stretched and looks pixelated. The visual effect is pretty jarring. With vectorized images, the client can scale each element appropriately, providing a much smoother zooming experience.

Navigation service

Next, let's deep dive into the navigation service. This service is responsible for finding the fastest routes. The design diagram is shown in Figure 3.17.

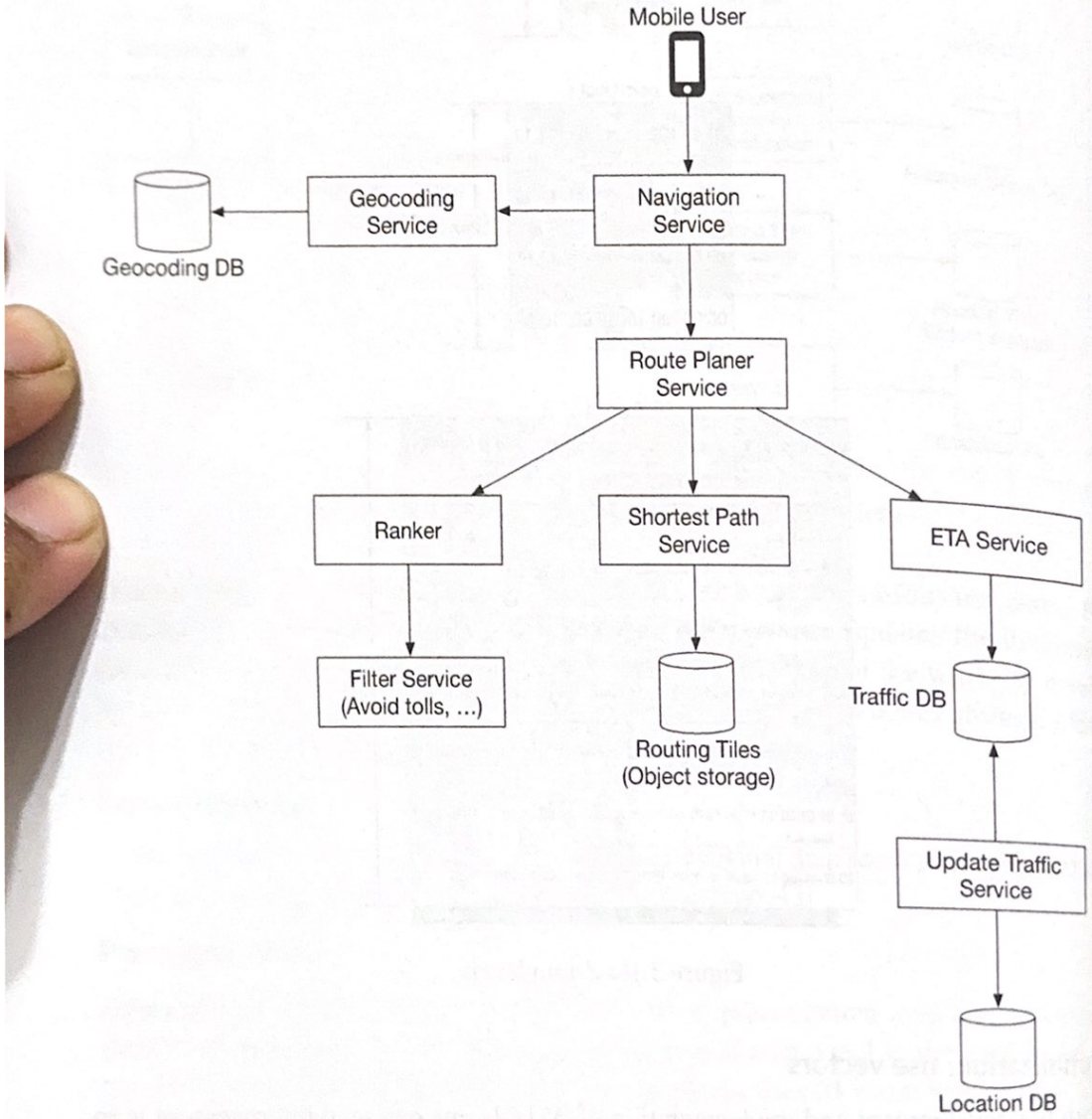


Figure 3.17: Navigation service

Let's go over each component in the system.

Geocoding service

First, we need to have a service to resolve an address to a location of a latitude and longitude pair. An address could be in different formats, for example, it could be the name of a place or a textual address.

Here is an example request and response from Google's geocoding API.

Request:

```
https://maps.googleapis.com/maps/api/geocode/json?address=1600+
  Amphitheatre+Parkway,+Mountain+View,+CA
```

JSON response:

```
{
  "results" : [
  {
    "formatted_address" : "1600 Amphitheatre Parkway, Mountain
       View, CA 94043, USA",
    "geometry" : {
      "location" : {
        "lat" : 37.4224764,
        "lng" : -122.0842499
      },
      "location_type" : "ROOFTOP",
      "viewport" : {
        "northeast" : {
          "lat" : 37.4238253802915,
          "lng" : -122.0829009197085
        },
        "southwest" : {
          "lat" : 37.4211274197085,
          "lng" : -122.0855988802915
        }
      }
    },
    "place_id" : "ChIJ2eUgeAK6j4ARbn5u_wAGqWA",
    "plus_code": {
      "compound_code": "CWC8+W5 Mountain View, California,
         United States",
      "global_code": "849VCWC8+W5"
    },
    "types" : [ "street_address" ]
  }
  ],
  "status" : "OK"
}
```

The navigation service calls this service to geocode the origin and the destination before passing the latitude/longitude pairs downstream to find the routes.

Route planner service

This service computes a suggested route that is optimized for travel time according to current traffic and road conditions. It interacts with several services which are discussed next.

Shortest-path service

The shortest-path service receives the origin and the destination in lat/lng pairs and returns the top-k shortest paths without considering traffic or current conditions. This computation only depends on the structure of the roads. Here, caching the routes could be beneficial because the graph rarely changes.

The shortest-path service runs a variation of A* pathfinding algorithms against the routing tiles in object storage. Here is an overview:

- The algorithm receives the origin and destination in lat/lng pairs. The lat/lng pairs are converted to geohashes which are then used to load the start and end-points of routing tiles.
- The algorithm starts from the origin routing tile, traverses the graph data structure, and hydrates additional neighboring tiles from object storage (or its local cache if it has loaded it before) as it expands the search area. It's worth noting that there are connections from one level of tile to another covering the same area. This is how the algorithm could "enter" the bigger tiles containing only highways, for example. The algorithm continues to expand its search by hydrating more neighboring tiles (or tiles at different resolutions) as needed until a set of best routes is found.

Figure 3.18 (based on [14]) gives a conceptual overview of the tiles used in the graph traversal.

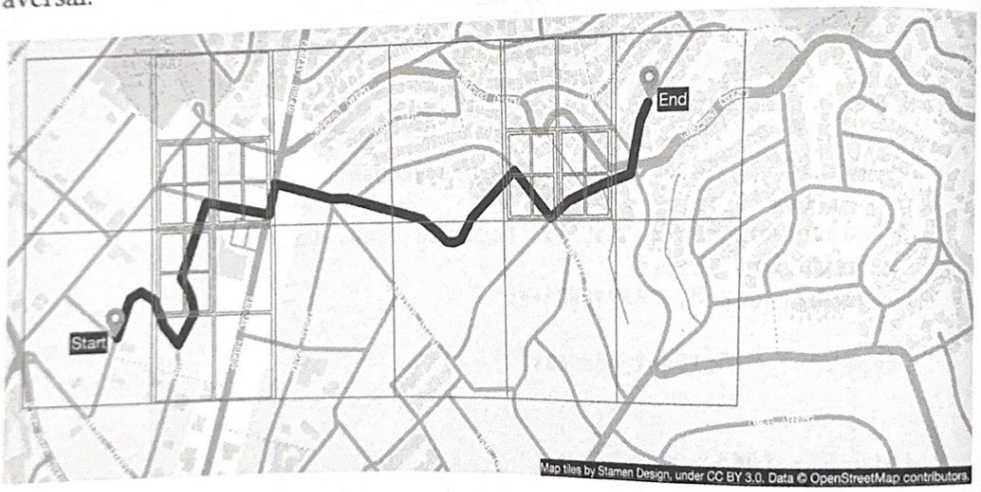


Figure 3.18: Graph traversal

ETA service

Once the route planner receives a list of possible shortest paths, it calls the ETA service for each possible route and gets a time estimate. For this, the ETA service uses machine learning to predict the ETAs based on the current traffic and historical data.

One of the challenges here is that we not only need to have real-time traffic data but also to predict how the traffic will look like in 10 or 20 minutes. These kinds of challenges need to be addressed at an algorithmic level and will not be discussed in this section. If

you are interested, refer to [15] and [16].

Ranker service

Finally, after the route planner obtains the ETA predictions, it passes this info to the ranker to apply possible filters as defined by the user. Some example filters include options to avoid toll roads or to avoid freeways. The ranker service then ranks the possible routes from fastest to slowest and returns top-k results to the navigation service.

Updater services

These services tap into the Kafka location update stream and asynchronously update some of the important databases to keep them up-to-date. The traffic database and the routing tiles are some examples.

The routing tile processing service is responsible for transforming the road dataset with newly found roads and road closures into a continuously updated set of routing tiles. This helps the shortest path service to be more accurate.

The traffic update service extracts traffic conditions from the streams of location updates sent by the active users. This insight is fed into the live traffic database. This enables the ETA service to provide more accurate estimates.

Improvement: adaptive ETA and rerouting

The current design does not support adaptive ETA and rerouting. To address this, the server needs to keep track of all the active navigating users and update them on ETA continuously, whenever traffic conditions change. Here we need to answer a few important questions:

- How do we track actively navigating users?
- How do we store the data, so that we can efficiently locate the users affected by traffic changes among millions of navigation routes?

Let's start with a naive solution. In Figure 3.19, `user_1`'s navigation route is represented by routing tiles `r_1`, `r_2`, `r_3`, ..., `r_7`.

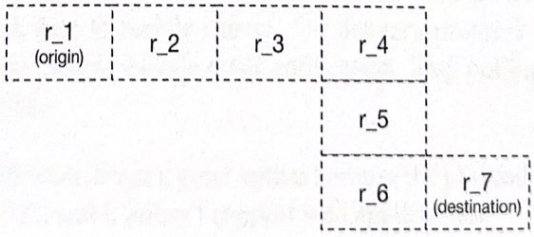


Figure 3.19: Navigation route

The database stores actively navigating users and routes information which might look like this:

`user_1: r_1, r_2, r_3, ..., r_k`

```
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_1
```

Let's say there is a traffic incident in routing tile 2 (`r_2`). To figure out which users are affected, we scan through each row and check if routing tile 2 is in our list of routing tiles (see example below).

```
user_1: r_1, r_2, r_3, ..., r_k
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_1
```

Assume the number of rows in the table is n and the average length of the navigation route is m. The time complexity to find all users affected by the traffic change is $O(n \times m)$.

Can we make this process faster? Let's explore a different approach. For each actively navigating user, we keep the current routing tile, the routing tile at the next resolution level that contains it, and recursively find the routing tile at the next resolution level until we find the user's destination in the tile as well (Figure 3.20). By doing this, we can get a row of the database table like this.

```
user_1, r_1, super(r_1), super(super(r_1)), ...
```

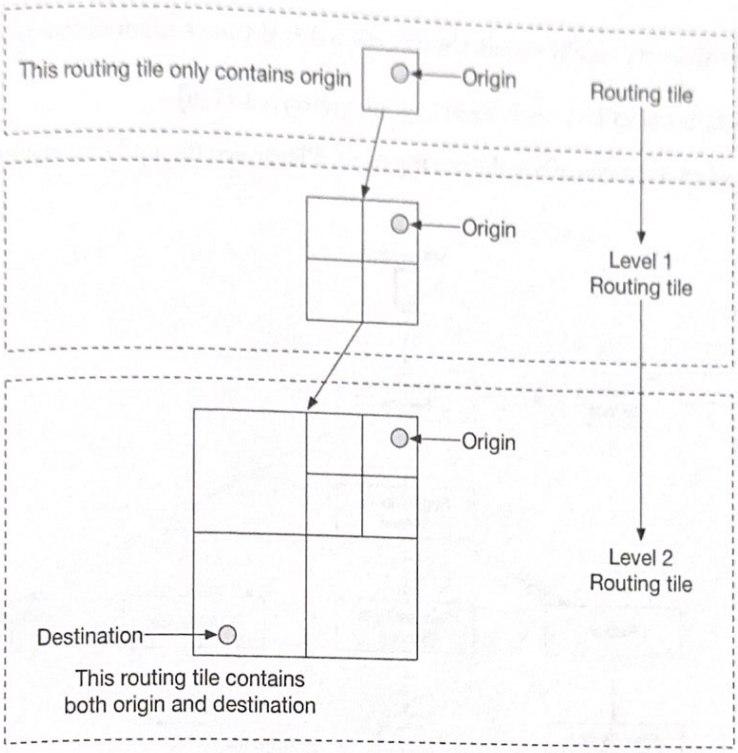


Figure 3.20: Build routing tiles

To find out if a user is affected by the traffic change, we need only check if a routing tile is inside the last routing tile of a row in the database. If not, the user is not impacted. If it is, the user is affected. By doing this, we can quickly filter out many users.

This approach doesn't specify what happens when traffic clears. For example, if routing tile 2 clears and users can go back to the old route, how do users know rerouting is available? One idea is to keep track of all possible routes for a navigating user, recalculate the ETAs regularly and notify the user if a new route with a shorter ETA is found.

Delivery protocols

It is a reality that during navigation, route conditions can change and the server needs a reliable way to push data to mobile clients. For delivery protocol from the server to the client, our options include mobile push notification, long polling, WebSocket, and Server-Sent Events (SSE).

- Mobile push notification is not a great option because the payload size is very limited (4,096 bytes for iOS) and it doesn't support web applications.
- WebSocket is generally considered to be a better option than long polling because it has a very light footprint on servers.
- Since we have ruled out the mobile push notification and long polling, the choice is mainly between WebSocket and SSE. Even though both can work, we lean towards WebSocket because it supports bi-directional communication and features such as

last-mile delivery might require bi-directional real-time communication.

For more details about ETA and rerouting, please refer to [15].

Now we have every piece of the design together. Please see the updated design in Figure 3.21.

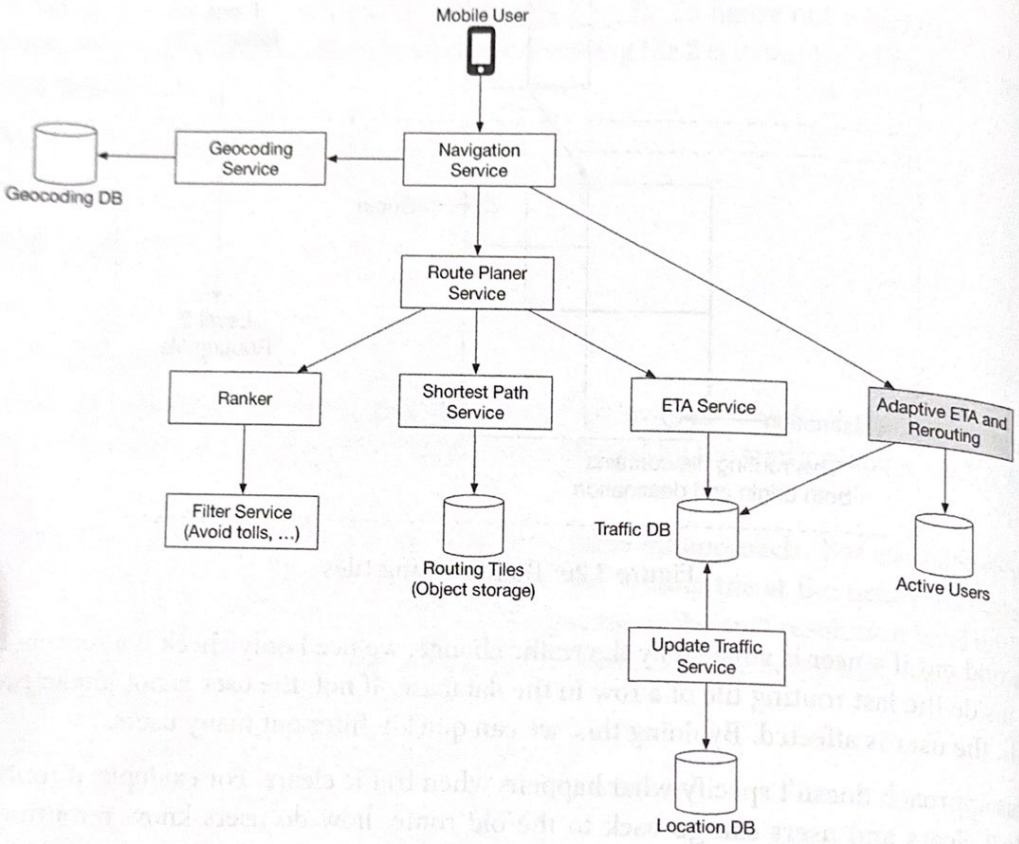


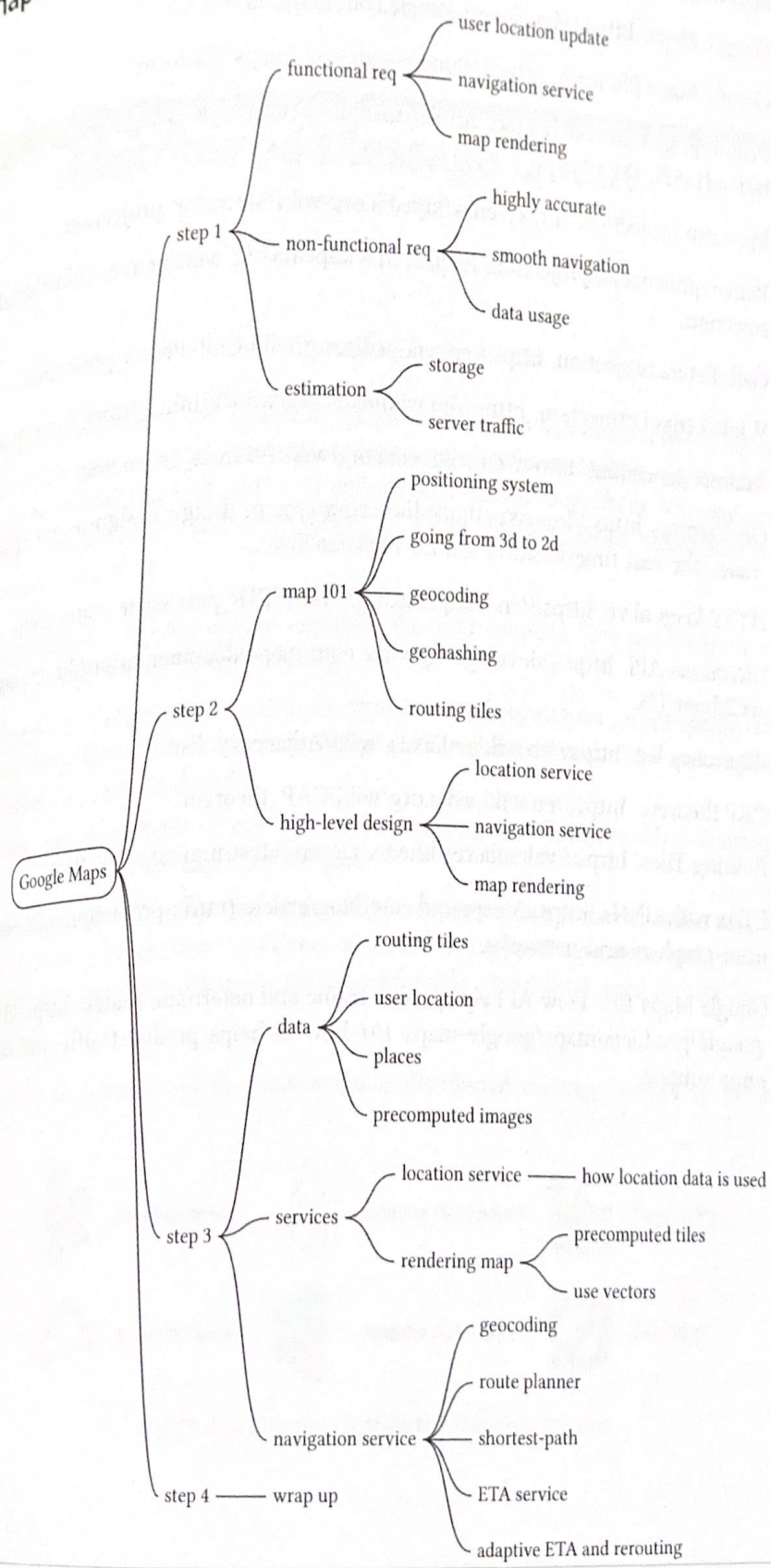
Figure 3.21: Final design

Step 4 - Wrap Up

In this chapter, we designed a simplified Google Maps application with key features such as location update, ETAs, route planning, and map rendering. If you are interested in expanding the system, one potential improvement would be to provide multi-stop navigation capability for enterprise customers. For example, for a given set of destinations, we have to find the optimal order in which to visit them all and provide proper navigation, based on live traffic conditions. This could be helpful for delivery services such as DoorDash, Uber, Lyft, etc.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Google Maps. https://developers.google.com/maps?hl=en_US.

[2] Google Maps Platform. https://cloud.google.com/maps-platform/.

[3] Prototyping a Smoother Map. https://medium.com/google-design/google-maps-b0326d165f5.

[4] Mercator projection. https://en.wikipedia.org/wiki/Mercator_projection.

[5] Peirce quincuncial projection. https://en.wikipedia.org/wiki/Peirce_quincuncial_projection.

[6] Gall–Peters projection. https://en.wikipedia.org/wiki/Gall–Peters_projection.

[7] Winkel tripel projection. https://en.wikipedia.org/wiki/Winkel_tripel_projection.

[8] Address geocoding. https://en.wikipedia.org/wiki/Address_geocoding.

[9] Geohashing. https://kousiknath.medium.com/system-design-design-a-geo-spatial-index-for-real-time-location-search-10968fe62b9c.

[10] HTTP keep-alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.

[11] Directions API. https://developers.google.com/maps/documentation/directions/start?hl=en_US.

[12] Adjacency list. https://en.wikipedia.org/wiki/Adjacency_list.

[13] CAP theorem. https://en.wikipedia.org/wiki/CAP_theorem.

[14] Routing Tiles. https://valhalla.readthedocs.io/en/latest/mjolnir/why_tiles/.

[15] ETAs with GNNs. https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks.

[16] Google Maps 101: How AI helps predict traffic and determine routes. https://blog.google/products/maps/google-maps-101-how-ai-helps-predict-traffic-and-determine-routes/.