Developing an Intelligent Chatbot: Final Report

Group 30: Christian Cabria(100387527), John Callaghan(100384938) + Maximilian Froggatt(100378090)

June 29, 2025

Abstract

This report presents the development of our intelligent chatbot system for train ticket services. It outlines the project goals, methods used, tools and technologies implemented, and the key results from system testing and evaluation.

1 Introduction

1.1 Background and Motivation

Chatbots are computer programs that can talk with users using natural language. They are being used more in different industries to improve customer service, answer common questions, and make tasks easier for users by helping them. Specifically, for transport services the chatbots can help travel users by giving quick and useful information about tickets, prices, and travel plans. For this coursework we are building a smart chatbot for a train company. The main goal for it is to help customers find the cheapest train tickets for their trips in the UK. It will additionally provide better customer support by handling more advanced questions such as telling users when a delayed train will arrive. This project allows us to use AI methods for a real world scenario and create a service that matches the expectations of public transport users.

1.2 Aim and Objectives of this coursework

The main aim of this coursework is to design and implement an intelligent chatbot capable of handling user queries related to train travel in the UK. The main objectives are:

- To make the chatbot help users in finding the cheapest available train ticket based on the users travel details such as the date, time, origin, destination, and ticket type (single/return).
- To improve the chatbot's functionality by adding a predictive model that estimates new arrival times when trains are delayed, improving the customer service.

1.3 Difficulties and Risks

Developing this system will have several challenges and risks involved such as:

- **Data Accessibility:** Some of the ticket booking websites might not offer APIs or allow scraping which could prevent real time ticket and price retrieval .
- **Natural Language Understanding:** Accurately understanding the user inputs can be difficult especially if they are unclear or it has typos. So effective language processing is needed to manage this efficiently.

- **Prediction Accuracy:** The delay prediction model depends on how good the historical data is which could impact the model's accuracy.
- **Testing and Debugging:** The chatbot includes lots of interacting components, so finding and fixing the errors could be difficult, specifically when errors happen between the NLP, database or the prediction model.
- **System Integration:** Combining the user interface, NLP, knowledge base, and prediction model into a fully functional chatbot may be difficult and will need good teamwork.

1.4 Work Plan

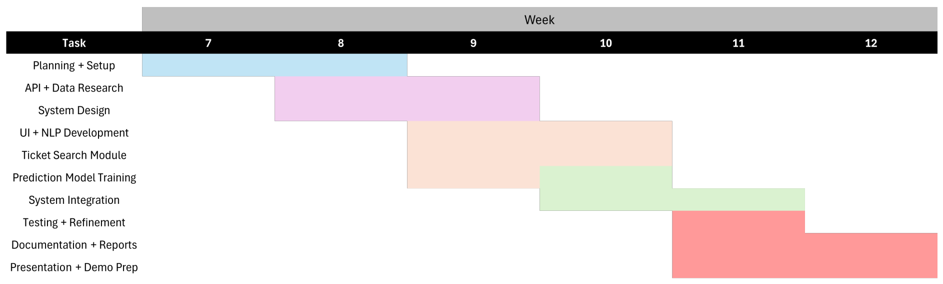


Figure 1: Gantt Chart of project work plan

2 Related Work

In this section we explore existing chatbot systems especially related to ticketing and transportation to understand their features and how they might influence our project design. We look at how these systems handle user interactions, natural language processing (NLP), ticket searches, and delay management.

2.1 SNCF's OUIbot

OUIbot was the official chatbot developed by SNCF which is France's national railway company designed to help passengers book train tickets, planning journeys, and getting travel information by chatting with the user. It was integrated into platforms like the OUI.sncf website and Facebook Messenger, making use of natural language processing (NLP) to guide users' step by step through the ticket booking process, making it easier for users to plan their trips without having to manually fill in a form.

In January 2022, OUI.sncf was rebranded to SNCF Connect. The chatbot functionalities had been improved where SNCF Connect now offers an integrated virtual assistant available on its website, mobile app, WhatsApp, and Messenger. This assistant helps users find routes, get train information, and answers questions using natural language.

While the virtual assistant makes the user interaction more efficient and reduces the need for human based customer support, it depends a lot on SNCF's setup which makes it less flexible. In our project we aim to build a similar user experience but with better control over the chatbot's logic. By using spaCy for natural language understanding and PyKnow for rule-based reasoning we can develop a more adaptable and flexible system. Additionally, our chatbot will include delay prediction through machine learning, which will make it more advanced than SNCF's current assistant.

2.2 Greater Anglia and National Rail

Greater Anglia offers websites and mobile applications that allow users to check train times and purchase tickets. However, their digital platforms does not currently feature a chatbot, meaning that users have to manually navigate and complete a form to find the information they want.

National Rail provides similar services through its website and app. While it also does not feature a chatbot, it offers travel alerts and notifications via WhatsApp and Facebook Messenger, allowing users to receive real-time updates on train services.

Our chatbot aims to improve this experience by offering a chat based interaction with users to gather trip details in a user friendly way. If a user provides only part of the information, the chatbot can ask follow up questions making the process more natural and helpful rather than the user having to fill out a form.

2.3 Deutsche Bahn Navigator App

The Deutsche Bahn Navigator app is a widely used service that provides live updates and predicts delays using real-time data. It uses AI driven forecasting based on historical and real-time data to estimate when trains arrive and depart. However, it does not include a chatbot interface to display this information.

In our project, we aim to improve this by integrating delay predictions directly into our chatbot. Our chatbot interacts with users and provides delay predictions during the conversation. It will use machine learning models like k-nearest neighbours, linear regression, and a basic neural network trained on historical journey data. This approach allows our chatbot to give accurate real time predictions in a more interactive way compared to the what the DB Navigator app currently offers.

2.4 Rasa Framework Use Cases

Rasa is an open source framework for building chatbots where they offer features like understanding user intent, identifying key information, and managing conversations. While it is a powerful tool it needs a lot of setup and training data which can be difficult for smaller projects like ours. To keep things simpler we chose to use spaCy for natural language processing and PyKnow for rule based reasoning. These tools are easier to use but still give us the functions we need for our project.

3 Methods, Tools and Frameworks

3.1 Methods

We are going to use a graphical user interface for our chatbot developed using `Tkinter`, which will allow users to interact with the system through a simple desktop application. The chatbot will guide users through a step by step conversation to collect necessary travel details such as departure and destination stations, travel dates, and ticket types.

For Natural Language Processing (NLP) and understanding, we will use the `spaCy` library to process user input, identify key entities, and extract relevant information. This will be supported by custom rules and regular expression matching to handle specific patterns.

The chatbot's decision making will rely on rule-based reasoning methods, using clearly defined logic to respond to different user intents. These rules will guide the flow of conversation and manage the chatbot's responses based on context.

For the delay prediction we will implement machine learning methods, testing models such as k-Nearest Neighbours (kNN) and **linear regression**. These models will be trained using historical train data to predict expected arrival times following a delay.

3.2 Languages, Packages, Tools

Our project will be developed in Python due to its wide range of libraries and strong support for machine learning and NLP. For NLP, we will use `spaCy` as our main processing library, with additional use of regular expressions for pattern matching. This is an alternative to NLTK, and is selected for its modern, efficient design and ease of use. For the KnowledgeBase and reasoning engine we plan to use `PyKnow`, which allows us to define rules and build a basic expert system framework. Our database will be managed using `SQLite`, which is lightweight and easy to integrate for storing station data, logs, and conversation history. To collect ticket pricing data from real train websites, we will use `Selenium` which allows us to interact with and scrape content from web pages that load dynamically using JavaScript. For machine learning tasks we will use `scikit-learn` for model development and training, alongside `pandas` and `numpy` for data handling and preprocessing.

3.3 Development Framework

The chatbot will be developed as a stand alone desktop application using `Tkinter` for the user interface. This approach offers a lightweight and interactive way for users to engage in dialogue with the chatbot.

Collaboration and progress will be managed using `GitHub`, allowing our group to work concurrently.

4 Design of the Chatbot

4.1 The Architecture of the chatbot

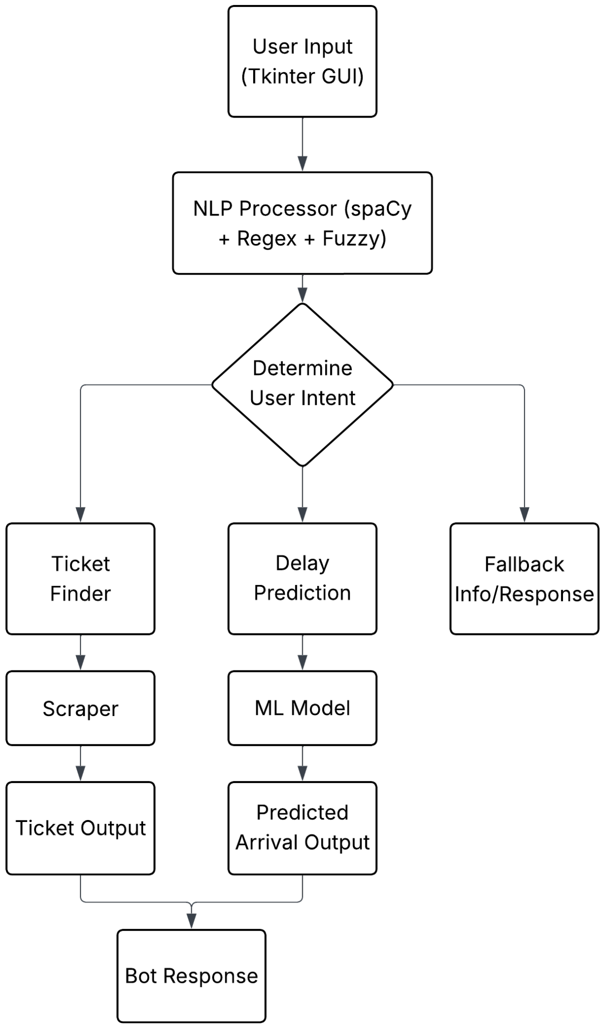


Figure 2: Functional flow diagram of the chatbot system.

4.2 User Interface

The user interface is built using Python's Tkinter library which provides a simple and effective way to create desktop applications. Our chatbot window includes a number of useful features:

- **Conversation Display:** A scrollable text area shows the full chat history between the user and the bot. This area automatically scrolls to the most recent message and is set to read-only so users can't accidentally modify any responses.
- **Status Updates:** A label in the interface displays live updates, such as when a ticket is being searched or when the system is waiting for a response.

- **User Input Area:** Users type their messages into a dedicated input box, with a send button to submit their query. The send button is only enabled when the input is not empty and under 500 characters, preventing invalid or excessively long submissions.
- **Reset Button:** There is also a reset option that clears the current conversation and restarts the chatbot, while still keeping a record of the messages in the database.
- **Clickable Links:** Any URLs detected in the bot's replies are highlighted and clickable, opening in the user's default browser.

4.3 Natural Language Processing

We handle language understanding using a class called *NLPProcessor*, which combines rule-based logic with pattern matching and fuzzy string comparison.

1. **Station Data:** We load a CSV file containing all UK train stations along with alternative names, codes, and identifiers. This helps us recognise a station even if a user types a short name or a common abbreviation.
2. **Entity Detection:** We use the spaCy library for basic processing like tokenisation and part of speech tagging. All known station names are registered as patterns so we can detect them directly in user input.
3. **Intent Recognition:** We apply a simple scoring system based on keywords. For example if the input contains words like "ticket" or "price", we assume the user is searching for a journey. If it includes words like "delay", the chatbot switches to predicting arrival times.
4. **Date and Time Parsing:** Phrases like "today" or "tomorrow" are converted into actual dates. Times are normalised to a standard 24 hour format. For more complex inputs we use the *dateparser* library to handle flexible date expressions.
5. **Station Extraction and Fuzzy Matching:** If we can't find a clear "from...to..." pattern, we then use the detected stations in order. For typo tolerant matching, we use fuzzy search techniques to suggest the closest known station.
6. **Other Info Extraction:** Regular expressions are used to detect trip types (single or return), train IDs, and any mentioned delays.

All this information is gathered in a function that returns a summary of the user's intent and any useful details extracted from their message. Another function checks which details are missing and guides the conversation accordingly.

4.4 Inferring Engine

The chatbot logic is managed by a class that maintains conversation state and makes decisions based on what the user says.

- **Tracking State:** The chatbot keeps a record of what the user has said, including the intent and extracted details like the stations or travel time. It also remembers the last completed journey so the user can easily ask for a return ticket.
- **Conversation Flow:**
  - If the user only says "return", the chatbot reuses the previous route by swapping origin and destination.

  - Each new message is scanned again for dates and times just in case the user adds more information later.
  - Once both stations are detected, the chatbot confirms the route before proceeding.
  - If any details are missing like the time or ticket type, it asks the user specific follow up questions.
  - When all the required details are available, it searches for the cheapest ticket using a background task. If anything goes wrong, the chatbot shows an appropriate error message.
  - After showing the result, it resets its internal state to prepare for the next request.

- **Logging and Debugging:** Every important action from matching stations to sending responses, is logged. This helps us debug any issues and understand how the chatbot behaves in different situations.

4.5 Delay Prediction Models

To provide accurate arrival estimates when trains run late, we developed a regression pipeline using historical service data from 2022-2024 that contained performance information about each service such predicted arrival/delay and actual arrival/delay. Our goal was to predict the end of journey arrival delay (in minutes) at each stop, given:

- Historical baseline: average arrival deviation for each service (RID) over the past year.
- Departure deviations: departure delay at the user's boarding station, and recorded deviations at the first and second stops of the trains route.
- Calendar features: day of month, hour of day, minute of the day, weekday flags, peak/off-peak and weekend indicators.
- Route position: stop number in the sequence, journey progression, and remaining stops.
- Direction: binary flag for direction a train travels during a journey. E.g. Norwich to London Liverpool Street or London Liverpool Street to Norwich

4.5.1 Candidate Algorithms

We compared four standard regressors in scikit-learn:

1. **Linear Regression:** A baseline model to assess linear relationships between features and delay.
2. **k-Nearest Neighbours:** A instance based method that predicts the mean delay of the k most similar historical observations.
3. **Random Forest:** A model made up of multiple full decision trees to capture non-linear interactions and feature importance.
4. **Artificial Neural Network (MLP):** A single layer perceptron with an adaptive learning rate, used to model complex patterns.

4.5.2 Training Procedure

- **Data split:** We reserved 20% of the 2022–24 London–Norwich dataset for testing. The remainder was used for 3 fold cross validation.
- **Pipelines:** Each model sat at the end of a `StandardScaler` → `Estimator` pipeline.

- **Hyperparameter tuning:** We performed a grid search over key parameters:
  - `n_neighbors`={3,5,7} and `weights`={uniform, distance} for kNN.
  - `n_estimators`={100,200}, `max_depth`={None,10,20} for Random Forest.
  - `hidden_layer_sizes`={(50,), (100,)}, `alpha`={1e-4,1e-3}, `learning_rate_init`={1e-4,1e-3}, `max_iter`={200,500} for ANN.
- **Evaluation metrics:** A Negative MSE was used for model selection; final performance was reported as test set MSE, MAE, and R^2.
- **Model persistence:** The top-performing pipeline for each algorithm was saved to disk using `joblib`, allowing it to be loaded later for use in the chatbot.

4.6 Conversation Control

The Conversation Control component controls the the dialogue, maintains context, and decides when to call external services like ticket search and delay prediction. It focuses on:

1. **State Tracking** A central state dictionary stores:
   - Current intent (`find_ticket`, `predict_delay`, etc.)
   - Slot values as they are filled (depart, destination, date, time, trip_type, rid, station, reported_delay)
   - Confirmation flags (to avoid asking the same question twice)
2. **Slot Filling and Prompting**
   - On each user message, the NLP module returns extracted slots and intent.
   - The controller checks for missing required slots and issues a specific friendly prompt (e.g. "What time would you like to travel?").
   - If the user provides partial information the system only asks for the remaining slots.
3. **Confirmation Step** Once all mandatory slots for an intent are collected, the bot formulates a single confirmation question:

   > "Just to confirm: you want a single ticket from Norwich to London on 2025-07-20 at 09:00, correct?"

   A "no" answer clears the relevant slots and restarts slot filling for that part of the query.
4. **Intent Switching** If mid dialogue the user expresses a different intent (e.g. asks about a delay while planning a ticket), the controller:
   - Saves the current partial state.
   - Switches to the new intent's slot requirements.
   - After completing the new intent, it can offer to resume the previous task.
5. **Action Invocation** When slots are confirmed:
   - For ticket search it calls `plan_journey_with_cheapest_ticket()` in a background thread.
   - For delay prediction it calls the `predict_arrival_time()` function immediately.

Results are asynchronously returned to the user with status updates in the GUI (e.g. “Searching…”, “Done”).

6. **Error Handling and Recovery**

   - On unexpected failures (API timeouts, scraping errors), the controller logs the error and informs the user:

     “Sorry, I’m having trouble right now please try again in a moment.”

   - For invalid slot values (unrecognized station, impossible date), the bot provides suggestions or requests clarification.

5 Implementation

5.1 Graphical User Interface (GUI)

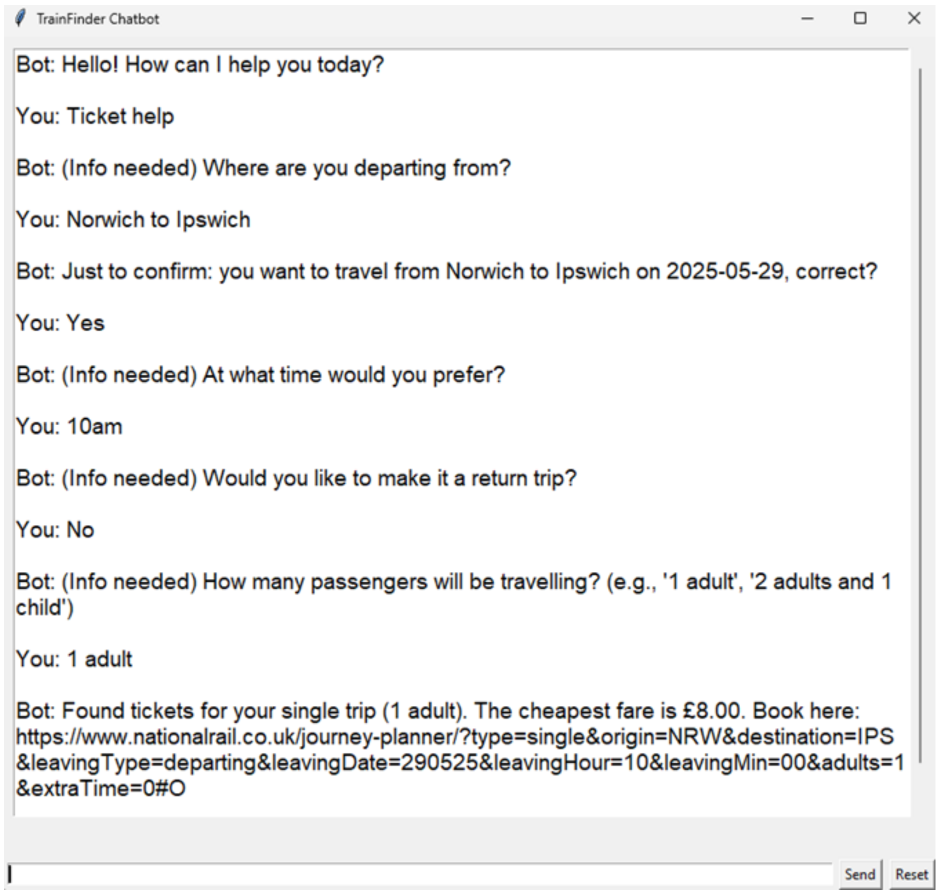
The GUI is built using Tkinter which provides the user a simple and responsive desktop interface. The user messages are submitted using a send button which is dynamically enabled or disabled based on whether the input is empty or exceeds 500 characters. The chat history is displayed in a read-only, auto scrolling text widget.

To maintain responsiveness, message processing is offloaded to a background thread using the threading module. UI updates (e.g., new messages, link insertion) are then safely pushed to the main thread using Tkinter’s after() method.

Handling chatbot responses asynchronously with threading and after():

```python
def _process_user_message(user_message):
    try:
        response = bot.respond(user_message)
    except Exception:
        logger.exception("Error in bot.respond")
        response = "Sorry, something went wrong. Please try again."
    root.after(0, lambda: _display_bot_response(response))
```

URLs detected in bot responses are made clickable using regex and tag bindings. The Reset button clears the conversation and restarts the chatbot logic, while keeping the session data stored in the database.



```
Bot: Hello! How can I help you today?

You: Ticket help

Bot: (Info needed) Where are you departing from?

You: Norwich to Ipswich

Bot: Just to confirm: you want to travel from Norwich to Ipswich on 2025-05-29, correct?

You: Yes

Bot: (Info needed) At what time would you prefer?

You: 10am

Bot: (Info needed) Would you like to make it a return trip?

You: No

Bot: (Info needed) How many passengers will be travelling? (e.g., '1 adult', '2 adults and 1
child')

You: 1 adult

Bot: Found tickets for your single trip (1 adult). The cheapest fare is £8.00. Book here:
https://www.nationalrail.co.uk/journey-planner/?type=single&origin=NRW&destination=IPS
&leavingType=departing&leavingDate=290525&leavingHour=10&leavingMin=00&adults=1
&extraTime=0#O
```

Figure 3: Tkinter GUI during a conversation with URL link.

5.2 Natural Language Processing

The NLP logic is implemented in `nlp_module.py` through the NLPProcessor class. This class combines rule-based parsing with spaCy's PhraseMatcher to recognize station names and extract other relevant entities.

Key functions include:

- **Station Recognition:** A CSV of UK station names and aliases is loaded and indexed by the CRS code. The PhraseMatcher identifies these station names in input text, supporting a wide range of formats and spelling variants.
- **Intent Detection:** Intents are classified using a keyword scoring approach. The input is scanned for terms like "ticket" or "price" (for journey planning) and "delay" or "arrival" (for delay prediction). A fallback "unsupported" intent is returned if confidence is low.

  Intent classification using keyword scoring:

  ```
  def predict_intent(self, text: str) -> tuple[str,float]:
      txt = text.lower()
      scores = { intent: sum(1 for kw in kws if kw in txt) for intent,kws in self.
          intent_keywords.items() }
      best_intent, best_score = max(scores.items(), key=lambda x: x[1])
      total = sum(len(kws) for kws in self.intent_keywords.values())
      confidence = best_score/total if total else 0.0
  ```

```
        if best_score == 0:
            return "unsupported", confidence
        return best_intent, confidence
```

- **Date and Time Parsing:** Uses both regex and the dateparser library to extract temporal expressions, converting natural phrases like “tomorrow at 7 pm” into datetime objects. Past dates are adjusted to the future to avoid invalid results.

  Date and time extraction using literals, regex, and dateparser:

```
slots = {}
now = datetime.now()
low_text = text.lower()

# Recognize "today" or "tonight"
if "today" in low_text or "tonight" in low_text:
    slots['date'] = now.date()

# Recognize "tomorrow"
elif "tomorrow" in low_text:
    slots['date'] = (now + timedelta(days=1)).date()

# Extract time (e.g., 7:30pm)
t_re = re.compile(r'\b(\d{1,2})(?::(\d{2}))?\s*(am|pm)?\b', re.IGNORECASE)
t_match = t_re.search(text)
if t_match:
    h = int(t_match.group(1))
    m = int(t_match.group(2) or 0)
    ampm = t_match.group(3)
    if ampm:
        if ampm.lower() == 'pm' and h < 12: h += 12
        if ampm.lower() == 'am' and h == 12: h = 0
    slots['time'] = dptime(hour=h, minute=m)
    text = text.replace(t_match.group(0), '') # Remove matched time

# Extract date using dateparser
date_re = re.compile(
    r'\b(\d{1,2}(?:st|nd|rd|th)?(?: of)? [A-Za-z]+(?: \d{4})?|\d{4}-\d{2}-\d{2})\b',
    re.IGNORECASE
)
d_match = date_re.search(text)
if d_match:
    dt = dateparser.parse(d_match.group(1), settings={
        "PREFER_DATES_FROM": "future",
        "DATE_ORDER": "DMY",
        "RELATIVE_BASE": now
    })
    if dt:
        parsed_date = dt.date()
        if parsed_date < now.date():
            parsed_date = parsed_date.replace(year=parsed_date.year + 1)
        slots['date'] = parsed_date
        text = text.replace(d_match.group(0), '')
```

- **Slot Filling and Correction:** If no “from. . . to. . . ” structure is found, the processor uses matcher results in sequence. It applies `difflib.get_close_matches()` to correct minor typos or incomplete station names.

- **Additional Parsing:** Regular expressions detect other contextual slots such as "single"/"return" trip type, specific train numbers, or delay durations (e.g., "15 minutes late").

The NLP module returns a dictionary of slot values and the detected intent, along with a confidence score. Another function identifies which required slots are still missing so that the chatbot can guide the conversation accordingly.

5.3 Reasoning Engine and Dialogue Logic

The dialogue control is implemented in the Chatbot class in `chatbot_logic.py`, acting as a rule-based engine to manage conversation state and handle actions. The chatbot maintains a state dictionary containing the current intent, all recognized slot values, and a confirmation flag.

Main Features:

- **Slot Confirmation:** Once origin and destination are detected, the chatbot explicitly confirms them with the user. If the user rejects the confirmation the relevant slots are cleared.

  Dynamic confirmation message construction based on filled slots:

```
if not self.confirm_done:
    if "time" in s and "date" not in s:
        s["date"] = date.today().isoformat()
    if "departure" in s and "destination" in s:
        dep = code_to_name[s["departure"]]
        dst = code_to_name[s["destination"]]
        parts = [f"from {dep} to {dst}"]
        if "date" in s:
            parts.append(f"on {s['date']}")
        if "time" in s:
            parts.append(f"at {s['time']}")
        if s.get("trip_type") == "return":
            if "return_date" in s:
                parts.append(f"returning on {s['return_date']}")
            if "return_time" in s:
                parts.append(f"at {s['return_time']}")
        if "trip_type" in s:
            parts.append(f"as a {s['trip_type']} trip")
        self.confirm_done = True
        self.logger.info("Asking detailed confirmation")
        return (
            "Just to confirm: you want to travel "
            + " ".join(parts) + ", correct?"
        )
    if "departure" not in s:
        return "(Info needed) Where are you departing from?"
    if "destination" not in s:
        return "(Info needed) Where are you going to?"
```

- **Dialogue Flow:** The chatbot dynamically prompts the user for missing slots (e.g., travel time, trip type). This is done through the missing_slots() function, which maps slot requirements to tailored follow-up questions.

- **Asynchronous Execution:** Once all slots are filled, the chatbot calls the find_cheapest_ticket() function in a separate thread using ThreadPoolExecutor. Results are returned asynchronously and pushed to the GUI with appropriate status updates.
- **Return Shortcut:** If a completed journey exists, users can simply type "return," and the chatbot swaps the previous origin/destination to initiate a return search using cached slot data.
- **Robust Logging:** Key events such as slot updates, confirmations, and failures are printed for debug visibility. This was useful during testing to track state transitions and conversation logic.

This setup makes the conversation feel smooth and responsive. It also connects well with both the NLP system and the database, helping the chatbot find train tickets accurately and in real time.

5.4 Cheapest Ticket Retrieval System

For retrieving the cheapest train tickets, we developed a scraping and data validation pipeline that integrates live web scraping from National Rail with historical journey validation using the National Rail Darwin dataset. This pipeline is responsible for supplying the chatbot with real-time ticket prices and validated journey data, making it essential to the system's overall functionality.

5.5 Data Collection Pipeline Overview

The main function we implemented is `plan_journey_with_cheapest_ticket()`, which carries out the following main steps:

1. Stores the user's journey query in an SQLite database.
2. Retrieves and parses Darwin timetable data to validate whether valid journeys exist for the selected origin, destination, and time.
3. Launches a Selenium WebDriver instance to automate the National Rail journey planner.
4. Extracts visible ticket prices from the results page and identifies the cheapest one.
5. Stores the result, including the booking link, in the database.

This approach ensures that even if no valid journey is found in the Darwin data, the user still receives ticket results through scraping, with a warning included in the chatbot's output message.

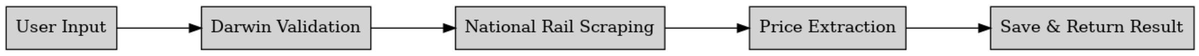


Figure 4: Flow diagram showing the steps from user input to final ticket result.

5.6 Darwin Timetable Validation

To validate journeys, we used the Darwin timetable feed provided by National Rail, stored as compressed `.xml.gz` files on an AWS S3 bucket. We implemented a parser in `parse_journey_file()` that:

- Loads the latest file version using `boto3`.
- Maps station CRS codes (e.g., "NRW") to TIPLOC codes using a custom CSV lookup (`station_lookup.py`).
- Parses each `<Journey>` node to extract intermediate stops and departure/arrival times.

- Filters journeys that match the user's route and time constraints.

Using the Darwin timetable data allows the chatbot to validate that a requested journey is officially scheduled and feasible before attempting to scrape ticket prices. This improves the accuracy and reliability of journey results while reducing unnecessary scraping.

Code Snippet: TIPLOC Matching and Journey Filtering

```python
origin_tiploc = get_tiploc_from_crs(origin_crs)
dest_tiploc = get_tiploc_from_crs(dest_crs)
if not origin_tiploc or not dest_tiploc:
    print(f"Could not find TIPLOCs for {origin_crs} or {dest_crs}")
    return []
for journey in journeys:
    stops = []
    for tag in ['OR', 'IP', 'PP', 'DT']:
        for loc in journey.findall(f'tt:{tag}', NS):
            crs = loc.attrib.get('tpl')
            dep = loc.attrib.get('ptd')
            stops.append({'crs': crs, 'dep': dep})

    crs_list = [s['crs'] for s in stops]

    if origin_tiploc in crs_list:
        o_idx = crs_list.index(origin_tiploc)
        d_idx = crs_list.index(dest_tiploc) if dest_tiploc in crs_list else -1
        departure_time = stops[o_idx]['dep']

        if d_idx > o_idx and departure_time and departure_time >= latest_dep_time:

            arrival_time = journey.find(f'tt:DT', NS).attrib.get('pta') or journey.find(f'tt:DT'
                , NS).attrib.get('arr')
            station_names = [get_tiploc_from_crs(t) or t for t in crs_list]

            dep_arr_key = (departure_time, arrival_time)
            if dep_arr_key not in seen_times:
                seen_times.add(dep_arr_key)
                matched_journeys.append({
                    'origin': origin_crs,
                    'destination': dest_crs,
                    'departure_time': departure_time,
                    'arrival_time': arrival_time,
                    'tiploc_route': crs_list,
                    'station_route': station_names,
                    'matched': True
                })
```

5.7 Web Scraping with Selenium (National Rail)

For finding the live ticket prices we performed web scraping on the National Rail website. We initially attempted to use Trainline, but due to restrictions and dynamic content loading we were unable to reliably extract ticket prices. Because of this, we decided to switch to using National Rail, which was more suitable due to its lightweight HTML structure and minimal JavaScript complexity.

Using Selenium, the `plan_journey_with_cheapest_ticket()` script:

- Opens the National Rail site and populates the form with origin, destination, departure/return dates and times, and passenger details.
- Submits the form and waits until ticket prices are visible.
- Extracts all prices containing the pound symbol (£).
- Filters invalid entries and selects the cheapest price.

The extracted ticket price and booking URL are saved to the database for use in the chatbot's response.

Code Snippet: Form automation and price extraction using Selenium

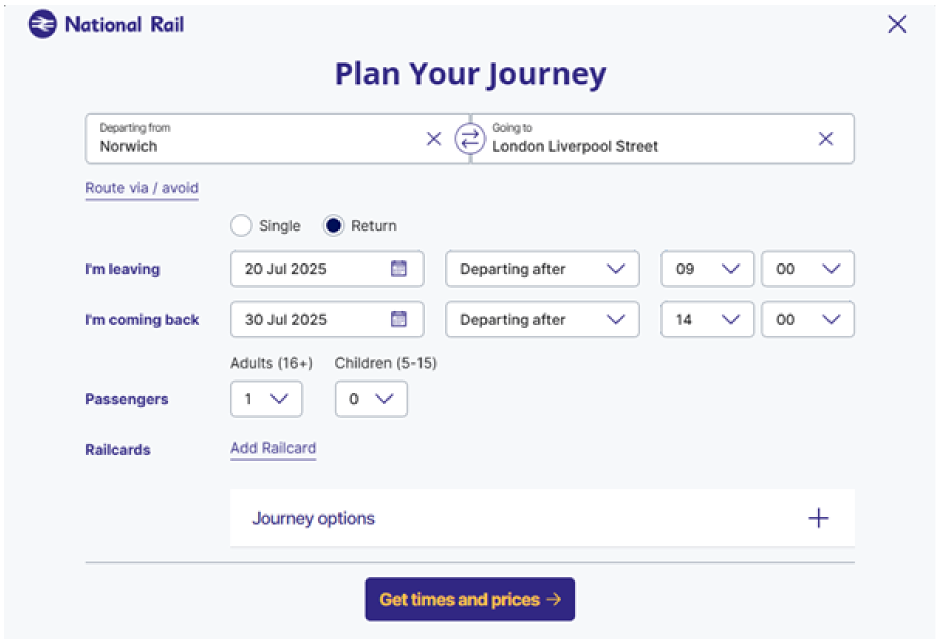
```
# Fill in origin and destination
wait.until(EC.presence_of_element_located((By.ID, "jp-origin"))).send_keys(origin, Keys.TAB)
wait.until(EC.presence_of_element_located((By.ID, "jp-destination"))).send_keys(destination,
    Keys.TAB)

# Select date and time
driver.find_element(By.ID, "leaving-date").send_keys(Keys.CONTROL, "a", Keys.BACKSPACE)
driver.find_element(By.ID, "leaving-date").send_keys(dep_date, Keys.TAB)
Select(driver.find_element(By.ID, "leaving-hour")).select_by_value(dep_hour)
Select(driver.find_element(By.ID, "leaving-min")).select_by_value(dep_min)

# Extract all visible prices
price_elements = driver.find_elements(By.XPATH, "//*[contains(text(), 'GBP')]")

# GBP symbol causeS LaTeX encoding issues, so we use a placeholder here 'GBP'
prices = []
for el in price_elements:
    text = el.text.strip()
    if "GBP" in text:
        try:
            price = float(text.split("GBP")[1].split()[0].replace(",", ""))
            if price >= 1.00:
                prices.append(price)
        except:
            continue

cheapest_price = min(prices) if prices else None
```

Automated Form Submission Example

When testing with:

```
if __name__ == "__main__":
    plan_journey_with_cheapest_ticket(
        origin="NRW",
        destination="LST",
        dep_date="20 Jul 2025",
        dep_hour="09",
        dep_min="00",
        return_date="30 Jul 2025",
        return_hour="14",
        return_min="00",
        adults="1",
        children="0"
    )
```

This is what is automatically filled in the National Rail form by the Selenium script:



Departing from: Norwich
Going to: London Liverpool Street

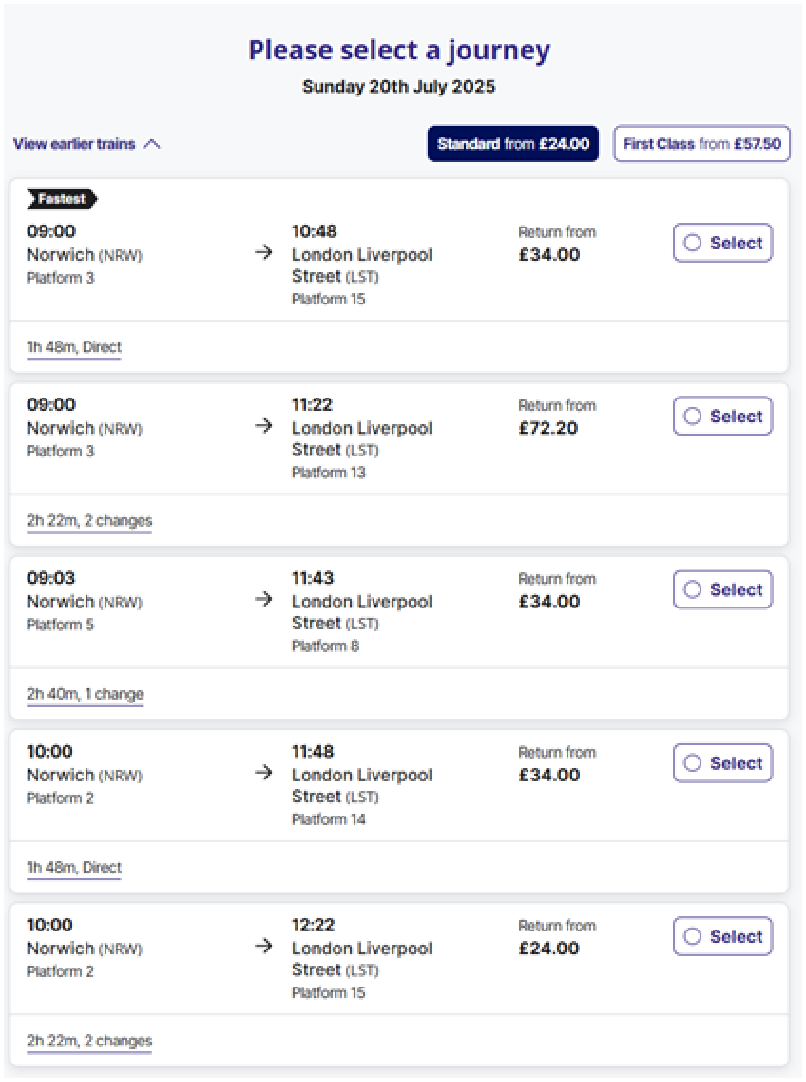
Single [✓ Return]

I'm leaving	20 Jul 2025	Departing after	09	00
I'm coming back	30 Jul 2025	Departing after	14	00

Passengers: Adults (16+) 1, Children (5-15) 0

Figure 5: National Rail journey planner auto-filled by Selenium using test inputs.

Then it will automatically click 'Get times and prices' and bring us to this page:



Please select a journey

Sunday 20th July 2025

View earlier trains | Standard from £24.00 | First Class from £57.50

Fastest
09:00 Norwich (NRW) Platform 3 → 10:48 London Liverpool Street (LST) Platform 15 — Return from £34.00 — Select
1h 48m, Direct

09:00 Norwich (NRW) Platform 3 → 11:22 London Liverpool Street (LST) Platform 13 — Return from £72.20 — Select
2h 22m, 2 changes

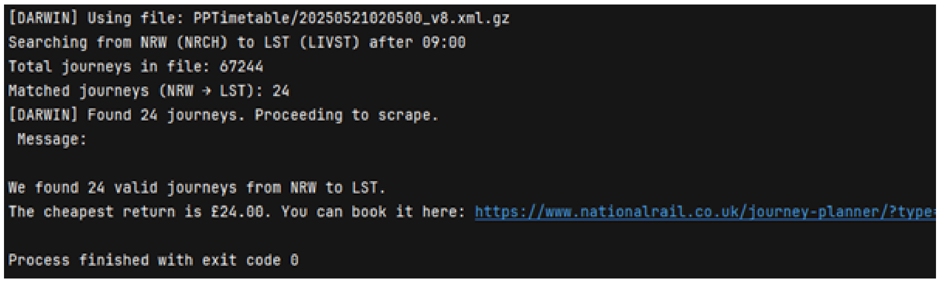
09:03 Norwich (NRW) Platform 5 → 11:43 London Liverpool Street (LST) Platform 8 — Return from £34.00 — Select
2h 40m, 1 change

10:00 Norwich (NRW) Platform 2 → 11:48 London Liverpool Street (LST) Platform 14 — Return from £34.00 — Select
1h 48m, Direct

10:00 Norwich (NRW) Platform 2 → 12:22 London Liverpool Street (LST) Platform 15 — Return from £24.00 — Select
2h 22m, 2 changes

Figure 6: Ticket results page displaying available train options and prices.

The chatbot then prints a formatted summary message to the console:



```
[DARWIN] Using file: PPTimetable/20250521020500_v8.xml.gz
Searching from NRW (NRCH) to LST (LIVST) after 09:00
Total journeys in file: 67244
Matched journeys (NRW → LST): 24
[DARWIN] Found 24 journeys. Proceeding to scrape.
 Message:

We found 24 valid journeys from NRW to LST.
The cheapest return is £24.00. You can book it here: https://www.nationalrail.co.uk/journey-planner/?type=

Process finished with exit code 0
```

Figure 7: Output console

5.8 SQLite Database Storage

We structured the chatbot's backend data using these SQLite tables:

- `journey_queries`: Stores the full user input from each chatbot query, including origin, destination, departure/return dates and times, number of adults and children, and whether the journey is a return. Each query is assigned a unique `id`, which serves as a foreign key in the `darwin_journeys` table.
- `darwin_journeys`: Logs individual train journeys matched using Darwin timetable data. Each entry links back to a user query via `journey_query_id`, and includes the origin, destination, departure time, TIPLOC route, and a boolean indicating whether it matched the criteria.
- `ticket_results`: Records the scraped ticket price and booking URL for a completed journey. This includes the origin, destination, journey type (single or return), departure and return dates, the cheapest price found, and a timestamp.
- `stations`: Populated from the CSV this table contains reference data for UK stations including CRS codes (`alpha3`), TIPLOC codes, station names, and aliases. It supports CRS to TIPLOC conversion during Darwin parsing.

All tables include `created_at` to track when records were added. The `journey_queries` table acts as the central link between user searches, matched Darwin journeys, and ticket price results.

This structure makes it easy to trace chatbot responses and supports future analysis, such as tracking price trends, journey frequency, or model accuracy.

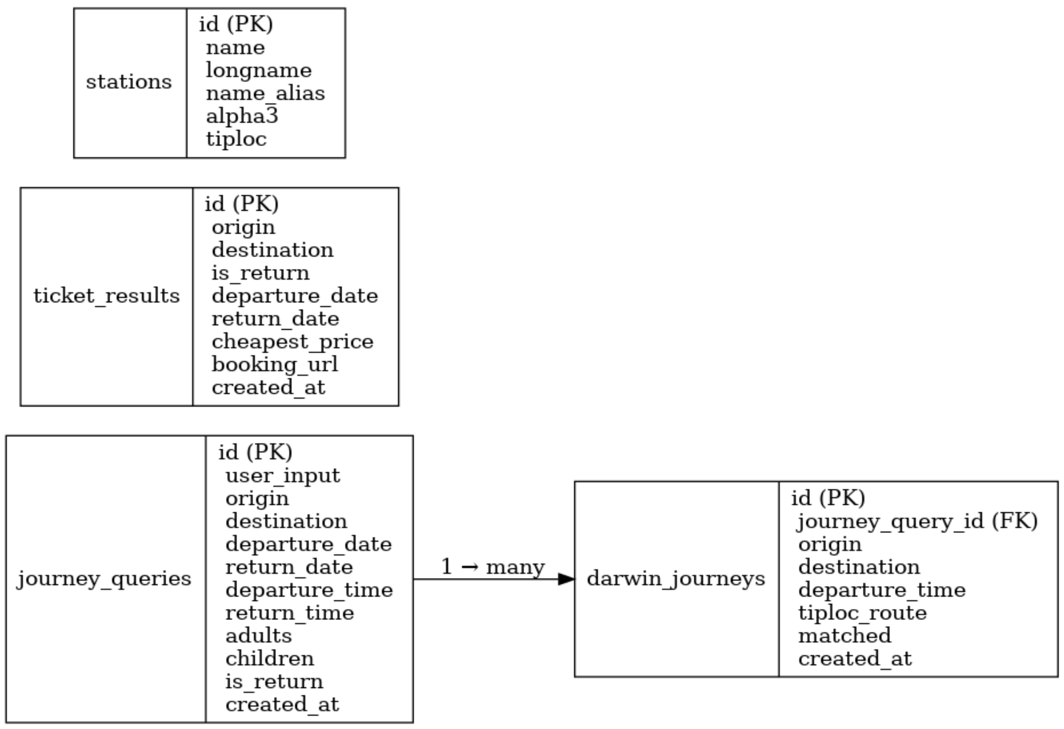


Figure 8: Entity-Relationship diagram showing key tables and relationships.

5.9 Model implementation

Overview Task 2 required the integration of a predictive model that assists users predict the arrival time of a delayed train. This was achieved using a machine learning model called `RandomForestRegressor` being implemented into the chatbot logic, to enable arrival times based on user inputs and pre-processed schedule data. For this task we would implement the use of previous historical data in order for the chatbot to make predictions. We used this instead of implementing a real time API that gives current data due to time constraints. The chatbot is still able to make predictions based on this.

Model training and storage

- **Model used:** `RandomForestRegressor`
- **Trained_Features:** The model uses features like `rid`, `location`, `service_historical_average_delay`, `departure_deviation`, `first_stop_deviation`, `second_stop_deviation`, day of month, hour of day, `is_peak`, `is_weekend`, `stop_number`, `stop_fraction`, `stops_remaining` and direction of train.
- **Data Sources:**
  - `Schedule.csv` and `rid_hist_avg.csv` to provide train performance and schedule data
- **Stored file:** The trained model is stored as `Random_forest_best_model.joblib` for easy reusability

Code Structure

1. **delay_predictor.py**
   Contains the function `predict_arrival_time(query)` that loads the saved model and makes a prediction using the provided query data. Ensures input data is transformed into the appropriate format expected by the model. Returns a user-friendly string indicating the predicted arrival time.
2. **get_prediction_schedule.py**
   Provides the function `get_prediction_schedule(...)` which extracts timetable information from csv schedule files based on user inputs like `rid`, current station, and destination. Returns data needed to complete the feature set for the prediction.
3. **chatbot_logic.py**
   Integrates the above two modules into the chatbot's dialogue flow.
   - Handles the `predict_delay` intent using the `_handle_predict_delay()` method.
   - This method collects four essential slot values: `rid`, `station`, `reported_delay`, and `destination`.
   - It retrieves the schedule from `get_prediction_schedule()` and passes this along with user input to `predict_arrival_time()`.
   - Returns the model's predicted arrival time as a conversational response.

   Example user interaction:

   User: 1042253, IPS, 10, LST
   Bot: Based on this delay and schedule, your train is expected to arrive at 14:42.

The chatbot uses natural language processing to extract relevant entities (slots) from user messages. If any fields are missing from input the chat asks the user to provide the necessary fields again Once all information is gathered, the chatbot calls the functions to do the prediction.

Implements safeguards for date parsing and schedule availability. Exceptions during prediction are caught and logged, and the bot responds with a graceful fallback message:

"Sorry, I couldn't calculate the arrival time right now."

Resets internal state after prediction to prepare for a new conversation.

6 Testing

6.1 Unit Testing

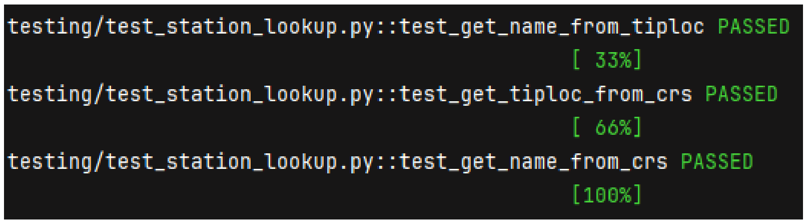
To ensure the reliability of individual components we performed unit testing using the `pytest` framework. Each module was tested in seperately with controlled inputs to verify correct behavior.

6.1.1 Station Lookup Functions

This module provides functions to map CRS codes to TIPLOCs and station names based on a CSV dataset. The following functions were tested:

- `get_name_from_tiploc(tiploc_code)`
- `get_tiploc_from_crs(crs_code)`
- `get_name_from_crs(crs_code)`

We used mocked data representing a subset of known station records (e.g., Norwich, Cambridge, Liverpool Street) to verify the logic. The tests confirmed correct mapping and proper handling of unknown values.



```
testing/test_station_lookup.py::test_get_name_from_tiploc PASSED
                                                   [ 33%]
testing/test_station_lookup.py::test_get_tiploc_from_crs PASSED
                                                   [ 66%]
testing/test_station_lookup.py::test_get_name_from_crs PASSED
                                                   [100%]
```

Figure 9: Unit tests for stationlookup.py

These results demonstrate that the station lookup utility is functioning correctly and robustly handles both valid and invalid inputs.

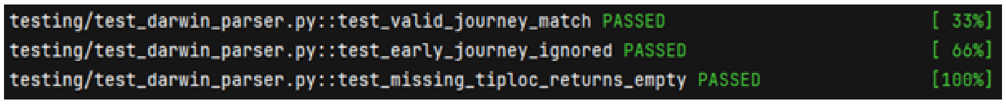
6.1.2 Darwin Timetable Parser

We made unit testing for the `parse_journey_file()` function in `darwin.py`, which processes compressed Darwin XML timetable files and extracts valid train journeys based on origin, destination, and time constraints.

The tests were broken down into smaller cases:

- **test_valid_journey_match:** Confirms that a journey from Norwich (NRWCH) to London Liverpool Street (LIVST) with a valid departure time is correctly matched and returned.
- **test_early_journey_ignored:** Checks that trips leaving before the user's chosen time are properly left out.
- **test_missing_tiploc_returns_empty:** Ensures the function handles unknown or missing station codes appropriately by returning an empty result.

These tests were performed using the `pytest` framework along with `unittest.mock` to simulate AWS S3 responses and override station lookup functions. The Darwin XML data was also mocked and compressed in memory to reflect the real .xml.gz file structure.



```
testing/test_darwin_parser.py::test_valid_journey_match PASSED                [ 33%]
testing/test_darwin_parser.py::test_early_journey_ignored PASSED              [ 66%]
testing/test_darwin_parser.py::test_missing_tiploc_returns_empty PASSED       [100%]
```

Figure 10: Unit tests for darwin.py

This confirms that the main logic for filtering and matching journeys in our Darwin timetable parser works correctly.

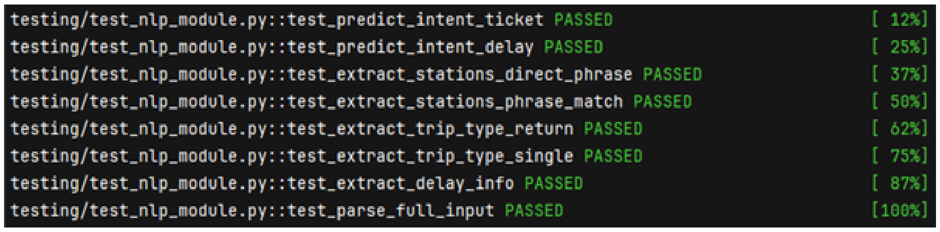
6.1.3 NLP Module

We created unit tests for the `NLPProcessor` class in `nlp_module.py`, which handles user intent detection and slot extraction.

The tests covered:

- `predict_intent()` – Classifies input as `find_ticket` or `predict_delay`.
- `extract_stations()` – Finds departure and destination stations.
- `extract_trip_type()` – Detects whether the user wants a `return` or `single` ticket.
- `extract_train_info()` – Extracts train ID and delay minutes.
- `parse()` – Runs full message parsing.

We used mocked station data and tested typical user inputs. The tests showed the NLP works well for common travel queries, even with different phrasing.



```
testing/test_nlp_module.py::test_predict_intent_ticket PASSED                 [ 12%]
testing/test_nlp_module.py::test_predict_intent_delay PASSED                  [ 25%]
testing/test_nlp_module.py::test_extract_stations_direct_phrase PASSED        [ 37%]
testing/test_nlp_module.py::test_extract_stations_phrase_match PASSED         [ 50%]
testing/test_nlp_module.py::test_extract_trip_type_return PASSED              [ 62%]
testing/test_nlp_module.py::test_extract_trip_type_single PASSED              [ 75%]
testing/test_nlp_module.py::test_extract_delay_info PASSED                    [ 87%]
testing/test_nlp_module.py::test_parse_full_input PASSED                      [100%]
```

Figure 11: NLP module unit test results

The tests showed that the NLP pipeline works well with common user inputs and can handle small changes in how things are phrased. Confidence thresholds were tuned based on the number of intent keywords, making the intent detection more accurate and realistic.

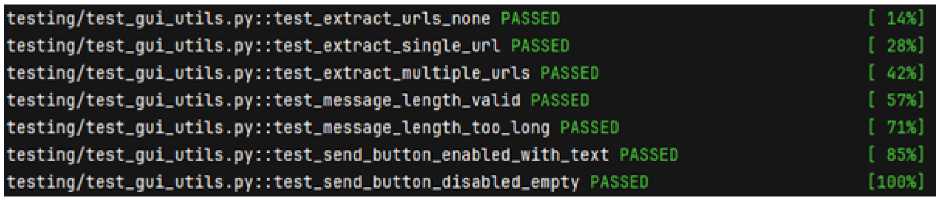
6.1.4 GUI Utility Function

To test important parts of our graphical interface, we moved reusable helper functions into a separate `gui_utils.py` module. This made it possible to test input checks and link detection without relying on the Tkinter interface.

We wrote six unit tests which covered:

- `extract_urls_from_text()` – Validates link detection in chatbot messages.

- `is_valid_message_length()` – Checks whether user messages stay within the 500 character limit.
- `should_enable_send_button()` – Determines if the send button should activate based on input.



```
testing/test_gui_utils.py::test_extract_urls_none PASSED                    [ 14%]
testing/test_gui_utils.py::test_extract_single_url PASSED                   [ 28%]
testing/test_gui_utils.py::test_extract_multiple_urls PASSED                [ 42%]
testing/test_gui_utils.py::test_message_length_valid PASSED                 [ 57%]
testing/test_gui_utils.py::test_message_length_too_long PASSED              [ 71%]
testing/test_gui_utils.py::test_send_button_enabled_with_text PASSED        [ 85%]
testing/test_gui_utils.py::test_send_button_disabled_empty PASSED           [100%]
```

Figure 12: Unit testing GUI.

These tests confirm that core interface features behave correctly and improve the chatbot's usability and robustness.

6.2 Integration Testing

We used Integration testing to verify that the main components of the chatbot system work together under real conditions. We used live timetable data, web scraping, and the actual SQLite database to test full workflows.

6.2.1 Ticket Search Workflow

We tested the entire journey from user input to retrieving the cheapest ticket using: `NLPProcessor`, `DialogManager`, `DarwinParser`, `SeleniumScraper`, and `SQLite`. A sample input like:

”I want a single from Norwich to London tomorrow at 09:00”

which led to successful intent detection, timetable validation, scraping of live prices, and database logging. The chatbot responded with the cheapest ticket and a booking link.

6.2.2 Delay Prediction Workflow

We tested delay prediction using the trained `RandomForest` model and historic timetable records. The chatbot used actual data to estimate the arrival time based on the current delay and responded accordingly.

6.2.3 Session Recovery

We confirmed that ticket queries persist between sessions. After restarting the chatbot the users could retrieve their last search using a follow up query.

Console Output Example

```
[DARWIN] Using file: PPTimetable/20250522020500_v8.xml.gz
Searching from NRW (NRCH) to IPS (IPSWICH) after 09:00
Total journeys in file: 67259
Matched journeys (NRW → IPS): 25
[DARWIN] Found 25 journeys. Proceeding to scrape.
```

Summary

Running integration tests with real data and components showed that the system works as expected in real-world conditions and handles full end to end scenarios accurately.

6.3 System Testing

System testing was used to check if the chatbot system in its complete deployed form. We tested interactions through the Tkinter GUI, ensuring that NLP, backend logic, prediction models, and database operations all functioned correctly with live data.

6.3.1 End-to-End Ticket Search

Users entered full travel queries in the GUI, such as:

> ”Return from Norwich to Ipswich on July 20 at 14:00, returning July 30 at 10:00.”

The chatbot processed the message, retrieved valid journeys, scraped ticket prices, and responded with the cheapest fare and a clickable booking link. Data was logged in the database under `ticket_results`.

6.3.2 Delay Prediction

We tested delay predictions by entering inputs of the trains service id/RID, station user is at currently, how late the train is and their destination:

> “20020345, IPS, 10, LST ”

The chatbot extracted delay info from historic train data, used the trained model to estimate arrival time, and returned a response such as:

> “Your train is running 5 minutes late. Expected arrival: 11:02.”

6.3.3 Error Handling and Recovery

Invalid inputs like unknown station names were tested. For example:

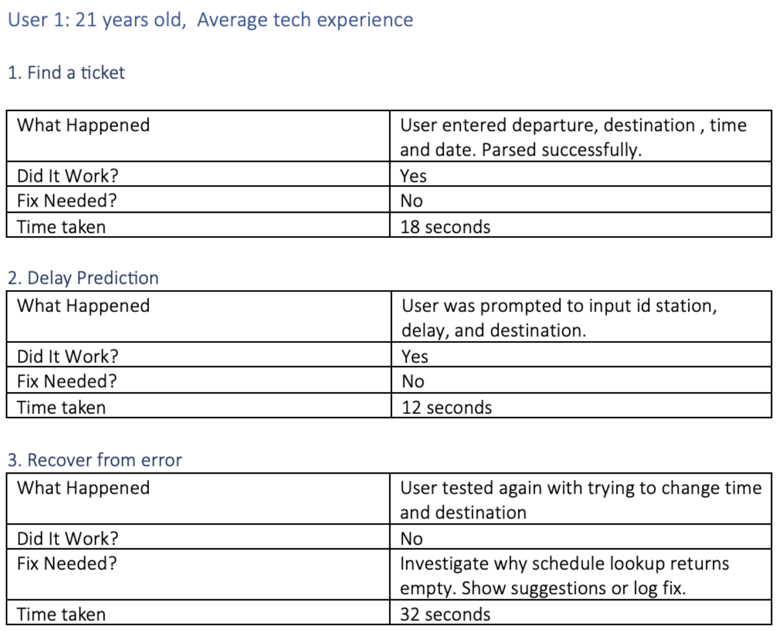
> ”I’m at Queens square station.”

The chatbot was able to handle certain misspellings using the fuzzy library however if station name could not match, The chatbot would prompt the user for another input and say it was unclear.

6.4 Usability Testing

To evaluate usability, we tested the chatbot with 3 participants to complete three different tasks:

- Find a ticket (e.g. ”Single from Norwich to London tomorrow at 09:00”)
- Get a delay prediction
- Recover from an error



User 1: 21 years old, Average tech experience

1. Find a ticket

What Happened	User entered departure, destination , time and date. Parsed successfully.
Did It Work?	Yes
Fix Needed?	No
Time taken	18 seconds

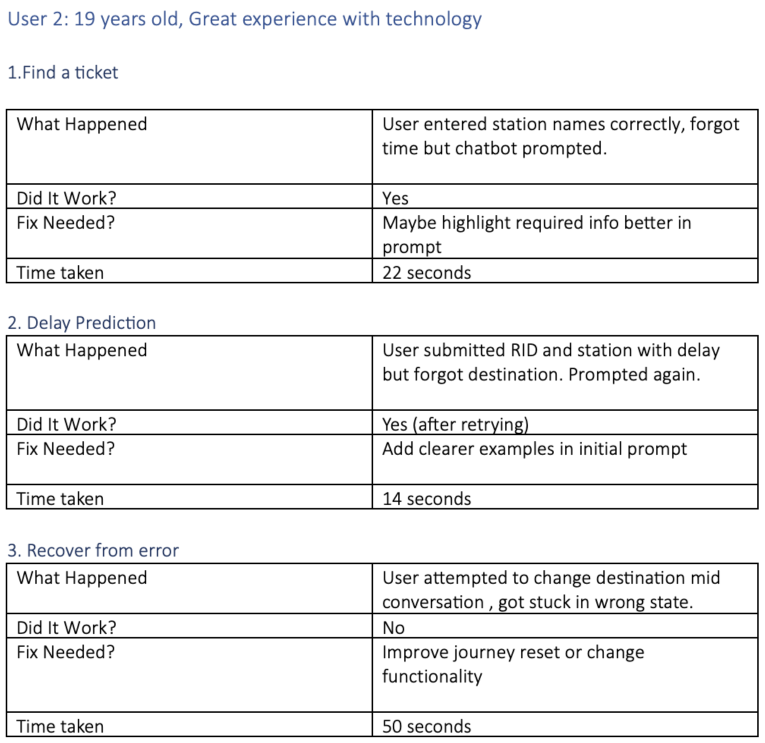
2. Delay Prediction

What Happened	User was prompted to input id station, delay, and destination.
Did It Work?	Yes
Fix Needed?	No
Time taken	12 seconds

3. Recover from error

What Happened	User tested again with trying to change time and destination
Did It Work?	No
Fix Needed?	Investigate why schedule lookup returns empty. Show suggestions or log fix.
Time taken	32 seconds

Figure 13: User 1 user testing.



User 2: 19 years old, Great experience with technology

1.Find a ticket

What Happened	User entered station names correctly, forgot time but chatbot prompted.
Did It Work?	Yes
Fix Needed?	Maybe highlight required info better in prompt
Time taken	22 seconds

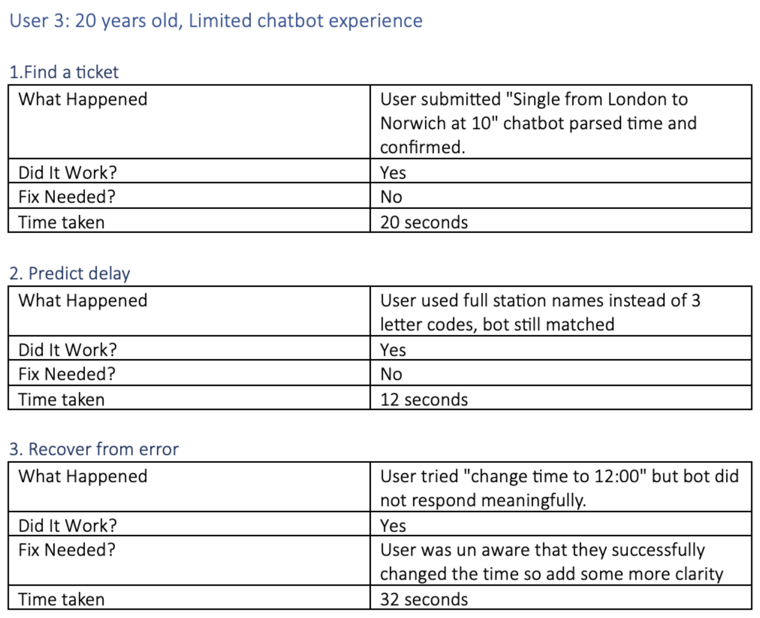
2. Delay Prediction

What Happened	User submitted RID and station with delay but forgot destination. Prompted again.
Did It Work?	Yes (after retrying)
Fix Needed?	Add clearer examples in initial prompt
Time taken	14 seconds

3. Recover from error

What Happened	User attempted to change destination mid conversation , got stuck in wrong state.
Did It Work?	No
Fix Needed?	Improve journey reset or change functionality
Time taken	50 seconds

Figure 14: User 2 user testing.



User 3: 20 years old, Limited chatbot experience

1.Find a ticket

What Happened	User submitted "Single from London to Norwich at 10" chatbot parsed time and confirmed.
Did It Work?	Yes
Fix Needed?	No
Time taken	20 seconds

2. Predict delay

What Happened	User used full station names instead of 3 letter codes, bot still matched
Did It Work?	Yes
Fix Needed?	No
Time taken	12 seconds

3. Recover from error

What Happened	User tried "change time to 12:00" but bot did not respond meaningfully.
Did It Work?	Yes
Fix Needed?	User was un aware that they successfully changed the time so add some more clarity
Time taken	32 seconds

Figure 15: User 3 user testing.

All three users tested the system without any prior experience or guidance. Their interactions were observed, and any issues or difficulties they encountered were carefully noted. Despite some users having trouble with changing journey details mid conversation, they were still able to complete the main tasks such as ticket search and delay prediction.

Based on their feedback, we made several improvements. Clearer prompts for editing journeys, a help message for guidance, and a fix for a bug in the delay prediction input handling. These changes addressed the problems users experienced and improved the overall usability of the chatbot.

7 Evaluation and Discussion

Our chatbot was tested in several ways including how well the delay prediction works, how smooth the conversation is, and how easy it is for users to get what they need. Overall, the system performed well although we also found some areas that could be improved.

7.1 Prediction Performance

7.1.1 Results

Model	**Test MAE (min)**	**Test RMSE (min)**	R^2
Linear Regression	1.92	4.29	0.006
k-Nearest Neighbours	2.01	4.44	-0.063
Random Forest	1.87	4.22	0.039
Neural Network (MLP)	1.88	4.21	0.040
With associated features added:			
Linear Regression	1.10	1.97	0.790
k-Nearest Neighbours	0.72	1.52	0.875
Random Forest	**0.34**	0.94	0.952
Neural Network (MLP)	0.46	**0.87**	**0.959**

Table 1: Comparison of delay-prediction models before and after adding "associated journey" features.

Before adding the associated journey features, all the models had a mean absolute error (MAE) of around 2 minutes and R^2 values close to zero, meaning that they weren't very good at predicting arrival times. After adding four new features average delay for the service, departure delay, and delays at the first and second stops the Random Forest model's MAE dropped to just 0.34 minutes (about 20 seconds), and its R^2 increased to 0.952. The neural network also improved significantly, with an MAE of 0.46 minutes and an R^2 of 0.959. This shows that including real time delay data makes a difference and makes more accurate predictions.

We chose the Random Forest model because its performance remained very strong overall and it achieved the lowest mean absolute error (MAE) of 0.34 minutes. While the neural network showed slightly better R^2 and RMSE values, the consistently low average error of Random Forest made it the preferred choice for accurate and reliable arrival time predictions.

7.2 System Strengths and Limitations

The chatbot could handle full journey searches and return trips, collecting details through a conversation with the user. Once it had everything it needed, it would then check the Darwin timetable to make sure there are valid journeys and then scrape ticket prices from National Rail. The results were saved to a local database, so the chatbot could refer to past queries.

However, the web scraping component isn't perfect. If National Rail updates their website the scraper might stop working. The NLP logic, while it's good for common phrases, it sometimes struggled with less usual and unclear inputs. Additionally, when live delay data wasn't available, the system had to rely on average delays, which were not always accurate or useful.

7.3 User Feedback

We ran tests with three people and asked them to complete a few tasks like finding a ticket, checking a delay, and handling an input error. Everyone managed the first two easily. One person needed a bit of help when they misspelled a station badly. Overall, users liked the flow of the conversation and found the clickable ticket links especially helpful. Based on their feedback we made some tweaks like adding better help messages and making the chatbot more forgiving when users make mistakes.

In the end, the chatbot showed it could handle real user queries reliably and make the experience feel smooth and helpful.

8 Conclusion

This project focused on building a smart chatbot to help users plan train journeys in the UK, find the cheapest tickets, and get accurate delay predictions. To do this we used a combination of useful technologies including natural language processing, rule-based logic, web scraping, and machine learning into a single, easy-to-use system.

One of the most useful features turned out to be the delay prediction. Initially, the predictions weren't very accurate but once we included live delay data from earlier stops on the route, the results improved significantly. The chatbot could then provide users not just with a delay notification but a realistic estimate of when their train would actually arrive.

The ticket search feature also worked reliably. Users could simply type in their travel plans, and the chatbot would walk them through any missing details before returning the cheapest available ticket along with a direct booking link.

There are things we'd improve if we had more time. Since scraping websites is never a perfect solution and relying on user typed input can cause confusion if the input isn't clear. In the future we would look at using official APIs, improving the dialogue engine, and expanding what the chatbot can help with.

Also by combining historical train service data with performance indicators and selecting a Random Forest regression pipeline, our chatbot was able inform passengers of accurate expected arrival times.

Overall, the chatbot achieved its goal. It combined useful AI tools into one system and showed how they can be used to solve a real problem in transport customer service.

References

[1] SNCF Voyageurs, "SNCF virtual agent," [Online]. Available: `https://www.sncf-voyageurs.com/en/contact-us/contact-us/sncf-virtual-agent/`.

[2] Alcméon, "SNCF Connect Streamlines Customer Service with 35% Hybrid Messaging," [Online]. Available: `https://www.alcmeon.ai/customer-stories/multichannel-integration-synergy-between-bots-ai-and-advisors-sncf-connect-at-the-forefront-o`

[3] National Rail Enquiries, "Travel Alerts and Notifications," [Online]. Available: `https://www.nationalrail.co.uk/travel-information/alerts-and-notifications/`.

[4] Deutsche Bahn AG, "Artificial Intelligence at DB," [Online]. Available: `https://www.deutschebahn.com/en/artificial_intelligence-6935068`.

[5] Rasa, "Introduction to Rasa Open Source & Rasa Pro," [Online]. Available: `https://rasa.com/docs/rasa/`.