыс заказов. Перед созданием одного заказа пользователь просматривает 10 различных ресторанов по 10 разных товаров (один товар ~500kB). Следовательно, трафик будет: 100K * 10маг * 10тов * 500kB / 100k = **~400Mb/s просмотр товаров.** Для хранения инфо о заказе выделим 10kB.

Итого (на горизонте 5 лет):

История заказов: 100K (зак/день) * 10kB * 2000 дней = 2 TB. +25% запас = **2.5TB**

Трафик: 400Mb/s * 2000 дней * 100K = $8*10^7$ Gb. Если 0.1$ за 1 Gb, то: **~10M$**.

Connections:

Нужно держать connections для курьеров (передача местоположения), магазинов (получения нотификации о новом заказе) и для покупателей (отслеживание статуса товара и положение курьера). Магазин держит соединение всегда во время работы, курьер держит соединение когда выходит на смену, пользователь держит соединение когда отслеживает статус заказа (пусть в среднем заказ доставляется за полчаса). 100K заказов в день, доставка занимает 30мин. Следовательно, одновременно трекающих заказ 1.8K Тогда:
$1K_{ресторанов} + 10K_{курьеров} + 1.8K =$ **~15K** одновременных connections (что не проблема).

RPS:

Основная нагрузка будет на трекинг, поиск/просмотр товаров.
Трекинг: $4K_{курьеров} * 0.1rps = 400$ rps
Просмотр товаров: $100K_{заказов} * 100_{просмотров} / 100K_{секунд} = 100rps$ (SLA_{99} = 1c => 100 connections)

Storage (profile):

Для хранения одного профиля выделим 100kB => 1M * 100kB = **100GB**

High-Level Design:
Для покрытия основных функциональных требований рассмотрим вот такой дизайн:
У каждого пользователя будет своя точка входа. Каждую точку входа рассмотрим отдельно:
Рестораны:

Управление профилем, подтверждение/отмена заказов, просмотр/выгрузка статистики можно делать через ***Restaurant Service***, для этого у сервиса должен быть доступ к базе с профилями, заказами и меню.

Покупатели:

Они хотят делать поиск по товарам/ресторанам (***Explore Service***), класть товары в корзину (***Checkout Service***), редактировать профиль и просматривать историю заказов (***Customer Service***), мониторить положение курьера с его заказом (***Tracking Service***).

Курьеры:

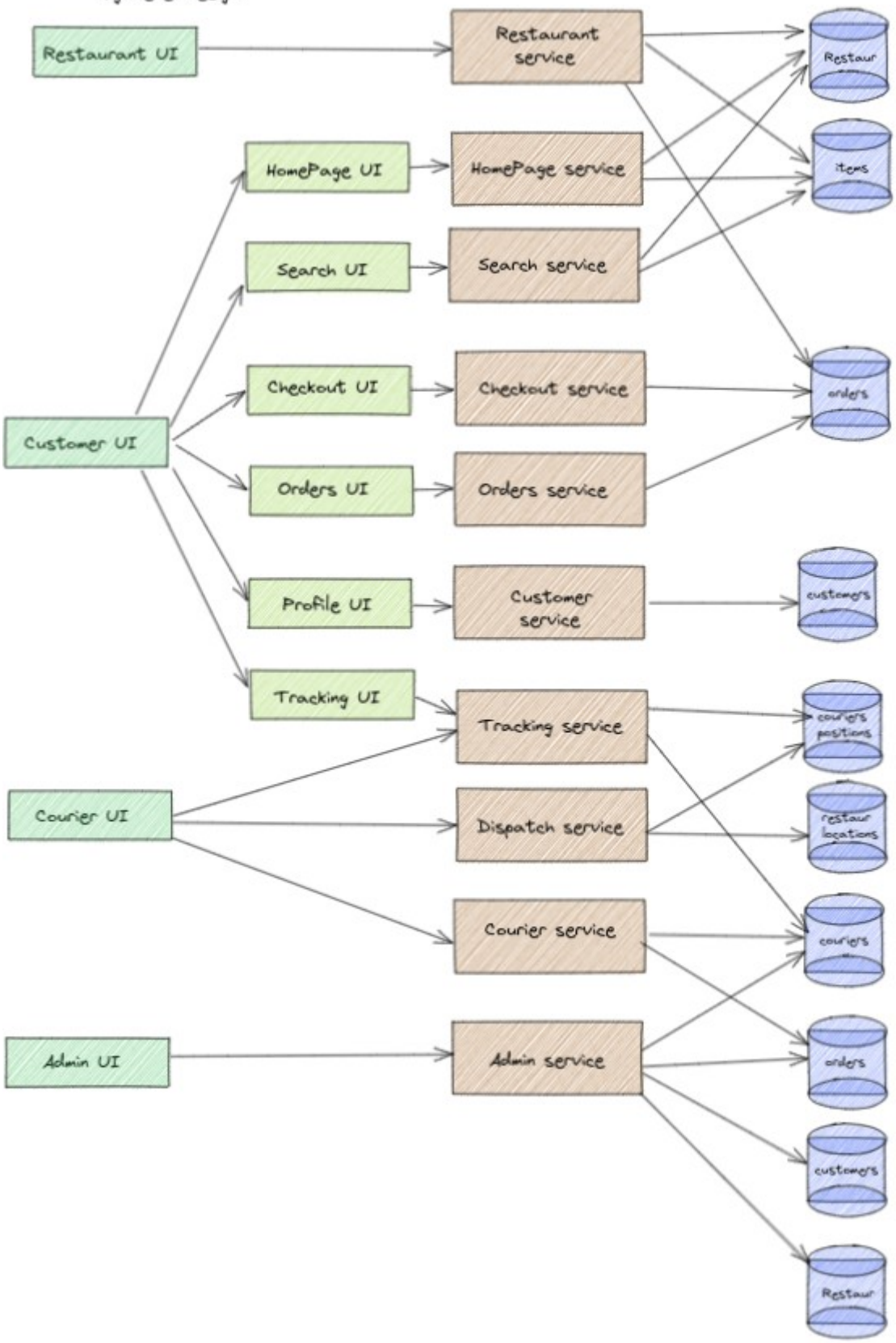
Редактировать профиль (***Courier Service***), принимать входящий заказ (***Dispatch Service***), транслировать свои координаты (***Tracking Service***).

Админы:

Валидировать профили, просмотр/выгрузка статистики и валидация профилей **(*Admin Service*).**

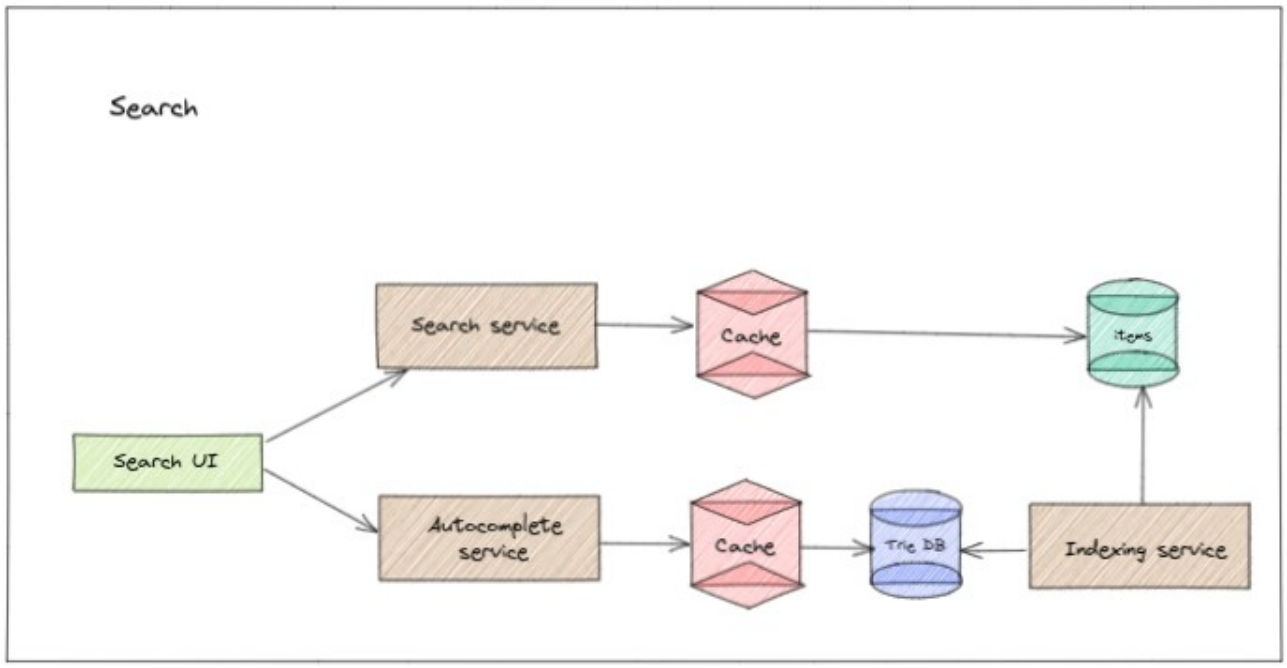
Данный высокоуровневый дизайн покрывает наши функциональные требования. Далее рассмотрим отдельно каждую компоненту.

High-level design:

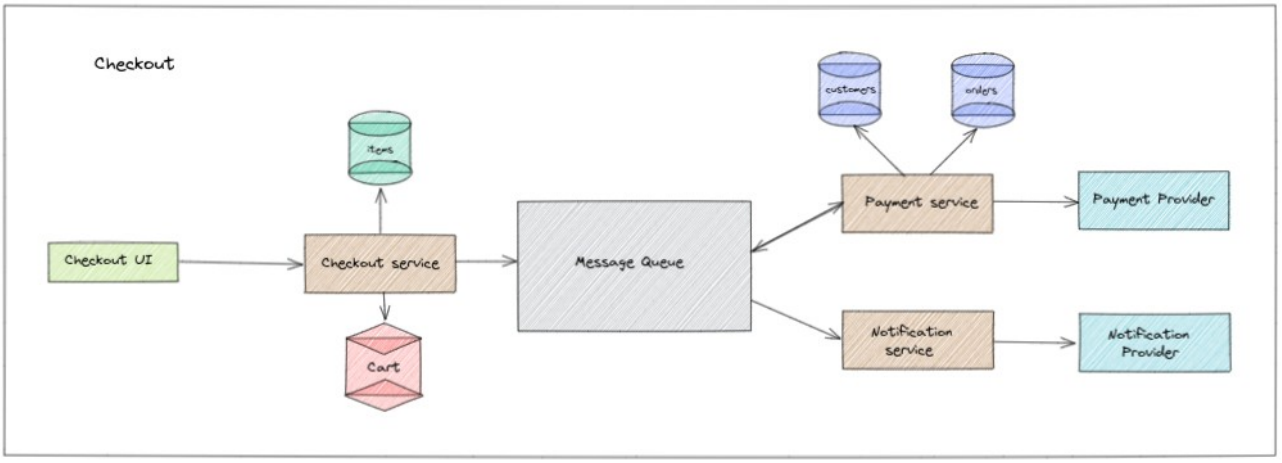


Component design:

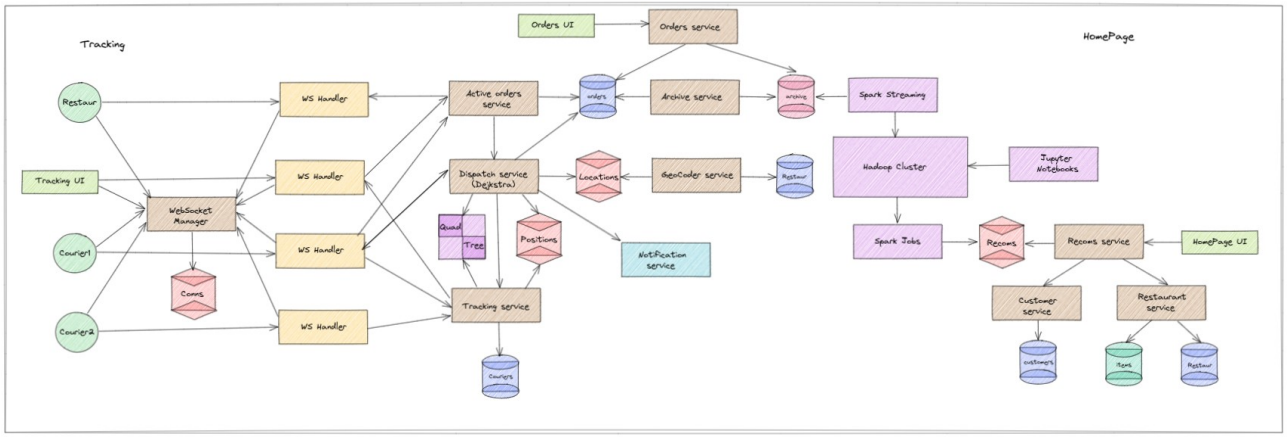
Реализуем основной функционал платформы: (Checkout, Search, Orders, Homepage, Tracking). Остальные компоненты системы (редактирование профил/меню, страница админки) реализуются достаточно просто.



Search UI отвечает за поиск еды/ресторанов по названию. Все довольно просто: search service отвечает за поиск, где самые популярные items кладем в кэш (можно использовать LFU например). Autocomplete service отвечает за автодополнение (берет из кэша построенное предварительно Trie из названий).



Checkout UI отвечает за пользовательскую корзину. Какие-то товары кладутся в корзину (из одного ресторана). Корзина хранится в redis для быстрого доступа к ней. Дальше отправляются товар на оплату (в message queue шлется сообщение, которое разгребается воркерами). Payment service имеет доступ к привязанной карте, данные которой отправляются во внешний payment провайдер партнера. В случае успешной оплаты Payment Service обновляет статус заказа и, через message queue шлет нотификацию об оплате.



Оставшиеся важные точки входа представлены на картинке выше (**Tracking UI**, **Orders UI**, **HomePage UI**). Рассмотрим по порядку:

Orders UI:

После успешной оплаты заказ пишется в базу orders. Orders service предоставляет доступ к активным и архивированным заказам.

HomePage UI:

Команда ML на основе исторических данных из таблички archive. Spark jobs строит рекоммендации и кладет их в редис. Пользователь при этом на главной странице может увидеть рекомендованные для него товары.

Tracking UI:

Для трэкинга заказа пользователем, получения координат курьеров, отправки заказов ресторанам и получения подтверждения удобно использовать вебсокеты для двухстороннего взаимодействия. Для моей системы в среднем придется держать ~7K connections (что не проблема). Сервис Active Orders берет из базы оплаченные заказы и отправляет его в Dispatch Service для поиска курьера и построения маршрута (возможно использовать алгоритм Дейкстры). Dispatch Service берет ближайших к ресторану курьеров из кэше в redis (координаты ресторана получены из сервиса geocoder, который из адреса ресторана возвращает координаты) и отправляет этим курьерам заказ. Какой-либо курьер его принимает (кто не принимает – штрафуется рейтингом) и отправляет свой id в Dispatch. После этого Dispatch обновляет статус заказа, шлет нотификацию “курьер найден” и отправляет заказ в Tracking сервис. Этот сервис получает координаты курьеров, обновляет их в базе и отправляет эти координаты пользователям (того курьера, который везет заказ). После того, как курьер доставит заказ, он отправляет подтверждение в “Active Orders Service”. Этот сервис обновляет статус заказа на “Finished”. “Archive Service” берет завершенные заказы и архивирует их в колоночную БД (Cassandra) для дальнейшего анализа командой аналитики.

Redis

PostgreSQL

MongoDB

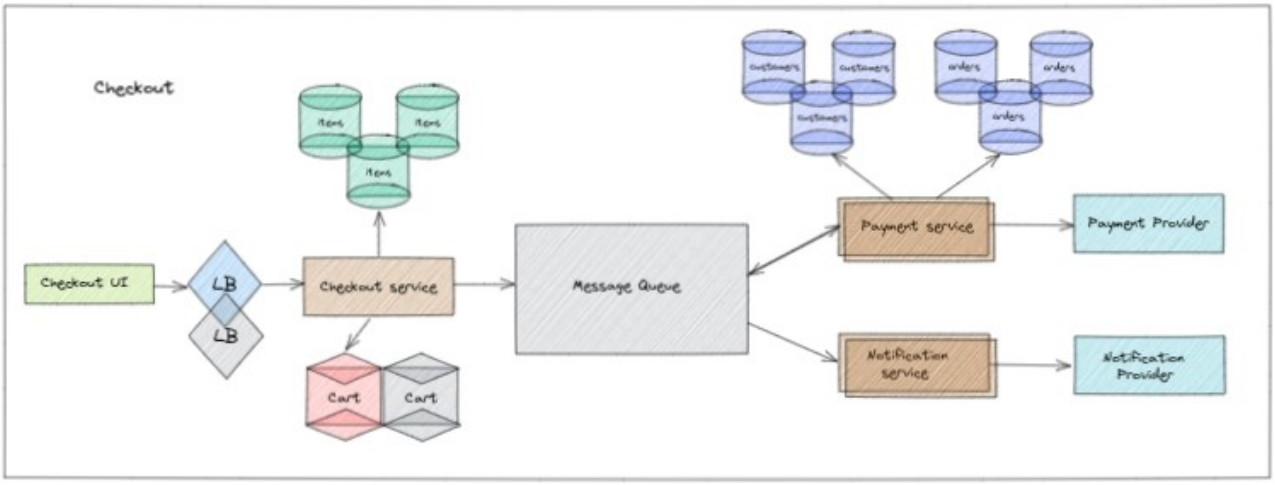
Cassandra

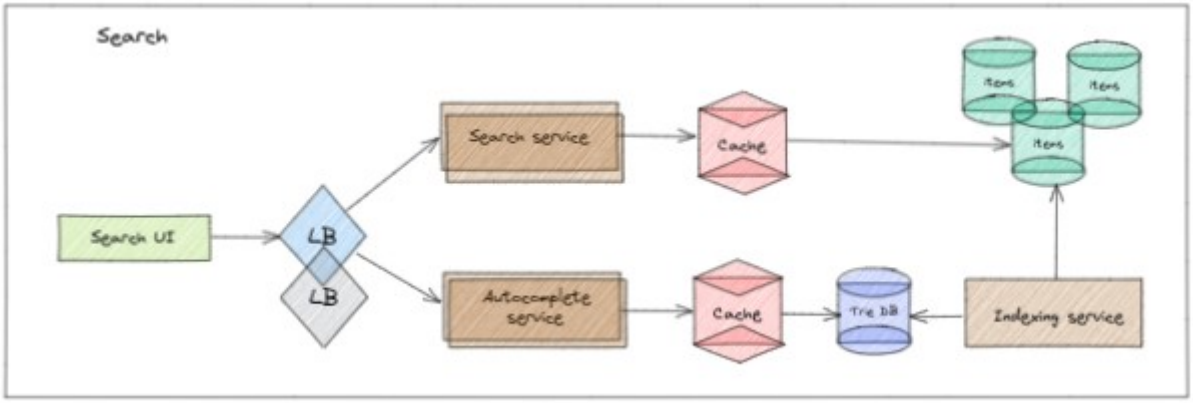
Повышение отзывчивости и отказоустойчивости:

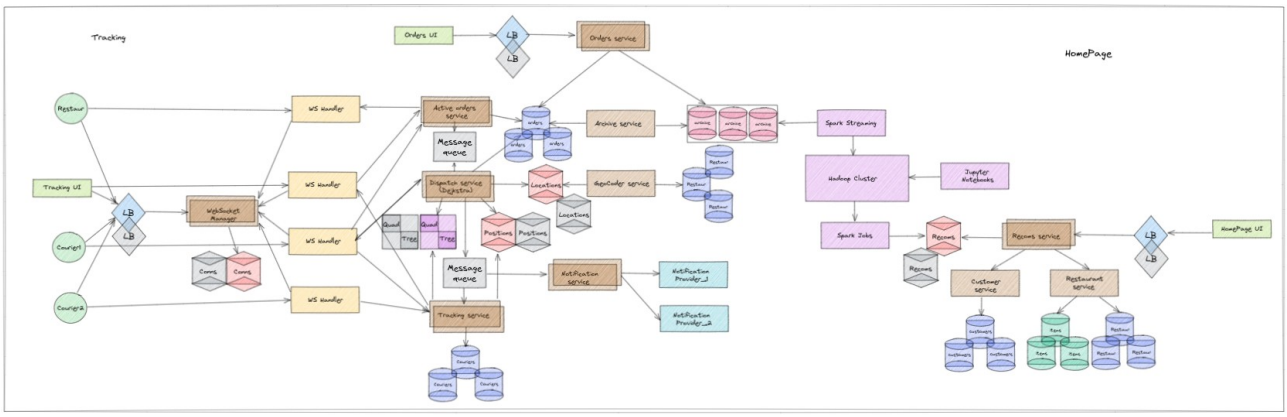
Для достижения отзывчивости самые горячие данные необходимо положить в LFU && LRU кэш (redis). Можно применить принцип Паретто и держать часть данных (напр. 10%). Для успешного деплоя необходимо иметь несколько инстансов каждого сервиса (чтобы исключить время shutdown и повысить availability). Критически важно, чтобы главный функционал оставался работоспособным (просмотр еды, создание заказов). Повышением количества инстансов можно балансировать нагрузку (поставив перед этим load balancers). Таким образом мы сможем переживать пиковые нагрузки (200%-300% от обычной). Для ограничения нагрузки со стороны пользователя нужно поставить rate limiters (nginx это умеет). Данные о еде удобно хранить в документоориентированных БД (типа MongoDB), т.к. товар не имеет строгой структуры полей. MongoDB достаточно легко масштабировать используя шардирование (consistency hashig). Данные пользователей/ресторанов/курьеров удобно хранить в ACID базах данных (PostgreSQL). Масштабировать PostgreSQL можно используя несколько реплик (напр. 2 реплики). Заказ имеет статусную модель (конечный автомат), поэтому данные о заказе удобно хранить в ACID БД. Также для повышения отказоустойчивости сервис можно деплоить сразу на несколько ДЦ. Консистентность при этом уменьшается (есть репликейшн лаг между БД), но это нам не критично. Гораздо более важно, чтобы сервис оставался доступным 24/7. При этом не стоит гонять большой трафик между ДЦ (особенно для команды ML), т.к. это может быстро забить весь пропускной канал.

Мониторинг и аллертинг:

Очень важно правильно настроить мониторинг и аллертинг, чтобы первыми узнавать, что что-то происходит не так. Нужно мониторить тайминги и rps ручек, ресурсы машинки (cpu wait, кол-во свободных потоков, память), ресурсы БД (время выполнения запросов, кол-во соединений к базе, лаг репликации), network (кол-во открытых соединений, пропускной канал).






Tracking
Orders UI
LB
LB
Orders service
HomePage
Restaur
WS Handler
Active orders service
orders
Archive service
archive
Spark Streaming
Message queue
Tracking UI
LB
LB
WebSocket Manager
WS Handler
Dispatch service (Logistic)
Locations
RepCenter service
Restaur
Hadoop Cluster
Jupyter Notebooks
Courier1
Comm
Comm
WS Handler
Quad Tree
Quad Tree
Locations
Spark Jobs
Recom
Recom
Recoms service
LB
LB
HomePage UI
Courier2
WS Handler
Positions
Positions
Message queue
Notification service
Notification Provider_1
Notification Provider_2
Customer service
Restaurant service
Tracking service
customers
Item
Item
Restaur
Restaur
couriers
couriers
couriers