Group 5 Project Document

Project idea: Beauty Haul Generator

Team members:
- Tiffany Scott-Vaughan
- Ellen Daly
- Laura Wheaton
- Ekum Jaswal
- Lauren Blayney

CONTENTS

INTRODUCTION

The beauty industry offers an overwhelming amount of choice when it comes to skincare and makeup products, making it difficult for consumers to find the best options suited to their unique needs — from skin type and budget to ethical considerations.

We aim to develop a personalised **Beauty Haul Generator** — a Python-based console application that simplifies the decision-making process by recommending a curated set of beauty products tailored to a user's preferences. The app seeks to reduce decision fatigue, enhance accessibility to relevant products, and support conscious consumerism (e.g., vegan, eco-friendly products).

What are we building?
A custom recommendation engine that:

- Collects user preferences (skin type, product types, concerns, budget.).
- Fetches product data via a public beauty API.
- Scores products based on how well they match the user's preferences.
- Returns a final routine ("haul") which includes one product per selected type.

BACKGROUND

The problem?
Many consumers struggle to navigate the saturated beauty market. Most online retailers offer basic filtering but lack deep, personalised recommendation logic. This leads to user frustration, indecision, and often unsatisfactory purchases.

Our value proposition:
1. **Personalisation:** A lot of skincare and makeup recommendations are generic, yet every person's skin type and desires vary. Our project will give a solution tailored to the individual.
2. **Time saving:** Our project will give users instant recommendations, cutting out the time spent researching individual products and reducing decision fatigue.
3. **Inclusive:** The beauty industry is vast and many users struggle to find the right products for their personal skin concerns. Our project will help them to discover the right products for their unique needs.
4. **Budget friendly:** Beauty products vary significantly in price, from high-street to high-end. Our tool will allow users to set a budget and receive suggestions within their price range.

SPECIFICATIONS AND DESIGN

User Flow

1. User starts the app via the terminal.
2. App prompts user for their personal preferences.
3. App fetches and filters data from the Makeup API.
4. Recommendations are scored and formatted into a haul.
5. Routine is displayed in the terminal.
6. Results are saved to a lightweight database (CSV) at the user's request, or they can begin again.

Key Features

- **Input Parameters:**
  - Skin type
  - Product type

    - Budget range
    - Ethical filters (vegan, eco-friendly, natural ingredients)

- **Filtering & Scoring:**
    - Each product is scored based on preference matching.
    - Final routine selects top-scoring products by type.

- **Output:**
    - One tailored product per category.
    - Displayed neatly with formatted descriptions.

Tech stack:

Language: Python (using Object-Oriented Programming principles)

Frontend: Terminal (Console UI)
The frontend of our application is a text-based interface that runs in the terminal, providing a simple yet effective way for users to interact with the system. It is responsible for guiding the user through the process of generating their personalised beauty haul and includes the following key functionality:

- **Welcoming the user**: Presents a clear and friendly introduction to the app and what it offers.
- **Collecting user preferences**: Prompts the user to input their skin type, product types of interest, budget, and any ethical preferences (e.g. vegan-friendly, natural ingredients).
- **Formatting and displaying results**: Outputs the recommended product routine to the user in a clean and readable format, including product names and descriptions.
- **Ending the program**: Offers the user a clean exit, asking whether they'd like to save their results or restart the process.

Backend: External API integration (https://makeup-api.herokuapp.com/)
The backend is responsible for orchestrating the logic that powers the beauty haul generator. It begins by receiving user-generated inputs from the frontend (skin type, budget, product types, and ethical preferences). Using this data, it dynamically constructs and sends API requests to the external makeup API. Once the API returns product data, the backend processes this information by filtering and scoring each product based on how closely it aligns with the user's preferences. It then compiles a personalised beauty 'haul'—one product per selected category—and sends the final list back to the frontend for display. This architecture allows for flexible filtering, personalised recommendations, and a seamless user experience.

Database:
For our MVP, we implemented a lightweight database solution using a CSV file, which is managed programmatically via Python. This approach allowed us to persist user-generated data without the overhead of integrating a full database system, making development faster and more straightforward for the scope of this project.

Storing generated routines allows us to lay the foundation for future features (which are outlined on page 8). While CSV lacks the features of a full relational or NoSQL database (e.g. indexing, concurrency control), it was a deliberate and practical choice for our MVP.

Tools and libraries

We made use of a wide range of Python's built in libraries, including:

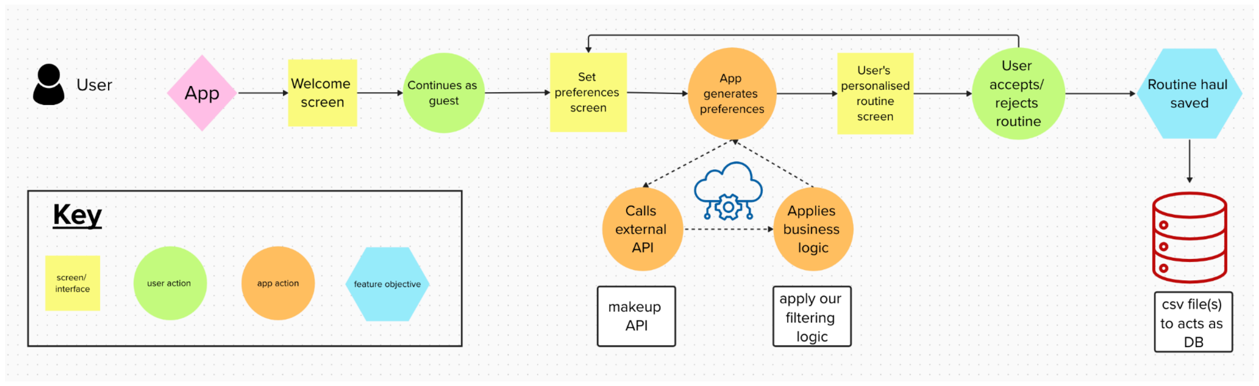
- `unittest`: Used for writing and running tests to verify that our functions behave as expected.
- `time`: Used for managing delays, which controlled the speed of the spinner UI.
- `multiprocessing`: Enables concurrent execution, allowing the spinner to run independently while API calls are processed.
- `csv`: Used to write to our CSV file, serving as a lightweight database.
- `os`: Helps to construct file paths and locate files (like our CSV "database") in a cross-platform way.
- `re`: Using regex matching to identify and remove unexpected whitespace and line breaks in product descriptions, ensuring data is displayed as expected in the terminal output to the user.

We also installed the following external library:

- `requests`: Used to send HTTP requests, enabling interaction with our chosen external API (i.e., retrieving responses).

Architecture Diagram:

The below diagram demonstrates the designed architecture and user flow through our app:



(Outline of user flow through the minimum viable product (MVP) beauty haul application).

IMPLEMENTATION AND EXECUTION

Team member roles:

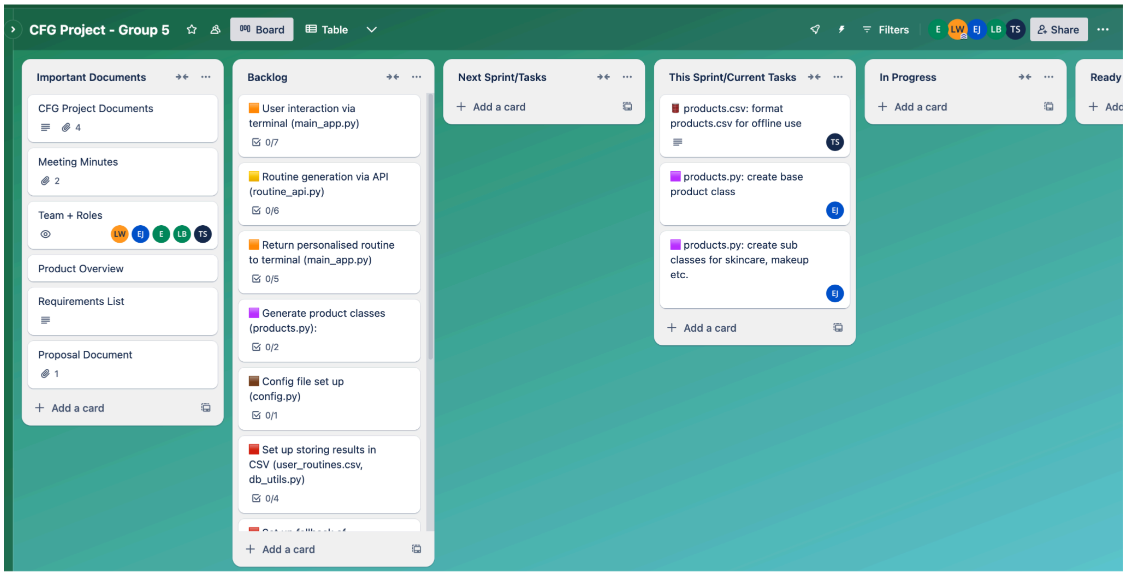
We began by each having a specific area of the code to focus on, as per the below table. As the project progressed, all team members contributed to the refinement and debugging of the codebase.

Team Members	**Allocated Roles**
Tiffany Scott-Vaughan	Error handling, Readme
Ellen Daly	User inputs and validation, and advanced filtering

Laura Wheaton	Formatting and displaying routine, saving to DB
Ekum Jaswal	Products, API and scoring system
Lauren Blayney	User preferences

Project management:
We adopted an agile development approach because our product idea was well-suited to iterative development. Taking an agile approach enabled us to plan what should be incorporated to produce our MVP while maintaining a list of features to add incrementally, ensuring continuous improvement and adaptation during the development process.

We used Trello to manage tasks and track our progress, following a scrum format. Tasks were broken down by application requirement, ensuring that code reviews were manageable and pull requests (PRs) were submitted regularly. We each reviewed other developers' PRs in accordance with our availability on a given day/week.



Important Documents
- CFG Project Documents — 4
- Meeting Minutes — 2
- Team + Roles — LW, EJ, E, LB, TS
- Product Overview
- Requirements List
- Proposal Document — 1
- Add a card

Backlog
- 🟧 User interaction via terminal (main_app.py) — 0/7
- 🟨 Routine generation via API (routine_api.py) — 0/6
- 🟧 Return personalised routine to terminal (main_app.py) — 0/5
- 🟪 Generate product classes (products.py): — 0/2
- 🟫 Config file set up (config.py) — 0/1
- 🟥 Set up storing results in CSV (user_routines.csv, db_utils.py) — 0/4
- Add a card

Next Sprint/Tasks
- Add a card

This Sprint/Current Tasks
- products.csv: format products.csv for offline use — TS
- 🟪 products.py: create base product class — EJ
- 🟪 products.py: create sub classes for skincare, makeup etc. — EJ
- Add a card

In Progress
- Add a card

Ready

Development Methodology

We used the Agile methodology, working in short iterative cycles with regular check-ins. Our Trello board (scrum format) helped divide tasks into manageable sprints, allowing us to:

- Set clear MVP goals
- Incorporate regular code reviews
- Implement features incrementally
- Adapt based on blockers and feedback

Achievements

- **Functional MVP delivered on time**: We successfully met our minimum viable product goals, which included collecting user preferences, fetching and filtering product data via an external API, and presenting tailored product routines in the terminal.

- **Modular, maintainable codebase**: We followed object-oriented programming principles, allowing clear separation of concerns (e.g., input handling, API interaction, data storage).
- **Collaborative version control**: Our team used GitHub effectively for managing code, resolving merge conflicts, and performing code reviews via pull requests.
- **Reusable filtering and scoring logic**: Built a flexible system that can easily be expanded with more product attributes or new filters in future development stages.

Implementation Challenges

- **API data limitations**: The free Makeup API included broken links, outdated data, and inconsistent tag use. Some parameters couldn't be combined in a single API call, which required us to write extra logic to get the data we wanted.

- **Data formatting issues**: Inconsistent formatting in API responses (e.g., whitespace and empty fields) meant we had to preprocess the data using regex and string manipulation to improve the terminal display.

Decisions to Change or Adapt

- **Shifted from product links to plain-text descriptions**: Due to broken image URLs and dead product pages, we focused on clear, informative descriptions rather than relying on links or visuals.
- **Moved from single API call to multiple batch calls**: Some filters could not be combined in query strings and some combinations of filters returned zero results, so we adjusted our logic to batch requests by category and then aggregate and score the results.
- **Used CSV instead of cloud database**: For simplicity and faster development, we used a lightweight CSV file as our temporary database. This decision allowed us to focus on logic and performance, while keeping room to scale to a real database if we were to develop this further in the future.

TESTING AND EVALUATION

Testing and Evaluation Strategy:

Unit tests were written using `unittest` for:

- Validating user input.
- Filtering logic (use MagicMock to cleanly mock products).
- Product class methods
- Database utilities

Console output was functionally tested by manually running the app in the terminal, inputting different combinations of user preferences, and checking whether:

- The correct prompts appeared
- The expected product haul was generated
- Filtering and scoring logic behaved correctly

- Errors or invalid inputs were handled gracefully
- The haul was displayed in a clean, readable format

System limitations:
- No GUI — limited to terminal interface
- Free public API with incomplete/inconsistent data
- Product links/images not always functional
- Static CSV storage (non-dynamic unless maintained)

FUTURE DEVELOPMENT

Planned feature roadmap:

- **User Login System:** Enable users to save/update preferences and revisit routines.
- **API Data Caching:** Improve performance and reduce API calls by caching API data**.**
- **DB Caching:** Return past recommendations when identical user inputs are detected.
- **Refined Logic:** Allow product rejection and regeneration of specific items, not the whole haul.
- **Enhanced UI/UX:** Potential GUI or web-based interface in future phases.
- **Migration to a full database** (e.g., MySQL, PostgreSQL, or Firebase), enabling more complex queries, user authentication, and integration with a future web frontend.
- **Incorporate reviews to make our filtering logic smarter:** Analyse product reviews for sentiment. Boost relevance scores for positive reviews aligned with a user's skin type and penalise products with consistently negative feedback.
    - This would involve scraping or importing review data and performing text classification or sentiment analysis using tools like NLTK, spaCy, or TextBlob.

CONCLUSION

The Beauty Haul Generator addresses a clear user pain point by providing curated, personalised beauty product recommendations in real-time. Our MVP successfully integrates multiple components — API handling, user input validation, and backend storage — into a cohesive, functional app. With a solid base in place, we are well-positioned to scale this project further.

Through collaborative, agile working and applying core software engineering principles, we’ve taken a conceptual idea and delivered a tangible, real-world application.

APPENDIX

Team SWOT Analysis:

Strengths	Weaknesses
- Strong UX focus and use of agile methodology. - Experience with JavaScript/TypeScript aids Python transition. - Proactive and clear team communication. - Real-world relevance: solves a common problem in beauty product selection. - Shared responsibilities enabled better collaboration across frontend, backend, and data handling. - Scalable foundation: can migrate from CSV to a full database or real-time API integration.	- Learning curve with new skills (e.g., OOP, decorators, testing). - CSV database requires manual updates; lacks real-time data.

Opportunities	Threats
- Develop and apply OOP in a real-world context. - Strengthen project management and collaborative development skills. - Gain hands-on experience with agile, GitHub workflows, and API integration. - Learn end-to-end product development, from concept to deployment.	- API limitations: outdated or broken data (e.g., links/images). - Team time zone and schedule conflicts. Mitigated via recorded meetings and flexible scheduling. - Time constraints; agile approach helps focus efforts on MVP first. - Technical issues (e.g., bugs in decorators or recursion logic) could slow progress. - Limited user profiles during testing may reduce the accuracy or diversity of feedback.