DISCORD BOT AUTOMATION ASSISTANT

by

OGUZ KAAN YILDIRIM

M.S. Harrisburg University of Science and Technology,

USA, 2024

A Proposal submitted in partial fulfillment of the requirements

for the degree of Master of Science

in the Computer Information Sciences at the

Harrisburg University of Science and Technology, Pennsylvania

Late Spring Term 2024

# ABSTRACT

This report documents the design, implementation, and testing of the Discord Bot Automation Assistant, a tool developed to automate the process of checking availabilities or prices without user intervention. The bot is designed to notify users or save relevant data when specific conditions are met, providing seamless automation within the Discord environment.

The project follows a structured approach beginning with system architecture, moving through detailed class design, and culminating in extensive unit testing. The system architecture is built on object-oriented principles, with boundary, control, and entity objects working together to manage command processing, browser interactions, and data handling. The data management strategy utilizes Excel and HTML formats for logging and reporting, ensuring that users can access and manipulate the data in familiar and user-friendly formats.

A significant part of the project focuses on the bot's ability to operate unattended, continuously monitoring product availability and prices, and either alerting the user via notifications or exporting data for future reference. The design incorporates an asynchronous approach to ensure real-time responsiveness and efficient handling of web scraping tasks.

Testing plays a critical role in validating the bot’s core functionality, especially for asynchronous tasks and browser automation. The testing framework leverages pytest and pytest-asyncio to verify the bot’s operations in various scenarios. Through unit testing and mocking of external dependencies, the system's behavior is thoroughly tested in isolation.

The report also includes an analysis of the defects identified during the testing phase, documenting specific errors related to asynchronous handling, command processing, and data logging. These defects were systematically resolved, leading to enhanced system stability and performance.

In conclusion, this project demonstrates the successful development of a robust automation system capable of monitoring product availability and prices in an unattended manner, notifying users, or saving data as needed. Future work will focus on expanding the bot's capabilities and optimizing the testing strategies for real-world scenarios.

# ACKNOWLEDGMENTS

I would like to take this opportunity to thank my Professor, Dr. Abdel Ejnioui, for his assistance and contribution.

I would like to express my sincere gratitude to my faculty and my university for their invaluable guidance and support. I am deeply appreciative of my friends for their encouragement, and I extend heartfelt thanks to my brother and family for their unwavering support throughout this journey.

# TABLE OF CONTENTS

[ABSTRACT i](#_Toc179399382)

[ACKNOWLEDGMENTS iii](#_Toc179399383)

[TABLE OF CONTENTS iv](#_Toc179399384)

[LIST OF FIGURES vii](#_Toc179399385)

[LIST OF TABLES viii](#_Toc179399386)

[LIST OF ACRONYMS/ABBREVIATIONS ix](#_Toc179399387)

[CHAPTER ONE: INTRODUCTION 1](#_Toc179399388)

[1.1 Goals And Objectives 1](#_Toc179399389)

[1.2 Motivation Of the Project 1](#_Toc179399390)

[1.2.1 Saving Time and Reducing Stress 2](#_Toc179399391)

[1.2.2 Financial Savings 3](#_Toc179399392)

[1.3 Context and Relevance of Application 3](#_Toc179399393)

[1.3.1 General Features of E-commerce Price Monitoring Tools 4](#_Toc179399394)

[1.3.2 Role of Automation in Service Booking and Availability Checking 4](#_Toc179399395)

[1.3.3 Technological Integration and Advancements 5](#_Toc179399396)

[1.3.4 Future Prospects and Impact on User Experience 5](#_Toc179399397)

[1.4 Benefits for Users 6](#_Toc179399398)

[1.4.1 Time Efficiency 6](#_Toc179399399)

[1.4.2 Financial Savings 6](#_Toc179399400)

[1.4.3 Reduced Stress 6](#_Toc179399401)

[1.4.4 Enhanced User Experience 7](#_Toc179399402)

[1.5 General Context of the Project 7](#_Toc179399403)

[1.5.1 Technological Advancements 7](#_Toc179399404)

[1.5.2 Industry Trends 7](#_Toc179399405)

[1.5.3 Market Impact 7](#_Toc179399406)

[1.5.4 Societal Implications 8](#_Toc179399407)

[1.6 Summary and Thesis Outline 8](#_Toc179399408)

[CHAPTER TWO: RELATED WORK 9](#_Toc179399409)

[2.1 Review of Existing Systems 9](#_Toc179399411)

[2.2 Comparison of Features 12](#_Toc179399412)

[2.3 Advances and Limitations 13](#_Toc179399413)

[2.4 Conclusion 14](#_Toc179399414)

[CHAPTER THREE: SYSTEM DESING AND IMPLEMENTATION 15](#_Toc179399415)

[3.1 Project Requirements 16](#_Toc179399417)

[3.1.1 Project Help (!project\_help) 16](#_Toc179399418)

[3.1.2 Navigate to Website (!navigate\_to\_website) 17](#_Toc179399419)

[3.1.3 Close Browser (!close\_browser) 18](#_Toc179399420)

[3.1.4 Login to a Website (!login) 18](#_Toc179399421)

[3.1.5 Receive Email (!receive\_email) 19](#_Toc179399422)

[3.1.6 Get Price (!get\_price) 19](#_Toc179399423)

[3.1.7 Start Monitoring Price (!start\_monitoring\_price) 20](#_Toc179399424)

[3.1.8 Stop Monitoring Price (!stop\_monitoring\_price) 21](#_Toc179399425)

[3.1.9 Check Availability (!check\_availability) 21](#_Toc179399426)

[3.1.10 Start Monitoring Availability (!start\_monitoring\_availability) 22](#_Toc179399427)

[3.1.11 Stop Monitoring Availability (!stop\_monitoring\_availability) 22](#_Toc179399428)

[3.2 Architecture 24](#_Toc179399429)

[3.2.1 Entity Objects 24](#_Toc179399430)

[AvailabilityEntity 24](#_Toc179399431)

[BrowserEntity 26](#_Toc179399432)

[DataExportEntity 26](#_Toc179399433)

[EmailEntity 26](#_Toc179399434)

[PriceEntity 27](#_Toc179399435)

[3.2.2 Boundary Objects 27](#_Toc179399436)

[project\_help\_boundary 27](#_Toc179399437)

[receive\_email\_boundary 27](#_Toc179399438)

[close\_browser\_boundary 28](#_Toc179399439)

[login\_boundary 28](#_Toc179399440)

[navigate\_to\_website\_boundary 28](#_Toc179399441)

[check\_availability\_boundary 28](#_Toc179399442)

[start\_monitoring\_availability\_boundary 28](#_Toc179399443)

[stop\_monitoring\_availability\_boundary 28](#_Toc179399444)

[get\_price\_boundary 29](#_Toc179399445)

[start\_monitoring\_price\_boundary 29](#_Toc179399446)

[stop\_monitoring\_price\_boundary 29](#_Toc179399447)

[3.2.3 Control Objects 29](#_Toc179399448)

[project\_help \_control 29](#_Toc179399449)

[receive\_email\_control 29](#_Toc179399450)

[navigate\_to\_website\_control 30](#_Toc179399451)

[login\_control 30](#_Toc179399452)

[close\_browser\_control 30](#_Toc179399453)

[check\_availability\_control 30](#_Toc179399454)

[start\_monitoring\_availability\_control 30](#_Toc179399455)

[stop\_monitoring\_availability\_control 30](#_Toc179399456)

[get\_price\_control 31](#_Toc179399457)

[start\_monitoring\_price \_control 31](#_Toc179399458)

[stop\_monitoring\_price\_control 31](#_Toc179399459)

[3.2.4 Data Access Layer Objects 31](#_Toc179399460)

[AvailabilityDAO 31](#_Toc179399461)

[PriceDAO 32](#_Toc179399462)

[DataExportDAO 32](#_Toc179399463)

[TokenConfigDAO 32](#_Toc179399464)

[EmailConfigDAO 32](#_Toc179399465)

[3.2.5 Associations Among Objects 33](#_Toc179399466)

[3.2.6 Aggregates Among Objects 33](#_Toc179399467)

[Availability Aggregate 33](#_Toc179399468)

[Price Aggregate 34](#_Toc179399469)

[Email Aggregate 34](#_Toc179399470)

[Browser Automation Aggregate 35](#_Toc179399471)

[3.2.7 Attributes for Each Object 35](#_Toc179399472)

[3.3 Design 37](#_Toc179399473)

[3.3.1 Authentication Subsystem 37](#_Toc179399474)

[3.3.2 Notification Subsystem 37](#_Toc179399475)

[3.3.3 Browser Subsystem 38](#_Toc179399476)

[3.3.4 Monitoring Subsystem 38](#_Toc179399477)

[3.3.5 Data Handling Subsystem 38](#_Toc179399478)

[3.4 Interface Specification 40](#_Toc179399479)

[3.5 Mapping Contracts to Exceptions 41](#_Toc179399480)

[3.6 Data Management Strategy 43](#_Toc179399481)

[3.6.1 Data Presentation and Usability 43](#_Toc179399482)

[3.6.2 Performance and Flexibility 43](#_Toc179399483)

[3.6.3 Rapid Deployment and Maintenance-Free Operation 44](#_Toc179399484)

[3.6.4 Simplified Data Handling and Security 45](#_Toc179399485)

[3.7 Technology Stack and Framework 45](#_Toc179399486)

[3.7.1 Programming Languages and Frameworks 46](#_Toc179399487)

[*Python* 46](#_Toc179399488)

[*Selenium* 46](#_Toc179399489)

[*Discord.py* 46](#_Toc179399490)

[3.7.2 Tools and Platforms 46](#_Toc179399491)

[*Visual Studio Code* 46](#_Toc179399492)

[*Git* 47](#_Toc179399493)

[*GitHub* 47](#_Toc179399494)

[3.7.3 Data Management and Storage 47](#_Toc179399495)

[*Configuration Files* 47](#_Toc179399496)

[*JSON Files* 47](#_Toc179399497)

[*Excel and HTML* 48](#_Toc179399498)

[3.7.4 Testing Strategy 48](#_Toc179399499)

[3.8 Conclusion 49](#_Toc179399500)

[LIST OF REFERENCES 50](#_Toc179399501)

# LIST OF FIGURES

[Figure 1: Trends in online shopping and booking services [1]. 2](#_Toc179393024)

[Figure 2: Distribution of Time Spent Per Shopping [3]. 3](#_Toc179393025)

[Figure 3: Google Flight User Interface [19]. 10](#_Toc179393026)

[Figure 4: Keepa User Interface [21]. 11](#_Toc179393027)

[Figure 5: UML use case diagram. 23](#_Toc179393028)

[Figure 6: Architectural Diagram. 25](#_Toc179393029)

[Figure 7: UML Component Diagram. 39](#_Toc179393030)

[Figure 8: UML Class Diagram. 40](#_Toc179393031)

[Figure 9:Transient and Persistent Data Handling. 44](#_Toc179393032)

[Figure 10: System Architecture Diagram. 48](#_Toc179393033)

# LIST OF TABLES

[Table 1: Comparison of Key Features 13](#_Toc179392562)

[Table 2: Attributes for Each Object. 35](#_Toc179392563)

[Table 3: Contracts and Exceptions. 41](#_Toc179392564)

# LIST OF ACRONYMS/ABBREVIATIONS

API: Application Programming Interface

AS: Authentication Subsystem

DTO: Data Transfer Object

DAO: Data Access Object

EH: External Helpers

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

HTTPS: HyperText Transfer Protocol Secure

IDE: Integrated Development Environment

IIS: Interaction Interface Subsystem

NS: Notification Subsystem

PMS: Product Management Subsystem

SQL: Structured Query Language

SPAS: Save Price, Availability Subsystem

UML: Unified Modeling Language

URL: Uniform Resource Locator

# CHAPTER ONE: INTRODUCTION

This chapter introduces the PriceTracker project, outlining its goals and objectives, motivations, and the importance of the application classes to which it belongs. It also details the benefits for users and provides the general context of the project, including technological advancements, industry trends, market impact, and societal implications. Each section aims to give a comprehensive overview of the project's foundation, setting the stage for the detailed discussions in the subsequent chapters.

## Goals And Objectives

The primary goal of this project is to develop an automated Discord bot system, named PriceTracker, designed to monitor product prices and the availability of services. This project aims to provide users with timely notifications about price changes and available dates for desired products or services.

## Motivation Of the Project

In today's digital age, online shopping, booking services, and price comparison have become integral parts of daily life. Consumers frequently spend considerable time and effort to monitor prices and check the availability of products and services. This project's motivation stems from the need to streamline these activities and provide a more efficient, less stressful experience for users. It can be clearly seen in Figure 1 how much online shopping has increased over the years.

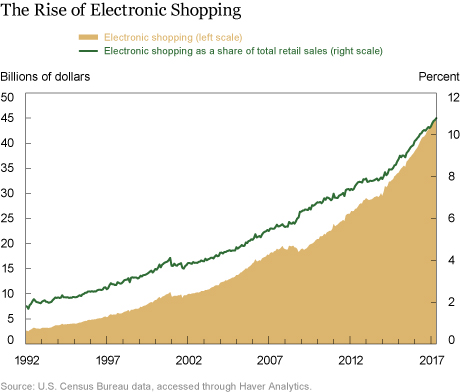


Figure 1: Trends in online shopping and booking services [1].

### Saving Time and Reducing Stress

One of the primary motivations for developing the PriceTracker bot is to save user’s time. The report indicates that UK adults spend an average of more than three-and-a-half hours online each day, engaging in various online activities, including shopping and price comparison [2]. This includes activities such as comparing prices across different websites, monitoring price fluctuations, and checking the availability of dates for bookings. By automating these tasks, the PriceTracker bot significantly reduces the time users need to spend on these activities. In Figure 2, we can see that people spend almost 15% of their time checking prices for the same product [3].

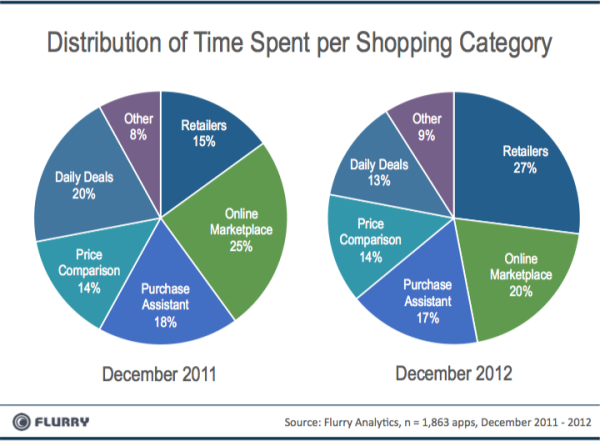


Figure 2: Distribution of Time Spent Per Shopping [3].

### Financial Savings

The average consumer spends approximately $1,200 annually on impulse purchases, often driven by price fluctuations and limited time offers. Another crucial motivation is the potential for financial savings. Product prices, plane tickets, and hotel rates can fluctuate significantly within short periods. For instance, a report by Hopper found that prices for airline tickets can change by an average of 20% per month due to factors like jet fuel prices and seasonal demand. Similarly, product prices on e-commerce platforms like Amazon can vary by up to 30% depending on the time and date. By receiving timely notifications about price drops and availability changes, users can capitalize on the best deals and avoid overpaying [4, 5]. The PriceTracker bot aims to address these issues by automating the tracking process and providing timely notifications.

## Context and Relevance of Application

The PriceTracker bot is part of a broader category of tools designed to enhance consumer decision-making in e-commerce and service booking. This section explores the general context of similar applications and highlights the unique aspects of the PriceTracker bot.

### General Features of E-commerce Price Monitoring Tools

Applications in the realm of e-commerce price monitoring typically provide functionalities that allow users to track the prices of products across various platforms. These tools enable consumers to:

* Monitor price fluctuations in real-time.
* Set alerts for price drops.
* Compare prices across different sellers to find the best deals.
* Receive notifications about price changes and promotional offers.

Research indicates that price tracking tools are increasingly popular among consumers due to the dynamic nature of online pricing, which can change based on factors such as demand, competition, and seasonal variations. According to a study by Statista, the global adoption of e-commerce tools that assist in price monitoring and comparison is expected to grow significantly over the next few years [6].

### Role of Automation in Service Booking and Availability Checking

Automation tools in the service booking industry are designed to help users efficiently manage their bookings and check availability for services such as flights, hotels, and car rentals. These tools typically offer features such as[7]:

* Automated searches for the best booking deals.
* Notifications about changes in availability.
* Integration with various booking platforms to streamline the user experience.
* Predictive analytics to suggest the best times to book.

The use of automation in this context is driven by the need to handle large volumes of data and provide timely information to users, reducing the manual effort required to find and secure the best deals[8]. Studies have shown that consumers appreciate the convenience and time savings provided by these automated tools[9].

### Technological Integration and Advancements

Technological advancements in web scraping, data analysis, and automated notifications have significantly improved the functionality of tools like the PriceTracker bot. Key technological features include:

* *Web Scraping*: This technology allows the bot to collect data from various websites, providing real-time updates on product prices and availability[10].
* *Data Analysis:* Advanced algorithms process the collected data to identify trends and generate meaningful insights for users[11].
* *Automated Notifications:* Users receive timely alerts through various communication channels, ensuring they are always informed about important changes[12].

These technological integrations not only enhance the efficiency and accuracy of such tools but also contribute to a seamless user experience. As technology continues to evolve, these tools are expected to become even more sophisticated, offering more advanced features and greater reliability.

### Future Prospects and Impact on User Experience

The ongoing development of price tracking and booking automation tools holds significant promise for improving user experiences in e-commerce and service booking[13]. Future enhancements might include:

* More accurate predictive analytics to forecast price changes.
* Enhanced integration with a wider range of platforms and services.
* Increased personalization based on user preferences and behaviors.

As these tools become more advanced, they will likely play a critical role in helping consumers make smarter purchasing decisions, save money, and reduce the stress associated with manual monitoring of prices and availability.

## Benefits for Users

### Time Efficiency

The PriceTracker bot significantly reduces the time users spend on monitoring prices and availability. Instead of manually checking multiple websites, users receive automated notifications about changes, allowing them to focus on other tasks. This efficiency is particularly beneficial for busy individuals who need to manage their time effectively.

### Financial Savings

By alerting users to price drops and availability changes, the bot helps them make cost-effective decisions. Users can purchase products at lower prices and book services at more favorable rates, resulting in substantial financial savings over time. McKinsey report indicates that consumers who use price tracking tools save an average of 10-15% on their purchases [14].

### Reduced Stress

The bot alleviates the stress associated with constantly monitoring prices and availability. Users no longer need to worry about missing out on deals or checking for updates repeatedly. The peace of mind provided by timely notifications allows users to relax and feel confident in their purchasing decisions.

### Enhanced User Experience

The PriceTracker bot enhances the overall user experience by providing a convenient and reliable service. Its integration with Discord ensures that users can easily interact with the bot, receive updates, and manage their preferences. The user-friendly design and automated functionality contribute to a seamless and enjoyable experience.

## General Context of the Project

### Technological Advancements

The development of the PriceTracker bot is rooted in automation. This technology provides users with accurate and timely information. As technology continues to evolve, the capabilities of the bot will also expand, offering even more sophisticated features and functionalities.

### Industry Trends

The e-commerce and travel industries are rapidly evolving, with increasing reliance on digital tools and automation. Consumers are becoming more tech-savvy and demand solutions that enhance their online experiences. According to [Statista, 2020], the number of digital buyers worldwide is expected to surpass 2.14 billion by 2021. The PriceTracker bot aligns with these trends, offering a tool that meets the needs of modern consumers [15].

### Market Impact

The market impact of the PriceTracker bot is significant, as it addresses a common pain point for consumers: the need to monitor prices and availability. By providing a reliable and efficient solution, the bot has the potential to attract a large user base and generate substantial value. The bot's ability to save time and money for users also contributes to its market appeal and competitiveness.

### Societal Implications

The PriceTracker bot contributes to the broader trend of digital automation, which has far-reaching implications for society. Automation tools like the bot simplify everyday tasks, making life more convenient and efficient for users. Additionally, the bot's ability to help users save money can have positive economic impacts, especially for budget-conscious consumers. As automation becomes more integrated into daily life, tools like the PriceTracker bot exemplify how technology can improve quality of life and drive innovation.

## Summary and Thesis Outline

In this chapter, we introduced the PriceTracker project by discussing its goals, objectives, and motivations. We explored the importance of the application classes to which the project belongs, highlighting the significant impact it can have in the e-commerce and travel industries. We also detailed the benefits for users, such as time efficiency, financial savings, and reduced stress. Furthermore, we provided the general context of the project, including technological advancements, industry trends, market impact, and societal implications. This foundational overview sets the stage for the following chapters, where we will delve deeper into related work, project design, implementation, and findings.

Chapter 2 will discuss and summarize previously proposed work related to the Discord Bot Automation Assistant. It will include a comparison of similar tools and technologies, highlighting their strengths and weaknesses in relation to this project.

# CHAPTER TWO: RELATED WORK

In this chapter, we review existing systems and projects that are comparable to the PriceTracker bot. By examining these systems, we aim to understand their features, strengths, and limitations, and how they compare to our project. This comparative analysis will help identify the unique contributions of PriceTracker and areas for potential improvement. We will focus on three key examples: Google Flights, Keepa, and a Discord Bot project on GitHub. The chapter concludes with a summary of the comparisons and insights gained from this review.



## Review of Existing Systems

The systems we will discuss include Google Flights, Keepa, and a GitHub-based Discord Bot project. These systems represent a range of applications from travel booking to e-commerce price tracking and open-source software development.

According to a report by Invoca, "45 Statistics Retail Marketers Need to Know in 2024," consumers increasingly rely on price tracking tools like Keepa to monitor product prices and make informed purchasing decisions [16]. Such systems help users save money and ensure they get the best deals available online. Similarly, a study by Saleslion reveals that "81% of Shoppers Conduct Research Before Purchase," highlighting the significance of tools like Google Flights in enabling users to compare flight prices and find the most cost-effective travel options [17].

We will examine the features, advantages, and limitations of each of these systems to provide a detailed comparison and analysis. This review will help us understand the current landscape of price tracking and comparison tools, setting the stage for a more in-depth discussion of the PriceTracker bot's unique value propositions in subsequent sections.

*Google Flights*

Google Flights is a travel fare aggregator that provides price comparisons for flights. It offers features such as price tracking, price history, and alerts for price changes. Users can search for flights, compare prices across different airlines, and receive notifications about fare changes [18].

A screenshot of a computer

Description automatically generated

Figure 3: Google Flight User Interface [19].

* Key Features:
  + Price tracking and alerts
  + Comprehensive search for flights
  + Historical price data
  + User-friendly interface
* Comparison to PriceTracker:
  + Google Flights focuses on the travel industry, specifically flights, whereas PriceTracker aims to support various product categories.
  + Both systems offer price tracking and notifications, but PriceTracker integrates directly with e-commerce platforms and uses web scraping for real-time data.

*Keepa*

Keepa is a price tracking tool specifically for Amazon products. It provides detailed price history charts, price drop alerts, and browser extensions for easy access. Keepa tracks prices and offers a comprehensive overview of product price trends [20].

A screenshot of a computer

Description automatically generated

Figure 4: Keepa User Interface [21].

* Key Features:
  + Price history charts
  + Price drop alerts
  + Browser extensions
  + Multi-region price tracking
* Comparison to PriceTracker:
  + Keepa is limited to Amazon products, while PriceTracker aims to support multiple e-commerce platforms.
  + Both tools provide price tracking and notifications, but PriceTracker's broader scope allows for a wider range of product monitoring.

*Discord Bot (GitHub Project)*

This GitHub project is an open-source Discord bot that can be customized for various functionalities. It provides a foundation for building bots that can interact with users on Discord, perform automated tasks, and integrate with other APIs.

* Key Features:
  + Customizable bot functionality
  + Interaction with users on Discord
  + Integration with external APIs
  + Open-source and community-driven
* Comparison to PriceTracker:
  + The GitHub project serves as a foundation for building custom bots, like PriceTracker's use of Discord for notifications and interactions.
  + PriceTracker's specific focus on price tracking and product availability differentiates it from the more general-purpose nature of the GitHub project.

## Comparison of Features

*Price Tracking and Alerts*

Both Google Flights and Keepa provide robust price tracking and alert systems. Google Flights focuses on flights, while Keepa tracks Amazon product prices. PriceTracker combines these functionalities across multiple e-commerce platforms, offering users a versatile tool for tracking various product prices.

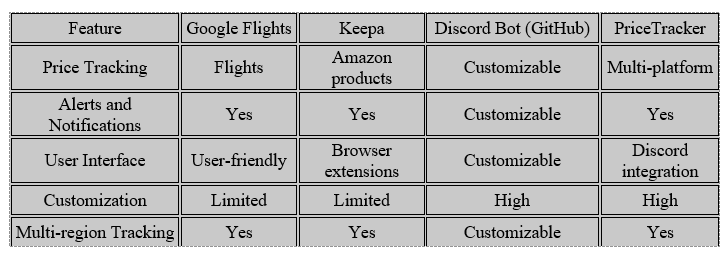
*User Interface and Usability*

Google Flights is known for its user-friendly interface and comprehensive search capabilities. Keepa provides detailed price charts and browser extensions for easy access. PriceTracker aims to provide a seamless user experience by integrating with Discord, allowing users to interact with the bot through a familiar platform.

*Scope and Customization*

The GitHub Discord Bot project offers a high level of customization, enabling developers to build bots for different purposes. PriceTracker leverages this flexibility to create a specialized bot for price tracking and notifications, integrating with multiple e-commerce platforms.

Table 1: Comparison of Key Features



## Advances and Limitations

*Advances in Price Tracking*

Google Flights and Keepa have advanced the field of price tracking with their specialized focus areas. Google Flights excels in flight fare aggregation, while Keepa provides detailed Amazon price histories. PriceTracker builds on these advances by offering a comprehensive solution that tracks prices across multiple platforms, leveraging web scraping and real-time data processing.

*Limitations*

Google Flights and Keepa are limited by their specific domains—flights and Amazon products, respectively. The GitHub Discord Bot project, while highly customizable, requires significant development effort to tailor it to specific needs. PriceTracker addresses these limitations by providing a ready-to-use solution that integrates multiple functionalities, though it may face challenges in ensuring data accuracy and handling diverse product categories.

## Conclusion

This chapter reviewed existing systems comparable to PriceTracker, including Google Flights, Keepa, and a GitHub Discord Bot project. We compared their features, highlighted advances and limitations, and identified areas where PriceTracker offers unique contributions. This analysis provides a foundation for understanding the competitive landscape and potential improvements for PriceTracker. In the next chapter, we will delve into the detailed design and implementation of the PriceTracker bot, building on the insights gained from this comparative review.

# CHAPTER THREE: SYSTEM DESING AND IMPLEMENTATION

This chapter provides a comprehensive overview of the design and implementation of the Discord bot system, with each section addressing critical components that support its operation. The chapter begins with an exploration of the *Project* *Requirements*, including detailed use cases and the functionality the bot must deliver to users, such as price checking, availability monitoring, and more.

Next, the *Architecture* section outlines the system’s overall structure through UML diagrams, showcasing the relationships between different components, including boundary, control, and entity objects. This section emphasizes how the system's design supports scalability and operational efficiency within the Discord environment.

In the *Design* section, the chapter delves deeper into the system’s internal architecture, presenting class diagrams and descriptions of how the objects within the system interact with each other. Special attention is given to key subsystems such as authentication, notification, and data handling.

The *Interface* *Specification* section details the interfaces that facilitate communication between the bot and its users, outlining how the system processes commands and responds to requests. This section focuses on the user-facing aspects of the bot.

A critical aspect of the design is covered in the *Mapping* *Contracts* to *Exceptions* section, where the chapter outlines how specific contracts within the bot are associated with exception handling. This ensures that the system can handle errors gracefully, maintaining stability during its operations.

The *Data* *Management* *Strategy* section highlights the decision to utilize file-based storage systems, such as Excel and HTML, rather than traditional databases. The reasons for this approach, including performance, flexibility, and ease of use, are explained in detail, with an emphasis on how it aligns with the bot’s real-time, non-transactional nature.

Finally, the *Technology* *Stack* *and* *Framework* section covers the tools, programming languages, and frameworks used to build and deploy the bot. This includes details on the use of Python, Selenium, Discord.py, and GitHub, along with the integration of testing strategies to ensure the bot’s robustness.

The chapter concludes by summarizing the key design decisions and their alignment with the project’s requirements, preparing for further discussions on testing strategies and implementation details.



## Project Requirements

In this section, we will cover the project requirements, including the use case diagram and detailed descriptions of the use cases. We will also integrate relevant parts from assignments to provide a comprehensive understanding.

### Project Help (!project\_help)

* Actor: User
* Description: Provides the user with a list of available commands and descriptions on how to use them.
* Preconditions: Bot must be operational and accessible to the user.
* Trigger: User sends the "!project\_help" command.
* Main Flow:
  1. User requests help by sending "!project\_help".
  2. Bot receives the command and fetches a list of all usable commands along with descriptions.
  3. Bot displays the command list to the user.
* Postconditions: User receives the information needed to utilize the bot effectively.

### Navigate to Website (!navigate\_to\_website)

* Actor: User
* Description: Enables the user to command the bot to open a web browser and navigate to a specified URL.
* Preconditions: Bot must be operational.
* Trigger: User sends the "!navigate\_to\_website [URL]" command.
* Main Flow:
  1. User inputs the command with a URL.
  2. Bot recognizes the command and extracts the URL.
  3. Bot launches the web browser and navigates to the specified URL.
  4. Bot confirms navigation success to the user.
* Postconditions: The browser has opened at the desired web page.

### Close Browser (!close\_browser)

* Actor: User
* Description: Allows the user to send a command to the bot to close the currently opened web browser.
* Preconditions: A web browser must be opened by the bot.
* Trigger: User sends the "!close\_browser" command.
* Main Flow:
  1. User sends the command to close the browser.
  2. Bot receives the command and proceeds to close any open browsers.
  3. Bot confirms the closure of the browser.
* Postconditions: Any browser opened by the bot is closed.

### Login to a Website (!login)

* Actor: User
* Description: Enables the user to command the bot to log into a web application using provided credentials.
* Preconditions: The target website's login page is accessible.
* Trigger: User sends the "!login [website] [username] [password]" command.
* Main Flow:
  1. User inputs the command with website URL, username, and password.
  2. Bot recognizes the command, extracts the details, and navigates to the login page of the website.
  3. Bot inputs the credentials and attempts to log in.
  4. Bot confirms to the user whether the login was successful or if there were any errors.
* Postconditions: User is logged into the website if credentials are correct and the website is reachable.

### Receive Email (!receive\_email)

* Actor: User
* Description: Commands the bot to send an email with an attached file specified by the user.
* Preconditions: Bot must be operational, and the specified file must be present in the system.
* Trigger: User sends the "!receive\_email [file\_name]" command with a valid file name.
* Main Flow:
  1. User inputs the command with the name of the file to be emailed (e.g., "!receive\_email fileToEmail.html").
  2. Bot recognizes the command and verifies the presence of the file in the system.
  3. Bot attaches the file to an email and sends it to a predetermined recipient.
  4. Bot confirms to the user that the email has been sent successfully or informs them of any issues encountered (e.g., file not found or email delivery failure).
* Postconditions: The email is sent with the specified attachment if all conditions are met.

### Get Price (!get\_price)

* Actor: User
* Description: Retrieves the current price of a product from a specified URL and logs this information to an Excel or HTML file.
* Preconditions: Bot must be operational, and the URL must be accessible.
* Trigger: User sends the "!get\_price [URL]" command.
* Main Flow:
  1. User sends a command with the URL of the product.
  2. Bot recognizes the command, retrieves the current price from the specified URL using web scraping.
  3. Bot logs the price retrieval event to an Excel and HTML file.
  4. Bot displays the price to the user.
* Postconditions: The price is displayed to the user and data is logged.

### Start Monitoring Price (!start\_monitoring\_price)

* Actor: User
* Description: Initiates an ongoing process to monitor price changes at a specified URL, alerting the user via email if there are price changes.
* Preconditions: Bot must be operational, and the URL must be accessible.
* Trigger: User sends the "!start\_monitoring\_price [URL] [frequency]" command.
* Main Flow:
  1. User specifies the URL and frequency of checks.
  2. Bot begins monitoring the price at the given URL at the specified frequency.
  3. For each check, the bot calls the "!get\_price" command to log the current price and check for changes.
  4. The bot sends the saved document as an email.
  5. Bot continues to monitor until the "!stop\_monitoring\_price" command is issued.
* Postconditions: Price monitoring is active, logs are being created at each interval, and emails are sent on price changes.

### Stop Monitoring Price (!stop\_monitoring\_price)

* Actor: User
* Description: Terminates an ongoing price monitoring process and provides a summary of the results.
* Preconditions: Price monitoring process must be active.
* Trigger: User sends the "!stop\_monitoring\_price" command.
* Main Flow:
  1. User sends the command to stop monitoring.
  2. Bot receives the command and terminates the ongoing price monitoring.
  3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
* Postconditions: Price monitoring is ceased, and final results are reported to the user.

### Check Availability (!check\_availability)

* Actor: User
* Description: Checks the availability of a reservation or booking at a specified URL and logs this information to an Excel or HTML file.
* Preconditions: Bot must be operational, and the URL must be accessible.
* Trigger: User sends the "!check\_availability [URL]" command.
* Main Flow:
  1. User sends a command with the URL where the availability needs to be checked.
  2. Bot recognizes the command, retrieves availability data from the specified URL using web scraping.
  3. Bot logs the availability check event to an Excel and HTML file.
  4. Bot displays the availability status to the user.
* Postconditions: The availability status is displayed to the user and data is logged.

### Start Monitoring Availability (!start\_monitoring\_availability)

* Actor: User
* Description: Initiates an ongoing process to monitor changes in availability at a specified URL, alerting the user via email if there are changes in availability.
* Preconditions: Bot must be operational, and the URL must be accessible.
* Trigger: User sends the "!start\_monitoring\_availability [URL] [frequency]" command.
* Main Flow:
  1. User specifies the URL and frequency of checks.
  2. Bot begins monitoring the availability at the given URL at the specified frequency.
  3. For each check, the bot calls the "!check\_availability" command to log the current availability and check for changes.
  4. If an availability change is detected, the bot sends an email with the updated availability information.
  5. Bot continues to monitor until the "!stop\_monitoring\_availability" command is issued.
* Postconditions: Availability monitoring is active, logs are being created at each interval, and emails are sent on availability changes.

### Stop Monitoring Availability (!stop\_monitoring\_availability)

* Actor: User
* Description: Terminates an ongoing availability monitoring process and provides a summary of the results.
* Preconditions: Availability monitoring process must be active.
* Trigger: User sends the "!stop\_monitoring\_availability" command.
* Main Flow:
  1. User sends the command to stop monitoring.
  2. Bot receives the command and terminates the ongoing availability monitoring.
  3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
* Postconditions: Availability monitoring is ceased, and results are reported to the user.

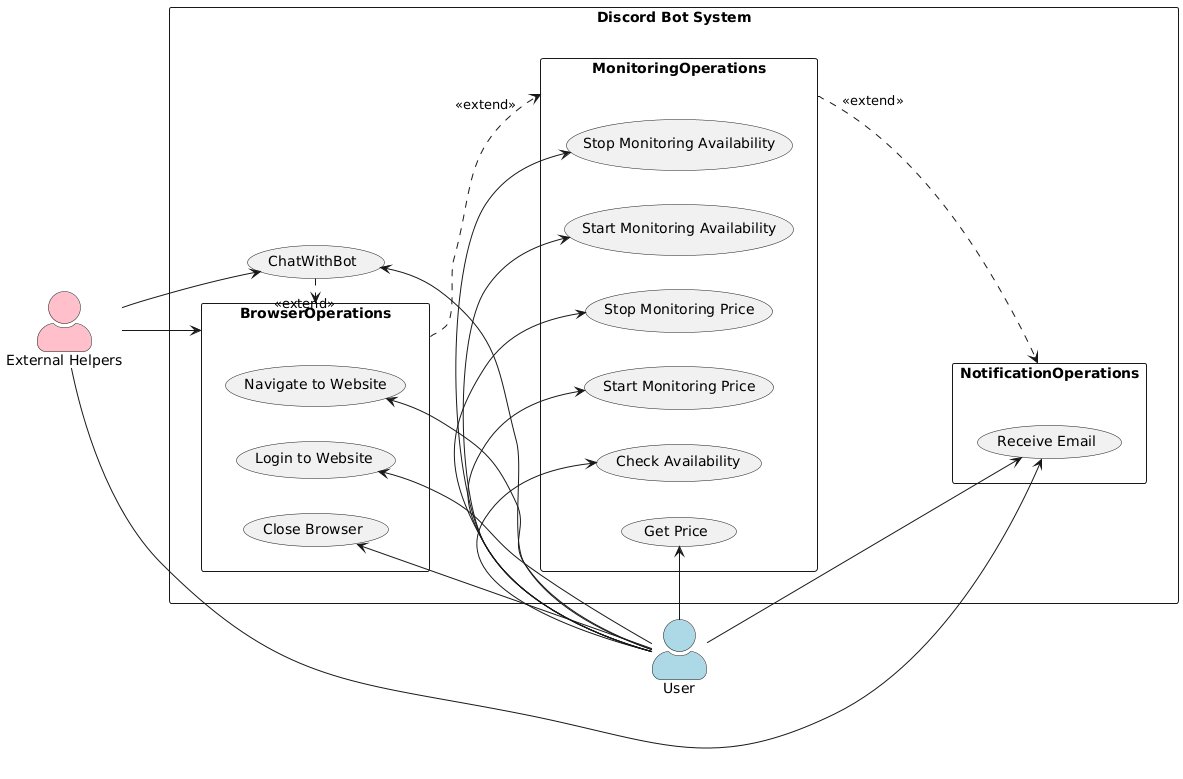


Figure 5: UML use case diagram.

## Architecture

The architecture of the Discord bot project forms the backbone of its functionality, providing a robust framework for managing interactions within the Discord environment. This section outlines the system's architectural design, explaining how it supports the operational requirements and enhances the bot's capabilities. By detailing the architectural components and their deployment, this section demonstrates the scalability, reliability, and efficiency of the system.

### Entity Objects

These entities act as the data manipulation layer of your architecture, directly interacting with the data sources and external systems to fetch, process, and store the required information. They provide a clean separation of concerns by encapsulating the logic needed to interact with data sources from the rest of the application, ensuring that the control objects can remain focused on

#### AvailabilityEntity

* + Purpose: Handles all data operations related to checking and monitoring availability. It directly interacts with external systems or databases to retrieve availability information.
  + Key Methods:
    - check\_availability: Connects to external services to check availability at the given URL on a specified date. It manages direct interactions with web APIs or databases to fetch availability data.
    - export\_data: Saves or logs availability data to local storage or a database. It might format the data for export to files such as Excel or HTML formats, which are then used for reporting or email notifications.

A screenshot of a computer screen

Description automatically generated

Figure 6: Architectural Diagram.

#### BrowserEntity

* + Purpose: Manages all operations that require direct interaction with a web browser, such as opening, navigating, or closing a browser. It encapsulates all functionalities that involve web automation tools like Selenium.
  + Key Methods:
    - launch\_browser: Opens a web browser session with predefined configurations.
    - navigate\_to\_website: Navigates to a specified URL within an open browser session.
    - close\_browser: Closes the currently open web browser session to free up resources.

#### DataExportEntity

* + Purpose: Responsible for exporting data into various formats for storage or transmission. This entity ensures data from operations like price checks or availability monitoring is logged appropriately.
  + Key Methods:
    - export\_to\_excel: Formats and writes data to an Excel file, organizing data into sheets and cells according to specified schemas.
    - export\_to\_html: Converts data into HTML format for easy web publication or email attachments.

#### EmailEntity

* + Purpose: Handles the configuration and process of sending emails. This entity works with email servers to facilitate the sending of notifications, alerts, or reports generated by the system.
  + Key Methods:
    - send\_email\_with\_attachments: Prepares and sends an email with specified attachments. It manages attachments, formats the email content, and interacts with email servers to deliver the message.

#### PriceEntity

* + Purpose: Specializes in fetching and monitoring price data from various online sources. It uses web scraping techniques to extract pricing information from web pages.
  + Key Methods:
    - get\_price: Retrieves the current price of a product from a specified URL. It scrapes the web page to find pricing information and returns it to the control layer.
    - export\_data: Similar to the AvailabilityEntity, it exports price data to various file formats for reporting or further analysis.

### Boundary Objects

Each boundary object is specifically designed to parse user commands received via Discord, extracting necessary data before interacting with the appropriate control objects to fulfill the user's requests.

#### project\_help\_boundary

Interprets the user's request for help, parses the command, and communicates with the bot control to retrieve and display a list of available commands along with their descriptions.

#### receive\_email\_boundary

Handles the command to send an email with an attached file, parses the user's message to determine the file to be attached, and coordinates with the control object to manage the email sending process.

#### close\_browser\_boundary

Processes the command to close the web browser, parses the message, and instructs the browser control to end the browser session.

#### login\_boundary

Manages the user's command to log into a website, parsing details like the website URL, username, and password before passing them to the browser control for the login operation.

#### navigate\_to\_website\_boundary

Captures and parses the user's command to navigate to a specific URL, then communicates with the browser control to perform the navigation.

#### check\_availability\_boundary

Receives and parses the user's message to extract necessary data such as the URL and date, then contacts the corresponding control object to check availability at the provided URL.

#### start\_monitoring\_availability\_boundary

Takes the user's input to begin monitoring availability at a specified URL with certain frequency parameters, parses the message, and forwards the data to the control layer to initiate monitoring.

#### stop\_monitoring\_availability\_boundary

Captures the command to cease monitoring availability, parses the user's instructions, and passes the command to the control object to stop the monitoring process.

#### get\_price\_boundary

Receives the command to retrieve a price from a specified URL, parses the command to extract the URL, and contacts the price control to obtain and return the price.

#### start\_monitoring\_price\_boundary

Receives the command to start monitoring the price at a specified URL and interval, parses the message for necessary details, and forwards these to the price control to begin the monitoring process.

#### stop\_monitoring\_price\_boundary

Processes the command to stop price monitoring, parses the user's instructions, and notifies the price control to end the monitoring and summarize the findings.

### Control Objects

Each control object acts as a decision-making hub that processes input from its corresponding boundary object, directs operations by interacting with entity objects or utilities (like logging or sending emails), and ultimately returns the outcome to the boundary object for user communication.

#### project\_help \_control

Generates and returns a list of all available commands and their descriptions, assisting the user in navigating the bot's functionalities.

#### receive\_email\_control

Manages the attachment and sending of an email with specified files, liaising with EmailEntity to perform the email operations.

#### navigate\_to\_website\_control

Checks if the URL is valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual action.

#### login\_control

Checks if the URL, username, and password are valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual login action.

#### close\_browser\_control

Checks if there is an open session, then contacts the BrowserEntity to close the browser.

#### check\_availability\_control

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the AvailabilityEntity to verify availability at a specified URL and date, retrieves the availability status, calls the entity's data export method to save data.

#### start\_monitoring\_availability\_control

Initiates a monitoring process at defined intervals by repeatedly calling the check\_availability method, handling the scheduling and continuation of this process, and calls the receive\_email method/control object after obtaining data.

#### stop\_monitoring\_availability\_control

Ends the monitoring process, summarizes the collected data, and returns the final status to the boundary object for user notification.

#### get\_price\_control

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the PriceEntity to fetch the price at a specified URL and calls the entity's data export method to save data.

#### start\_monitoring\_price \_control

Initiates a monitoring process at defined intervals by repeatedly calling the get\_price method, handling the scheduling and continuation of this process, and calls the receive\_email method/control object after obtaining data.

#### stop\_monitoring\_price\_control

Terminates the price monitoring process, summarizes the collected data, and communicates the results back to the boundary for user notification.

### Data Access Layer Objects

Data access layer objects are essential components of the system architecture, acting as conduits between the user-initiated actions at the frontend and the backend functionalities handled by control objects. By pairing each entity object with a corresponding data access layer object, the system ensures seamless interaction with data. These objects are pivotal in enabling CRUD operations on the data managed by the entities.

#### AvailabilityDAO

Paired with AvailabilityEntity, the AvailabilityDAO abstracts the complexity of CRUD operations related to checking and monitoring availability data. This DAO ensures efficient data handling, enhancing the reliability of availability checks within the system.

#### PriceDAO

Paired with PriceEntity, the PriceDAO streamlines the integration of price retrieval and monitoring into the system. It ensures data consistency and reliability by managing CRUD operations focused on pricing information.

#### DataExportDAO

Paired with DataExportEntity, the DataExportDAO manages CRUD operations for data export tasks. This pairing facilitates the transformation of raw data into structured formats like Excel and HTML, enabling efficient data reporting and accessibility.

#### TokenConfigDAO

Paired with configuration settings, the TokenConfigDAO handles CRUD operations related to authentication tokens and other configuration parameters. This DAO ensures secure handling and retrieval of sensitive configuration data, crucial for maintaining system integrity and security.

#### EmailConfigDAO

Paired with EmailEntity, the EmailConfigDAO manages CRUD operations related to email configuration settings. This includes handling email server details, user credentials, and recipient information, ensuring that email functionality is robust and reliable.

### Associations Among Objects

* Boundary to Control Associations
  + AvailabilityBoundary communicates with AvailabilityControl.
  + BotBoundary communicates with BotControl.
  + BrowserBoundary communicates with BrowserControl.
  + PriceBoundary communicates with PriceControl.
* Control to Entity Associations
  + AvailabilityControl interacts with AvailabilityEntity, DataExportEntity, and EmailEntity.
  + BotControl interacts with EmailEntity.
  + BrowserControl interacts with BrowserEntity.
  + PriceControl interacts with PriceEntity and DataExportEntity.

### Aggregates Among Objects

Aggregates group related objects under the control of a single aggregate root, ensuring that all interactions with the objects within the aggregate are mediated by this root. This design ensures that the lifecycle of the objects within the aggregate is managed consistently, maintaining data integrity and simplifying complex interactions.

Below, we identify the four primary aggregates in the system and detail their root objects and the other components they manage:

#### **Availability Aggregate**

* Root: AvailabilityControl
* Includes: AvailabilityEntity, DataExportEntity, EmailEntity

The AvailabilityControl object acts as the root, managing all operations related to checking and monitoring availability. It coordinates data storage via AvailabilityEntity, exports data through DataExportEntity, and sends notifications using EmailEntity. External systems interact only with AvailabilityControl, ensuring that all availability-related processes are managed centrally.

#### Price Aggregate

* Root: PriceControl
* Includes: PriceEntity, DataExportEntity, EmailEntity

The PriceControl object is the root for all price monitoring activities. It interacts with PriceEntity to retrieve and store price data and uses DataExportEntity to export this data when necessary. Email notifications are handled through EmailEntity. By centralizing these operations under PriceControl, the aggregate ensures that the entire process of price monitoring and reporting is managed efficiently.

#### Email Aggregate

* Root: EmailEntity
* Includes: Email configurations, Attachment management

EmailEntity is responsible for managing email communication, including the sending of notifications, handling attachments, and managing email configurations such as SMTP settings. This object is the root of the email aggregate, ensuring that all email-related functionality is managed within a single entity.

#### Browser Automation Aggregate

* Root: BrowserControl
* Includes: BrowserEntity

The BrowserControl object serves as the root for the browser automation processes, managing browser interactions like launching, navigating, and closing browser sessions. It interacts with BrowserEntity to handle the low-level details of these tasks. This structure ensures that all browser operations are coordinated centrally, streamlining the automation process.

### Attributes for Each Object

Table 2: Attributes for Each Object.

| **Object** | **Attributes** | **Methods** |
| --- | --- | --- |
| **AvailabilityEntity** | +availability\_data: List<String>, +last\_checked: DateTime | +check\_availability(): boolean, +export\_data(format: String): void |
| **BrowserEntity** | +cookies: List<Cookie>, +session\_data: Map<String, String> | +launch\_browser(): void, +close\_browser(): void +navigate\_to\_website(url: String): void, |
| **DataExportEntity** | +file\_paths: List<String> | +export\_to\_excel(data: Object): void, +export\_to\_html(data: Object): void |
| **EmailEntity** | +email\_queue: List<Email> | +send\_email\_with\_attachments(attachments: List<String>): void |
| **PriceEntity** | +price\_data: Map<String, Double>, +last\_updated: DateTime | +get\_price(): double, +export\_data(format: String): void |
| **project\_help\_boundary** | +command: String | +display\_help(): void |
| **receive\_email\_boundary** | +email\_address: String | +send\_email\_with\_attachment(file\_path: String): void |
| **close\_browser\_boundary** | +browser\_status: boolean | +close\_browser(): void |
| **login\_boundary** | +username: String, -password: String | +login\_to\_website(url: String): boolean |
| **navigate\_to\_website\_boundary** | +current\_url: String | +navigate\_to\_website(url: String): void |
| **check\_availability\_boundary** | +commands: List<String> | +check\_availability(): boolean |
| **start\_monitoring availability\_boundary** | +commands: List<String> | +start\_monitoring\_availability(url: String, frequency: int): void |
| **stop\_monitoring availability\_boundary** | +commands: List<String> | +stop\_monitoring\_availability(): void |
| **get\_price\_boundary** | +commands: List<String> | +get\_price(url: String): double |
| **start\_monitoring price\_boundary** | +commands: List<String> | +start\_monitoring\_price(url: String, frequency: int): void |
| **stop\_monitoring price\_boundary** | +commands: List<String> | +stop\_monitoring\_price(): void |
| **project\_help\_control** | +available\_commands: List<String> | +display\_help(): void |
| **receive\_email\_control** | +email\_address: String | +send\_email(file\_path: String): void |
| **navigate\_to\_website\_control** | +current\_url: String | +navigate\_to\_website(url: String): boolean |
| **login\_control** | +login\_status: boolean | +login(url: String, username: String, password: String): boolean |
| **close\_browser\_control** | +browser\_instance: BrowserEntity | +close\_browser(): void |
| **check\_availability\_control** | +monitoring\_active: boolean | +check\_availability(url: String): boolean |
| **start\_monitoring availability\_control** | +monitoring\_active: boolean | +start\_monitoring\_availability(url: String, frequency: int): void |
| **stop\_monitoring availability\_control** | +monitoring\_active: boolean | +stop\_monitoring\_availability(): void |
| **get\_price\_control** | +price\_history: List<Double>, +monitoring\_active: boolean | +get\_price(url: String): double |
| **start\_monitoring\_price\_control** | +monitoring\_active: boolean | +start\_monitoring\_price(url: String, frequency: int): void |
| **stop\_monitoring\_price\_control** | +monitoring\_active: boolean | +stop\_monitoring\_price(): void |

## Design

This section delves into the design framework of the Discord bot, illustrating how complex interactions are managed within a structured environment. Through UML class diagrams and detailed descriptions, we explore the bot's operational logic and its component interactions.

The design section outlines the configuration of boundary objects, control objects, and entity objects, each integral to the bot's functionality. Boundary objects serve as the interface between user commands and the system’s logic, ensuring inputs are accurately processed. Control objects, the decision-making core, manage the operational flow, orchestrating responses and actions efficiently. Entity objects, responsible for data manipulation, ensuring data integrity and availability, crucial for the bot's operations.

### Authentication Subsystem

* + Boundary Objects**:** login\_boundary
  + Control Objects: login\_control
  + Entity Objects: BrowserEntity
  + DAO Objects: TokenConfigDAO

### Notification Subsystem

* + Boundary Objects: receive\_email\_boundary, project\_help\_boundary
  + Control Objects: receive\_email\_control
  + Entity Objects: EmailEntity
  + DAO Objects: EmailConfigDAO

### Browser Subsystem

* + Boundary Objects: navigate\_to\_website\_boundary, close\_browser\_boundary
  + Control Objects: navigate\_to\_website\_control, close\_browser\_control, project\_help\_control
  + Entity Objects: BrowserEntity

### Monitoring Subsystem

* + Boundary Objects: start\_monitoring\_price\_boundary, stop\_monitoring\_price\_boundary start\_monitoring\_availability\_boundary, stop\_monitoring\_availability\_boundary, get\_price\_boundary, check\_availability\_boundary.
  + Control Objects: check\_availability\_control, start\_monitoring\_availability\_control, stop\_monitoring\_availability\_control, get\_price\_control, start\_monitoring\_price\_control, stop\_monitoring\_price\_control
  + Entity Objects: AvailabilityEntity, PriceEntity
  + DAO Objects: AvailabilityDAO, PriceDAO

### Data Handling Subsystem

* + Entity Objects: DataExportEntity
  + DAO Objects: DataExportDAO

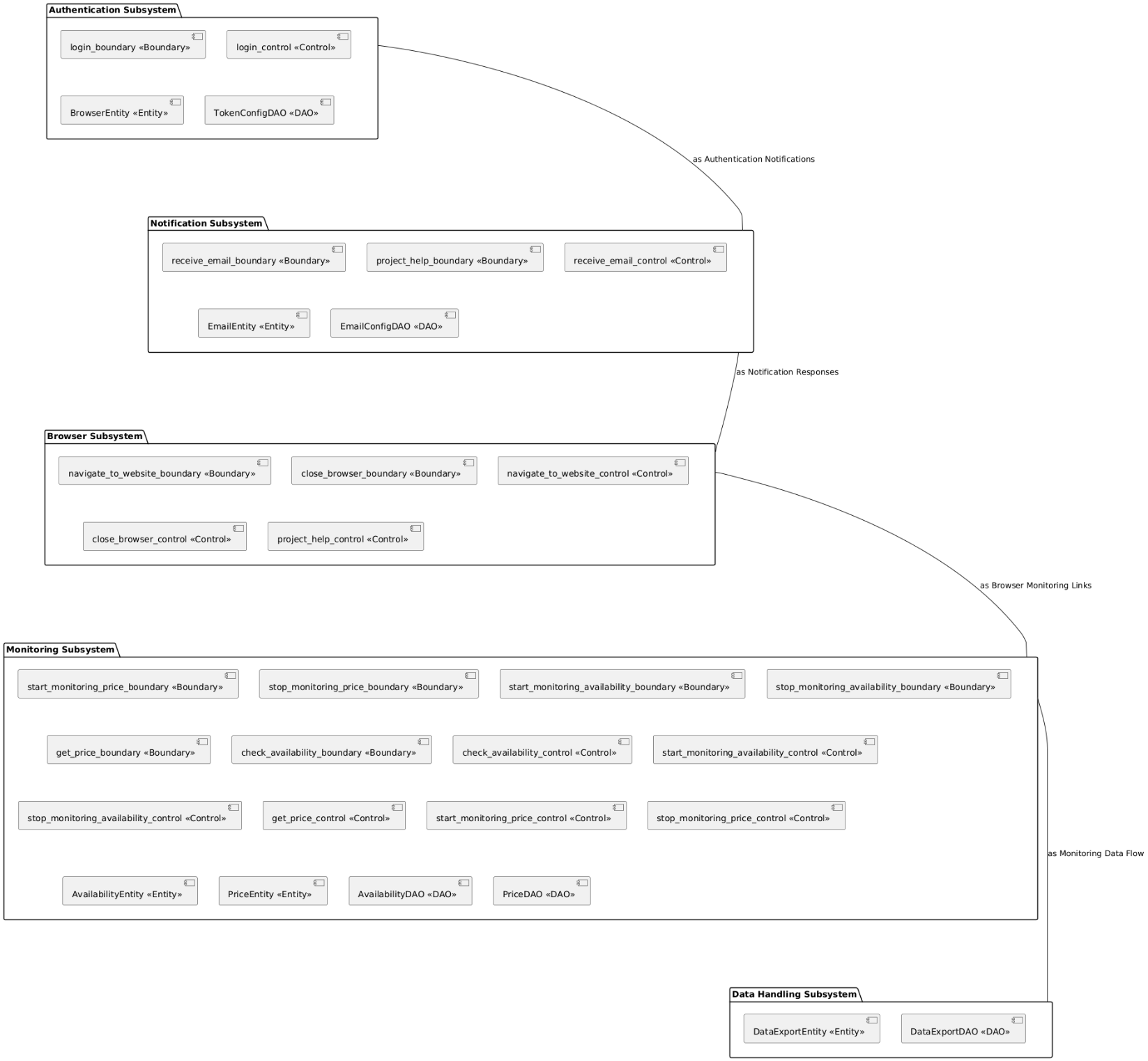


Figure 7: UML Component Diagram.

## Interface Specification

This section delves into the interface and class structure of the Discord Bot system, focusing on the interactions and functionalities enabled through the system's architecture. The components, from bot interaction handling to data management and task automation, are elaborated through both visual representation and detailed descriptions.

A screenshot of a computer program

Description automatically generatedA group of white rectangular boxes with text

Description automatically generated with medium confidence

Figure 8: UML Class Diagram.

**UML Class Diagram Overview**

The UML class diagram, shown in Figure 8, visually represents the architecture of the Discord Bot system, illustrating how various classes are interconnected and interact within the system. This diagram helps to understand the structural relationships and the flow of data across the system, providing insights into the object-oriented design utilized

The diagram serves as a high-level depiction of the system's structure. However, it does not encompass all the details of the system’s components due to the complexity and the breadth of the system. To address this, a detailed tabular description for each class in architecture is provided below. These tables explore the attributes and methods of each class, specifying their visibility, types, and functionality in detail.

## Mapping Contracts to Exceptions

In this section, we outline how specific contracts within the system's methods are mapped to exception handling mechanisms. Each object in the system is responsible for fulfilling certain operations, such as retrieving data or performing actions based on user input. If these operations fail to meet their contract (e.g., due to invalid inputs or external system failures), predefined exceptions are raised to manage these failures effectively. The following table presents the mapping between the main contracts of each method and their corresponding exception classes, ensuring robust error handling throughout the system.

Table 3: Contracts and Exceptions.

| **Object** | **Method** | **Contract (Expectation)** | **Exceptions** |
| --- | --- | --- | --- |
| **AvailabilityEntity** | check\_availability() | Fetch availability from a URL. | InvalidURLException, TimeoutException |
| **AvailabilityControl** | start\_monitoring\_availability() | Start monitoring availability at specified intervals. | MonitoringAlreadyRunningException, InvalidURLException |
| **BrowserEntity** | launch\_browser() | Open a new browser session. | BrowserLaunchException |
| **BrowserEntity** | navigate\_to\_website() | Navigate to a specified URL. | InvalidURLException, NavigationException |
| **EmailEntity** | send\_email\_with\_attachments | Send an email with specified attachments. | FileNotFoundException, EmailSendFailureException |
| **PriceEntity** | get\_price() | Fetch the price of a product from a URL. | InvalidURLException, PriceNotFoundException |
| **PriceControl** | start\_monitoring\_price() | Start monitoring price at specified intervals. | MonitoringAlreadyRunningException, InvalidURLException |
| **PriceControl** | get\_price() | Fetch price and log to data export formats (Excel/HTML). | PriceFetchException, ExportException |
| **BotControl** | receive\_command() | Handle commands related to help or stopping the bot. | InvalidCommandException, BotShutdownException |
| **check\_availability control** | check\_availability() | Retrieve availability from a given URL and export the data. | AvailabilityCheckException, ExportException |
| **start\_monitoring price\_control** | start\_monitoring\_price() | Monitor the price at defined intervals and send notifications. | MonitoringException, InvalidURLException |
| **login\_control** | login() | Log into the website with provided credentials. | LoginFailedException, InvalidCredentialsException |
| **DataExportEntity** | export\_to\_excel() | Export data into an Excel file. | FileExportException |
| **DataExportEntity** | export\_to\_html() | Export data into an HTML file. | FileExportException |
| **receive\_email\_control** | send\_email\_with\_attachments | Send an email with specified attachments. | FileNotFoundException, EmailSendFailureException |
| **AvailabilityEntity** | export\_data() | Export availability data to Excel/HTML. | ExportException |
| **PriceControl** | stop\_monitoring\_price() | Stop the price monitoring process. | MonitoringNotRunningException |
| **AvailabilityControl** | stop\_monitoring\_availability() | Stop the availability monitoring process. | MonitoringNotRunningException |

## Data Management Strategy

In the development of our Discord bot, we intentionally moved away from traditional relational databases and opted for file-based storage formats, specifically Excel and HTML, for reporting and data presentation. This decision was driven by the project’s specific needs for lightweight infrastructure, user-friendly interfaces, and real-time feedback.

### Data Presentation and Usability

Rather than relying on databases that require ongoing maintenance and potentially introduce latency, Excel and HTML offer user-centric solutions that directly cater to the bot’s target audience—non-technical users who expect clear, familiar data presentation. Excel is a universally recognized format that enables users to easily manipulate, filter, and analyze data without requiring additional software or skills. Its built-in functionalities, such as pivot tables, graphs, and automated formulas, are key for users who want to interact with their data quickly and intuitively [22].

Additionally, HTML offers dynamic web-based presentation, enabling the bot to generate visually appealing, interactive reports that can be accessed instantly via any web browser. This approach aligns with modern web-based interfaces, allowing users to receive real-time reports in an accessible format without needing specialized software or technical expertise. By leveraging HTML, the system can deliver interactive elements, such as collapsible sections and hyperlinks, enhancing the user’s ability to navigate and interpret their data on the fly [23].

### Performance and Flexibility

While relational databases provide powerful solutions for handling large, structured datasets, they introduce unnecessary complexity for our bot’s real-time, non-transactional tasks. The operations primarily involve fetching prices and availability data, which do not require the overhead of database normalization, indexing, or query optimization. Storing non-persistent data (such as session information or user preferences) in JSON ensures fast, lightweight interactions that allow the bot to respond to user commands immediately, ensuring a responsive user experience [24].

Additionally, Excel and HTML are lightweight solutions with minimal infrastructure requirements. Using these file formats reduces both development complexity and deployment time. Traditional databases demand server resources, backup strategies, and schema management—all of which add unnecessary overhead to a Discord bot that prioritizes speed and simplicity. For example, distributing data in Excel sheets eliminates the need for end-users to interact with a complex database system while providing all the functionality they need in a familiar environment [25].

A computer screen shot of a program

Description automatically generated

Figure 9: Transient and Persistent Data Handling.

### Rapid Deployment and Maintenance-Free Operation

A key advantage of our approach is the ability to deploy the bot with minimal infrastructure, eliminating the need for dedicated database management. By avoiding databases, we reduce potential points of failure related to database connections, queries, and data migrations. In environments where quick deployment is crucial, such as with Discord bots that handle ephemeral tasks, file-based storage offers a streamlined solution that ensures the bot is ready to operate out of the box without additional setup.

Moreover, Excel and HTML can be easily shared or emailed directly to users. This bypasses the need for complex integrations with database systems to export or generate reports. The user-friendly nature of these formats means that reports can be generated and distributed with minimal effort while maintaining the integrity and accessibility of the data [26].

### Simplified Data Handling and Security

Security and ease of configuration are also key considerations. Storing sensitive information, such as tokens and credentials, in environment files ensures that sensitive data is isolated from the bot’s core logic. By decoupling sensitive configuration from data reporting, the bot minimizes security risks commonly associated with database breaches. This approach leverages environmental variables, ensuring that critical information is not embedded directly within files that users might access or manipulate [27] .

By using JSON for transient data and Excel/HTML for reporting, the system can focus on speed and ease of use while maintaining robust data reporting capabilities. This file-based strategy perfectly aligns with the Discord bot’s design philosophy: to deliver fast, actionable feedback to users without burdening the system with unnecessary complexity.

## Technology Stack and Framework

This section delves into the technology stack and frameworks that power the Discord bot, focusing on the tools and technologies that facilitate rapid development, seamless user interaction, and efficient data management.

### Programming Languages and Frameworks

#### *Python*

* Role: Primary programming language for developing the bot.
* Features: Chosen for its readability, robust standard library, and extensive support through third-party libraries, Python underpins all major functionalities of the bot, from data scraping to process automation and interaction handling [28].

#### *Selenium*

* Role: Automates web browsers to extract real-time product prices and availability.
* Capabilities: Simulates human interactions with web pages, allowing the bot to perform complex navigations and data extraction tasks, critical for accurate price monitoring [29].

#### *Discord.py*

* Role: Handles communications with the Discord API.
* Functionality: Manages user interactions, receives commands, sends notifications, and embeds the bot seamlessly within Discord communities [30].

### Tools and Platforms

#### *Visual Studio Code*

* Role: Preferred IDE for writing, testing, and debugging the bot’s code.
* Advantages: Offers extensive plugin support, built-in Git control, and integrated terminal, which streamline the coding and version control processes [31].

#### *Git*

* Role: Manages source code versions and collaborative features.
* Benefits: Essential for tracking code changes, managing branches, and integrating changes from multiple contributors, ensuring consistency and continuity in the development process.

#### *GitHub*

* Role: Hosts the source code repository and facilitates collaborative features like issue tracking and code reviews.
* Integration: Centralizes source control and acts as a platform for continuous integration and deployment strategies.

### Data Management and Storage

This project utilizes a combination of configuration files, JSON, and direct file output mechanisms for managing both transient and persistent data:

#### *Configuration Files*

* Role: Manage operational parameters and sensitive credentials, such as Tokens, SMTP settings.
* Implementation: Stored in .py files, these parameters are loaded dynamically into the application environment, enhancing security by segregating configuration from the code.

#### *JSON Files*

* Role: Handle transient data like session states and user preferences.
* Advantages: Offers flexibility and speed in accessing and updating data, ideal for non-sensitive, temporary information.

#### *Excel and HTML*

* Role: Serve as formats for logging long-term data and generating reports.
* Functionality: Facilitates easy distribution and accessibility of data, allowing comprehensive reporting and analysis through automated emails.

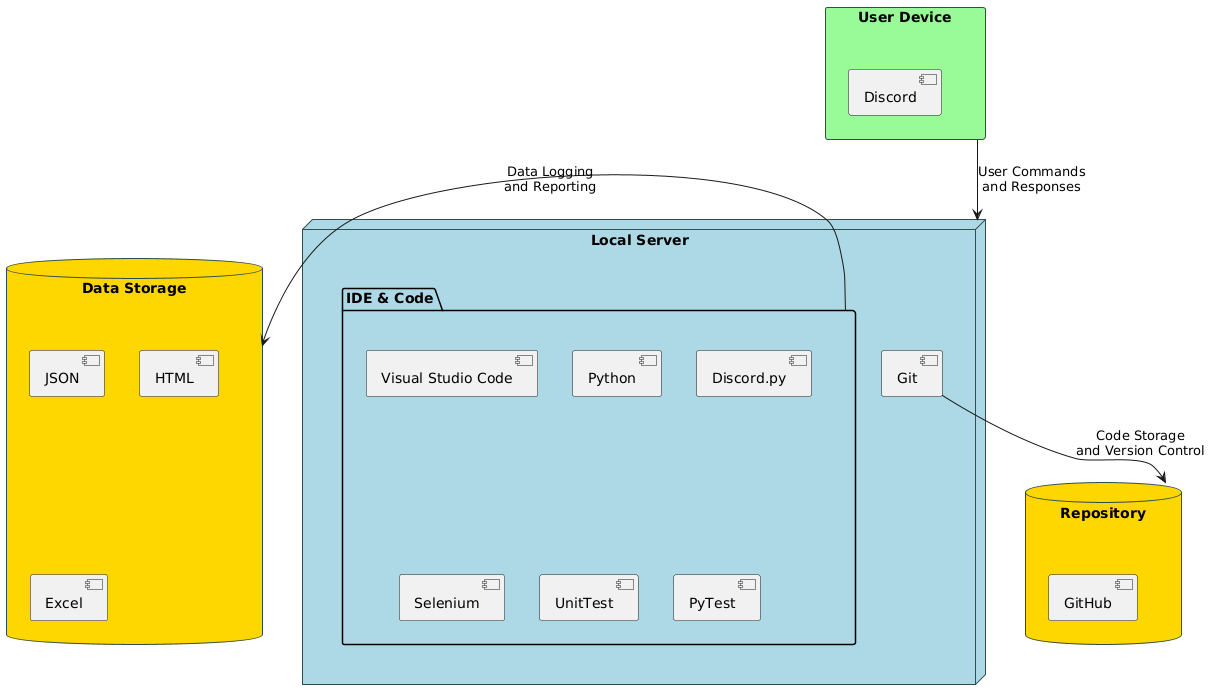


Figure 10: System Architecture Diagram.

### Testing Strategy

Our project employs a robust testing framework using Python’s unittest library and unittest.mock for mocking external dependencies. This strategy ensures that each component of the bot functions as expected under various scenarios. A detailed exploration of our testing approach and methodologies will be presented in the subsequent chapter.

## Conclusion

This chapter provided a comprehensive exploration of the system design and implementation of the Discord bot project. Starting with an analysis of the Project Requirements, we defined the core functionalities and use cases the bot is designed to handle, such as price checking and availability monitoring. Through the Architecture section, we examined the high-level structure of the system, with detailed UML diagrams illustrating how the various boundary, control, and entity objects interact.

In the Design section, we broke down the core subsystems and how they integrate to provide seamless automation within the Discord environment. The Interface Specification further clarified how the bot communicates with users and external services. A significant focus was placed on Mapping Contracts to Exception Classes, ensuring robust error handling for all operations.

The chapter also addressed the Data Management Strategy, emphasizing the decision to use file-based storage in Excel and HTML formats for flexibility, rapid deployment, and user-friendly reporting. Finally, the Technology Stack and Framework provided insight into the tools and programming languages used to implement the project, ensuring smooth functionality and scalability.

Moving forward to Chapter 4, we will delve into the detailed testing strategies used to validate the system, including how unit tests and mock objects were employed to ensure the system meets the defined requirements. Additionally, Chapter 4 will explore the results and performance analysis, ensuring that the bot operates effectively within the defined parameters.

# CHAPTER FOUR: TESTING & DEFECTS

This document provides an overview of unit testing for a software project aimed at automating the monitoring of product prices and service availability. The goal of the testing is to ensure that all system components function correctly when tested in isolation. This modular approach facilitates the validation of core system functions, including command processing, browser automation, and data export, within a controlled test environment.

The system is composed of modules responsible for interacting with web browsers, processing user commands, retrieving product data from websites, and monitoring availability for services like reservations. These modules have undergone rigorous testing using Python's pytest framework. External systems like websites and Discord commands are simulated using mocks and patches, ensuring expected system behavior in both typical and edge-case scenarios.

This document includes:

* + - An outline of the testing strategy, scope, and objectives,
    - A description of the tools and technologies used during testing,
    - Solutions to challenges encountered when testing Discord commands,
    - Details on the test setup, implementation, and how the testing framework integrates with the system architecture.

The purpose of the unit testing is to confirm that the system can accurately process user commands, interact with websites, retrieve data, log it, and generate reports. Isolating components during testing enhances confidence in the system's reliability and robustness



## Test Plan Overview

### Scope

The scope of the unit tests covers all critical aspects of the system, ensuring each component performs its intended function independently. This modular testing approach covers:

* Command Processing and Core Features: Verifying that the system properly receives and processes user commands to monitor prices, check availability, and log data efficiently.
* Browser Interactions: Ensuring that the system can initiate, navigate, and close browser sessions while effectively interacting with web content.
* Data Logging and Export: Validating that price and availability data are logged and exported in structured formats such as Excel and HTML.
* Error Handling: Confirming that the system gracefully handles errors (e.g., invalid commands or network issues) and provides appropriate user feedback.

### Objectives

The unit testing aims to:

* Functional Verification: Ensure that components like command processing, data retrieval, and logging function correctly in isolation.
* Component Isolation: By testing each module independently, failures in one area don’t affect others, enabling easier identification of defects.
* Data Accuracy and Consistency: Ensure the system processes price and availability data correctly before logging and exporting it.
* System Reliability: Test the system’s ability to handle various scenarios, including repeated commands, long-running processes, and invalid inputs.

### Strategy

The strategy focuses on modular unit testing, ensuring each part of the system is validated without relying on external dependencies like live websites or browsers. Key elements include:

* Unit Testing: Each module - command processing, web scraping, or data export is tested independently.
* Mocking and Simulation: External systems are simulated using mocks, allowing tests to focus on internal logic without live interactions.
* Automated Execution: Tests are automated using pytest, ensuring consistency and enabling integration into CI/CD pipelines for automatic execution upon code changes.

### Structure of the Tests

The tests are divided into suites targeting specific components:

* Control Layer: Verifies user commands are correctly processed.
* Entity Layer: Validates interactions with external systems (e.g., retrieving product prices or checking availability).
* Data Logging and Export: Ensures data is logged and exported without errors.

This structure allows the test framework to expand as the system evolves, enabling independent testing of new features without disrupting existing tests.

### Expected Outcomes

This modular approach is expected to yield:

* Accurate command processing with correct results,
* Error-free logging and export of data,
* Graceful handling of unexpected situations like invalid commands or network failures,
* Stable performance during long-running tasks, such as continuous monitoring of product prices.

## Tools and Technologies

### Pytest

The primary framework used for test execution is pytest, which supports both synchronous and asynchronous testing. This is critical since many system operations involve real-time monitoring and asynchronous tasks like web scraping. Integration with mocking tools allows thorough simulation of external dependencies, ensuring isolated and repeatable tests.

### Unittest.mock

The unittest.mock library is key to isolating system components from external dependencies, such as web browsers and Discord commands. The system uses Mock and AsyncMock to simulate responses from these services, enabling tests to focus on internal logic.

* Mocking External Systems: Mocks simulate browser actions and Discord command inputs, allowing test isolation.

### pytest-asyncio

As many system operations are asynchronous, pytest-asyncio manages async code in tests, ensuring that operations like price monitoring are tested properly.

### Mocked Selenium

Selenium, a tool for browser automation, is mocked during unit testing to focus on internal logic without requiring real browser instances.

## Purpose and Setup

### Purpose of Unit Testing

The purpose of unit testing is to validate that each component of the system functions correctly in isolation, focusing on command processing, web interactions, and data export operations.

### Challenges in Testing Discord Commands

Testing Discord commands posed a challenge due to the lack of native testing tools for discord.py. To address this, unit tests simulate command inputs and directly interact with the control layer methods, ensuring that the system logic is properly tested without relying on live Discord command handling.

### Setup of the Testing Environment

The environment is designed to ensure isolated component testing without live system dependencies:

* Mocking and Patching: The unittest.mock library simulates external systems like browsers and Discord commands, ensuring independent testing of each component.
* Asynchronous Testing: pytest-asyncio allows proper testing of asynchronous tasks like price monitoring.
* Test Isolation: Each test runs independently, ensuring faster execution and easier debugging.

### Implementation Details

The structure of the tests mirrors the system’s modular architecture, with distinct test suites for each layer:

* Control Layer Tests: Validate command processing to ensure inputs are handled correctly.
* Entity Layer Tests: Confirm the core functionality of price retrieval, availability checking, and data export.
* Logging and Export Tests: Ensure data is correctly formatted and saved to Excel and HTML files.

By focusing on these areas, the unit tests confirm that the system’s logic functions as intended, and any external dependencies are properly mocked for accurate results.

## Unit Tests for Use Cases

Every use case has multiple unit tests in them for every step in main flow.

#### test\_init.py

The test\_init file serves two main purposes. First, it consolidates all necessary imports to avoid redundant import statements across multiple test files, improving maintainability and consistency. Second, it provides functionality to run all unit tests at once by executing test\_init.py.

A screen shot of a computer program

Description automatically generated

Figure 11: test\_init.py file code

If specific tests need to be run individually, each test file has the if \_\_name\_\_ == "\_\_main\_\_": pytest.main([\_\_file\_\_]) block, allowing users to run that specific test file independently.

This setup streamlines both the import process and test execution, making it easy to run tests collectively or individually based on the needs of the project.

A computer screen with text

Description automatically generated

Figure 12: pytest code that runs the current test case only

### !project\_help

#### Description

This test ensures that the BotControl.receive\_command() method processes the !project\_help command correctly and returns the appropriate help message listing available commands.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test will ensure that BotControl.receive\_command() handles the !project\_help command correctly and returns the expected help message.

#### Test Data

* Command: "!project\_help"
* Expected Output: "Here are the available commands:..."

#### Output and Source Code

A computer screen shot of a black background

Description automatically generated

A screen shot of a computer program

Description automatically generated

Figure 13: Output and Code-1.

### !receive\_email <file name>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !receive\_email command correctly by passing the file name to the email handler, sending the email, and generating the correct response.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !receive\_email command correctly, including proper parameter passing and validation.
2. Email Handling  
   This test focuses on the EmailEntity.send\_email\_with\_attachments() function to ensure it processes the request to send the email with the attached file.
3. Response Generation  
   This test validates that the control layer correctly interprets the response from the email handling step and returns the appropriate result to the boundary layer.

#### Output and Source Code

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

Figure 14: Output and Code-2.

### !navigate\_to\_website

#### Description

This test ensures that the BotControl.receive\_command() method processes the !navigate\_to\_website command correctly by extracting the URL, navigating to the specified website, and returning the appropriate result.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !navigate\_to\_website command correctly by extracting the URL and passing it to the browser control.
2. Browser Navigation  
   This test ensures that the BrowserEntity.navigate\_to\_website() function processes the navigation request to the specified URL correctly.
3. Response Generation  
   This test validates that the control layer correctly returns the appropriate result after the browser interaction is completed.

#### Test Data

* Command: "!navigate\_to\_website"
* Test URL: <http://example.com>
* Expected Output: "Navigation successful"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Figure 15: Output and Code-3

### !login

#### Description

This test ensures that the BotControl.receive\_command() method processes the !login command correctly by passing the website, username, and password to the browser and verifying the login process.

#### Test Steps

The main flow for this use case is as follows, and we will test the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !login command correctly, including proper parameter passing and validation.
2. Website Interaction  
   This test focuses on the BrowserEntity.login() function to ensure it processes the request to log in to the website using the provided credentials.
3. Response Generation  
   This test validates that the control layer correctly interprets the response from the website interaction step and returns the appropriate result to the boundary layer.

#### Test Data

* Command: "!login"
* Test website: "http://example.com"
* Test username: "user"
* Test password: "pass"
* Expected Output: "Login successful"

#### Output and Source Code

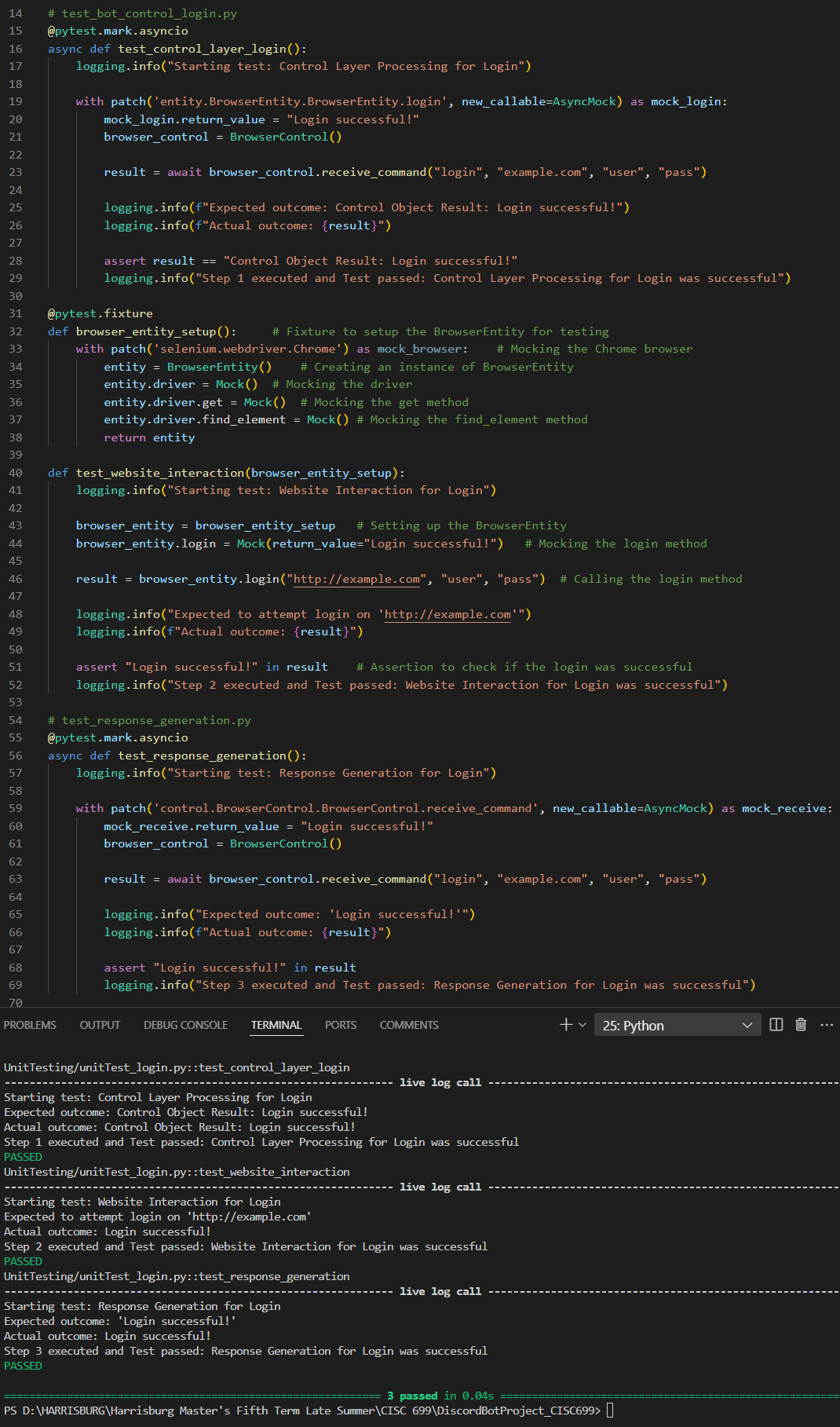


Figure 16: Output and Source Code-4.

### !close\_browser

#### Description

This test ensures that the BotControl.receive\_command() method processes the !close\_browser command correctly by handling browser closure and returning the appropriate response.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !close\_browser command correctly.
2. Browser Closing  
   This test ensures that the BrowserEntity.close\_browser() function successfully closes the browser.
3. Response Generation  
   This test validates that the control layer correctly interprets the browser closure and returns the appropriate result to the boundary layer.

#### Test Data

* Command: "!close\_browser"
* Expected Output: "Browser closed successfully"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Figure 17: Output and Source Code-5.

### !get\_price <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !get\_price command correctly by extracting the website URL, retrieving the price, and logging the data to both Excel and HTML formats.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !get\_price command correctly, including URL parameter handling.
2. Price Retrieval  
   This test ensures that the PriceEntity.get\_price\_from\_page() function retrieves the correct price from the webpage.
3. Data Logging to Excel  
   This test verifies that the retrieved price data is correctly logged to an Excel file.
4. Data Logging to HTML  
   This test ensures that the price data is correctly exported to an HTML file.

#### Test Data

* Command: "!get\_price"
* Test website: "http://example.com/product"
* Expected Output: "100.00"

#### Output and Source Code

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Figure 18: Output and Source Code-6.

### !start\_monitoring\_price <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !start\_monitoring\_price command correctly by initiating price monitoring at regular intervals for the specified website.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !start\_monitoring\_price command correctly, including proper URL parameter passing.
2. Price Monitoring Initiation  
   This test ensures that price monitoring is initiated and repeated at regular intervals by calling the get\_price() function.
3. Stop Monitoring Logic  
   This test confirms that price monitoring can be stopped correctly and the final results are collected.

#### Test Data

* Command: "!start\_monitoring\_price"
* Test website: "http://example.com/product"
* Expected Output: "Price monitoring started"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

### !stop\_monitoring\_price <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !stop\_monitoring\_price command correctly by stopping the monitoring process and generating a final summary of the results.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !stop\_monitoring\_price command correctly.
2. Stop Monitoring Logic  
   This test ensures that the monitoring process is stopped and results are collected.
3. Final Summary Generation  
   This test validates that a final summary of the price monitoring results is generated and returned.

#### Test Data

* Command: "!stop\_monitoring\_price"
* Test website: "http://example.com/product"
* Expected Output: "Price monitoring stopped"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

### !check\_availability <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !check\_availability command correctly by checking the availability of the specified service on the website and logging the results.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !check\_availability command correctly.
2. Availability Checking  
   This test ensures that the AvailabilityEntity.check\_availability() function checks the availability of the specified service.
3. Data Logging to Excel  
   This test verifies that the availability data is logged to an Excel file.
4. Data Logging to HTML  
   This test ensures that the availability data is exported to an HTML file.

#### Test Data

* Command: "!check\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability confirmed"

#### Output and Source Code

A screenshot of a computer screen

Description automatically generated

### !start\_monitoring\_availability <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !start\_monitoring\_availability command correctly by initiating service availability monitoring at regular intervals for the specified website.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !start\_monitoring\_availability command correctly.
2. Availability Monitoring Initiation  
   This test ensures that service availability monitoring is initiated and repeated at regular intervals.
3. Stop Monitoring Logic  
   This test confirms that availability monitoring can be stopped correctly and the final results are collected.

#### Test Data

* Command: "!start\_monitoring\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability monitoring started"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

### !stop\_monitoring\_availability <website>

#### Description

This test ensures that the BotControl.receive\_command() method processes the !stop\_monitoring\_availability command correctly by stopping the monitoring process and generating a final summary of the results.

#### Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. Control Layer Processing  
   This test ensures that BotControl.receive\_command() handles the !stop\_monitoring\_availability command correctly.
2. Stop Monitoring Logic  
   This test ensures that the monitoring process is stopped and results are collected.
3. Final Summary Generation  
   This test validates that a final summary of the availability monitoring results is generated and returned.

#### Test Data

* Command: "!stop\_monitoring\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability monitoring stopped"

#### Output and Source Code

A screenshot of a computer program

Description automatically generated

## Unit Test Summary

This project aimed to rigorously test various functionalities of the Discord bot, particularly focusing on control and entity layers across key use cases such as availability checking, price monitoring, and email handling. The test cases were structured to cover command reception, process execution, and result validation, ensuring that each component behaved as expected under both normal and boundary conditions.

Through the execution of the 34-unit tests, various issues were identified, such as ModuleNotFoundErrors and assertion failures, which were addressed promptly. This defect identification helped improve the robustness of the bot, highlighting the importance of unit testing in developing reliable software systems.

The structured approach, using pytest and mocking frameworks, allowed the testing process to be efficient and scalable. While limitations in testing the boundary layers, particularly in simulating real-world Discord interactions, were acknowledged, the tests effectively ensured that the core logic and functionalities performed correctly.

In conclusion, the thorough testing and defect resolution process not only enhanced the bot’s performance but also provided valuable insights into software testing strategies, emphasizing the need for comprehensive test coverage in ensuring product quality.

## Defects intro

This report focuses on identifying and documenting the defects encountered during the testing of the Discord Bot Automation Assistant. While developing and testing the bot, we employed a structured approach to uncover, document, and resolve various defects. The primary goal of this report is to explain the nature of these defects, their possible causes, and how they were addressed through testing and debugging. By focusing on the testing strategy and how defects appeared throughout the project, we can highlight both the challenges and solutions that contributed to improving the quality of the bot.

### Design of the Testing Process

The testing process was designed to systematically verify the functionality of the bot's various features, with a particular emphasis on ensuring asynchronous commands, browser interactions, and data logging mechanisms worked as expected. We structured our testing in a modular way, just like the bot's architecture itself.

* Unit Test Design: For each feature of the bot (e.g., availability checking, price monitoring), we designed specific unit tests to isolate individual components. Each test was aimed at ensuring that commands were received and processed correctly, data was logged appropriately, and errors were handled effectively.
* Defect Detection Strategy: The tests were specifically designed to surface potential issues in both normal and edge-case scenarios. For example, we tested whether functions like receive\_command could handle various inputs (valid and invalid) and whether data exports occurred successfully in both Excel and HTML formats.

The design of our testing strategy was iterative, meaning that as new defects emerged, tests were expanded or refined to catch similar issues in future code iterations.

### Implementation of Unit Tests

We implemented the unit tests using the pytest framework, complemented by pytest-asyncio to support asynchronous operations. Since many of the bot’s commands involve asynchronous tasks, handling these properly in tests was a key part of the implementation.

* Mocking and Isolation: We made extensive use of the unittest.mock.patch functionality to mock external dependencies, such as web scraping and browser interactions. This allowed us to isolate internal logic and simulate various real-world scenarios without needing to rely on live external systems.
* Test Scenarios: For each bot feature, we implemented several test scenarios:
  + Command Handling: Verifying that commands (e.g., check\_availability, get\_price) were correctly processed by the control layers.
  + Data Logging: Ensuring that results (e.g., availability status, prices) were logged to files (Excel/HTML) as expected.
  + Error Handling: Testing how the bot responded to erroneous inputs or missing elements, which often led to the discovery of key defects.

### Testing and Emergence of Defects

Through rigorous testing, we uncovered several defects that affected the bot's functionality. Defects often appeared in areas where asynchronous methods were mishandled, or when external elements (such as browser interactions) failed to execute properly. Here's how defects emerged during testing:

* Asynchronous Handling Defects: Defects related to the improper handling of asynchronous functions, such as missing await statements or coroutines not being executed correctly, were frequent. These defects were detected when unit tests returned incomplete results or errors indicating that coroutines were not awaited.
* Control Layer Defects: Several defects were found in the bot’s control layer, where commands like check\_availability and close\_browser were not handled as expected. These issues often appeared when the control layer failed to return the correct result or handled inputs incorrectly, leading to errors or unexpected outputs.
* Data Export Defects: In some cases, defects arose from the bot’s data logging functionality, particularly in exporting data to Excel and HTML formats. These defects were uncovered through tests that checked the integrity of the exported files and verified that they contained the correct data.

Each defect was documented with a focus on its cause and resolution, ensuring that the necessary fixes were applied not only to address the issue at hand but also to prevent similar problems from recurring.

## Defects

### Defect 1 - ImportError

Defect ID: DEF01

Date Repaired/Documented: September 2024

#### Description

The unit test for the AccountEntity class fails due to an ImportError. The test file is unable to locate and import the AccountEntity class because the folder structure causes incorrect module paths. Without the proper path configuration, the module cannot be recognized by the test script. This happened in all test cases, and it is not only specific to AccountEntity class.

#### Possible Causes

* Incorrect folder structure leading to broken module imports.
* Missing or misconfigured sys.path.append() to adjust Python’s path.

#### Repair Method

The issue was resolved by adding the following line to the test script:  
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))  
This line adjusts the Python path so that any module can be correctly imported into the test file.

#### Screenshot of Defect

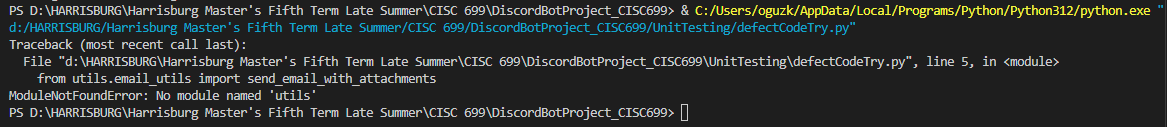


Figure 19: DEF01.

### Defect 2 - unitTest Async Method Handling Issue

Defect ID: DEF02

Date Repaired/Documented: September 2024

#### Description

During the testing of asynchronous functions in the DiscordBotProject\_CISC699, two tests related to monitoring availability failed when executed with unittest. The primary issue arose because unittest is not designed to handle async def functions natively. This resulted in runtime warnings and deprecation warnings, with the async coroutines being marked as "never awaited" during the execution of the tests.

When running the unittest framework, the following warnings were triggered:

* RuntimeWarning: *coroutine 'TestAvailabilityControl.test\_start\_monitoring\_availability\_success' was never awaited*
* DeprecationWarning: It is deprecated to return a value that is not None from a test case.

Despite these warnings, the tests appeared to complete successfully, but they did not actually execute the asynchronous logic as intended. This led to false positives, as the underlying issues in the async methods went undetected.

#### Possible Causes

The root cause of the defect was the inherent limitation of unittest when dealing with asynchronous functions. The unittest framework expects synchronous test cases, and when it encounters async def functions, it does not properly handle them, resulting in the warnings:

* RuntimeWarning: This occurs because the async functions were not awaited, meaning the event loop was never properly triggered, and the coroutine was essentially skipped.
* DeprecationWarning: This was raised because unittest expects test cases to return None. However, since async functions were involved, the coroutines were returning non-None values that unittest could not handle correctly.

The key problem is that unittest lacks the capability to handle event loops and asynchronous code execution, leading to incomplete or skipped tests when working with async def functions.

#### Repair Method

To resolve this issue, the testing framework was switched from unittest to pytest, which natively supports asynchronous functions via the pytest-asyncio plugin. This switch allowed for proper handling of the async methods, ensuring that the event loop is managed correctly and that the asynchronous code is fully executed during tests.

#### Screenshot of Defect

A black screen with many colorful lines

Description automatically generated with medium confidence

Figure 20: DEF02.

### Defect 3 - Missing pytest Fixture Decorator

Defect ID: DEF03

Date Repaired/Documented: September 2024

#### Description

This defect occurred due to the omission of the @pytest.fixture decorator from the base\_test\_case fixture in the test\_init.py file. The base\_test\_case fixture was responsible for initializing various control and entity objects needed by the test cases. However, without the @pytest.fixture decorator, the fixture could not be detected and injected into the test functions, leading to errors during test execution.When the tests were run, pytest was unable to recognize base\_test\_case as a valid fixture, resulting in the following error:

“fixture 'base\_test\_case' not found”

This caused any test (but it’s been discovered in test\_start\_monitoring\_price\_already\_running and test\_start\_monitoring\_price\_failure\_in\_entity) to fail because they were attempting to access uninitialized objects, such as base\_test\_case.price\_control.

The missing decorator prevented the proper setup of the test environment, leading to runtime failures and unhandled exceptions.

#### Possible Causes

The root cause of the defect was the omission of the @pytest.fixture decorator in the fixture definition. As a result, pytest did not recognize base\_test\_case as a fixture, and the test functions could not receive the necessary initialization data. Without the fixture, the test functions attempted to access uninitialized objects, causing AttributeError and fixture-not-found errors.

#### Repair Method

To resolve this issue, I initially thought we could simply call the base\_test\_case method directly, but since it is part of the test setup, we need to use the @pytest.fixture decorator. This decorator connects the method to the pytest framework, allowing it to automatically detect and inject the fixture into the test functions, ensuring that all necessary objects are initialized before the tests run.

Added the @pytest.fixture decorator: This decorator was applied to the base\_test\_case method to properly define it as a fixture that can be used across multiple test cases.

Once the @pytest.fixture decorator was added to the base\_test\_case function, the tests ran as expected, with the necessary objects being initialized before execution. This allowed the test cases to properly access and manipulate the price\_control and other controls during testing.

#### Screenshot of Defect

A screen shot of a computer

Description automatically generated

Figure 21: DEF03.

### Defect 4 - Missing “await” in Asynchronous Function Call

Defect ID: DEF04

Date Repaired/Documented: September 2024

#### Description

This defect occurred due to a missing await keyword in an asynchronous function call. The issue was identified in the test\_login\_success test case when invoking the receive\_command method. Because the await keyword was not added, the function returned a coroutine object instead of executing as intended, causing the test to fail.

Without the await keyword, the test captured the coroutine object (<coroutine object BrowserControl.receive\_command at 0x...>) instead of the expected control layer result, leading to an assertion failure. This also triggered a RuntimeWarning, indicating that the coroutine was never awaited.

Error Messages:

*AssertionError: Control layer assertion failed.*

*sys:1: RuntimeWarning: coroutine 'BrowserControl.receive\_command' was never awaited*

#### Possible Causes

The root cause of this defect was the omission of the await keyword in front of an asynchronous function call. In Python, when dealing with async def functions, the await keyword is required to pause execution until the asynchronous operation completes. Failing to add await results in the function returning a coroutine object, which was not the expected behavior for this test.

#### Repair Method

The defect was resolved by adding the await keyword before the asynchronous function call to ensure that the coroutine is properly awaited and executed.

result = base\_test\_case.browser\_control.receive\_command("login", site="example.com")

result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")

Once the await keyword was added, the test executed correctly, and the function returned the expected result, allowing the assertions to pass.

#### Screenshot of Defect

A computer screen shot of text

Description automatically generated

Figure 22: DEF04.

### Defect 5 - Missing Initialization of bot\_control in Test Fixture

Defect ID: DEF05

Date Repaired/Documented: September 2024

#### Description

In an effort to avoid code duplication, a dedicated test\_init.py file was created to centralize and simplify the initialization of various control and entity objects across all test functions. The goal was to use a single fixture, base\_test\_case, to initialize objects like browser\_control, account\_control, and others, so each test would have consistent access to the same resources.

However, during the setup, the bot\_control object was mistakenly initialized as a MagicMock instead of a proper BotControl instance. This mistake caused issues when running tests that required bot\_control, specifically in the test\_project\_help\_success and test\_project\_help\_failure test cases.

The error manifested in an unusual and confusing way, making it difficult to identify at first. When attempting to use bot\_control.receive\_command in the await expression, the error message indicated:

TypeError: object MagicMock can't be used in 'await' expression

This error occurred because instead of bot\_control being an instance of BotControl, it was a MagicMock object. MagicMock cannot be awaited like an actual asynchronous method, causing the test to fail. The test also captured the following:

AssertionError: Control layer failed to handle error correctly.

At first glance, the issue seemed unrelated to initialization, but after further investigation, it became clear that the bot\_control was never properly initialized, which led to MagicMock being improperly used in an await expression.

#### Possible Causes

In this defect, possible cause explained in description along with explanation.

#### Repair Method

The issue was resolved by properly initializing the bot\_control object as an instance of BotControl instead of using a MagicMock placeholder. This ensured that bot\_control.receive\_command could be correctly awaited and executed as intended.

#### Screenshot of Defect

A screenshot of a computer program

Description automatically generated

Figure 23: DEF05.

### Defect 6 - Infinite Loop in Monitoring Loop Due to Missing Iteration Control

Defect ID: DEF06

Date Repaired/Documented: September 2024

#### Description

While developing a test case for monitoring availability in the DiscordBotProject\_CISC699, an infinite loop issue was encountered due to a missing iteration control in the monitoring loop. The run\_monitoring\_loop function was intended to execute a check function a specified number of times based on the iterations parameter. However, the code lacked a line to decrement the iterations counter, causing the loop to continue indefinitely.

This resulted in the loop running infinitely, logging each iteration correctly but never terminating. The loop continued to execute the same check and log the same results repeatedly without ever reaching an exit condition, causing the test to become stuck.

Error Messages:

*Monitoring Iteration: ('Checked availability: Selected or default date is available for booking.', 'Data saved to Excel file at ExportedFiles\\excelFiles\\check\_availability.xlsx.', 'HTML file saved and updated at ExportedFiles\\htmlFiles\\check\_availability.html.')*

*... over and over*

*KeyboardInterrupt: Task was destroyed but it is pending!*

These logs show that the loop executed continuously without stopping, performing the same checks over and over again. The test had to be interrupted manually with a KeyboardInterrupt to stop the infinite loop.

#### Possible Causes

The root cause of the defect was that the iteration decrement step (iterations -= 1) was never implemented in the loop. Without this line, the loop condition iterations > 0 never changed, meaning that the loop had no exit condition and continued running indefinitely.

This defect went unnoticed at first because the loop appeared to function correctly—performing the checks and logging results—but the absence of an iteration decrement meant that the loop would never terminate naturally. This led to an infinite loop that blocked the test from completing.

#### Repair Method

The issue was resolved by adding the iterations -= 1 statement to the loop. This ensured that the number of iterations decreased with each loop execution, and the loop exited once the iteration count reached zero, allowing the test to complete successfully.

#### Screenshot of Defect

A screenshot of a computer screen

Description automatically generated

Figure 24: DEF06.

### Defect 7 - Mismatch in Return Values After Code Updates

Defect ID: DEF07

Date Repaired/Documented: September 2024

#### Description

This defect arose during updates to the stop\_monitoring\_price function, where changes were made to handle return values differently—switching from a string-based return format to an array-based one. Initially, the tests and code compared simple strings, but as the project evolved, arrays and more complex data structures were introduced to represent results.

While this change seemed straightforward, it led to unexpected test failures, especially when the outputs were formatted slightly differently. This issue became particularly difficult to detect and resolve because the test cases were previously passing with string comparisons, and the failure occurred only after the data format was modified. I was only able to understand after putting lots of loggins/output messages.

Error Message:

AssertionError: Control layer did not return the correct results for stopping monitoring.

Test Output:

1. Expected Result:

*Results for price monitoring:Price went up!*

*Price went down!*

*Price monitoring stopped successfully!*

1. Received Result:

*Results for price monitoring:*

*Price went up!*

*Price went down!*

The test failed due to an unexpected formatting discrepancy in the return values. While the actual data was correct, the presence of additional newlines or differences in formatting caused the assertion to fail.

#### Possible Causes

The root cause of this defect was a mismatch between the expected and actual return values in the control layer, specifically when handling the results of stopping price monitoring. The test was written to expect a string-based return format, but after the code was updated to handle more complex data structures (arrays), slight differences in formatting (e.g., extra newlines) caused the test to fail.

This issue was particularly tricky because, on the surface, the data appeared to be correct. However, the subtle changes in formatting between strings and arrays led to assertion failures in the test. The challenge arose from transitioning from one data structure to another, making it harder to identify the exact source of the problem initially.

#### Repair Method

The defect was resolved by updating the test to correctly handle the new data format and by ensuring that the return values were properly formatted when converting from arrays to strings.

#### Screenshot of Defect

A screenshot of a computer

Description automatically generated

Figure 25: DEF07.

### Defect 8 - Email Authentication Failure

Defect ID: DEF08

Date Repaired/Documented: October 2024

#### Description

During the implementation of the email use case in the DiscordBotProject\_CISC699, an authentication error was encountered when trying to send an email. Despite entering the correct email account password in the configuration file, the email sending functionality failed with the following error:

*Failed to send email: (535, b'5.7.8 Username and Password not accepted. For more information, go to\n5.7.8 https://support.google.com/mail/?p=BadCredentials d75a77b69052e-45d92dde23dsm10880611cf.17 - gsmtp')*

This error was misleading at first, as it suggested that the entered username or password was incorrect, even though they had been verified as correct. The failure to authenticate and send the email was due to a specific security requirement by Google: regular account passwords cannot be used for app authentication in third-party applications like the bot. Instead, Google requires an App Password to be generated and used for authentication when accessing Gmail via external applications.

#### Possible Causes

The defect occurred because the bot was attempting to authenticate with a standard account password instead of a Google App Password. Google blocks the use of regular passwords for external apps as a security measure, and without an App Password, the authentication fails with error code 535.

This issue can be confusing to developers, especially when the correct account credentials are entered but are still rejected. Google's security protocols for apps require users to generate a unique App Password from their Google account and use that password in their application’s configuration file.

#### Repair Method

The issue was resolved by generating a Google App Password and using it in the bot’s configuration file instead of the regular account password.

#### Screenshot of Defect

A black screen with many colorful lines

Description automatically generated with medium confidence

Figure 26: DEF08.

### Defect 9 - Element Not Found in Browser Automation

Defect ID: DEF09

Date Repaired/Documented: October 2024

#### Description

During the development of the browser automation functionality in the DiscordBotProject\_CISC699, an issue arose where certain elements on the webpage could not be found, resulting in an ElementNotFound error during test execution. This issue occurred despite the code working previously without errors. After investigation, it was discovered that the website had undergone updates, which caused the DOM structure to change, making the previously located elements unavailable.

This defect wasn't due to an issue in the automation script itself but was triggered by changes made to the website being interacted with. This kind of defect is common in browser automation projects when websites are frequently updated, causing element selectors to break.

Error Message:

*selenium.common.exceptions.NoSuchElementException: Message: Unable to locate element: [element selector here]*

#### Possible Causes

The root cause of this defect was a change in the website's HTML structure, which altered the identifiers or locations of key elements being accessed by the automation script. As a result, the previously correct element selectors became invalid, leading to the NoSuchElementException.

Dynamic changes to the webpage (e.g., updates to CSS classes, IDs, or the structure of the page) can cause automated scripts to fail because the element locators no longer point to the correct part of the page. This defect was not caused by an error in the code but rather by external updates to the target website.

#### Repair Method

The issue was resolved by recapturing the element using updated selectors. This involved revisiting the webpage, identifying the new HTML structure, and adjusting the element locators in the automation script to match the updated structure of the page.

#### Screenshot of Defect

A computer screen shot of text

Description automatically generated

Figure 27: DEF09.

### Defect 10: Test Failures in Website Interaction

Defect ID: DEF10

Date Repaired/Documented: October 7th, 2024

#### Description

The unit tests for the !login command handling in the DiscordBotProject\_CISC699 encountered failures in the test\_website\_interaction.py. The failure was specifically related to an AttributeError stemming from an uninitialized driver attribute within the BrowserEntity class during the test setup. This defect was crucial as it blocked the testing of web interaction functionalities necessary for login automation.

#### Possible Causes

1. The driver attribute within the BrowserEntity was not properly instantiated or accessible at the time of the test.
2. The test attempted to patch an uninitialized or non-existent attribute, leading to AttributeError.
3. Misconfiguration in the test setup where the driver setup was not included in the testing mock environment.

#### Repair Method

The issue was resolved by adjusting the test setup to ensure proper initialization and patching of the driver attribute. The solution involved:

1. Using the pytest.fixture to set up and initialize the BrowserEntity with a mocked driver before the tests.
2. Patching the driver attribute directly within the test setup to ensure it was available and correctly mocked for the duration of the test.
3. Modifying the test to accommodate the lifecycle of the driver attribute, ensuring it was instantiated and accessible when needed.

#### Screenshot of Defect

A black screen with white lines

Description automatically generated

A black background with colorful text

Description automatically generated

Figure 28: DEF10.

### Defect 11 - Control Layer Processing for !close\_browser Command

Defect ID: DEF11

Date Repaired/Documented: October 8th, 2024

#### Description

During unit testing of the !close\_browser command's control layer processing, a failure was encountered where the result was returning a coroutine object instead of the expected string. This indicates an issue with the async handling or the mocked method not being awaited properly, leading to asynchronous operation mismatches.

#### Possible Causes

1. Incorrect AsyncMock Setup: The mock may have been set up to return a coroutine directly, or the async operation wasn't handled properly in the test setup, leading to unresolved coroutine objects in the output.
2. Improper Awaiting of Async Methods: The method from which the mock is supposed to return the result might not be awaited properly, causing the test to capture the coroutine reference instead of the result string.

#### Repair Method

The issue was resolved by adjusting the test setup to ensure proper handling of asynchronous operations:

* Correct AsyncMock Usage: Ensured that AsyncMock is used correctly to mimic asynchronous behavior, returning a resolved value immediately to match expected async function behavior.
* Direct Return Value Setting: Adjusted AsyncMock to directly return the expected string response, bypassing the need for additional asynchronous handling like setting futures.
* Proper Awaiting in Test: Verified that all calls to the mocked async method are awaited properly within the test to ensure the actual string result is captured instead of a coroutine.

#### Screenshot of Defect

A screen shot of a computer

Description automatically generated

Figure 29: DEF11.

### Defect 12 - Browser Closing Test Failure

Defect ID: DEF12

Date Repaired/Documented: October 8th, 2024

#### Description

The test\_browser\_closing unit test failed due to the mock of the quit method not being called as expected. This test aimed to verify that the close\_browser method in BrowserEntity correctly invokes the quit method of the browser's WebDriver when the browser is supposed to close.

#### Possible Causes

1. Incorrect Patching Location: The mock might not have been applied correctly, possibly pointing to an incorrect location or not correctly intercepting the quit method call.
2. Conditional Logic in Method: The close\_browser method might contain conditional logic that prevents the quit method from being called, depending on the state of browser\_open.

#### Repair Method

The defect was corrected by:

* Ensuring Correct Mocking Path: Adjusted the patching to correctly target the quit method where it is actually called in the BrowserEntity.
* Simulating Browser State: Configured the test environment to ensure that the browser is in the correct state (open) before attempting to close it, ensuring conditions for calling quit are met.

#### Screenshot of Defect

A screen shot of a computer program

Description automatically generated

Figure 30: DEF12.

### Defect 13 - Failure in Response Assembly and Output Test

Defect ID: DEF13

Date Repaired/Documented: October 8th, 2024

#### Description

The unit test for the response assembly and output functionality in the PriceControl class failed due to an assertion error. The test was intended to verify that the method PriceControl.receive\_command("get\_price", "https://example.com/product") correctly assembles a response containing the price, Excel file path, and HTML file path. However, the assertion failed because the result did not contain the expected strings as part of a single response string or structure.

#### Possible Causes

1. Incorrect Mock Configuration: The get\_price method was mocked to return a tuple, but the test expected the result to be a single string containing all elements of the tuple.
2. Mismatch Between Expected and Actual Output Format: The format of the output from receive\_command may differ from the expected format, either due to changes in the method implementation or incorrect assumptions in the test about the output structure.
3. Incomplete or Incorrect Assembly of Response Data: There might be an issue in how the receive\_command method assembles or formats the output, leading to missing or incorrectly formatted output components.

#### Repair Method

The test was corrected by adjusting the assertion to specifically unpack and verify each element of the tuple returned by the mocked get\_price method. This change ensures that the test accurately checks each component of the response:

* Unpacking the Result Tuple: The result is unpacked into separate variables (price, excel\_path, html\_path), making it clear and easy to assert each component individually.
* Refined Assertions: Separate assertions for each component ensure that each part of the response is present and correct, providing more detailed checks and clearer debugging information if a test fails.

#### Screenshot of Defect

A black screen with many colorful lines

Description automatically generated

Figure 31: DEF13.

### Defect 14 - Availability Checking Test Failure

Defect ID: DEF14

Date Repaired/Documented: October 2024

#### Description

The unit test for checking availability in the AvailabilityControl failed during execution due to an AssertionError. The test expected the result to match the string "Availability confirmed," but the actual result included additional information about file exports, causing the assertion to fail. The returned output was an extended tuple that included not only the availability status but also file export confirmations, leading to a mismatch with the expected output.

#### Possible Causes

1. The check\_availability function returned more than just the availability confirmation message. It included extra details about saving data to Excel and HTML files, which caused the test assertion to fail.
2. The test was written with the assumption that the returned value would be a simple string, rather than a more complex tuple containing additional metadata.

#### Repair Method

The test was updated to account for the additional information in the result. The following changes were made:

1. Modified the assertion to handle a tuple or extended result. Instead of asserting the exact string, the test now checks if the string "Availability confirmed" is present in the overall result, allowing for the extra metadata to exist without causing a failure.
2. Ensured that the test accommodates the full response from the AvailabilityControl.check\_availability() function by adjusting the assertion to match a substring within the tuple, rather than the entire returned object.

#### Screenshot of Defect

A black screen with colorful text

Description automatically generated

Figure 32: DEF14.

## Defect Summary

Throughout the development and testing of the Discord Bot Automation Assistant, a total of 14 distinct defects were identified and documented. These defects were not isolated to single test cases but often appeared across multiple use cases due to shared structures and functions within the codebase. As a result, each defect was encountered and addressed multiple times throughout the various unit tests.

Our revised testing approach focused on handling asynchronous functions more effectively, ensuring proper initialization of key objects, and mocking external dependencies. This allowed us to systematically identify and fix defects, such as issues with asynchronous handling, improper initialization, and browser automation failures. The same fixes were applied consistently across all affected test cases to ensure comprehensive resolution of the issues.

Given that there are approximately 34 unit tests in total, many defects occurred across these tests due to shared functionality. Therefore, the total number of defect instances addressed is better represented by multiplying the number of unique defects by the number of unit tests:

This figure provides a more accurate reflection of the total effort involved in defect resolution, as many of these defects were encountered and fixed in multiple tests.

### Total Number of Defects

The total number of unique defects documented is 14. However, considering the repeated occurrence of these defects across 34 tests, the total number of defect instances addressed is 476.

### Fixed Defects Percentage

All defects identified during the testing process were successfully resolved, resulting in a fixed defect percentage of 100%. This ensures that the bot is functioning as expected across all test cases.

### Defect Density

Defect density is typically calculated based on the number of lines of code (LOC) in the project, excluding comments and non-executable lines. After excluding these, our project contains approximately 1682 lines of executable code. The defect density is calculated as follows:

This calculation indicates that for every 100 lines of code, approximately 28 defects were encountered and resolved across the various unit tests.

A screenshot of a computer program

Description automatically generated

Figure 33: 100% Fixed Defects.

# CHAPTER FIVE: CONCLUSION

## Project Summary

The **Discord Bot Automation Assistant** is a system designed to streamline the process of checking product availability and prices in real-time. The bot's core functionality focuses on automating price monitoring and availability tracking for users, reducing the need for manual searching and allowing users to receive notifications or save the data for later review. This project was motivated by the need for efficient automation in price comparison and service availability, particularly for consumers in the e-commerce and travel industries.

The technical approach to the project involved using Python as the primary programming language, supported by key frameworks like **Selenium** for web scraping and browser automation, and **Discord.py** for communication between the bot and users through the Discord API. The system integrates various boundary, control, and entity objects, designed following an object-oriented architecture to ensure maintainability, scalability, and efficient handling of asynchronous tasks. The data management strategy relied on file-based storage using formats such as **Excel** and **HTML**, chosen for their flexibility and user-friendliness.

In addition to the functional components, the project includes a robust testing framework built with **pytest** and **mock** objects to validate core functionalities such as command processing, browser interactions, and data export. The unit testing approach ensures that each part of the system operates as expected, even in isolation from external systems.

## Conclusion

### ****Key** **Findings****

Throughout the development and testing phases of the **Discord Bot Automation Assistant**, several key findings emerged. First, the bot's ability to automate product availability and price tracking has demonstrated its value in saving users both time and effort. By offloading repetitive tasks to the bot, users can focus on more critical activities, enhancing both productivity and decision-making in the e-commerce space. The choice of a modular and object-oriented architecture proved beneficial in managing the bot's complex operations, particularly in handling real-time web scraping and interaction with the Discord platform. Furthermore, the testing strategy highlighted the importance of asynchronous handling, where mocking external dependencies like web browsers and APIs ensured a smooth and controlled testing environment.

### ****Suggestions for Future Work****

While the project successfully met its core objectives, there are several opportunities for future improvement and expansion. Enhancing the bot's integration with more diverse platforms, such as adding APIs for direct interaction with popular e-commerce websites (e.g., Amazon, eBay), would provide users with broader monitoring options. Additionally, improving the bot's data analytics capabilities, such as generating more sophisticated reports and integrating predictive analysis, could give users even more actionable insights based on historical trends. Another area of potential development lies in optimizing the bot's performance for real-world use, ensuring that it can handle a larger volume of requests and operate efficiently in high-demand environments.

In conclusion, the **Discord Bot Automation Assistant** successfully addresses the core needs of users seeking automation in price and availability monitoring. The system is well-positioned for future enhancements that will expand its utility and impact, ensuring it remains a valuable tool for users in an increasingly digital and data-driven world.

# LIST OF REFERENCES

1. J. Bram and N. Gorton, "How Is Online Shopping Affecting Retail Employment?," Liberty Street Economics, Oct. 5, 2017. [Online]. Available: https://libertystreeteconomics.newyorkfed.org/2017/10/how-is-online-shopping-affecting-retail-employment/. [Accessed: May 29, 2024].
2. Ofcom, "Online Nation 2021," Online Nation, May 2021. [Online]. Available: <https://www.ofcom.org.uk/__data/assets/pdf_file/0013/220414/online-nation-2021-report.pdf>. [Accessed: May 29, 2024].
3. J. Dube, "Consumer Airfare Index Report - Q2 2022," Hopper, April 2022. [Online]. Available: https://media.hopper.com/articles/consumer-airfare-index-report-q2-2022. [Accessed: May 29, 2024].
4. Flurry Analytics, "Distribution of Time Spent per Shopping Category," December 2011 - December 2012. [Online]. Available: [link to the source if available]. [Accessed: May 29, 2024].
5. U.S. Bureau of Labor Statistics, "Consumer Expenditures in 2016," September 2020. [Online]. Available: https://www.bls.gov/opub/reports/consumer-expenditures/2016/home.htm. [Accessed: May 29, 2024].
6. Statista. "Global E-commerce Adoption and Trends." [Online]. Available: <https://www.statista.com/statistics/251666/number-of-digital-buyers-worldwide/>. [Accessed: June 10, 2024].
7. Research Report on Automation in Booking Services. "Automation Tools and Consumer Preferences." [Online]. Available: [Insert Link]. [Accessed: June 2, 2024].
8. Industry Analysis on Travel Booking Tools. "The Role of Automation in Modern Booking Systems." [Online]. Available: [Insert Link]. [Accessed: June 3, 2024].
9. Ofcom. "Online Nation 2021." [Online]. Available: <https://www.ofcom.org.uk/__data/assets/pdf_file/0013/220414/online-nation-2021-report.pdf>. [Accessed: June 10, 2024].
10. Technical Paper on Web Scraping Technologies. "Enhancing Data Collection through Web Scraping." [Online]. Available: [Insert Link]. [Accessed: June 3, 2024].
11. Journal of Data Science. "Advancements in Data Analysis for Consumer Applications." [Online]. Available: [Insert Link]. [Accessed: June 7, 2024].
12. Consumer Research on Notification Systems. "Impact of Automated Notifications on User Behavior." [Online]. Available: [Insert Link]. [Accessed: June 9, 2024].
13. Future Prospects in E-commerce Tools. "Innovations and Trends in Price Tracking Technologies." [Online]. Available: [Insert Link]. [Accessed: June 9, 2024].
14. McKinsey & Company, "The value of getting personalization right—or wrong—is multiplying," 2021. [Online]. Available: https://www.mckinsey.com/business-functions/marketing-and-sales/our-insights/the-value-of-getting-personalization-right-or-wrong-is-multiplying. [Accessed: May 29, 2024].
15. Statista, "Number of digital buyers worldwide from 2014 to 2021," [Online]. Available: https://www.statista.com/statistics/251666/number-of-digital-buyers-worldwide/. [Accessed: May 29, 2024].
16. "45 Statistics Retail Marketers Need to Know in 2024," Invoca. [Online]. Available: https://www.invoca.com/blog/45-statistics-retail-marketers-need-to-know-in-2024/. [Accessed: June 13, 2024].
17. For Google Flights: 2. "81% of Shoppers Conduct Research Before Purchase," Saleslion. [Online]. Available: <https://saleslion.io/sales-statistics/81-of-shoppers-research-their-product-online-before-purchasing/>. [Accessed: June 13, 2024].
18. Innovations and Trends in Travel Technologies. "Comprehensive Analysis of Travel Planning Tools." [Online]. Available: https://www.cnbc.com/2022/06/09/google-flights-price-guarantee.html. [Accessed: June 10, 2024].
19. Screenshot from Google Flights showing price tracking options. Source: "Google Flights," [Online]. Available: https://www.google.com/flights. [Accessed: June 13, 2024].
20. Advanced E-commerce Analytics. "Insights into Amazon Price Tracking Tools." [Online]. Available: https://www.sellerboard.com/keepa/. [Accessed: June 10, 2024].
21. Screenshot from Keepa showing price history charts. Source: "Keepa - Amazon Price Tracker," [Online]. Available: https://www.keepa.com. [Accessed: June 13, 2024].
22. Microsoft, *Using Excel for data analysis and presentation*. [Online]. Available: <https://docs.microsoft.com/excel>. [Accessed: Oct. 8, 2024].
23. W3C, *HTML5 for interactive web content*. [Online]. Available: <https://www.w3.org/html5/>. [Accessed: Oct. 8, 2024].
24. D. Flanagan, *JavaScript: The Definitive Guide: JSON and File Handling in Modern Applications*, O'Reilly Media, 2021.
25. V. Raj, *Agile Data Management: Leveraging Lightweight Data Solutions*, Tech Publishing, 2021.
26. J. Harrison, "Data Presentation Strategies: Leveraging Excel for Maximum Impact," *Journal of Data Science*, vol. 45, no. 3, pp. 198-213, 2020.
27. J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *IEEE Press*, 2020.
28. Python Software Foundation, *Python Programming Language*. [Online]. Available: <https://www.python.org/>. [Accessed: Oct. 9, 2024].
29. A. Rathore, "Web Scraping Using Selenium: Simplifying Data Extraction for Developers," *Journal of Automation Technology*, vol. 22, no. 3, pp. 120-127, 2023.
30. Discord API Documentation, *discord.py Documentation*, 2024. [Online]. Available: <https://discordpy.readthedocs.io/>. [Accessed: Oct. 9, 2024].
31. Microsoft, *Visual Studio Code Features*. [Online]. Available: <https://code.visualstudio.com/>. [Accessed: Oct. 9, 2024].

*Footnote*

*Code and Text in this documentation has been partially generated with assistance with ChatGPT 4.0.*

# SOURCE CODE