GauchoEats

Shanmukha Sahiti Challa, Praneeth Balasubramanian, Abhishek Kumar, Bhargavi Kurukunda, Ege Demirci

CS291 Scalable Internet Services - Fall 2024, UC Santa Barbara

1. Introduction

In today's interconnected world, the restaurant industry faces increasing competition and evolving customer expectations. The rise of online platforms has transformed how diners discover, evaluate, and engage with dining establishments. As students living in Santa Barbara, we wanted to do something related to our community. Therefore, we present **Gaucho Eats**, an internet service designed to bring consumers and restaurant owners together.

We aimed to create a centralized platform where authenticated users can effortlessly explore a diverse array of restaurants within Santa Barbara. By providing detailed information, user ratings, and reviews, the platform empowers diners to make informed decisions based on collective experiences. Simultaneously, it offers restaurant owners the tools necessary to manage their business profiles, respond to feedback, and enhance their online presence.

Traditional methods of restaurant discovery often rely on limited sources of information, such as word-of-mouth or sporadic online reviews, which may not present a comprehensive view of a dining establishment's quality and offerings. Gaucho Eats addresses this limitation by aggregating user-generated content, thereby offering a more reliable and nuanced perspective. Considering the context of our course, we designed Gaucho Eats to accommodate a growing user base and an expanding feature set. The service's robust database schema and seamless integration provides a foundation from which one could easily make future developments.

This report provides an in-depth analysis of Gaucho Eats, includes service's conceptual framework, architectural design, and implementation strategies. In the first part of the report, we will talk about app features in general, then we will talk about our database structure and models, and in the last part, we will focus on load testing and optimizations.

2. Features

2.1. User Stories

2.1.1. Authenticated Users

1. **View All Restaurants:** *As an authenticated user, I can view all restaurants in Santa Barbara on the dashboard.*

   This feature allows users to access a centralized dashboard displaying a comprehensive list of restaurants within Santa Barbara. The dashboard provides essential information such as restaurant names, locations, and overall ratings, so that user can browse and discover dining options efficiently.

2. **View Restaurant Details:** *As an authenticated user, I can view any restaurant details, browse through the restaurant photos and look at the listed dishes.*

3. **Access Ratings and Comments:** *As an authenticated user, I can view the ratings and user comments for each restaurant.*

   Users can examine detailed ratings and read reviews from other customers and also see the business owner's replies. This information aids users in making informed decisions based on collective feedback.

4. **Add Reviews and Ratings:** *As an authenticated user, I can add my own reviews and ratings to businesses to share my experiences and provide insights to others.*

   This functionality empowers users to contribute to the community by submitting their personal reviews and ratings. By sharing their dining experiences, users help build a reliable repository of feedback that benefits other potential diners.

2.1.2. Unauthenticated Users

1. **View All Restaurants:** *As an unauthenticated user, I can view all restaurants in Santa Barbara on the dashboard.*
2. **View Restaurant Details:** *As an unauthenticated user, I can view any restaurant details, browse through the restaurant photos and look at the listed dishes.*
3. **Access Ratings and Comments:** *As an unauthenticated user, I can view the ratings and user comments for each restaurant.*

2.1.3. Restaurant Owners

1. **Manage Business Profile:** *As a business owner, I can add and manage my business profile to provide accurate information.*

   Restaurant owners have access to tools that allow them to create and update their business profiles. This includes adding details such as restaurant name, address, contact information, and descriptions, so that potential customers have access to up-to-date and accurate information.

2. **Respond to Customer Feedback:** *As a business owner, I can view ratings and comments and reply to customer feedback.*

   Owners can engage with customers by responding to reviews and comments. This interaction sıpports a sense of community and give a chance to the owner for improving customer satisfaction and continuous improvement.

3. **Add Dishes:** *As a business owner, I can add dishes to my restaurant's profile to showcase my offerings.*

   This feature allows owners to highlight their menu items by adding detailed descriptions and pricing for each dish. Displaying the menu helps attract customers by providing a clear understanding of the restaurant's culinary offerings.

3. User Interface Design

The application uses a cohesive user interface design, incorporating the specified color palette and typography to ensure an intuitive and aesthetically pleasing user experience. We used **React JS** for our front end and integrated react app to Rails backend APIs. Figures 3a to 8b illustrate various components of the user interface, including the signup and login screens, home pages for different user roles, the create restaurant page, and the restaurant profile pages for both customers and restaurant owners.

3.1. Why React?

We decided to adopt a React frontend with a Rails API-only backend to achieve a loosely coupled architecture. This design choice allows us to focus entirely on developing and optimizing APIs, making them suitable for rigorous load testing. By decoupling the frontend and backend, we enable easier scalability and maintenance. The modular approach also facilitates parallel development by different team members, as changes to one component doesn't directly impact the other.

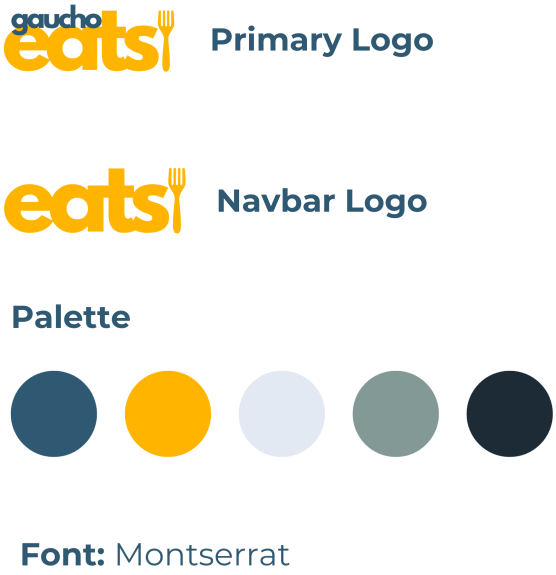


Figure 1. Gaucho Eats user interface specifications.

3.1.1. GitHub Pages

Additionally, separation of frontend and backend components provides the flexibility to host our frontend independently rather than solely alongside our Rails app on AWS Elastic Beanstalk (EBS). We successfully demonstrated this by deploying the React frontend on GitHub Pages and integrating it seamlessly with the Rails APIs as shown in fig 2. This setup shows that the versatility of the backend, as it can serve multiple clients across different hosting platforms.

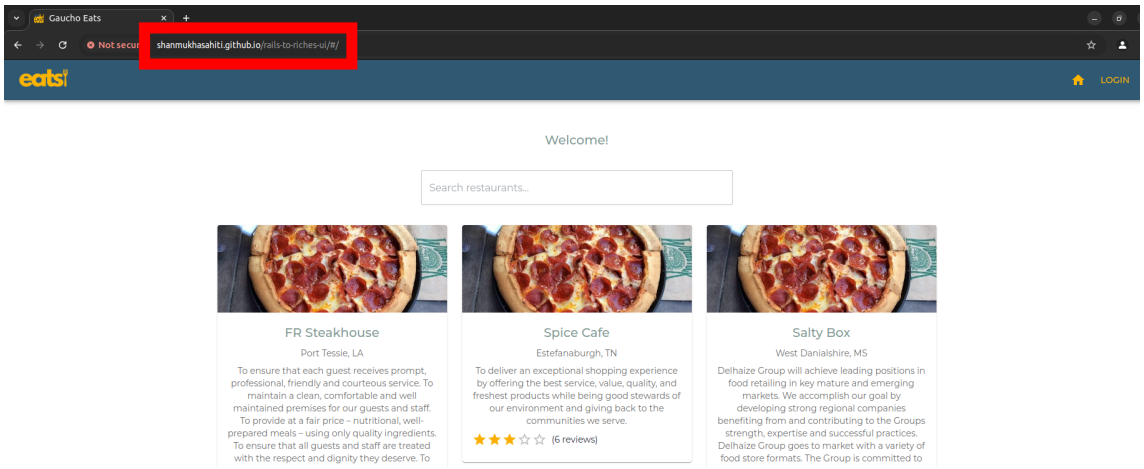
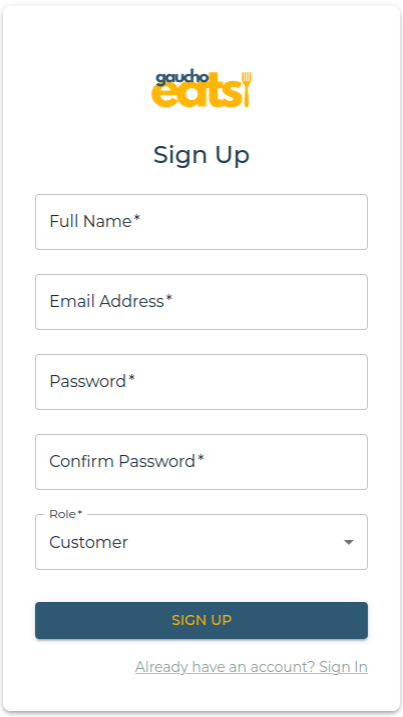


Figure 2. Gaucho Eats on GitHub Pages

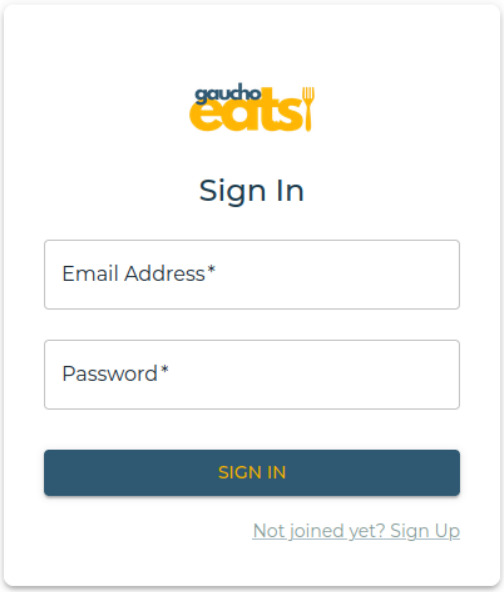
3.2. Application Pages and Functionalities

3.2.1. Authentication Pages

- **Signup Screen:** Users and restaurant owners can register by providing necessary information such as name, email, password, and selecting their user type.
- **Login Screen:** Both users and restaurant owners can authenticate their credentials to access the platform's features.
- **Navigation Bar:** A consistent navigation bar facilitates easy access to different sections of the application, enhancing user experience.



(a) Signup Component



(b) Login Component

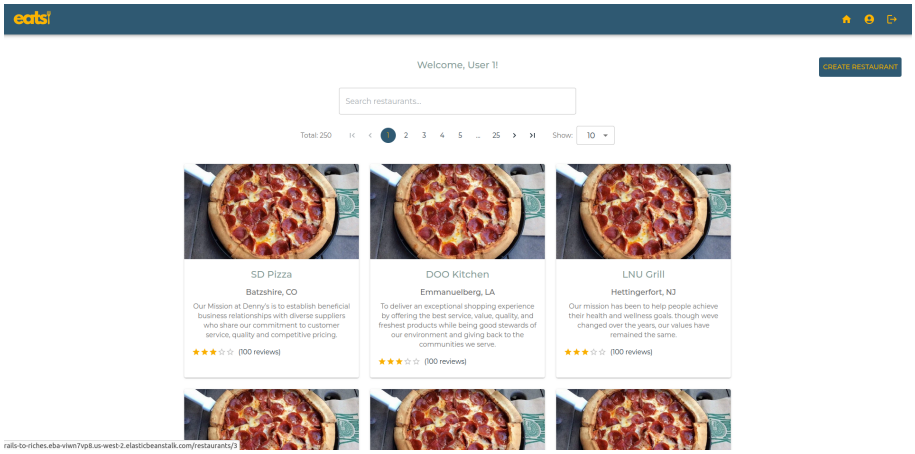
Figure 3. Authentication Interfaces: Signup and Login Screens

3.2.2. Home Pages

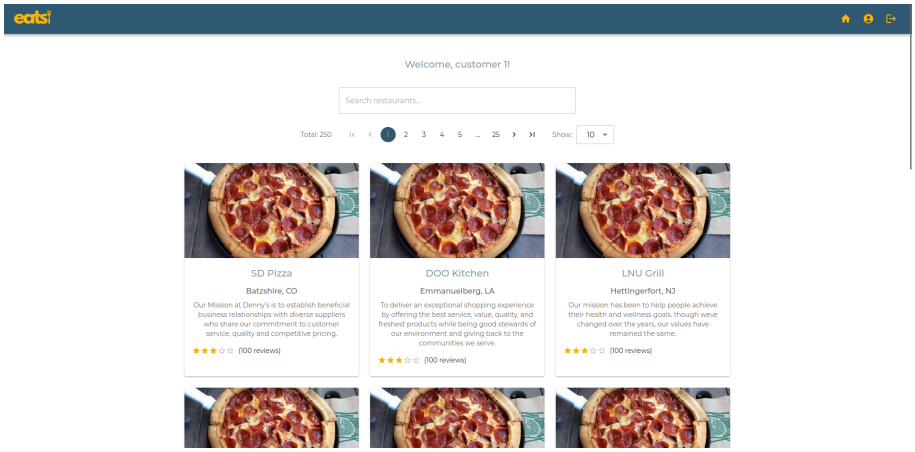
- **User Home Page:** Displays a list of all restaurants along with their ratings and allows users to navigate to individual restaurant profiles.
- **Restaurant Owner Home Page:** In addition to viewing restaurants, owners have access to a "Create Restaurant" page where they can add new establishments to the platform.

3.2.3. Create Restaurant Page

- **Form Submission:** Restaurant owners can input essential details about their restaurant, including name, address, contact information, and description.
- **Photo Upload:** Owners can upload photos of their restaurant, providing visual appeal and attracting potential customers.



(a) Home page for owners.



(b) Home page for customers.

Figure 4. Home pages

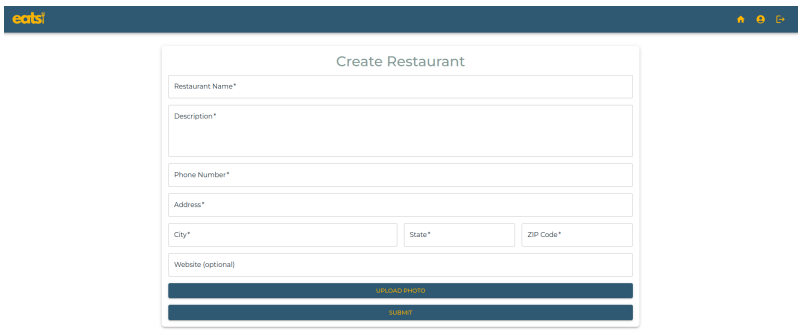


Figure 5. Create restaurants page.

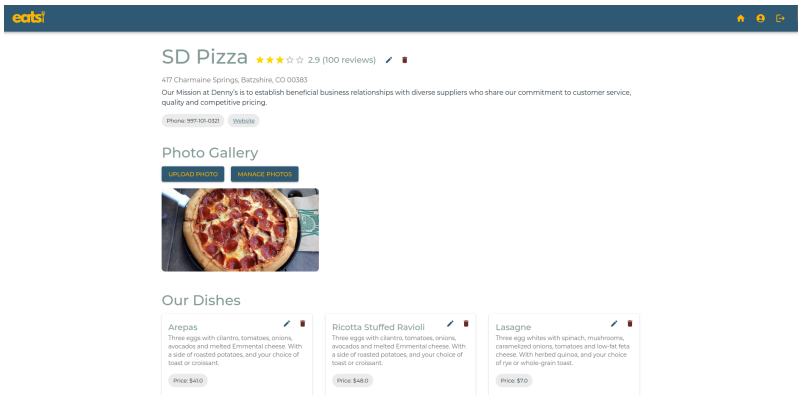
3.2.4. Restaurant Profile Page

The restaurant profile page is tailored to different user roles, offering distinct functionalities for customers and restaurant owners.

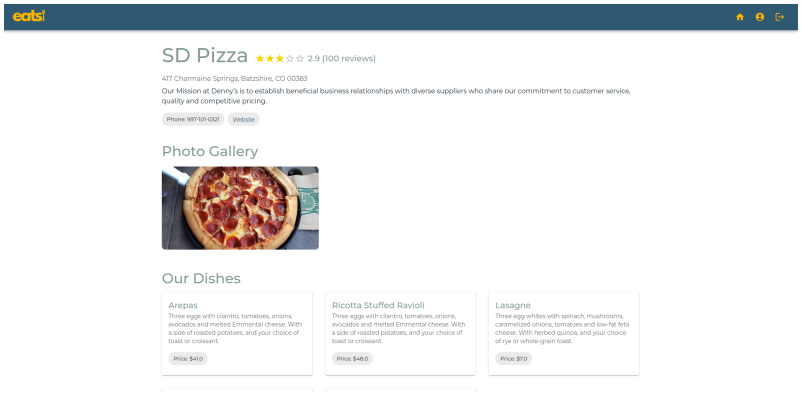
- **Customer Version:**
  - **View Restaurant Information:** Customers can access detailed information about the restaurant, including address, contact details, and descriptions.

  - **Read and Create Reviews:** Users can read existing reviews and submit their own ratings and comments based on their dining experiences, with the option to post anonymously if they wish to do so.

- **Restaurant Owner Version:**
  - **Respond to Reviews:** Owners can reply to customer reviews, addressing feedback and engaging with their clientele.
  - **Add Dishes:** Owners can manage their menu by adding new dishes, including descriptions and pricing, to showcase their offerings.



(a) Restaurant profile for business owners.



(b) Restaurant profile for customers.

Figure 6. Restaurant profiles.

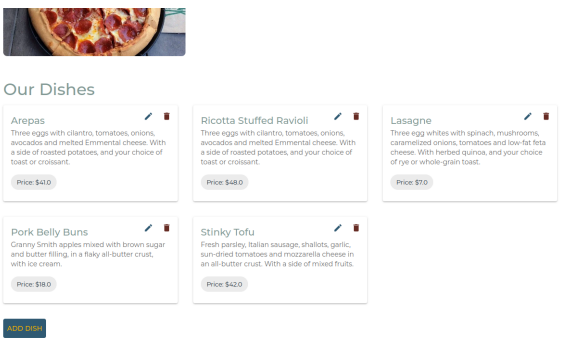
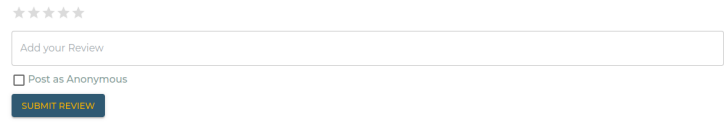
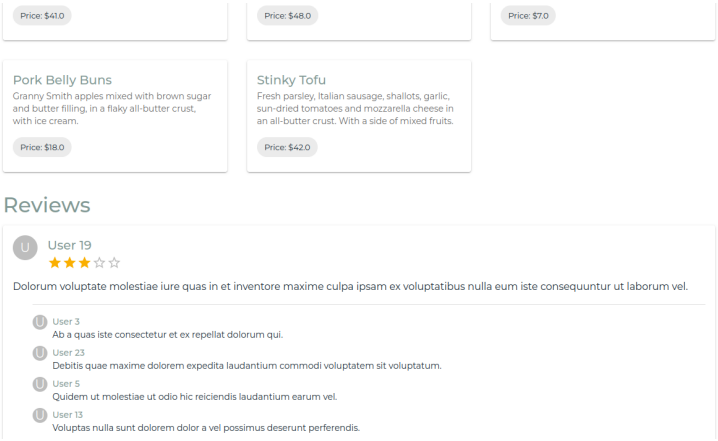


Figure 7. Add dishes for business owners.



(a) Component for writing a review.



(b) Reviews section.

Figure 8. Reviews.

3.3. Additional Functionalities

- **Photo Management:** Restaurant owners can add or delete photos to their restaurant profiles, to improve the visual representation of their establishments.
- **Review and Comment Deletion:** Both users and restaurant owners have the capability to update or delete their reviews and comments, so the content remains relevant and appropriate.
- **Review Restrictions for Owners:** To maintain impartiality, restaurant owners are restricted from reviewing their own establishments.
- **Restaurant Information Updates:** Owners can update or delete their restaurant information as needed, so the platform reflects the most current details.

4. Database Schema

The design of the Gaucho Eats platform relies on a robust and normalized database schema to manage and store data efficiently. The database consists of several interrelated tables that capture information about users, restaurants, dishes, reviews, comments, and photos. This section provides a detailed description of each table, including column definitions, data types, constraints, and relationships.

4.1. Users Table

The `users` table stores information about all registered users, encompassing both customers and restaurant owners.

4.2. Restaurants Table

The `restaurants` table contains data about each restaurant listed on the platform.

4.3. Dishes Table

The `dishes` table records information about the dishes offered by restaurants.

Foreign Key Constraint: `restaurant_id` references `restaurants.id`

Table 1. Schema of the `users` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`name`	String	
`email`	String	Unique, Not Null
`password_digest`	String	Not Null
`role`	String	Default: 'customer'
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

Table 2. Schema of the `restaurants` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`name`	String	Not Null
`address`	String	Not Null
`city`	String	Not Null
`state`	String	Not Null
`zip`	String	Not Null
`description`	Text	
`phone_number`	String	
`website`	String	
`average_rating`	Float	
`total_reviews`	Integer	
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

Table 3. Schema of the `dishes` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`name`	String	Not Null
`description`	Text	
`price`	Decimal	Precision: 10, Scale: 2
`restaurant_id`	Integer	Foreign Key (`restaurants.id`)
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

4.4. Reviews Table

The `reviews` table captures user reviews associated with restaurants.

Table 4. Schema of the `reviews` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`user_id`	Integer	Foreign Key (`users.id`)
`restaurant_id`	Integer	Foreign Key (`restaurants.id`)
`rating`	Integer	Not Null
`content`	Text	
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

Foreign Key Constraints: `user_id` references `users.id`; `restaurant_id` references `restaurants.id`

4.5. Comments Table

The `comments` table stores comments made on reviews by users.

Table 5. Schema of the `comments` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`user_id`	Integer	Foreign Key (`users.id`)
`review_id`	Integer	Foreign Key (`reviews.id`)
`content`	Text	Not Null
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

Foreign Key Constraints: `user_id` references `users.id`; `review_id` references `reviews.id`

4.6. Photos Table

The `photos` table holds images associated with restaurants.

Table 6. Schema of the `photos` Table

Column Name	Data Type	Constraints
`id`	Integer	Primary Key, Auto-increment
`restaurant_id`	Integer	Foreign Key (`restaurants.id`)
`primary`	Boolean	
`created_at`	Datetime	Not Null
`updated_at`	Datetime	Not Null

Foreign Key Constraint: `restaurant_id` references `restaurants.id`

4.7. Database Relationships

The database schema establishes several key relationships between tables to maintain referential integrity:

- **One-to-Many Relationships:**
  - A user can have multiple reviews (`users` to `reviews`).
  - A restaurant can have multiple reviews (`restaurants` to `reviews`).
  - A review can have multiple comments (`reviews` to `comments`).
  - A restaurant can have multiple dishes (`restaurants` to `dishes`).
  - A restaurant can have multiple photos (`restaurants` to `photos`).

These relationships are enforced through foreign key constraints, ensuring data consistency across the database.

4.8. Entity-Relationship Diagram

An Entity-Relationship (ER) diagram illustrating the database schema and the relationships between tables is provided in Figure 9.

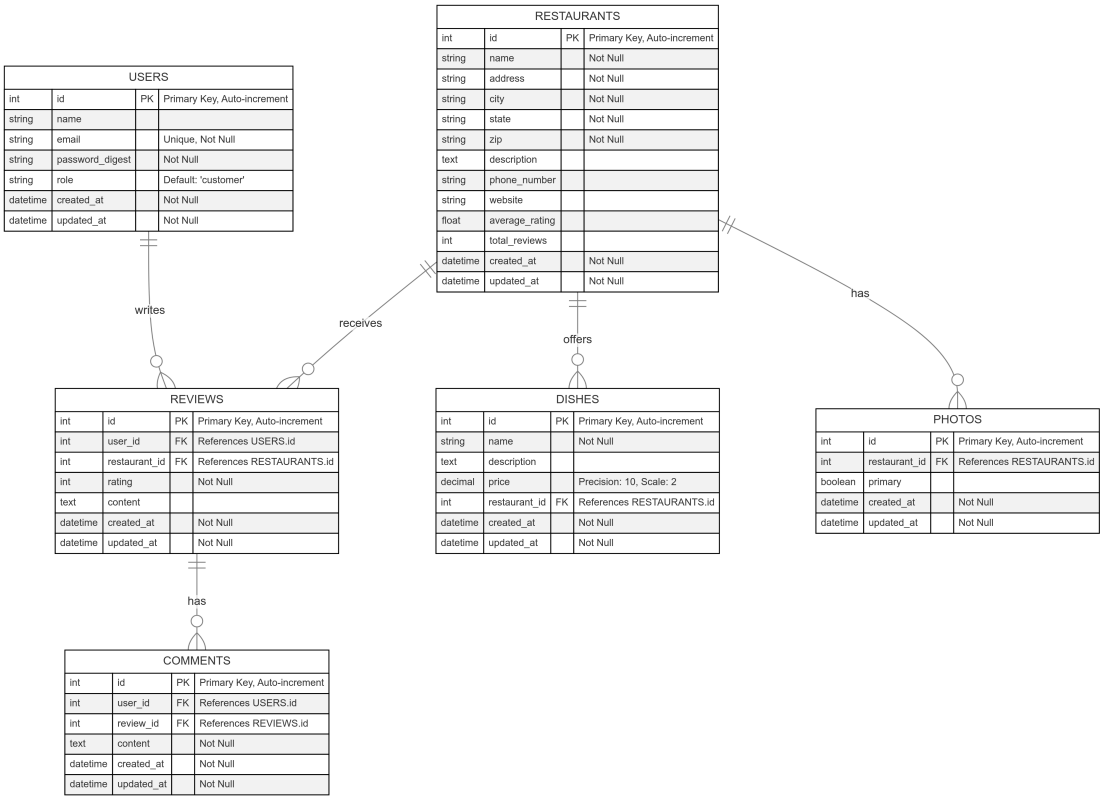


Figure 9. ER diagram for the database.

4.9. Error Handling

All API endpoints adhere to standard HTTP status codes to indicate the success or failure of requests. In the case of errors, the API returns a JSON object containing an error message and relevant details to aid in debugging and resolution.

4.10. Authentication and Authorization

Access to certain endpoints may require authentication and appropriate user roles. The API uses JWT tokens passed in the `Authorization` header to authenticate requests. Authorization checks ensure that users can only perform actions permitted by their roles (e.g., only restaurant owners can create or update restaurants).

5. Load Testing

To measure whether **Gaucho Eats** platform can handle high traffic and maintain optimal performance under various conditions or not, comprehensive load testing was conducted using **Tsung**, a high-performance benchmarking tool. This section outlines the workflows tested, the testing scenarios implemented, and the analysis of the results obtained from these tests.

The load testing scenarios consist of six workflows, each designed to simulate typical user interactions with the **Gaucho Eats** platform. These workflows include actions such as user signup, login, restaurant browsing, and photo management.

5.1. Detailed Workflow Definitions

This section outlines the updated workflow steps for each session defined in the Tsung configuration file. These workflows simulate typical user interactions with the application.

5.1.1. Workflow 1: Signup Test

1. User navigates to the **Signup** page.
2. User fills out the signup form with `name`, `email`, `password`, and `password_confirmation`.
3. User submits the signup form to create a new account.
4. User receives a JWT token upon successful signup.
5. User navigates to the **Dashboard** page to view all available restaurants.
6. For each restaurant, user views the primary photo.

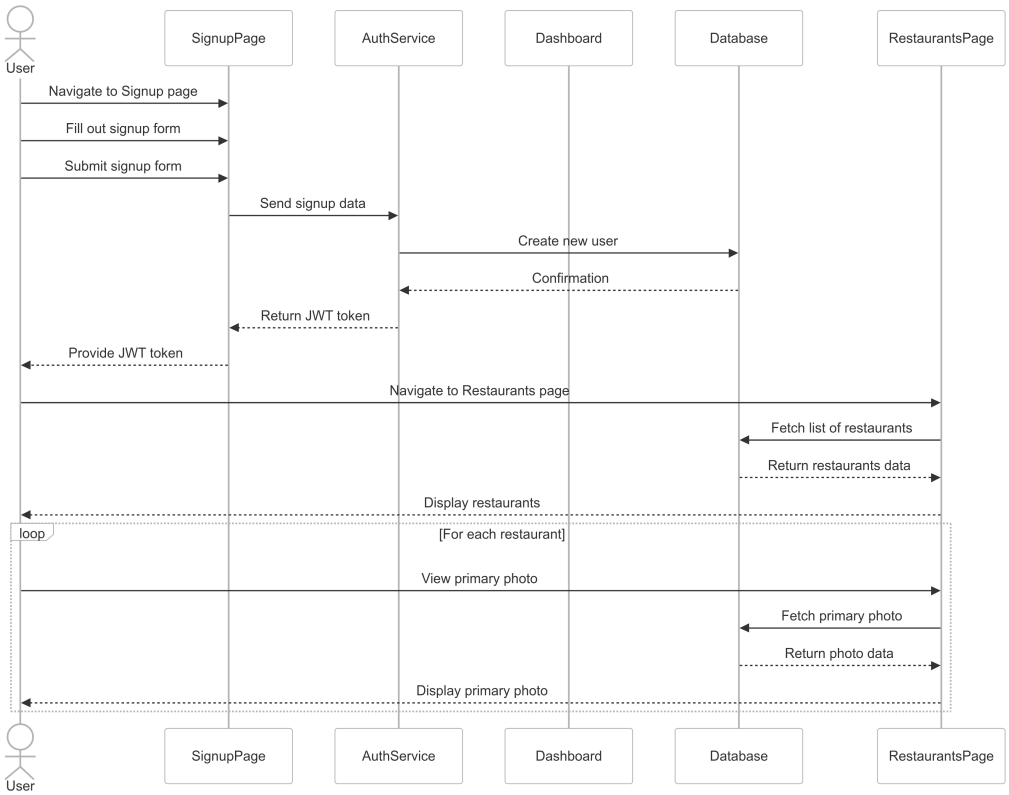


Figure 10. Workflow 1: Signup

5.1.2. Workflow 2: Login Home

1. User navigates to the **Login** page.
2. User enters `email` and `password`.
3. User submits the login form and receives a JWT token.
4. User navigates to the **Dashboard** to view all restaurants.
5. For each restaurant, user views the primary photo.

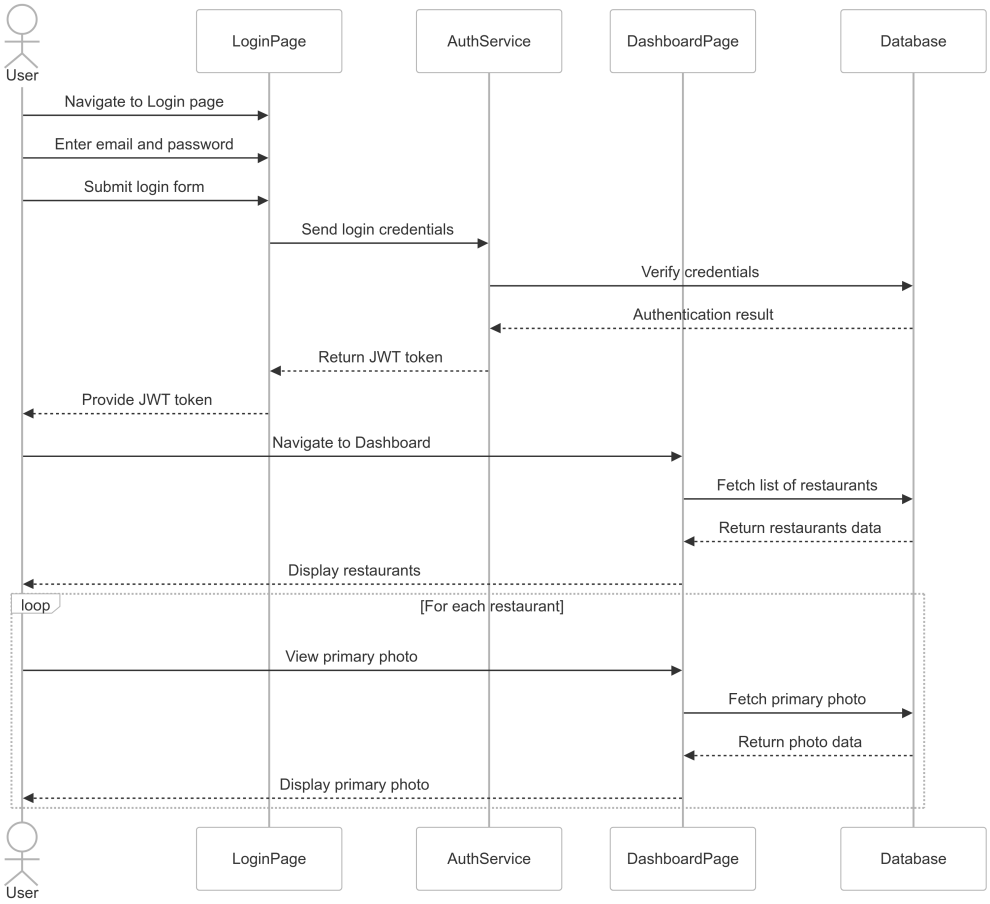


Figure 11. Workflow 2: Login Home

5.1.3. Workflow 3: Manage Restaurant Profile

1. User navigates to the **Login** page.
2. User enters `email` and `password`.
3. User submits the login form and receives a JWT token.
4. User navigates to the **Dashboard** to view all restaurants.
5. For each restaurant, user views the primary photo.
6. User creates a new restaurant by filling out the restaurant creation form.
7. User views the details of the newly created restaurant.
8. User updates the information of the restaurant.
9. User adds dishes to the restaurant's menu.

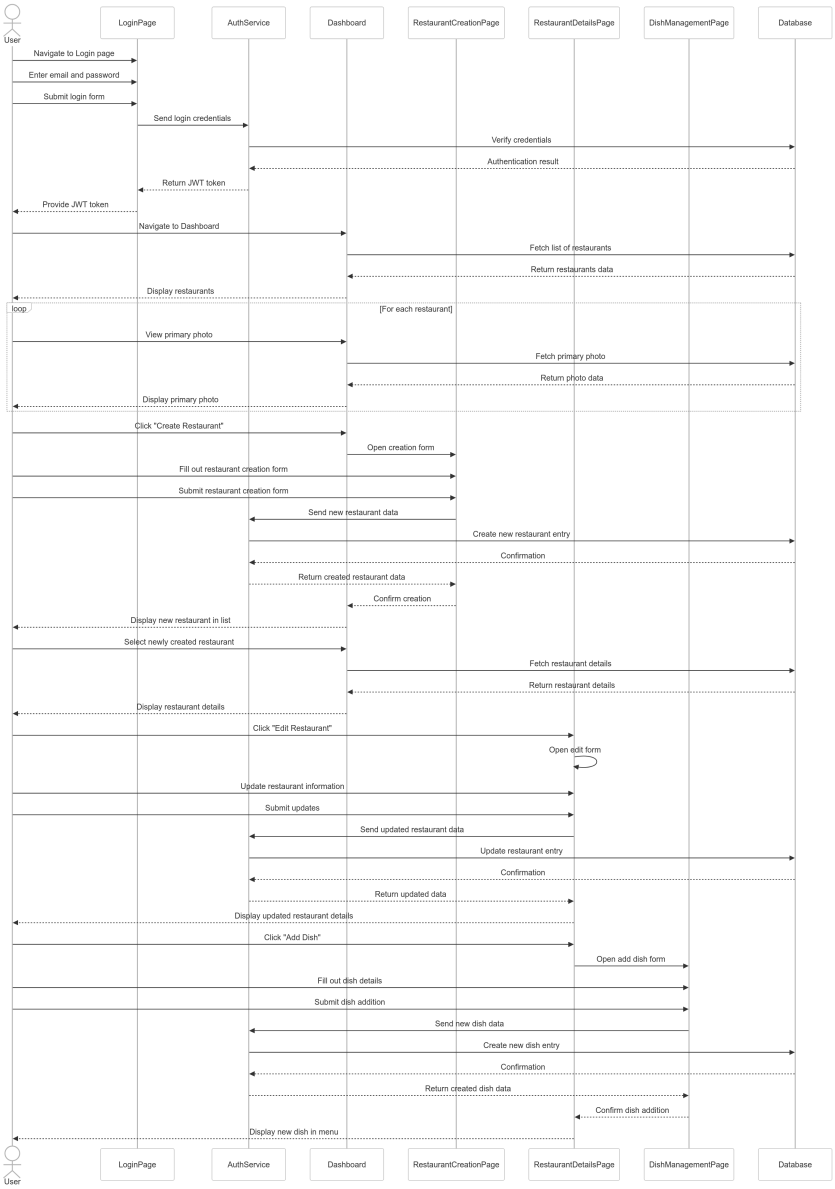


Figure 12. Workflow 3: Manage Restaurant Profile

5.1.4. Workflow 4: Browse Restaurants (Unauthenticated)

1. User navigates to the **Dashboard** page.
2. For each restaurant, user views the primary photo.

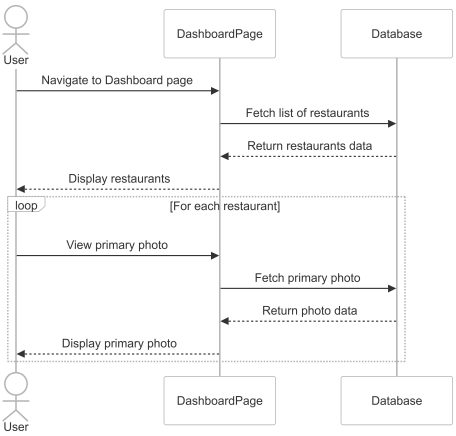


Figure 13. Workflow 4: Browse restaurants unauthenticated

5.1.5. Workflow 5: Add Reviews

1. User navigates to the **Login** page.
2. User enters `email` and `password`.
3. User submits the login form and receives a JWT token.
4. User navigates to the **Restaurant Details** page for restaurant ID 1.
5. User views the dishes offered by restaurant 1.
6. User views the photos for restaurant 1.
7. User views the reviews for restaurant 1.
8. User submits a review for restaurant 1.

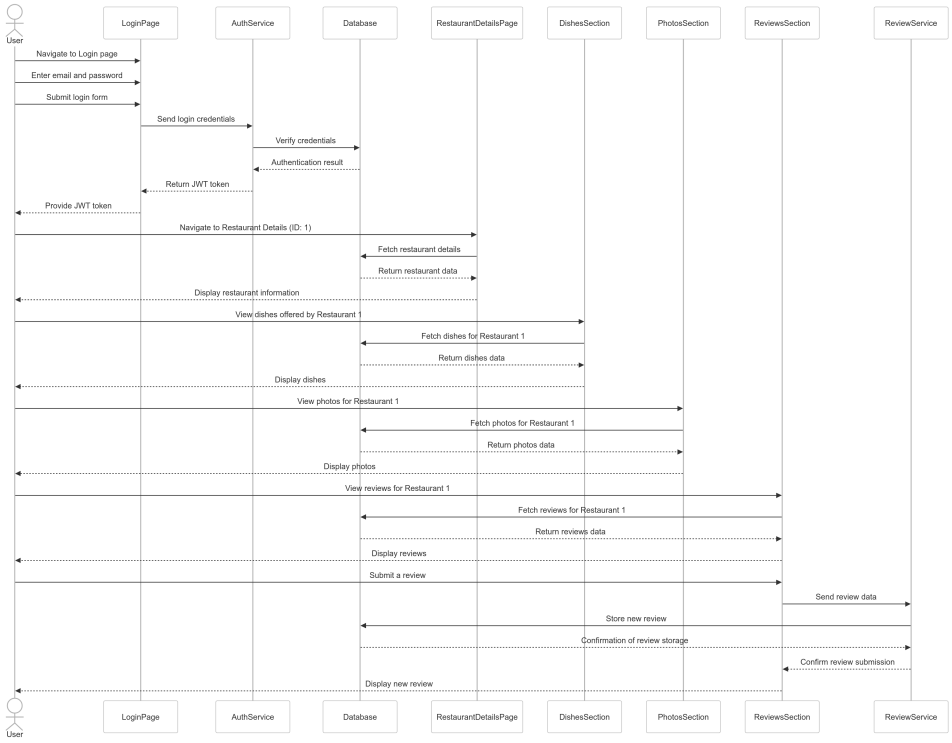


Figure 14. Workflow 5: Add reviews

5.1.6. Workflow 6: Photo Gallery Browsing and Uploading

1. User navigates to the **Login** page.
2. User enters `email` and `password`.
3. User submits the login form and receives a JWT token.
4. User navigates to the **Restaurant Details** page for restaurant ID 1.
5. User views the dishes offered by restaurant 1.
6. User views the photos for restaurant 1.
7. User uploads a new photo to the restaurant's gallery.
8. User views the updated photo gallery.

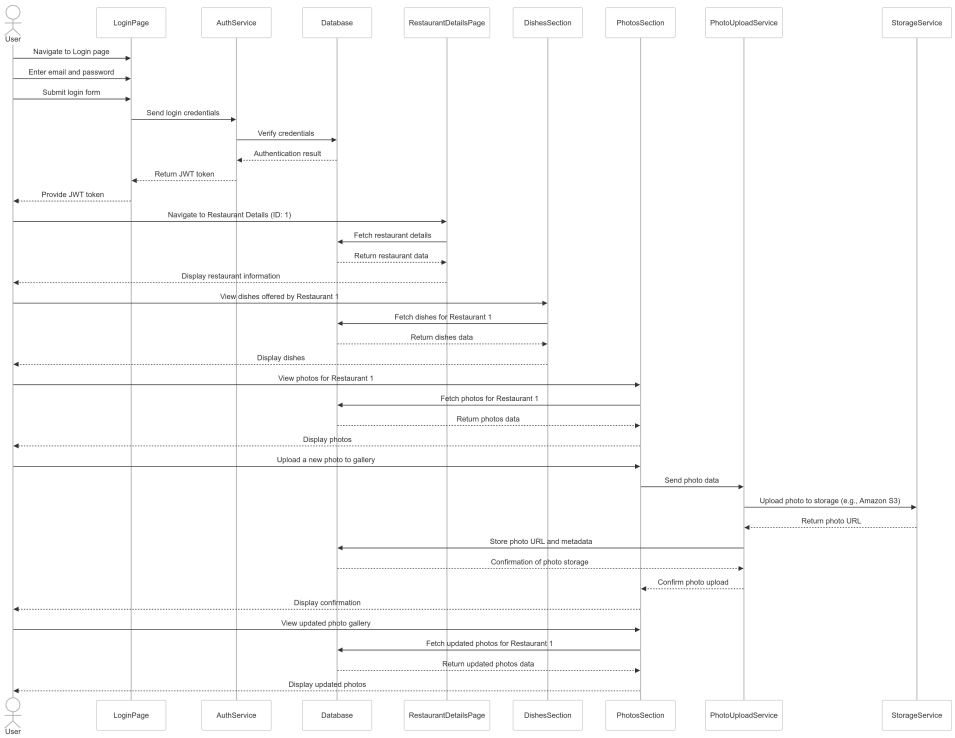


Figure 15. Workflow 6: Managing photos

6. Optimizations

6.1. Pagination

Pagination is a critical optimization for high-traffic systems, particularly when working with large datasets. For the first experiment in terms of optimization, we evaluated the performance impact of implementing pagination for the `/api/restaurants` endpoint. Two configurations were tested: requests with pagination and requests without pagination. The dataset contained seed data for 250 restaurants, and the performance was measured under increasing user loads.

6.1.1. Experimental Setup

- APIs :
  - Without Pagination: /api/restaurants
  - With Pagination: /api/restaurants_paged
- Instance :
  - Application server: c5.xlarge
  - Database: db.m5.xlarge
- Tsung load: Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 16 users/sec
  - Phase 2: 32 users/sec
  - Phase 3: 64 users/sec
  - Phase 4: 128 users/sec
  - Phase 5: 256 users/sec
  - Phase 6: 512 users/sec
  - Phase 7: 1024 users/sec

The arrival rate of users was increased in phases, starting from 16 users per second and doubling in each subsequent phase, up to 1024 users per second. The following scenarios were tested:

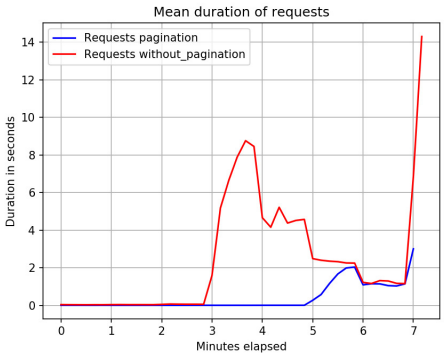
- **With Pagination:** Requests were made to the paginated endpoint where only 10 restaurants were returned per request.
- **Without Pagination:** Requests were made to the non-paginated endpoint, which returned all 250 restaurants in a single request.

6.1.2. Results

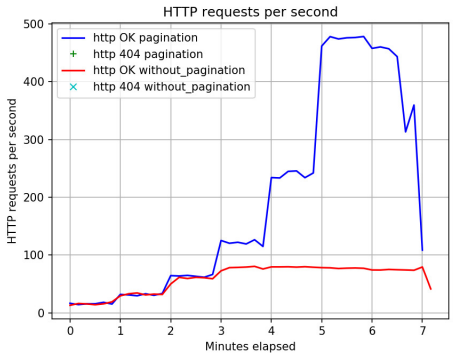
As shown in Figure 16a, the mean duration of requests without pagination increased significantly as the user load increased. By contrast, the mean duration of requests with pagination remained consistently low, even under high traffic. At the highest load phase, the duration of non-paginated requests exceeded 14 seconds, while paginated requests stayed just above 2 seconds.

Figure 16b illustrates the throughput in terms of HTTP requests per second. The system with pagination maintained higher throughput under increasing load, reaching over 500 requests per second at its peak. In contrast, the non-paginated configuration struggled to handle the load, with throughput dropping significantly after phase 5 due to resource exhaustion and higher response times.

We can see the significant performance improvements achieved by implementing pagination. The non-paginated endpoint placed a heavy burden on both the application server and the database, resulting in longer response times and reduced throughput. By limiting



(a) Mean duration of requests with and without pagination.



(b) HTTP requests per second with and without pagination.

Figure 16. Performance comparison of paginated and non-paginated requests.

the amount of data returned per request, pagination effectively distributed the load across multiple smaller queries, ensuring the system remained responsive under high traffic.

Furthermore, the consistent performance of the paginated endpoint indicates that the system can scale to support larger datasets and higher user loads. This improvement is particularly relevant for real-world scenarios where the number of restaurants or users may grow significantly over time.

6.2. Caching

API : /api/restaurants_paged
Caching is a critical optimization strategy to reduce the computational load on servers and databases by precomputing and storing frequently accessed data. Therefore, we wanted to analyze the impact of caching on the performance of the `/api/restaurants_paged` endpoint. The experiment compared two configurations: requests with caching and requests without caching.

6.2.1. Experimental Setup

The endpoint retrieves a paginated list of restaurants, including their average ratings. The average rating was computed differently in the two configurations:

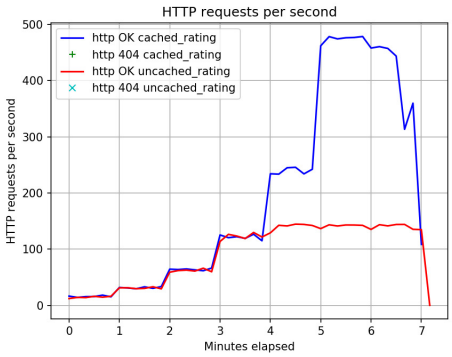
- **Without Caching:** The average rating was calculated dynamically by querying the database for each restaurant at the time of the request. This required an "additional" SQL lookup for every restaurant.
- **With Caching:** The average rating was precomputed and stored. Whenever a new review was posted, the average rating was recalculated and updated. During the request, the precomputed value was retrieved, reducing database query overhead.

Just like in the first experiment, the arrival rate of users was increased in phases, starting from 16 users per second and doubling in each subsequent phase, up to 1024 users per second. The instances are the same as the earlier one.

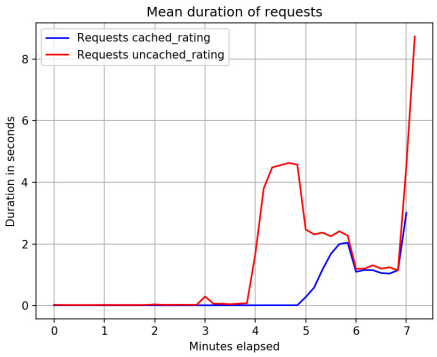
6.2.2. Results

The results of the experiment are summarized in the figures below.

Figure 17a shows the throughput in terms of HTTP requests per second. The system with caching maintained higher throughput under increasing load, peaking at over 500 requests per second. In contrast, the uncached configuration struggled to keep up, with not much



(a) HTTP requests per second with and without caching.



(b) Mean duration of requests with and without caching.

Figure 17. Performance comparison of caching.

increase in throughput even when the load is significantly high after phase 4 due to increased server and database overhead.

As shown in Figure 17b, the mean duration of uncached requests increased significantly as the user load increased. At the highest load phase, the duration of uncached requests exceeded 8 seconds, while cached requests remained consistently below 3 seconds. This demonstrates the significant latency introduced by dynamic computations in the absence of caching.

Caching provided a substantial performance improvement by reducing the computational overhead of dynamically calculating average ratings for each request. By precomputing and storing these values, the system significantly reduced database queries and server processing time, so we are able to handle higher traffic with lower response times.

The cached configuration also demonstrated better scalability, which maintains consistent performance even under high user loads. This shows the importance of caching as a strategy for optimizing database-heavy operations, particularly for endpoints accessed frequently in high-traffic environments.

6.3. Image Storage Optimization Using Amazon S3

Efficient storage and retrieval of images is also an important metric in terms of optimization. Initially, restaurant images in the Gaucho Eats platform were stored directly in the database, which posed significant challenges in terms of performance and scalability. To address these challenges, the image storage mechanism was optimized by migrating to Amazon S3.

In the initial implementation, restaurant images were stored as binary data in the `photos` table of the database. When a user requested an image, the server retrieved the binary data, encoded it in base64 format, and returned it as part of the JSON response. With this approach we can certainly know that the image is available in the database but there are several problems:

- Storing large binary files in the database significantly increased its size and degraded query performance.
- Encoding binary data to base64 format and transferring it in JSON responses added overhead, increasing response times.
- Databases are not optimized for storing and serving large binary data, making this approach unsuitable for handling a growing number of images.

6.3.1. Optimized Approach with Amazon S3

To overcome these limitations, restaurant images were migrated to Amazon S3. Instead of storing images in the database, a link to the image stored on S3 is now returned in the API

response. This approach has several advantages:

- By offloading image storage to S3, the database size and query complexity were reduced, improving overall performance.
- Transmitting image URLs instead of base64-encoded data reduced the size of API responses and accelerated response times.
- S3 is designed for storing and serving large amounts of data, so it is well-suited for handling a growing number of images.

6.3.2. Experimental Setup

- API : /api/restaurants/:id/primary_photo
- Instance :
  - Application server: c5.xlarge
  - Database: db.m5.xlarge
- Tsung load: Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 4 users/sec
  - Phase 2: 8 users/sec
  - Phase 3: 16 users/sec
  - Phase 4: 32 users/sec
  - Phase 5: 64 users/sec
  - Phase 6: 128 users/sec
  - Phase 7: 256 users/sec

The following scenarios were tested:

- **Database Storage:** Images were stored in the database and returned as base64-encoded binary data.
- **Amazon S3 Storage:** Images were stored in S3, and only the S3 link was returned in the API response.

The load was increased in phases, starting from 4 users per second and doubling in each subsequent phase, up to 256 users per second.

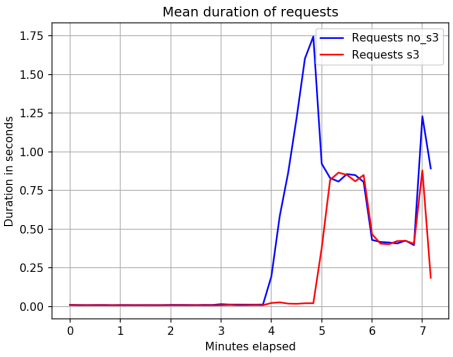
6.3.3. Results

The results of the experiment are summarized in Figure 18b and Figure 18a.

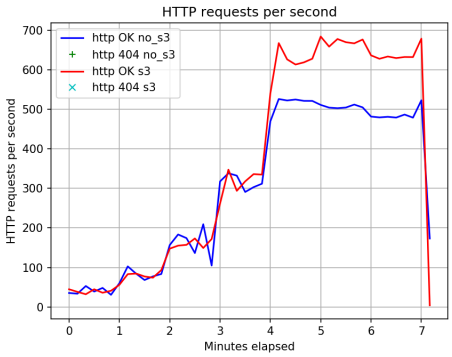
Figure 18b illustrates the throughput in terms of HTTP requests per second. The S3 configuration demonstrated significantly better scalability, which maintains higher throughput as user load increased. At its peak, the S3 configuration handled almost 700 requests per second, whereas the database-based configuration lagged behind, struggled to maintain throughput under higher loads.

As shown in Figure 18a, the mean duration of requests without s3 increased significantly as the user load increased. At the highest load phase, the duration of non s3 requests was almost 1.75 seconds, while s3 requests remained consistently below 1 second. This demonstrates the significant latency introduced when loading images from the database.

The results clearly show the benefits of migrating image storage to Amazon S3 over our initial implementation. By reducing the load on the database and utilizing S3's efficient content delivery network, the system was able to achieve higher throughput and better performance under increasing traffic.



(a) Mean duration of requests with and without S3



(b) HTTP requests per second with and without S3

Figure 18. Performance comparison of using S3 and not using S3.

6.4. N+1 Query Optimization

N+1 query optimization is another experiment we tried. N+1 query problems arise when an application executes one query to fetch a parent object (e.g., reviews) and then executes additional queries to fetch related objects (e.g., comments) for each parent. This results in a significant number of redundant database queries, especially when the parent object has many associated records, as shown in Figure 19. Therefore, we wanted to analyze the impact of N+1 optimization on the performance of fetching paginated reviews and their associated comments for a specific restaurant, as shown in Figure 20.

```
web-1  |   Review Load (0.5ms)  SELECT "reviews".* FROM "reviews" WHERE "reviews"."
restaurant_id" = $1 LIMIT $2 OFFSET $3  [["restaurant_id", 1], ["LIMIT", 5], ["OFFS
ET", 0]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.4ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" = $1  [["review_id", 1]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.7ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" = $1  [["review_id", 2]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.4ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" = $1  [["review_id", 3]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.5ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" = $1  [["review_id", 4]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.3ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" = $1  [["review_id", 5]]
```

Figure 19. Illustration of redundant queries caused by the N+1 problem.

```
web-1  |   Review Load (0.6ms)  SELECT "reviews".* FROM "reviews" WHERE "reviews"."
restaurant_id" = $1 LIMIT $2 OFFSET $3  [["restaurant_id", 1], ["LIMIT", 5], ["OFFS
ET", 0]]
web-1  |   ↳ app/controllers/api/reviews_controller.rb:18:in `paged_index'
web-1  |   Comment Load (0.6ms)  SELECT "comments".* FROM "comments" WHERE "comment
s"."review_id" IN ($1, $2, $3, $4, $5)  [["review_id", 1], ["review_id", 2], ["revi
ew_id", 3], ["review_id", 4], ["review_id", 5]]
```

Figure 20. Optimized query execution using N+1 optimization.

6.4.1. Experimental Setup

– API : /api/restaurants/:id/reviews_paged

- Instance :
  - Application server: c5.xlarge
  - Database: db.m5.xlarge
- Tsung load: Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 1 users/sec
  - Phase 2: 2 users/sec
  - Phase 3: 4 users/sec
  - Phase 4: 8 users/sec
  - Phase 5: 16 users/sec
  - Phase 6: 32 users/sec
  - Phase 7: 64 users/sec

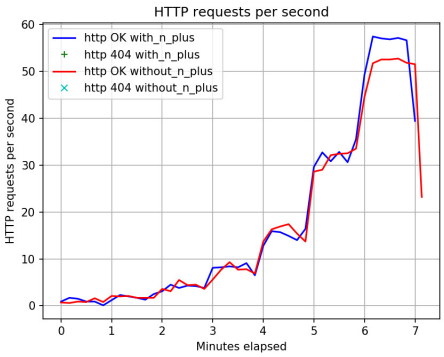
The arrival rate of users was increased in phases, starting from 1 user per second and doubling in each subsequent phase, up to 64 users per second. The endpoint retrieves paginated reviews, including comments, for a specific restaurant. The following configurations were tested:

- **Without Optimization:** For each review fetched, a separate query was executed to fetch its associated comments, resulting in 50 additional queries per review.
- **With Optimization:** The `.includes` method was used to perform eager loading of comments along with the reviews in a single query, significantly reducing the number of database queries.

Seed data consisted of 50 comments per review, with a total of 10 reviews fetched per page.

6.4.2. Results

The results of the experiment are summarized in the figures below.



(a) HTTP requests per second with and without N+1 optimization.

(b) Mean duration of requests with and without N+1 optimization.

Figure 21. Performance comparison of N+1 optimized and non-optimized requests.

Figure 21a shows the throughput in terms of HTTP requests per second. The optimized configuration achieved higher throughput under increasing user load, maintaining steady performance up to 64 users per second. The non-optimized configuration struggled as the load increased, with no significant change in throughput due to the overwhelming number of queries generated by the N+1 problem. We also would like to mention that we tested this scenario with 50 comments per review and that could be the reason why the difference between the optimized and non-optimized versions is not very high. Had we tested the same

scenario with even larger seed data, say 500 comments per review, the difference between the two would have been significantly larger.

As shown in Figure 21b, the mean duration of requests without optimization increased dramatically as the user load grew, exceeding 12 seconds in the highest load phase. In contrast, the optimized configuration maintained much lower response times, staying under 3 seconds even at the peak load.

The results clearly demonstrate the negative impact of the N+1 query problem on system performance. Without optimization, the database was overwhelmed by redundant queries, which leads to higher latency and reduced throughput. The optimized configuration using eager loading (`.includes`) significantly reduced the number of queries, which allows the system to handle higher traffic with lower response times.

We can say that, N+1 query optimization is a good practice for improving the performance and scalability of applications with complex database relationships. In our case, by minimizing the number of redundant queries through techniques such as eager loading, our system could handle higher traffic with improved response times and throughput.

7. Load Testing Realistic Scenaraio

We came up with a realistic scenario comprising of all the workflow sessions we mentioned earlier. We used this scenario configuration to perform load testing when Gaucho Eats is horizontally and vertically scaled. Table 7 summarizes the session probabilities used in the test configuration.

Table 7. Session Probabilities

Session Name	Probability (%)	Description
`signup_test`	10	Simulates a new user signing up and browsing restaurants.
`login_home`	25	Simulates an existing user logging in and accessing the dashboard.
`manage_restaurant_profile`	10	Simulates a restaurant owner managing their restaurant profile and adding dishes.
`browse_restaurants`	30	Simulates unauthenticated users browsing restaurants and viewing their photos.
`add_reviews`	20	Simulates users viewing restaurant details and submitting reviews.
`photo_gallery_browsing_and_uploading`	5	Simulates authenticated users browsing and uploading photos to the restaurant gallery.

The probability values indicate the likelihood of a particular session being executed during the load test. For instance, the `browse_restaurants` session has the highest probability (30%), as browsing is expected to be the most frequent action performed by users. In contrast, the `photo_gallery_browsing_and_uploading` session has the lowest probability (5%) due to its specialized nature and limited expected use cases.

7.1. Tsung Load Configuration

The Tsung load configuration defines a series of arrival phases to simulate an increasing number of users interacting with the application.

- **Phase 1:** 1 user/sec for 60 seconds.
- **Phase 2:** 2 users/sec for 60 seconds.
- **Phase 3:** 4 users/sec for 60 seconds.
- **Phase 4:** 8 users/sec for 60 seconds.

- **Phase 5:** 16 users/sec for 60 seconds.
- **Phase 6:** 32 users/sec for 60 seconds.
- **Phase 7:** 64 users/sec for 60 seconds.

So in the end, the **ramp-up** approach gradually increases user load, which allowed us for the identification of performance bottlenecks under realistic stress conditions. As we already mentioned, each session is assigned a probability (e.g., `signup_test`: 10%, `browse_restaurants`: 30%) to reflect expected usage patterns. The use of `thinktime` introduces realistic pauses between user actions, which simulates human interaction behavior.

8. Horizontal Scaling Analysis

Horizontal scaling, also known as scale-out, refers to the practice of increasing the number of computational instances to distribute the workload across multiple machines. Unlike vertical scaling, which involves upgrading a single machine's resources (e.g., CPU, memory), horizontal scaling enhances system capacity by adding more servers or instances to the infrastructure. This approach is particularly effective for handling increased traffic and user loads, as it allows the workload to be divided among additional instances, which reduces the strain on any single machine.

In our project, we implemented horizontal scaling using AWS EC2 instances with varying configurations. Specifically, we scaled from a single instance (`c5-xl`) to two instances (`c5-xl-2x`), four instances (`c5-xl-4x`) and further to eight instances (`c5-xl-8x`). The scaling process involved replicating the server setup across multiple EC2 instances and balancing incoming traffic using a load balancer. The requests were evenly distributed among the available instances, effectively utilizing the added computational resources, for this setup we used our **realistic scenario** we explained earlier.

8.1. HTTP Requests per Second

Figure 22. HTTP Requests per Second for Different Configurations. Solid lines represent successful requests (`HTTP 200`), while dotted lines represent failed requests (`HTTP 502`).

Figure 22 illustrates the throughput of HTTP requests for the `c5-xl`, `c5-xl-2x`, `c5-xl-4x`, and `c5-xl-8x` configurations. The results demonstrate a clear improvement in scalability with horizontal scaling.

The `c5-xl` configuration reaches its peak throughput early, after which the number of failed requests (`HTTP 502`) begins to rise. The `c5-xl-2x` configuration significantly reduces the failure rate while sustaining a higher throughput, indicating an improvement in resource availability. The `c5-xl-8x` configuration achieves the highest throughput with minimal failures, showcasing its ability to handle increased loads effectively. So, we can say that horizontal scaling from `c5-xl` to `c5-xl-8x` results in substantial performance gains, with `c5-xl-8x` being the most reliable configuration under high loads.

8.2. Mean Duration of HTTP Requests

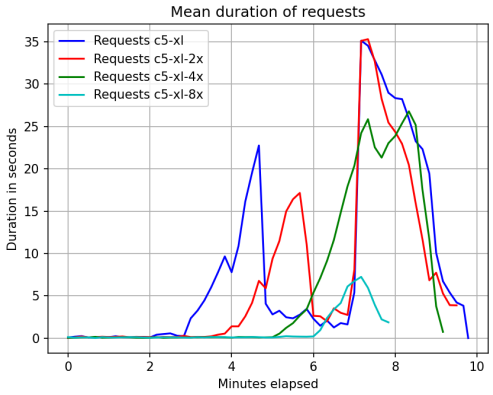


Figure 23. Mean Duration of HTTP Requests Over Time.

Figure 23 shows the mean duration of HTTP requests, which reflects the system's responsiveness. The `c5-xl` configuration shows a significant increase in request durations as the load grows, indicating resource saturation. The `c5-xl-2x` configuration maintains lower response times across the test, which shows improved resource allocation. The `c5-xl-8x` configuration provides the most consistent performance, with some increases in request durations even at peak loads. We can still say that horizontal scaling reduced response times effectively, with the `c5-xl-8x` configuration offering the best responsiveness under high load conditions.

The results emphasize the effectiveness of horizontal scaling in improving system performance. The `c5-xl-2x`, `c5-xl-4x` and `c5-xl-8x` configurations outperform the `c5-xl` configuration in all key metrics. Among these, the `c5-xl-8x` configuration demonstrates the best overall performance, which handles high user loads with minimal failures and consistent response times.

9. Vertical Scaling Analysis

Vertical scaling, also known as scaling up, involves enhancing the computational capacity of a single machine by upgrading its resources, such as CPU, memory, and storage. This approach is particularly useful for applications requiring consistent performance under high workloads without adding the complexity of managing multiple servers. In this section, we analyze the performance of different instance sizes—`c5-large`, `c5-xlarge`, `c5-2xlarge`, and `c5-4xlarge`—to determine how resource upgrades impact HTTP requests/sec, and system latency.

9.1. HTTP Requests Per Second

Figure 24 illustrates the rate of successful (HTTP 200) and failed (HTTP 502) requests per second. The results emphasize the limitations of smaller instances under heavy load. For instance, the `c5-large` instance encounters a sharp increase in HTTP 502 errors as the traffic intensity grows, indicating resource exhaustion. On the other hand, the `c5-4xlarge` instance consistently delivers nearly 100% success rates, even under peak traffic. This demonstrates the ability of larger instances to manage higher throughput while maintaining reliability.

Interestingly, the `c5-xlarge` and `c5-2xlarge` instances display intermediate performance. While they reduce failure rates significantly compared to the `c5-large` instance, they still experience minor instability under extreme traffic conditions, which shows the need for additional resources for applications with high user concurrency.

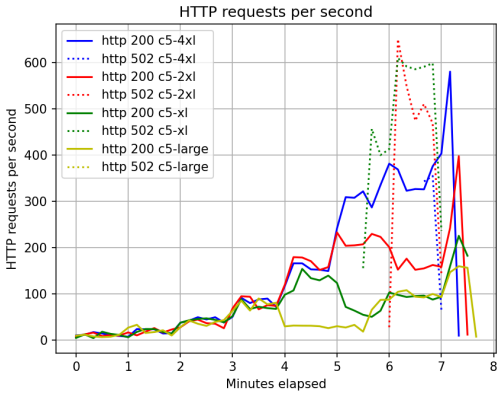


Figure 24. HTTP request success and failure rates across different instance sizes.

9.2. Mean Request Duration

The mean duration of HTTP requests is depicted in Figure 25. Smaller instances, such as **c5-large**, exhibit prolonged request durations as traffic increases, likely due to resource contention and queuing delays. In contrast, larger instances like **c5-4xlarge** maintain low and consistent request durations throughout the experiment. The improved request handling efficiency observed in larger instances shows their suitability for scenarios where latency-sensitive operations are critical.

The results of the vertical scaling analysis demonstrate the clear benefits of upgrading to larger instance sizes for applications with significant traffic demands. Smaller instances, such as **c5-large**, quickly reach their performance limits, which results in increased latency, dropped requests, and lower throughput. Larger instances, like **c5-4xlarge**, are capable of managing high traffic loads while maintaining stability and low latency, which makes them ideal for high-performance environments.

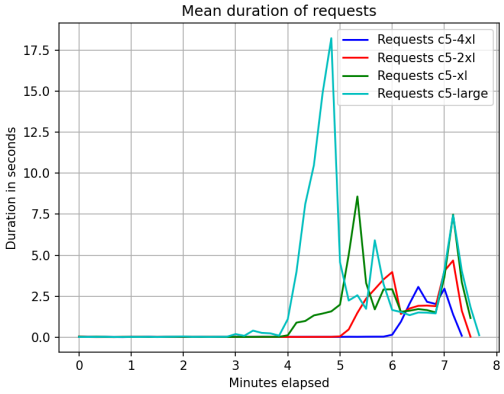


Figure 25. Mean request duration across instance sizes.

10. Conclusion

In this project, we developed and extensively tested **Gaucho Eats**, a scalable internet service platform tailored for both restaurant owners and customers in Santa Barbara. Our system was architected as a React frontend coupled with a Rails backend, which allowed for a clean separation of concerns, modular code organization, and straightforward scalability. We implemented a robust database schema to efficiently store and manage complex relationships between entities such as restaurants, users, dishes, reviews, and comments.

To ensure that the platform can handle real-world demands, we conducted comprehensive load testing using Tsung. By simulating realistic user behavior through carefully crafted workflows, we were able to pinpoint bottlenecks and evaluate our optimization efforts. Notable performance enhancements included:

- **Pagination:** Reduced response times and improved throughput under high concurrency by limiting large dataset fetches.
- **Caching:** Precomputed values for frequently accessed data minimized database lookups, leading to lower latency and higher throughput.
- **N+1 Query Optimization:** Eager loading of associated records prevented redundant database queries, allowing the system to handle heavier loads efficiently.
- **Optimized Image Storage with Amazon S3:** Offloading image handling to S3 decreased database load and response sizes, resulting in improved performance.

After applying these optimizations, we assessed the platform's ability to scale both vertically (upgrading instance sizes) and horizontally (adding more instances). The load tests showed clear performance improvements as we increased resources. Horizontal scaling across multiple EC2 instances and vertical scaling to larger instance types both led to substantial enhancements in request throughput and reduced latency. At higher levels of scaling, the platform sustained minimal failures while maintaining low request durations, demonstrating robust scalability.

In summary, our efforts resulted in a performant, responsive, and maintainable system capable of accommodating a growing number of users and evolving business requirements. Our methodology and results underscore the critical importance of systematic load testing, iterative optimization, and strategic scaling to build and maintain high-quality, user-centric web services.