



TRACK MAZON

**PRICE TRACKING APP
FOR YOUR WISHED
AMAZON PRODUCTS**

**Final Project
CFGDegree
Software Engineering Stream
2024**

DEVELOPED BY

EVA PEREZ CHIRINOS

VIOLETA PEREDA

LEYLA BUSH

SHAIRA JIWANY

IKRAM AHMED

In an ever-changing e-commerce market, where trust is often compromised,

TRACKMAZON aims to bring clarity and confidence back to your online shopping experience.

CONTENTS

1. Introduction & Project Goals.....	3
2. Background & User Journey.....	3
2.1 What is TRACKMAZON ?.....	3
2.2 How can our app help the shopper?.....	3
3. Specifications and Design.....	4
3.1 Functional vs non-functional requirements.....	4
3.2 Design and architecture.....	4
4. Implementation and Execution.....	6
4.1 Development approach and team member roles.....	6
4.2 Tools and libraries.....	7
4.3 Implementation process.....	8
4.4 Agile development.....	8
4.5 Development Challenges	9
5. Testing and Evaluation.....	9
5.1 Testing strategy.....	9
5.2 Functional and user testing.....	9
5.3 System limitations.....	10
6. Conclusion.....	10

CLICK TO FIND US ON



1. Introduction & Project Goals

In a nutshell, **TrackMazon** is a price-tracking app for Amazon products that offers users to receive an email notification when a tracked product falls within their set desired price.

The **core aim** of this project is “to develop a fully deliverable, *plug-and-play*, price tracking application.”

Like this, **we target the non-developer user***. We aim to deploy a product *anyone can use*, without needing any prior software development knowledge.

*** At the moment, the fully automated version of this app (trackmazon.exe) is only available for Windows users. Unix-like users will need to run this app through the source code.**

Our main **objectives** are:

- ★ To develop a fully encapsulated system to avoid any manual user configuration, and achieve a “*plug-and-play*” UX.
- ★ To develop a user-friendly CLI interface for managing tracked products and email preferences.
- ★ To develop an in-code, automated daily web scraping system to retrieve product information from specified and stored URLs.
- ★ To store and manage user and product data securely in a SQLite database.
- ★ To implement MailJet Email API system to send email notifications.

2. Background & User Journey

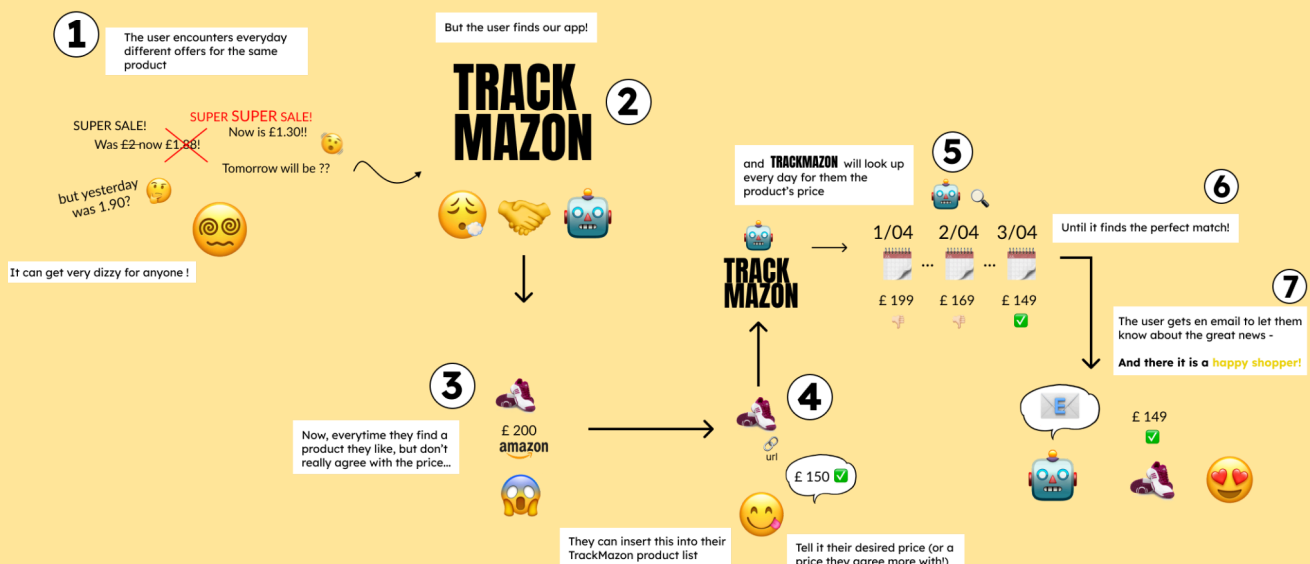
2.1 What is **TRACKMAZON** ?

The modern world of e-commerce can be challenging for consumers to navigate. It can be volatile - and sometimes manipulative; with continuous fluctuating prices, as well as misleading promotional offers.

We all know Amazon and how daunting it can get to get the best deal for your desired product.

But not to worry, **TrackMazon** *wants to become your shopping ally*.

2.2 How can our app help the shopper?

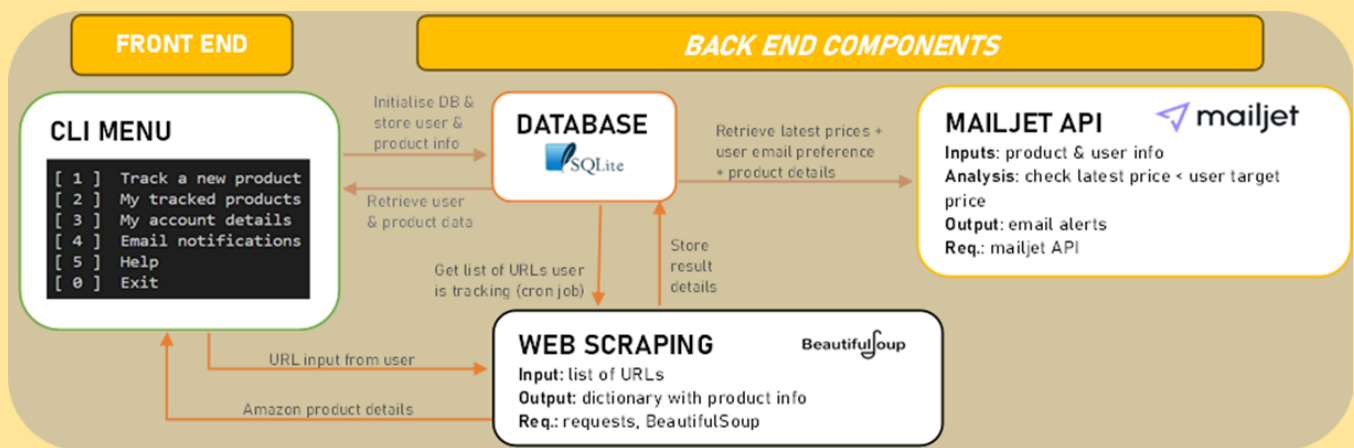


3. Specifications and Design

3.1 Functional vs non-functional requirements

Functional requirements	Non-functional requirements
User Interface - provide a friendly CLI for users to interact with the system	Usability - the CLI should be easy to navigate for users with simple computer skills
User Management - users must be able to create an account, add products to track, set a threshold price and manage their account information via the front-end	Scalability - using SQLite means we only need to store information for one user at a time; eliminates the issue of having to manage a large and complex database with many users
Database configuration and interactions - store and retrieve user data, product information, and price history	Security - user data must be stored securely in the database
Web Scraping - the application must be able to extract product information (title, current price, currency) from a given url (namely Amazon product pages)	Performance - web scraping should be complete within a reasonable time frame (such as under 5 seconds per product)
Price Alert Email Notification - An email notification must be sent to the user when the current price of a product they are tracking falls below their set threshold	Robust code that handles errors gracefully - error messages should be displayed to the user in a clear and easy to understand manner

3.2 Design and architecture



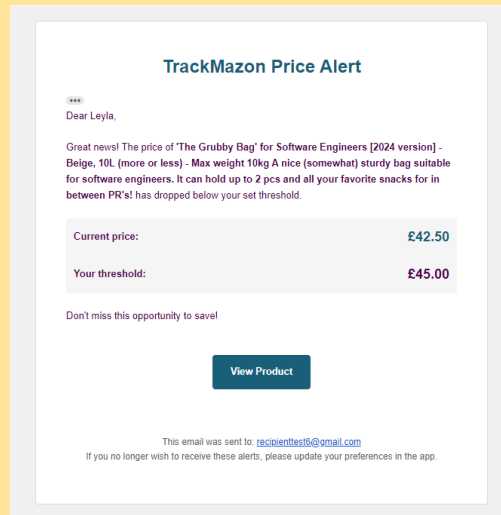
TrackMazon's Architecture Diagram

We opted for CLI for the **Front End** side of the project ([See our Figma Project](#)) and a cyclical UX flow revolving around the app's Main Menu selection (See diagram below).

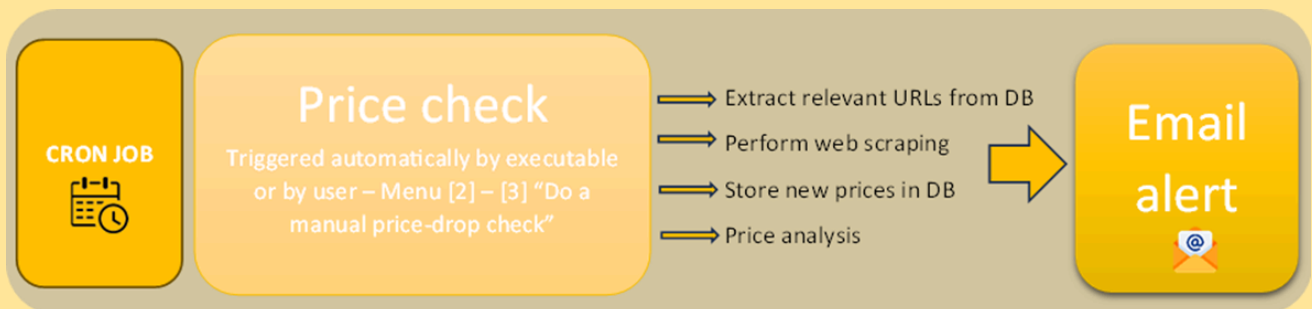
For **Web Scraping**, we considered both BeautifulSoup and Selenium libraries, but ruled out the latter which opens up a browser while scraping as it would interfere with the user experience . We decided the combination of BeautifulSoup and requests libraries was ideal for our project.

We decided to use **MailJet API** for sending email notifications to the user when the price of their chosen product falls below their threshold price; MailJet was a suitable option as it is well-documented and allows us to send up to 200 emails a day on a free plan. We registered an account on MailJet with a sender email address: group6.cfgdegree24@gmail.com, to access the API credentials required for authentication. We also created another file called `email_api_db_interaction.py` which was used to retrieve the user and product

information from the database. After several iterations, we finalised this simple and personalised email design:



In order to automate price-drops checks and user notification, we introduced a **Cron-Job workflow** to perform the actions described below.



Cron Job Diagram

Database - Our project’s goal is to create a fully deliverable, plug-and-play software application — something that users can download and use immediately, without needing any technical expertise or additional setup. We found MySQL to be impractical for this purpose, and **SQLite** to perfectly fit our criteria:

- MySQL presented two significant challenges:

✗
Need deployment and Maintenance on a cloud platform
like AWS or Azure, which would incur costs and require expertise beyond the scope of our project.

✗
Otherwise, users would need to set up their own MySQL server and manually configure DB configuration
Increasing the risk of app malfunction, and limits our target audience to one with software development knowledge

These obstacles made MySQL an unsuitable option for our project’s vision.

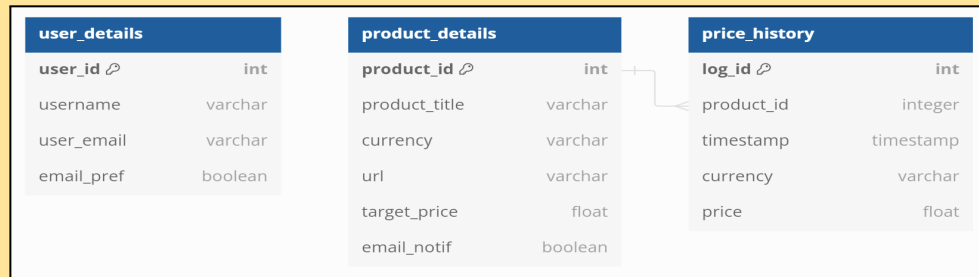
- We found SQLite, on the other hand, to fit perfectly to our project’s aim for the following reasons:

✓
It is self-contained:
requiring minimal support from the operating system that it is run on, thus making it compatible for either Windows or MacOSX.

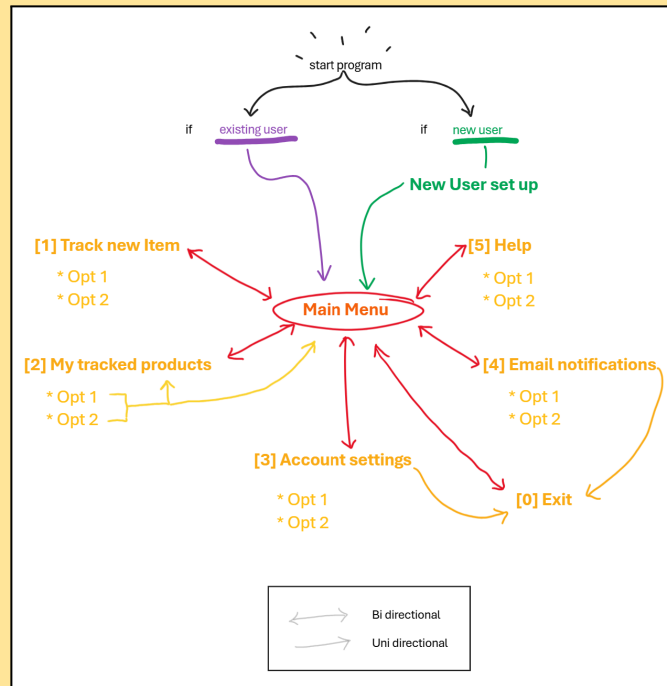
✓
Needs zero-configuration from user:
streamlining the app’s use and eliminating the risk of malfunction.

✓
It is serverless:
its architecture revolves around directly interacting with a single file (.db) that can be created and managed directly within our Python code.

✓
It is lightweight and independent:
The db is created directly on the user’s machine. Like this, the user will not host or interact with any other user’s data, but only their own.



DB Schema Diagram



App's UX Flow

4. Implementation and Execution

4.1 Development approach and team member roles

We embraced the core principles of the Agile methodology. For example, designing around the user, building small pieces of working code, regular group meetings for feedback and project alignment.

During week 1, we researched various Git workflows and decided to follow the **Git Feature Branch Workflow**, which delays merging our work into the main branch until it's complete. It is easy to understand, not only for our team members but also for other users, helping to minimise errors and conflicts, and easily track changes.

Please click here to see the original Github Workflow document:

[GitHub Workflow](#)

In order to divide the workload, we initially identified five key features of our project and assigned them to each team member. However, we were adaptable to support each other, and prepared to take on additional tasks depending on the project's priorities and team member availability.

Workflow Diagram:

```

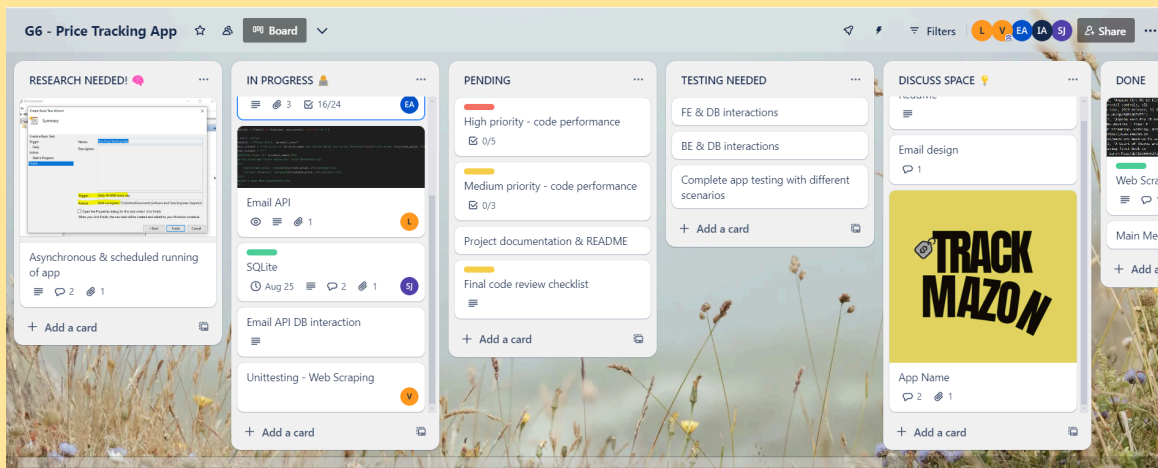
main
|
|_ dev
|   |
|   |_ feature/web-scraping
|   |
|   |_ feature/database-setup
|   |
|   |_ feature/user-interface
|   |
|   |_ feature/email-alerts
|   |
|   |_ test/web-scraping
  
```

TASK	TEAM MEMBER
Front End	Eva
Database	Configuration - Shaira Interactions - Shaira, Leyla, Eva & Violeta
Web Scraping	Violeta
Email API	Leyla
Data Visualization	Ikram
Testing	Unit testing - Eva (input validation), Violeta (web scraping), Leyla (email API) Mock website - Eva
Cron Job	Scheduling - Eva Code to run - Violeta & Leyla

4.2 Tools and libraries

Collaboration tools

- **Github.** [Click to see the app's Github repository](#)
- **Trello.** Our project management tool. It has greatly helped us to stay organised; we can see visually what each team member is working on and collaborate on brainstorming ideas.



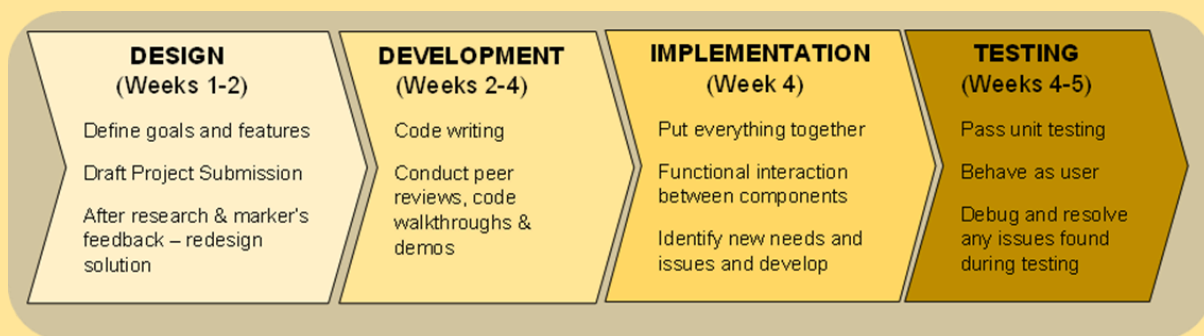
- **Slack.** Place to share updates, useful resources, video demos and pin links to all key project resources, such as Trello or Google Drive. We recurrently held 45-minute huddles before CFG classes from Monday to Thursday to share our progress, ideas and next steps.
- **Google Drive.** We kept a group Drive folder containing all relevant documents pertaining to the project, including architecture diagrams, and project documentations.

Python libraries

- ❖ **time.sleep** - Used to provide the user time to naturally read the text on the screen. Additionally, used on the web scraping side to avoid overwhelming the server or triggering anti-scraping mechanisms.
- ❖ **os.path** - Helpful to interact with the file system paths, such as checking if a file exists.
- ❖ **sys** - Combined with the above, this library helped us solve issues importing modules from files within the directories.
- ❖ **sqlite3** - Lightweight database solution within Python. Data persists across user sessions.
- ❖ **webbrowser** - Displays web-based documents to users. Used to direct users to the project README file in GitHub.
- ❖ **unittest** - Testing library used for writing and running tests to ensure the code's correct performance. It verifies all components are working as expected.

- ❖ **requests** - Allows sending HTTP requests to interact with web services. In our app, it's used to fetch the HTML content of product pages.
- ❖ **beautifulSoup** - Library for parsing HTML and XML documents. It's used in the app to extract product details from Amazon's website after fetching the HTML content.
- ❖ **datetime** - Provides classes for date manipulation, such as obtaining the current timestamp to keep track of pricing results timeline.
- ❖ **random** - Library used to get a random header in each web request to mimic human browsing behaviour.
- ❖ **mailjet_rest** - A client library for the MailJet API - used to send a price alert email notification to the user.

4.3 Implementation process



4.4 Development Challenges

During our first tries performing web scraping, we were constantly getting blocked by Amazon. To avoid this we implemented several methods to mimic human behaviour:

- Pass a comprehensive HTTP header with the web request.
- Within that header, set up a rotating User-Agent.
- Include a 5 second pause for every bunch of requests.
- Create error logs to easily debug and analyse in which cases we encountered blocks.

We also needed to consider timeout errors when incorrect urls were inputted by the user, as well as urls with format not accepted by Python's request module (not including 'http://', for example).

Whilst implementing the email API, we faced challenges such as subject lines exceeding MailJet's 255-character limit, which we resolved by creating a function to enforce this restriction. We also came up with a way to validate email addresses by developing a function that checks for the presence of "@" and "." symbols.

Developing and implementing the automation of the "Price-drop check" inherent to our app system and ideation was a challenging task. Primordially due to Windows path logic, the obligation of handling an external application (Task Scheduler) and CMD syntax. The latter raised an unavoidable hindrance for the normalisation of the executable: the presence of whitespace within the .exe's path. This is something we weren't able to resolve - thus becoming something that the user will need to keep in mind.

Additionally, due to unforeseen challenges and time constraints, the data visualisation aspect was not readily implemented as first intended. The complexities of integrating data visualisation with real-time updates and handling various technical issues significantly extended the time required for development.

Adding this functionality of data visualisations into existing code would have been challenging due to the need for seamless integration with multiple components, such as data retrieval, processing, and visualisation. Ensuring compatibility with the existing codebase and avoiding conflicts would have required significant testing and debugging, which time did not permit, thus leading to the decision not to implement this fully into the application.

5. Testing and Evaluation

5.1 Testing strategy

Unit testing for individual modules

We created unit testing code files for different modules, namely for the front-end, web scraping and email API codes. This allowed us to test them in different scenarios, ensuring that the modules work as intended without any problematic errors. Specifically, we made use of mocked objects in the web scraping and email API testing to replace external system components and also to reduce testing time (see image below).


```
.....
Ran 5 tests in 1.130s

OK
PS D:\Violeta\Documents\Software\Python\Project\src\tests\email_tests>
on.exe "d:/Violeta/Documents/Software/Python/Project/src/
.....
-----
Ran 5 tests in 0.005s

OK
```

Testing time before
applying mock objects
(1.130s) vs after applying
them (0.005s)

Mock website

We built a [mock website](#)  mimicking Amazon's HTML structure on which we changed the price of the product on a daily basis. This allowed us to test our application and see if the email notification was successfully triggered when the current price fell below the user's threshold price.

5.2 Functional and user testing

	Functional testing	User testing
User interface functionality	<ul style="list-style-type: none">• Test all CLI commands to see if they execute correctly• Check if invalid inputs are handled well with clear error messages• Ensure all menu options and navigation paths work as expected	<ul style="list-style-type: none">• Check if users can successfully perform tasks like adding a product, checking price history, and setting alerts• Ask users to enter edge cases to see if the application can interpret it• Receive feedback on UX flow and ease of CLI
Web scraping accuracy	<ul style="list-style-type: none">• Test web scraping on various Amazon product pages• Ensure that the product name, url, currency and price are all extracted properly• Error logging to keep track of issues during price checks• Check sale items to see if the current price is extracted, not presale price	<ul style="list-style-type: none">• Ask users to track a specific url and see if the correct information is extracted and entered into the database• Check with users if there are any discrepancies between the scraped data and what they see on the website• Review error logs• Enter incorrect urls to test behaviour
Email notification reliability	<ul style="list-style-type: none">• Test if the email is only sent when the price of the tracked product falls below the user's threshold price• Check the recipient's email to see the content and formatting of the email notification• Check handling of email sending failures - clear error messages should be displayed	<ul style="list-style-type: none">• Set up test scenarios - database includes a product where current price is lower than threshold price - check if user receives a price alert notification• Receive feedback on the format and usefulness of the content• Ask users to update their email preferences - toggling notifications ON/OFF for a specific product and disabling emails altogether
Database	<ul style="list-style-type: none">• Test CRUD for user_details,	<ul style="list-style-type: none">• Ask users to perform database

operations	product_details and price_history tables <ul style="list-style-type: none"> • Test error handling for database connection • Check execution of database queries 	operation tasks e.g. adding username and email address, adding a product, deleting a product <ul style="list-style-type: none"> • Ask user to review the database to see if it matches their input
-------------------	---	---

5.3 System limitations

Cron job

X Limitation: The cron job is scheduled to run daily, but as per workings of Windows system, this subprocess will only run if the machine is plugged in - so there is a potential for missed price changes between intervals.

✓ Improvement: Host the app in a server or cloud where it would be scheduled to run every 15 minutes without relying on the user's computer being on or interfering with their computing resources.

Web scraping

X Limitation: Performs web scraping only on Amazon product pages

✓ Improvement: If we had more time, we could have worked on scraping various global e-commerce platforms as well, specifically targeting other big e-commerce marketplaces such as eBay and Walmart. This could be easily done by adding subclasses of *WebScraping* (following the same structure as *AmazonWebScraper* class).

Email API

X Limitation: Users need to manually replace the API keys with their own keys; they would need to register an account with MailJet to access these credentials.

✓ Improvement: If we had more time, we could have worked on hiding and encrypting the API keys to improve security and user experience.

Executable

X Limitation: The executable file is only compatible with Windows operating systems; users with Mac or Linux systems would need to run the application through the source code (*main.py*).

✓ Improvement: If we had more time, we would have developed a universal executable system that allows users across all platforms to easily run our application.

6. Conclusion

Our team has successfully created a functional solution and helped consumers regain agency and trust of their online shopping experience. We have been able to achieve this by adopting an Agile development approach, efficiently managing data, as well as integrating Object Oriented Programming concepts into our code i.e. encapsulation in classes, creating multiple python codes to achieve modularity and inheritance to extend functionality and reuse code. Additionally, we frequently tested individual modules and overall integration to ensure the product works as intended

Overall, we feel that we have achieved our goal with the project, and were able to keep our aims and objectives in mind whilst making our decisions. We were happy with our choices of web scraping and email API tools, and using SQLite to create our database. We collaborated well as a team, communicating our progress and challenges in our frequent group calls, and were able to make a functional price tracking application as a result.