Этап 1. Расчет базового MVP

1.1 Границы проекта (Scope Refinement)

Мы создаем глобальный сервис заказа такси.

Базовое MVP решение:

- Два мобильных приложения: для пассажира и для водителя.
- Регистрация/Авторизация/Профили.
- Геолокация: Пассажир видит машины рядом; Водитель транслирует свою позицию.
- Процесс поездки: Заказ -> Матчинг (поиск) -> Поездка -> Оплата -> История.
- Базовые уведомления (Push/SMS).

Получение продвинутого продукта:

- Сложные алгоритмы динамического ценообразования (Surge pricing).
- Карпулинг (совместные поездки).
- Планирование поездок заранее (Scheduled rides).
- Глубокая аналитика и ML-прогнозы.

1.2 Функциональные требования

Пассажир:

1. Может видеть водителей поблизости на карте в реальном времени.
2. Может запросить поездку, указав точку отправления и назначения.
3. Видит расчетную стоимость и время подачи (ETA) перед заказом.
4. Получает уведомления о статусе (Водитель найден, Водитель прибыл, Поездка началась/завершена).
5. Может просматривать историю поездок.

Водитель:

1. Может менять статус (Онлайн/Офлайн).
2. Получает уведомления о новых заказах поблизости.

3. Может принять или отклонить заказ (с таймером).
4. Видит навигацию к пассажиру и к точке назначения.
5. Видит свой заработок (историю поездок).

Система (Backend):

1. **Matchmaking:** Находит свободного водителя рядом с пассажиром.
2. **Dispatching:** Распределяет заказы.
3. **Tracking:** Отслеживает координаты во время поездки.

1.3 Нефункциональные требования

1. **Высокая доступность и надежность:**
   - Система должна работать 24/7. Простой сервиса в час пик стоит огромных денег и потери репутации.
   - Согласно CAP-теореме, в моменты сетевых сбоев мы выберем **AP** (доступность) для функций поиска водителей (лучше показать чуть устаревшие данные о машинах, чем ошибку).
   - Для оплаты и завершения поездки нам нужна строгая консистентность (**CP**), чтобы не списать деньги дважды.
2. **Низкая задержка (Low Latency):**
   - Обновление позиций машин на карте должно быть плавным (задержка < 2-3 сек).
   - Матчинг должен происходить быстро (не заставлять пассажира ждать поиска 2 минуты).
3. **Масштабируемость (Scalability):**
   - Система должна выдерживать пиковые нагрузки (утро/вечер, праздники, концерты).

1.4 API Дизайн (Базовые методы)

Определим основные эндпоинты (REST / HTTP), следуя модульной логике:

Auth Service:

- `POST /users/login` — вход.
- `POST /users/register` — регистрация.

Passenger API:

- `GET /ride/nearby?lat={lat}&lon={lon}&radius={r}` — получить список водителей рядом (для отображения на карте).
- `POST /ride/request` — создать заказ. *Body: {src_lat, src_lon, dest_lat, dest_lon, type}*.
- `GET /ride/{id}/status` — поллинг статуса (или через WebSocket).

Driver API:

- `POST /driver/status` — переключить (Active/Inactive).
- `PUT /driver/location` — обновление координат (частый запрос, позже переведем на сокеты).
- `POST /ride/{id}/accept` — принять заказ.

1.5 Высокоуровневый дизайн (High-Level Design)

Предлагается использовать микросервисную архитектуру, так как система сложная и нагруженная.

Основные компоненты:

1. **Clients (Mobile Apps):** Общаются с бэкендом через Load Balancer.
2. **Load Balancer (Nginx/Cloud LB):** Распределяет трафик, терминирует SSL.
3. **API Gateway:** Единая точка входа, маршрутизация запросов к микросервисам, аутентификация.
4. **Сервисы:**
   - **User/Driver Service:** Управление профилями, рейтингом.
   - **Ride Service (Trip Service):** Основная бизнес-логика. Создание заказа, изменение статуса (Requested -> Matched -> InProgress -> Finished).
   - **Location Service (Geo Service):** Самый нагруженный. Получает координаты от водителей, хранит гео-индекс, отвечает на запросы "кто рядом".
   - **Notification Service:** Отправка Push-уведомлений.
5. **Хранилища данных (Databases):**
   - **RDBMS (PostgreSQL):** Для пользователей, заказов, транзакций (ACID).
   - **NoSQL / In-Memory (Redis + Geo):** Для "горячих" гео-данных и быстрого поиска.

Схема потока данных (Flow):

> **Пассажир** -> **Load Balancer** -> **API Gateway** -> **Ride Service** (Создает заказ)
>
> **Ride Service** -> **Location Service** (Ищет водителей рядом) -> **Возвращает список кандидатов**.
>
> **Ride Service** -> **Notification Service** -> **Отправляет предложение водителю**.
>
> **Водитель** (принимает заказ) -> **Ride Service** (обновляет статус) -> **Notification Service** (уведомляет пассажира).

Этап 2. Расчёт нагрузки и проектирование данных

2.1 Оценка нагрузки

Используем вводные данные: **100M MAU** (Monthly Active Users) и **10M DAU** (Daily Active Users).

A. Трафик

Предположим следующие поведенческие паттерны:

1. **Поездки:** В среднем каждый активный пользователь (DAU) делает 1 поездку в день.
   - Всего поездок в день: **10 миллионов**.
2. **Водители:** Обычно соотношение водителей к активным пассажирам около 1:20 или 1:10.
   - Возьмем **1M активных водителей** (Daily Active Drivers), находящихся на линии.
3. **RPS (Запросы в секунду) на создание поездок:**
   - В сутках 86400 секунд ($\approx 10^5$).
   - $10000000 \text{ trips} / 100000 \text{ sec} = \mathbf{100} \text{ RPS}$ (среднее).
   - В пиковые часы (утро/вечер) коэффициент x5: **~500 RPS** на создание заказов.
4. **RPS на обновление геолокации (самое узкое место):**
   - Водитель отправляет координаты каждые **5 секунд**.
   - $1,000,000 \text{ drivers} / 5 \text{ sec} = \mathbf{200,000} \text{ RPS}.$
   - *Вывод:* Обычная реляционная база (Postgres/MySQL) не выдержит 200k записей в секунду без сложного шардирования. Нам нужно in-memory решение для "горячих"

данных.

B. Хранилище

1. **Профили пользователей:**
   - 100M пользователей. Пусть профиль занимает 1 KB.
   - $100\ \text{M} \times 1\ \text{KB} = \mathbf{100}\ \text{GB}$. (Легко помещается в одну RDBMS, но лучше шардировать/реплицировать).
2. **История поездок (Metadata):**
   - 10M поездок в день. Запись о поездке (ID, who, where, when, cost) $\approx$ 1 KB.
   - День: 10 GB.
   - Год: $10\ \text{GB} \times 365 \approx \mathbf{3.65}\ \text{TB}$.
   - За 5 лет: $\approx \mathbf{18}\ \text{TB}$. (Здесь нужна БД, которая хорошо пишет и горизонтально масштабируется).
3. **GPS-треки:**
   - Мы не храним все 200k RPS вечно. Мы храним их только во время активной поездки в кэше, а по завершении сбрасываем "сжатый" трек в холодное хранилище или БД истории.

2.2 Выбор баз данных (Data Storage Design)

В соответствии с выбором подходящих баз данных, мы разделим данные на 3 категории:

1. "Горячие" данные и Геолокация (High Performance)

- **Что храним:** Текущее местоположение водителей, статус (Online/Busy), сессии.
- **Требования:** Сверхбыстрая запись (200k RPS), быстрый гео-поиск.
- **Технология: Redis (Cluster)**.
  - Используем `GEOADD` и `GEORADIUS` для поиска.
  - Данные эфемерны: если сервер упадет, водители просто пришлют координаты снова через 5 секунд. (Жертвуем Durability ради Performance — теорема PACELC).

2. Транзакционные данные (OLTP / ACID)

- **Что храним:** Профили пользователей, водителей, платежные данные, *активные* заказы (состояние машины автомата заказа).

- **Требования:** Строгая консистентность (ACID). Нельзя списать деньги дважды или потерять заказ.
- **Технология: PostgreSQL** (с шардированием по `User_ID`).
    - Почему не NoSQL? Нам нужны транзакции и надежность связей (Foreign Keys) для биллинга.

3. Архивные данные и История (Big Data)

- **Что храним:** Завершенные поездки, логи действий, чаты.
- **Требования:** Огромный объем (десятки ТБ), редкие обновления, высокая скорость записи (Write-heavy).
- **Технология: Cassandra** (или ScyllaDB/HBase).
    - Колоночная БД идеально подходит для хранения истории (Time-series data). Линейно масштабируется при росте объема данных.

2.3 Схема данных (Data Schema)

Предположим структуру основных таблиц:

PostgreSQL (Users & Drivers):

```
Table Users {
    user_id: PK, BigInt (Snowflake/UUID)
    email: Varchar, Unique
    phone: Varchar
    payment_methods: JSON
}

Table Drivers {
    driver_id: PK
    user_id: FK
    car_plate: Varchar
    status: Enum (Online, Offline, InRide)
    rating: Float
}
```

PostgreSQL (Active Rides - "State Machine"):

Для активных поездок нам важен статус. Как только поездка завершена — переносим в

Cassandra.

```
Table ActiveRides {
    ride_id: PK
    passenger_id: FK, Index
    driver_id: FK, Index
    status: Enum (Requested, Matching, PickedUp, InProgress)
    pickup_lat: Float
    pickup_lon: Float
    created_at: Timestamp
}
```

Redis (Geo Location):

Используем структуру Geospatial Index.

```
Key: "drivers:location"
Value: GeoSet (Longitude, Latitude, Member: driver_id)
TTL: 15 seconds (автоудаление, если водитель пропал со связи)
```

Cassandra (Ride History):

Оптимизировано для просмотра историии поездок.

```
Table RidesByPassenger {
    passenger_id: Partition Key
    created_at: Clustering Key (DESC) -- для сортировки по дате
    ride_id: UUID
    driver_id: UUID
    start_loc: Text
    end_loc: Text
    total_cost: Decimal
    trace_url: Text -- ссылка на S3 с детальным треком GPS
}
```

Этап 3. Геолокация, масштабирование и алгоритмы

3.1 Геолокация и обработка координат

Нам нужно обрабатывать 200,000 обновлений позиций в секунду (Write Heavy) и тысячи запросов на поиск (Read Heavy).

Алгоритм хранения (GeoHash vs QuadTree)

Простой поиск по координатам (`WHERE lat > x AND lon > y`) работает медленно. Нам нужен пространственный индекс.

- **Выбор:** Мы будем использовать **GeoHash** (строковое представление координат, где общий префикс означает близость) внутри **Redis**.
- **Преимущества** Redis поддерживает Geo-команды из коробки (`GEOADD` , `GEORADIUS`) и работает в оперативной памяти, что критично для низкой задержки.

Поток данных (Data Flow):

1. **Driver App** открывает постоянное **WebSocket** соединение с `Location Service` . Это эффективнее, чем слать HTTP-запросы каждые 5 секунд.
2. **Location Service** получает координаты `{driver_id, lat, lon}` .
3. Сервис пишет данные в **Redis**:
    - Команда: `GEOADD drivers:active <lon> <lat> <driver_id>`
    - Устанавливает TTL (время жизни) записи в 15-20 секунд. Если водитель потерял связь, он автоматически пропадет с карты.

3.2 Стратегия Шардирования (масштабирование)

Один Redis сервер не выдержит весь мир (поскольку память и CPU ограничены). Нам нужно разбить данные.

Почему стандартный Hash-шардинг не подходит?

Если мы сделаем шардинг по `driver_id` (как в обычном вебе), то водители одного города окажутся на разных серверах. Чтобы найти "всех водителей в Нью-Йорке", нам придется опрашивать *все* шарды. Это неэффективно.

Решение: Гео-шардинг (Region-based Sharding)

Мы разбиваем мир на регионы (города или крупные гео-зоны).

1. **Map Service:** Хранит маппинг `Region_ID -> Redis_Cluster_IP`.
2. Когда водитель шлет координаты, мы определяем, в каком он городе, и пишем в соответствующий Redis.
3. Когда пассажир ищет машину, мы ищем только в Redis-кластере его города.

Нюанс (Edge case): Если пользователь на границе двух зон, мы ищем в двух соседних шардах.

3.3 Алгоритм Матчинга для распределения заказов

Как только `Ride Service` получает заказ, он обращается к `Location Service`.

Алгоритм "Жадный с блокировкой" (Greedy with Locking):

1. **Поиск:** `Location Service` делает запрос в Redis (`GEORADIUS`) и находит, скажем, 10 ближайших водителей в радиусе 2 км.
2. **Фильтрация:** Исключаем тех, кто уже занят (проверка статуса в Redis или кэше состояний).
3. **Сортировка:** Считаем реальное время подачи (ETA) через OSRM (Open Source Routing Machine) или Google Routes. Близость по прямой (Redis) ≠ быстрота подачи.
4. **Блокировка (Soft Lock):**
   - Мы не можем отправить заказ всем 10 водителям одновременно (Race condition).
   - Мы выбираем ТОП-1 водителя и ставим на него временную блокировку (в Redis: `SET driver:lock:{id} ride:{id} EX 15`).
5. **Предложение:** Отправляем водителю Push/Socket уведомление. У него есть 15 секунд.
   - *Принял:* Отменяем блокировку, помечаем статус `BUSY`, уведомляем пассажира.
   - *Отказал / Тайм-аут:* Снимаем блокировку, берем следующего водителя из списка.

3.4 Динамическое ценообразование (Surge Pricing)

Чтобы считать цену в реальном времени, нам нужно знать баланс спроса и предложения.

1. Мы используем **GeoHash** (длиной 5-6 символов, точность ~1км).
2. В Redis заводим счетчики для каждого хеша:
   - `demand:{geohash}` — инкрементируем при открытии приложения/заказе.
   - `supply:{geohash}` — количество активных водителей в этом хеше (берем из Location Service).
3. Каждые пару минут воркер считает коэффициент: $Ratio = Demand / Supply$.
4. Если $Ratio > 1$, включаем множитель цены (x1.2, x1.5 и т.д.).

3.5 Карты и Маршруты

Хранить карту всего мира и строить маршруты — это очень сложно и дорого (графы, петабайты данных).

- **Решение для MVP:** Использовать внешние API (Google Maps, Mapbox) для отображения карт на клиенте.
- **Для расчета времени/расстояния (Backend):** Чтобы не разориться на Google API, мы поднимем свой кластер **OSRM** (Open Source Routing Machine) на базе карт OpenStreetMap. Это позволит делать тысячи расчетов ETA бесплатно и быстро.

3.6 Безопасность (Security Scope)

Для продукта сервиса такси важно включить основы безопасной разработки. Это критично, так как мы храним данные карт, маршруты и персональные данные.

1. Шифрование и передача данных:

- **Data in Transit:** Весь трафик (Mobile <-> API Gateway <-> Microservices) шифруется через **TLS 1.3** (HTTPS/WSS).
- **Data at Rest:** Чувствительные данные в БД (Postgres, S3) лежат на зашифрованных дисках (AES-256).

2. Аутентификация и Авторизация:

- **Auth Service:** Выдает **JWT (JSON Web Tokens)** при логине.

    - Токены содержат `user_id` и `role` (driver/passenger).
    - Токены имеют короткий срок жизни (например, 15 минут), обновляются через `Refresh Token`.
- **API Gateway:** Проверяет подпись JWT на входе. Не пускает запросы без валидного токена внутрь периметра микросервисов.

3. Управление секретами:

- Не храним пароли от БД и API ключи в коде.
- Используем **HashiCorp Vault** (или AWS Secrets Manager). Сервисы при старте получают нужные доступы оттуда.

4. Защита персональных данных:

- Для аналитиков (в Data Lake / ClickHouse) данные **обезличиваются**. Аналитик видит "Пользователь 123 проехал 5 км", но не видит имя и телефон.
- Телефонные номера маскируются при звонке: водитель и пассажир звонят на подменный номер сервиса, который переадресует вызов.

Схема взаимодействия компонентов

> 1. **Пассажир** создает заказ -> **Ride Service**.
> 2. **Ride Service** -> **Pricing Service** -> Проверяет GeoHash зоны, отдает цену.
> 3. **Ride Service** -> **Matching Service**.
> 4. **Matching Service** -> **Location Service (Redis Shard)** -> Находит кандидатов.
> 5. **Matching Service** -> **OSRM** -> Считает точный ETA.
> 6. **Matching Service** -> **Notification Service** -> Присылает заказ водителю.

Этап 4. Big Data, ML и масштабирование

На этом этапе мы превращаем «просто работающий сервис» в интеллектуальную платформу, способную обрабатывать **1 миллион событий в секунду** (телеметрия, клики, статусы) и использовать эти данные для оптимизации бизнеса.

4.1 Сбор данных (Data Ingestion)

Обработать 1M RPS синхронно (записью в базу) невозможно и дорого. Нам нужен буфер.

Архитектура конвейера (Pipeline)

1. **Источники:** Приложения водителей (GPS раз в 1-5 сек), приложения пассажиров (поиск, клики), внутренние сервисы (логи транзакций).
2. **Сборщик (Log Service):** Легковесный сервис (на Go/Rust/C++), который принимает HTTP/UDP пакеты, валидирует их и сразу отправляет в брокер сообщений.
3. **Брокер сообщений: Apache Kafka**:
   - *Почему Kafka?* Высочайшая пропускная способность, персистентность (хранит историю), возможность многократного чтения (replay) для разных консьюмеров.

Структура Топиков (Kafka Topics):

- `telemetry_gps` : Координаты, скорость, ID водителя. (Самый объёмный топик).
  - *Partition Key:* `driver_id` (чтобы трек одного водителя всегда попадал в одну партицию и сохранялся хронологический порядок).
- `ride_events` : Создан заказ, принят, отменен, завершен.
- `app_events` : Клики, просмотры экранов (для аналитики воронок).

4.2 ETL и Обработка (Processing)

Данные из Kafka расходятся по двум путям: **Горячий путь (Real-time)** и **Холодный путь (Batch)**.

A. Горячий путь (Stream Processing)

Используем **Apache Flink** или **Spark Streaming**.

- **Задача:** Агрегация на лету.
- **Пример:** Считаем количество свободных машин в каждом GeoHash за последние 5 минут для сервиса динамического ценообразования.
- **Fraud Detection:** Выявление аномалий (водитель «телепортировался», поездка длится 10 часов).

B. Холодный путь (Data Lake)

- **Задача:** Долгосрочное хранение и обучение ML.
- **Kafka Connect:** Автоматически выгружает данные из топиков в объектное хранилище (Amazon S3 / Hadoop HDFS).
- **Формат:** Parquet (колоночный формат, сжимается в 10 раз лучше JSON).
- **Аналитика:** Поверх S3 настраиваем **ClickHouse** или **Presto** для SQL-запросов аналитиков (BI-отчеты).

4.3 Внедрение ML-компонент

Модели обучаются на данных из «Холодного пути», а применяются в реальном времени.

ML 1: Прогнозирование времени подачи (ETA)

- *Проблема:* OSRM считает "чистое" время по графу дорог, не учитывая пробки, погоду и привычки водителя.
- *Решение:* Модель (Gradient Boosting / Neural Net), которая берет базовый ETA от OSRM и корректирует его на основе исторических данных.
- *Inference:* Сервис ETA запрашивает модель при каждом расчете матчинга.

ML 2: Динамическое ценообразование (Surge Pricing)

- *Вход:* Текущий спрос (из Kafka Stream), прогноз спроса на час вперед (ML), погода, наличие концертов.
- *Выход:* Коэффициент цены для конкретного GeoHash.
- *Хранение:* Результат расчета кладется в Redis каждые 5 минут. `Ride Service` просто читает готовый коэффициент.

ML 3: Перераспределение водителей (Dispatch Optimization)

- *Проблема:* Утром все едут в центр, спальные районы пустеют.
- *Решение:* Предсказать дефицит в районе X через 30 минут.
- *Реализация:* Показывать водителям зоны с повышенными коэффициентами.

ML 4: Кластеризация и Персонализация (Passenger LTV & Upsell)

Цель: Увеличить средний чек и удержание (как в Яндекс.Такси или Uber).

Как это работает:

1. **Сбор фичей (Feature Engineering):**
   - *Исторические:* Как часто ездит? Ездит ли в экономе, но возвращается на комфорте? Оставляет ли чаевые?
   - *Контекстные:* Какой сейчас заряд батареи? (Если 5% — человек готов заплатить больше, чтобы уехать срочно). Какая модель телефона? (iPhone 17 Pro vs Xiaomi Ultra Max 200MP). Где находится? (ресторан, аэропорт, университет).
2. **Модель:** Используем алгоритмы кластеризации (K-means) и рекомендательные системы (Matrix Factorization или Contextual Bandits).
3. **Применение (Inference):**
   - **Сценарий А (Upsell):** Пользователь открывает приложение. Модель видит: "Пятница, вечер, пользователь с высоким LTV (Lifetime Value)".
   - *Действие:* Показать тариф "Комфорт+" первым в списке или предложить скидку 10% на "Бизнес", чтобы спровоцировать пробу более дорогого тарифа.
   - **Сценарий Б (Cross-sell):** Пользователь едет домой.
   - *Действие:* Предложить заказать еду (Яндекс.Еда/Eats) в эту же точку через 15 минут.

ML 5: Оценка безопасности вождения (Driver Safety Scoring)

Цель: Снижение аварийности и расходов на страховку, блокировка опасных водителей.

Как это работает:

1. **Сбор данных (Telematics):** Приложение водителя собирает данные с акселерометра и гироскопа телефона с частотой 50Hz (при отсутствии собственных машин).
2. **События:** Резкое торможение, резкий разгон, "шашки" (резкие перестроения), превышение скорости (сравниваем GPS скорость с разрешенной на этом участке карты).
3. **Модель:** Классификатор (Random Forest или LSTM для временных рядов), который отличает экстренное торможение от агрессивного вождения.
4. **Результат:**
   - Водитель получает "Score" (0-100).
   - Если балл падает — система присылает предупреждение.
   - Если не исправляется — временная блокировка или пониженный приоритет в выдаче заказов.

4.4 Надежность и Мониторинг (Reliability)

1. **Rate Limiting (Ограничение нагрузки):**
   - Защита от DDoS и сбоящих клиентских приложений.
   - Ставим **API Gateway** с алгоритмом "Token Bucket" (Ведро с токенами). Ограничиваем RPS на создание поездок и смену статуса.
2. **Graceful Degradation (Плавная деградация):**
   - Если упал сервис `Pricing (ML)` , мы не блокируем заказы. Мы просто считаем коэффициент по базовому алгоритму.
   - Если упал `Redis Geo` , мы ищем водителей через `Postgres` (медленнее, но работает) или расширяем радиус поиска.
3. **Мониторинг:**
   - **Метрики (Prometheus/Grafana):** RPS, Latency (p95, p99), Error Rate, Lag в Kafka (если консьюмеры не успевают).
   - **Distributed Tracing (Jaeger):** Чтобы видеть, где именно тормозит запрос (в матчинге, в базе или в ML-модели).

Финальная Архитектура

> 1. **Mobile Apps** -> **LB** -> **API Gateway**.
> 2. **API Gateway** -> **Rate Limiter**.
> 3. **Services:**
>   * **Ride Service** (Оркестратор).
>   * **Location Service** (Redis Geo + WebSockets).
>   * **Matching Service** -> **OSRM** + **ML ETA Model**.
> 4. **Async Flow (Kafka):**
>   * Все события летят в **Kafka**.
>   * **Stream Processors** считают метрики спроса.
>   * **Data Lake (S3)** копит петабайты истории.
> 5. **Databases:**
>   * **Postgres** (Биллинг, Юзеры).
>   * **Cassandra** (История поездок).
>   * **Redis** (Кэш, Гео, Сессии).