The Primary Keys

COS 221 Practical Assignment 5

u24569608 - Mr. Connor Bell
u23536030 - Mr. Dewald Colesky
u24634434 - Mr. Adriano Jorge
u05084360 - Miss. Zoë Joubert
u23544563 - Miss. Inge Keyser
u23608821 - Miss. Megan Lai
u24594475 - Mr. Ferdinand Johannes Nel

Task 1: Research

1. What Consumers Buy Online

Clothing and shoes are the most popular online purchases worldwide. In the U.S., 57% of shoppers bought clothes online in the past year, while 47% bought footwear. Electronics are also a big category, with 40% of U.S. consumers making purchases in this area. In Germany, fashion leads at 66%, followed by electronics (50%), and groceries—which saw a huge jump from just 7% in 2020 to 25% in 2024. Personal care products also grew significantly, doubling from 18% to 35% in the same period. Globally, 34% of shoppers buy something online at least once a week, showing how common digital shopping has become [1][2][4].

2. How Shoppers Compare Prices

Price comparison tools help consumers find the best deals by showing prices from different retailers in one place. Popular options include Google Shopping, Shopzilla, and Camelcamelcamel. For businesses, tools like 42Signals track competitor prices in real time, monitor stock levels, and even detect when sellers break pricing agreements. Research shows these tools make shoppers more likely to switch stores for a better price—especially those who are budget-conscious. Newer systems even use web scraping to provide instant price updates across major online stores [5][6][7][11].

3. Top Online Shopping Categories

Fashion is still the biggest e-commerce category globally, followed by electronics, books/media, and cosmetics. In the U.S., household appliances (27%) and furniture/home goods (19%) are also popular. Fast-growing niches include sports gear, toys, and DIY/garden supplies. Interestingly, over half of online shoppers (52%) buy from international retailers, proving that e-commerce has no borders [1][4].

4. Why User Experience (UX) Matters

A smooth, easy-to-use website directly impacts sales—better UX means more conversions, higher spending, and fewer abandoned carts. Studies show that poor design can drive away up to 80% of potential customers, especially if the checkout process is too complicated. Key features that improve UX include:

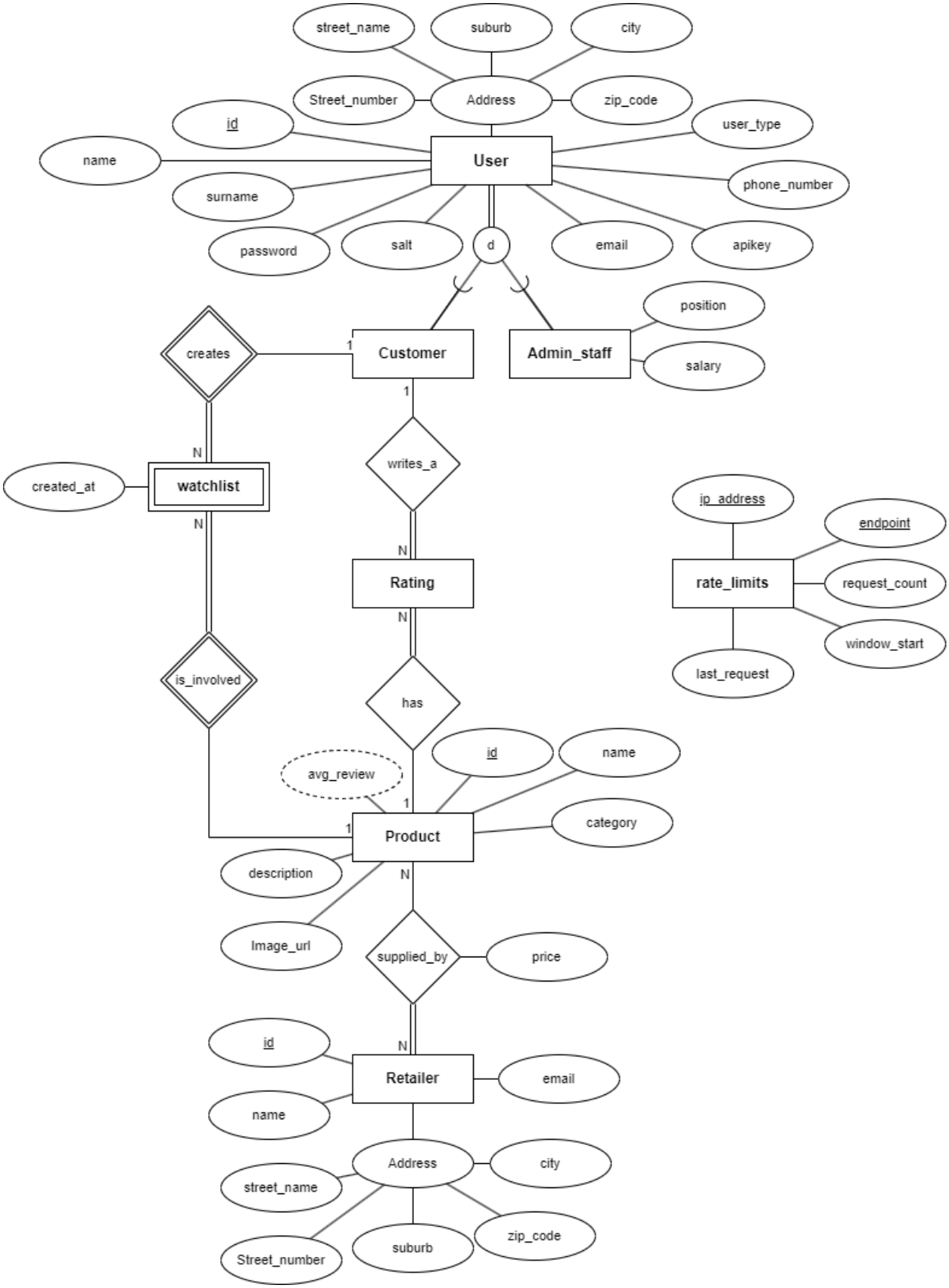
- Simple navigation and search
- Quick, hassle-free checkout
- Clear product reviews and reviews

- Mobile-friendly design Companies that invest in UX gain a long-term advantage because shoppers now expect fast, intuitive online experiences [7][8][9][10].

References:

[1] Sellers Commerce, “51 eCommerce Statistics In 2025 (Global and U.S. Data),” Apr. 15, 2025.
[2] C. Stacey, “Tech-wary Germans get hooked with online shopping habit,” Reuters, Nov. 25, 2024.
[3] A. Rogers and J. Hitchcock, “20 Trending Products and Things to Sell Online (2025),” Shopify, Mar. 25, 2025.
[4] “Most Popular Online Shopping Categories,” Mobiloud, 2025.
[5] “23 Price Comparison Apps, Tools, and Websites (2025),” Shopify, 2025.
[6] “The Art of Price Comparison: Tools and Tips for Online Retailers,” 42Signals, 2023.
[7] A. Smith et al., “The Influence of Price Comparison Websites on Online Switching Behavior,” PMC, 2020.
[8] “The Importance Of User Experience In eCommerce Design,” Wizzy.ai, 2023.
[9] J. Doe, “Clunky Websites Are Costing Companies Money, Scaring Away Shoppers,” Investopedia, 2024.
[10] “Benefits of UX design in E-commerce development,” Capturly, 2023.
[11] F. Chen et al., “Research on Real-time E-commerce Price Comparison System Using Python Web Scraping Technology,” ResearchGate, 2024.

Task 2: (E)ER-Diagram


street_name
suburb
city
Street_number
Address
zip_code
id
user_type
name
User
surname
phone_number
password
salt
d
email
apikey
position
creates
1
Customer
Admin_staff
salary
1
N
writes_a
created_at
watchlist
ip_address
endpoint
N
Rating
rate_limits
request_count
N
window_start
last_request
is_involved
has
id
name
avg_review
1
category
1
Product
description
N
Image_url
supplied_by
price
N
id
Retailer
email
name
Address
city
street_name
zip_code
suburb
Street_number

Comments:

- User is a strong entity with a disjoint specialization based on user_type. It has total participation: every User must be either a Customer or Admin_staff. Disjointness means that a user can be only one of the two, never both. A person in our system must have a role (either admin or customer), and cannot hold both roles simultaneously. This enforces clear access control logic (e.g., customers view and watchlist products, admins manage products, add retailers, etc.).

- avg_review is a derived attribute of Product. It is calculated from review entries (score) that are associated with this product.

- Watchlist is a weak entity: It cannot exist without both a Customer and a Product. It has no unique attribute to identify it alone. It is identified by a composite key (customer_id and product_id).

- review is a strong entity: It has its own primary key (id), and while it references both Customer and Product, it can exist independently.

- The rate_limits entity is a standalone system-level control table used to enforce rate-limiting mechanisms on the API. It is not part of the business (e.g., users, products, orders), but is essential to prevent API abuse. endpoint and ip_address form a composite primary key, uniquely identifying each tracked client-endpoint pair. This entity has no relationships to any other entities - its role is entirely operational/systemic.

Relationships:

- Customer creates Watchlist: One customer creates many watchlist entries. A customer might not create any watchlist entries. Every watchlist entry must be created by a customer. Customers can save products to their watchlist, but aren't required to

- Product is_involved Watchlist: One product can be in many watchlists. A product might not be in any watchlists. Every watchlist entry must reference a product. Products can be watched by multiple customers

- Customer writes_a review: One customer writes many reviews. A customer might not write any reviews. Every review must be written by a customer. Customers can review products, but aren't required to

- Product has review: One product has many reviews. A product might not have any reviews yet. Every review must be for a product. Products accumulate reviews over time

- Product supplied_by Retailer: Many products are supplied by many retailers. A product might not be supplied by any retailer. A retailer might not supply any products. Retailers carry inventory of products at specific prices. The price for this product as sold by this retailer is stored in a "price" attribute.

Task 3: (E)ER-diagram to Relational Mapping

Step 1: Mapping of regular entity types: Regular entities are mapped in this step.

rate_limits

ip_address	endpoint	request_count	window_start	last_request

User

id	name	surname	user_type	email	password	salt	phone_number	apikey	street_number	street_name	suburb	city	zip_code

Rating

id	score	description	updated_at

Product

id	name	description	Image_url	category

Retailer

id	name	email	street_number	street_name	suburb	city	zip_code

Step 8: Mapping specialisation and generalisation: We need to do this step early because we need to use CUSTOMER in the coming steps. User is totally disjoint, specialised into two types: CUSTOMER and ADMIN_STAFF. We use approach 8A in the slides of L14: That is, create a relation for each subclass (customer and admin_staff) with the key of the superclass (user) and the attributes of this subclass. The key of the superclass will be the key of the subclasses, and links it to the superclass.

rate_limits

ip_address	endpoint	request_count	window_start	last_request

User

id	name	surname	user_type	email	password	salt	phone_number	apikey	street_number	street_name	suburb	city	zip_code

Customer

user_id

Admin_staff

user_id	salary	position

Rating

id	score	description	updated_at

Product

id	name	description	Image_url	category

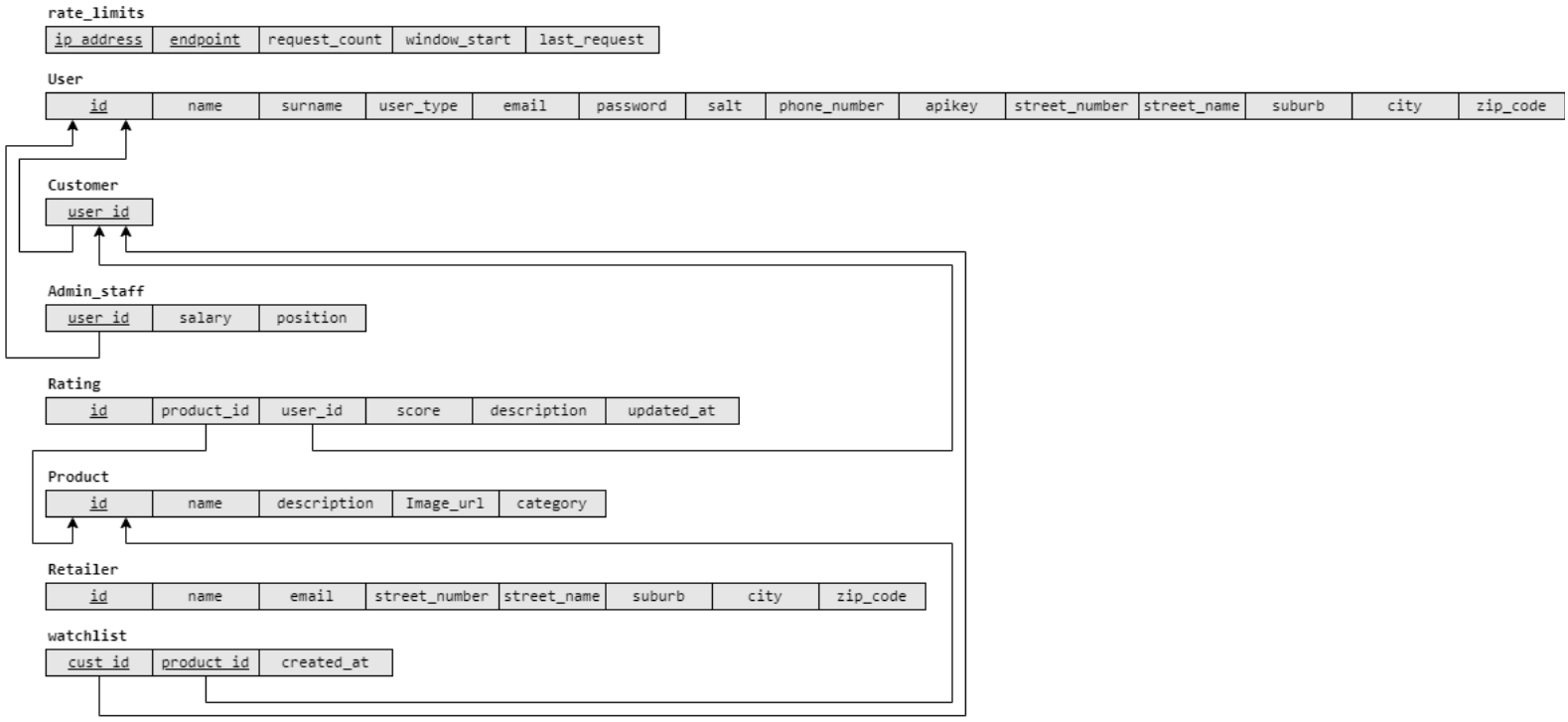
Retailer

id	name	email	street_number	street_name	suburb	city	zip_code

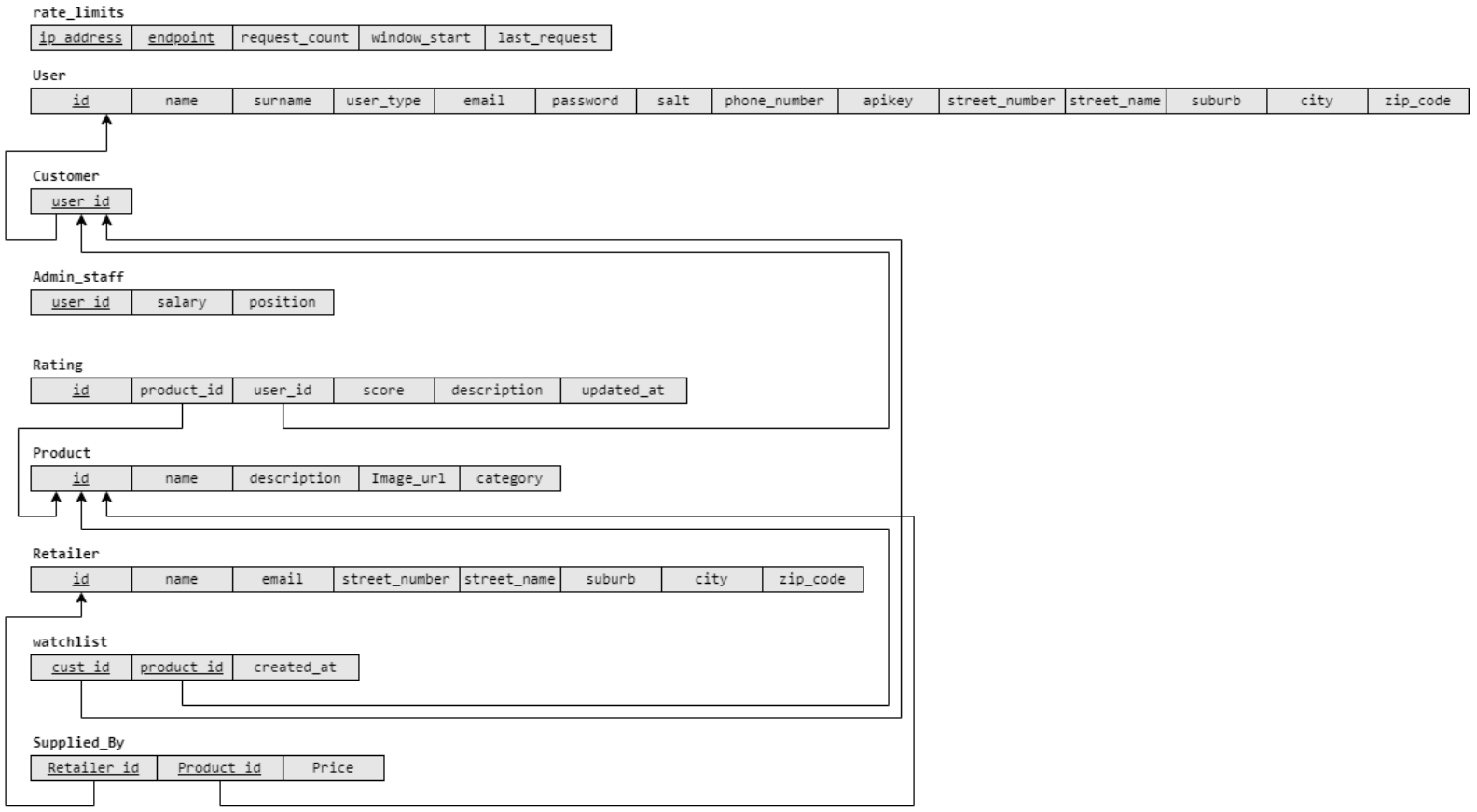
Step 2: Mapping of weak entity types: We map weak entities in this step. WATCHLIST is the only weak entity in our EERD, since it depends on both a CUSTOMER and a PRODUCT to exist: Without them, the watchlist entry cannot exist. review is not a weak entity since reviews can be updated over time, reviews may need to be deleted, edited or moderated independently of the user or product that it is linked to.

Step 3: Mapping of Binary 1:1 relationships: **We do not have binary 1:1 relationships in our EERD. We do nothing in this step**

Step 4: Mapping of Binary 1:M relationships: **We have four 1:M relationships in our EERD: CUSTOMER creates WATCHLIST, PRODUCT is_invloded WATCHLIST (both of these are already mapped in Step 2), CUSTOMER writes_a review and PRODUCT has review. We use Approach 1 in the slides of L14**



Step 4: Mapping of Binary N:M relationships: We have one binary N:M relationship in our EERD: PRODUCT supplied_by RETAILER. We create a new relation called SUPPLIED_BY to support this and we add a price attribute since it is an attribute that belongs to the supplied_by relationship.



Step 6: Mapping of multivalued attributes: **We do not have multivalued attributes in our EERD. We do nothing in this step.**

Step 7: Mapping of N-ary relationships: **We do not have N-ary relationships in our EERD. We do nothing in this step.**

Step 8: Mapping specialisation and generalisation: **Step 8 was already done directly after step 1 to ensure that CUSTOMER exists before we map relationships. We do not have other specialisations in our EERD.**

Step 9: Mapping unions: **We do not have unions in our EERD. We do nothing in this step.**

FINAL RELATIONAL MAPPING

Task 4: Relational Schema

```
CREATE TABLE `Admin_staff` (
  `user_id` int(11) NOT NULL,
  `salary` decimal(10,2) DEFAULT NULL,
  `position` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

CREATE TABLE `Customer` (
  `user_id` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

```sql
CREATE TABLE `Product` (
  `id` int(11) NOT NULL,
  `name` varchar(100) NOT NULL,
  `description` text DEFAULT NULL,
  `Image_url` text DEFAULT NULL,
  `category` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

CREATE TABLE `rate_limits` (
  `ip_address` varchar(45) NOT NULL,
  `endpoint` varchar(50) NOT NULL,
  `request_count` int(11) NOT NULL DEFAULT 1,
  `window_start` datetime NOT NULL DEFAULT current_timestamp(),
  `last_request` datetime NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

CREATE TABLE `Retailer` (
  `id` int(11) NOT NULL,
  `name` varchar(100) NOT NULL,
  `email` varchar(100) NOT NULL,
  `suburb` varchar(100) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL,
  `street_name` varchar(100) DEFAULT NULL,
  `street_number` varchar(10) DEFAULT NULL,
  `zip_code` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

CREATE TABLE `Supplied_By` (
  `Retailer_id` int(11) NOT NULL,
  `Product_id` int(11) NOT NULL,
  `Price` float NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

```sql
CREATE TABLE `User` (
  `id` int(11) NOT NULL,
  `name` varchar(50) DEFAULT NULL,
  `surname` varchar(50) DEFAULT NULL,
  `phone_number` varchar(22) DEFAULT NULL,
  `apikey` varchar(255) DEFAULT NULL,
  `email` varchar(100) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `street_number` varchar(10) DEFAULT NULL,
  `street_name` varchar(100) DEFAULT NULL,
  `suburb` varchar(100) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL,
  `zip_code` varchar(15) DEFAULT NULL,
  `user_type` varchar(50) DEFAULT NULL,
  `salt` text NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

CREATE TABLE `watchlist` (
  `cust_id` int(11) NOT NULL,
  `product_id` int(11) NOT NULL,
  `created_at` timestamp NOT NULL DEFAULT current_timestamp()
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

ALTER TABLE `Admin_staff`
  ADD PRIMARY KEY (`user_id`);

ALTER TABLE `Customer`
  ADD PRIMARY KEY (`user_id`);

ALTER TABLE `Product`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `rate_limits`
  ADD PRIMARY KEY (`ip_address`,`endpoint`);
```

```sql
ALTER TABLE `Rating`
  ADD PRIMARY KEY (`id`),
  ADD KEY `user_id` (`user_id`),
  ADD KEY `product_id` (`product_id`);

ALTER TABLE `Supplied_By`
  ADD PRIMARY KEY (`Retailer_id`,`Product_id`),
  ADD KEY `product` (`Product_id`);

ALTER TABLE `watchlist`
  ADD PRIMARY KEY (`cust_id`,`product_id`),
  ADD KEY `fk_watchlist_product` (`product_id`);

ALTER TABLE `Product`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=60;

ALTER TABLE `Rating`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=658;

ALTER TABLE `Retailer`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=18;

ALTER TABLE `User`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=10031;

ALTER TABLE `Admin_staff`
  ADD CONSTRAINT `Admin_staff_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `User` (`id`) ON DELETE CASCADE;

ALTER TABLE `Customer`
  ADD CONSTRAINT `Customer_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `User` (`id`) ON DELETE CASCADE;

ALTER TABLE `Rating`
  ADD CONSTRAINT `rating_1` FOREIGN KEY (`user_id`) REFERENCES `User` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `rating_2` FOREIGN KEY (`product_id`) REFERENCES `Product` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `Supplied_By`
  ADD CONSTRAINT `product` FOREIGN KEY (`Product_id`) REFERENCES `Product` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `retailer` FOREIGN KEY (`Retailer_id`) REFERENCES `Retailer` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `watchlist`
  ADD CONSTRAINT `fk_watchlist_customer` FOREIGN KEY (`cust_id`) REFERENCES `Customer` (`user_id`) ON DELETE CASCADE,
  ADD CONSTRAINT `fk_watchlist_product` FOREIGN KEY (`product_id`) REFERENCES `Product` (`id`) ON DELETE CASCADE,
  ADD CONSTRAINT `watchlist_customer` FOREIGN KEY (`cust_id`) REFERENCES `Customer` (`user_id`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `watchlist_product` FOREIGN KEY (`product_id`) REFERENCES `Product` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;
```

Task 5: Web-based Application

Web UI

1. **Landing Page**
    - The user can register an account.
    - The user can log in if they already have an account.
2. **Admin Panel**
    - Admins can edit products.
    - Admins can add products.
    - Admins can delete products.
3. **Product Listings**
    - The user can view all products
    - The user can filter products based on:
        - category
        - name
        - price
        - retailer
    - The user can expand product information to:
        - see reviews,
        - see average review and
        - see retailers,
        - visit retailer website and
        - watchlist a product
4. **Review Products**
    - The user can review a product
    - The user can update their review of a product
5. **Watchlist**
    - Add products to watchlist
    - Remove products from watchlist
    - Receive updates

API

1. **User management**
    - The user can register an account using a valid reCAPTCHA token.
    - The user can log in using a valid reCAPTCHA token, and their user_type, API key and name is then returned by the API.
2. **Admin functionality**
    - Admins can view:
        - All users and their account details (excluding personal information like passwords).
        - All products and their related attributes.
        - All retailers and their associated details.

        - Recent reviews.
    - Admins can add:
        - Users (customers or staff)
        - Products
        - A price and retailer associated with a product
        - Retailers
    - Admins can edit:
        - Users (including their user_type and other information)
        - Products
        - Prices and retailers associated with a product
        - Retailers
    - Admins can delete:
        - users and their associated information,
        - products and all information associated with the products,
        - retailers and all information associated with the retailers and
        - the reviews.
3. **Customer functionality**
    - Customers can view:
        - All the unique categories of products.
        - All the products filtered by category, name and/or minimum average review.
        - All the products are sorted by cheapest price, name or average review.
        - Detailed information about a product (including retailers, prices and reviews).
        - Products in their watchlist.
        - Their product reviews (including product details and retailer details).
        - Their account details .
        - General statistics about reviews.
    - Customers can add:
        - Products to their watchlist.
        - One review per product.
    - Customers can edit:
        - Their product reviews.
        - Their account details.
    - Customers can delete:
        - A product from their watchlist.
        - A review that the customer wrote.
4. **Security functionality**
    - All endpoints, except for login and register, should require the current user's API key.
    - Register and login endpoints should require and authenticate a reCAPTCHA token.
    - Rate limiting should be enforced per IP per endpoint to prevent API abuse.
    - All inputs should be validated and protected from SQL injection.
5. **General functionality**
    - All requests to the API should be made using POST with a JSON object as the payload.
    - All responses from the API should be in JSON format.
    - Successful responses should include a status, timestamp, message and data.

- Error responses should include a status, timestamp and error message.
- All responses should use correct and appropriate HTTP status code.

Database

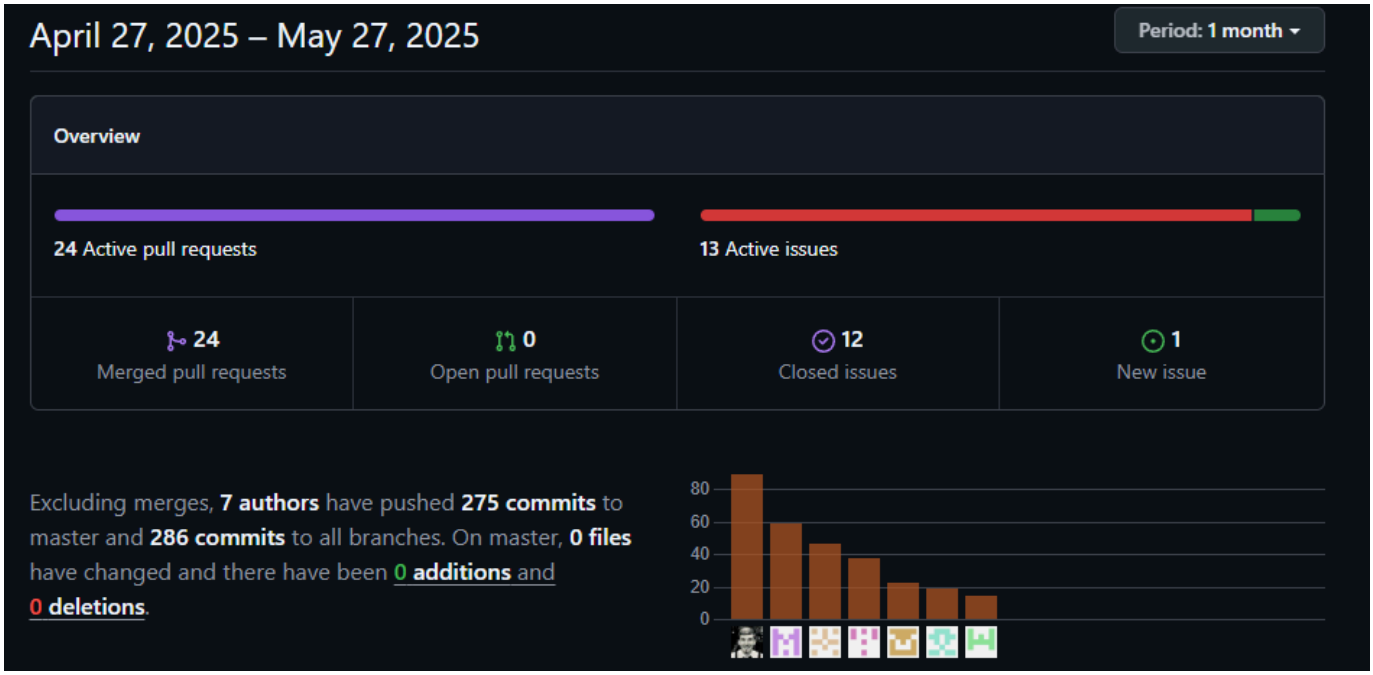
1. **Storage**
   - The database should store all required information
     - Customer information
     - Admin information
     - Product information
     - review information
     - Customer watchlists
2. **Access**
   - Only authorized users can have access to the database

Task 6: Data

- **https://www.pexels.com/**
  - Images for products
- **https://www.mockaroo.com/**
  - Retailer information
- **Custom python scripts**
  - Python scripts for:
    - Products
    - reviews
    - Supplied By

```
    sql_statements = []
    for k in range(1,52): #for each product
        for i in range(1,11): #create 10 reviews
            score = random.randint(1,5)
            if score < 2:
                description = random.choice(list(bad_description.values()))
            else:
                description = random.choice(list(good_description.values()))

            sql = f"""INSERT INTO Rating (score, description, user_id, product_id)
    VALUES ('{score}', '{description}', '{i}', '{k}');"""
            sql_statements.append(sql)

    with open(output_sql_file, 'w') as f:
        for statement in sql_statements:
            f.write(statement + '\n')

    print(f"SQL statements written to {output_sql_file}")

if __name__ == "__main__":
    convert_to_sql('rating.sql')
```

Task 8: Development



Task 9: Demo

Connor Bell:

- Implemented and managed the client side reCAPTCHA for login and register.
- Created the customer side front end GUI. Created the GUI for the all products page, top-rated products page, reviews dashboard page, product view page, my details, my reviews and my watchlist, login and register pages.
- implemented JS to handle the theme for the website and the user dropdown box.
- handled JS to populate the user name into the user dropdown box.
- managed the website on Wheatley.

Dewald Colesky:

- Implemented JS for product view page. (Fetch product information, connected suppliers, reviews, ability to review)
- Implemented JS for review dashboard. (Fetch all reviews, put into charts and summarize scores, as well as getting high and low scores)
- ReadME
- Functional Requirement document
- Initial API functionality (Login, register, view all products,rate product, addproduct, updateproduct, delete product, view reviews, filter products, update customer, update admin, view supplier)

- Data generation for most tables (Excluding User,Customer, Admin, Watchlist)

Adriano Jorge:

- Designed the admin.html page later converted to php and implemented it with js. This allowed the admin to quickly add a user, see recent reviews and quickly edit product prices.
- Implemented the gui and working functionality for admin_products with allows admins to add a product, edit a product, delete a product as well as a table view of the products.
- Implemented and designed the admin_users allowing to add a staff user edit their salary and delete users,Aswell as added a table to view all users staff and customers.
- Implemented and designed the admin_retailers , adding editing and deleting. As well as table view of retailers.
- Implemented the JavaScript to use the api to load products . View the products(view.js) and add the products to watchlist as well as the watchlist page JavaScript .
- Implemented multiple versions of the EERD diagram.

Zoë Joubert:

- Designed and implemented the Products (products.html) and Highest Rated Products (highest_rated.html) pages, including separate HTML, CSS, and JavaScript files for each.
- Built the full frontend layout and styling using custom CSS, ensuring consistency with the overall site design.
- Developed the JavaScript logic in products.js and highest_rated.js to load product data dynamically from the API using XMLHttpRequest.
- Integrated filter functionality by calling API endpoints to fetch filtered product data based on criteria such as category and rating, updating the UI in real time.
- Collaborated on UI/UX decisions across pages to ensure a responsive and user-friendly interface.

Inge Keyser:

- Worked with Megan on login scripts
- Implemented the registration script
- Worked on the powerpoint presentation
- Revised my_reviews.php and implemented my_reviews.js script
- Task 7: Analyse and Optimise a query

Megan Lai:

- Launch page: Designed and implemented the launch page. This is the first page that will be seen on our website. It prompts the user to either login or register.

- Login page (JS functionality): Implemented the javascript for the login page. The javascript makes requests to the api and does client side validation of the user's email and password. It also stores the user's api key in local storage as well as redirects the user to the correct page depending on whether they are a customer or admin upon successful login.
- Register page (JS functionality): Implemented the javascript for the login page. The javascript makes requests to the api and does client side validation to check that the information being entered is valid. It checks that all the fields are filled in. The validation ensures that the user's name and surname are more than two characters long, the email address contains the '@' symbol and the password is at least 8 characters long, contains both uppercase and lowercase letters, numbers and special characters.
- My details page (JS functionality): Implemented the javascript for the My Details page. Allows a customer to update their personal details and allows the user to add additional information such as phone number and their house address.
- .env file usage for localhost testing:Implemented code for all pages to retrieve the correct credentials from the .env file and uses these credentials with basic authentication to access the API and database on the Wheatley server in order to test our website on localhost first before uploading to the Wheatley server.
- Github management:Created and set up the GitHub repository. Ensured that the .env is never uploaded to the GitHub by setting up the gitignore to ignore all files with that file extension. Setup the branching system. Reviewed, edited and merged code from different branches to ensure that all pages work individually as well as with the rest of the website.

Johan Nel:

- Reviewed and approved the EERD diagrams created by Adriano, ensuring they adhered to database design best practices and to Chen's notation.
- Performed the relational mapping of our EERD using the steps outlined in Lecture 14's slides.
- Designed, updated, and implemented a PHP 7.3 API that supports a wide range of request types as required by the front-end team. The API allows for secure user authentication, retrieval, addition, modification, and deletion of database records. It also includes product sorting, filtering, fuzzy search, and aggregated data such as average product reviews, cheapest product price, etc.
- Wrote comprehensive API documentation to assist front-end developers in efficiently using the API. The documentation clearly outlines endpoints, expected request formats, and response structures to minimise confusion.
- Integrated reCAPTCHA verification into the registration and login API endpoints. This ensures only human users can create or access accounts by validating the front-end (client-passed) reCAPTCHA token via Google's reCAPTCHA API.
- Implemented database-driven rate limiting to prevent abuse of API endpoints. The mechanism tracks requests per IP and endpoint to enforce cooldown periods.

- Documented the API's functional requirements in a structured format. These include features related to user management, product and retailer listings, reviews, watchlists, administrator functionality, customer functionality and review statistics.
- API implementation and documentation

Task 10: Bonus Task - Push the Boundaries

2. Security First - Secure Your App

We use password hashing implementing salt strings together with hashing to secure passwords.

When a user registers, changes their password or get manually added by an admin, the API follows a multi-step process to protect the user's password:

1. Password Validation: The API checks the password against a set of rules (At least 8 characters, must include upper and lower case letters, a number, and a special character.) to ensure it meets minimum security standards.
2. Salt Generation: A unique salt is generated for each user.
    - The salt is concatenated with the user's plain-text password before their password. The salt is generated using bin2hex(random_bytes(16)), which creates a 32-character hexadecimal string. random_bytes(16) is cryptographically secure and generates 16 random bytes, which is then converted to a hexadecimal string (32 characters).
    - The salt is stored in the database alongside the user's hashed password to allow for password verification during login.
    - Salt is used to protect users against rainbow table attacks.hashed (see below).

3. Password Hashing: The API uses PHP 7.3's password_hash() function to hash the salted password. This function uses the bcrypt algorithm by default, which is a strong and secure hashing algorithm that is slow by design (to protect against brute-force attacks).
    - The password_hash() function automatically generates a secure hash which is stored in the database.
    - The resulting hash is a 60-character string that includes the algorithm, cost factor, salt, and hashed password. This means that the password is effectively double-salted:
        - The first salt is the one generated by random_bytes(16) and stored in the database.
        - The second salt is automatically generated by password_hash() and is included in the resulting password hash (the 60 character string that is stored in the database).
    - The user's password itself is never stored in the database, and the API never logs or exposes the plain-text password in any way.
    - When using password_hash() the user's password can never be retrieved in its original form, as the hashing process is one-way and irreversible (the hash is immutable).
4. Password Verification: To verify a user's password during login, the API uses password_verify() to compare the plain-text password provided by the user with the salted and hashed password

stored in the database. This function automatically handles the salt and hashing algorithm used during registration to verify the password without needing to know the original password.

Login attempts are rate-limiting:
How Rate Limiting Works

Database-Backed:

Rate limiting is implemented using a dedicated rate_limits table in the database. This allows for accurate and persistent tracking of request counts and time windows to ensure that rate limits are enforced correctly.

Per-IP and Per-Endpoint:

Limits are tracked for each unique combination of IP address and the endpoint that this IP is accessing. Each endpoint may have a different rate limit.

Time Window:

The time window for rate limiting is typically 60 seconds (1 minute), but may vary per endpoint.

Limits:

The number of allowed requests per endpoint per minute varies based on how computationally expensive the endpoint is.

For example:

Register and Login: 5 requests per minute per IP
getReviewStats (all stats): 3 requests per minute per IP
getReviewStats (single stat): 10 requests per minute per IP

Most other endpoints allow for 10–20 requests per minute per IP, depending on the endpoint's complexity.
How it works:

On each request, the API checks the rate_limits table for your IP and the endpoint.
If you have not exceeded the allowed number of requests in the current time window, your request proceeds and your request count is incremented.
If you exceed the limit, the API immediately returns a 429 error and does not process your request.
When the time window expires, your request count is reset.

Error Response Example
If you exceed the rate limit, you will receive a response like this:


```
{
   "status": "error",
   "timestamp": 1748000000000,
```

```
    "code": 429,
    "message": "Rate limit exceeded. Please try again later."
}
```