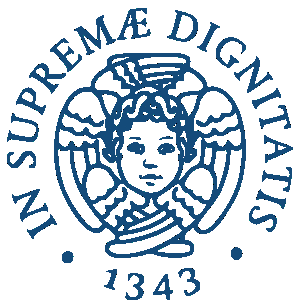


Università di Pisa

Master Of Science in Computer Engineering

Large-Scale and
Multi-Structured Databases

CellMap

Project members:
Dario Bandecchi
Klaudio Ciacia
Francesco De Lucchini

Academic Year 2024/2025

Contents

1 Introduction

1.1 Project description

This document presents CellMap, a community-driven application designed to map Base Transceiver Stations (BTS) and monitor mobile internet speeds across Italy. As mobile networks evolve, especially with the expansion of 5G, it is important to have clear, user-sourced data on coverage and performance, which is usually kept hidden by mobile carriers. CellMap achieves this by allowing users to actively build and update a shared database.

The application collects data primarily when users run speed tests (measuring download, upload, and latency) and share the details of the BTS they are connected to. To keep the map accurate, the system relies on a community validation process. If users notice a missing antenna or incorrect details, they can open a "ticket" to suggest a change. System administrators then review these tickets and decide whether to approve or discard them before updating the main database. Additionally, Internet Service Providers (ISPs) can participate as "host" users by providing the servers needed to run the speed tests.

Beyond data collection, CellMap offers useful analytics. Users can search for a specific location in Italy, find the closest antennas, check their technical characteristics (such as operating bands and 5G availability), and compare the average speeds delivered by different mobile carriers in that area.

2 System requirements

2.1 Main actors

The CellMap application interacts with three primary categories of users, each characterized by specific roles and access permissions within the system:

- **Regular users:** Authenticated members of the community who run internet speed tests and contribute to the database by submitting tickets (e.g., can propose the addition, modification, or deletion of BTSs)
- **Administrators:** System operators responsible for data quality and moderation. They manage the queue of user-submitted tickets and decide whether to approve, modify, or discard proposed changes before they are applied to the main database
- **Host users:** ISPs that choose to host speed-test servers on the platform for public relations and publicity purposes, allowing regular users to test their connections against these specific nodes

2.2 Functional requirements

The core functionalities that the system must provide are the following:

- The system must allow users to register, log in, view their speed-test history, and execute new network performance measurements
- The system must enable users to open tickets when they detect unmapped or broken cells, and must provide administrators with a dedicated interface to process these requests
- Users must be able to search for specific locations or provinces in Italy to locate nearby antennas, view detailed node characteristics (such as operating frequencies and 5G status), and view average performance metrics aggregated by mobile carrier

2.3 Non-functional Rrequirements

The technical constraints defined for the infrastructure are the following:

- The system must be implemented as a RESTful API
- The system must be deployed on a distributed system consisting in at least 3 nodes, and the adopted databases must be replicated on each node of the cluster
- Data consistency must be maintained across the adopted database architectures
- The system must prefer availability and partition tolerance over consistency (for this application, its better to receive outdated data than no data at all)

3 User interface mock-ups

To visualize the functional requirements, several mock-ups have been designed.

3.1 User view

Upon launching the application, users are greeted by the authentication view (left screen). Once logged in, the user view provides the speed-test history (central screen). By clicking a specific speed-test, the user can see its details (right screen)

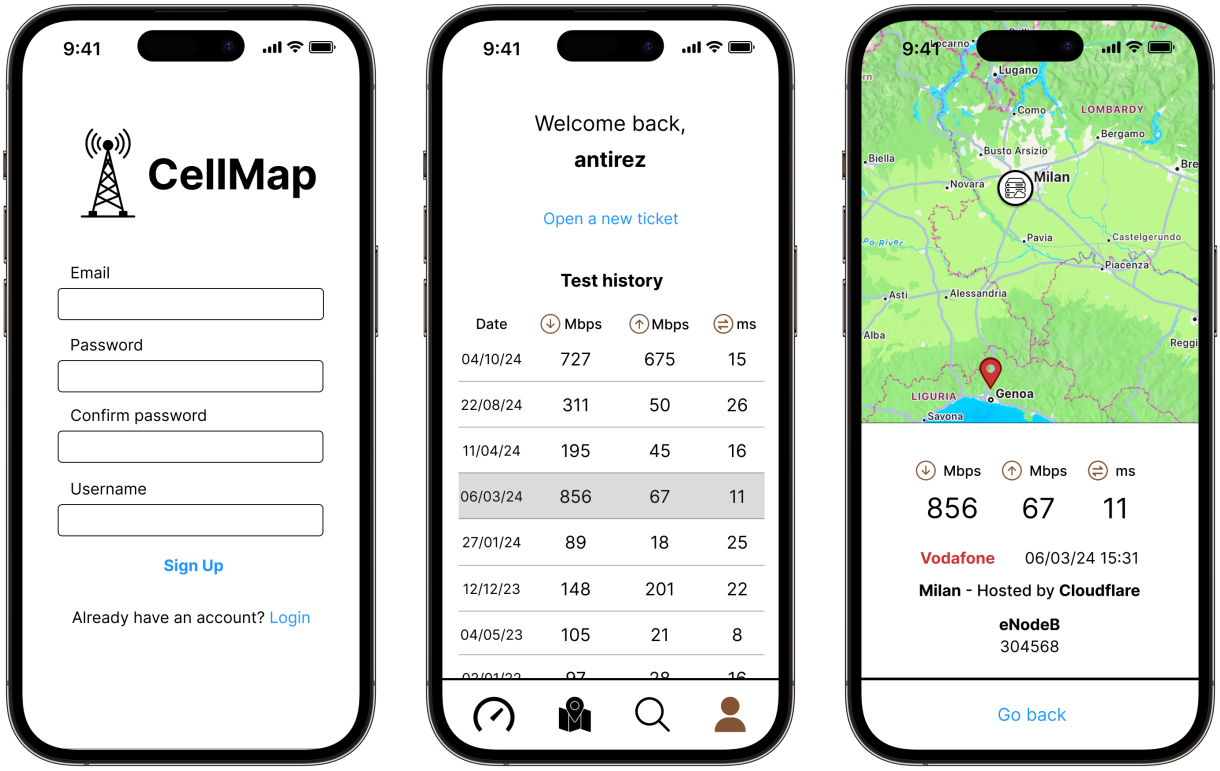


Figure 1: User authentication, speed-test history and speed-test details

3.2 Speed-test view

The main screen contains the "GO" button to start the test (left screen). By clicking on "Change test server", the server list pops up (central screen). Finally, by clicking on a specific server, its details are shown (right screen).

Figure 2: Speed-test screen, servers list and server details

3.3 Map view

The main screen shows the current user location and the nearby BTSs (left screen). By clicking on a BTS, its details are shown (central-left screen). By clicking on "Analyze the speeds", the average download and upload speeds of the selected BTS and carrier are shown (central-right screen). Finally, when a new cell is detected, a popup reminds the user to contribute to the database by opening a new ticket (right screen).

Figure 3: Map screen, BTS details, BTS speeds and "New cell detected" ticket popup

3.4 Search view

When users search for a specific province (e.g. Pisa), a summary of the BTSs present in the area is shown (left screen). Users can also search for a specific location to identify the closest BTS and compare the performance of available carriers (central and right screens).

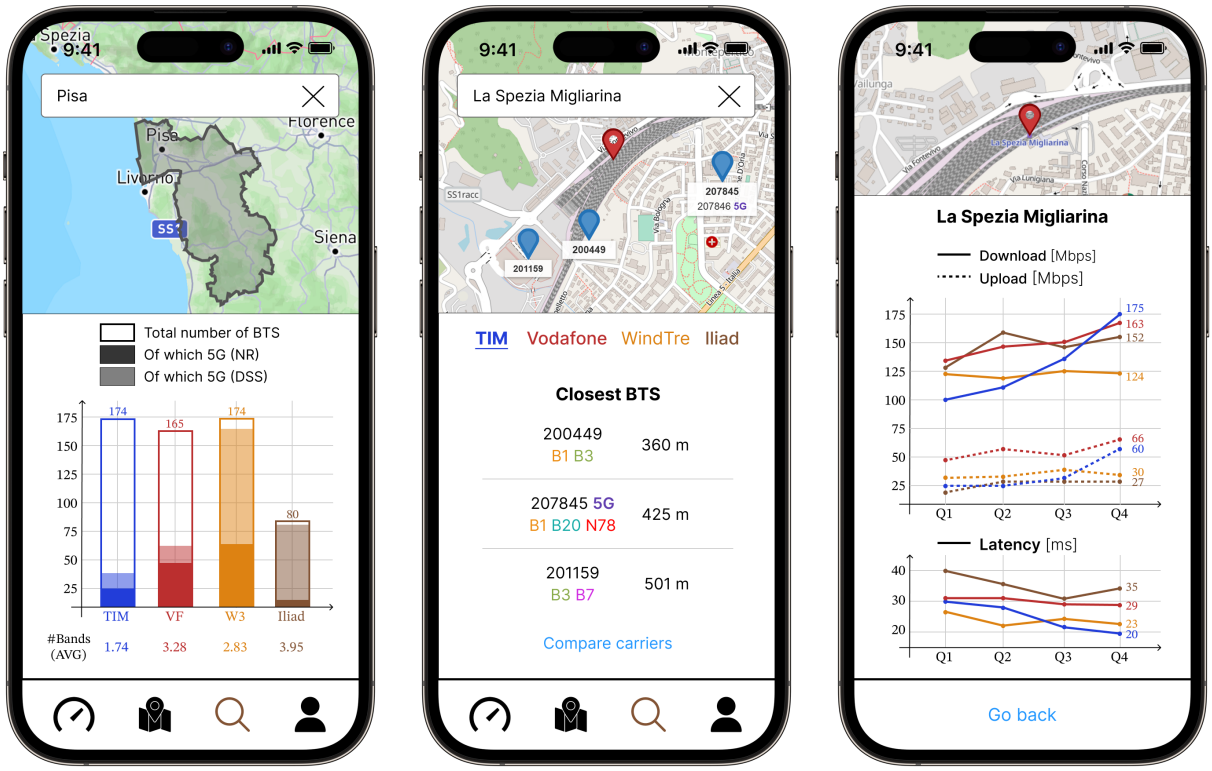


Figure 4: Details of a province, details and comparison charts of a specific place

3.5 Administrator view

System administrators utilize a dedicated ticket management dashboard, which presents a sequential queue of community-submitted tickets.

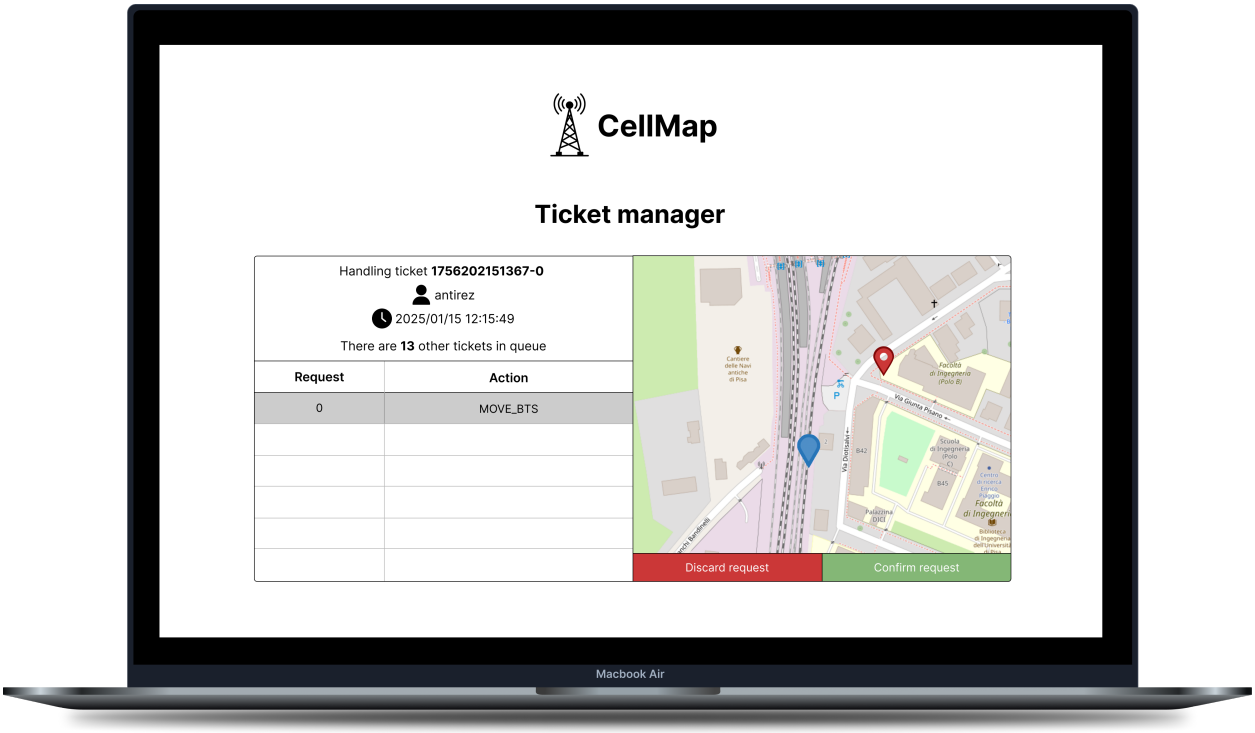


Figure 5: Administrator ticket management dashboard

4 Database design

4.1 UML class diagram

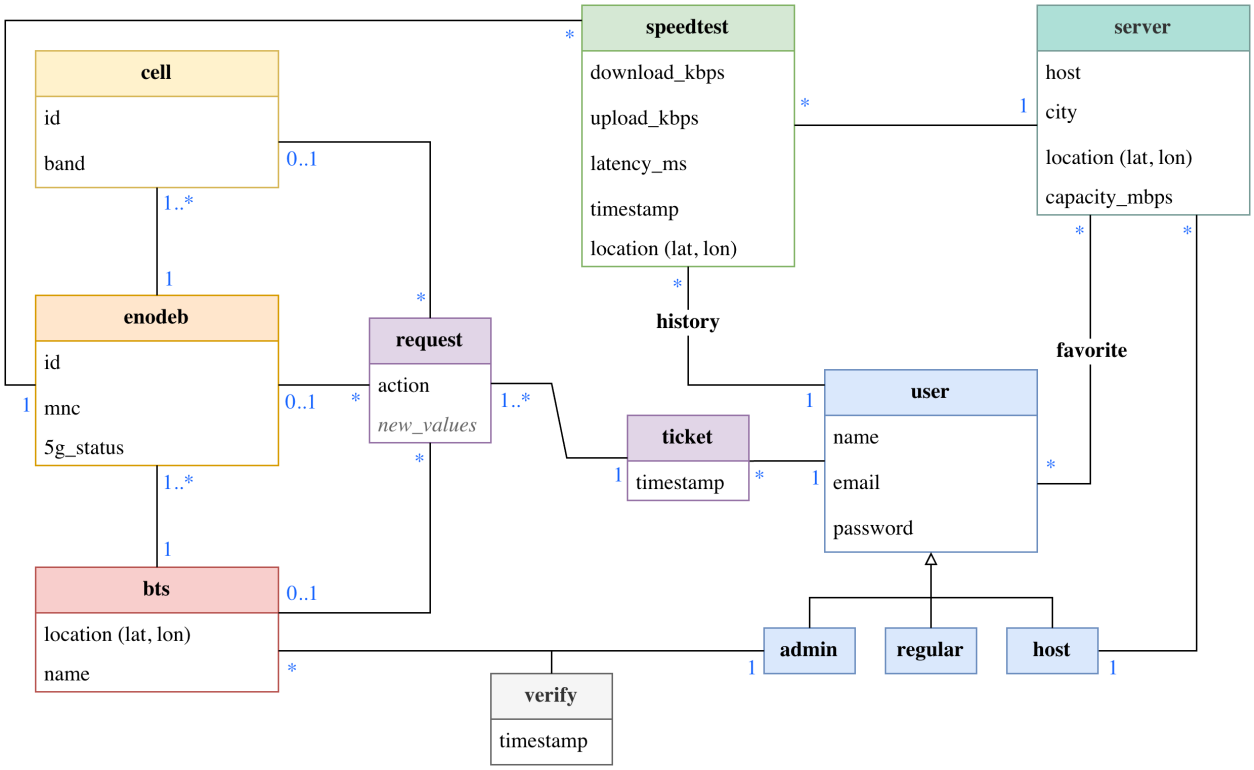


Figure 6: UML class diagram

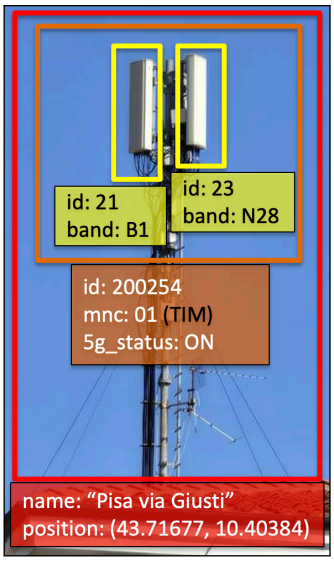
- **Telecommunication hierarchy**

  A *BTS* represents a physical tower, which can host hardware from multiple carriers. An *eNodeB* is basically a router (a control center) and belongs to exactly one carrier (MNC stands for Mobile Network Carrier).

  A BTS is associated with one or more *eNodeBs* (1 to 1..∗), as a tower must have at least one operator to exist in the system, and an eNodeB belongs to exactly one BTS.

  Similarly, an eNodeB handles multiple *Cells* (individual antennas for different bands or 4G/5G technologies), forming another 1 to 1..∗ relationship.

  Note that **a cell id is unique only within the same eNodeB** and **an eNodeB id is unique only within a carrier network**.



- **Speedtests**

  A *Speedtest* is a singular event tied to three specific entities. It is performed on exactly one *Server*, by exactly one *User*, which was connected to exactly one specific *eNodeB*. Conversely, *Users*, *eNodeBs*, and *Servers* can have zero or multiple (*) *Speedtests* associated with them over time

- **Users and servers**

  A *Host* user (e.g., an ISP) manages the physical infrastructure for the tests. A *Host* can provide one or more *Servers* (1 to 1..*), but each *Server* is maintained and owned by exactly one *Host*. Finally, each *User* can have zero or more (*) favorite *Servers*.

- **Ticketing system**

  A *Ticket* is composed by one or more *Requests* (1 to 1..*). For example, if a user wants to report some new cell in an existing eNodeB, he must open one *Ticket* with with as many *Requests* as the number of new cells.

  Each *Request* is linked to **zero or exactly one one of** {*Cell*, *eNodeB*, *BTS*}, is composed by an action and, optionally, a list of values (which represents the new fields of the linked entity). Possible ticket actions are: `ADD_BTS, DEL_BTS, MOVE_BTS, RENAME_BTS ADD_ENODEB, DEL_ENODEB, UPDATE_5G_STATUS_ENODEB ADD_CELL, DEL_CELL` .

  Tickets are deleted after being either merged or discarded by an administrator. If they are merged in the database, the edited *BTS* is updated with the merge timestamp and the administrator who approved the changes.

4.2 Dataset preparation

Before selecting the optimal database architecture, we must know how big the UML entities are and how fast they change. In Table 1 we present the chosen data sources, their post-process[1] size, and their *estimated* velocity.

Domain	Source	Records	Size (MB)	Velocity
Speed-tests	Ookla Open Data (2024)	623,709	163.2	∼1,000/day
BTS	LTE Italy	96,199	56.6	few/day
Servers	Speedtest.net	242	–	few/month
Users	Synthetically generated	20,000	4.7	few/day
Tickets	Synthetically generated	10	–	few/day

Table 1: Summary of data sources

[1]Raw data is filtered, transformed, enriched and re-organized in order to form the entities and the relationships seen in Figure 6.

4.3 Document database

We leverage **MongoDB** for entities that require: *complex analytics* (multi-stage aggregation pipelines), *long-term storage* (data that is retained indefinitely rather than discarded after use), and *large volumes of data* (collections that grow over time and may require horizontal scaling). With respect to Section 4.1, these entities are the *users*, the *speed-tests*, and the *BTS, eNodeB, cell* group.

4.3.1 Users collection

Each document in the `users` collection represents a single account. The document stores the account's credentials (`email`, `password`, `name`), its `role`, and an array `favorite_servers` containing the identifiers of the servers marked as favorites.

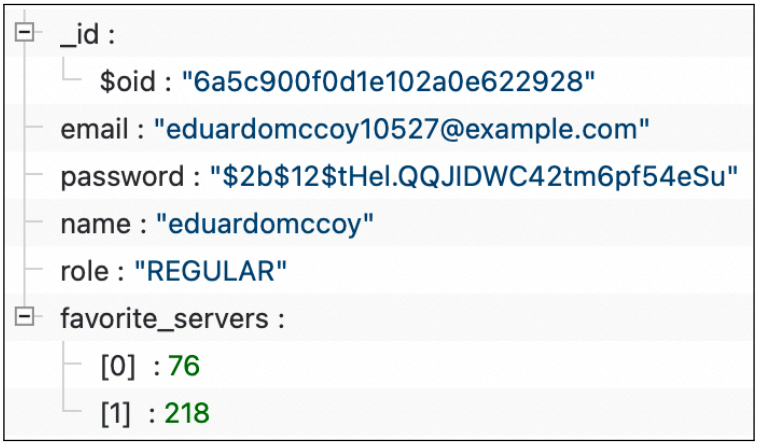


Figure 7: Example document from the `users` collection

Note that `favorite_servers` only stores the numeric identifiers of the servers, rather than embedding their full details (hostname, city, capacity, ...). This is a deliberate choice: the *server* entity lives exclusively in Redis (see Section 4.4), and embedding its fields inside a MongoDB document would mean replicating data across two different DBMSs, requiring a dual-write on every server update just to keep the copies in sync. Moreover, the screen that lists favorite servers (Figure 2) also needs to query the available servers, which always requires a round-trip to Redis regardless – so embedding the favorites' details in MongoDB would not save any query, only duplicate data that is already fetched elsewhere in the same screen.

4.3.2 Speed-tests collection

Each document in the `speedtests` collection represents a single speed-test event, storing the measured `download_kbps`, `upload_kbps`, and `latency_ms`, the `timestamp` of the test, a GeoJSON `location` field (latitude, longitude), and references to the `user` who performed it, the `server` used, and the `enodeb` it was connected to.

The `speed-test - node` relation shown in Figure 6 is implemented by copying the `id` and `mnc` attributes into the `speed-test` document. This is because, in order to execute queries *i* and *ii* efficiently, those information must already be present in the document.

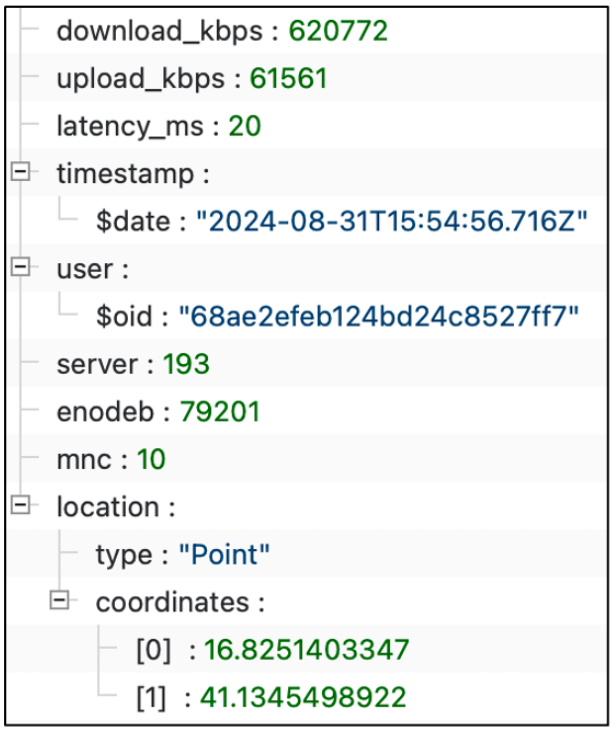


Figure 8: Example document from the `speedtests` collection

We deliberately keep this collection as a set of flat, individually referenced documents, rather than embedding speed-tests inside the `bts` or `users` collections. This is because the three collections are mostly accessed independently. Moreover, given the `speedtests` entity's high velocity (∼1,000 new tests/day), embedding them in other documents would eventually approach MongoDB's 16 MB document limit.

Query (i) Given an eNodeB (identified by `mnc` and node `id`) and a timestamp, compute, considering the speed-tests taken from the node since the specified timestamp, the median download, upload and latency metrics, along with the number of tests, aggregated by three-hour time slots (e.g., 15-18, 18-21 ...).

```
db.speedtests.aggregate([
  // Stage 1: Filter the speed-tests by node and timestamp
  { $match: { enodeb: enodeb, mnc: mnc, timestamp: { $gte: timestamp } } },
  // Stage 2: Group the filtered speed-tests by three-hour time slots and compute
  ↪  the median metrics
  {
    $group: {
      _id: {
        $hour: { $dateTrunc: { date: "$timestamp", unit: "hour", binSize: 3 } }
      },
      number_of_tests: { $sum: 1 },
      median_download_kbps:
        { $median: { input: "$download_kbps", method: "approximate" } },
      median_upload_kbps:
        { $median: { input: "$upload_kbps", method: "approximate" } },
      median_latency_ms:
        { $median: { input: "$latency_ms", method: "approximate" } }
    }
  }
]);
```

The result of this query is visible in the center-right screen of Figure 3.

Query (ii) Given a location L (lat, lon) and a radius R (in meters), compute, for each carrier, the median download, upload, and latency metrics recorded within R meters from L in the last four complete trimesters.

```
db.speedtests.aggregate([
  { // Stage 1: Filter speed-tests within the specified region
    $geoNear: {
      near: { type: "Point", coordinates: [lon, lat] },
      distanceField: "distance_from_L_meters",
      maxDistance: radius,
      spherical: true
    }
  },
  // Stage 2: Keep only speed-tests from the last four complete trimesters
  { $addFields: { start_of_current_quarter: { $dateTrunc: { date: "$$NOW", unit:
  ↪  "quarter" } } } },
  {
    $match: { $expr: { $and: [
      { $lt: [ "$timestamp", "$start_of_current_quarter" ] },
      { $gte: [ "$timestamp", { $dateSubtract: { startDate:
        "$start_of_current_quarter", unit: "year", amount: 1 } } ] }
    ] } }
  },
  { // Stage 3: Add the year and month fields to identify the trimester
    $addFields: {
      year: { $year: "$timestamp" },
      month: { $dateTrunc: { date: "$timestamp", unit: "quarter" } }
    }
  },
  { // Stage 4: Group by carrier and trimester, computing the median values
    $group: {
      _id: { mnc: "$mnc", year: "$year", month: "$month" },
      number_of_tests: { $sum: 1 },
      median_download_kbps:
        { $median: { input: "$download_kbps", method: "approximate" } },
      median_upload_kbps:
        { $median: { input: "$upload_kbps", method: "approximate" } },
      median_latency_ms:
        { $median: { input: "$latency_ms", method: "approximate" } }
    }
  }
]);
```

The result of this query is shown in the left screen of Figure 4.

4.3.3 BTS collection

Each document in the `bts` collection represents a physical tower, embedding the hierarchy described in Section 4.1: a `bts` document has an array of `nodes` (the eNodeBs installed on it), each of which has an array of `cells`. This embedding is justified, unlike the previous cases, because the cardinality is small and stable: a physical tower hosts a limited, slowly-changing number of eNodeBs and cells, so there is no risk of unbounded document growth, and the three levels are always read together.

The `last_verified_by` field materializes the "verify" relation between `bts` and `user`, as shown in Figure 6; since this field is mainly used to show the name of the admin that last modified the data (as seen in the center-left screen of Figure 3) along with the id, also the name of the user is duplicated in the `bts` document.

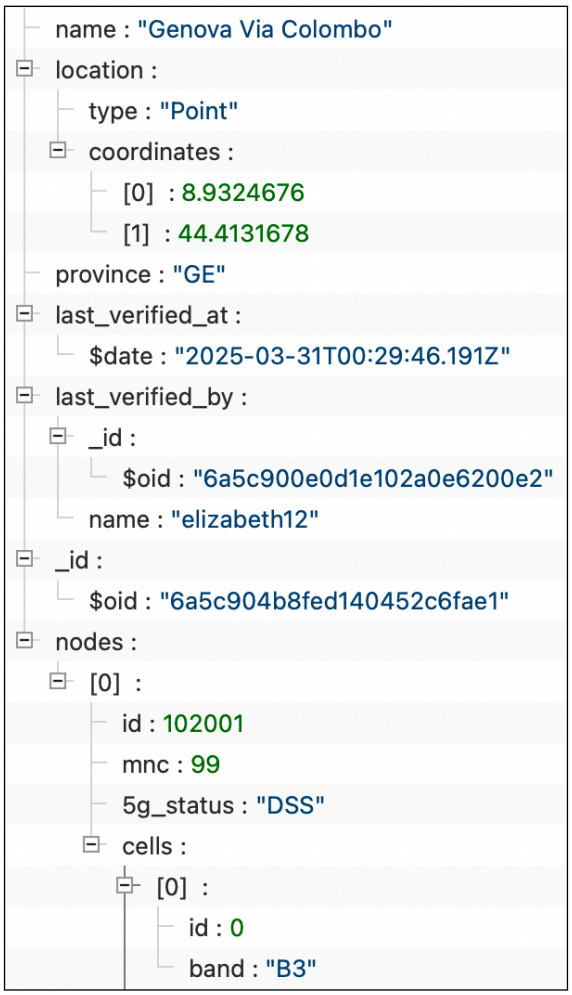


Figure 9: Example document from the `BTS` collection

Query (iii) Given a province (e.g., "PI" for Pisa), compute, for each carrier:

- The total number of BTS, along with the percentage of 5G ones (both DSS and NR)
- The average number of bands per BTS (an indicator of cell quality)

```
db.bts.aggregate([
  // Stage 1: Filter the BTS and unwind the nodes array
  { $match: { province: province } },
  { $unwind: "$nodes" },

  // Stage 2: Compute BTS-specific results for each carrier
  {
    $group: {
      _id: { bts_id: "$_id", mnc: "$nodes.mnc" },
      has_5g_nr:
        { $max: { $cond: [{ $eq: ["$nodes.5g_status", "YES"] }, 1, 0] } },
      has_5g_dss:
        { $max: { $cond: [{ $eq: ["$nodes.5g_status", "DSS"] }, 1, 0] } },
      nested_bands: { $push: "$nodes.cells.band" }
    }
  },
  // Count the unique bands
  { $addFields: { unique_bands_count: { $size: { $reduce: { input:
  ↪  "$nested_bands", initialValue: [], in: { $setUnion: ["$$value", "$$this"] }
  ↪  } } } } },

  // Stage 3: Group by carrier to compute the final statistics
  {
    $group: {
      _id: "$_id.mnc",
      bts_count: { $sum: 1 },
      bts_with_5g_nr_percent: { $avg: { $cond: ["$has_5g_nr", 100, 0] } },
      bts_with_5g_dss_percent: { $avg: { $cond: ["$has_5g_dss", 100, 0] } },
      avg_bands_per_bts: { $avg: "$unique_bands_count" }
    }
  }
]);
```

The results of this query are shown in the left screen of Figure 4.

4.4 Key-Value database

We leverage **Redis** for entities that require: *simple* and *fast* queries over data that is either *volatile* (created and destroyed within a short lifespan) or *static* (rarely updated). With respect to Section 4.1, these entities are the *server* list and the *tickets*.

4.4.1 Servers

The entire server list is kept exclusively on Redis, since it is always queried (every speed-test needs to pick a server) but hardly ever changes, and does not require complex analytics.

We adopt the following key structure:

- `servers:active` – a `SORTED SET` populated via `GEOADD`, indexing all currently active servers by geographic position
- `servers:id_counter` – a `STRING` counter, incremented *before* inserting a new server, used to generate unique server identifiers
- `server:<id>` – a `HASH` storing all the fields of a server (`host`, `lon`, `lat`, `provider_name`, `city`, `provider`, `capacity_mbps`), since these fields are mostly accessed together

Query (iv) Find the N closest active servers to the user's position

```
// Step 1: Fetch the ids of nearby servers
GEORADIUS servers:active <lon> <lat> 500 km ASC COUNT 10 WITHDIST

// Step 2: Fetch the details of the returned ids (pipelined)
HGETALL server:<id>
...
HGETALL server:<id>
```

If a server becomes temporarily unavailable (e.g., for scheduled maintenance), it is sufficient to remove its identifier from the `servers:active` set: its `server:<id>` hash is left untouched, preserving consistency with any `speedtest.server` reference or `user.favorite_servers` entry pointing to it (see Section 4.5).

4.4.2 Tickets

Users open *tickets* to report missing or incorrect information about the BTS infrastructure; each ticket is composed of one or more *requests* (proposed changes to the database). Tickets and requests are kept exclusively in Redis because they are *volatile* – once accepted or discarded by an administrator, they are removed from the database – and, again, do not require complex analytics.

Query (v) Fetch the oldest pending ticket, with all its details and requests.

Naive approach At first, one might come up with a similar design:

- `ticket:<id>:timestamp` – `STRING`
- `ticket:<id>:user` – `STRING`
- `ticket:<id>:<request_id>` – `HASH` (containing the `action` and action-specific values)

This design has **significant downsides**. First, implementing the query would require **additional bookkeeping keys** (i.e., `ticket:oldest_id` to track the oldest pending ticket, and `ticket:<id>:request_count` to know how many requests belong to a ticket); this would also require a minimum of **three round-trips** to execute the query: (1) fetch the oldest ticket id, (2) fetch its details and request count, and (3) fetch all its requests (pipelined). Second, **concurrency is not handled** by default: two administrators could simultaneously fetch and start working on the same ticket.

Redis Streams We instead use a single key `tickets` of type `STREAM`. Redis streams behave like append-only logs and natively support producer/consumer patterns at the database level, whichis exactly what we need: users produce tickets, administrators consume them.

```
// Create the consumer group 'admins' and the stream 'tickets'
XGROUP CREATE tickets admins $ MKSTREAM

// Insert a ticket (* means: auto-generate the new ticket's id)
XADD tickets * \
    user '{ "_id": "68a785e9b5bcf321922694e4", "name": "antirez" }'  \
    requests '[{ "action": "MOVE_BTS", "bts": "68a78cd88742bc1b450e62bc", "lat":
    ↪  45.467, "lon": 9.19 }]'

// === Query implementation ===
// Step 1: Claim the oldest ticket not yet assigned to any admin
XREADGROUP GROUP admins <admin_name> COUNT 1 STREAMS tickets >
// Step 2: Work...
// Step 3: Acknowledge (and remove) the processed ticket
XACKDEL tickets admins <id>
```

The individual *requests* are embedded as a JSON array directly inside the ticket entry, rather than stored as separate keys. This can be done because: a) they are always fetched together with the ticket, and b) they are never modified after creation.

Compared to the naive approach, this solution offers:

- A single round-trip (1 RTT) query implementation
- Thread-safety by default, via consumer groups
- Automatic ticket id management, via the stream's auto-generated entry ids

4.5 Handling inter-database consistency

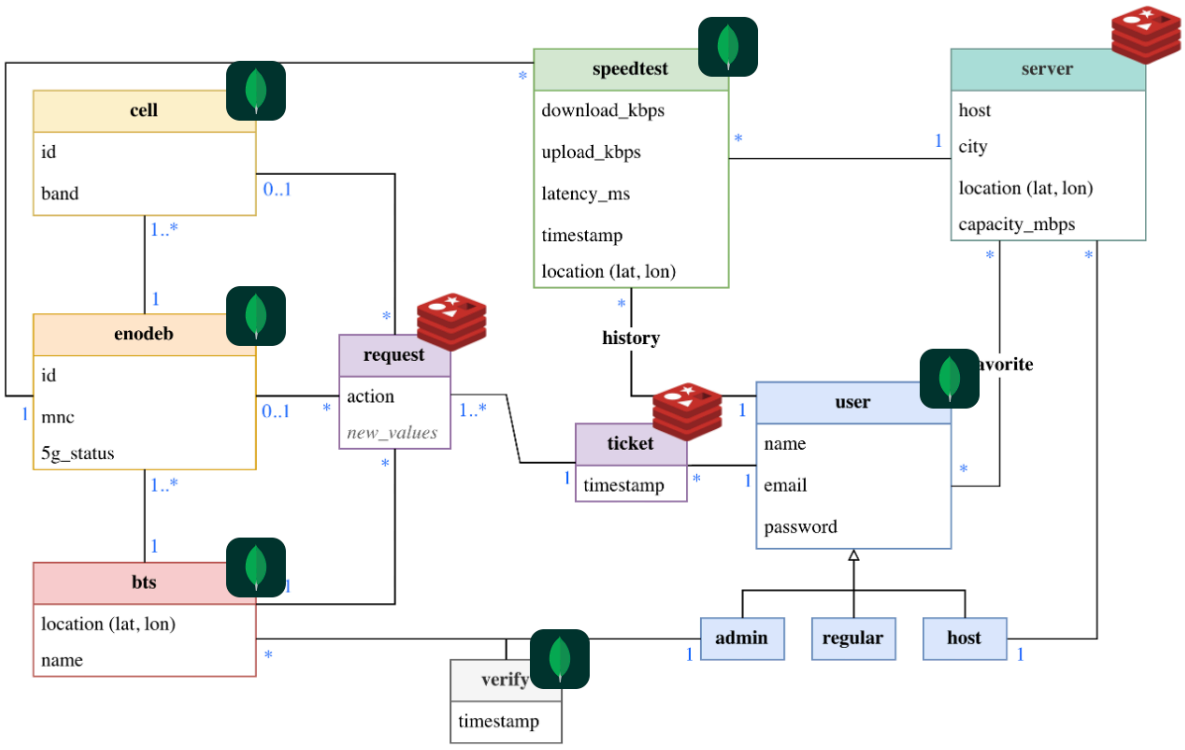


Figure 10: Entity-to-database diagram

By design, no entity is stored redundantly on both MongoDB and Redis: every entity lives in exactly one of the two databases. Therefore, the main concern is the handling of cross-database references.

Tickets

The list of things that could go wrong is relatively small:

- Referenced `BTS/eNodeB/cell` no longer exists or already has the proposed changes: the api must check this before returning the ticket to the admin, in case the check fails, the ticket should be discarded and the next should be considered (until either a valid ticket is found or no tickets are available)
- Referenced `user` no longer exists or its username changed: the ticket can be left unchanged, the admins just won't be able to see the user that published it (or would see an outdated name)

Anyway, tickets are assumed to be resolved relatively quickly, so the probability of referenced entities changing in the meantime should be small.

Servers

Things are only slightly more complex:

- A host `user` is deleted: we can keep its servers, just move them out of the active set. The API endpoint returning a user's favorite list, should always check that the returned servers are active, and, if they are not, permanently remove them from the list
- A host `user` changes name: rare, since host users are usually ISPs; should propagate the change to the `provider_name` field. This should be implemented by the api when handling renaming

As a final note, there is generally no reason to delete a server: it can simply be removed from the active set; users won't see it when looking for servers and the speed-tests can still reference it.

5 Database implementation

This section describes the distributed environment, the replication and partitioning strategies applied to MongoDB and Redis, and the concrete implementation of the application API layer.

5.1 Distributed environment

In accordance with the non-functional requirements, the system is deployed on a distributed 3-node cluster, as shown in Figure 11.

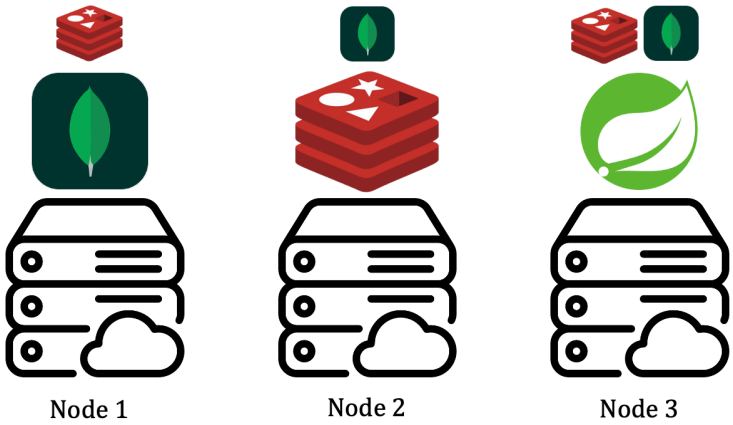


Figure 11: Distributed deployment

To optimize resource utilization and achieve high availability without single points of failure (at least for the storage layers), the software stack is distributed across the three nodes as follows:

- **Node 1**: serves as the MongoDB primary node, while also hosting a Redis replica instance and a Redis sentinel instance to participate in the automated failover orchestration
- **Node 2**: serves as the Redis master node. Simultaneously, it hosts a MongoDB Secondary instance and the second Redis sentinel instance
- **Node 3**: hosts the Spring Boot REST API backend application. To optimize data retrieval performance, it also hosts a MongoDB secondary instance and a Redis replica plus sentinel instance; this allows the local Spring Boot application to leverage the `nearest` read preference, fetching data directly from the local host memory

5.2 Replication

Both database management systems are configured in a distributed, replicated setup to prioritize system availability (A) and partition tolerance (P) over strict immediate consistency (C), aligning with the AP characteristics required by the application (see Section 2.3).

5.2.1 MongoDB

Node priorities To control election behaviors, we assigned specific priorities to each member of the replica set. Particularly, `Node 1` has the highest probability of becoming the primary in an election, instead, `Node 3`, which already hosts the API, has the lowest.

```
rsconf = {
  _id: "CellMapUnipi3",
  members: [
    {_id: 1, host: "127.0.0.1:27017", priority: 3},
    {_id: 2, host: "127.0.0.1:27018", priority: 2},
    {_id: 3, host: "127.0.0.1:27019", priority: 1}
  ]
};
```

Write concern We set the write concern to `{w: "1", j: false}` for the `speedtests` collection, since pure ingestion speed is required. Instead, we set the write concern to `{w: "majority", j: false}` for the `bts` and `users` collections, as we expect a smaller volume of writes and we can therefore afford a slower but safer write policy.

Read concern and preference We use the default value of `local` for the concern and `nearest` for the preference: this way, the node hosting the API can respond faster by reading the data stored in its own, local, MongoDB replica.

5.2.2 Redis

Since Redis only stores very few records ($< 1\,\text{MB}$), most of which are always queried and rarely updated (i.e., the server list), then:

a) The default key eviction policy (`noeviction`) is fine - RAM will never be a problem

b) We can get away with snapshotting the whole database every 10 minutes (`save 600 1`) - this maximizes the performances (no frequent AOF writes) at the (totally acceptable) risk of losing a few new tickets in case of disaster

c) We can provide high availability (as listed in the non-functional requirements) with Redis sentinel alone - there is no need for sharding/horizontal scaling (features offered by Redis cluster)

5.3 Indexes

To improve the performance of our main query patters, specific indexes have been defined.

`speedtests.enodeb` In Query *i*, which performs:

```
$match: { enodeb: enodeb, mnc: mnc, timestamp: { $gte: timestamp } }
```

we created an index on the `enobed` field, which reduced query execution time from 283 ms to just 4 ms.

Why not creating a compound index on all matched fields? It's a valid alternative, however, since:

a) The `speedtests` collection grows rapidly (see Table 1)

b) Just by indexing `enodeb` the performance is already great

maintaining a more complex index (which translates to slower write operations) is not justified with the current volumes. It might become justified later on, if the `speedtests` collections grows even more than expected.

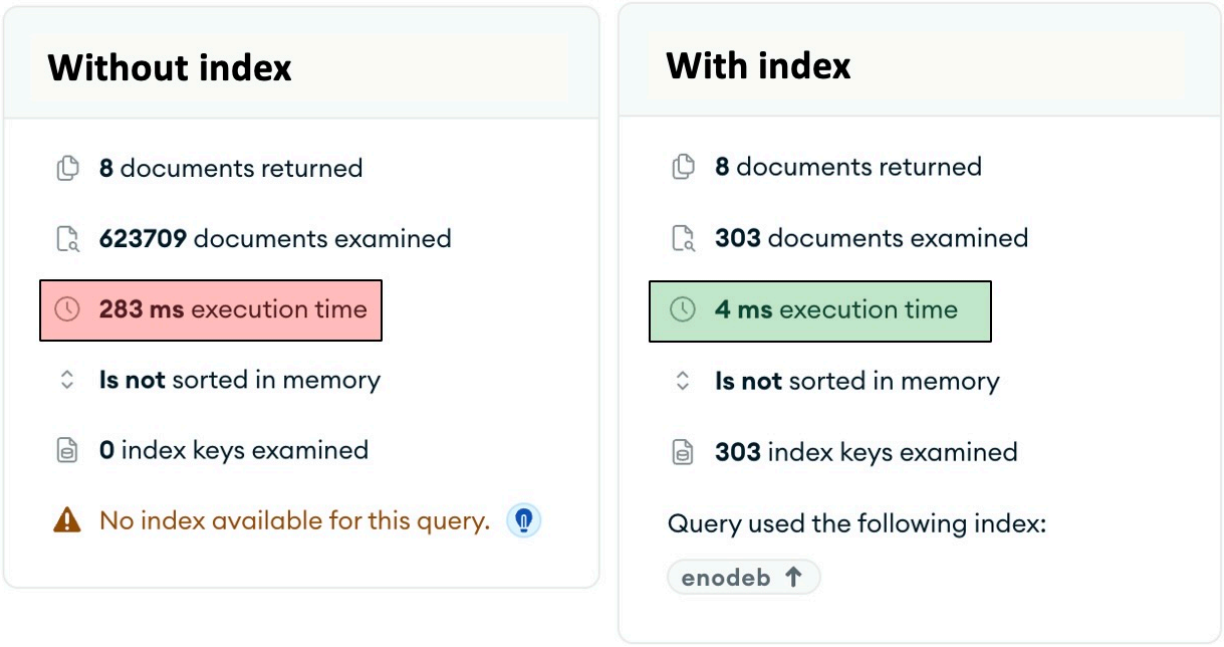


Figure 12: `speedtests.enodeb` index performance

`bts.location`, `speedtests.location` This one is simple, `2dsphere` indexes are mandatory for GeoJSON spatial fields. Therefore, Query *ii*, which performs `$geoNear`, is already optimized by default.

`bts.province` In Query *iii*, which performs:

```
$match: { province: province }
```

we created an index on the `province` field, which reduced query execution time from 86 ms to 11 ms. The are no other indexes that can help speed up this query.

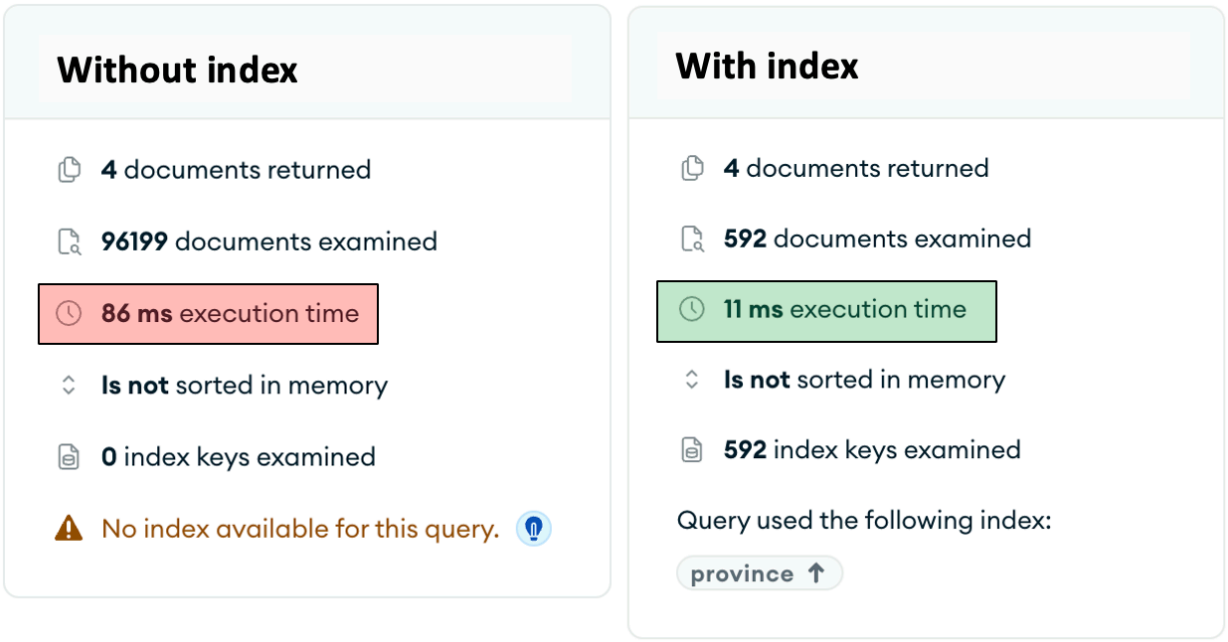


Figure 13: `bts.province` index performance

Secondary indexes Finally, we also implemented indexes for secondary queries, such as:

- `users.email`: speeds up the authentication process (the database fetches the password hash by matching the users on the provided login email)
- `bts.nodes.id`: helps in checking whether a cell already exists in the database (used in Figure 3)
- `speedtests.user`, `speedtests.timestamp`: composite index that speeds up the query to retrieve the speedtest history of a particular user (used in Figure 1)

5.4 Sharding

As the volume of user-submitted data grows over time, horizontal partitioning across multiple servers (i.e., data sharding in MongoDB terms) could be necessary. The sharding policy varies by collection volume and volatility (see Table 1):

- **`users`**: the collection is small (∼4.7 MB) as is not expected to grow rapidly. If, somehow, the application went viral and users spiked exponentially, a solution would be to simply shard by `_id`
- **`bts`**: the collection is fairly small (∼56.6 MB) as is not expected to grow rapidly (a new BTS means that a mobile network carrier has to physically invest real money). If, somehow, BTS grew exponentially, a solution would be to shard by `province` (Query *iii* would still run efficiently, as all BTSs of a single province would be stored in a single server)
- **`speedtests`**: the collection is, at the moment, still small enough to fit on a single server (∼163.2 MB). However, it will undoubtedly grow over time. A possible sharding strategy could be based on the location field - something known as **geohashing**. This would result in geographically close speed-tests being stored in the same servers, maintaining the efficiency of queries *i* and *ii*)

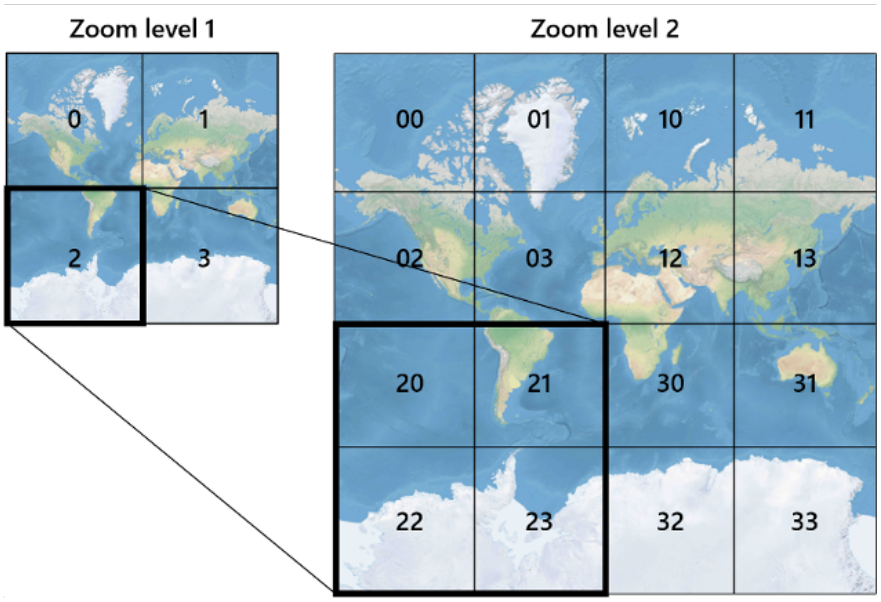


Figure 14: Geohash-based Sharding example

6 API

6.1 Introduction

The backend application is developed in **Java 17** using the **Spring Boot** ecosystem. Data stores are abstracted via *Spring Data MongoDB* and *Spring Data Redis* extensions.

Security and access control are handled by **Spring Security**, enforcing stateless HTTP basic authentication (i.e., users have to send `email` and `password` with each request) and role-based access control (based on the `user.role` field). Public endpoints (such as network searches and account registration) are open to all clients.

The source code is structured in the following way:

- **`controllers`:** Contains the REST Controller classes (`UserController`, `BTSController`, `SpeedTestController`, `ServerController`, `TicketController`, `AnalyticsController`). They manage incoming HTTP requests, validate input parameters, and route responses
- **`database`:** Encapsulates data structures and business logic services partitioned by domain entity (`user`, `bts`, `speedtest`, `server`, `ticket`):
  - **Entities:** Domain objects annotated for database mapping (e.g., `@Document` for MongoDB, `@RedisHash` for Redis)
  - **Repositories:** Spring Data interfaces (`MongoRepository`, `CrudRepository`) providing automated CRUD operations
  - **Services:** Business logic components (e.g., implement the various queries)
- **`dtos`:** Data Transfer Objects decorated with bean validation annotations (e.g., `@Valid`, `@NotNull`, `@NotBlank`) to decouple internal database entities from external API request/response payloads
- **`security`:** Security configuration enforcing stateless authentication
- **`GlobalExceptionHandler`:** Centralized exception handling component decorated with `@ControllerAdvice` that intercepts application errors and returns consistent HTTP error responses
  - **`400 Bad Request`:** Malformed JSON payload or invalid/missing parameters
  - **`401 Unauthorized`:** Invalid HTTP Basic credentials (e.g., wrong email/password)
  - **`403 Forbidden`:** Not enough privileges (e.g., `REGULAR` user testing `/admin` endpoints)
  - **`404 Not Found`:** Returned whenever explicit lookup identifiers (such as a speed-test target server, a BTS, or a ticket id) point to elements that do not exist
  - **`409 Conflict`:** Returned during resource registration conflicts, such as attempting to register an already existing email address

6.2 Endpoints

6.2.1 Users and authentication

- `POST /api/users`
  - **Description:** Creates a new account in the `users` collection with the `REGULAR` role
  - **Payload (JSON):** `email`, `password`, `name`
  - **Outputs:** `HTTP 201` (created) status code
- `POST /api/users/auth/login`
  - **Description:** Authenticates a user against the `users` collection
  - **Outputs:** The authenticated user's profile object containing `_id`, `email`, `name`, `role`, and `favorite_servers`
- `GET /api/users/me`
  - **Description:** Retrieves the profile of the currently authenticated user
  - **Outputs:** `email`, `name`, `role`, and the `favorite_servers`
- `PUT /api/users/me`
  - **Description:** Updates the profile of the currently authenticated user. If the user is a *host*, the change is automatically propagated to `provider_name` on all of its associated server entries in Redis, as discussed in Section 4.5
  - **Payload (JSON):** Any subset of the mutable user fields (for now, just the `name`)
  - **Outputs:** The updated user profile
- `DELETE /api/users/me`
  - **Description:** Deletes the currently authenticated user's account. Any of the user's open tickets are removed in cascade from the Redis `tickets` stream; if the user is a *host*, all of its hosted servers are deactivated (Section 4.5)
  - **Outputs:** `HTTP 204` (no content) status code
- `GET /api/users/me/speedtests`
  - **Description:** Retrieves the authenticated user's speed-test history
  - **Inputs:** `page`, `page_size` (optional, useful for pagination)
  - **Outputs:** A list of the user's speed-tests, sorted by `timestamp` descending (last test comes first)
- `GET /api/users/me/favoriteServers`
- `POST /api/users/me/favoriteServers/{server}`

- `DELETE /api/users/me/favoriteServers/{server}`
  - **Description:** Retrieves, adds, or removes a server ID to/from the authenticated user's `favorite_servers` array. The `GET` endpoint verifies server activity and returns full active server details
  - **Outputs:** A list of active favorite servers for `GET`, or `HTTP 201/204` status code for `POST`/`DELETE`

6.2.2 BTS, eNodeB and cell

Note: direct write access is deliberately *not* exposed due to the ticketing mechanism.

- `GET /api/bts/search`
  - **Description:** Returns all BTS located within a specific geographic region
  - **Inputs:** `lat` (double), `lon` (double), `radius` (int, in meters)
  - **Outputs:** A list of BTS objects
- `GET /api/bts/{id}`
  - **Description:** Retrieves the full details of a single BTS by its document id (used in Figure 3)
  - **Outputs:** The requested BTS object
- `GET /api/cells/check`
  - **Description:** Checks whether the connected cell exists in the database (used in Figure 3 in order to show the "Create ticket" popup)
  - **Inputs:** `mnc` (int), `enodeb` (int), `cell_id` (int)
  - **Outputs:** Boolean status indicating whether the cell exists in the database
- `GET /api/analytics/carrier-comparison`
  - **Description:** Implements query *iii* (check Section 4.3.3)
  - **Inputs:** `province` (string - e.g., `"PI"` for Pisa)
  - **Outputs:** An aggregated JSON payload mapping statistics (`totalBts`, `fiveGNrPercent`, `fiveGDssPercent`, and `avgBands`) grouped by carrier (e.g., TIM, Vodafone, WindTre, Iliad)

6.2.3 Speed-tests

- `POST /api/speedtests`
  - **Description:** Submits a completed speed-test execution

  - **Payload (JSON):** A complete speed-test object (check Section 4.3.2)
  - **Outputs:** The newly generated database `_id` and submission `timestamp`

- `GET /api/speedtests/{id}`
  - **Description:** Retrieves the full details of a single speed-test (used for the "Details of the selected speed-test" screen of Figure 1)
  - **Outputs:** The complete speed-test document

- `GET /api/analytics/hourly-performance`
  - **Description:** Implements query *i* (check Section 4.3.2)
  - **Inputs:** `enodeb` (int), `mnc` (int), `timestamp` (ISO string)
  - **Outputs:** An array of 8 objects representing the three-hour time-slots (e.g., `00-03`, `12-15`) containing the calculated median download/upload rates, median latency, and total tests

- `GET /api/analytics/regional-performance`
  - **Description:** Implements query *ii* (check Section 4.3.2)
  - **Inputs:** `lat` (double), `lon` (double), `radius` (int, in meters)
  - **Outputs:** A dataset grouped by carrier and trimester, including the approximate median download speed, median upload speed, median latency, and the number of processed speed-tests

6.2.4 Servers

- `GET /api/servers/nearest`
  - **Description:** Implements query *iv* (check Section 4.4)
  - **Inputs:** `lat` (float), `lon` (float), `radius` (int, optional, default: 500 – in kilometers), `count` (int, optional, default: 10 – maximum number of servers to return)
  - **Outputs:** An ordered array of up to `count` closest active servers, along with the calculated physical distance in km

- `POST /api/servers`
  - **Description:** Registers a new speed-test server. Performs a socket reachability health-check. Restricted to *host* users. Increments `servers:id_counter`, stores the server record, and adds the active server to `servers:active` via `GEOADD`
  - **Payload (JSON):** `hostname`, `port`, `city`, `lat`, `lon`, `capacity_mbps`
  - **Outputs:** `HTTP 201` (created) status code

- `DELETE /api/servers/{id}`

- **Description:** Deactivates a server (e.g., for scheduled maintenance). Restricted to the owning *host* user. As discussed in Section 4.4, this removes the server's ID from `servers:active`; the server record is preserved to keep any `speedtest.server` references consistent
- **Outputs:** `HTTP 204` (no content) status code

6.2.5 Tickets

- `POST /api/tickets`
  - **Description:** Submits a new ticket. The submitting user's metadata is automatically attached from the authenticated session
  - **Payload (JSON):** A ticket object
  - **Outputs:** Success acknowledgment containing the auto-generated Redis Stream entry ID
- `GET /api/admin/tickets`
  - **Description:** Returns the number of tickets currently pending in the queue, without claiming any of them. Restricted to *admin* users
  - **Outputs:** The number of unclaimed entries in the `tickets` stream
- `POST /api/admin/tickets/claim`
  - **Description:** Claims the oldest pending ticket in the queue that has not yet been assigned to any administrator
  - **Inputs:** `admin_name` (string – In Redis Streams, consuming messages via a consumer group requires specifying both the group name and a consumer name)
  - **Outputs:** The oldest available ticket. Returns empty if the queue is clear
- `POST /api/admin/tickets/resolve`
  - **Description:** Finalizes a claimed ticket. If `status` is `APPROVE`, the system propagates the changes, if `DISCARD`, no change is propagated and the entry is simply discarded
  - **Payload (JSON):** A document with the `ticket_id` field and the `status` (`APPROVE` / `DISCARD`) field
  - **Outputs:** A JSON confirmation showing the resulting status and the number of modified records (if approved)

6.3 Testing

The testing infrastructure leverages **JUnit 5** and Spring Boot Test (`@SpringBootTest`), using **REST Assured** for end-to-end HTTP request and response validation. All integration tests are executed directly against the active distributed infrastructure.

The automated test suite comprises **83 tests** organized into **6 specialized test classes** mirroring the controller architecture:

- **`UserControllerTests`:** Validates account registration, HTTP Basic authentication, user profile retrieval (`GET /users/me`) and updates (`PUT /users/me`), host provider name propagation across managed servers, admin user management, user speed-test history retrieval, and favorite servers management
- **`ServerControllerTests`:** Tests active server discovery via Redis `GEOSEARCH`, server creation with ID auto-increment, dynamic activation and deactivation via the Redis geospatial index (`servers:active`), host ownership authorization, and favorite server operations
- **`TicketControllerTests`:** Validates the end-to-end crowd-sourced ticketing lifecycle: regular user submission to the Redis stream, admin pending queue counting, consumer group ticket claiming (`XREADGROUP`), admin resolution (`APPROVE`/`DISCARD`), payload schema validation, and strict role-based access control (`403 Forbidden` for non-admin users)
- **`SpeedTestControllerTests`:** Validates speed-test submission, strict GeoJSON coordinate validation (latitude/longitude range boundaries and array format checks), and single speed-test retrieval by ID.
- **`BTSControllerTests`:** Verifies geospatial infrastructure searches (`$geoNear`), BTS details retrieval by ID and real-time cell existence verification
- **`AnalyticsControllerTests`:** Validates regional network performance aggregation, carrier-based performance statistics, and input parameter boundary checks (latitude $[-90, 90]$, longitude $[-180, 180]$, and positive radius limits)

7 Conclusions

This document presented the design, implementation, and verification of *CellMap*, a distributed platform designed to collect, process, and map crowd-sourced mobile network measurements across Italy. The primary goal of the project was to provide a transparent view of cellular coverage and performance by leveraging community-driven speed tests and ticket submissions, bypassing the restricted data typical of mobile service providers.

To satisfy both high write-throughput and complex analytic requirements, the system adopted a **hybrid dual-database architecture**. **MongoDB** was selected for persistent storage and complex spatial aggregationssuch as calculating median download, upload, and latency metrics grouped by carrier or geographic regions using `$geoNear` and `$group` pipelines. Conversely, **Redis** was leveraged for volatile stream-based ticket processing via `Redis Streams` and low-latency geospatial lookups for speed-test servers via `GEOADD` and `GEORADIUS`. This clear separation of responsibilities prevented cross-database data duplication while keeping operations performant.

System availability and partition tolerance (the **AP paradigm** in the CAP theorem) were prioritized by deploying the backend across a distributed 3-node environment. This infrastructure combined a **MongoDB Replica Set** with a **Redis** deployment managed by `Sentinel` instances to provide automated failover and node redundancy without single points of failure. Furthermore, query execution times were significantly reduced by creating targeted secondary and geospatial ($2dsphere$) indexes; for instance, indexing the `speedtests.enodeb` and `bts.province` fields decreased key aggregation latency down to single-digit figures.

Finally, the platform's core business logic was exposed via a stateless **Spring Boot** RESTful API enforced by role-based access control (`REGULAR`, `HOST`, and `ADMIN`). The reliability of the API endpoints, database interactions, and administrative workflows was subsequently verified through an automated end-to-end integration test suite built with **JUnit 5** and **REST Assured**. Overall, the resulting system meets all functional and non-functional requirements set for the project in a maintainable and pragmatic structure.