Comparative Evaluation and Simulated Implementation of NLU Frameworks for Chatbots

Name: Edupulapati Sai Praneeth

Registration no: 23BCE8762

College: VIT- AP University

Date: October 27, 2023

Objective: To compare and evaluate four major Natural Language Understanding (NLU) frameworks — Google Dialogflow (ES), Rasa NLU (Open Source), Microsoft LUIS, and OpenAI GPT (via API) — by conceptually building and simulating a simple chatbot use case across all platforms. The comparison focuses on Customization, Ease of Use, and Deployment aspects. A simulated Python command-line chatbot demonstrates how NLU results from these platforms might be integrated into application logic.

1. Introduction

Natural Language Understanding (NLU) is a critical component of modern chatbots, enabling them to interpret user input, identify intents, and extract relevant information (entities). Several powerful frameworks exist, each with its strengths and weaknesses. This document evaluates Google Dialogflow, Rasa NLU, Microsoft LUIS, and OpenAI GPT by considering a common chatbot use case: booking an appointment.

We will:

1. Define the simple chatbot use case.
2. Provide a comparative analysis of the frameworks based on key criteria.
3. Present a Python script simulating a command-line chatbot that integrates the *conceptual* NLU outputs from these frameworks to illustrate their usage in a basic dialogue flow.

2. Use Case Definition: Simple Haircut Appointment Booker

- **Goal:** Allow users to book a haircut appointment via a command-line interface.
- **Key Intents:**
  - greet: User initiates conversation (e.g., "Hi", "Hello").
  - book_appointment: User expresses the desire to book (e.g., "I need an appointment").
  - provide_details: User provides necessary information (e.g., "Haircut tomorrow at 2 pm"). *This might be combined with book_appointment depending on the framework/design.*
  - goodbye: User ends the conversation (e.g., "Bye", "Thanks").
  - affirm: User confirms (e.g., "yes").
- **Key Entities:**
  - service_type: The type of service requested (e.g., "haircut", "coloring", "wash"). [Custom Entity]
  - date: The desired date (e.g., "tomorrow", "next Tuesday", "May 5th"). [System/Prebuilt or Custom]
  - time: The desired time (e.g., "3 pm", "around noon", "morning"). [System/Prebuilt or Custom]

3. NLU Framework Overviews (Brief)

- **Google Dialogflow (ES):** A cloud-based platform primarily configured via a web GUI. Known for its ease of use, strong system entity support, and integration with Google Cloud services. Fulfillment (backend logic) is often handled via webhooks (e.g., Cloud Functions).
- **Rasa NLU (Open Source):** An open-source framework configured via YAML/Markdown files. Offers deep customization of the NLU pipeline components. Can be deployed on-premises or on any cloud. Requires more setup and understanding of its components.
- **Microsoft LUIS:** A cloud-based service within Azure AI. Similar GUI-driven approach to Dialogflow ES, with strong prebuilt entity support and integration with the Microsoft Azure ecosystem (especially Azure Bot Service).
- **OpenAI GPT (API):** Leverages large language models (LLMs) via API calls. NLU is achieved through carefully crafted prompts (few-shot learning) or by fine-tuning a base model. Offers high flexibility in understanding nuanced language but requires prompt engineering skills and managing API costs/reliability for structured output.

4. Comparative Evaluation

Feature	Google Dialogflow (ES)	Rasa NLU (Rasa Open Source)	Microsoft LUIS	OpenAI GPT (API-based NLU)
Customization	**Moderate-High:** GUI-based intent/entity config. Good system entities. Custom entities (List, Regex, Composite). Fulfillment via webhooks for custom logic. Less control over underlying ML models than Rasa.	**Very High:** File-based config (YAML/MD). Full pipeline control (tokenizers, featurizers, classifiers). Custom entity extractors. Regex, lookups, synonyms. Open source allows core modification. Custom Actions (Python) for backend logic.	**Moderate-High:** GUI config similar to Dialogflow. Strong prebuilt domains/entities. Custom entities (Simple, List, Regex, ML, Composite). Pattern features. Integrates with Azure Bot Framework for logic.	**High (via Prompting/Fine-tuning):** NLU via prompts or fine-tuning. No rigid intent/entity structure required upfront. Extreme flexibility in interpreting nuances. Output format controlled by prompt/fine-tuning. Can generate complex responses directly. Requires prompt engineering or data preparation for fine-tuning
Ease of Use	**Very High:** Intuitive web GUI. Quick to start. Prebuilt agents. Easy annotation. Integrated testing console. Point-and-click integrations (basic).	**Moderate:** Requires local setup/Docker. Steeper learning curve (YAML, pipeline concepts). CLI-driven initially (Rasa X provides UI). More complex debugging/testing setup.	**High:** Intuitive web GUI in Azure portal. Good tutorials. Similar feel to Dialogflow ES. Integrated testing. Active learning suggestions.	**High (for simple cases) / Moderate (for reliability):** Quickest to get *some* result via API call with a good prompt. No platform setup. Requires significant prompt iteration or fine-tuning for *consistent*, *reliable*, structured NLU output. Debugging involves prompt tweaking. Cost per API call.
Deployment	**Easy (Cloud):** Managed GCP service. Scalable. Integrations (Web, Slack, Assistant). Client SDKs (Python, Node, etc.).	**Flexible (Requires Ops):** Self-hosted (on-prem, any cloud). Deploy via Docker/Kubernetes (Helm charts available). Requires infrastructure management. Rasa Enterprise offers more tools. REST API. Python SDK for Actions.	**Easy (Cloud):** Managed Azure service. Scalable. Tight Azure Bot Service integration. Container export possible (specific plans). Client SDKs (.NET, Python, Node, etc.). REST API.	**Easy (API):** Cloud-only API endpoint (OpenAI/Azure). Highly scalable. Integrate via HTTPS calls from any backend. No NLU infrastructure to manage. Official SDKs (Python, Node).

5. Simulated Chatbot Implementation

The following Python script implements a simple command-line appointment booking chatbot.

Crucially, this script *simulates* the NLU process. It does not make live API calls to Dialogflow, Rasa, LUIS, or OpenAI due to the complexities of managing multiple authentications, potential costs, and required service deployments (especially for Rasa) within a single script.

Instead, the get_..._nlu functions use basic keyword matching to return a Python dictionary structured *similarly* to what each real NLU service *might* return for the given input. The main chatbot logic then uses the simulated Dialogflow NLU output to drive the conversation flow, demonstrating how intents and entities guide the interaction.

Code:

```
import json
import datetime
import random # For simple varied responses


# ------------------------------------
# NLU Simulation Functions
# (These functions mimic the *structure* of expected responses, not real NLU)
# ------------------------------------


def get_dialogflow_nlu(text_input, needed_entities=None):
    """Simulates Dialogflow ES NLU."""
    print("... (Simulating Dialogflow NLU call)")
    # Basic intent detection based on keywords
    intent_name = "unknown"
    confidence = 0.5
    params = {}

    text_lower = text_input.lower()

    if any(w in text_lower for w in ["hi", "hello", "hey"]):
        intent_name = "greet"
        confidence = 0.95
    elif any(w in text_lower for w in ["bye", "goodbye", "see ya"]):
        intent_name = "goodbye"
        confidence = 0.95
    elif any(w in text_lower for w in ["book", "appointment", "schedule"]):
        intent_name = "book_appointment"
        confidence = 0.85
    elif any(w in text_lower for w in ["haircut", "coloring", "wash", "tomorrow", "pm", "am",
"monday", "tuesday", "yes"]): # Added 'yes'
         # Assume providing details or confirming if keywords present
         intent_name = "provide_details" if not text_lower == 'yes' else 'affirm'
         confidence = 0.80 if intent_name != 'affirm' else 0.90

    # Basic entity extraction simulation
    if "haircut" in text_lower or "cut" in text_lower:
        params["service_type"] = "haircut"
    elif "coloring" in text_lower:
        params["service_type"] = "coloring"
    elif "wash" in text_lower or "shampoo" in text_lower:
         params["service_type"] = "wash"

    if "tomorrow" in text_lower:
         # Crude simulation - real system handles dates robustly
         tomorrow_date = (datetime.date.today() + datetime.timedelta(days=1)).isoformat()
         params["date"] = f"{tomorrow_date}T00:00:00+00:00" # Example format
    # Add more crude date/time parsing if needed for simulation

    if "pm" in text_lower or "afternoon" in text_lower:
        params["time"] = "T14:00:00+00:00" # Simulate 2 PM
    elif "am" in text_lower or "morning" in text_lower:
        params["time"] = "T10:00:00+00:00" # Simulate 10 AM

    # Simulate prompting if needed (simplified)
    fulfillment_text = "Okay."
    all_required_present = True
    if needed_entities:
        missing = []
        # Check against params FOUND in *this* turn
        if "service_type" not in params and "service_type" in needed_entities:
            missing.append("service type")
            all_required_present = False
        if "date" not in params and "date" in needed_entities:
             missing.append("date")
             all_required_present = False
        if "time" not in params and "time" in needed_entities:
             missing.append("time")
             all_required_present = False

        # Refine fulfillment text based on what's missing GLOBALLY
        global conversation_context # Need global context to know what's truly missing
        truly_missing = []
        if not conversation_context["service_type"] and 'service_type' not in params:
truly_missing.append("service type")
        if not conversation_context["date"] and 'date' not in params:
truly_missing.append("date")
        if not conversation_context["time"] and 'time' not in params:
truly_missing.append("time")


        if truly_missing:
            fulfillment_text = f"Okay, what {' and '.join(truly_missing)} would you like for the
appointment?"
        elif not all_required_present: # If something needed wasn't provided this turn, but
context might have it
             fulfillment_text = "Can you please provide the remaining details?"



    return {
      "queryResult": {
        "queryText": text_input,
        "parameters": params,
        "allRequiredParamsPresent": all_required_present, # Indicates if *this query* filled all
requirements
        "fulfillmentText": fulfillment_text if intent_name not in ["greet", "goodbye", "affirm"]
else random.choice(["Hi there!", "Hello!"]),
        "intent": {"displayName": intent_name},
        "intentDetectionConfidence": confidence,
        "languageCode": "en"
      }
    }


def get_rasa_nlu(text_input):
    """Simulates Rasa NLU."""
    print("... (Simulating Rasa NLU call)")
    # Very basic simulation - real Rasa is much more sophisticated
    intent_name = "unknown"
    confidence = 0.5
    entities = []
    text_lower = text_input.lower()

    # Crude intent/entity mapping for simulation
    if any(w in text_lower for w in ["hi", "hello", "hey"]):
        intent_name = "greet"
        confidence = 0.95
    elif any(w in text_lower for w in ["bye", "goodbye", "see ya"]):
        intent_name = "goodbye"
        confidence = 0.95
    elif any(w in text_lower for w in ["book", "appointment", "schedule"]):
        intent_name = "book_appointment"
        confidence = 0.85
    elif text_lower == 'yes':
        intent_name = 'affirm'
        confidence = 0.90
    elif any(w in text_lower for w in ["haircut", "coloring", "wash", "tomorrow", "pm", "am",
"monday", "tuesday"]):
        intent_name = "provide_details" # Assume providing details if keywords present
        confidence = 0.80



    if "haircut" in text_lower: entities.append({"entity": "service_type", "value": "haircut"})
    if "coloring" in text_lower: entities.append({"entity": "service_type", "value": "coloring"})
    if "wash" in text_lower: entities.append({"entity": "service_type", "value": "wash"})
    if "tomorrow" in text_lower: entities.append({"entity": "date", "value": "tomorrow"}) #
Raw value
    if "pm" in text_lower: entities.append({"entity": "time", "value": "pm"}) # Raw value
    if "am" in text_lower: entities.append({"entity": "time", "value": "am"}) # Raw value

    return {
        "text": text_input,
        "intent": {"name": intent_name, "confidence": confidence},
        "entities": entities
    }

def get_luis_nlu(text_input):
    """Simulates LUIS NLU."""
    print("... (Simulating LUIS NLU call)")
    # Very basic simulation
    top_intent = "None"
    score = 0.5
    entities = {}
    text_lower = text_input.lower()

    # Crude intent/entity mapping for simulation
    if any(w in text_lower for w in ["hi", "hello", "hey"]):
        top_intent = "Greet"
        score = 0.95
    elif any(w in text_lower for w in ["bye", "goodbye", "see ya"]):
        top_intent = "Goodbye"
        score = 0.95
    elif text_lower == 'yes':
        top_intent = 'Confirm' # Example name for affirmation
        score = 0.90
    elif any(w in text_lower for w in ["book", "appointment", "schedule"]):
        top_intent = "BookAppointment"
        score = 0.85
    elif any(w in text_lower for w in ["haircut", "coloring", "wash", "tomorrow", "pm", "am",
"monday", "tuesday"]):
         top_intent = "ProvideDetails" # Assume providing details if keywords present
         score = 0.80

    service_type = None
    if "haircut" in text_lower: service_type = "Haircut"
    if "coloring" in text_lower: service_type = "Coloring"
    if "wash" in text_lower: service_type = "Wash"
    if service_type:
        entities["ServiceType"] = [{"normalizedValue": service_type}] # LUIS structure

    datetime_val = None
    if "tomorrow" in text_lower and "pm" in text_lower:
        # Very crude simulation of datetimeV2 resolution
        tomorrow_date = (datetime.date.today() + datetime.timedelta(days=1)).isoformat()
        datetime_val = f"{tomorrow_date}T14:00:00" # Example 2 PM tomorrow
    elif "tomorrow" in text_lower:
         tomorrow_date = (datetime.date.today() + datetime.timedelta(days=1)).isoformat()
         datetime_val = f"{tomorrow_date}" # Example date only

    if datetime_val:
        dt_type = "datetime" if "T" in datetime_val else "date"
        entities["datetimeV2"] = [{"type": dt_type, "values": [{"timex": datetime_val,
"resolution": [{"value": datetime_val}]}]}]

    return {
        "query": text_input,
        "prediction": {
            "topIntent": top_intent,
            "intents": {top_intent: {"score": score}},
            "entities": entities
        }
    }

def get_openai_nlu(text_input):
    """Simulates OpenAI Prompt-based NLU."""
    print("... (Simulating OpenAI NLU call)")
    # Very basic simulation based on keywords
    intent = "unknown"
    entities = {}
    text_lower = text_input.lower()

    if any(w in text_lower for w in ["hi", "hello", "hey"]): intent = "greet"
    elif any(w in text_lower for w in ["bye", "goodbye", "see ya"]): intent = "goodbye"
    elif text_lower == 'yes': intent = 'affirm'
    elif any(w in text_lower for w in ["book", "appointment", "schedule"]): intent =
"book_appointment"
    elif any(w in text_lower for w in ["haircut", "coloring", "wash", "tomorrow", "pm", "am",
"monday", "tuesday"]): intent = "provide_details"



    if "haircut" in text_lower: entities["service_type"] = "haircut"
    if "coloring" in text_lower: entities["service_type"] = "coloring"
    if "wash" in text_lower: entities["service_type"] = "wash"
    if "tomorrow" in text_lower: entities["date"] = "tomorrow" # Raw
    if "pm" in text_lower: entities["time"] = "pm" # Raw
    if "am" in text_lower: entities["time"] = "am" # Raw

    # Simulate the JSON structure we would *expect* back from the LLM
    return {
        "intent": intent,
        "entities": entities
    }

# -----------------------------------
# Basic Chatbot Logic
# -----------------------------------

# Store collected information (simple state) - GLOBAL
conversation_context = {
    "service_type": None,
    "date": None,
    "time": None,
    "booking_pending": False # Flag to indicate we are in the booking process
}

def extract_entities_from_dialogflow(df_result):
    """Helper to get entities from the simulated Dialogflow result."""
    params = df_result.get("queryResult", {}).get("parameters", {})
    return {
        # Only return non-None values found in this turn
        key: value for key, value in {
            "service_type": params.get("service_type"),
            "date": params.get("date"),
            "time": params.get("time")
        }.items() if value is not None
    }

def run_chatbot():
    """Main chatbot loop."""
    global conversation_context # Ensure we can modify the global context

    print("Chatbot: Hello! How can I help you book an appointment today?")
    print("Chatbot: (We offer haircut, coloring, or wash services)")
    print("Chatbot: (You can type 'quit' to exit)")
```

```
last_bot_response = "" # Keep track of the last response for 'yes' check

while True:
    user_message = input("You: ")
    if user_message.lower() == 'quit':
        print("Chatbot: Okay, goodbye!")
        break

    # --- NLU Phase (Simulated) ---
    print("\n--- NLU Analysis (Simulated) ---")
    # Decide which entities are currently needed if we are booking
    needed = []
    if conversation_context["booking_pending"]:
        if not conversation_context["service_type"]: needed.append("service_type")
        if not conversation_context["date"]: needed.append("date")
        if not conversation_context["time"]: needed.append("time")

    # Get simulated NLU results (passing needed entities for Dialogflow sim)
    df_nlu = get_dialogflow_nlu(user_message, needed if needed else None)
    rasa_nlu = get_rasa_nlu(user_message)
    luis_nlu = get_luis_nlu(user_message)
    openai_nlu = get_openai_nlu(user_message)

    # Display primary NLU result driving the logic
    print("\n--- [Dialogflow NLU Result (Simulated)] ---")
    print(json.dumps(df_nlu, indent=2))
    # --- You can uncomment these to see the other simulated outputs ---
    # print("\n--- [Rasa NLU Result (Simulated)] ---")
    # print(json.dumps(rasa_nlu, indent=2))
    # print("\n--- [LUIS NLU Result (Simulated)] ---")
    # print(json.dumps(luis_nlu, indent=2))
    # print("\n--- [OpenAI NLU Result (Simulated)] ---")
    # print(json.dumps(openai_nlu, indent=2))
    print("\n-----------------------------------\n")



    # --- Dialogue Management & Response Generation ---
    # Using Dialogflow's simulated result to drive the logic primarily

    intent = df_nlu["queryResult"]["intent"]["displayName"]
    entities_found_this_turn = extract_entities_from_dialogflow(df_nlu)

    bot_response = "Chatbot: Sorry, I didn't quite understand that. Can you rephrase?" #
Default fallback

    # Handle 'affirm' intent, specifically after confirmation question
    if intent == 'affirm' and "Is that correct?" in last_bot_response:
        bot_response = "Chatbot: Great! Your appointment is confirmed. See you then!"
        # Reset context fully
        conversation_context = {"service_type": None, "date": None, "time": None,
"booking_pending": False}

    elif intent == "greet":
        bot_response = "Chatbot: " + random.choice(["Hello!", "Hi there!", "Good day! How
can I help with booking?"])
    elif intent == "goodbye":
        bot_response = "Chatbot: " + random.choice(["Goodbye!", "See you later!", "Have a
great day!"])
        print(bot_response)
        break # End conversation

    elif intent == "book_appointment" or intent == "provide_details" or
conversation_context["booking_pending"]:
        conversation_context["booking_pending"] = True # Enter/stay in booking mode

        # Update context with newly found entities if not already set
        for key, value in entities_found_this_turn.items():
            if not conversation_context.get(key):
                conversation_context[key] = value
                print(f"... Context updated: {key} = {value}") # Debug message

        # Check if all required details are collected in the global context
        if conversation_context["service_type"] and conversation_context["date"] and
conversation_context["time"]:
            # All details present - Ask for confirmation
            svc = conversation_context['service_type']
            # Crude extraction - better parsing needed in real bot
            d = conversation_context['date'].split('T')[0] if conversation_context['date'] else
'unknown date'
            t_raw = conversation_context['time']
            t = t_raw.split('T')[1].split('+')[0][:-3] if t_raw and 'T' in t_raw else 'unknown time'

            bot_response = f"Chatbot: Okay, I have the following details: Service='{svc}',
Date='{d}', Time='{t}'. Is that correct? (Type 'yes' to confirm)"
            # Don't reset context here yet, wait for confirmation
        else:
            # Ask for missing information (using Dialogflow's simulated fulfillment text)
            # Ensure the fulfillment text is relevant by recalculating needed entities
            still_needed = []
            if not conversation_context["service_type"]: still_needed.append("service type")
            if not conversation_context["date"]: still_needed.append("date")
            if not conversation_context["time"]: still_needed.append("time")

            if still_needed:
                bot_response = f"Chatbot: Okay, what {' and '.join(still_needed)} would you like
for the appointment?"
            else: # Should ideally not happen if logic is correct, but fallback
                bot_response = "Chatbot: Can you please provide the remaining details?"
    # (Keep the default fallback response if no other condition matched)

    print(bot_response)
    last_bot_response = bot_response # Store the response for the next loop iteration
```

```
# --- Main Execution ---
if __name__ == "__main__":
   run_chatbot()
```

6. Code Explanation

- **NLU Simulation Functions (get_..._nlu):** These functions take user text and return a dictionary structured like the respective NLU service's output. They use simple string matching for intents and entities – **this is highly simplified and not representative of the actual NLU capabilities.**
- **Global Context (conversation_context):** A dictionary storing the state of the conversation, specifically the service_type, date, and time collected so far, and whether a booking is pending. The global keyword is used in run_chatbot to allow modification of this variable.
- **Main Loop (run_chatbot):**
    - Greets the user and enters a loop taking input.
    - Calls all simulated NLU functions for the input.
    - Prints the (simulated) Dialogflow NLU result for inspection.
    - Uses the **intent** and **entities** from the simulated Dialogflow result to drive the logic.
    - Updates the conversation_context with any newly found entities.
    - Checks if all required entities are in the conversation_context.
    - If complete, asks for confirmation.
    - If incomplete, uses the simulated fulfillmentText (which asks for missing info) as the response.
    - Includes basic handling for greet, goodbye, and affirm (confirmation) intents.
    - Stores the last_bot_response to correctly handle the 'yes' confirmation.

- **Entity Extraction (extract_entities_from_dialogflow):** A helper function to pull parameters from the simulated Dialogflow result structure.

7. Observations and Summary

- **Dialogflow & LUIS:** Offer the fastest path to a working prototype for common use cases due to their intuitive GUIs, managed cloud infrastructure, and strong handling of prebuilt entities (like date/time). Customization is good but operates within the platform's defined structures. Deployment is straightforward within their respective cloud ecosystems (GCP/Azure).
- **Rasa NLU:** Provides maximum control and transparency. Ideal when deep customization of the NLU pipeline is needed, specific algorithms are required, or when open-source and flexible deployment (on-prem/any cloud) are priorities. It has the steepest learning curve and requires managing the deployment infrastructure.
- **OpenAI GPT:** Offers unparalleled flexibility in understanding natural language nuances without strict intent/entity definitions. Getting started is quick via API. However, achieving *reliable* structured NLU output consistently requires significant prompt engineering effort or fine-tuning. Deployment is simple (API integration), but costs and potential latency/unpredictability need consideration for structured tasks.

8. Conclusion

The "best" NLU framework depends heavily on project requirements:

- For **ease of use, rapid development, and seamless cloud integration (GCP/Azure)**, **Dialogflow** and **LUIS** are strong contenders.
- For **maximum control, customization, open-source benefits, and deployment flexibility**, **Rasa NLU** is the preferred choice, accepting a higher initial complexity.
- For **leveraging state-of-the-art LLMs, handling highly conversational input, or when traditional NLU structures are**

insufficient, **OpenAI GPT** provides powerful capabilities, requiring careful prompt design or fine-tuning for reliable NLU extraction.

The provided Python code serves as a structural example of how NLU outputs (regardless of the source) are used within a simple chatbot's logic to manage dialogue state and generate appropriate responses. It highlights the core loop of receiving input, understanding it (NLU), deciding what to do (dialogue management), and responding.