

# Foundations of Conversational Artificial Intelligence

Phillip Schneider

25.04.2023, Practical Course NLP-based Software Engineering

Chair of Software Engineering for Business Information Systems (sebis)  
Department of Computer Science  
School of Computation, Information and Technology (CIT)  
Technical University of Munich (TUM)  
[www.matthes.in.tum.de](http://www.matthes.in.tum.de)

## Theoretical foundations of conversational agents

- Background knowledge for building conversational agents
- Core concepts of conversational agents
- Introduction to Rasa

## Building a conversational agent

- Creating a new agent
- Creating NLU training data
- Dialogue management



# Theoretical foundations for conversational agents

## Background knowledge for building conversational agents

- Types of conversational agents
- Modular pipeline architecture

## Core concepts of conversational agents

- Intents
- Entities
- Actions
- Domain
- Stories

## Introduction to Rasa

- Installing Rasa Open Source
- Rasa architecture

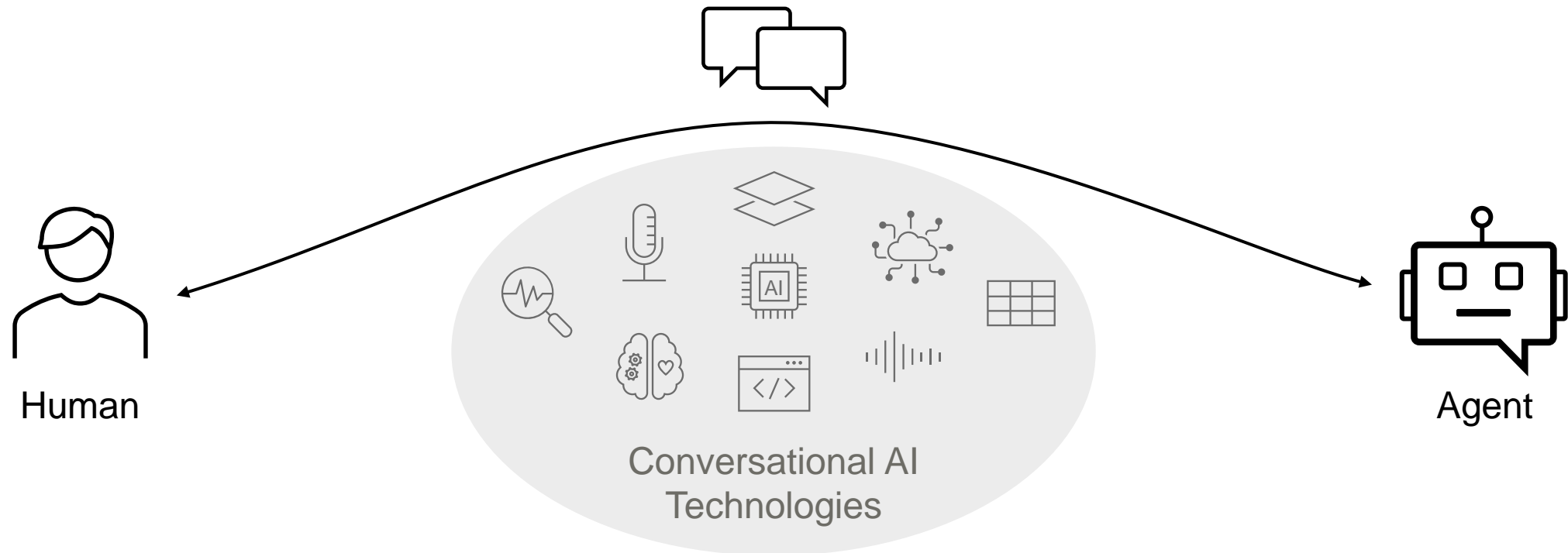
## Terminology

### Definition

Conversational Artificial Intelligence (Conversational AI) is a collective term referring to technologies for building conversational agents that are able to interact with humans through natural language.

### Warning

The term is defined and interpreted inconsistently in both academia and industry.



# Background knowledge for building conversational agents

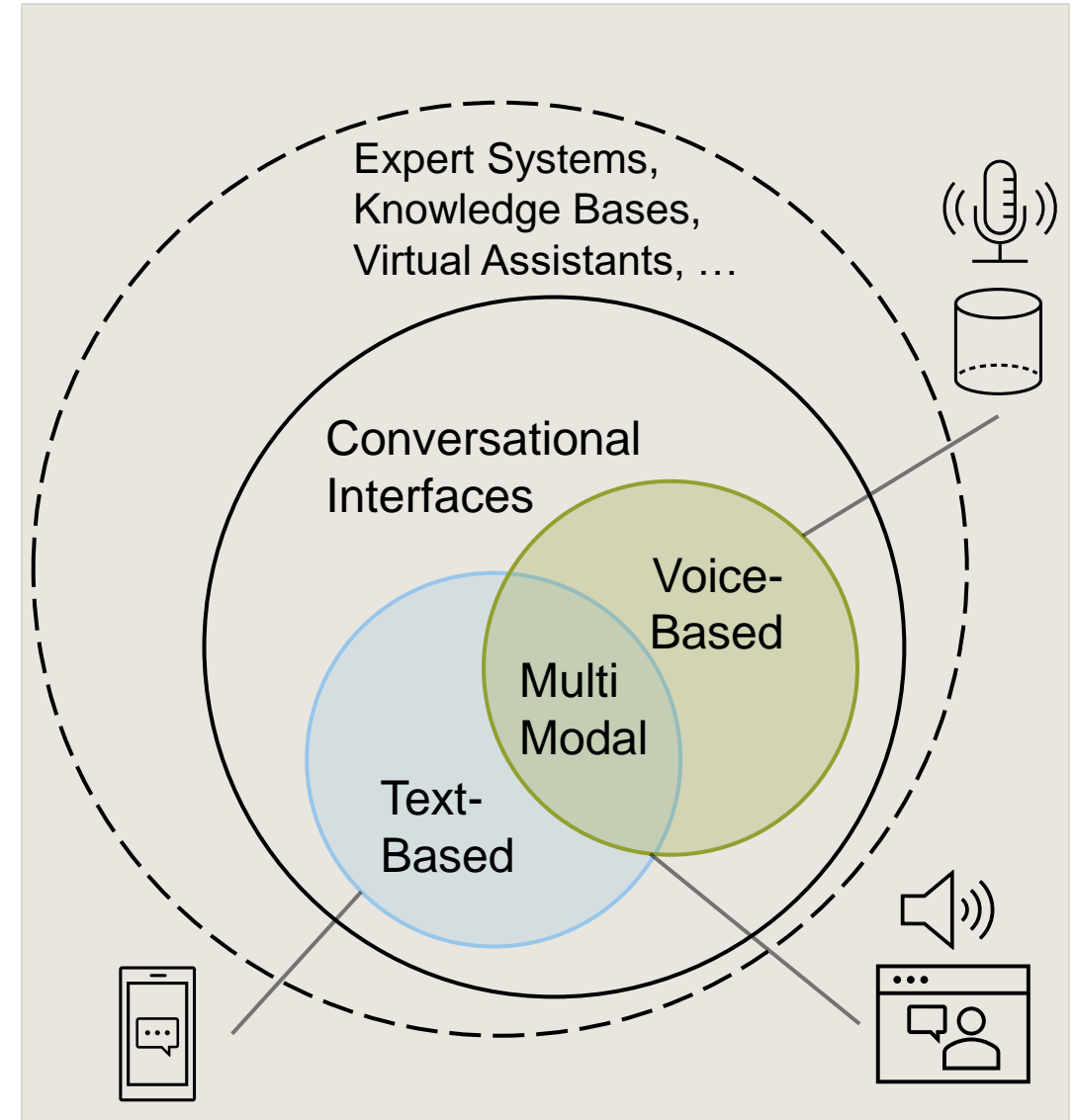
## Types of conversational agents

### Types of Conversational Interfaces

- Conversational interfaces can be broadly categorized into three types:
  - Text-based interfaces
  - Voice-based interfaces
  - Multi modal interfaces

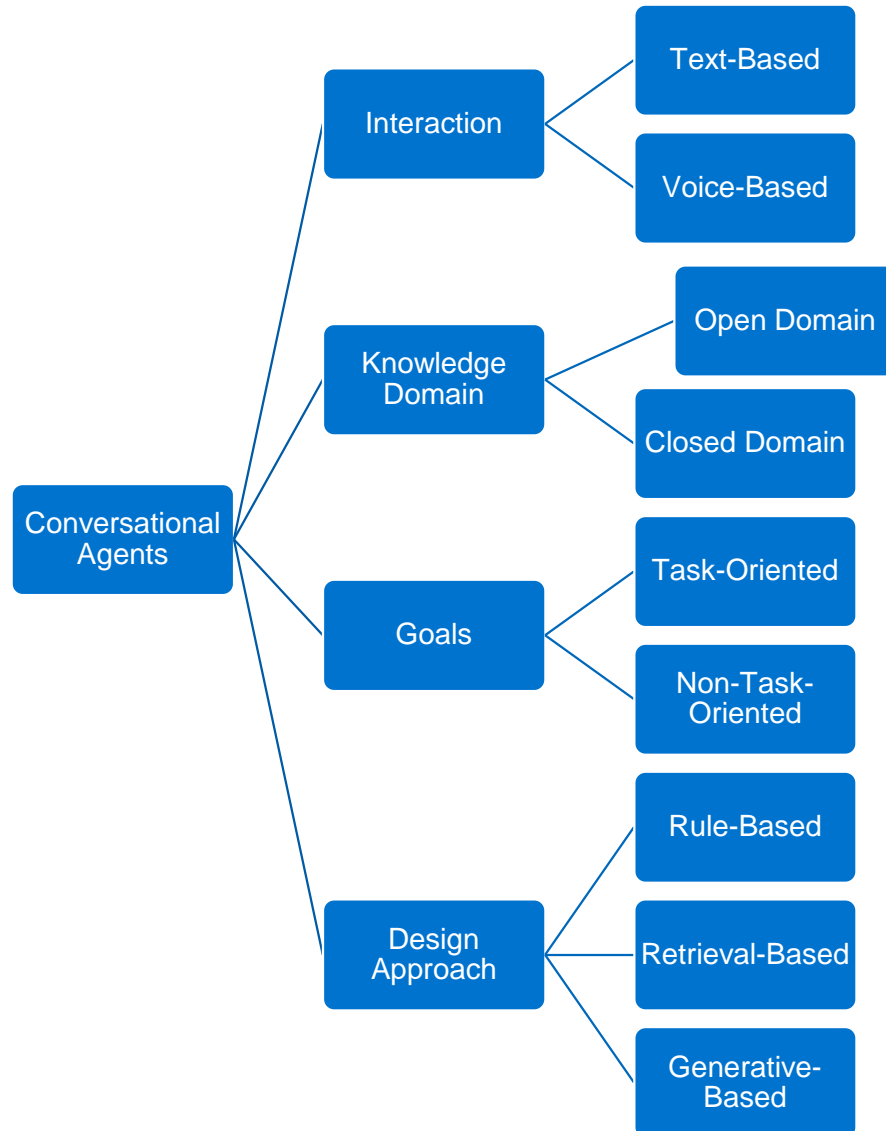
### Integration of Conversational Interfaces

- Many software agent systems incorporate conversational interfaces (conversational agents)
- Conversational interfaces are often embedded in smart devices



# Background knowledge for building conversational agents

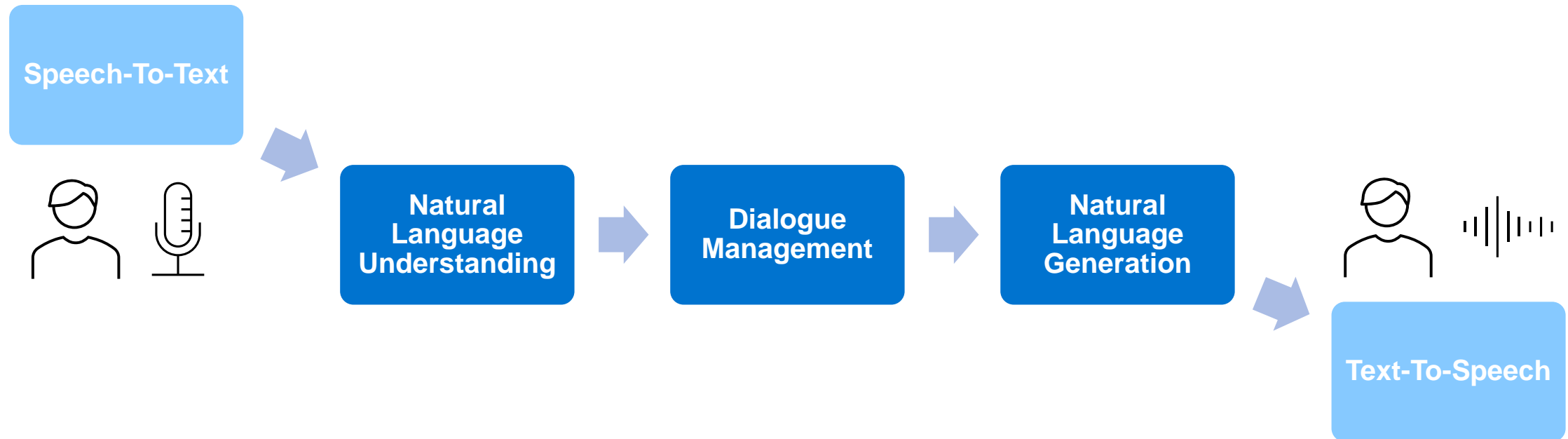
Types of conversational agents by Hussain et al. (2019)



- Broader categorization of conversational agents based on several criteria, e.g., mode of interaction, knowledge domain, their usage and the design techniques
- Specifically with regard to goals, chatbots are classified into two main categories:
  - Task-oriented
    - Designed for a particular task
    - Set up to have short conversations, usually within a closed domain
  - Non-task oriented
    - Can simulate a conversation with a person
    - Seem to perform chitchat for entertainment purpose in open domains

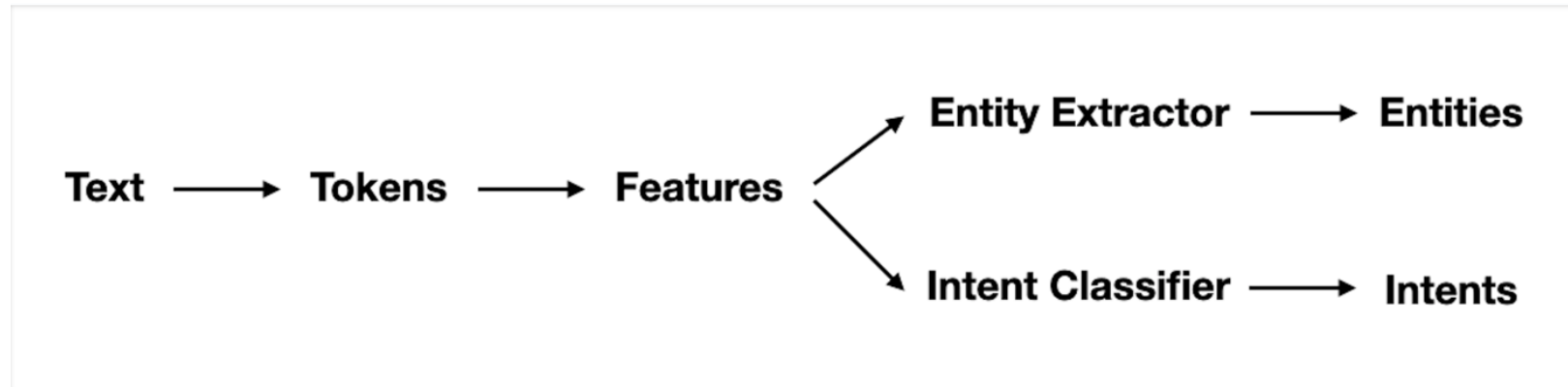
# Background knowledge for building conversational agents

## Modular pipeline architecture



### Natural language understanding (NLU)

- Goal: to extract structured information from user messages, usually includes the user's intent and any entities their message contains
- Rasa NLU has a pipeline architecture



<https://rasa.com/blog/intents-entities-understanding-the-rasa-nlu-pipeline/>



### Natural language understanding (NLU)

NLU will take in a sentence such as

*"I am looking for a French restaurant in the center of town"*

and return structured data like:

```
{
  "intent": "search_restaurant",
  "entities": {
    "cuisine": "French",
    "location": "center"
  }
}
```

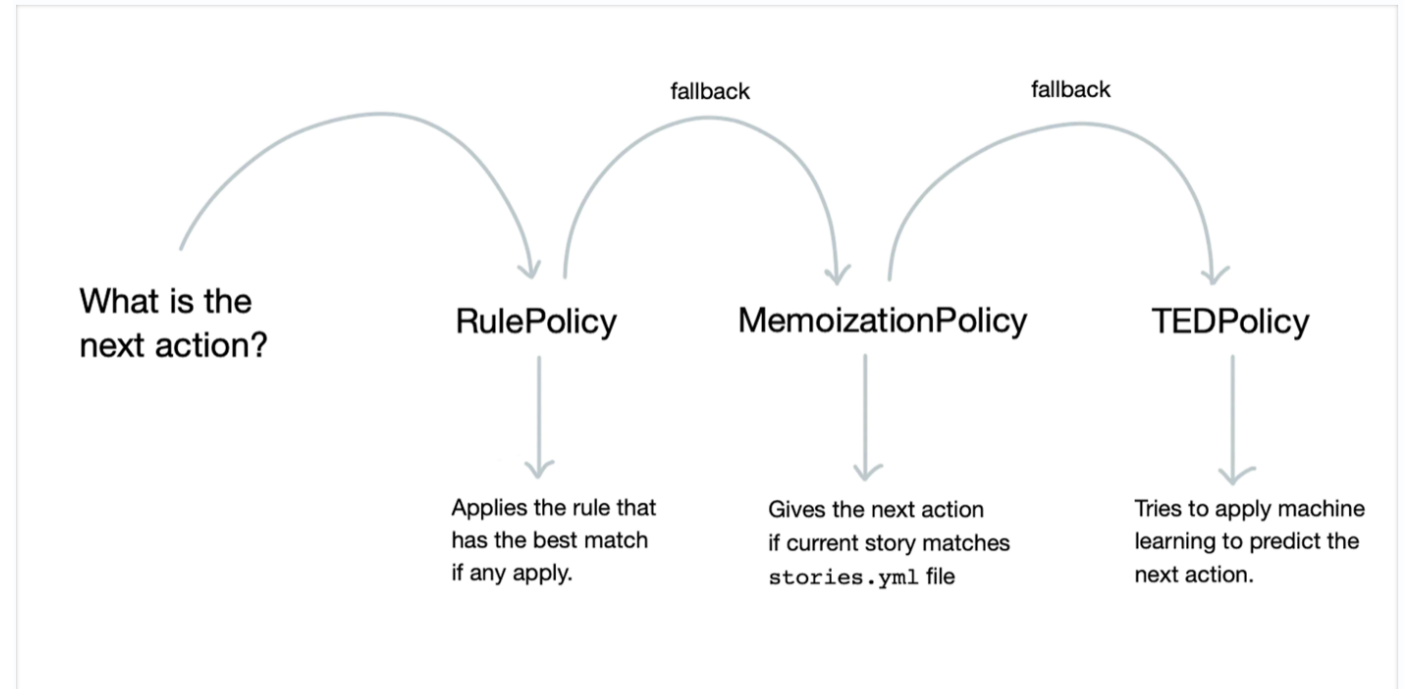
<https://rasa.com/docs/rasa/generating-nlu-data>

### Dialogue management (DM)

Dialogue management is the function that controls the next action the assistant takes during a conversation.

Based on the intents and entities extracted by Rasa NLU, as well as other context, like the conversation history, Rasa decides which text response should be sent back to the user or whether to execute custom code, like querying a database.

Rasa assistant uses policies to decide which action to take at each step in a conversation. There are three different policies that the default `config.yml` file

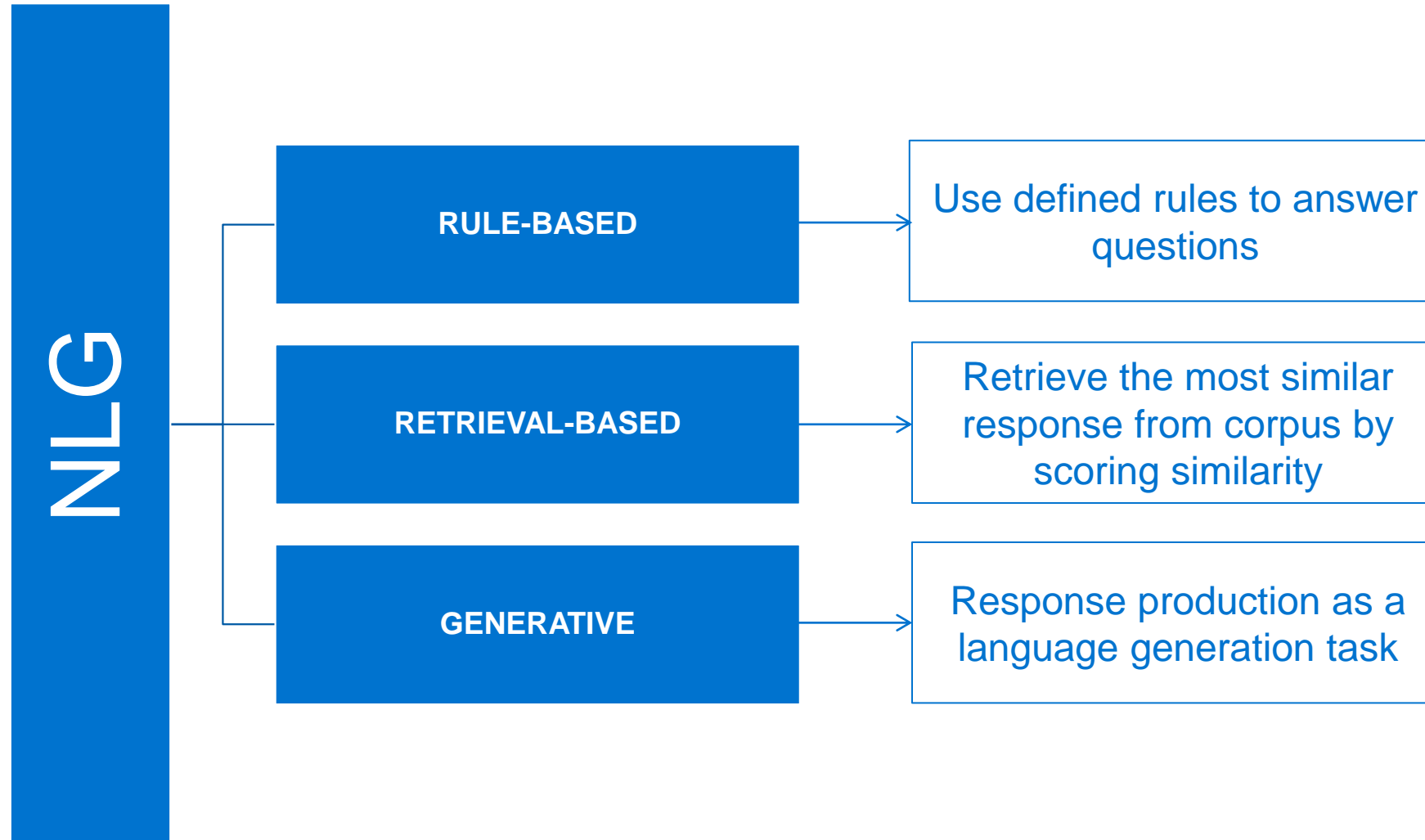


<https://rasa.com/blog/dialogue-policies-rasa-2/>

# Modular pipeline architecture

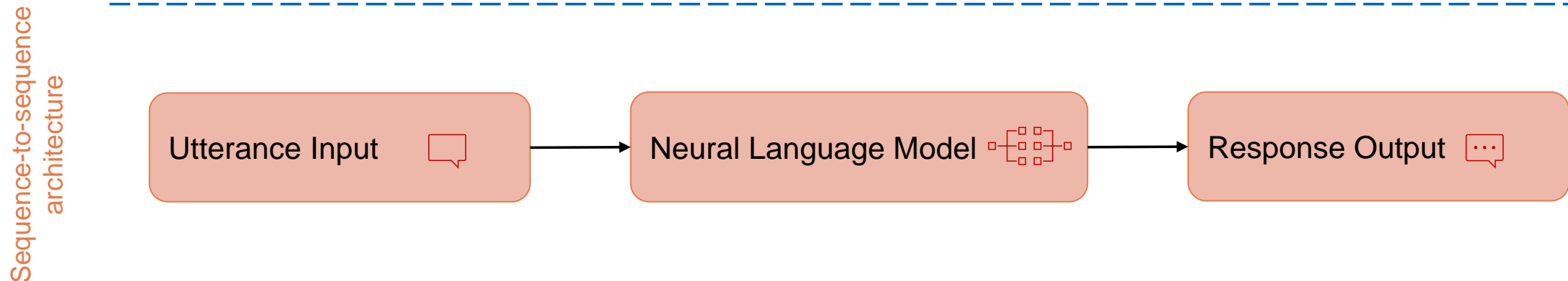
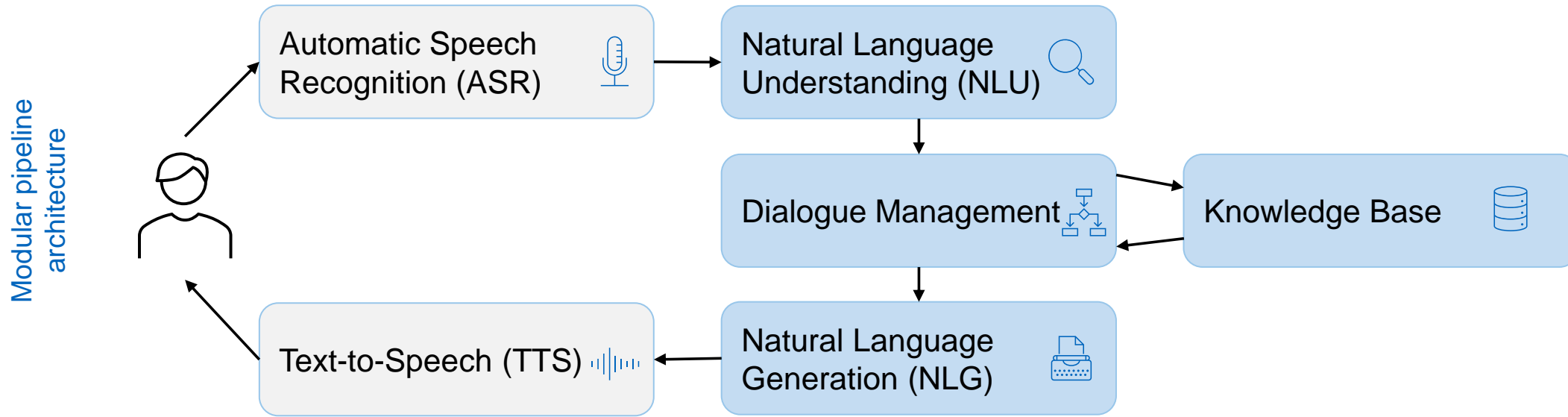
Modular pipeline architecture

## Natural Language Generation (NLG)



# Modular pipelines versus sequence-to-sequence architectures

Comparison of pipeline-based and sequence-to-sequence architectures



# Excursus: Sequence-to-sequence architectures (Transformers)

Neural Language Model 

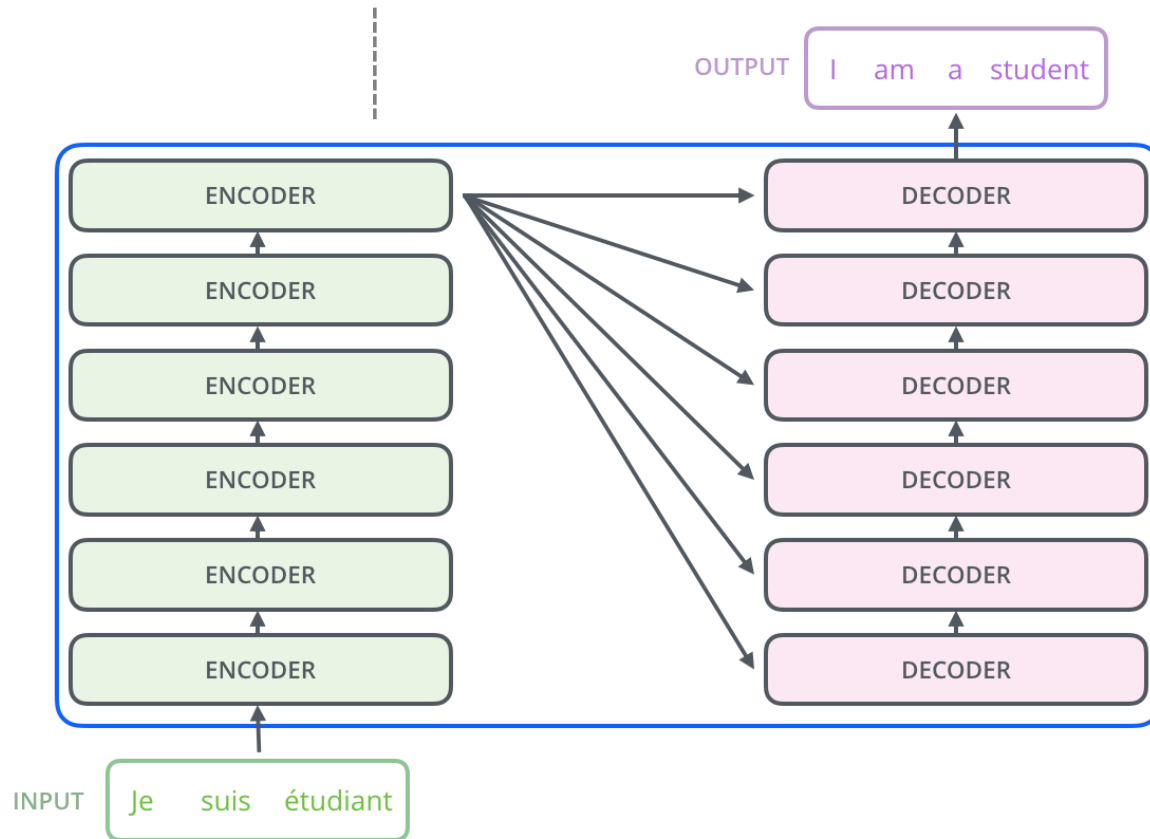


Figure taken from: [The Illustrated Transformer](#) (Alammar, 2018)

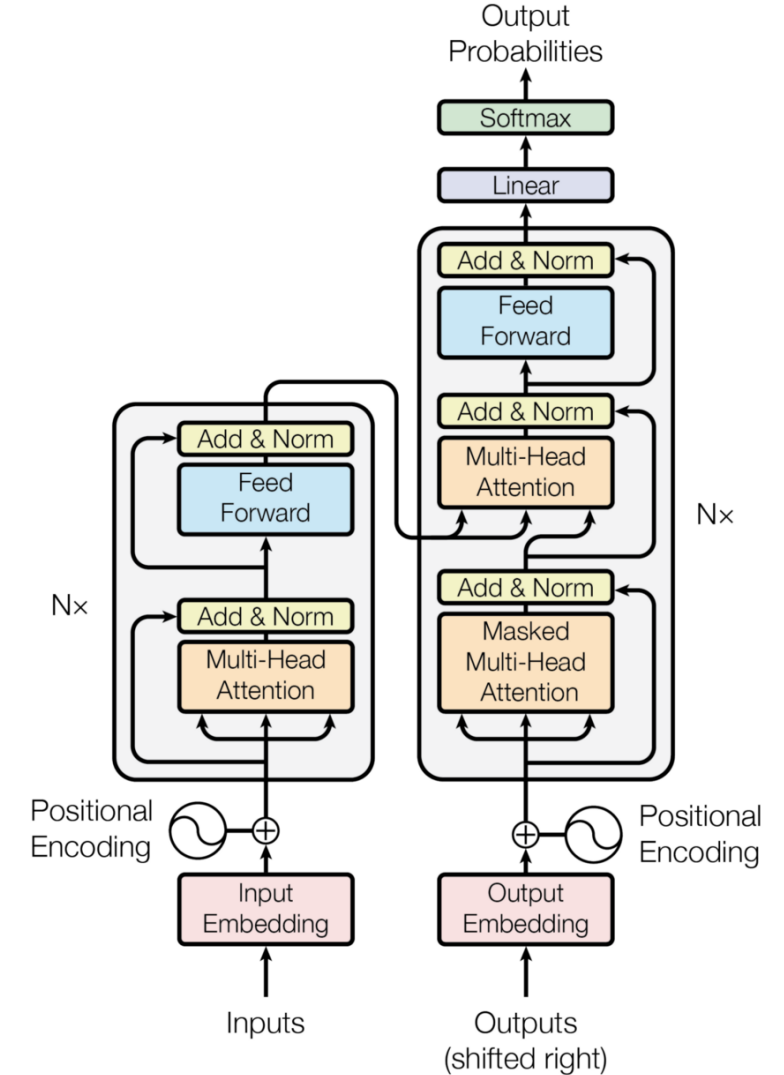
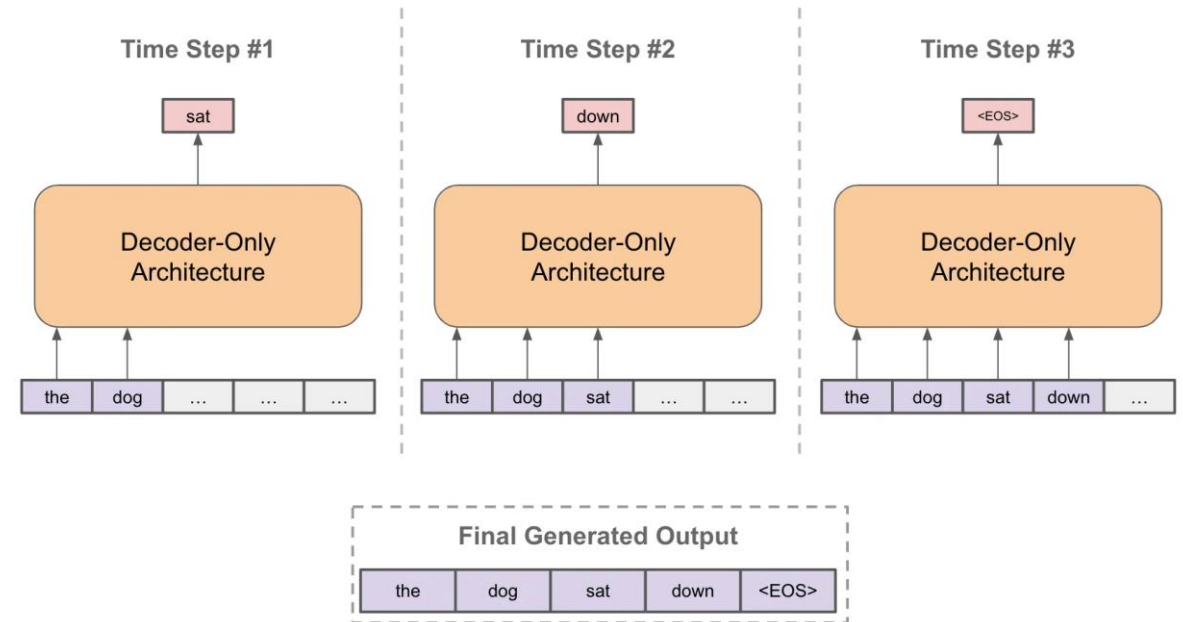
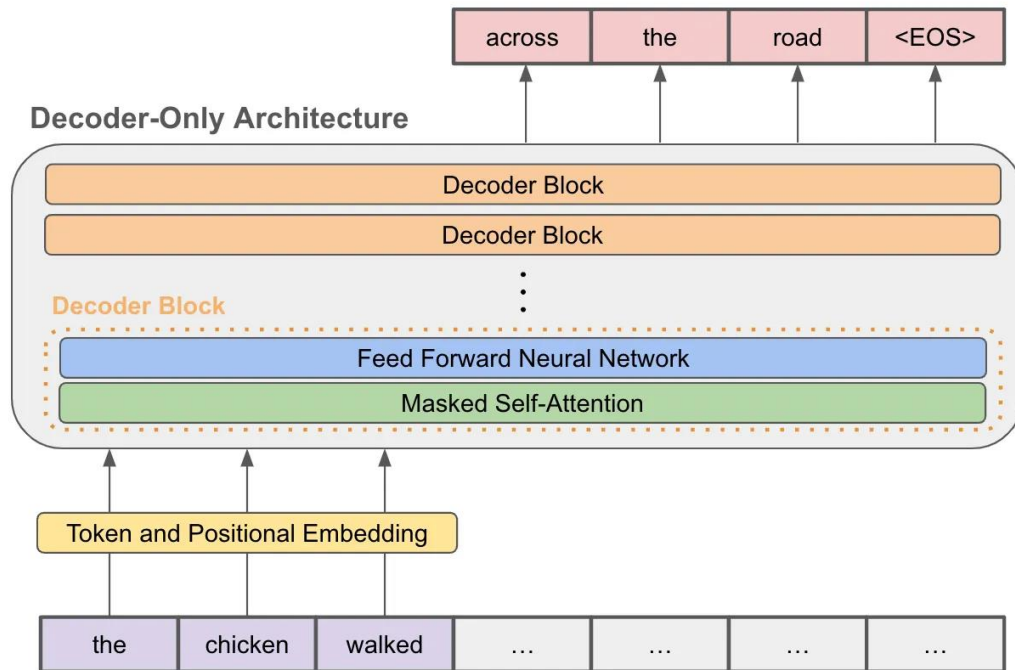


Figure taken from: [Attention Is All You Need](#) (Vaswani et al., 2017)

# Excursus: Sequence-to-sequence architectures (decoder-only)



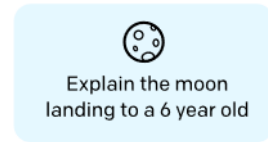
Figures taken from: [Understanding the Open Pre-Trained Transformers \(OPT\) Library](#) (Wolfe, 2022)

# Excursus: Alignment of large language models with human intentions: Reinforcement Learning from Human Feedback

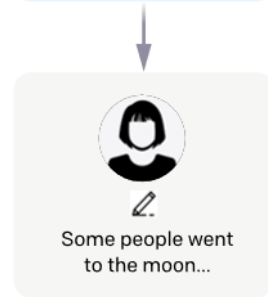
Step 1

**Collect demonstration data,  
and train a supervised policy.**

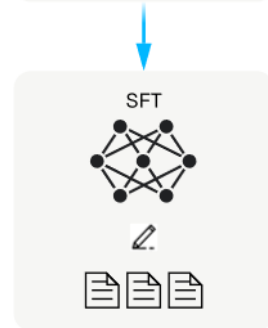
A prompt is  
sampled from our  
prompt dataset.



A labeler  
demonstrates the  
desired output  
behavior.



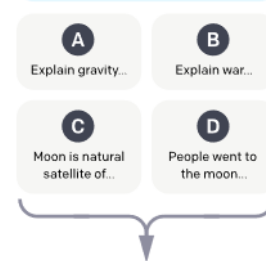
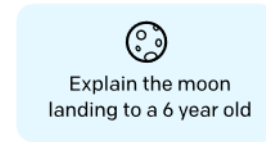
This data is used  
to fine-tune GPT-3  
with supervised  
learning.



Step 2

**Collect comparison data,  
and train a reward model.**

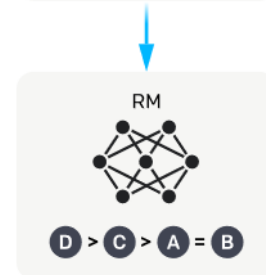
A prompt and  
several model  
outputs are  
sampled.



A labeler ranks  
the outputs from  
best to worst.



This data is used  
to train our  
reward model.



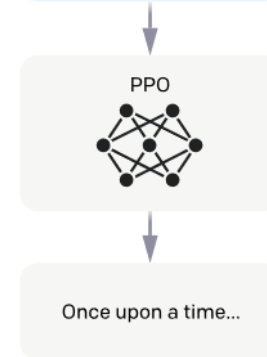
Step 3

**Optimize a policy against  
the reward model using  
reinforcement learning.**

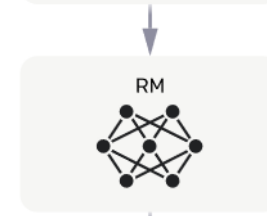
A new prompt  
is sampled from  
the dataset.



The policy  
generates  
an output.



The reward model  
calculates a  
reward for  
the output.



The reward is  
used to update  
the policy  
using PPO.



## Warning

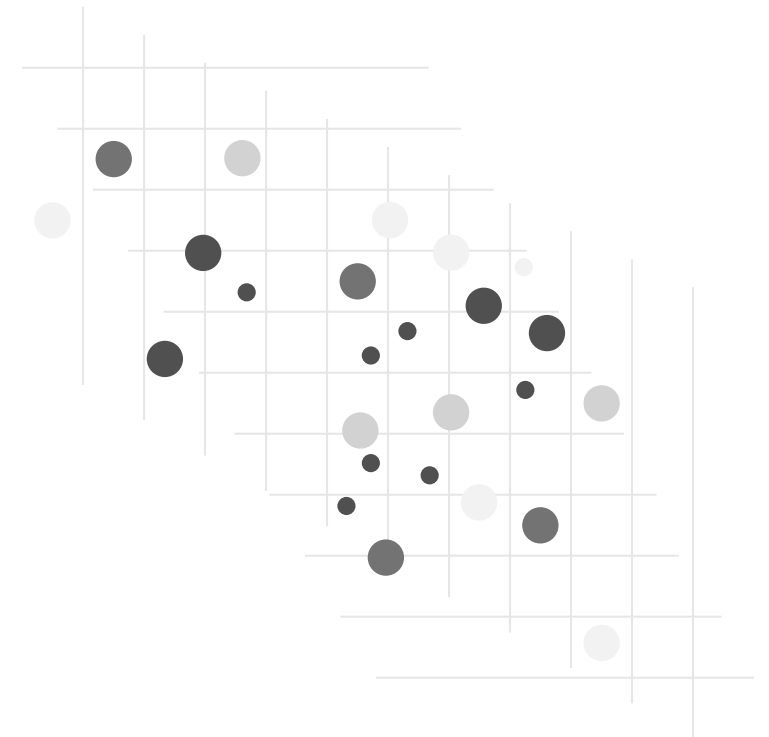
Auto-regressive large language models (LLMs) show exceptional performance in text generation, but should be used with caution in the context of dialogue systems due to the following shortcomings:

- Weak representational power
- Factual errors and hallucinations
- Logical errors and lack of commonsense
- Inconsistency and not controllable
- Bias and toxicity

→ Large language models have no knowledge of the underlying reality and do not apply human-level reasoning

## Recently released LLMs

- GPT-3.5, GPT-4, L, Bard, Claude, LLaMA, Alpaca, Vicuna, Dolly





# Theoretical foundations for conversational agents

## Background knowledge for building conversational agents

- Types of conversational agents
- Modular pipeline architecture

## Core concepts of conversational agents

- Intents
- Entities
- Actions
- Domain
- Stories

## Introduction to Rasa

- Installing Rasa Open Source
- Rasa architecture

# Core concepts of conversational agents

## 💡 Intents

The aim or target in a given user message is intent.

For example, if the user says

*“I want to order a book” or “Placing order for books”,*

the user is essentially placing an order. So we can group these into a single intent called **order**.

Now, whenever the bot gets a user message that is similar to other phrases in order, the bot will classify it as belonging into the **order** intent

A broader intent can be useful to trigger your assistant to do other things:

*“I like puppies.”*

*“We’ll leave at 4:00”*

*“Oh, I’m allergic to shellfish.”*

*“No, she hasn’t made reservation yet.”*

intent: inform

Using intents means you treat modelling what a user does in a conversation as a multiclass classification problem.

## 📌 Entities

Things that can be extracted from a user message. For example: a telephone number, a person's name, a location, the name of a product.

Example of entities in a user input:

- I would like to book a flight to **Sydney**.

```
entity: destination  
value: Sydney
```

Entity roles and groups allow you to add more details to your entities.

Entity roles allow you to define the roles of the entities of the same groups.

- I am looking for a flight from **New York** to **Boston**

```
entity: destination  
value: Sydney  
role: origin
```

```
entity: destination  
value: Boston  
role: destination
```

Entities groups allow you to put extracted entities under a specific group.

- I would like a large pepperoni with **cheese** and one with **mushrooms**.

```
entity: toppings  
value: mushrooms  
group: 1
```

```
entity: toppings  
value: cheese  
group: 2
```

## ▶ **Actions**

The model predicts an action that the assistant should perform next after each user message.

An overview of different types of actions:

Responses

Custom actions

Forms

Most commonly used

Default actions

Slot validation actions

## Domain

The domain file is a directory of everything your assistant knows:

- Responses: Things assistant can say to users
- Intents: Categories of things users say
- Slots: Variables remembered over the course of a conversation
- Entities: Pieces of information extracted from incoming text
- Forms and actions: Add application logic and extend what your agent can do

# Core concepts of conversational agents

## Stories

A type of structured data used to train an assistant's dialogue management model. Stories can be used to train models that are able to generalize to unseen conversation paths.

Format of stories:

A story is a representation of a conversation between a user and an agent, converted into a specific format:

- user inputs are expressed as intents (and entities when necessary)
- the assistant's responses and actions are expressed as action names

### Example of a dialogue in the Rasa story format

stories:

- story: collect restaurant booking info *# name of the story - just for debugging*

steps:

- intent: greet *# user message with no entities*

- action: utter\_ask\_howcanhelp

- intent: inform *# user message with entities*

entities:

- location: "rome"

- price: "cheap"

- action: utter\_on\_it *# action that the bot should execute*

- action: utter\_ask\_cuisine

- intent: inform

entities:

- cuisine: "spanish"

- action: utter\_ask\_num\_people

# Core concepts of conversational agents

## Stories

Stories are composed of:

- **story:** The story's name. The name is arbitrary and not used in training; you can use it as a human-readable reference for the story.
- **metadata:** arbitrary and optional, not used in training, you can use it to store relevant information about the story like, e.g., the author
- **a list of steps:** The user messages and actions that make up the story

Each step can be one of the following:

- A user message, represented by intent and entities.
- An or statement, which includes two or more user messages under it.
- A bot action.
- A form.
- A slot was set event.
- A checkpoint, which connects the story to another story.

### Example

stories:

- story: Greet the user

metadata:

author: Somebody

key: value

steps:

*# list of steps*

- intent: greet

- action: utter\_greet

# Theoretical knowledge for conversational agents

## Background knowledge for building conversational agents

- Types of conversational agents
- Modular pipeline architecture

## Core concepts of conversational agents

- Intents
- Entities
- Actions
- Domain
- Stories

## Introduction to Rasa

- Installing Rasa Open Source
- Rasa architecture



# Introduction to Rasa

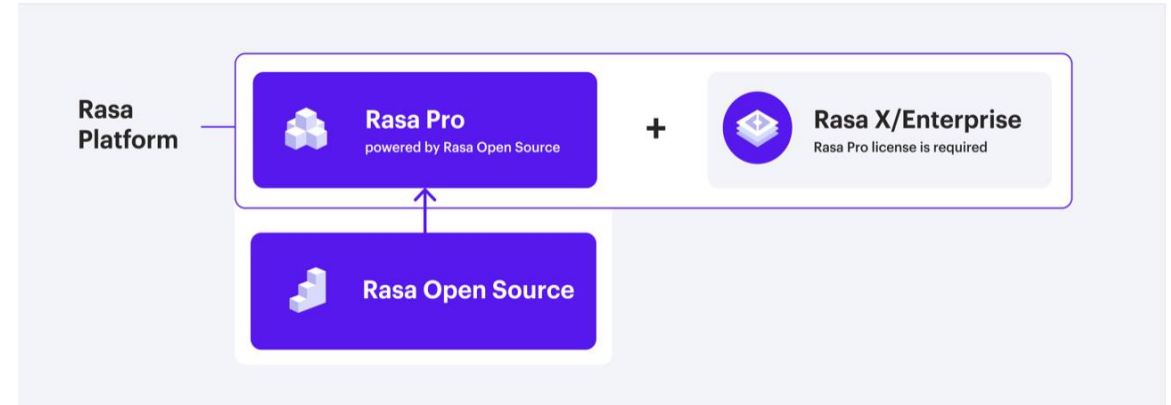
## Rasa Open Source & Rasa Pro

### Rasa Open Source

Rasa Open Source is an open source conversational AI platform that allows you to understand and hold conversations, and connect to messaging channels and third party systems through a set of APIs. It supplies the building blocks for creating virtual (digital) assistants or chatbots.

### Rasa Pro

Rasa Pro is a conversational AI framework powered by Rasa Open Source, and includes additional features, APIs, and services that serve enterprise specific needs around security, observability, and scale.



With over 25 million downloads, Rasa Open Source is the most popular open source framework for building chat and voice-based AI assistants.

### Exploring Rasa Open Source online using Rasa Playground before you install

At the end of the tutorial, you can download the resulting assistant, install Rasa on your machine and continue development locally.

<https://rasa.com/docs/rasa/playground/>

### Python environment requirement

Currently, rasa supports the following Python versions: 3.7, 3.8, 3.9 and 3.10

### Installing Rasa Open Source

To install Rasa Open Source:

Ubuntu / macOS / Windows

```
pip3 install rasa
```

You can now create a new project with:

```
rasa init
```

#### Additional dependencies

For some machine learning algorithms, you need to install additional python packages.

The page on

<https://rasa.com/docs/rasa/tuning-your-model/> will help you pick the right

configuration for your assistant and alert you to additional dependencies.

### Upgrading Versions

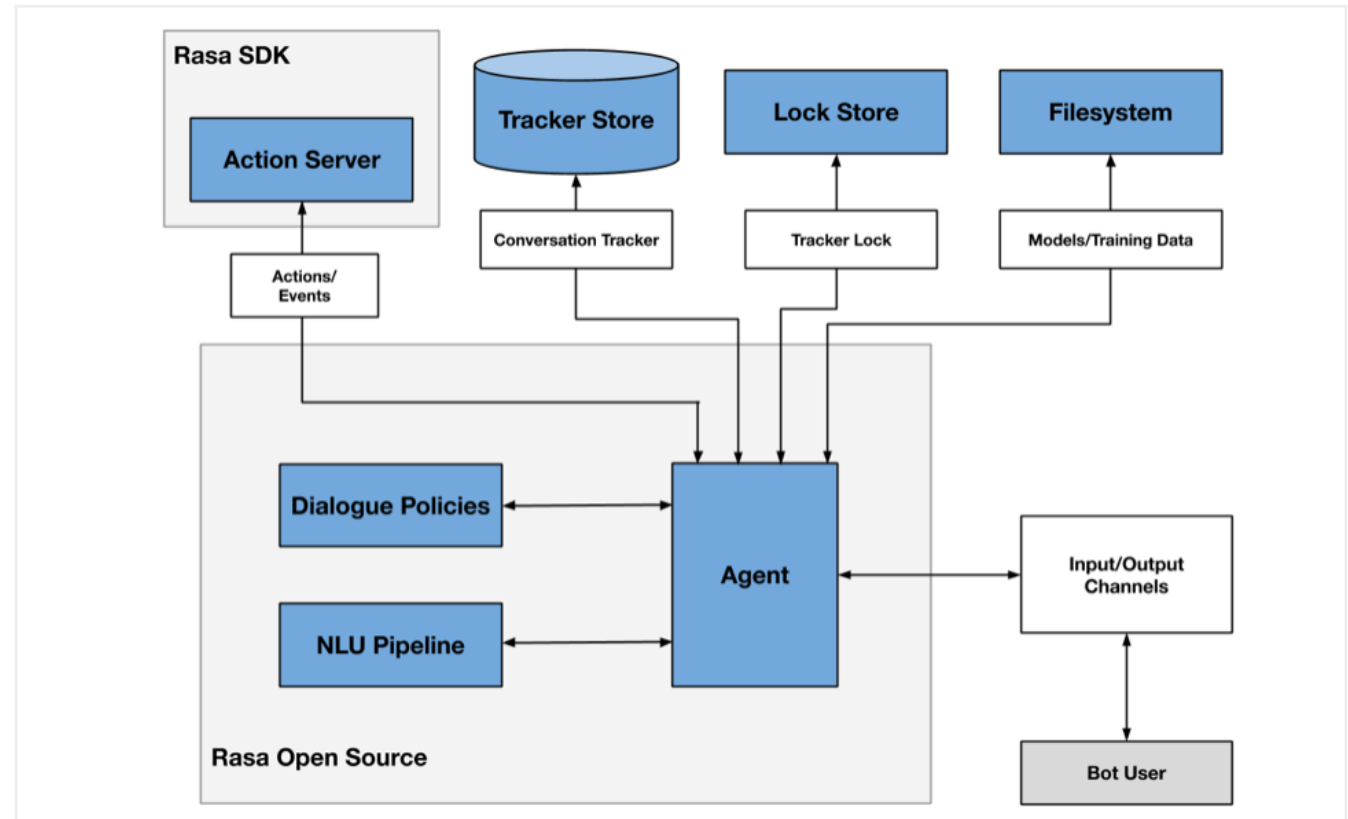
To upgrade your installed version of Rasa Open Source to the latest version from PyPI:

```
pip3 install --upgrade rasa
```

To download a specific version, specify the version number:

```
pip3 install rasa==3.0
```

- Two primary components are Natural Language Understanding (NLU) and dialogue management
- NLU is the part that handles intent classification, entity extraction, and response retrieval. It's shown below as the NLU Pipeline because it processes user utterances using an NLU model that is generated by the trained pipeline.
- The dialogue management component decides the next action in a conversation based on the context. This is displayed as the Dialogue Policies in the diagram.



<https://rasa.com/docs/rasa/arch-overview>

## Creating a new agent

- Files
- Commands

## Creating NLU training data

- Basics of conversational design
- Defining intents and entities
- NLU training pipeline

## Dialogue management

- Stories
- Rules
- Domain, custom actions, and slots
- Dialogue policies

- File structure of the project you have just created

```
/path/to/project
├── actions
│   ├── __init__.py
│   └── actions.py
├── data
│   ├── nlu.yml
│   ├── rules.yml
│   └── stories.yml
├── models
├── tests
│   └── test_stories.yml
├── config.yml
├── credentials.yml
├── domain.yml
└── endpoints.yml
```

- The `domain.yml` file is the file where everything comes together
- The `config.yml` file contains the configuration for your machine learning models
- The data folder contains data that your assistant will learn from
- The `nlu.yml` file contains examples for your intents and entities
- The `stories.yml` file contains examples of conversations turns
- The `rules.yml` file contains predefined rules for the dialogue policies

<https://learning.rasa.com/conversational-ai-with-rasa/creating-a-new-assistant/>

- `rasa init` allows you to start a new Rasa project
- `rasa train` allows you to train a new assistant based on your current training data
- `rasa shell` allows you to chat with a trained assistant
- `rasa -h` allows you get receive relevant help text for a command
- `rasa --debug` gives you extra log output when running commands

## Creating a new agent

- Files
- Commands

## Creating NLU training data

- Basics of conversational design
- Defining intents and entities
- NLU training pipeline

## Dialogue management

- Stories
- Rules
- Domain, custom actions and slots
- Dialogue policies

- Three important planning steps:
  - Asking who your users are
  - Understanding the assistant's purpose
  - Documenting the most typical conversations users will have with the assistant
    - Gathering possible questions
    - Outlining the conversation flow

### Note

Conversation design is a challenging task. It's difficult to anticipate the back and forth interactions in real-life conversations. You should only rely on hypothetical conversations in the early stages of development and train your assistant on real conversations as soon as possible.



### NLU model

- An NLU model is used to extract meaning from text input
- We will create training data which contains labelled examples of intents and entities
- Training an NLU model on this data allows the model to make predictions about the intents and entities in new user messages
- NLU models are created by a training pipeline
  - Rasa provides two pre-figured pipelines, defined in `config.yml` file
  - Configuring a custom training pipeline is also possible

### Word embeddings

- Word embeddings convert words to vectors, or dense numeric representations based on multiple dimensions.
- Similar words are represented by similar vectors, which allows the model to capture their meaning

# Creating NLU training data

## Defining intents and entities

Intents are defined using a double hashtag. Each intent is followed by multiple examples of how a user might express that intent.

### Best practices

- You don't need to write every possible utterance to train an intent, but you should provide at least 15-20 examples.
- Make sure you provide high-quality data to train your model. Examples should be relevant to the intents, and be sure that there's plenty of diversity in the vocabulary you use in your examples.

```
nlu.md
1 ## intent:greet
2 - hey
3 - hello
4 - hi
5 - good morning
6 - good evening
7 - hey there
8
9 ## intent:goodbye
10 - bye
11 - goodbye
12 - see you around
13 - see you later
14
15 ## intent:affirm
16 - yes
17 - indeed
18 - of course
19 - that sounds good
```

<https://rasa.com/blog/the-rasa-masterclass-handbook-episode-2/>

# Creating NLU training data

## Defining intents and entities

Entities are labelled with square brackets and tagged with their type in parentheses

Example: `nlu.md` file for a Medicare Locator

```
nlu.md
16 - [Sitka](location)
17 - [Juneau](location)
18 - [Virginia](location)
19 - [Cusseta](location)
20 - [Chicago](location)
21 - [Tuscon](location)
22 - [Columbus](location)
23 - [San Francisco](location)
24
25 ## intent:search_provider
26 - I need a [hospital](facility_type)
27 - find me a nearby [hospital](facility_type)
28 - show me [home health agencies](facility_type)
29 - [hospital](facility_type)
30 - find me a nearby [hospital](facility_type) in [San
    Francisco](location)
31 - I need a [home health agency](facility_type)
32
```

<https://rasa.com/blog/the-rasa-masterclass-handbook-episode-2/>

### Choosing a pipeline configuration

Rasa comes with two default, pre-configured pipelines for intent classification and entity extraction:

- **Pretrained\_embeddings\_spacy**
  - Advantages:
    - Boosts the accuracy of your models, even if you have very little training data
    - Faster training
  - Considerations:
    - Complete and accurate word embeddings are mostly in English
    - Word embeddings don't cover domain-specific words
- **Supervised\_embeddings** (training model from scratch)
  - Advantages
    - Can adapt to domain-specific words and messages, because the model is trained on your training data.
    - Language-agnostic. Allows you to build assistants in any language.
    - Supports messages with multiple intents.
  - Considerations
    - Need more training examples(1000 or more) for your model to start understanding unfamiliar user inputs

### Understanding individual pipeline components

Basic sequence of training for both pipelines:

1. Load pre-trained language model (optional). Only needed if you're using a pre-trained model like spaCy.
2. Tokenize the data.  
Splits the training data text into individual words, or subwords.
3. Named Entity Recognition.  
Teaches the model to recognize which words in a message are entities and what type of entity they are.
4. Featurization  
Converts tokens to vectors, or dense numeric representations of words.
5. Intent Classification.  
Trains the model to make a prediction about the most likely meaning behind a user's message

### Training pipeline components

#### supervised\_embeddings

```
language: "en"

pipeline:
- name: "WhitespaceTokenizer"
- name: "RegexFeaturizer"
- name: "CRFEntityExtractor"
- name: "EntitySynonymMapper"
- name: "CountVectorsFeaturizer"
- name: "CountVectorsFeaturizer"
  analyzer:"char_wb"
  min_ngram: 1
  max_ngram: 4
- name: "EmbeddingIntentClassifier"
```

#### pretrained\_embeddings\_spacy

```
language: "en"

pipeline:
- name: "SpacyNLP"
- name: "SpacyTokenizer"
- name: "SpacyFeaturizer"
- name: "RegexFeaturizer"
- name: "CRFEntityExtractor"
- name: "EntitySynonymMapper"
- name: "SklearnIntentClassifier"
```

#### 🔑 SpacyNLP

The `pretrained_embeddings_spacy` pipeline uses the SpacyNLP component to load the spaCy language model so it can be used by subsequent processing steps.

You only need to include this component in pipelines that use spaCy for pre-trained embeddings, and it needs to be placed at the very beginning of the pipeline

### Tokenizer: Splitting texts into smaller chunks

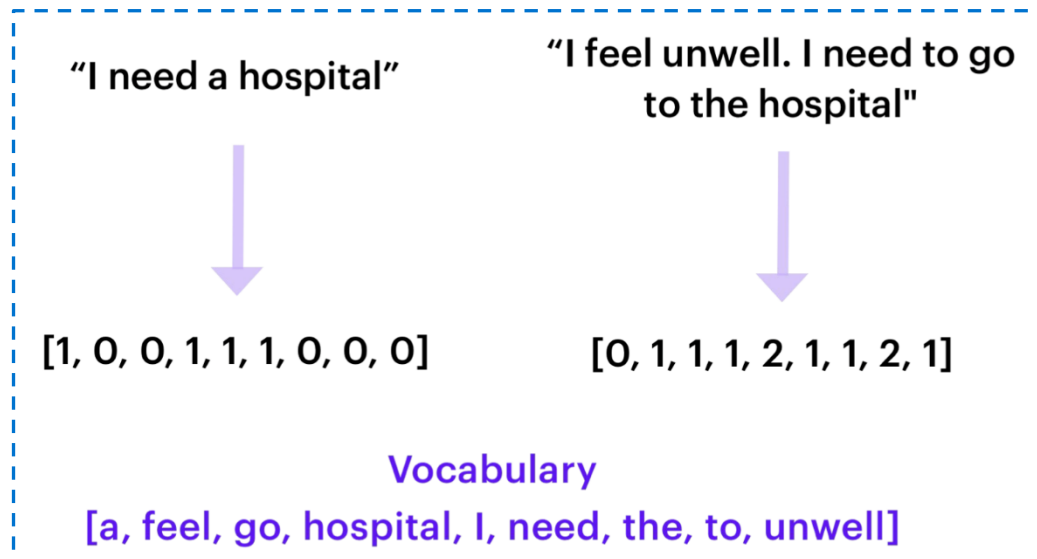
Tokenizer	
Supervised embeddings	Whitespace Jieba (Chinese)
Pre-trained embeddings	SpacyTokenizer

### Named entity recognition: Extracting entities from user messages

Named Entity Recognition	
Supervised embeddings	CRFEntityExtractor DucklingHttpExtractor Regex_featurizer
Pre-trained embeddings	SpacyEntityExtractor

**Intent classification:** Featurizers and intent classification models work together to classify intents

- **Featurizers:** Take tokens, or individual words, and encode them as vectors
  - **CountVectorsFeaturizer**
    - Creates a bag-of-words representation of a user's message using sklearn's CountVectorizer
    - Counts how often certain words from your training data appear in a message and provides that as input for the intent classifier



- **SpacyFeaturizer:** For pre-trained embeddings



- **Intent classification models**

- **EmbeddingIntentClassifier**

- Use EmbeddingIntentClassifier if you use CountVectorsFeaturizer
    - The features extracted by the CountVectorsFeaturizer are transferred to the EmbeddingIntentClassifier to produce intent predictions

- **SklearnIntentClassifier**

- When using pre-trained word embeddings, you should use the SklearnIntentClassifier component for intent classification
    - An SVM model predicts the intent of user input based on observed text features

<b>Intent Classification</b>		
<b>Pipeline</b>	<b>Featurizer</b>	<b>Intent Classifier</b>
Supervised embeddings	CountVectorsFeaturizer	EmbeddingIntentClassifier
Pre-trained embeddings	SpacyFeaturizer	SklearnIntent Classifier

Featurizers and their corresponding intent classifier for intent classification

## Training the model

- Your conversational agent's processing pipeline is defined in the `config.yml` file
- An example of configuring the `supervised_embeddings` pipeline:
  1. Define the language indicator and the pipeline name in `config.yml` file:

```
language: "en"  
pipeline: "supervised_embeddings"
```

2. Run the Rasa CLI command `rasa train nlu`

This command will train the model on your training data and save it in a directory called `models`

Test the newly trained model on the command line by running the command `rasa shell nlu`:

Type a message in your terminal, for example, “Hello there.” Rasa CLI outputs a JSON object containing several useful pieces of data:

- The intent the model thinks is the most likely match for the message
  - For example: {“name: greet”, “confidence: 0.95347273804”}. This means the model is 95% certain “Hello there” is a greeting
- A list of extracted entities
- A list of intent\_rankings
  - Results showing the intent classification for all of the other intents defined in the training data
  - Intents are ranked according to the intent match probability predictions generated by the model

# Building a conversational agent

## Creating a new agent

- Files
- Commands

## Creating NLU training data

- Basics of conversational design
- Defining intents and entities
- NLU training pipeline

## Dialogue management

- Stories
- Rules
- Domain, custom actions and slots
- Dialogue policies

Stories are basic training units of dialogue training data that detail the back and forth conversation between user and assistant

Stories contain:

- User messages (intent labels and entities extracted by the NLU model)
- Actions: All actions executed by the bot, including responses are listed in stories under the action key

### An example of stories file:

```
stories:  
- story: happy path  
  steps:  
  - intent: greet  
  - action: utter_greet  
  - intent: mood_great  
  - action: utter_happy
```

- Location: your\_rasa\_project\data\stories.yml

You can be quite expressive in a story file:

```
stories:  
- story: newsletter signup  
  steps:  
  - intent: signup_newsletter  
  - action: utter_ask_confirm_signup  
  - or:  
    - intent: affirm  
    - intent: thanks  
  - action: action_signup_newsletter
```

You could, for example, use `or` statements. The story above uses an `or` statement to indicate that a user can use either the `affirm` or the `thanks` intent to confirm a signup.

You can also to use checkpoints in your stories to connect stories:

```
stories:  
- story: beginning of conversation  
  steps:  
  - intent: greet  
  - action: utter_greet  
  - checkpoint: ask_feedback  
- story: provide feedback  
  - checkpoint: ask_feedback  
  - action: utter_ask_feedback  
  - intent: inform  
  - action: utter_thank_you  
  - action: utter_anything_else  
- story: no feedback  
  - checkpoint: ask_feedback  
  - action: utter_ask_feedback  
  - intent: deny  
  - action: utter_no_problem  
  - action: utter_anything_else
```

Put a checkpoint at the end of one story

Put the same checkpoint at the start of another story that you want to connect

### Designing Stories

Two groups of conversational interactions that need to be accounted for: happy and unhappy paths

- Happy paths describe when the user is following the conversation flow as you'd expect and always providing the necessary information when prompted
- unhappy path: users deviate from happy paths with questions, chit chat, or other asks

**Recommendation:** conversation-driven-development when designing unhappy paths

- Share your bot as early as possible with test users and collect real conversation data that tells you exactly how users diverge from the happy paths
- From this data, you can create stories to accomplish what the user is requesting and start to think about ways to guide them back into a happy path

A way to describe dialogue sequences that should always go the same way

Rules can be useful when implementing:

- One-turn interactions:
- Fallback behaviour
- Forms

Because rules do not generalize to unseen conversations, you should reserve them for single-turn conversation snippets, and instead use stories to train on multi-turn conversations.

### An example of rules file:

```
rules:  
  - rule: Greeting Rule  
    steps:  
      - intent: greet  
      - action: utter_greet
```

- whenever I see a user use the greet intent, the response should always be the `utter_greet` response



### The domain file

The `domain.yml` defines the environment in which the assistant operates.

It contains:

- **Responses:** the things the assistant can say to users.
- **Intents:** the different intentions users have
- **Slots:** Variables remembered over the course of a conversation.
- **Entities:** Pieces of information extracted from incoming text.
- **Forms and actions:** Add application logic & extend what your assistant can do.

Example snippet from a `domain.yml` file

```
responses:  
  utter_greet:  
    - text: "Hey there!"  
  utter_goodbye:  
    - text: "Goodbye :("  
  utter_default:  
    - text: "Sorry, I didn't get that, can you rephrase?"  
  utter_youarewelcome:  
    - text: "You're very welcome!"  
  utter_iamabot:  
    - text: "I am a bot, powered by Rasa."
```

- There is a `utter_<thing>` naming convention so that each response starts with "utter"
- Note that it is recommended to have a `utter_iamabot` in your domain file since because assistant should be able to explain that they are not a human

You can also define responses that are dynamic:

- In this case, Rasa will randomly select one of the two responses whenever it needs to send the `utter_greet` response. It will also fill in the `{name}` variable with a slot value if there is one that's available.

```
responses:  
  utter_greet:  
    - text: "Hey, {name}. How are you?"  
    - text: "Hey, {name}. How is your day going?"
```

# Dialogue management

## Domain, actions and slots

You are also able to define responses that contain images or buttons as well.

```
responses:
  utter_greet:
    - text: "Hey! How are you?"
      buttons:
        - title: "great"
          payload: "/mood_great"
        - title: "super sad"
          payload: "/mood_sad"
  utter_cheer_up:
    - text: "Here is something to cheer you up:"
      image: "https://i.imgur.com/nGF1K8f.jpg"
```

You can even customise the message based on the channel that you're using.

```
responses:
  utter_ask_game:
    - text: "Which game would you like to play on Slack?"
      channel: "slack"
    - text: "Which game would you like to play?"
```

This way, slack users will be able to get a different message.

### Actions

The section called actions should contain the list of all utterances and custom actions an assistant should use to respond to a user's input. These should come from your stories data in the `stories.md` file.

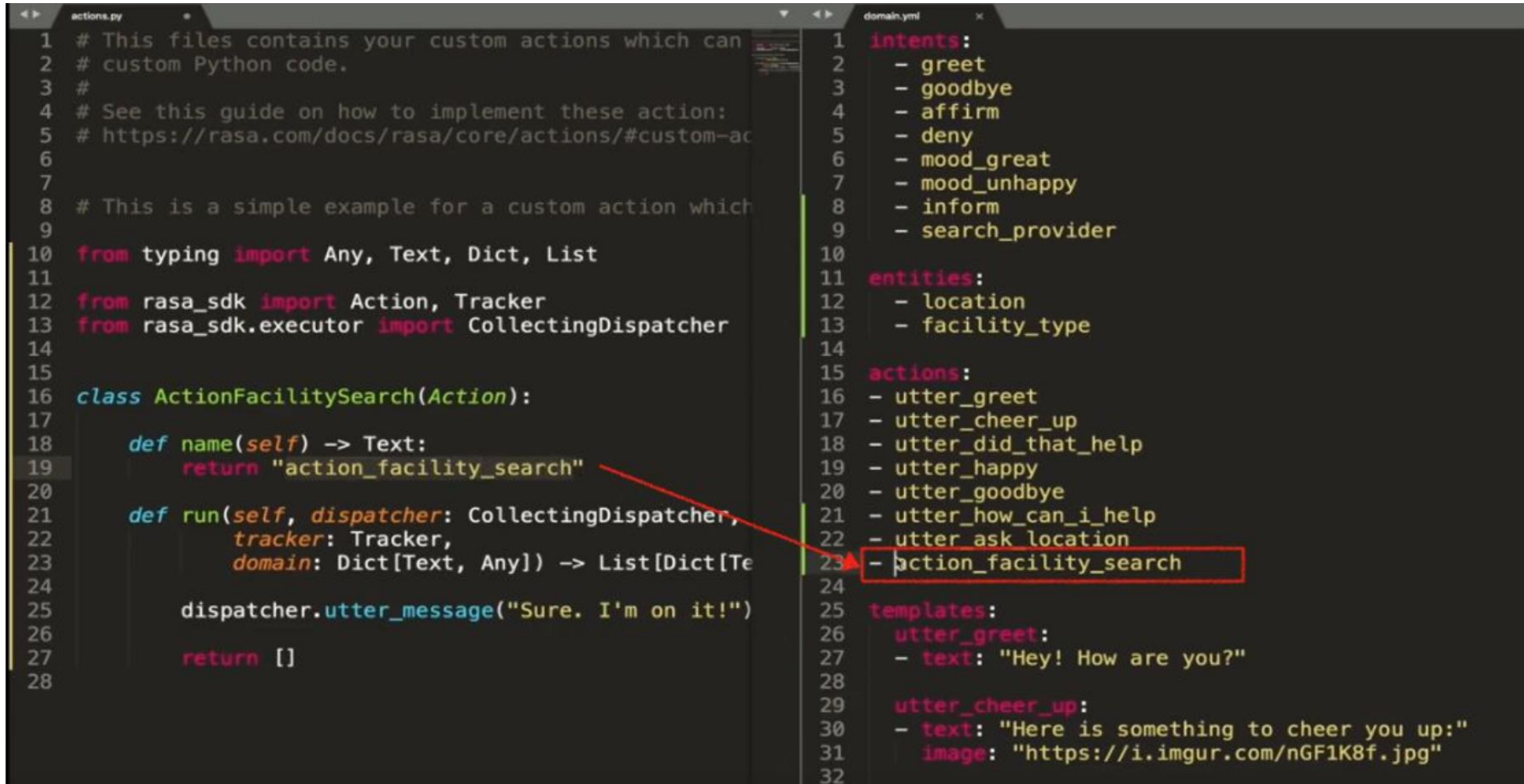
### Custom actions

- Custom actions are response actions which include custom code
- Can define anything from a simple text response to a backend integration - an API call, connecting to the database, or anything else your assistant needs to do
- Custom actions are defined in a file called `actions.py`

# Dialogue management

Domain, actions and slots

**Important:** the names of these actions must match the actions included in the domain file



```
actions.py
1 # This files contains your custom actions which can
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-ac
6
7
8 # This is a simple example for a custom action which
9
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14
15
16 class ActionFacilitySearch(Action):
17
18     def name(self) -> Text:
19         return "action_facility_search"
20
21     def run(self, dispatcher: CollectingDispatcher,
22            tracker: Tracker,
23            domain: Dict[Text, Any]) -> List[Dict[Te
24
25         dispatcher.utter_message("Sure. I'm on it!")
26
27         return []
28
domain.yml
1 intents:
2     - greet
3     - goodbye
4     - affirm
5     - deny
6     - mood_great
7     - mood_unhappy
8     - inform
9     - search_provider
10
11 entities:
12     - location
13     - facility_type
14
15 actions:
16     - utter_greet
17     - utter_cheer_up
18     - utter_did_that_help
19     - utter_happy
20     - utter_goodbye
21     - utter_how_can_i_help
22     - utter_ask_location
23     - action_facility_search
24
25 templates:
26     utter_greet:
27         - text: "Hey! How are you?"
28
29     utter_cheer_up:
30         - text: "Here is something to cheer you up:"
31           image: "https://i.imgur.com/nGF1K8f.jpg"
32
```

<https://rasa.com/blog/the-rasa-masterclass-handbook-episode-6-2/>

### Slots: the agent's long-term memory

- To store any information for later use
- Need two pieces of information:
  - slot name (Can match the names of the entities)
  - a slot type (Text, bool, categorical, float, list, unfeaturized)
- Difference to entity:
  - You could store any information in a slot, even if no entity has been detected
  - It is very common to fill a slot value with an entity value

```
entities:  
  - destination  
  
slots:  
  destination:  
    type: text  
    influence_conversation: false
```

### 🔑 **influence\_conversation tag**

Slots can influence a story.

If your slots are configured to influence the flow of the conversation, you have to include them in your training stories.

# Dialogue management

## Domain, actions and slots



All slots have to be listed in the domain file

```
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-ac
6
7
8 # This is a simple example for a custom action which
9
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14 from rasa_sdk.events import SlotSet
15
16
17 class ActionFacilitySearch(Action):
18
19     def name(self) -> Text:
20         return "action_facility_search"
21
22     def run(self, dispatcher: CollectingDispatcher,
23             tracker: Tracker,
24             domain: Dict[Text, Any]) -> List[Dict[Te
25
26         facility = tracker.get_slot("facility_type")
27         address = "300 Hyde St, San Francisco"
28         dispatcher.utter_message("Here is the address")
29
30         return [SlotSet("address", address)]
31
32
33 - greet
34 - goodbye
35 - affirm
36 - deny
37 - mood_great
38 - mood_unhappy
39 - inform
40 - search_provider
41
42 entities:
43 - location
44 - facility_type
45
46 slots:
47     location:
48         type: text
49     facility_type:
50         type: text
51     address:
52         type: unfeaturized
53
54 actions:
55 - utter_greet
56 - utter_cheer_up
57 - utter_did_that_help
58 - utter_happy
59 - utter_goodbye
60 - utter_how_can_i_help
61 - utter_ask_location
62 - action_facility_search
```

<https://rasa.com/blog/the-rasa-masterclass-handbook-episode-6-2/>

# Dialogue management

## Dialogue policies

Policies are components that train the dialogue model, and they play a very important role in determining its behaviour

- The policy configuration is defined by a list of policy names, along with optional parameters that can be configured by developers
- Dialogue policies run in parallel

Default configuration generated by `rasa init`:

```
config.yml
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: en
pipeline: pretrained_embeddings_spacy

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  - name: MemoizationPolicy
  - name: KerasPolicy
  - name: MappingPolicy
```

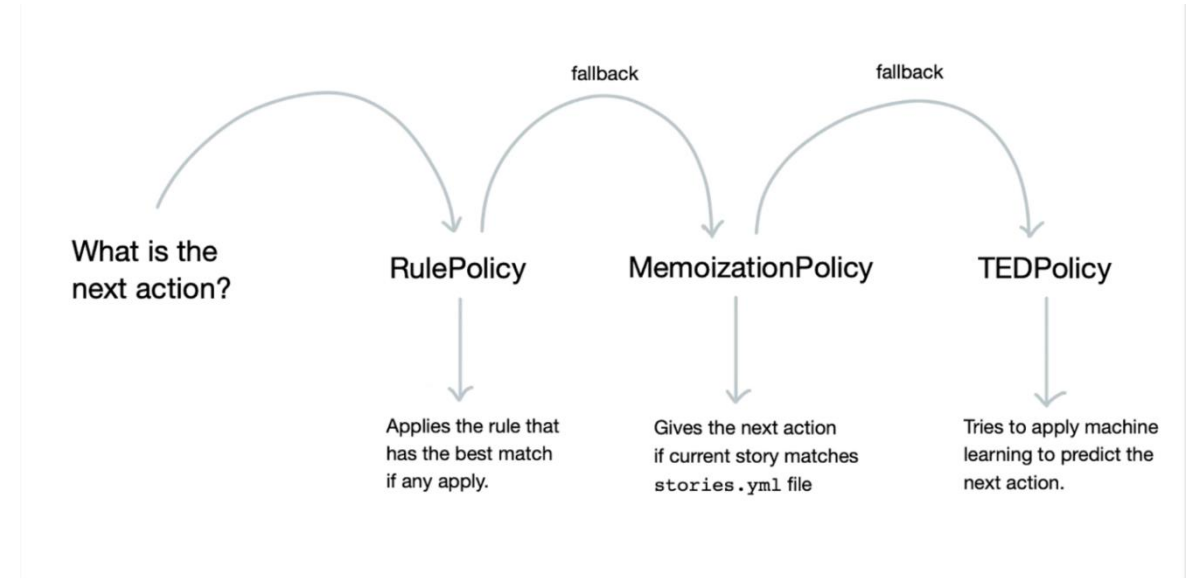
<https://rasa.com/blog/the-rasa-masterclass-handbook-episode-7/>



Major Rasa training policies:

- Rule Policy
  - handles conversations that match predefined rule patterns. It makes predictions based on any rules you have in your `rules.yml` file.
- Memorization Policy
  - checks if the current conversation matches any of the stories in your training data. If so, it will predict the next action from the matching stories.
- TED Policy
  - uses Transformer model to predict the next best action

These policies operate in a priority based hierarchy:



<https://rasa.com/blog/dialogue-policies-rasa-2/>

- Freed, A. (2021). Conversational AI: Chatbots that Work. Manning Publications.
- Harms, J. G., Kucherbaev, P., Bozzon, A., & Houben, G. J. (2018). Approaches for dialog management in conversational agents. IEEE Internet Computing, 23(2), 13-22.
- Hussain, S., Ameri Sianaki, O., & Ababneh, N. (2019). A survey on conversational agents/chatbots classification and design techniques. In Web, Artificial Intelligence and Network Applications: Proceedings of the Workshops of the 33rd International Conference on Advanced Information Networking and Applications (WAINA-2019) 33 (pp. 946-956). Springer International Publishing.
- McTear, M. (2020). Conversational AI: dialogue systems, conversational agents, and chatbots. Synthesis Lectures on Human Language Technologies, 13(3), 1-251.
- Ram, A., Prasad, R., Khatri, C., Venkatesh, A., Gabriel, R., Liu, Q., ... & Pettigrew, A. (2018). Conversational AI: The science behind the alexa prize. 1st Proceedings of Alexa Prize (Alexa Prize 2017).
- Getting started with Rasa, accessed January 2023. <https://rasa.com/docs/>
- NLU training data format, accessed January 2023. <https://rasa.com/docs/rasa/nlu-training-data/>
- Choosing a pipeline, accessed January 2023. <https://rasa.com/docs/rasa/tuning-your-model/>
- NLU components, accessed January 2023. <https://rasa.com/docs/rasa/components/#ducklinghttpextractor>
- Supervised Word Vectors from Scratch in Rasa NLU, accessed January 2023. <https://medium.com/rasa-blog/supervised-word-vectors-from-scratch-in-rasa-nlu-6daf794efcd8>
- SpaCy 101, accessed January 2023. <https://spacy.io/usage/spacy-101>
- Rasa: Open Source Language Understanding and Dialogue Management, accessed January 2023. <https://arxiv.org/abs/1712.05181>
- Dialogue policies, accessed January 2023. <https://rasa.com/docs/rasa/policies/>

**Acknowledgments:** The slides for this lecture were created by Phillip Schneider and Yuting Zhao.



## Phillip Schneider

Technical University of Munich (TUM)  
TUM School of CIT  
Department of Computer Science (CS)  
Chair of Software Engineering for Business  
Information Systems (sebis)

Boltzmannstraße 3  
85748 Garching bei München

+49.89.289.17131  
phillip.schneider@tum.de  
[www.matthes.in.tum.de](http://www.matthes.in.tum.de)

