

UNIVERSITÀ POLITECNICA DELLE MARCHE
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

**MeteoAssistantBot: Chatbot per Previsioni
Meteo con Rasa**



Corso di
DATA SCIENCE

Anno accademico 2024-2025

Studenti:

Barbarella Marco
Faccenda Andrea
Pepe Luigi

Professori:

Ursino Domenico
Buratti Christopher



Dipartimento di Ingegneria dell'Informazione

Indice

1	Introduzione	2
1.1	Definizione e problematiche delle previsioni meteo automatiche	2
1.2	Obiettivo del progetto	2
1.3	Panoramica della soluzione proposta (Rasa)	3
2	Dataset	4
2.1	Descrizione del dataset	4
2.2	Struttura della risposta OpenWeather Current Weather	4
2.3	Previsioni a 5 giorni (3-hour step)	5
2.4	Dataset: European Tour Destinations	6
3	Metodologia	8
3.1	Introduzione a Rasa	8
3.2	Architettura del bot	9
4	Conessione a Telegram & Test	18
4.1	Telegram	18
4.2	Test - MeteoAssistantBot	20
5	Conclusioni e sviluppi futuri	25

1 Introduzione

1.1 Definizione e problematiche delle previsioni meteo automatiche

Negli ultimi anni, l'accesso a informazioni meteorologiche precise e tempestive è diventato un requisito essenziale in numerosi contesti, sia personali che professionali. Dalla pianificazione di viaggi e attività all'aperto, alla gestione di eventi, fino ad applicazioni in ambito agricolo e logistico, la disponibilità di previsioni meteo accurate consente di prendere decisioni informate e ridurre rischi e imprevisti.

Tradizionalmente, l'utente deve cercare manualmente queste informazioni su siti web o applicazioni dedicate. Tuttavia, questo approccio presenta alcune problematiche:

- **Accessibilità:** non tutti gli utenti dispongono della stessa familiarità con strumenti tecnici o siti specializzati.
- **Personalizzazione limitata:** molte piattaforme offrono previsioni generiche, senza adattarle al contesto o alle richieste specifiche dell'utente.
- **Interazione poco naturale:** i servizi tradizionali raramente permettono di interrogare il sistema con linguaggio naturale e ottenere risposte conversazionali.

In questo scenario, l'adozione di chatbot basati su NLP (*Natural Language Processing*) può fornire un'interfaccia intuitiva, permettendo agli utenti di ottenere previsioni meteorologiche semplicemente dialogando in linguaggio naturale.

1.2 Obiettivo del progetto

L'obiettivo principale del *MeteoAssistantBot* è realizzare un sistema di assistenza virtuale capace di fornire informazioni meteorologiche affidabili tramite interazione conversazionale. Il bot è progettato per:

- Comprendere richieste espresse in linguaggio naturale, anche in forma colloquiale.
- Recuperare dati meteo aggiornati da fonti esterne (API dedicate).
- Presentare le informazioni in modo chiaro e personalizzato, adattandosi alle esigenze dell'utente.
- Essere facilmente integrabile in piattaforme di messaggistica come Telegram, Slack o web chat.

Questa soluzione si inserisce nell'ambito della Data Science applicata al linguaggio naturale, unendo competenze di *Natural Language Understanding* (NLU), *dialog management* e integrazione con servizi esterni.

1.3 Panoramica della soluzione proposta (Rasa)

La soluzione implementata si basa su **Rasa**, un framework open source per lo sviluppo di assistenti virtuali intelligenti.

La scelta di Rasa è motivata dalla sua architettura modulare, che combina:

- **Rasa NLU**: per il riconoscimento degli intenti e l'estrazione di entità dal testo inserito dall'utente.
- **Rasa Core**: per la gestione del flusso conversazionale e la selezione della risposta più appropriata.
- **Action Server**: per l'esecuzione di azioni personalizzate, come il recupero di dati da un'API meteo.

Il *MeteoAssistantBot* utilizza un modello NLU addestrato su un set di frasi di esempio che coprono vari scenari (richiesta di previsioni, domande sul meteo attuale, localizzazione geografica, ecc.). Le azioni personalizzate interrogano un servizio di previsioni meteorologiche esterno e restituiscono i dati elaborati in linguaggio naturale.

Questa architettura consente di:

- Separare in maniera chiara la comprensione linguistica dalla logica di business.
- Aggiornare facilmente il modello NLU con nuovi esempi per migliorarne l'accuratezza.
- Integrare il bot con diversi canali di comunicazione.

Nei capitoli successivi verranno descritti in dettaglio le fonti di dati, la metodologia di sviluppo, i risultati sperimentali e le possibilità di implementazione pratica.

2 Dataset

2.1 Descrizione del dataset

A differenza di progetti di classificazione basati su un dataset statico, il *MeteoAssistantBot* combina dati dinamici da un servizio esterno (API meteo) con un dataset statico dedicato alla conoscenza delle destinazioni. In pratica, le risposte sul meteo provengono dall'API OpenWeather, invocata tramite *custom actions* di Rasa, mentre le informazioni generali sulle città sono tratte dal file *European Tour Destinations*.

Il dataset meteo è costituito da risposte JSON in tempo reale ottenute dagli endpoint *Current Weather*, *Forecast 5 giorni* e, ove necessario, *One Call*.

2.2 Struttura della risposta OpenWeather Current Weather

Di seguito si sintetizzano i principali campi del JSON restituito da Current Weather, organizzati per sezione funzionale.

Campo	Descrizione
Metadati e localizzazione	
coord.lat, coord.lon	Coordinate in gradi decimali della località.
timezone, dt	Scostamento da UTC (secondi) e timestamp della misura (Unix, UTC).
id, name, sys.country	Identificativo città, nome città e codice paese (ISO).
Condizioni meteo	
weather.id/main/description	Condizione meteo (id), gruppo (es. Rain, Snow), descrizione.
clouds.all	Nuvolosità, in percentuale.
visibility	Visibilità media in metri (max 10 km).
Parametri termodinamici	
main.temp, main.feels_like	Temperatura e percepita. Unità: <i>standard</i> =K (default), <i>metric</i> =°C, <i>imperial</i> =°F.
main.temp_min, main.temp_max	Min/max osservati nel momento corrente (rilevanti in grandi aree urbane).
main.pressure, main.humidity	Pressione al livello del mare (hPa) e umidità relativa (%).

main.sea_level,	Pressione a livello del mare / suolo in hPa.
main.grnd_level	
Vento e precipitazioni	
wind.speed, wind.deg,	Velocità (m/s o mph), direzione (gradi meteo), raffiche
wind.gust	(se disponibili).
rain.1h, snow.1h	Precipitazione nell'ultima ora (mm/h).
Sole e altri campi	
sys.sunrise, sys.sunset	Orari di alba e tramonto (Unix, UTC).
base, cod	Parametri interni di servizio.

2.3 Previsioni a 5 giorni (3-hour step)

L'endpoint *5 day / 3 hour forecast* restituisce fino a 5 giorni di previsioni con passo tri-orario (tipicamente 40 timestamp). L'output è disponibile in JSON (default) e XML, con localizzazione tramite `lang` e controllo delle unità via `units`.

Campo	Descrizione
Metadati	
cod, message, cnt	Codice interno, messaggio, numero di passi tri-orari restituiti.
Lista previsioni (list[], passo 3h)	
list.dt, list.dt_txt	Timestamp (Unix, UTC) e data/ora testuale.
list.main.temp,	Temperatura e percepita (unità secondo <code>units</code>).
list.main.feels_like	
list.main.temp_min,	Min/Max del passo (in contesti urbani ampi).
list.main.temp_max	
list.main.pressure, sea_level, grnd_level, humidity	Pressione (hPa) e umidità (%).
list.weather.id/main/desc	Condizione meteo (id/gruppo/descrizione).
list.clouds.all, list.visibility	Nuvolosità (%) e visibilità (metri).
list.wind.speed/deg/gust	Vento: velocità, direzione, raffiche.
list.pop, list.rain.3h, list.snow.3h	Probabilità di precipitazione [0, 1] e volumi su 3h (mm).
list.sys.pod	Indicatore giorno/notte del passo (d/n).
Informazioni città	
city.id, city.name, city.country	Identificativo, nome e paese.

city.coord.lat/lon,	Coordinate e offset fuso orario (secondi).
city.timezone	
city.population,	Popolazione; alba e tramonto (Unix, UTC).
city.sunrise/sunset	

2.4 Dataset: European Tour Destinations

Accanto ai dati dinamici, il sistema utilizza un dataset statico di destinazioni turistiche europee¹ per arricchire il contesto geografico, migliorare l'estrazione dell'entità `location` e fornire cenni generali utili a contestualizzare il meteo (cosa vedere, periodo consigliato, cucina tipica).

Struttura del CSV

Campo	Descrizione
<i>Nota:</i> la struttura effettiva può variare a seconda della versione del dataset scaricata.	
Destination	Nome della città o destinazione turistica.
Country	Paese di appartenenza della destinazione.
Region	Macro-area europea (ad es. Western Europe, Southern Europe).
Latitude	Latitudine decimale della destinazione.
Longitude	Longitudine decimale della destinazione.
Category	Tipologia prevalente (ad es. <i>Cultural, Beach, Nature</i>).
Best Time to Visit	Periodo o mesi consigliati per la visita.
Famous Foods	Piatti tipici locali e specialità culinarie.
Language	Lingua o lingue principali.
Currency	Valuta locale.
Approximate Annual Tourists	Stima del flusso turistico annuale (valore approssimato).
Cost of Living	Indicatore del costo della vita (valore numerico o categoria).
Majority Religion	Religione prevalente (indicatore culturale).

¹Kaggle — European Tour Destinations Dataset: <https://www.kaggle.com/datasets/faizadani/european-tour-destinations-dataset>

Ruolo nel progetto

Il CSV non sostituisce i dati meteo, ma li integra in due direzioni: come fonte di coordinate per chiamare OpenWeather direttamente su latitudine e longitudine, evitando passaggi di geocoding; come base per arricchire la risposta con brevi note descrittive sul periodo migliore, sui punti d'interesse, attrazioni da visitare o sulla gastronomia locale.

Complessivamente, la combinazione di dati meteorologici in tempo reale e informazioni statiche sulle destinazioni turistiche consente al *MeteoAssistantBot* di fornire risposte non solo accurate dal punto di vista meteo, ma anche arricchite da un contesto culturale e pratico. Le previsioni di OpenWeather, integrate tramite *custom actions* di Rasa, garantiscono aggiornamenti affidabili e puntuali, mentre il dataset *European Tour Destinations* funge da base di conoscenza stabile per descrivere la località, suggerire periodi ottimali di visita e offrire indicazioni sulle attrazioni e tradizioni locali. Questo approccio ibrido, basato sulla fusione di dati dinamici e statici, ottimizza la user experience e valorizza l'interazione conversazionale, rendendo il sistema più utile, coinvolgente e informativo.

3 Metodologia

3.1 Introduzione a Rasa

Rasa è un framework *open source* progettato per la creazione di assistenti virtuali intelligenti e chatbot conversazionali. Il sistema è composto da due moduli principali, **Rasa NLU** e **Rasa Core**, che lavorano in stretta integrazione: il primo si occupa di comprendere il linguaggio naturale, mentre il secondo gestisce il flusso della conversazione e determina le risposte più appropriate.

Rasa NLU

Il modulo **Rasa NLU** (*Natural Language Understanding*) è responsabile della comprensione del messaggio inviato dall'utente. Le sue due funzioni principali sono:

- **Classificazione degli intenti** (*Intent Classification*): identificare l'obiettivo o l'azione che l'utente vuole compiere, ad esempio chiedere il meteo, richiedere un'informazione o prenotare un servizio.
- **Estrazione delle entità** (*Entity Extraction*): individuare informazioni specifiche nel testo, come nomi di città, date o tipologie di attività.

Per svolgere queste attività, Rasa NLU utilizza modelli di *machine learning* addestrati su un insieme di frasi di esempio fornite dallo sviluppatore. Il processo di comprensione segue una **pipeline di elaborazione** definita dal progettista, che stabilisce la sequenza di componenti e trasformazioni applicate al testo. Tra questi componenti vi sono, ad esempio, il modulo *Intent Classifier* e il modulo *Entity Extraction*, che operano in combinazione per produrre un'interpretazione strutturata del messaggio.

Rasa Core

Il modulo **Rasa Core** gestisce l'evoluzione del dialogo con l'utente. Partendo dalle informazioni prodotte da Rasa NLU (intenti ed entità), decide quale sia la prossima azione da eseguire. Questa scelta avviene sulla base di **Policy** definite nel sistema, che utilizzano modelli probabilistici — ad esempio reti neurali LSTM — per predire la risposta più adatta, tenendo conto dello stato attuale della conversazione e della sua cronologia.

Le azioni che Rasa Core può eseguire includono:

- **Risposte predefinite** basate su template di testo;
- **Azioni personalizzate** (*custom actions*) che permettono di integrare API esterne, interrogare basi di dati o eseguire elaborazioni complesse per restituire all'utente informazioni mirate.

Questa architettura rende Rasa altamente flessibile, consentendo di adattare sia la fase di comprensione linguistica sia la gestione della conversazione alle esigenze specifiche del dominio applicativo.

3.2 Architettura del bot

Il *MeteoAssistantBot* è organizzato secondo una tipica architettura Rasa, con i seguenti componenti principali:

- `domain.yml` — definisce il dominio del chatbot, includendo intenti, entità, slot, azioni e risposte. Rappresenta il “cuore” logico del bot, specificando cosa può comprendere e come può rispondere.
- `config.yml` — contiene la configurazione della pipeline di elaborazione del linguaggio naturale (NLU) e delle politiche di dialogo, ovvero i modelli e gli algoritmi utilizzati per l’addestramento.
- `data/` — cartella dedicata ai dati di addestramento:
 - `nlu.yml` — esempi di frasi per ogni intento e annotazioni per le entità.
 - `stories.yml` — esempi di conversazioni complete per addestrare il flusso dialogico.
 - `rules.yml` — regole predefinite che associano un contesto a una risposta o azione specifica.
- `actions/` — cartella contenente le *custom actions* (in Python) che permettono di integrare logica esterna o chiamate API (ad esempio OpenWeather). Il file `__init__.py` è necessario per rendere il modulo importabile, mentre `actions.py` contiene l’implementazione delle azioni personalizzate.
- `models/` — directory in cui vengono salvati i modelli addestrati di Rasa.
- `tests/` — include test automatici per validare il comportamento del chatbot, come ad esempio `test_stories.yml`.
- `credentials.yml` — configurazione delle credenziali per connettersi a canali esterni (es. Telegram, Slack).
- `endpoints.yml` — definisce gli endpoint utilizzati da Rasa (es. server di azioni, tracker store, broker di eventi).

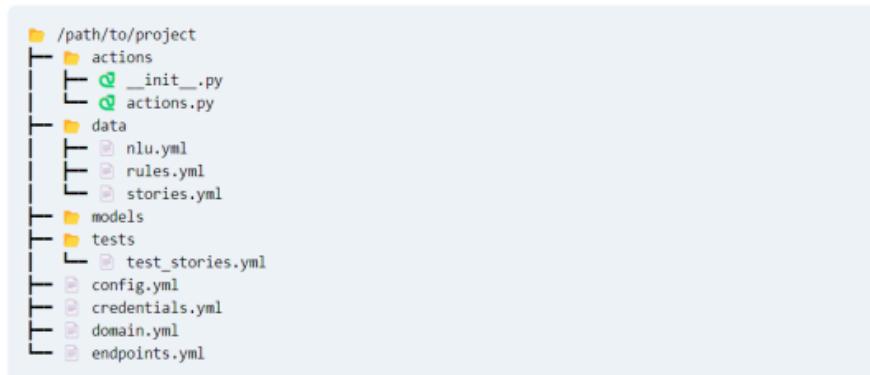


Figure 3.1: RASA - Struttura dei file.

Intent

Nel *MeteoAssistantBot*, gli intent rappresentano le intenzioni dell’utente, espresse tramite frasi o comandi, che il sistema deve riconoscere e interpretare per attivare la risposta corretta. Ogni intent è associato a un insieme di *utterance* di esempio, utili per addestrare il modello NLU a gestire le diverse formulazioni di una stessa richiesta. All’interno di queste frasi possono essere presenti entità (*entities*) come `city`, `date` o `activity`, le quali forniscono dettagli aggiuntivi per personalizzare l’output.

La Tabella 3.1 riassume gli intent principali implementati, con una breve descrizione delle tipologie di frasi di esempio che li caratterizzano.

Table 3.1: Intent principali del *MeteoAssistantBot* e loro descrizione

Intent	Descrizione
<code>Start</code>	Avvia o riavvia la conversazione, mostrando il messaggio di benvenuto all’utente.
<code>Help</code>	Mostra le funzionalità disponibili e le modalità di interazione con il bot.
<code>Chiedi meteo</code>	Richiesta di informazioni meteorologiche senza specificare luogo o data.
<code>Dammi meteo</code>	Richiesta di previsioni meteorologiche con indicazione di luogo e/o periodo.
<code>Consiglio abbigliamento</code>	Suggerimenti su cosa indossare in base alle condizioni meteorologiche.
<code>Consiglio attività</code>	Raccomandazioni su attività all’aperto in funzione del meteo.
<code>Qualità dell’aria</code>	Informazioni sullo stato dell’aria e sull’indice AQI di una località.
<code>Orari sole</code>	Indicazione dell’orario di alba e tramonto per una determinata città e data.
<code>Panoramica città</code>	Breve descrizione informativa della città indicata, con cenni culturali e turistici.
<code>Negazione</code>	Rifiuto o interruzione di un suggerimento o di un’azione proposta dal bot.

Tutti gli intent sono codificati in formato YAML, consentendo un’agevole estensione o modifica delle interazioni supportate senza intervenire sull’architettura complessiva del sistema.

Slot e Forms

Nel *MeteoAssistantBot*, gli **slot** rappresentano contenitori di informazioni estratte dal testo dell’utente e utilizzate per personalizzare la risposta. Ogni slot è associato a uno specifico

tipo di dato (ad esempio testo, numero, data) e dispone di **mappature** (*mappings*) che definiscono come il valore deve essere acquisito.

Le mappature possono provenire da:

- **Entity** (`from_entity`) — estrae il valore direttamente dall’entità individuata dal motore NLU.
- **Intent** (`from_intent`) — assegna un valore predefinito quando l’utente esprime un determinato intento.

Nella tabella 3.2 sono riportati gli slot principali implementati:

Table 3.2: Slot del MeteoAssistantBot e relative mappature

Slot	Descrizione e Mappature
<code>city</code>	<p>Nome della città per cui ottenere previsioni, consigli o altre informazioni.</p> <p>Mapping:</p> <ul style="list-style-type: none"> • <code>from_entity (city)</code>.
<code>date</code>	<p>Data di riferimento per le previsioni o suggerimenti.</p> <p>Mapping:</p> <ul style="list-style-type: none"> • <code>from_entity (date)</code> • <code>from_intent (chiedi_meteo) → “oggi”</code> • <code>from_intent (dammi_meteo) → “oggi”</code>
<code>activity</code>	<p>Tipo di attività all’aperto per cui valutare le condizioni meteo.</p> <p>Mapping:</p> <ul style="list-style-type: none"> • <code>from_entity (activity)</code>.

Per garantire che le informazioni minime necessarie vengano sempre raccolte, il bot utilizza i **forms**, che definiscono un insieme di slot obbligatori da compilare durante l’interazione. Un esempio è il `weather_form`, illustrato nella tabella 3.3:

Table 3.3: Form `weather_form` e slot richiesti

Slot richiesto	Funzione
<code>city</code>	Identifica la località per cui richiedere le previsioni meteo.

date	Specifica il giorno di interesse per le previsioni.
-------------	---

Con questa configurazione, quando l'utente chiede informazioni meteo senza fornire esplicitamente città o data, il bot attiva il `weather_form` e interagisce per ottenere i dati mancanti prima di chiamare le API meteorologiche.

Rules

Nel framework *Rasa*, le **rules** rappresentano meccanismi di risposta deterministici che collegano un evento conversazionale (come il riconoscimento di un *intent* o la compilazione di un form) a una sequenza fissa di azioni. A differenza delle *stories*, che modellano dialoghi più complessi e flessibili, le regole sono utilizzate per gestire comportamenti standard, ricorrenti e poco ambigui, garantendo che a determinati input corrisponda sempre la stessa uscita.

Nel progetto *MeteoAssistantBot* le regole sono organizzate in tre categorie principali:

1. **Comandi di servizio** — risposte immediate a comandi standard come `/start` e `/help`.
2. **Gestione del form meteo** — attivazione, compilazione e chiusura del `weather_form`.
3. **Azioni tematiche** — esecuzione diretta di *custom action* su specifici intent.

1. Comandi di servizio

Queste regole intercettano comandi basilari inviati dall'utente e restituiscono risposte predefinite, senza necessità di ulteriori elaborazioni.

```
- rule: /start -> benvenuto
  steps:
    - intent: start
    - action: utter_start

- rule: /help -> guida rapida
  steps:
    - intent: help
    - action: utter_help

- rule: chiusura conversazione
  steps:
    - intent: deny
    - action: utter_goodbye
```

2. Gestione del form meteo

Queste regole avviano il `weather_form` quando l'utente chiede informazioni meteo, raccolgono gli slot necessari (`city` e `date`) e, una volta completato, eseguono la sequenza di azioni per ottenere e mostrare il risultato.

```

- rule: chiedi_meteo -> avvia form
  steps:
    - intent: chiedi_meteo
    - action: weather_form
    - active_loop: weather_form

- rule: dammi_meteo -> avvia form
  steps:
    - intent: dammi_meteo
    - action: weather_form
    - active_loop: weather_form

- rule: completamento weather_form
  condition:
    - active_loop: weather_form
  steps:
    - action: weather_form
    - active_loop: null
    - action: utter_submit_weather_form
    - action: action_get_weather
    - action: utter_ask_more

```

3. Azioni tematiche

In questo gruppo ogni intent è collegato direttamente a una *custom action* specifica, seguita dalla domanda `utter_ask_more` per proseguire la conversazione.

```

- rule: consiglio abbigliamento
  steps:
    - intent: ask_clothing_advice
    - action: action_clothing_advice
    - action: utter_ask_more

- rule: consiglio attività all'aperto
  steps:
    - intent: ask_activity_advice
    - action: action_activity_advice
    - action: utter_ask_more

- rule: qualità dell'aria
  steps:
    - intent: ask_air_quality
    - action: action_get_air_quality
    - action: utter_ask_more

- rule: orari alba e tramonto
  steps:
    - intent: ask_sun_times
    - action: action_get_sun_times
    - action: utter_ask_more

- rule: panoramica città

```

```

steps:
- intent: ask_city_overview
- action: action_get_attractions
- action: utter_ask_more

```

Nota: la presenza di `utter_ask_more` alla fine di molte regole garantisce una continuità di dialogo, permettendo all'utente di proseguire con nuove richieste senza dover riavviare la conversazione.

Lookup tables

Le *lookup tables* (tabelle di ricerca) sono elenchi predefiniti di parole o frasi utilizzati per associare l'input dell'utente ad un'entità o intenzione specifica. Queste tabelle consentono di rendere il modello di comprensione del linguaggio naturale più robusto e accurato, poiché riescono a catturare variazioni, sinonimi e forme alternative della stessa espressione.

Nel nostro caso specifico sono state utilizzate tre *lookup tables*:

- **city**: tabella contenente tutti i nomi di città supportati, inclusi eventuali sinonimi o varianti linguistiche.
- **date**: tabella contenente tutte le date o intervalli temporali accettati in input, con diverse modalità di espressione (ad esempio “oggi”, “domani”, “prossimo weekend”).
- **activity**: tabella contenente tutte le tipologie di attività possibili, con sinonimi e termini affini (ad esempio “escursionismo”, “passeggiata”, “visita guidata”, “attività sportiva”).

Grazie all'utilizzo di queste tabelle di ricerca, il sistema è in grado di riconoscere in modo più efficace i riferimenti a luoghi, date e attività, anche quando l'utente utilizza terminologia diversa da quella presente nei dati di addestramento, migliorando così l'accuratezza dell'interpretazione.

Azioni personalizzate (Actions)

Nel framework Rasa, le **azioni personalizzate** (*custom actions*) rappresentano la componente che esegue logica applicativa esterna, integrando API, database o algoritmi complessi. Dopo che il modulo NLU ha identificato **intent** ed **entità** nel messaggio dell'utente, e Rasa Core ha deciso quale azione eseguire, viene invocata la **Action** corrispondente, che elabora i dati necessari e genera la risposta finale.

Nel presente progetto, le actions sono state implementate in Python tramite il pacchetto `rasa_sdk`. Esse integrano:

- il servizio **OpenWeather** per meteo, previsioni e qualità dell'aria;
- un **dataset turistico** (derivato da Kaggle) per informazioni su destinazioni europee;
- logiche di **analisi e scoring** per generare consigli contestuali (abbigliamento, attività, attrazioni).

Componenti comuni come il client **OpenWeatherClient**, la gestione di fusi orari e il parsing di date/ora sono riutilizzati trasversalmente da più actions.

Client meteo: OpenWeatherClient

Fornisce un’interfaccia centralizzata verso l’API OpenWeather (`/data/2.5`) utilizzando `requests.Session` e parametri globali (API key, unità metriche, lingua italiana). Espone i metodi:

1. `get_current(city)` — condizioni attuali (`weather`);
2. `get_forecast(city)` — previsioni a 5 giorni/3h (`forecast`);
3. `get_air_pollution(lat, lon)` — qualità dell’aria (`air_pollution`).

Le eccezioni sono catturate e loggiate; in caso di errore i metodi restituiscono `None`.

ActionGetWeather

Recupera e restituisce condizioni meteo attuali o previsioni future per una città, con eventuale arricchimento da metadati turistici:

1. Determina il tipo di richiesta in base allo slot `date`:
 - `oggi/ora/adesso` → condizioni attuali;
 - altrimenti → previsioni filtrate per la data target.
2. Utilizza il `OpenWeatherClient` per interrogare l’API e applica correzioni di fuso orario.
3. Integra regione e paese dal dataset turistico, se presenti.
4. Compone un testo con descrizione, temperatura, umidità, vento, visibilità e copertura nuvolosa, arricchito da *emoji* coerenti.

ValidateWeatherForm

Valida lo slot `city` prima di procedere:

1. Interroga l’endpoint `weather` con la città fornita;
2. in caso di codice **200**, accetta lo slot;
3. in caso di **404**, notifica `utter_invalid_city` e azzera lo slot;
4. per timeout o altri errori, notifica `utter_weather_unavailable`.

ActionClothingAdvice

Genera consigli di abbigliamento in base alle previsioni per città e data:

1. Determina il giorno target e segmenta la giornata in mattino, pomeriggio e sera.
2. Calcola, per ciascuna fascia, medie di temperatura e vento e individua la descrizione prevalente.
3. Applica regole predefinite per suggerire capi e accessori adeguati, aggiungendo raccomandazioni per vento, caldo estremo o precipitazioni.
4. Restituisce un testo strutturato per l’utente finale.

ActionGetAirQuality

Fornisce un quadro della qualità dell'aria nella città richiesta:

1. Recupera coordinate geografiche via `weather`;
2. Interroga l'endpoint `air_pollution`;
3. Traduce l'indice AQI in etichette italiane e valuta i principali inquinanti (`CO`, `NO2`, `O3`, ecc.) rispetto a soglie qualitative;
4. Restituisce un elenco con valori, giudizio e breve descrizione di ogni componente.

ActionGetSunTimes

Comunica orari di alba e tramonto:

1. Interroga `weather` per `sys.sunrise`, `sys.sunset` e `timezone`;
2. Converte i timestamp in orario locale;
3. Restituisce un messaggio formattato (`HH:MM`).

ActionGetAttractions

Restituisce informazioni turistiche dal dataset europeo:

1. Normalizza le colonne principali e crea una chiave in minuscolo per le città;
2. Cerca la corrispondenza esatta su `city`;
3. Se trovata, compone un testo con categoria, posizione, descrizione, turisti annui, lingua, valuta, religione, piatti tipici, periodo consigliato, costo della vita, sicurezza e rilevanza culturale;
4. Se assente, informa l'utente della mancanza di dati.

ActionActivityAdvice

Fornisce un verdetto sull'opportunità di svolgere un'attività, basato sulle condizioni meteo previste:

1. Determina la data e l'orario target, considerando espressioni relative e indizi temporali;
2. Recupera condizioni attuali o previsioni dal `OpenWeatherClient`;
3. Estraie indicatori (temperatura, percepita, vento, umidità, precipitazioni);
4. Applica una logica di scoring che restituisce uno stato (`ok`, `caution`, `no`) con motivazioni e consigli pratici;
5. Suggerisce alternative indoor/outdoor in caso di meteo sfavorevole.

Queste azioni, pur differendo negli obiettivi, condividono un'architettura modulare basata su componenti riusabili e gestione robusta degli errori. La localizzazione in lingua italiana e l'uso coerente di unità metriche assicurano risposte naturali e comprensibili, mentre l'integrazione di fonti esterne e dataset dedicati aumenta la rilevanza e il valore informativo delle interazioni con l'utente.

4 Conessione a Telegram & Test

4.1 Telegram

Nella fase finale del progetto il chatbot è stato collegato a una piattaforma di messaggistica istantanea, così da renderlo accessibile anche da smartphone.

La scelta è ricaduta su **Telegram** per un motivo:

- la possibilità di integrarsi facilmente con il servizio **ngrok**, utilizzato per eseguire il chatbot su server locale e renderlo accessibile dall'esterno.

Il bot è stato creato tramite *BotFather*, l'assistente ufficiale di Telegram per la generazione di bot in pochi passaggi. La Figura 4.1 mostra l'interfaccia di creazione.

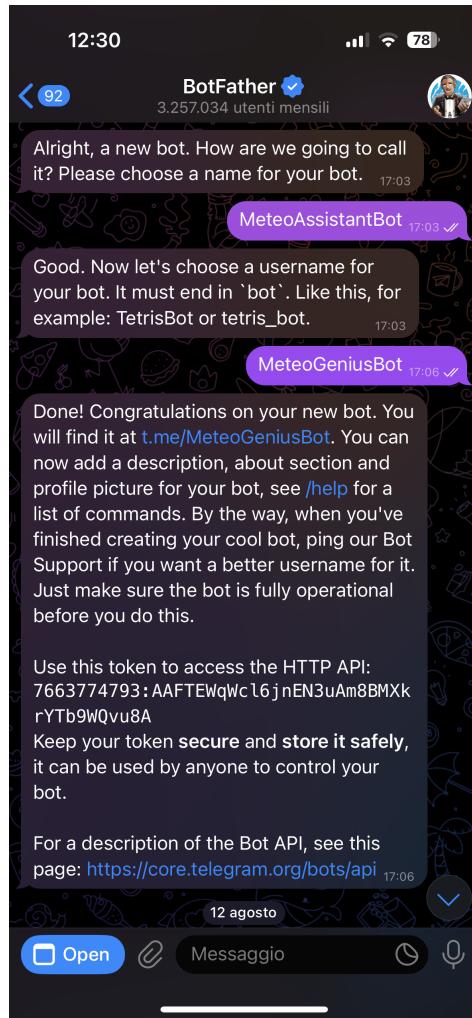


Figure 4.1: Creazione del bot tramite BotFather

Dopo la creazione, è stato necessario configurare i parametri di connessione nel file `credentials.yml` di Rasa, come mostrato in Figura 4.2. In questa configurazione sono presenti tre campi fondamentali:

- **access_token**: il token fornito da *BotFather* al momento della generazione del bot su Telegram;
- **verify**: il nome utente (*username*) del bot, definito anch'esso durante la fase di creazione;
- **webhook_url**: l'endpoint che consente la comunicazione tra il chatbot e il server locale, generato attraverso l'utilizzo di *ngrok*.

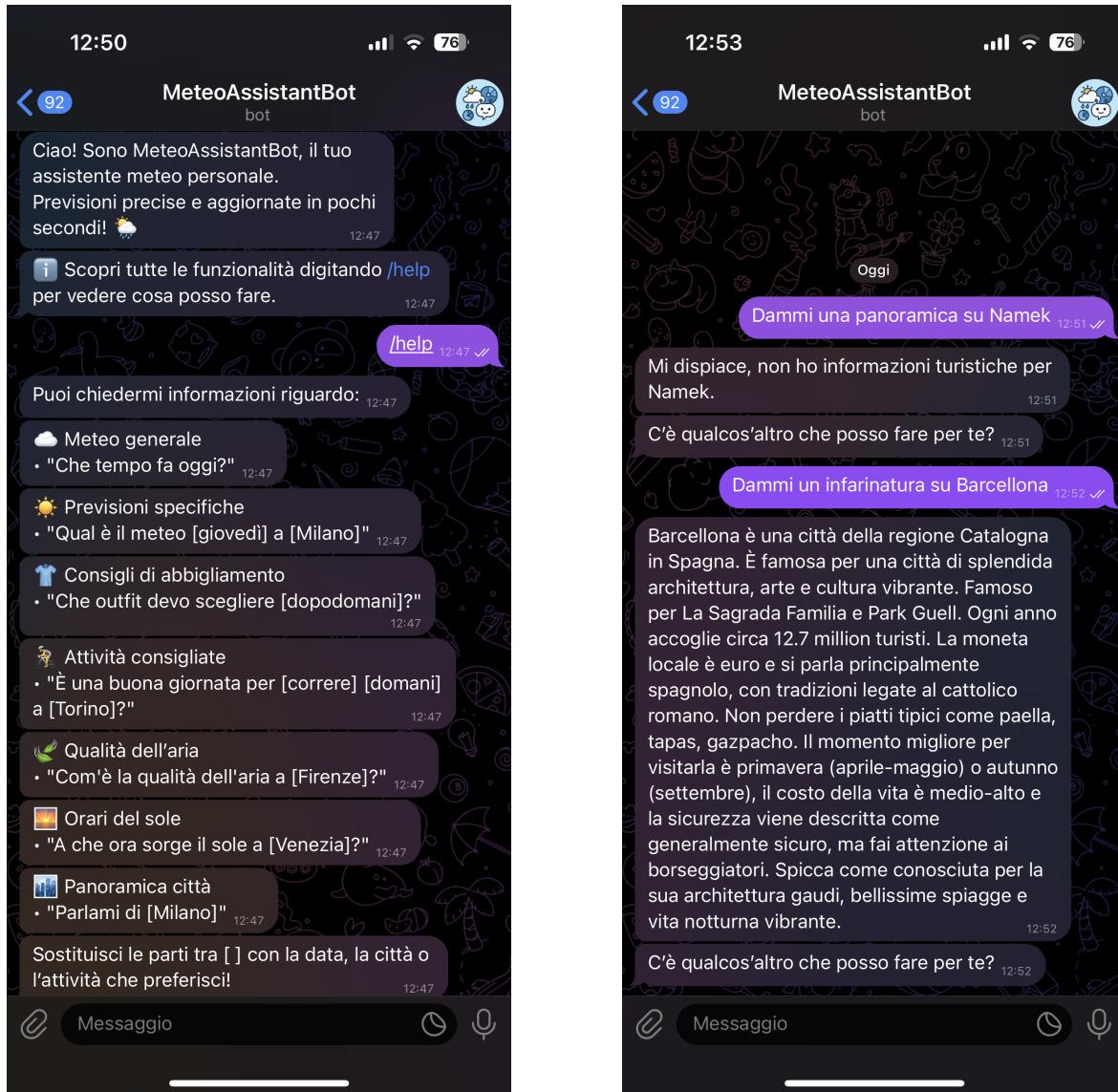
```
telegram:  
  access_token: "7663774793:AAFTEWqWcl6jnEN3uAm8BMXkrYTb9WQvu8A"  
  verify: "MeteoGeniusBot"  
  webhook_url: "https://8993d8f309e0.ngrok-free.app/webhooks/telegram/webhook"
```

Figure 4.2: Parametri configurati per la connessione del chatbot a Telegram

Grazie a questa configurazione, il chatbot può ricevere ed elaborare i messaggi inviati dagli utenti su Telegram, garantendo un'interazione semplice e immediata anche da dispositivi mobili.

4.2 Test - MeteoAssistantBot

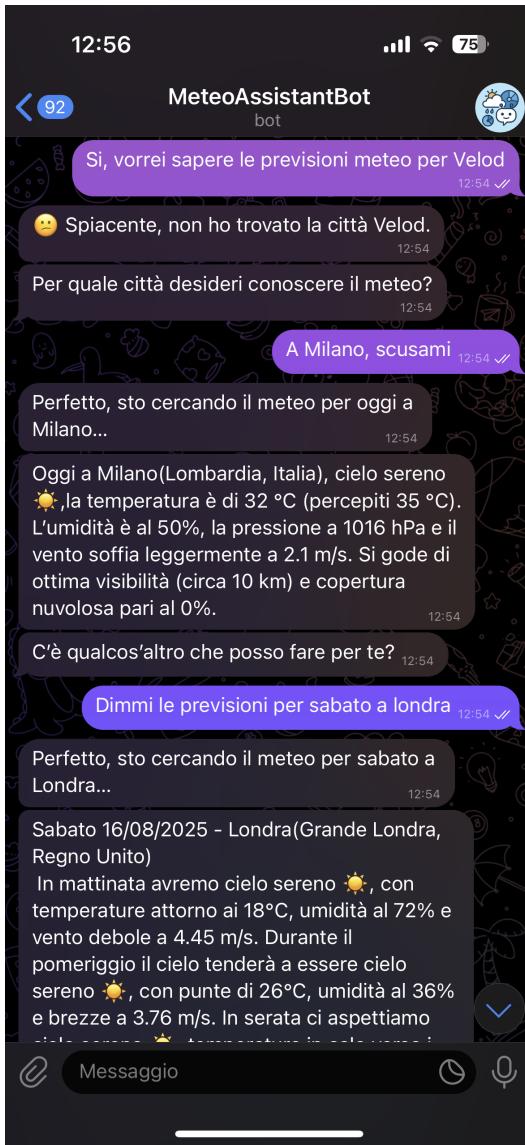
In questo capitolo vengono presentate alcune schermate acquisite durante una sessione di prova del chatbot su **Telegram**, con l'obiettivo di illustrare in modo pratico il suo funzionamento. Tali esempi visivi permettono di evidenziare concretamente i risultati ottenuti e di mettere in risalto gli aspetti discussi nel capitolo precedente.



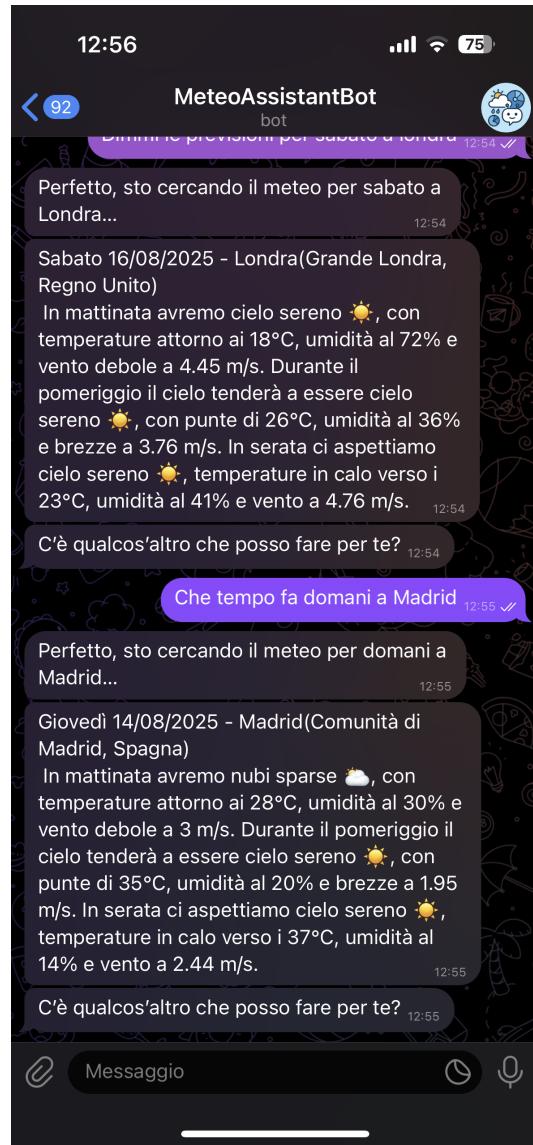
(a) Interazione iniziale con il MeteoAssistantBot su Telegram.

(b) Richiesta di informazioni turistiche, con esempio positivo e negativo.

Figure 4.3: Esempi di conversazioni con il MeteoAssistantBot su Telegram.



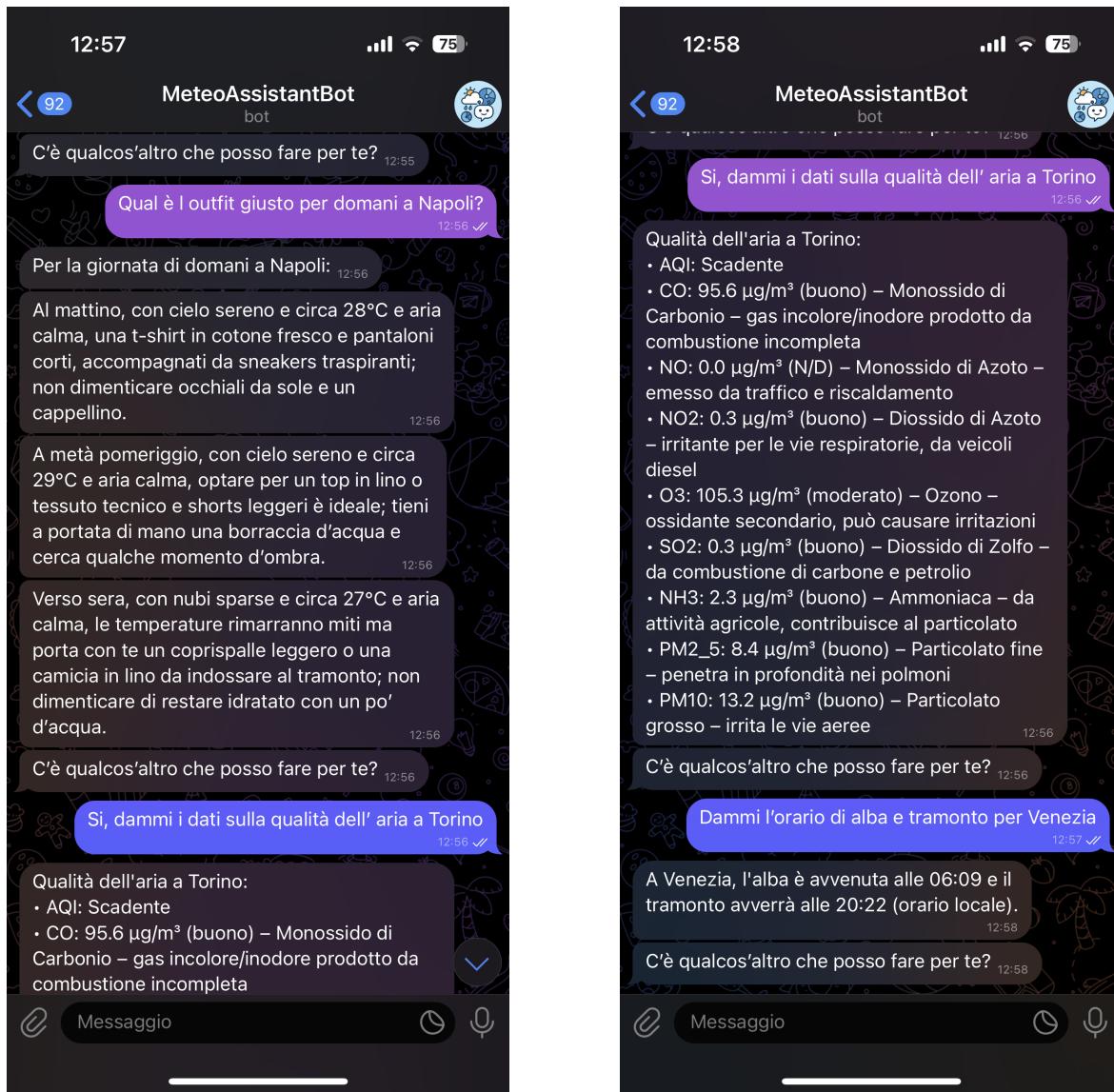
(a) Esempio di richiesta meteo con correzione della città



(b) Esempio di richiesta meteo per date e località differenti

Figure 4.4: Esempi di interazione con il bot per ottenere previsioni meteorologiche precise e personalizzate.

Nelle figure 4.4a e 4.4b sono mostrati due scenari di utilizzo del MeteoAssistantBot: il primo con la correzione di una città non trovata e il secondo con la consultazione del meteo in date e luoghi differenti.



(a) Esempio di suggerimento outfit per Napoli e verifica qualità dell'aria a Torino

(b) Richiesta dati qualità dell'aria a Torino e orari di alba/tramonto per Venezia

Figure 4.5: Esempi di interazione del chatbot con funzionalità di suggerimenti e dati ambientali

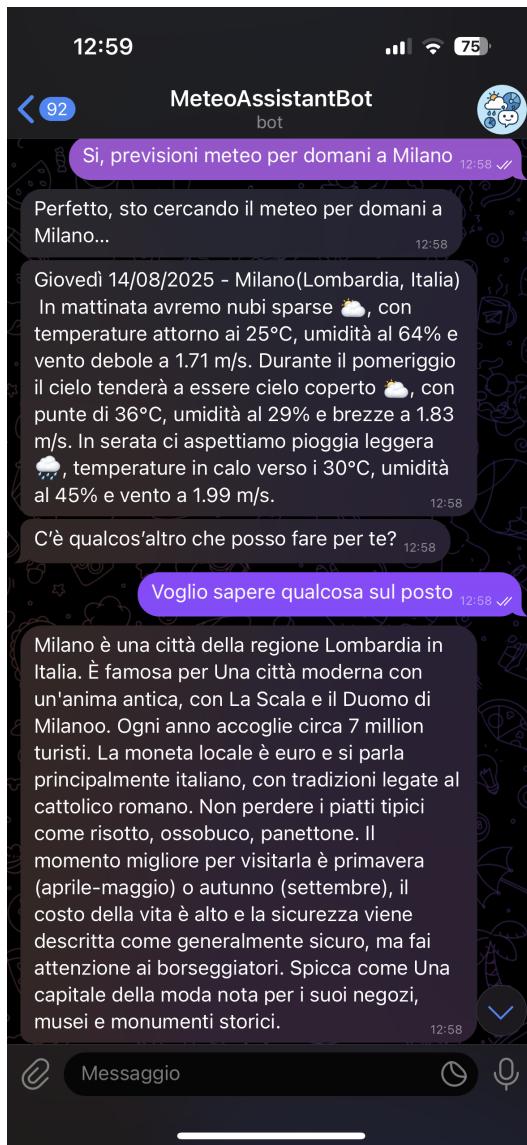


Figure 4.6: Esempio di interazione con richiesta di previsioni meteo dettagliate per una città e informazioni turistiche generali su quella città.

In questo esempio il chatbot utilizza uno *slot* per memorizzare il nome della città fornita dall'utente. Questo meccanismo consente di mantenere il contesto tra le richieste, evitando che l'utente debba reinserire più volte lo stesso dato. Ad esempio, dopo aver chiesto le previsioni meteo per *Milano*, il sistema ricorda automaticamente la città anche per richieste successive, come la descrizione turistica o altre informazioni correlate.

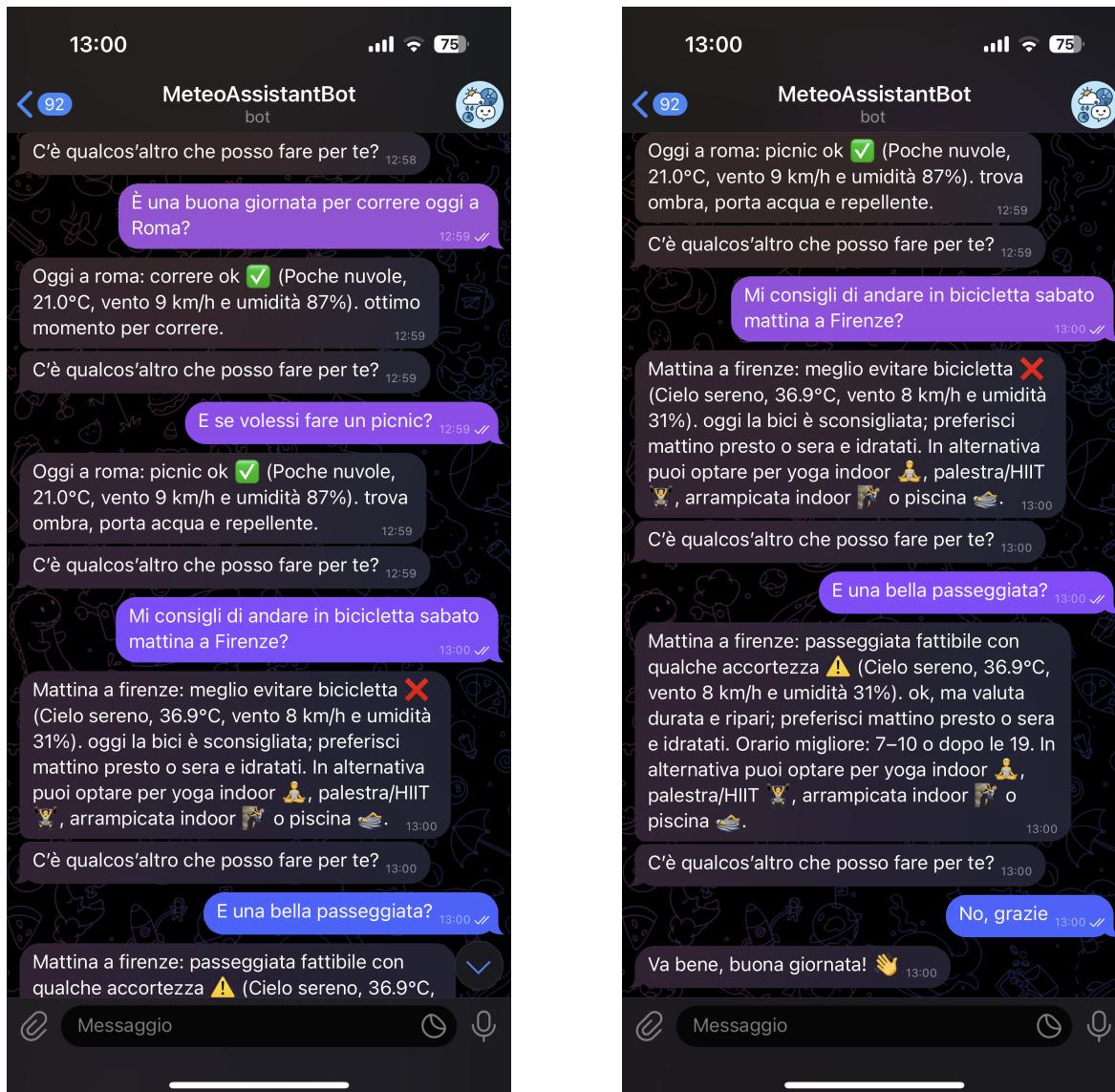


Figure 4.7: Interazioni del chatbot relative alla valutazione delle condizioni meteo per diverse attività all'aperto.

5 Conclusioni e sviluppi futuri

In questo progetto è stato utilizzato il framework **RASA** per realizzare un chatbot intelligente in grado di fornire consigli personalizzati basati sulle condizioni meteorologiche. Il sistema guida l'utente nella scelta di attività e abbigliamento adatti, integrandosi con un'API meteo per ottenere dati in tempo reale e restituendo risposte contestualizzate. Le funzionalità implementate (descritte nei capitoli precedenti) comprendono:

- analisi del linguaggio naturale per identificare intenzioni (*intent*) e parametri (*entity*);
- logiche di dialogo per gestire interazioni complesse;
- integrazione con **Telegram** per consentire un utilizzo immediato anche da smartphone.

Come possibili sviluppi futuri, si prevede di:

- estendere il database di risposte e scenari supportati;
- aggiungere funzionalità per la memorizzazione delle preferenze dell'utente;
- integrare altre fonti di dati (es. indice UV, eventi locali);
- migliorare la gestione delle variabili contestuali per fornire suggerimenti ancora più mirati.



[Link al repository GitHub](#)