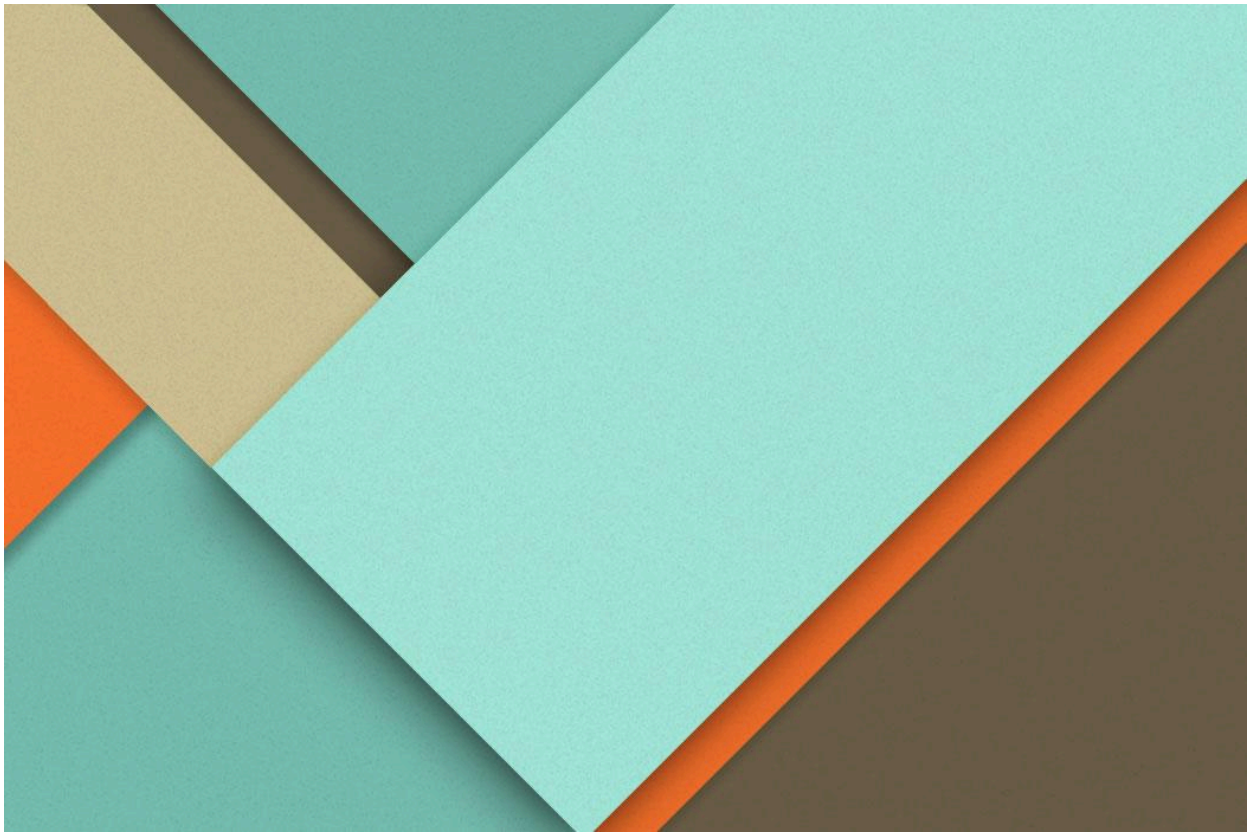


Resume Evaluation & Optimization

05-05-2025

Greeshma Nerella	sxn210104
Tanishq Nimale	tsn230000
Hamid Rouhani	hxr220010
Siddhu neehal rapeti	sxr230189

Contributions

Self-Evaluation

Tanishq Nimale

80 points	significant exploration beyond baseline	Implemented NLP concepts like fine-tuning, TF-IDF, RAG, Entity Extraction for Scoring Resume and generating specific feedback.
10 points	highlighted complexity	Built a crawler and a scraper along with a team member to get over 1600 Job descriptions from Linkedin and over 300 Resumes.
10 points	exceptional visualization	Our project comes with an end-to-end beautiful UI built in Next.js, including charts, tables and more.
10 points	discussion of lessons learned and potential improvements	Presented a Live demo in class(May 6th) where our team discussed setbacks, potential issues and future scope.
10 points	Innovation or Creativity (10/30 because failed attempt)	Attempted to Fine tune bert base model based on 2600 labelled dataset to classify a job match and generate some feedback.

Siddhu neehal rapeti

80 points	significant exploration beyond baseline	JSON ETL on the 1600 scraped linkedin job description to extract and create: YAKE 1–3 grams + POS buckets; TF-IDF diff; SentenceTransformer + cosine similarity embedding(all-MiniLM retrieval)
20 points	Innovation	Hybrid RAG (sparse + dense) · Resilient fallbacks · CLI-first automation
7 points	highlighted complexity	Sparse/dense integration, rule-based verb upgrade + single-keyword backup and multiple fall back options in case LLM fails.
7 points	Testing	Tested it with 3 of my undergrad peers, one grad peer and one ITS student from JSOM.
10 points	discussion of lessons learned and potential improvements	Presented a Live demo in class(May 6th) where our team discussed setbacks, potential issues and future scope.

Greeshma Nerella

80 points	significant exploration beyond baseline	Implemented NLP concepts like TF-IDF, RAG, Entity Extraction for Scoring Resume Impact and generating specific feedback. As well as learning to use various open source LLMs.
10 points	highlighted complexity	Built a crawler and a scraper along with a team member to get over 1600 Job descriptions from Linkedin. Also implemented authentication and security via google's firebase services.
10 points	exceptional visualization	Designed and prototyped a UI in figma, realized using Next.Js and tailwind css. Integrated functionality into a seamless user flow.
10 points	discussion of lessons learned and potential improvements	Presented a Live demo in class(May 6th) where our team discussed setbacks, potential issues and future scope.

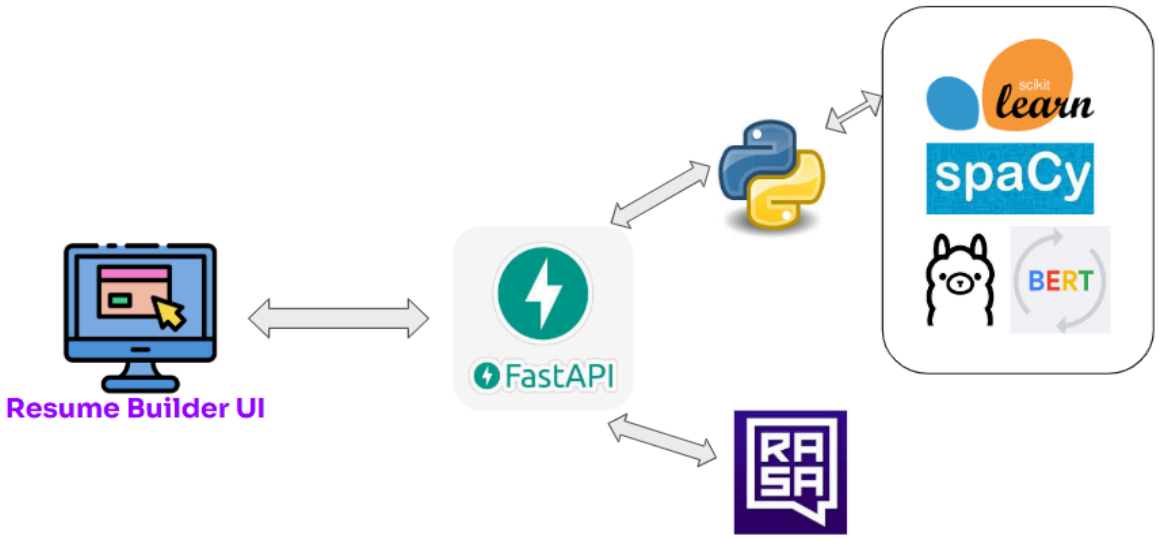
Hamid Rouhani

80 points	Significant exploration	Designed a complete Rasa pipeline with slot/form logic, intent handling, and custom actions for RPC and optimization, integrating all components via FastAPI for seamless chatbot–optimizer–UI communication.
10 points	Highlighted complexity	Developed dialogue models with entity-slot mapping for long-form input, high-quality intent examples, and intent disambiguation, ensuring full Rasa–backend integration and robust handling of unstructured input.
10 points	Production Readiness and Deployment	Engineered a production-ready system using Rasa Pro/Studio with modular, extensible, and structured dialogue control, resulting in a scalable management system exceeding traditional rule-based approaches.
10 points	Exceptional Intent Classification and Dialogue Management	Built a complete Rasa NLU pipeline with tokenizers, featurizers, DIETClassifier, and multi-policy control, enabling accurate intent/entity handling, fallback support, and production-ready dialogue management without relying on LLMs.
10 points	discussion of lessons learned and potential improvements	Presented a Live demo in class(May 6th) where our team discussed setbacks, potential issues and future scope.

Overview

This project presents a NLP pipeline for resume optimization using real-world job data scraped from LinkedIn. It begins with a two-part scraper combining Selenium and BeautifulSoup to collect over **1,500 job postings**, extracting structured data like titles, companies, and descriptions. Using this data, a **skill-based scoring** system applies **TF-IDF** to assess how well a resume aligns with job requirements. An additional **impact scoring** module evaluates action verbs and quantifies achievements using SpaCy and TF-IDF. For personalized job-experience feedback, two approaches are used: a fine-tuned **BERT** classifier and a retrieval-augmented generation **(RAG)** system powered by a local DeepSeek LLM. The pipeline includes a dense-vector retrieval engine for job-resume similarity and keyword extraction, with resume rewriting handled by LLMs through structured prompts and fallback models. This end-to-end system enables actionable resume improvements tailored to specific job roles. We used a **RASA** based chatbot to have a chatbot conversation to evaluate and optimize the Resume.

Architecture



Github Repository

https://github.com/mrfrozen97/nlp-resume-builder

Youtube Video

https://www.youtube.com/watch?v=w94-_X9GRA8

LinkedIn Job Scraper Pipeline (Selenium + BeautifulSoup)

This two-part system automates the process of collecting job listings from LinkedIn and extracting structured job data. It combines **Selenium** for dynamic interaction and **BeautifulSoup** for HTML parsing. ***"data/script/linkedin/link-scraper.py"***

Part 1: Job Link Collection with Selenium

Automates LinkedIn login and navigates to the Jobs section to search for roles (e.g., *DevOps*, *Data Engineer*). For each search:

- **Keyword Search**: Inputs a user-defined term like "DevOPS".
- **Link Scraping**: Iterates through job cards, clicks them, extracts job URLs, and saves them.
- **Pagination**: Automatically goes to the next page until a target number of links is collected.
- **Error Handling**: Resilient to missing elements and click errors.

Output: Saves job URLs to role-specific files like `links/devops_links.txt`. Over **1500 Links Scraped**.

Part 2: Job Data Extraction with BeautifulSoup

Parses saved job links to extract structured data from the job detail pages.

- **HTML Parsing**: Loads each job URL using `requests` + `BeautifulSoup`.

- **Field Extraction**: Captures:
    - Job Title
    - Company Name
    - Job Description
    - Location
- **Remove Duplicates**: Skips previously scraped jobs using job ID caching.
- **Storage**: Appends structured data to JSON files by role.

Output: Updates datasets like `data/devops_data.json`. Scraped over **1500 Job Descriptions**.

Resume Skills Scoring

This project introduces a skill-based resume scoring system that evaluates how well a resume aligns with a specific job description (JD), using over **1,500 scraped LinkedIn job descriptions** as training data.

The system uses a curated list of Computer Science skills and applies **TF-IDF weighting** to measure the importance of each skill within a JD. The resume is then analyzed to identify which of these weighted skills it contains. The final **score** reflects the overlap between JD-required skills and resume-present skills, normalized by the JD's total skill weight.
Code present in directory: ***"fastapi-backend/score_resumes.py"***

The goal is to provide both a numeric **impact score** and actionable insights to improve resumes for targeted job applications.

TF-IDF Training:

- The system is trained on job descriptions using a **TF-IDF Vectorizer** restricted to the predefined skills vocabulary.
- The result is a skill-weight mapping, saved as a `.csv` and a serialized vectorizer (`.pkl`).

Resume Scoring:

- A resume is scored by extracting its relevant skills and comparing them with the JD's TF-IDF skill weights.
- The score is **normalized**, indicating how closely the resume matches the JD's prioritized skills.
- Additionally, it outputs:
    - `Matched Skills`: Skills present in both resume and JD, sorted by importance.
    - `Missing Skills`: Important JD skills absent from the resume.

Resume Impact Scoring

The other component of the overall resume evaluation is the impact of the resume. This impactfulness of content is computed using a hybrid scoring approach, considering both the usage of **action words** and **quantifying** achievements.

The core idea for this approach stems from the prior knowledge that use of **certain verbs** called action words draws a more impactful picture of a candidate's experiences. Alongside this, both for human recruiter and ATS evaluation, the use of numbers makes for a stellar resume. An example would be "[Action Word] a system by [Method] to achieve X% improvement in performance". This is made possible through a **TF-IDF vectorizer** trained on a curated list of **250+ action verbs** and **SpaCy's entity recognition** for quantification of work.

This idea is implemented in **"fastapi-backend/score_impact.py"**. The goal is to ensure that experience bullets are concise, active, and quantifiable.

TF-IDF Training:

- The system uses a **TF-IDF vectorizer** trained on a predefined dataset of resumes from students and professionals in technology, with the vocabulary restricted to a curated list of tech-specific action verbs.

- The result is a verb-weight mapping, saved as a `.csv` and a serialized vectorizer (`.pkl`).

Impact Scoring:

- A resume is scored by extracting verbs using SpaCy's part-of-speech (POS) tagging. These verbs are then evaluated using a TF-IDF vectorizer to measure the importance of the action words used. The resulting score is **normalized** to indicate how effectively the resume incorporates impactful language.

- Additionally, the quantification of experiences is calculated using SpaCy's NER to detect numerical entities. A higher number of numerical entities boosts the quantification score.

- Finally the two scores are combined through a **weighted sum**, with each score getting an equal weight of **0.5** to the final impact score.

Work Experience, Projects – Job Matching & Feedback System

This system evaluates how well a given **work experience & Projects aligns with a job description** using two complementary approaches:

Approach 1: Fine-Tuned BERT Classifier

We initially attempted to train a supervised classifier by **fine-tuning `bert-base-uncased`** using **2,600+ labeled data points**.

- **Labeling Strategy**:
  We generated training labels using the **DeepSeek-1.5B** model (via Ollama), which was prompted to classify whether a work experience matches a JD. This created a large pseudo-labeled dataset for fine-tuning.

- **Model Details**:
  - Binary classification head (Match / No Match)
  - Trained on Hugging Face with 90/10 train-test split
  - Tokenized pairs of JD and work experience
  - Accuracy validated using confidence scores from softmax output

Despite promising results, the model required high compute and lacked flexibility in generating nuanced feedback. Hence, the RAG-based Ollama method was instead.

Approach 2: Retrieval-Augmented Generation (RAG) using Ollama + DeepSeek

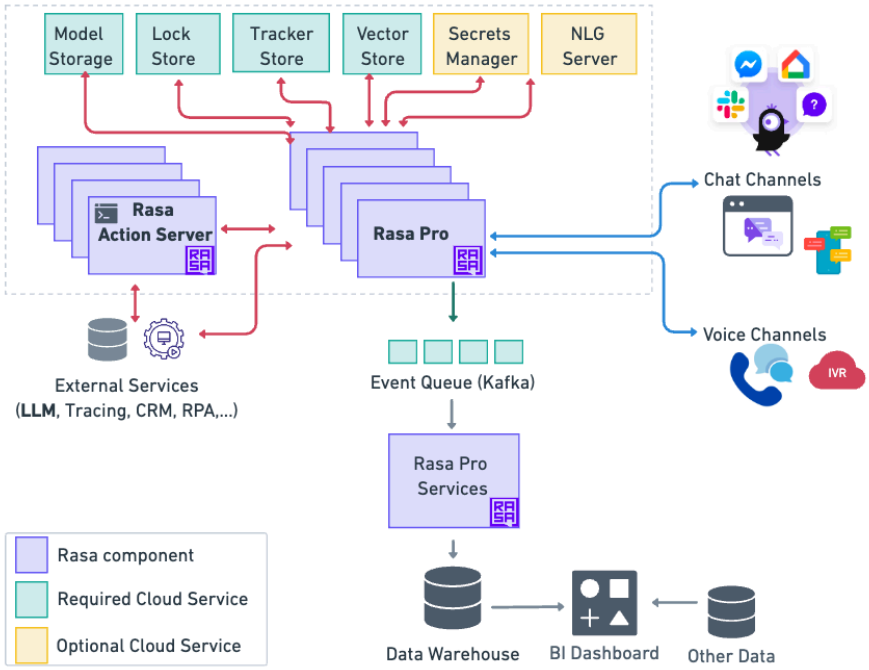
We built a class `WorkEX` that uses **Ollama** to locally run the **DeepSeek-1.5B** LLM for tasks:

1. **Binary Classification (Match / No Match)**
   It uses a prompt-driven RAG approach, feeding both the JD and work experience to the model. Based on the LLM's output, a simple keyword-based parser decides whether the work experience "matches" the JD.
2. **Feedback Generation**
   The same model is prompted to give **short, actionable feedback** on how to improve the work experience to better match the JD. This includes:
   - Missing **skills**
   - Lack of **impact language**

This technique does not require fine-tuning and provides immediate results thanks to the **zero-shot capabilities** of the LLM. (***"fastapi-backend/lib/worex/score_workex.py"***)

Chatbot Platform: Rasa

We used Rasa by RasaHQ to interpret user prompts and manage the dialogue flow.



Domain

In Rasa, the domain defines the universe in which the assistant operates. It specifies Intents, Responses, Entities, Slots, Forms, and Actions.

Intents

An intent represents the goal or purpose behind a user's message. We have defined these intents for our system:

greet, goodbye, add_experience, change_project_section, tailor_for_job, provide_job_title, ask_job_title, provide_company, ask_company, provide_skill, ask_skill, change_projects, optimize_resume, ask_score_resume, ask_score_impact, affirm, deny, bot_challenge.

Responses

Responses are predefined templated messages the assistant can send to users. We defined these responses for our system:

utter_greet, utter_did_that_help, utter_goodbye, utter_iamabot, utter_tailor_for_job, utter_ask_job_title, utter_ask_company, utter_resume_ready, utter_ask_add_more, utter_answer_job_title, utter_answer_company, utter_answer_skill, utter_answer_score_resume, utter_answer_score_impact, utter_answer_optimize_resume.

Entities

An entity is a specific piece of information extracted from a user's message. Entities provide context to the intent and help the assistant understand key details. We defined these entities for our system:

job_title, company, skill, optimized_resume, score_resume, score_impact.

Slots

Slots are variables that store information and act as the assistant's memory during a conversation. We defined these slots for our system:

optimized_resumejob_title, company, skill, score_resume, optimized_resume, score_impact.

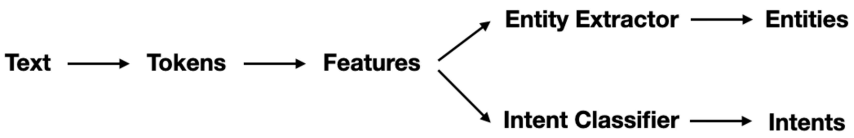
Actions

Actions are functions that can be triggered by dialogue policies to perform backend logic. We defined actions slots for our system:

action_job_title, action_company, action_skill, action_optimize_resume, action_score_resume, action_score_impact.

Pipeline

Rasa uses machine learning to make predictions about the users' intent as well as the next best action to take. Components like Tokenizers, Featurizers, Intent Classifiers, and Entity Extractors can form a pipeline.



There are components that we used in our pipeline:

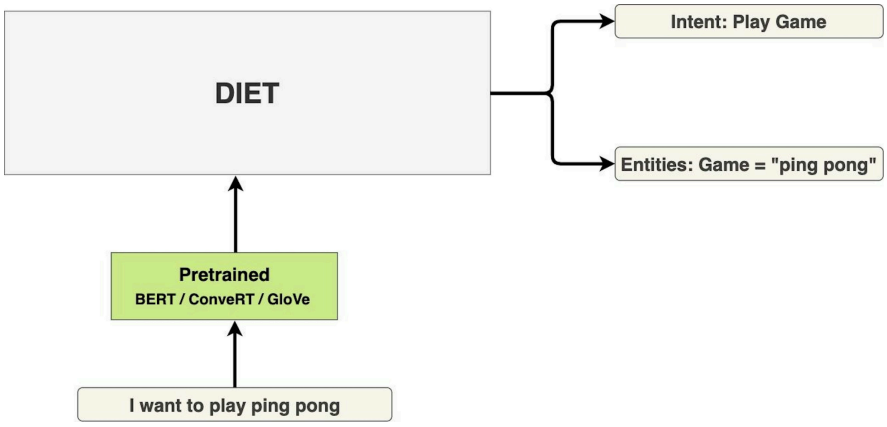
WhitespaceTokenizer: Creates a token for every whitespace separated character sequence.

RegexFeaturizer: Creates features for entity extraction and intent classification. During training the RegexFeaturizer creates a list of regular expressions defined in the training data format. For each regex, a feature will be set marking whether this expression was found in the user message or not.

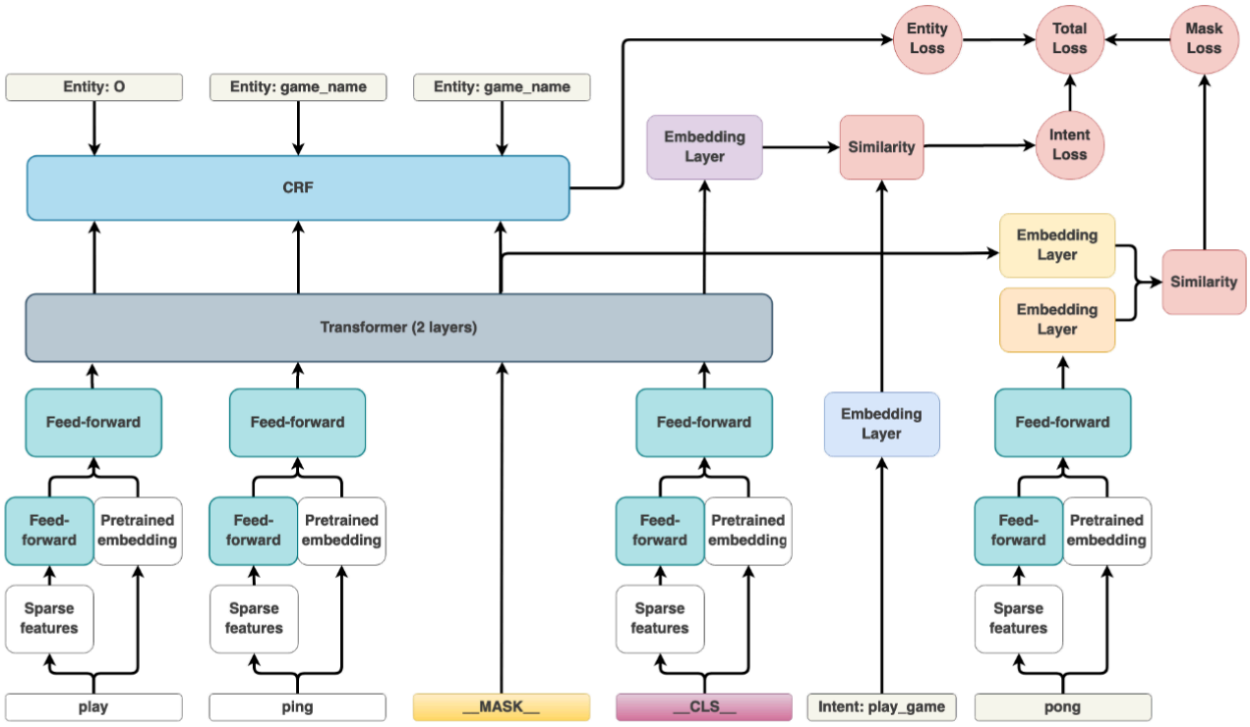
LexicalSyntacticFeaturizer: Creates features for entity extraction. Moves with a sliding window over every token in the user message and creates features according to the configuration.

CountVectorsFeaturizer: Creates features for intent classification and response selection. Creates bag-of-words representation of user message, intent, and response using sklearn's CountVectorizer.

DIETClassifier: DIET (Dual Intent and Entity Transformer) is a multi-task architecture for intent classification and entity recognition. The architecture is based on a transformer which is shared for both tasks. A sequence of entity labels is predicted through a Conditional Random Field (CRF) tagging layer on top of the transformer output sequence corresponding to the input sequence of tokens. This is a high level illustration of DIET:



The internal architecture of DIET is as follows.



A schematic representation of the DIET architecture. The phrase ”play ping pong” has the intent `play_game` and entity `game_name` with value ”ping pong”. Weights of the feed-forward layers are shared across tokens [1].

EntitySynonymMapper: If the training data contains defined synonyms, this component will make sure that detected entity values will be mapped to the same value.

ResponseSelector: Response Selector component can be used to build a response retrieval model to directly predict a bot response from a set of candidate responses. The prediction of this model is used by the dialogue manager to utter the predicted responses.

FallbackClassifier: The FallbackClassifier classifies a user message with the intent nlu_fallback in case the previous intent classifier wasn't able to classify an intent with a confidence greater or equal than the threshold of the FallbackClassifier.

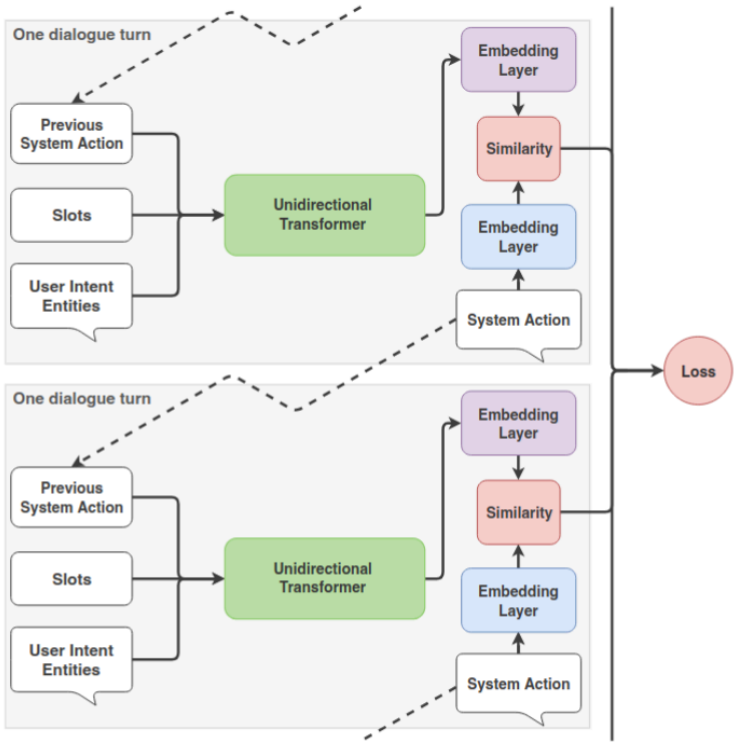
Policies

Rasa uses policies to decide which action to take at each step in a conversation. We used these policies for our system:

MemoizationPolicy: The MemoizationPolicy remembers the stories from training data. It checks if the current conversation matches the stories in the stories file. If so, it will predict the next action from the matching stories of the training data with a confidence of 1.0. If no matching conversation is found, the policy predicts None with confidence 0.0.

RulePolicy: The RulePolicy is a policy that handles conversation parts that follow a fixed behavior (e.g. business logic). It makes predictions based on any rules you have in the training data.

TEDPolicy: The TED Policy is a multi-task architecture used in Rasa for both next action prediction and entity recognition. It uses shared transformer encoders and a CRF layer for sequence labeling. For action prediction, it embeds dialogue context and actions into the same space and uses dot-product loss for classification. This is a schematic representation of two time steps of the Transformer Embedding Dialogue (TED) policy [2]:

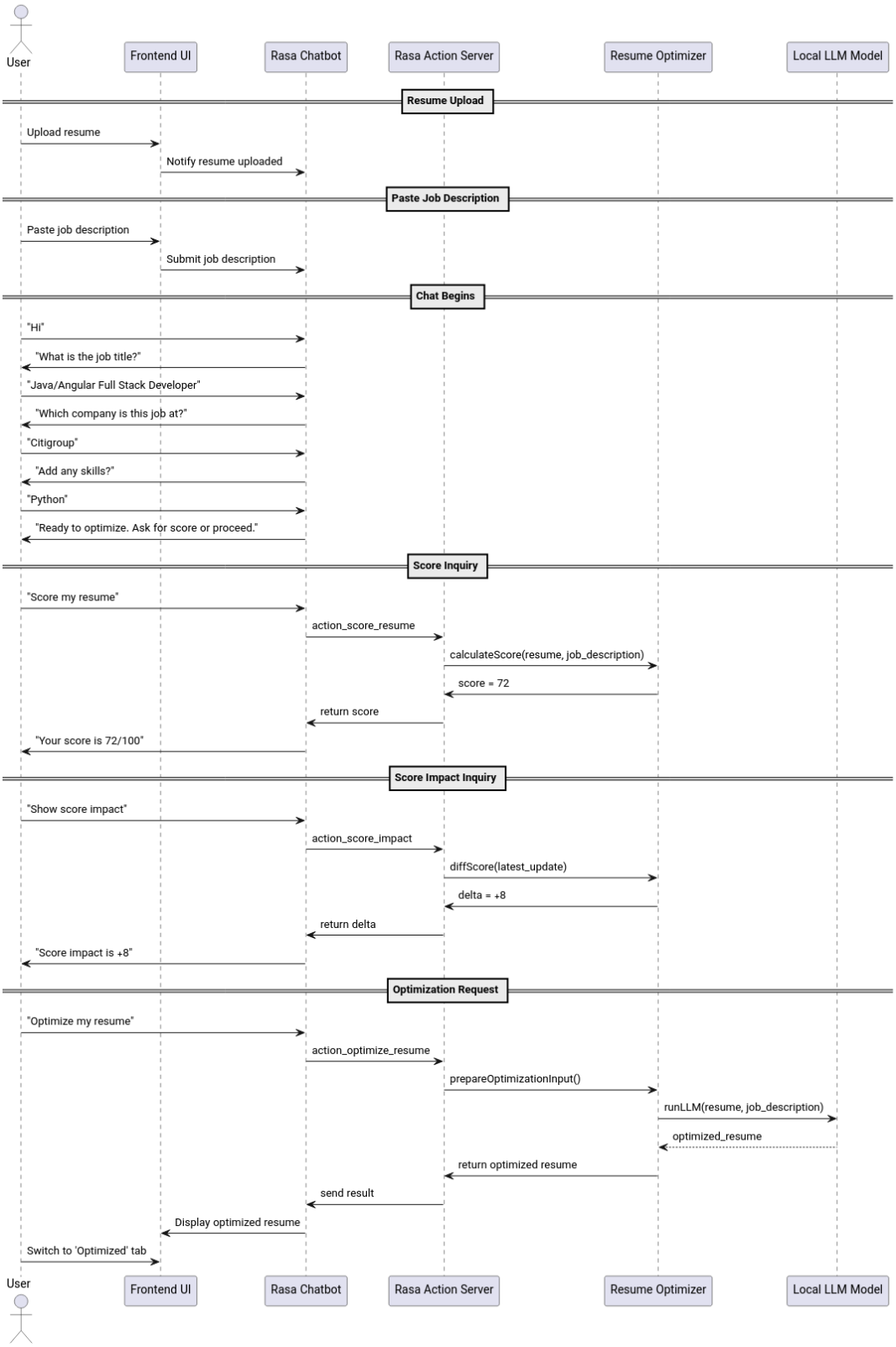


UnexpecTEDIntentPolicy: Helps reviewing conversations and also allows the bot to react to unlikely user turns. It is an auxiliary policy that should only be used in conjunction with at least one other policy, as the only action that it can trigger is the special action_unlikely_intent action.

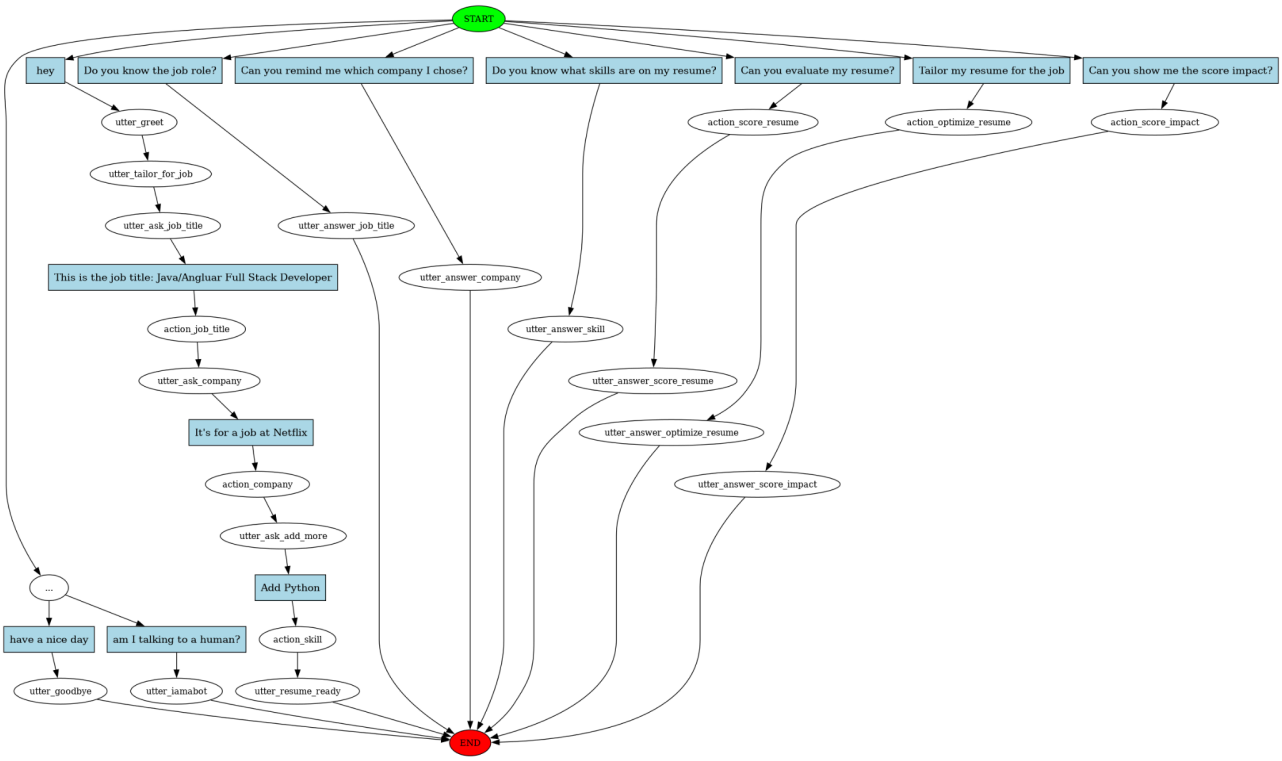
Architecture: Chatbot Sequence Diagram

The sequence diagram for end-to-end flow of our Rasa-based resume optimization assistant is as follows.

The user begins by uploading a resume and pasting a job description through the frontend UI. This data is passed to the Rasa chatbot, which initiates a conversation to collect the job title, company name, and optional skills using form-based slot filling. When the user requests a score or optimization, Rasa triggers custom actions on the action server. These actions interact with a local resume optimizer, which may call an LLM model to process and enhance the resume content. The optimized resume is then returned to the user through the chatbot and displayed in the UI. This structured sequence ensures that user inputs are guided, validated, and translated into meaningful backend operations.


User
Frontend UI
Rasa Chatbot
Rasa Action Server
Resume Optimizer
Local LLM Model
Resume Upload
Upload resume
Notify resume uploaded
Paste Job Description
Paste job description
Submit job description
Chat Begins
"Hi"
"What is the job title?"
"Java/Angular Full Stack Developer"
"Which company is this job at?"
"Citigroup"
"Add any skills?"
"Python"
"Ready to optimize. Ask for score or proceed."
Score Inquiry
"Score my resume"
action_score_resume
calculateScore(resume, job_description)
score = 72
return score
"Your score is 72/100"
Score Impact Inquiry
"Show score impact"
action_score_impact
diffScore(latest_update)
delta = +8
return delta
"Score impact is +8"
Optimization Request
"Optimize my resume"
action_optimize_resume
prepareOptimizationInput()
runLLM(resume, job_description)
optimized_resume
return optimized resume
send result
Display optimized resume
Switch to 'Optimized' tab
User
Frontend UI
Rasa Chatbot
Rasa Action Server
Resume Optimizer
Local LLM Model

Architecture: Chatbot Dialogue Flow (Stories)



This dialogue flow outlines how user interactions are handled step-by-step by the Rasa assistant. The conversation begins with a user greeting, followed by prompts from the bot to collect key information such as job title, company name, and additional skills. This flow is managed using Rasa Forms, which guide the user through structured data collection while dynamically handling affirmations, denials, and re-prompts. Once all required slots are filled, the bot transitions to actions like scoring the resume or optimizing it. Each branch in the dialogue graph reflects possible paths the conversation can take, ensuring the assistant remains both responsive and context-aware throughout the user interaction.

These are stories that are already defined in our chatbot system:

- Story 1: Resume Improvement
  - User greets the bot (greet)
  - Bot replies with:
    - Greeting (utter_greet)
    - Prompt to tailor resume (utter_tailor_for_job)
    - Asks for job title (utter_ask_job_title)
  - User provides job_title
  - Bot runs action_job_title

    - Bot asks for company (utter_ask_company)
    - User provides company
    - Bot runs action_company
    - Bot asks if user wants to add more info (utter_ask_add_more)
    - User affirms and provides skill
    - Bot runs action_skill
    - Bot confirms it's ready to proceed (utter_resume_ready)
- Story 2: User Asks Job Title
    - User asks for the current job title (ask_job_title)
    - Bot responds with stored job title (utter_answer_job_title)
- Story 3: User Asks Company
    - User asks for the selected company (ask_company)
    - Bot responds with stored company (utter_answer_company)
- Story 4: User Asks Skills
    - User asks which skills have been added (ask_skill)
    - Bot responds with the current list of skills (utter_answer_skill)
- Story 5: User Asks for Resume Score
    - User requests a resume score (ask_score_resume)
    - Bot calls action_score_resume
    - Bot responds with the score (utter_answer_score_resume)
- Story 6: User Asks for Resume Optimization
    - User requests to optimize the resume (optimize_resume)
    - Bot triggers action_optimize_resume
    - Bot replies with the optimized resume (utter_answer_optimize_resume)
- Story 7: User Asks for Score Impact
    - User asks about the score impact of changes (ask_score_impact)
    - Bot runs action_score_impact
    - Bot responds with the score change details (utter_answer_score_impact)

Data Extraction

The `prepare_job_data.py` delivers a focused, reliable JSON-only pipeline:

- **Robust JSON Loading**
  - Reads a map of `{job_id: job_dict}` using Python's built-in `json` library.
  - Filters out any non-`dict` entries, so only valid postings are kept.

- **Safe Output Handling**
  - Creates the target directory if missing.
  - Dumps the cleaned postings back to disk with `indent=2` for readability, and prints counts to the console.

Keyword Mining

In `extract_info.py`, we use YAKE and NLTK to generate actionable keyword lists with minimal code:

- **Unsupervised n-Gram Extraction**
  - Uses YAKE's `KeywordExtractor` for 1-, 2-, and 3-grams (`--uni`, `--bi`, `--tri`), for finding the top phrases over the concatenated descriptions.

- **Deduplication + POS-Based Bucketing**
  - Case-insensitive dedupe preserves input order.
  - NLTK's `pos_tag` splits phrases into:
    - **`action_verbs`** (any phrase with a VB.* tag)
    - **`skills`** (any phrase with NN.* or JJ.* tag)
  - Results are emitted as JSON per bucket, ready for downstream use.

Embedding & Retrieval (R in RAG)

The `resume_optimizer.py` implements dense-vector retrieval with SentenceTransformers and cosine similarity:

- **Model Initialization**
  - Loads `all-MiniLM-L6-v2` once at startup (first-use download).

- **Job & Resume Embeddings**
  - Concatenates each job's title, company, and first 1,000 chars of description into one string.
  - Encodes all jobs in batch via `model.encode(...)`, storing the job embeddings as `self.job_embeddings`.
  - Encodes the full resume text on each query.

- **Similarity Search**
  - Computes `cosine_similarity` between the resume embedding and every job embedding.

- Optionally filters by `job_type` substring on the title, then sorts by score and returns the top-K entries as the candidates for the hints.

Keyword Analysis & TF-IDF Scoring

Within the same `ResumeOptimizer` class, we extract and categorize keywords from the top job matches:

- **TF-IDF Vectorizer**
  - Instantiates `TfidfVectorizer(max_features=100, stop_words='english', ngram_range=(1,2))`.
  - Fits on the concatenated job descriptions and computes average TF-IDF per feature.
  - Sorts to select the top 30 keywords by importance.

- **Coverage Diff**
  - Splits keywords into `present_keywords` vs. `missing_keywords`, using a helper `_check_keyword_variant()` to catch plurals, hyphenated forms, and simple variants.

Generation Engine (G in RAG)

The rewrite step combines an on-device LLM with a cloud fallback and rule-based safeguards:

- **Primary Inference via Ollama** (`resume_optimizer_v2.py`)
  Sends a JSON POST to `http://localhost:11434/api/generate` with:
    - **Model**: `deepseek-r1:7b` (4-bit QLoRA quantized)
      Hyperparams: `temperature=0.2`, `max_tokens=2048`, `stream=False`
  - Uses strong `SYSTEM_PROMPT` and structured `USER_PROMPT_TEMPLATE` to enforce no hallucinations, preserved formatting, and logical keyword integration.

- **Cloud Fallback & Backup Models** (`resume_optimizer.py`)
  If Ollama fails or if we choose HuggingFace, the code can call Mistral-7B (`mistralai/Mistral-7B-Instruct-v0.2`) and, upon error, fall back to Google's `flan-t5-xl`.
  - All calls include `response.raise_for_status()` and a basic retry for the backup path.

Prompt Engineering & Output Post-processing

Building on our structured prompting approach, we funnel every optimized sentence through a unified quality-control gateway:

- **Difference-aware Prompt Templates**
  - **System Prompt**: Instructs the model to preserve original action verbs, integrate at least one missing keyword and never fabricate metrics.

- **User Prompt**: Combines the original bullet, comma-joined `present_keywords`/`missing_keywords` lists, and up to three retrieved job snippets.
- By sketching out exactly what belongs in "present" versus "missing," and by limiting contextual snippets, we steer the LLM away from hallucinations and keyword-stuffing.

- **Centralized Guardrails Hook**
  - Implemented as `apply_guardrails(text: str) → str` in `resume_optimizer_v2.py`, this is invoked immediately after generation.
  - Although it currently returns the raw text, its location is our single maintenance point for future filters(which are now just a part of the prompt engineering) such as:
    - **Repetition Limits**: reject or truncate bullets with > 15 % duplicate tokens.
    - **Keyword-stuffing Guards**: flag bullets that lean too heavily on one term.
    - **Metric Fabrication Checks**: reject any numeric claims not grounded in input snippets.

- **Rule-based Backup Fallback**
  - If the LLM path errors out (Ollama + any cloud fallback), the `_backup_suggestion_generation()` routine takes over.
  - It upgrades our original bullet by mapping verbs to stronger synonyms and appending exactly one missing keyword—ensuring every line meets an ATS threshold without over-engineering.

References

[1] Bunk, T., Varshneya, D., Vlasov, V., & Nichol, A. (2020). Diet: Lightweight language understanding for dialogue systems. *arXiv preprint arXiv:2004.09936*.

[2] Vlasov, V., Mosig, J. E., & Nichol, A. (2019). Dialogue transformers. *arXiv preprint arXiv:1910.00486*.