ChatGPT

Workflow and Architecture for Using Google Maps API Under Free Tier Limits

Introduction

The purpose of this document is to define a detailed workflow and **technical architecture** for a restaurant/delivery platform that uses the Google Maps Platform **within the free tier** (≈10,000 calls per SKU per month). The goal is to ensure that the system delivers accurate address validation, route calculation, estimated times of arrival (ETA), and tracking while **controlling API usage** to avoid unexpected costs. Insights in this document are grounded in case studies and Google best practices for API cost management [1].

Assumptions

- **Order volume:** ~50 orders/day, ~1,500 orders/month (well under the free 10,000 requests per SKU).
- **Platform:** A web/mobile app for customers and drivers, plus a cloud backend.
- **Usage patterns:** Customers check the tracking map once or twice per order; routing is computed once per order with occasional re-routing on deviations.
- **Rate limit:** Each API SKU offers ~10,000 free calls per month [1]; staying below this threshold is critical.

Architecture Overview

The system is divided into three layers: **frontend**, **backend**, and **external services**. It uses event-driven logic and caching to minimise API calls. A high-level architecture diagram is shown below:

```
flowchart LR
    subgraph Client (Web / Mobile)
        C1[Customer App]
        C2[Driver App]
    end

    subgraph Backend
        B1[Order Service]
        B2[Geocode Service]
        B3[Routing Service]
        B4[Route Cache]
        B5[Database]
        B6[Notification Service]
    end
    subgraph External
        E1[Google Maps Platform]
    end

    C1 -->|Place order| B1
    B1 -->|Lookup customer & kitchen coords| B2
    B2 -->|Check cache<br>for geocode| B4
    B4 -->|Miss| E1
    E1 -->|Geocode API| B2
    B3 <--|Route once per order| B1
    B3 -->|Check cache| B4
    B3 -->|Compute Route<br>(Directions/Routes API)| E1
    E1 -->|Return route| B3
    B4 -->|Store route & ETA| B3
    B3 --> B5
    B5 -->|Push updates to drivers & customers| B6
    C1 & C2 <--|Real-time updates & map render| B6

    classDef external fill:#FDF6C6,stroke:#DDC;
    class E1 external;
```

Key components

- **Customer App** – Lets users place orders, check order status and track drivers. Loads the delivery map only when needed, minimising map loads.
- **Driver App** – Provides pickup and delivery instructions. Uses simple markers and polylines instead of recomputing routes constantly.
- **Order Service** – Manages order lifecycle and orchestrates geocoding, routing, caching and notifications.
- **Geocode Service** – Resolves addresses to latitude/longitude using `Geocoding API` and `Places API`. Caches results (up to 30 days) and stores `place_id` for reuse [2].
- **Routing Service** – Calls the `Directions API` or `Routes API` once per order to compute optimal routes and ETAs. Checks the **route cache** before calling the API and stores results with a TTL (2–5 min) [3].
- **Route Cache** – Stores geocode results (30 days) and route results (short TTL) to avoid duplicate requests.
- **Notification Service** – Sends status updates to drivers and customers via websockets or push notifications. ETA updates are computed locally based on cached route distance and driver speed.
- **Google Maps Platform** – Provides geocoding, routing and static map tiles; usage is controlled via event-driven calls and caching.

Detailed Design Flow

This section outlines the sequence of operations from order placement to delivery tracking. It emphasises **rate-limited** API calls, caching and event-driven updates.

1. Order Placement and Address Resolution

1. **Order placement:** A customer selects items and places an order via the app.
2. **Address validation:** When the address is entered, the frontend uses `Autocomplete API` with a **session token** so multiple keystrokes are billed as a single session [4].
3. **Geocoding:** The backend checks the **geocode cache** for the `place_id`. If not found, it calls the **Geocoding API**; the result (lat/lng and `place_id`) is cached for 30 days [2].
4. **Store order:** The order and customer/kitchen coordinates are stored in the database.

2. Driver Assignment and Route Calculation

1. **Driver shortlist:** The Order Service selects a small pool of drivers based on straight-line (Haversine) distance; this avoids expensive matrix calls [3].
2. **Route computation:** For the assigned driver:
3. **Check route cache:** If a recent route with the same origin/destination exists, reuse it.
4. **Call `Routes API` or `Directions API`:** Compute the optimal route once per order, limiting waypoints (don't optimise for multiple stops unless needed) [5].
5. **Store results:** Save the polyline, distance and ETA in cache and database.
6. **Send pickup instructions:** The driver app receives the polyline and uses it for navigation (no additional API calls).

3. Tracking and ETA Updates

1. **GPS streaming:** The driver app sends location pings (via device GPS) to the backend. GPS data **does not incur API usage**.
2. **Local ETA calculation:** The backend computes remaining distance using the cached polyline and a rolling average of driver speed. ETA is updated server-side without calling the `Routes API` again.
3. **Event-based updates:** Push a limited number of status updates (e.g., order accepted, driver en route, driver near arrival, delivered) to customers instead of continuous ETA polling [3].
4. **Re-routing (optional):** Only call the Routes API again if the driver deviates by >150 m or at defined time intervals (e.g., every 5–10 min). This event-driven approach prevents "per-ping" billing.

4. Map Rendering

1. **Load map on demand:** The customer app loads the **Dynamic Maps** SDK only when the user opens the tracking screen, minimising map loads [6].
2. **Show markers and polylines:** The cached route is drawn on the map; driver location is updated from server data; there is no additional call to the Directions API.
3. **Unload map:** When the user navigates away, the map is destroyed to prevent hidden map loads.

API Usage Control Techniques

The design incorporates several controls recommended by Google and industry case studies to ensure the system stays under free limits:

Control	Description	Evidence
Cache geocode results	Store geocoding responses (lat/lng, `place_id`) for up to 30 days to avoid repeated calls. Cache results keyed by normalised address [2].	Google permits caching place IDs and suggests caching geocode responses for 30 days [2].
Route caching with TTL	Store route results (polyline, distance, ETA) for 2–5 minutes. Reuse results if a similar request arrives within the TTL.	8allocate used event-based updates and caching to cut API costs drastically [3].
Event-driven ETA updates	Update ETA only on significant events (e.g., departure, arriving at key waypoints) instead of polling every few seconds.	Constant ETA polling led to $25k/ month bills; switching to event-based triggers reduced costs to near zero [7].
Limit waypoints and features	Avoid unnecessary waypoints, avoid real-time traffic models unless needed, and request only required fields via field masks [4] [5].	Google recommends limiting waypoints and using field masks to reduce costs [4].
Use session tokens for Autocomplete	Group multiple keystrokes into a single Places session so you pay once per autocomplete session rather than per keystroke [4].	Session tokens are explicitly recommended by Google to reduce charges [4].
Load maps sparingly	Load dynamic maps only when necessary and reuse map instances; avoid multiple map loads per order [6].	Google's cost-management page advises loading maps only when needed and reusing a single map instance [6].
Shortlist drivers locally	Use Haversine distance in the backend to narrow down a list of candidate drivers; then call the Distance Matrix or Routes API only for the top candidates [3].	Reducing the number of API calls by pre-filtering drivers avoids costly matrix calculations.
Monitor usage and set quotas	Use Google Cloud Console to set daily quotas and receive alerts when approaching the free tier; implement circuit breakers in code.	Google suggests monitoring usage and controlling quotas to prevent runaway costs (implicit from pricing guidelines).
Separate API keys and restrict them	Use separate API keys for web, mobile and server; restrict each key by IP, referrer or package name; enforce quotas per key.	Google's security best practices recommend restricting API keys and monitoring usage per key [2].

Implementation Considerations

1. **Backend scaling:** Use a serverless or auto-scaling backend (e.g., Google Cloud Run or Kubernetes) to handle spikes in orders; this avoids over-provisioning and supports event-driven functions.
2. **Database and caching:** Use a relational database for persistent order data (e.g., Postgres). Use an in-memory store (e.g., Redis) for the route cache and geocode cache. Ensure TTLs are enforced.
3. **API rate limiting:** Implement a shared rate-limiting middleware for all Google API calls. When the limit is close, degrade gracefully (e.g., serve cached results or local estimates). Introduce jitter between retries to avoid hitting the `OVER_QUERY_LIMIT` threshold [8].
4. **Telemetry and cost dashboard:** Record each API call and its SKU. Build dashboards to visualise usage vs. the 10k free tier to detect trends early.
5. **Fallback strategy:** In the event that Google APIs become unavailable or exceed quotas, use local approximations (Haversine distance, average traffic speed) to produce rough ETAs until the next quota period.

Conclusion

By structuring the system around **one geocode call per address**, **one route calculation per order**, caching results, and using event-driven ETA updates, the platform can handle ~1,500 orders/month while staying well within the 10 000-call free tier per API SKU. The architecture emphasises separation of concerns, caching, intelligent triggers and monitoring, ensuring both efficiency and cost control. Adhering to these practices not only protects against unexpected bills but also builds a scalable foundation for future growth.

---

[1] Blog: How SimplyDelivery helps restaurants deliver great food quickly and profitably – Google Maps Platform
https://mapsplatform.google.com/resources/blog/how-simplydelivery-helps-restaurants-deliver-great-food-quickly-and-profitably/

[2] [8] Optimizing Web Service Usage | Google Maps Platform | Google for Developers
https://developers.google.com/maps/optimize-web-services

[3] [7] Client Success Story: How We Cut A $25,000 Google Maps API Bill To Near-Zero With Route-Based Tracking | 8allocate
https://8allocate.com/blog/client-success-story-how-we-cut-a-25000-google-maps-api-bill-to-near-zero-with-route-based-tracking/

[4] [5] [6] Manage Google Maps Platform costs | Pricing and Billing | Google for Developers
https://developers.google.com/maps/billing-and-pricing/manage-costs