Rasa Certification Workshop

Mady Mantha, Juste Petraityte, Karen White

Agenda

Part 1: Welcome and deep dive into Rasa

- Welcome! 👋
- Intro to Rasa
- Conversation Design
- Deep dive into NLU
- Deep dive into dialogue policies
- CDD and testing
- Demo

Part 2: Adding custom actions, forms and improving your assistant using Rasa X

- Reviewing the previous day 📚
- Events and Slots
- Custom actions and Forms
- Improving the assistant using Rasa X
- Connecting the assistant to the external messaging channels
- What's next

The Rasa Team

Mady Mantha
Sr. Technical Evangelist

Juste Petraityte
Head of Developer Relations

Karen White
Developer Marketing Manager

How to get help

- Please ask your questions in the **#workshop-help** Slack channel
- A note on time zones:
  - The Rasa team is based across the US and in Berlin. We'll do our best to answer questions within team members' working hours, but please keep in mind, some discussions may need to take place async rather than in real time.

Intro to Rasa

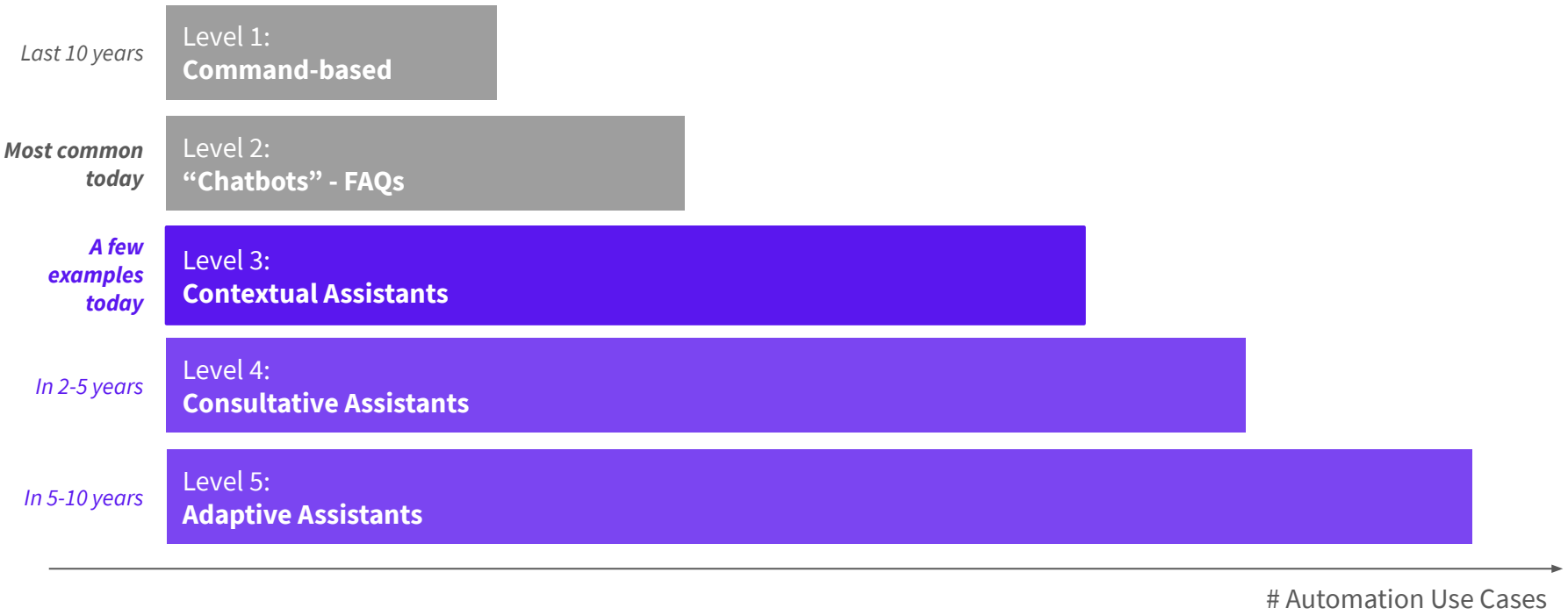
Part 1 Roadmap

- First half: talk
- Second half: we'll code together
- Recap: talk, with some time for questions

*More info blablabla

5 LEVELS OF CONVERSATIONAL AI MATURITY MODEL

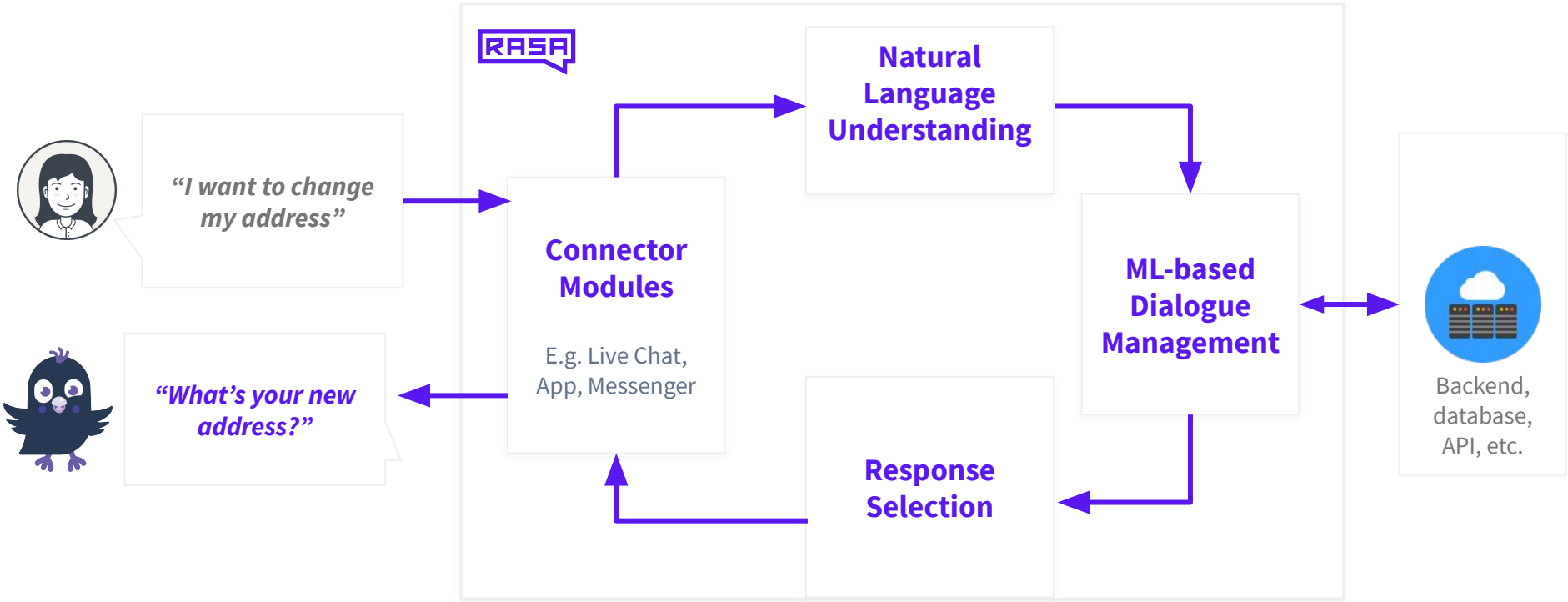
Contextual assistants are an important step on the journey to autonomous organizations



RASA

RASA OPEN SOURCE

Rasa Open Source

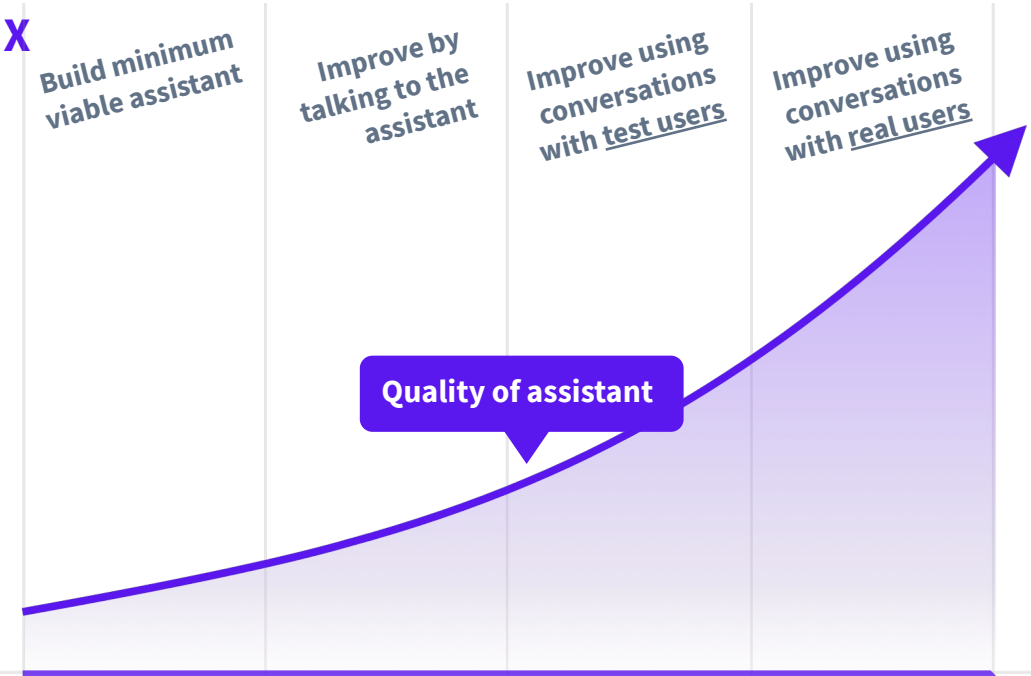


Conversational AI is not easy

Real conversations don’t follow the happy path

THE APPROACH

Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X



Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

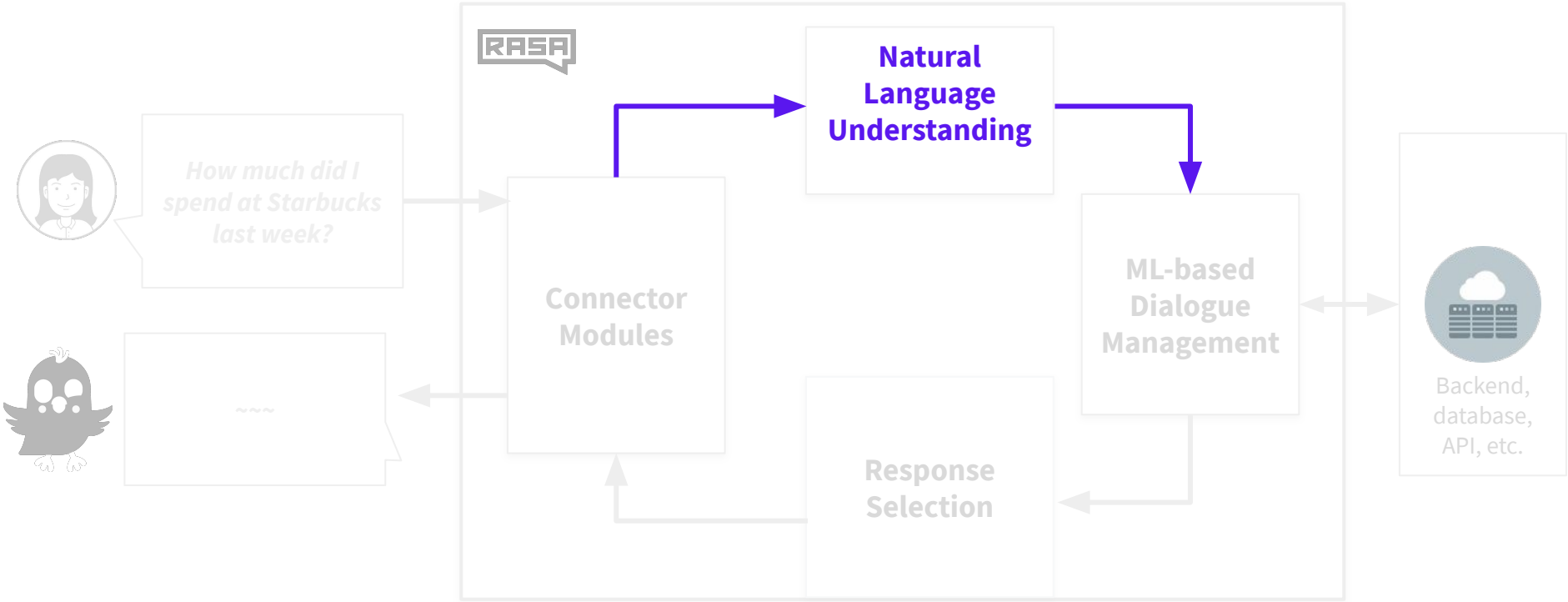
Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

RASA

NLU & Dialogue Management

NLU Deep Dive

Natural Language Understanding (NLU)



Natural Language Understanding (NLU)

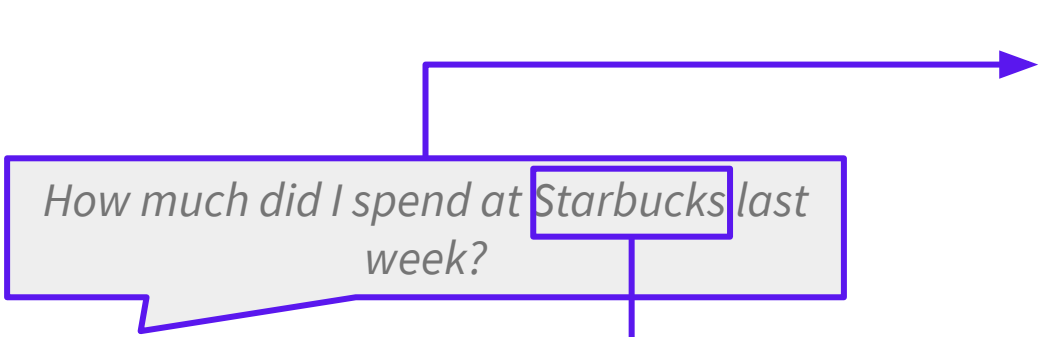
Goal: Extract structured information from messages

How much did I spend at Starbucks last week?


```
{...
    "intent": {
        "name": "search_transactions",
        "confidence": 0.96
    },
    "entities": [
        {
            "entity": "vendor_name",
            "value": "Starbucks",
              ...
        }
     ]
}
```

Intents and Entities

Two of the most common and necessary types of information to extract from a message

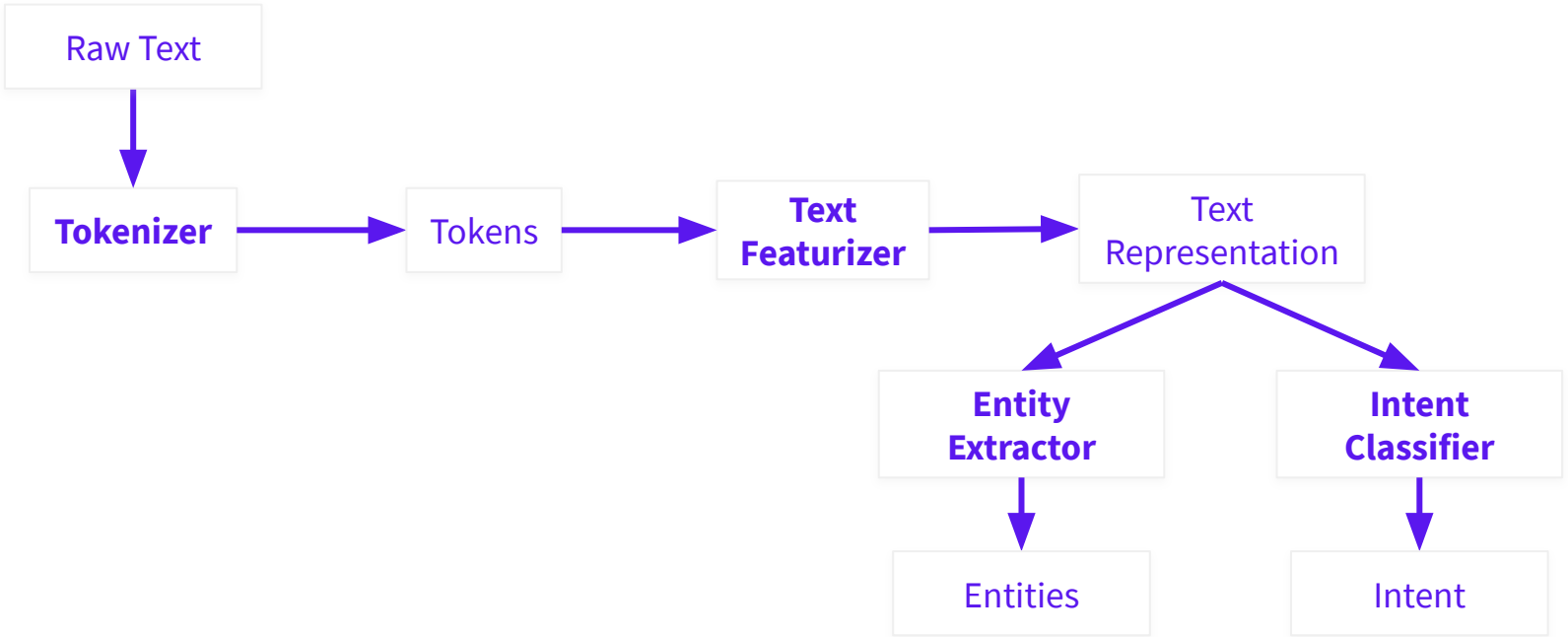


Intent: What the user wants to achieve

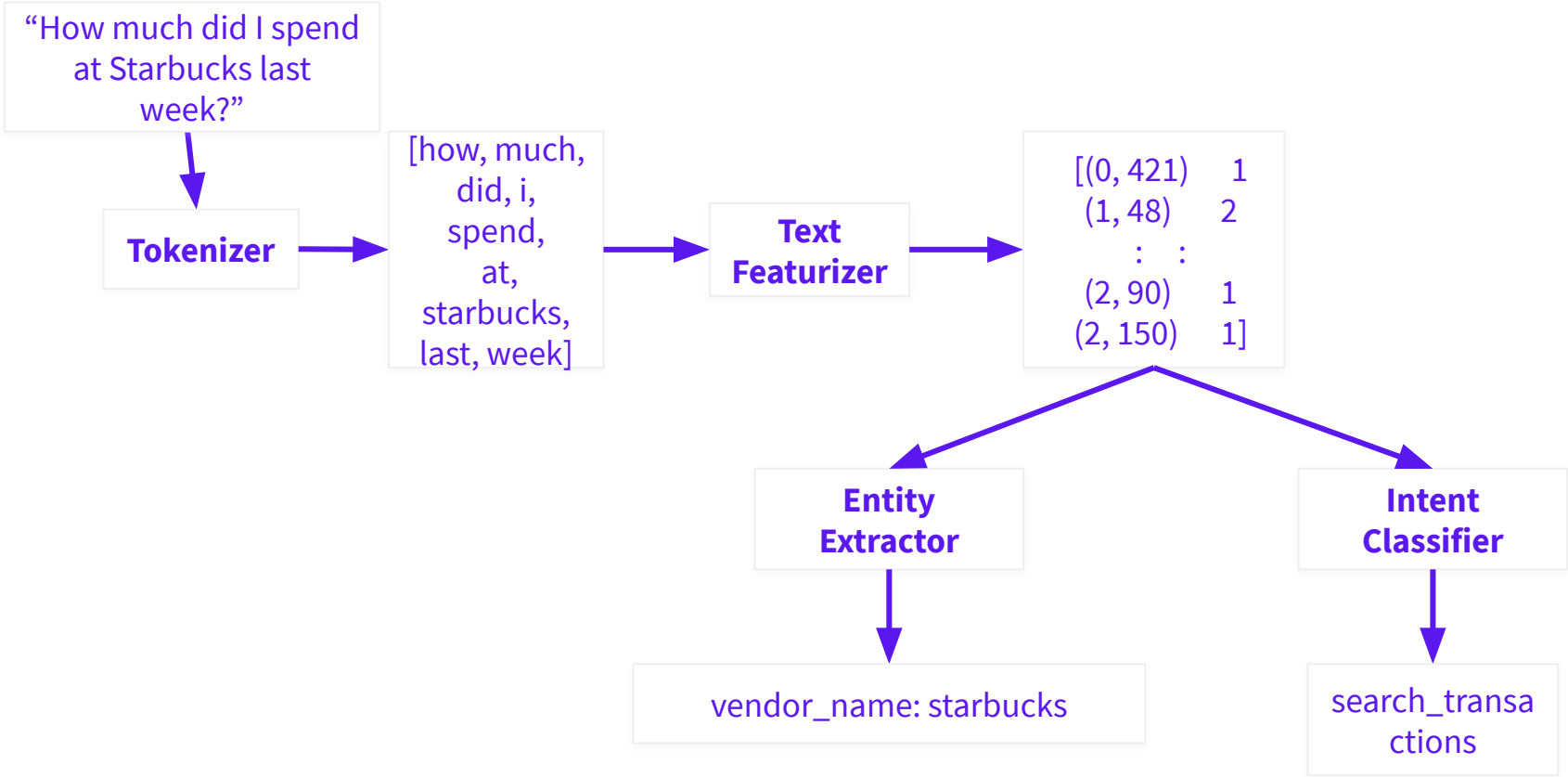
Entity: Key piece of information

NLU Pipeline: Input and Output

Many components take the output of earlier components as input

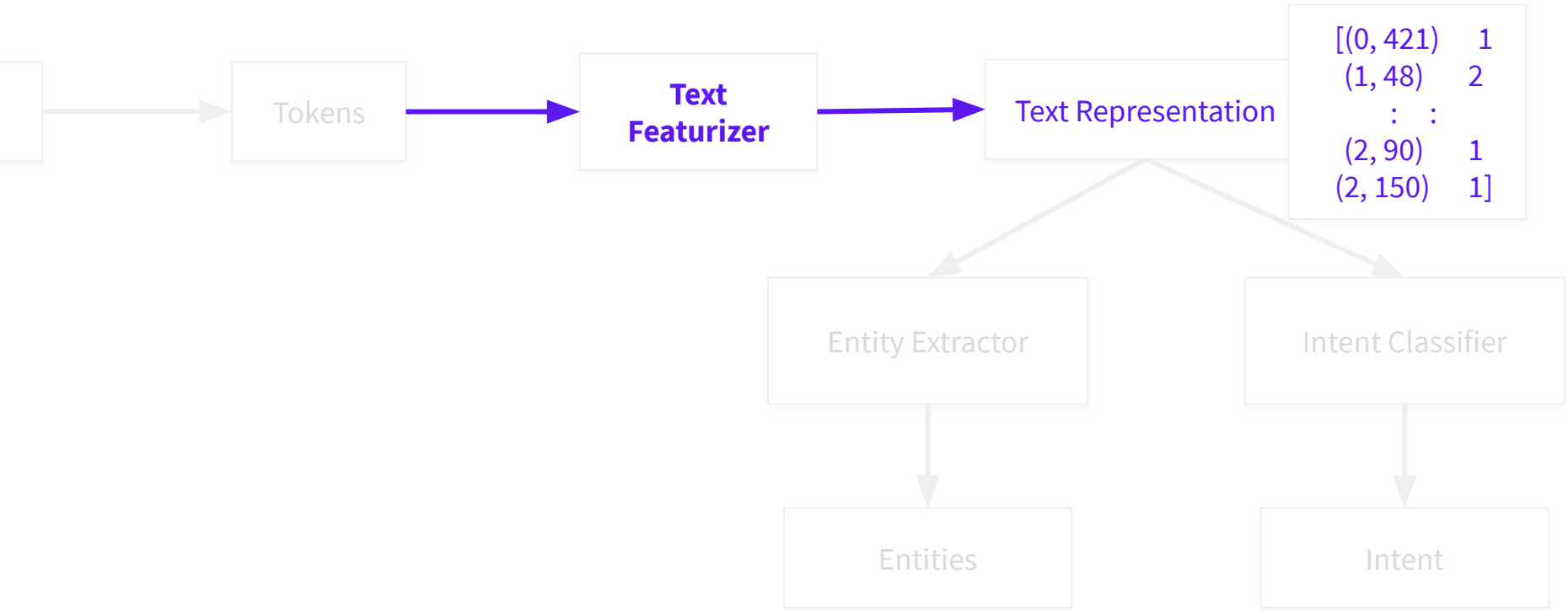


NLU Pipeline: Example



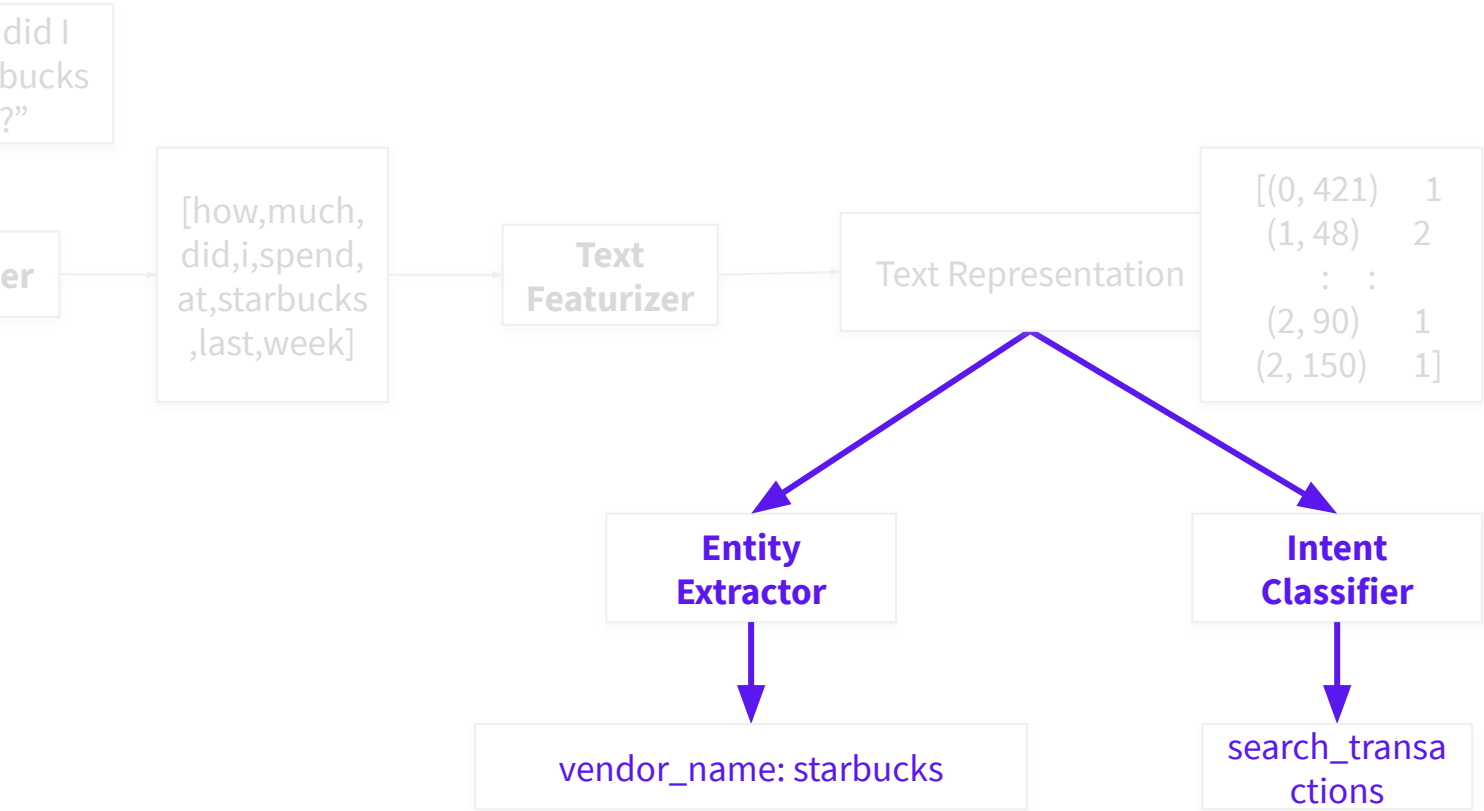
“Text Representation”

How do words turn into numbers?



NLU DEEP DIVE

Intent Classification & Entity Extraction



Text Representation: Word Vectors

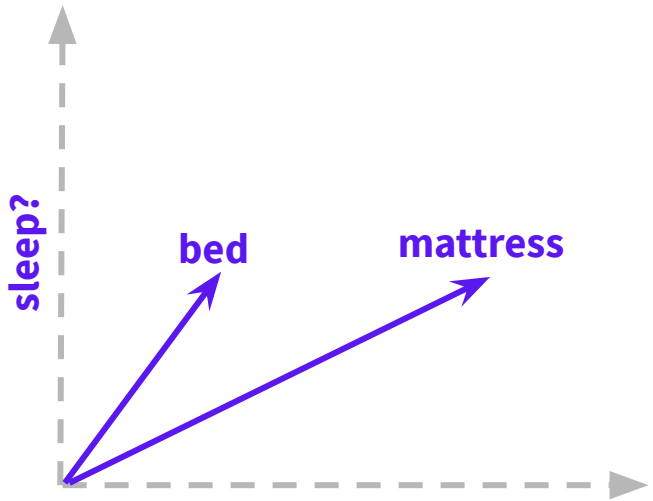
"Judge a Word by the Company it Keeps" — John R. Firth

"... *sleep* on a **bed**..."

"...*sleep* in a **bed**?"

"...*sleep* on a **mattress**..."

"...**mattress** to *sleep* on..."



These would be two of **many** dimensions!

Text Representation: How models use word vectors

Non-sequence model:

- One feature vector per input (whole message)
- Order of words not captured

Sequence model:

- One feature vector per token (word) & feature vector for whole message
- Word order captured

Configuration Files

Livecoding

Let’s start coding!

1. Let’s use a sandbox branch instead of the master branch you cloned yesterday. Make sure you are inside the financial-demo folder:
   cd financial-demo

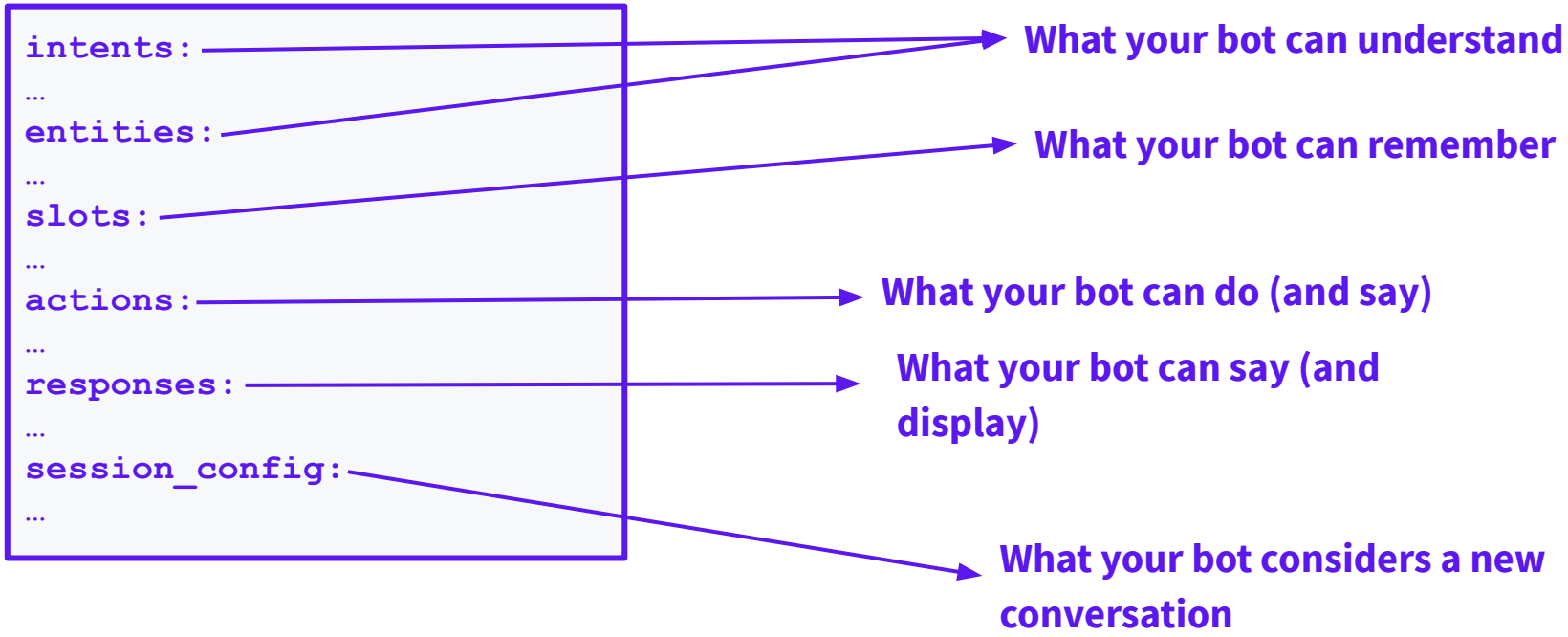
2. Fetch the remote branches:
   git fetch origin

3. Checkout the sandbox branch named “workshop_day_2_sandbox”:
   git checkout -b workshop_day_2_sandbox origin/workshop_day_2_sandbox

Core Deep Dive

Domain

Defines the “world” of your assistant - what it knows, can understand, and can do



Project setup: Files

`__init__.py`	an empty file that helps python find your actions
`actions.py`	code for your custom actions
`config.yml`	configuration of your NLU and dialogue models
`credentials.yml`	details for connecting to other services
`data/`**`nlu.md`**	your NLU training data
`data/`**`stories.md`**	your stories
`domain.yml`	your assistant's domain
`endpoints.yml`	details for connecting to channels like fb messenger
`models/<timestamp>.tar.gz`	your initial model

NLU Pipeline

Defines how a user message is processed & what information is extracted

```
language: "en"

pipeline:
  - name: "Whitespacetokenizer"
  - name: "ConveRTFeaturizer"
  - name: "RegexFeaturizer"
  - name: "LexicalSyntacticFeaturizer"
  - name: "CountVectorsFeaturizer"
```

Components

NLU Pipeline: Components

Built-in component types:

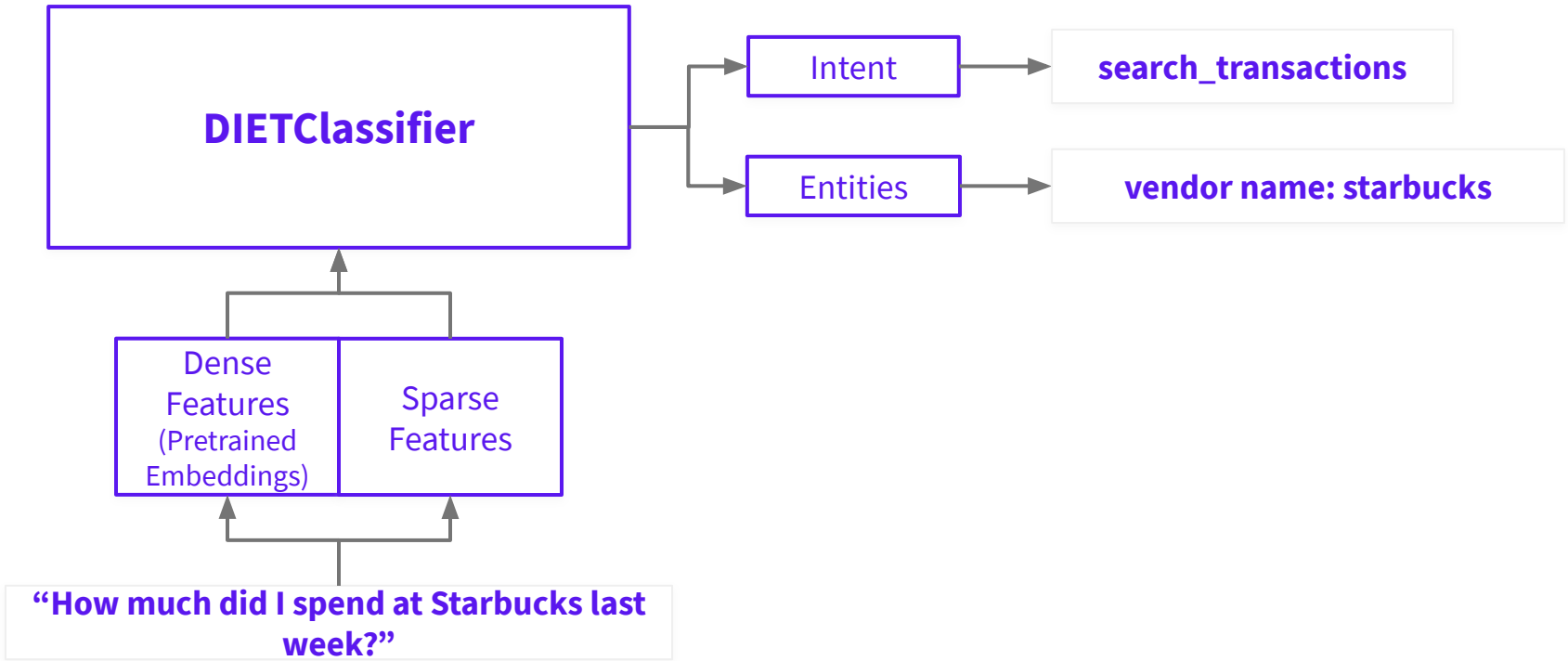
- Tokenizers
- Word Vector Sources
- Text Featurizers
- Entity Extractors
- Intent Classifiers
- Response Selectors

You can also add custom components for e.g.

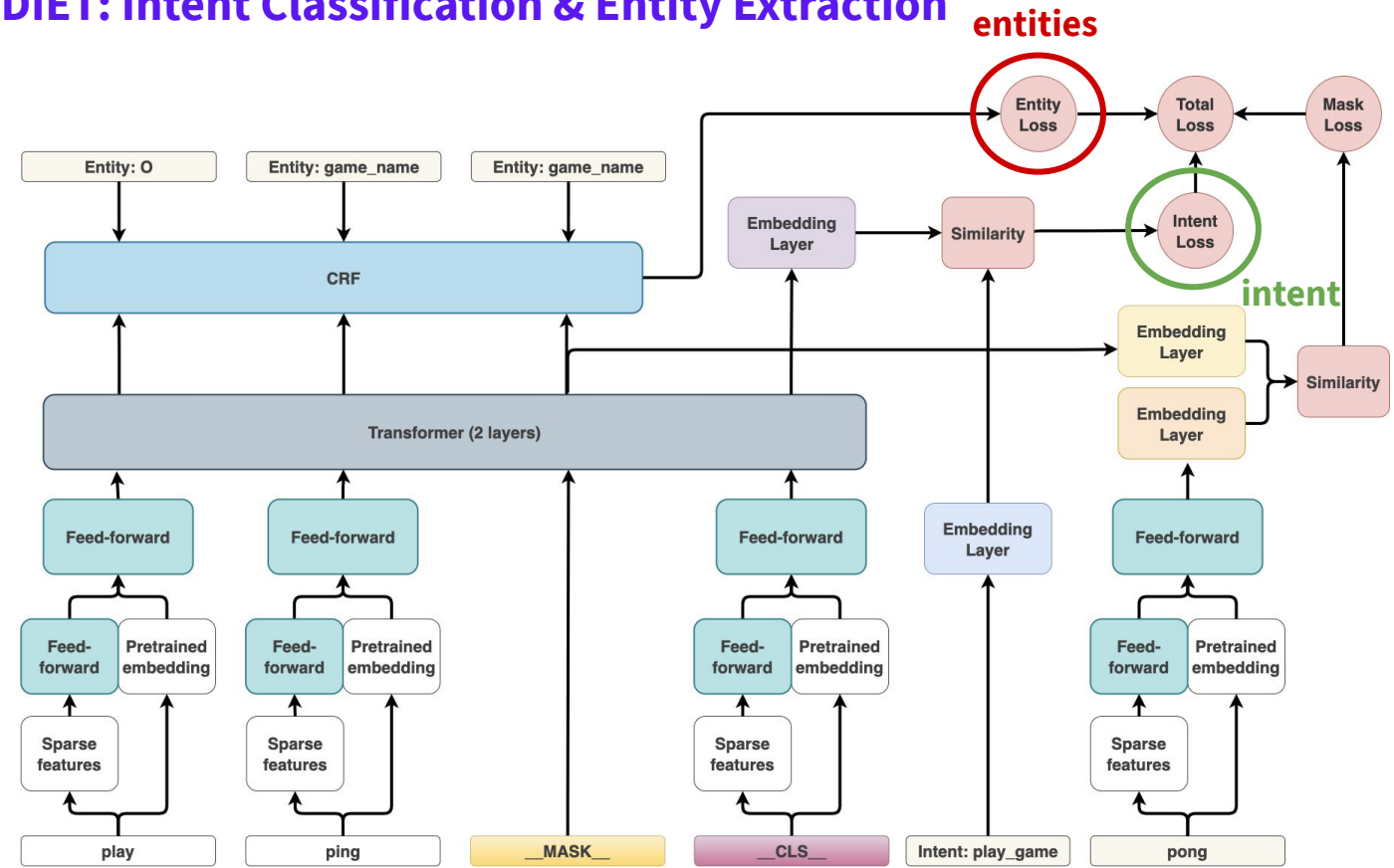
- Sentiment analysis
- Spell checking

DIETClassifier: Intent Classification & Entity Extraction

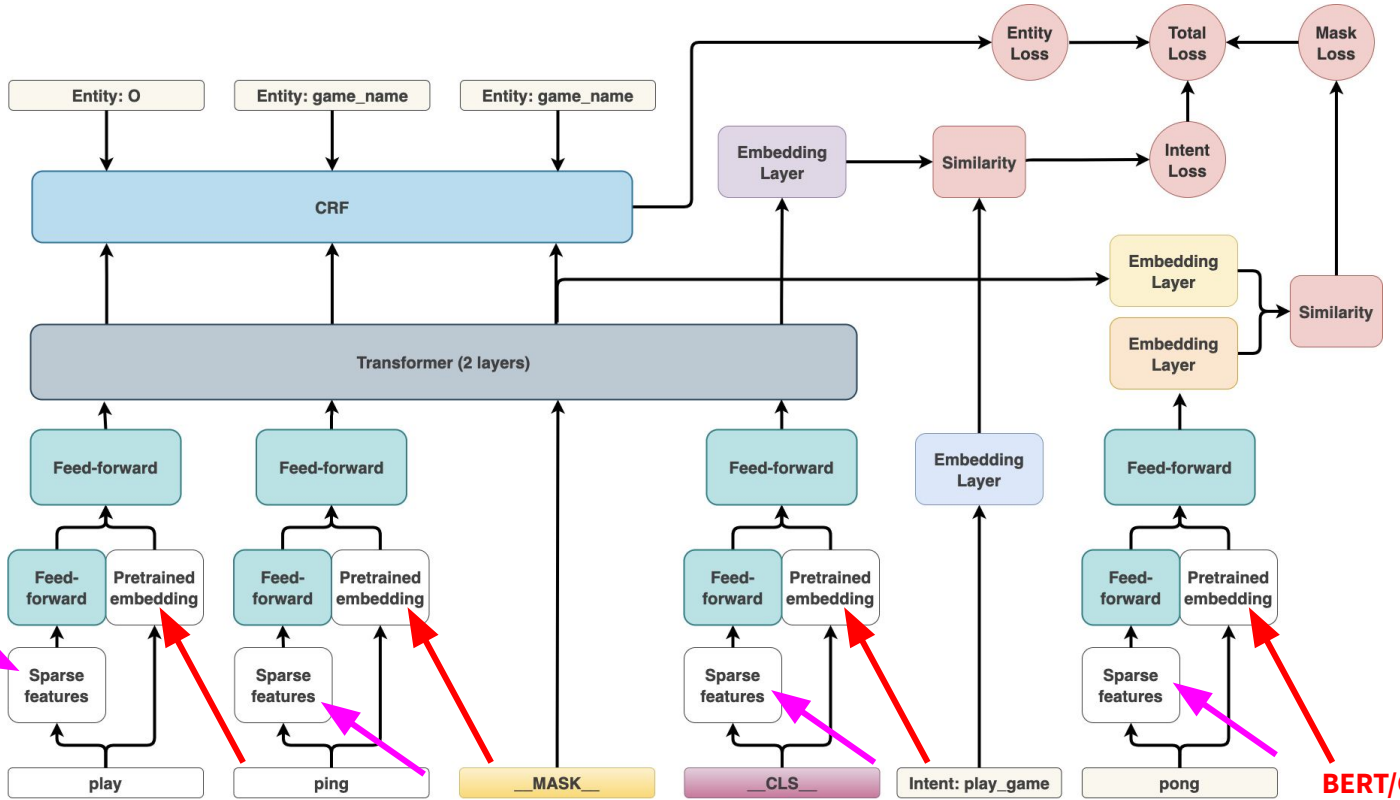
The Short Story



DIET: Intent Classification & Entity Extraction



Feature: Customize What Features Go In!



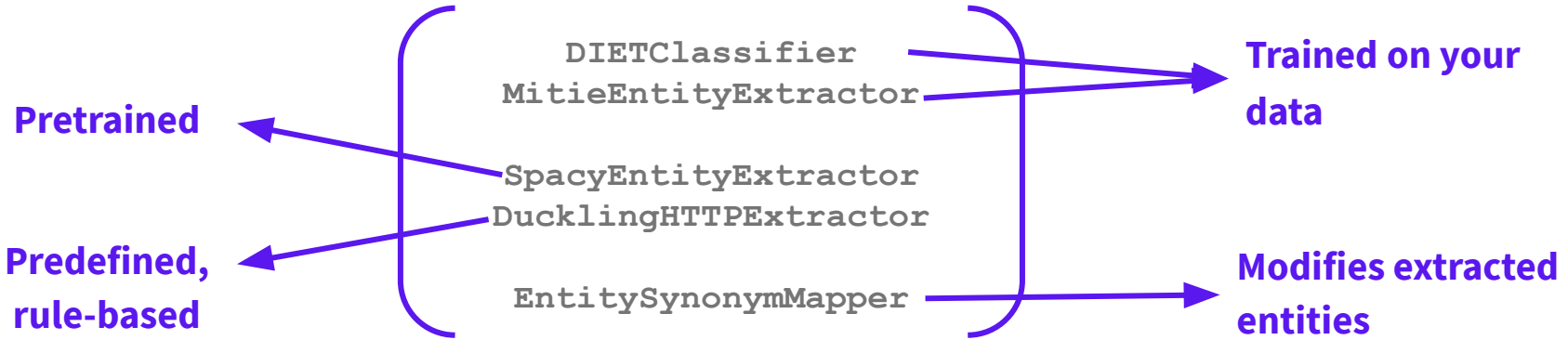
sparse features in pipeline

BERT/GloVe/pretrained and continuous features

Entity Extraction: Other options

Possible Approaches:

1. Rule-based entity extraction
2. Direct structured prediction of entity based on sentence context

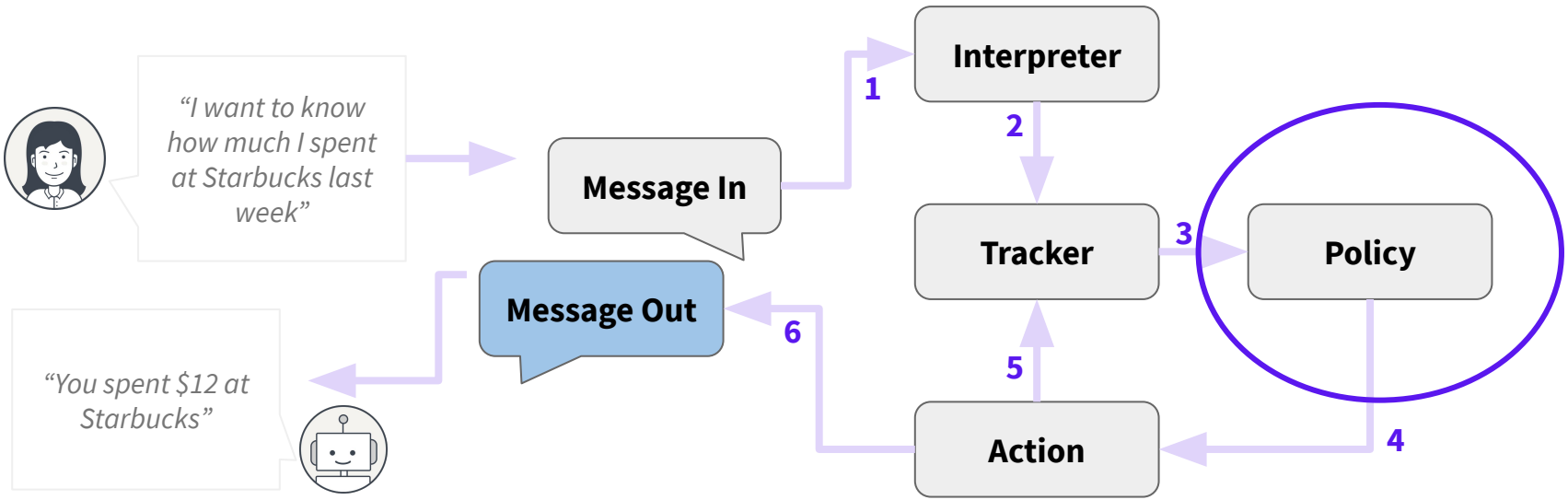


Livecoding

Let’s create some intents and entities!

Policies

- Decide which action to take at every step in the conversation
- Each policy predicts an **action** with some **probability.** This is called **core confidence**.



Rule Based Policies

- **MemoizationPolicy:** Memorizes your stories
  - Makes predictions based on your max_history value
- **AugmentedMemoizationPolicy:** Memorizes your stories with a twist
  - As above, but if no match is found, it will forget certain events until a match is found
- **FormPolicy:** Fills required slots
  - Asks for information until all required slots are filled
- **MappingPolicy:** Intents trigger actions
  - Will execute the specified action regardless of context unless superseded by another policy
- **FallbackPolicy**: Failing gracefully
  - If model confidence is below a certain value, it will send the user a “fallback message”, e.g. “Sorry, I didn’t understand you”
- **TwoStageFallbackPolicy**: Failing (more) gracefully
  - Attempts to disambiguate the user’s intent before sending a fallback message

Machine Learning Based Policies

These policies should be used in conjunction with rule-based policies

- **KerasPolicy:** Uses a standard LSTM to predict the next action
  - Learns the patterns of your stories
  - Good for handling stories that don't exactly match your training data
- **TED Policy:** Uses Attention to Handle Uncooperative Dialogue
  - Requires fewer story examples of uncooperative user dialogue
    - e.g. users who go off on tangents instead of providing the requested information
  - Effectively "ignores" irrelevant parts of the dialogue

Multiple Policies

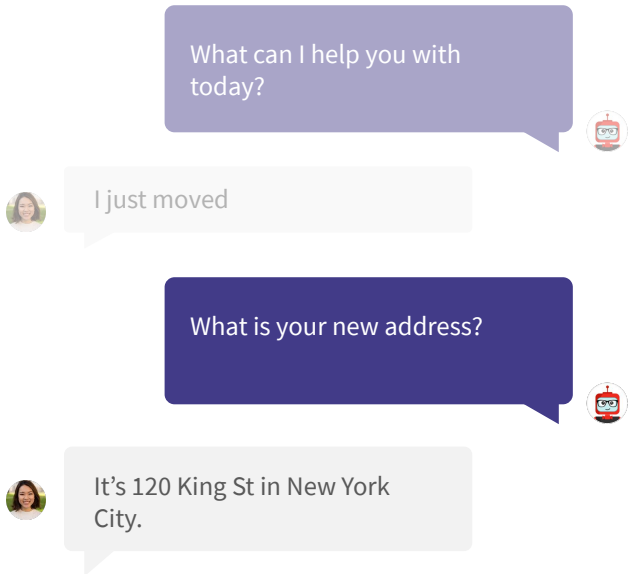
- The policy with the **highest confidence** wins.
- If the confidence of two policies is equal, the policy with the **highest priority** wins.

Rule-based policies have higher priority than ML-based policies

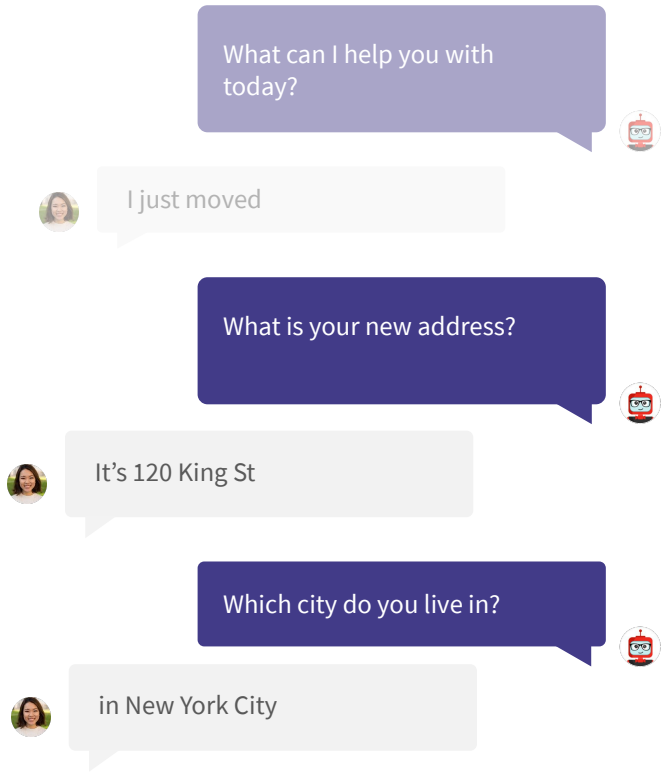
```
Policy priorities
(higher numbers = higher priority)

5. FormPolicy
4. FallbackPolicy, TwoStageFallbackPolicy
3. MemoizationPolicy, AugmentedMemoizationPolicy
2. MappingPolicy
1. EmbeddingPolicy, KerasPolicy, TED
```

Conversation Design

Initially: Conversations as simple question / answer pairs



Real conversations are more complex: Rely on context



Scope Conversation

How to get started with conversation design

The assistant’s purpose

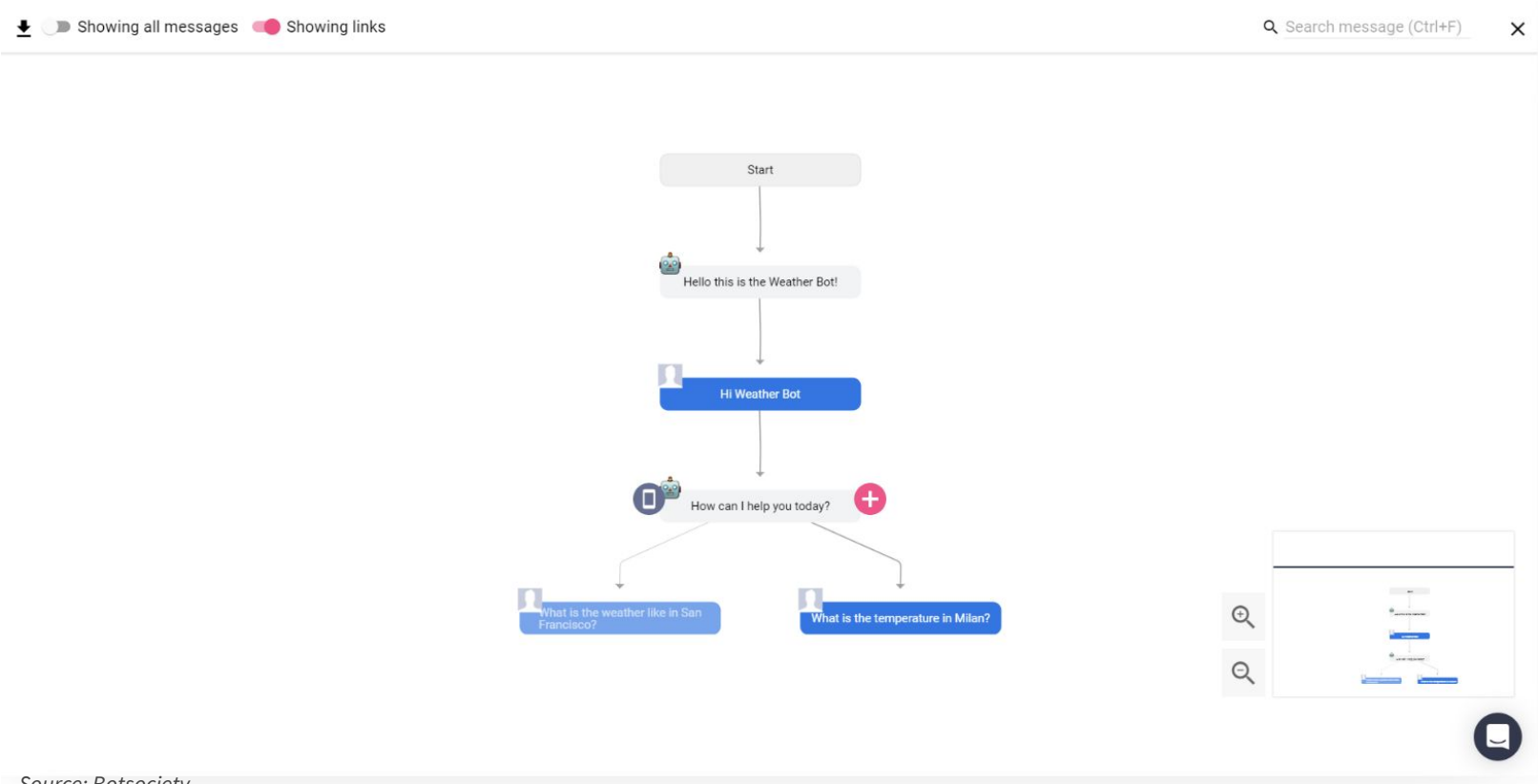
Leverage the knowledge of domain experts

Common search queries

FAQs and wikis

Conversation Design

Scope Conversation



Source: Botsociety

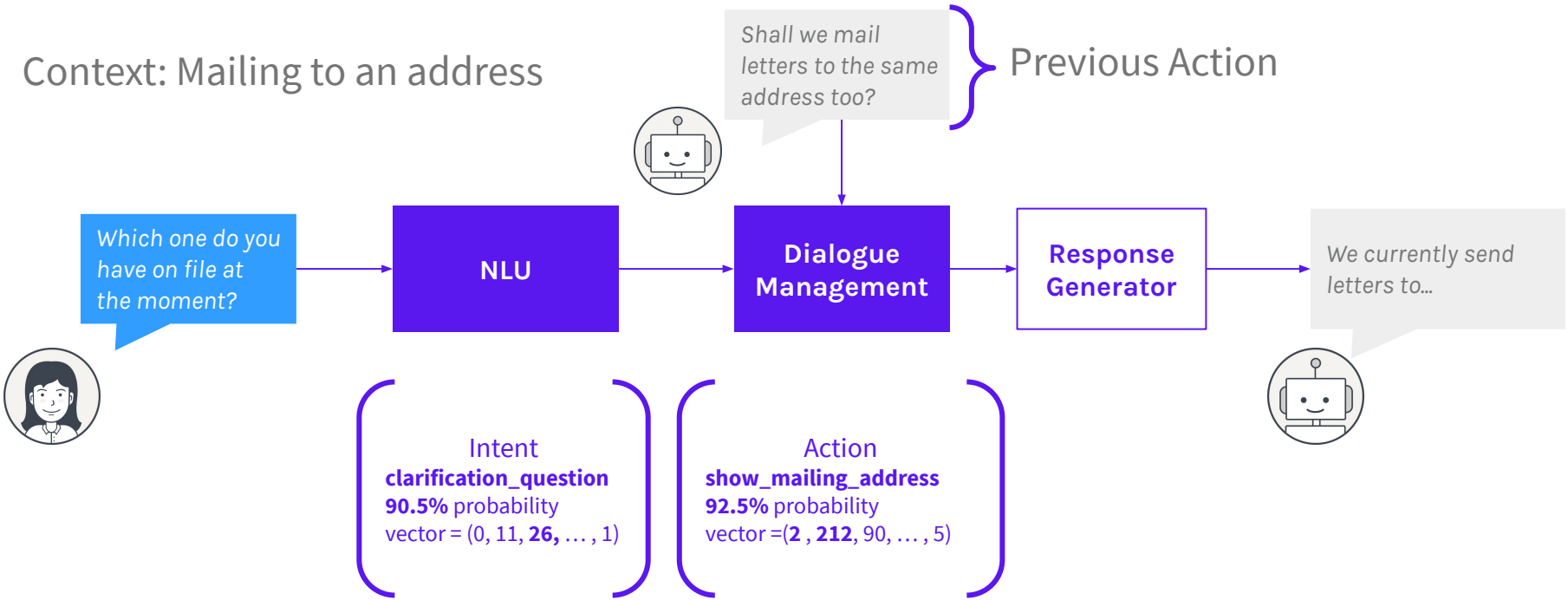
Livecoding

Let’s create some stories together!

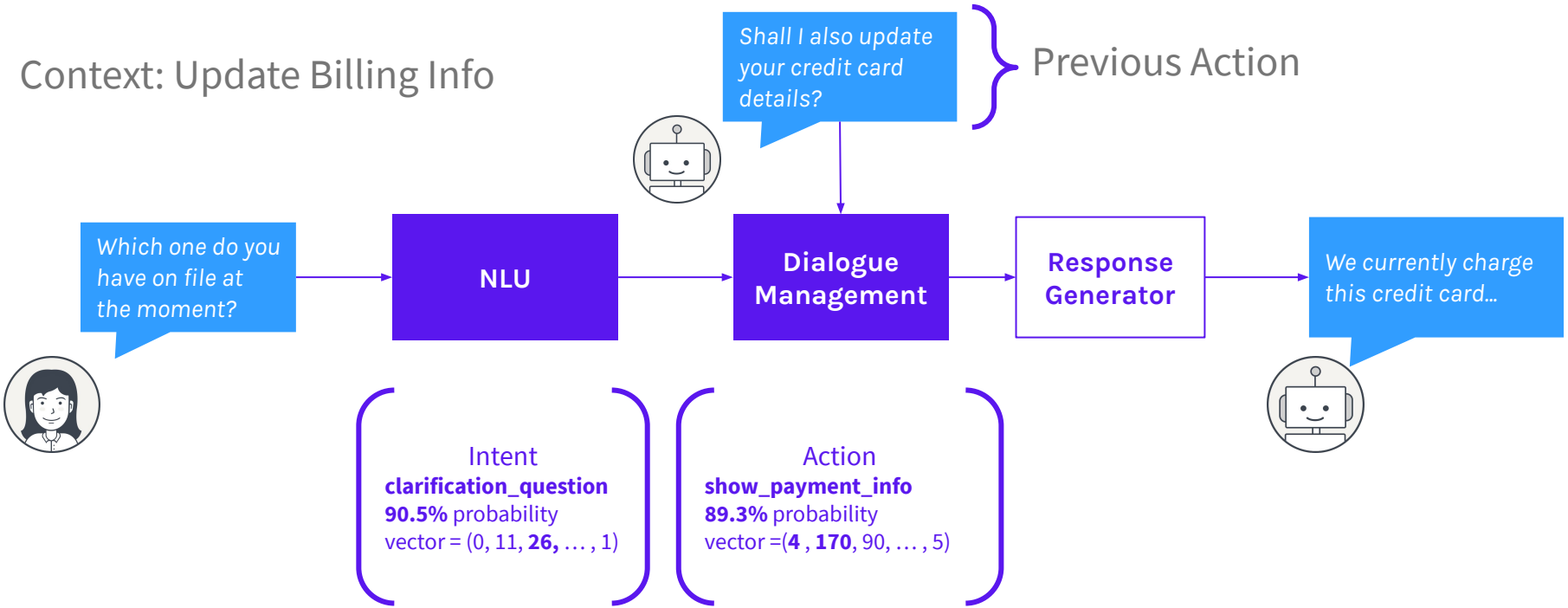
- Check account balance

- Search transactions

The importance of context

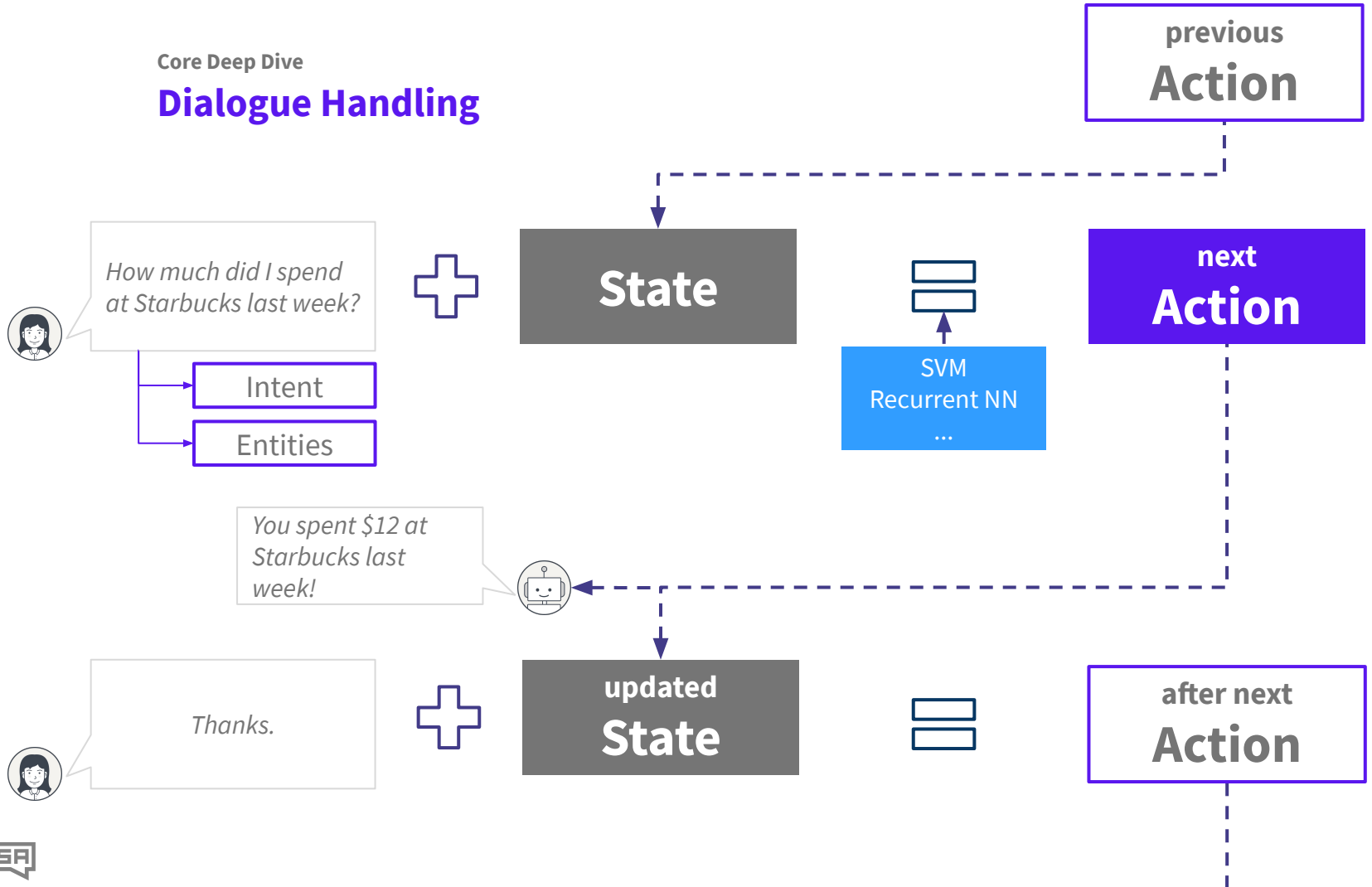


The importance of context

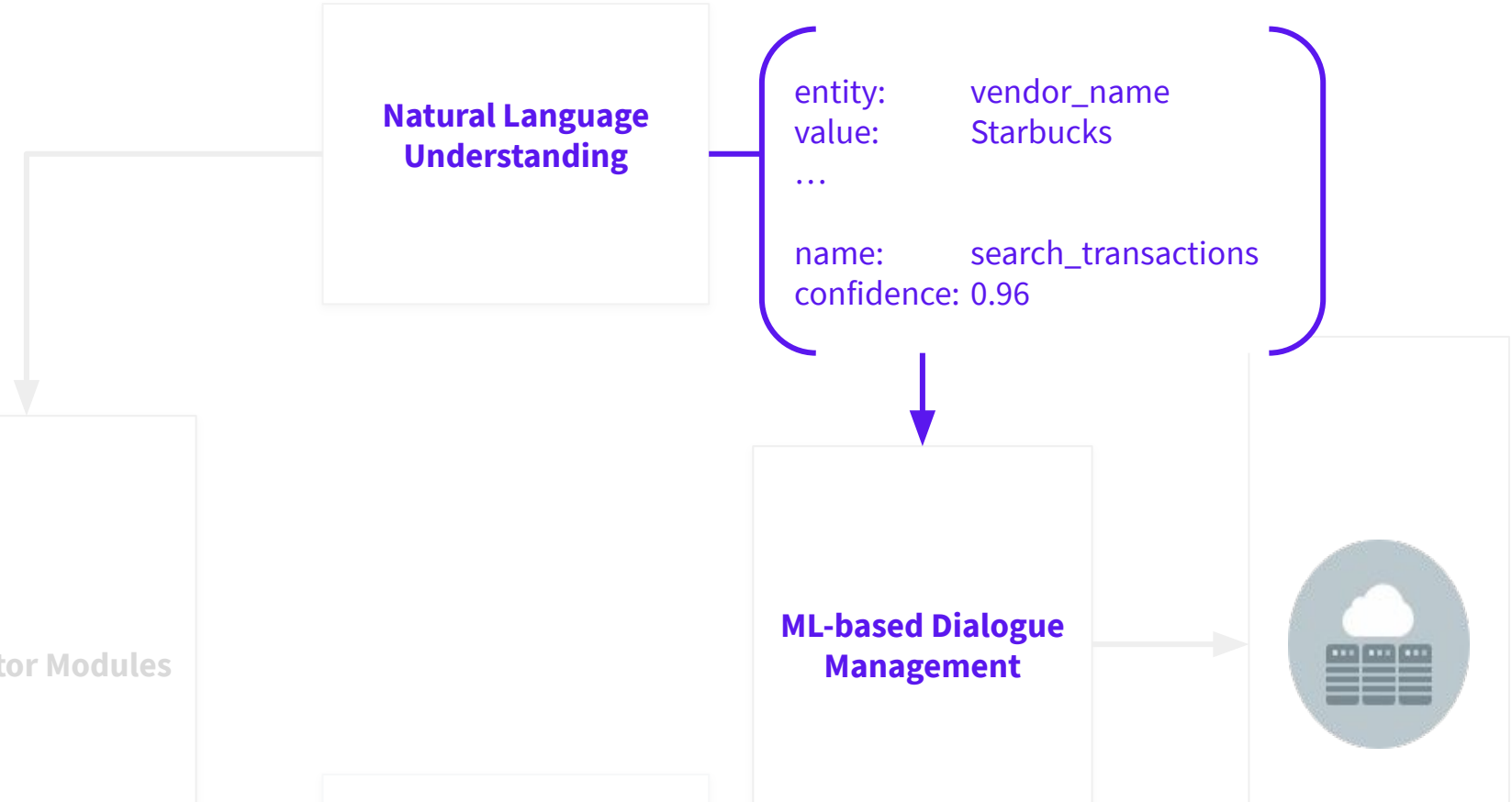


Core Deep Dive

Dialogue Handling

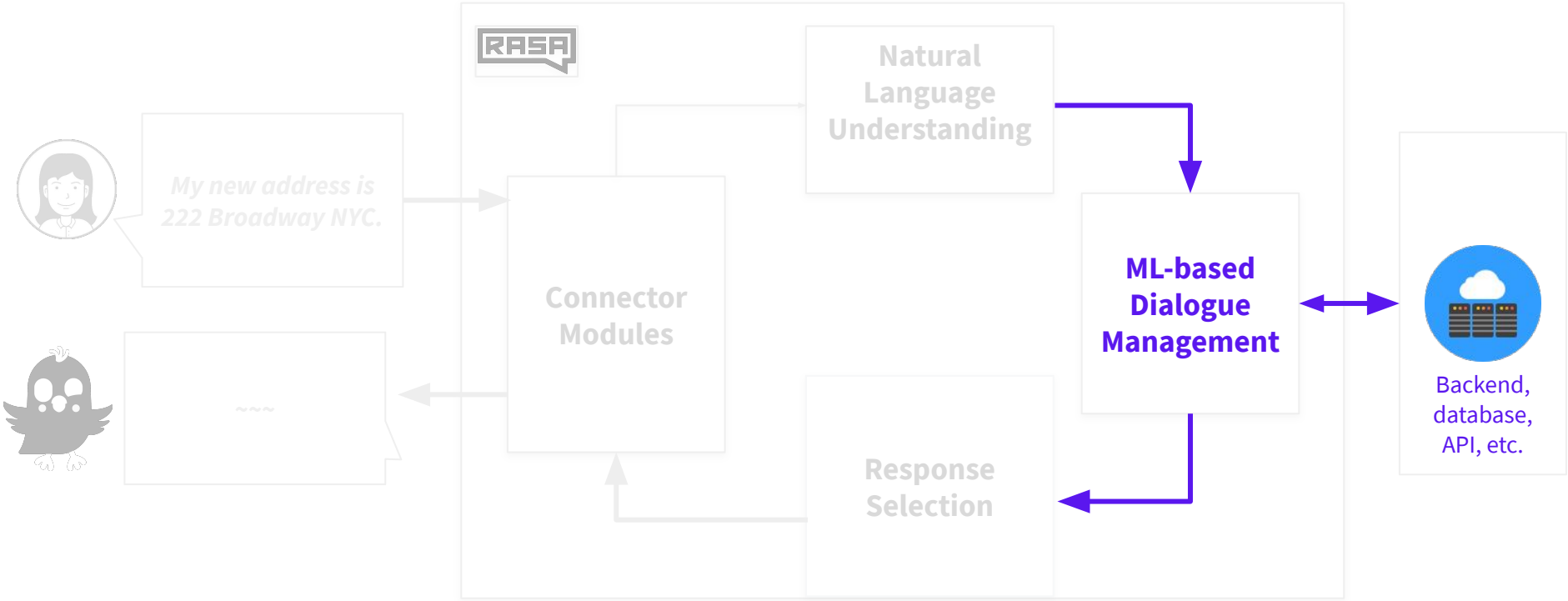


NLU: Output to Dialogue Management



Core Deep Dive

Dialogue Management (Core)



Training Data

Training Data for NLU and Dialogue Management

Machine learning models require training data that the models can generalize from

NLU needs data in the form of examples for intents

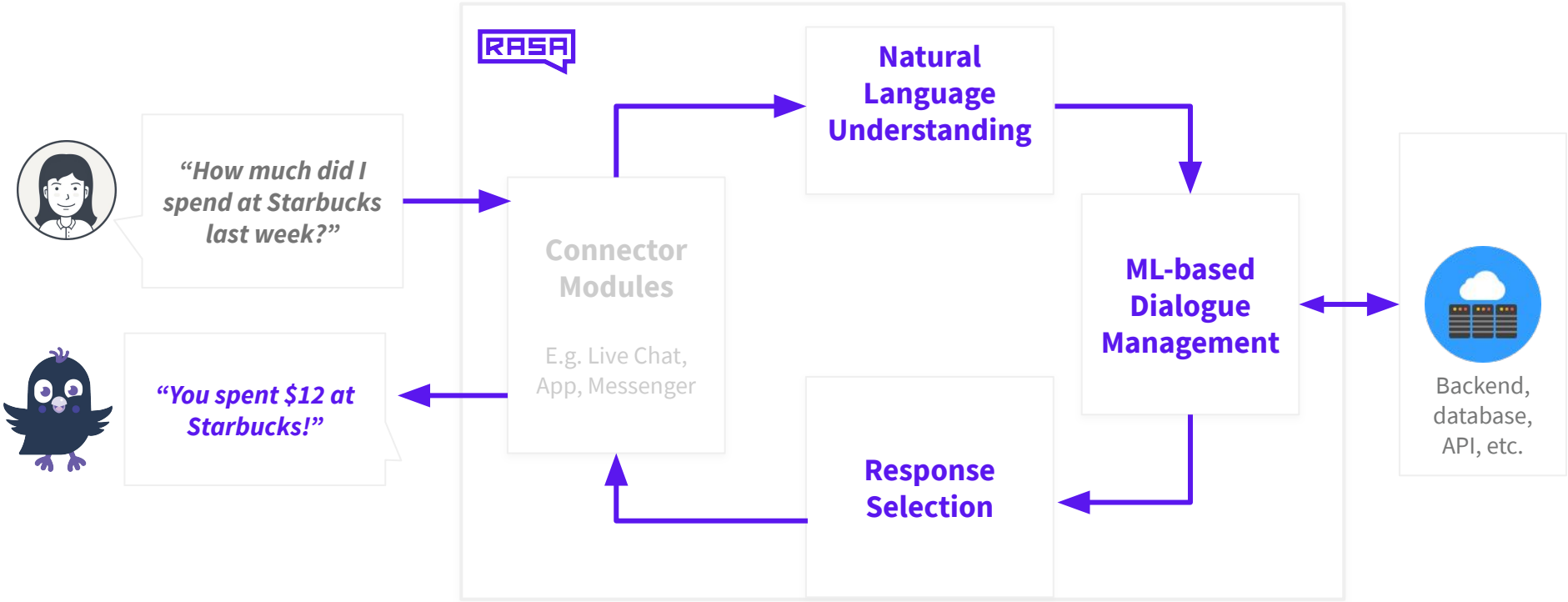
```
## intent:bot_challenge
- are you a bot?
- are you a human?
- am I talking to a bot?
- am I talking to a human?

## intent:i_like_food
- I like [apples](food)
- my friend likes [oranges](food)
- I'm a fan of [pears](food)
- do you like [coffee](food)
```

Dialogue management model needs data in the form of stories

```
## new to rasa at start
* how_to_get_started{"user_type":
"new"}
    - action_set_onboarding
    - slot{"onboarding": true}
    - utter_getstarted_new
    - utter_built_bot_before
* deny
    - utter_explain_rasa_components
    - utter_rasa_components_details
    - utter_ask_explain_nlucorex
```

RASA

Rasa Open Source Reviewed

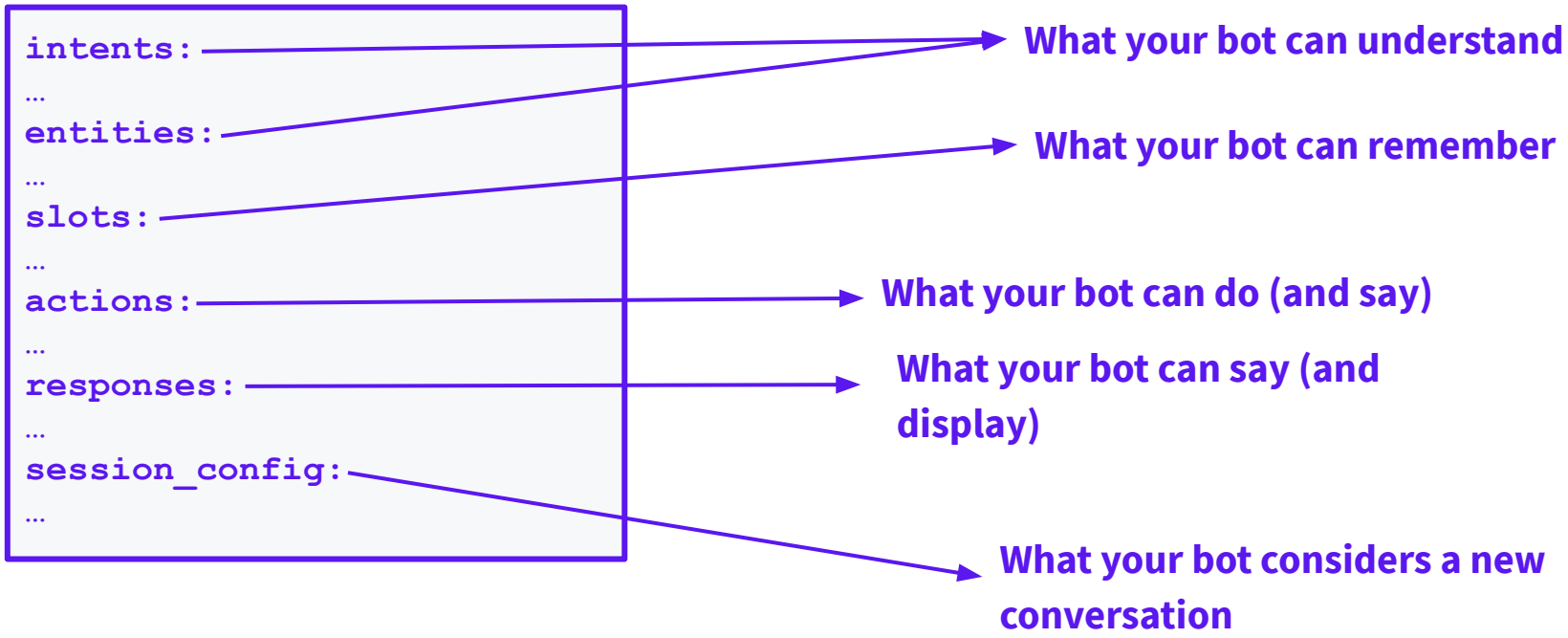


Deep dive into DIET, TED, and some testing!

Core Deep Dive

Domain

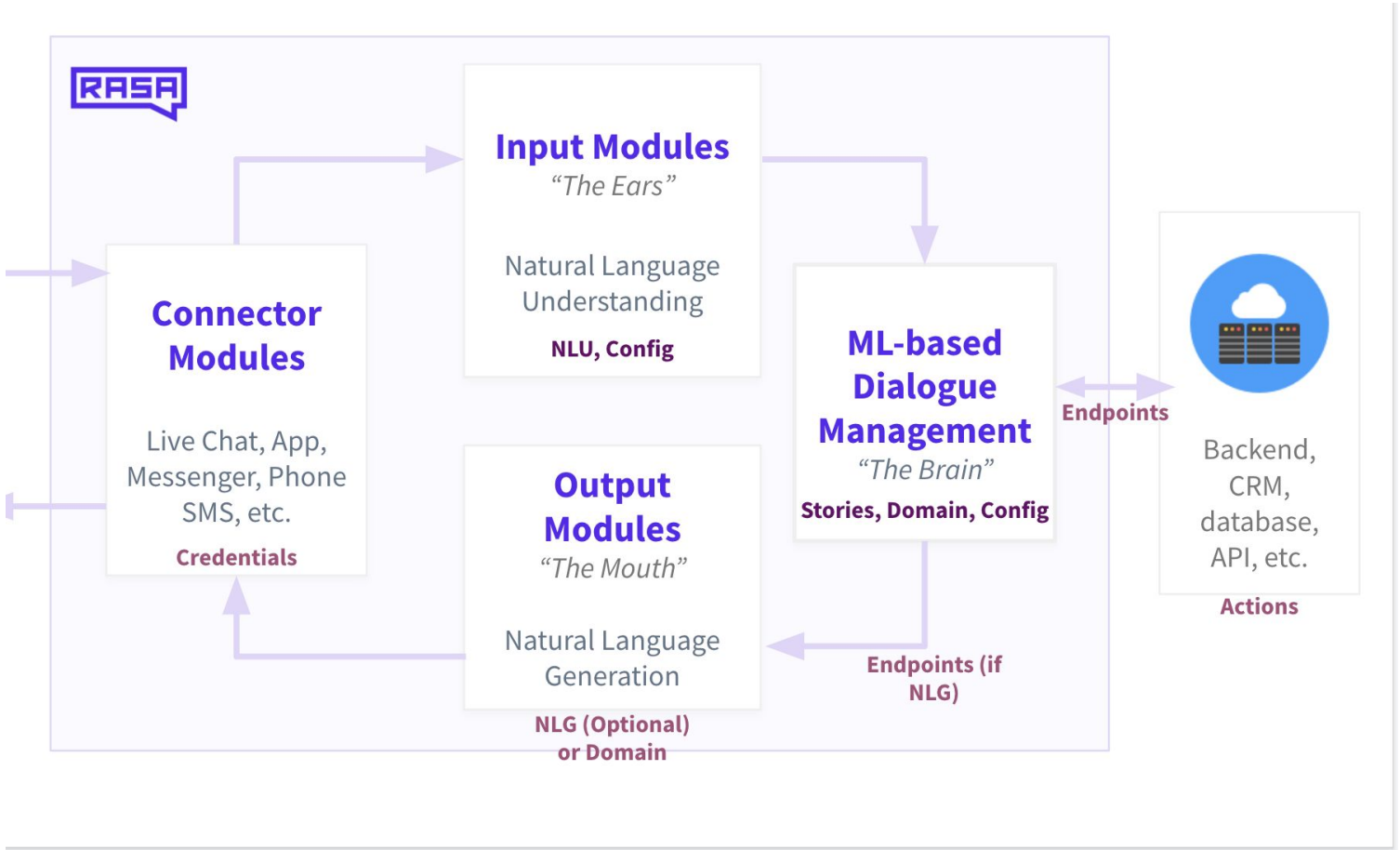
Defines the “world” of your assistant - what it knows, can understand, and can do



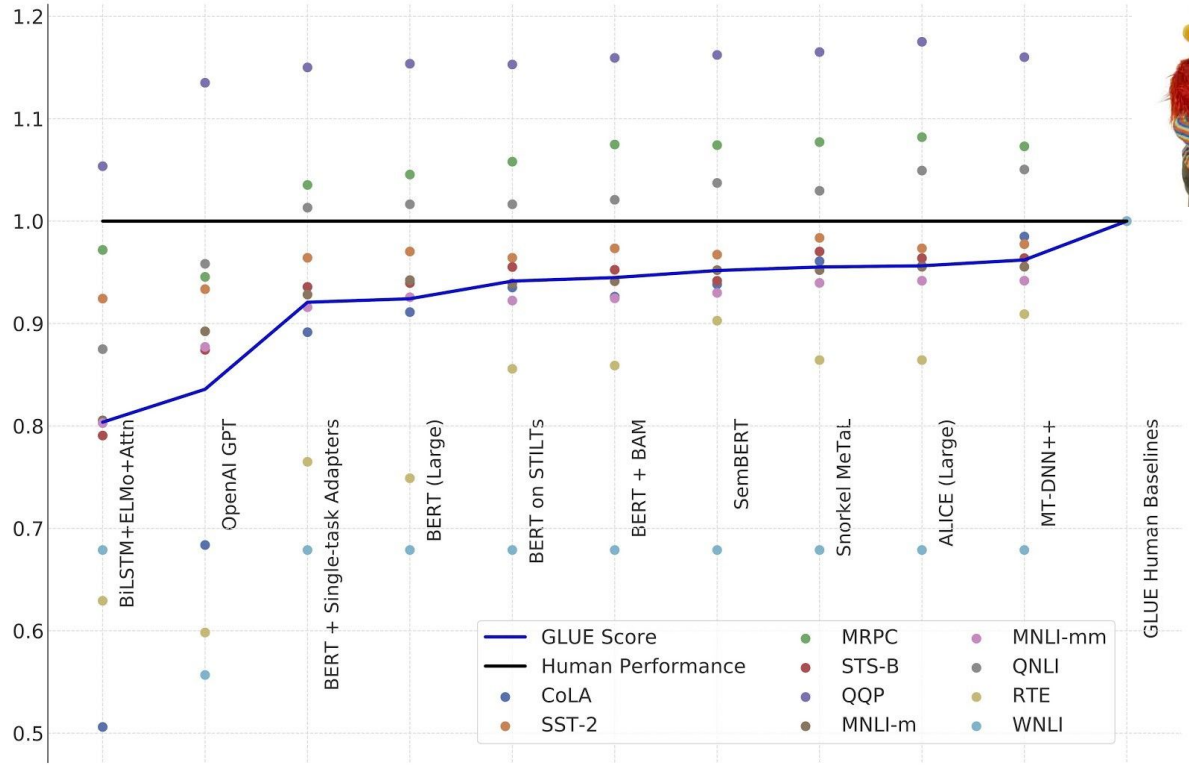
Project setup: Files

`__init__.py`	an empty file that helps python find your actions
`actions.py`	code for your custom actions
`config.yml`	configuration of your NLU and dialogue models
`credentials.yml`	details for connecting to other services
`data/`**`nlu.md`**	your NLU training data
`data/`**`stories.md`**	your stories
`domain.yml`	your assistant's domain
`endpoints.yml`	details for connecting to channels like fb messenger
`models/<timestamp>.tar.gz`	your initial model

In other words



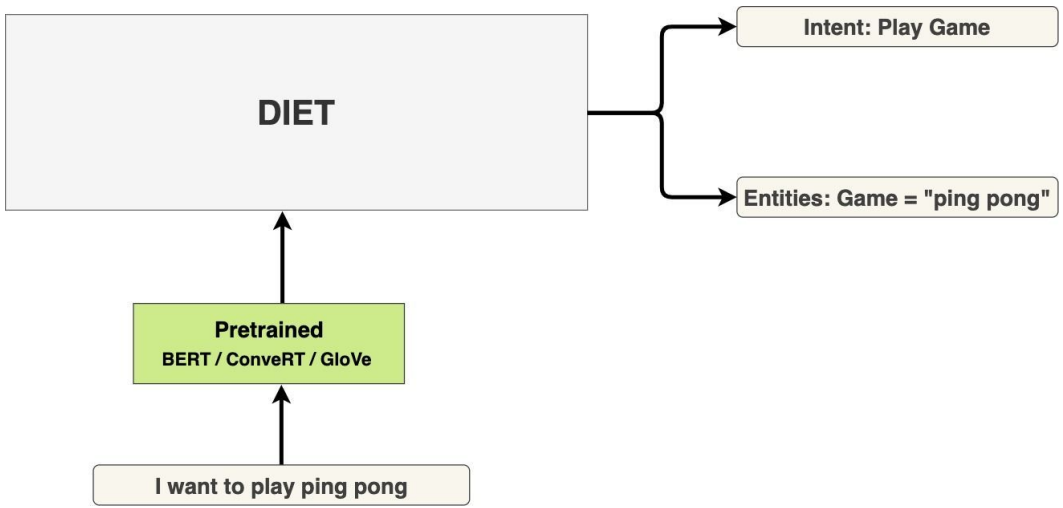
Glue is out, SuperGlue is in



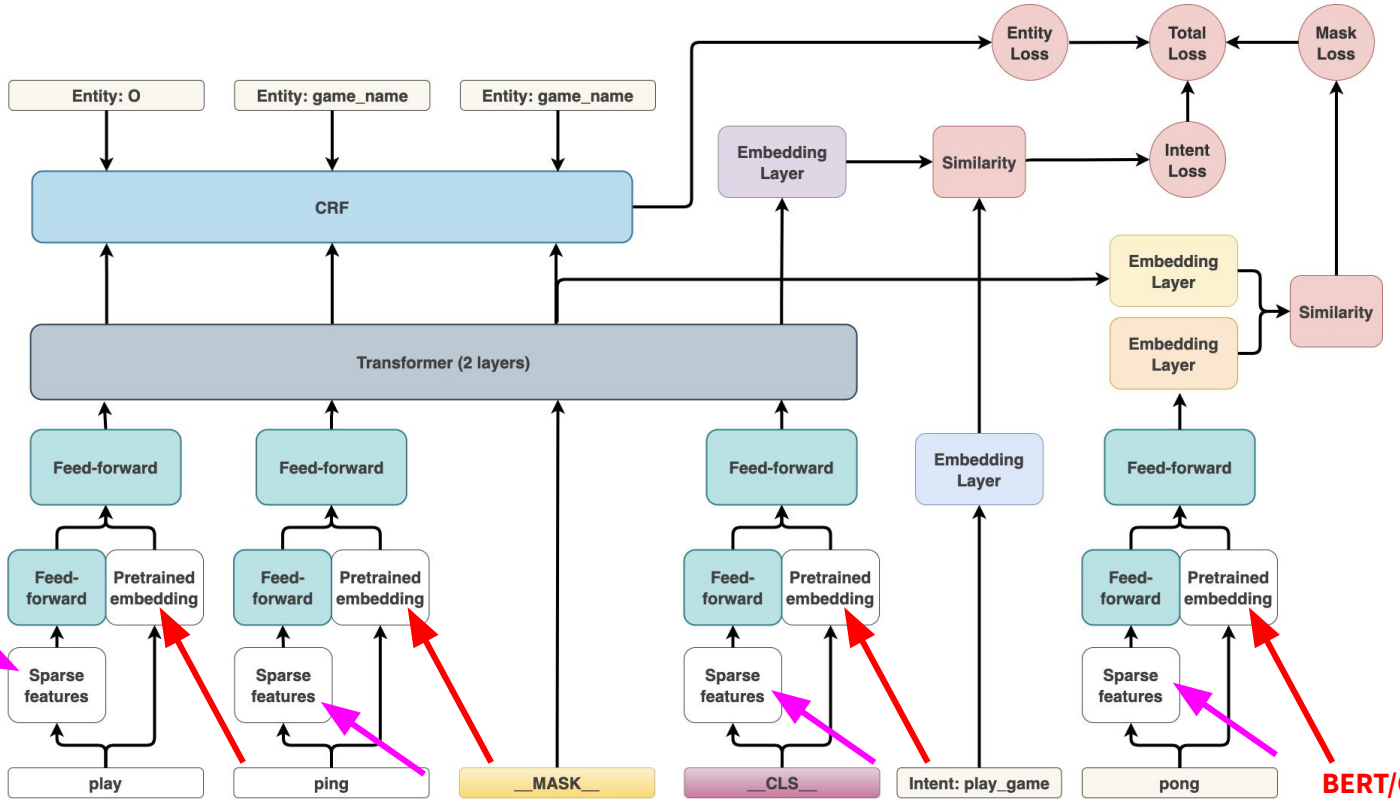
DIET is our new neural network architecture for NLU

What is DIET?

- New state of the art neural network architecture for NLU
- Predicts intents and entities together
- Plug and play pretrained language models



Feature: Customize what features go in!



sparse features in pipeline

BERT/GloVe/pretrained and continuous features

How to use DIET in your Rasa project

Here's an example config.yml

Before the DIET model, you can specify any featurizer.
In our experiments, we use:

- Sparse features (aka no pre-trained model)
- GloVe (word vectors)
- BERT (large language model)
- ConveRT (pre-trained encoder for conversations)

```yaml
language: en
pipeline:
- name: ConveRTTokenizer
- name: ConveRTFeaturizer
- name: DIETClassifier
  hidden_layers_sizes:
    text: [256, 128]
    label: []
  intent_classification: True
  entity_recognition: False
  use_masked_language_model: False
  BILOU_flag: False
  number_of_transformer_layers: 0
```

Experiments on the NLU-benchmark dataset

- Repo is on github
- Domain: human-robot interaction (smart home setting)
- 64 different intents
- 54 different entity types
- ~26k labelled examples

Previous state of the art:

- HERMIT NLU (Vanzo, Bastianelli, and Lemon @ SIGdial 2019)
- uses ELMo embeddings

Result 1: DIET outperforms SotA even without any pretrained embeddings

Previous state of the art: intent: 87.55 entities: 84.74

sparse	dense	mask loss	**Intent**	**Entities**
✓	✗	✗	87.10±0.75	83.88±0.98
✓	✗	✓	88.19±0.84	85.12±0.85
✗	GloVe	✗	89.20±0.90	84.34±1.03
✓	GloVe	✗	89.38±0.71	84.89±0.91
✗	GloVe	✓	88.78±0.70	85.06±0.84
✓	GloVe	✓	89.13±0.77	86.04±1.09
✗	BERT	✗	87.44±0.92	84.20±0.91
✓	BERT	✗	88.46±0.88	85.26±1.01
✗	BERT	✓	86.92±1.09	83.96±1.33
✓	BERT	✓	87.45±0.67	84.64±1.31
✗	ConveRT	✗	89.76±0.98	86.06±1.38
✓	ConveRT	✗	89.89±0.43	**87.38±0.64**
✗	ConveRT	✓	**90.15±0.68**	85.76±0.80
✓	ConveRT	✓	89.47±0.74	86.04±1.29

Result 2: GloVe embeddings perform better than BERT

sparse	dense	mask loss	**Intent**	**Entities**
✓	✗	✗	87.10±0.75	83.88±0.98
✓	✗	✓	88.19±0.84	85.12±0.85
✗	GloVe	✗	89.20±0.90	84.34±1.03
✓	GloVe	✗	89.38±0.71	84.89±0.91
✗	GloVe	✓	88.78±0.70	85.06±0.84
✓	GloVe	✓	89.13±0.77	86.04±1.09
✗	BERT	✗	87.44±0.92	84.20±0.91
✓	BERT	✗	88.46±0.88	85.26±1.01
✗	BERT	✓	86.92±1.09	83.96±1.33
✓	BERT	✓	87.45±0.67	84.64±1.31
✗	ConveRT	✗	89.76±0.98	86.06±1.38
✓	ConveRT	✗	89.89±0.43	**87.38±0.64**
✗	ConveRT	✓	**90.15±0.68**	85.76±0.80
✓	ConveRT	✓	89.47±0.74	86.04±1.29

Result 3: ConveRT embeddings perform best on the NLU-benchmark dataset

sparse	dense	mask loss	**Intent**	**Entities**
✓	✗	✗	87.10±0.75	83.88±0.98
✓	✗	✓	88.19±0.84	85.12±0.85
✗	GloVe	✗	89.20±0.90	84.34±1.03
✓	GloVe	✗	89.38±0.71	84.89±0.91
✗	GloVe	✓	88.78±0.70	85.06±0.84
✓	GloVe	✓	89.13±0.77	86.04±1.09
✗	BERT	✗	87.44±0.92	84.20±0.91
✓	BERT	✗	88.46±0.88	85.26±1.01
✗	BERT	✓	86.92±1.09	83.96±1.33
✓	BERT	✓	87.45±0.67	84.64±1.31
✗	ConveRT	✗	89.76±0.98	86.06±1.38
✓	ConveRT	✗	89.89±0.43	**87.38±0.64**
✗	ConveRT	✓	**90.15±0.68**	85.76±0.80
✓	ConveRT	✓	89.47±0.74	86.04±1.29

Result 4: DIET outperforms fine-tuning BERT

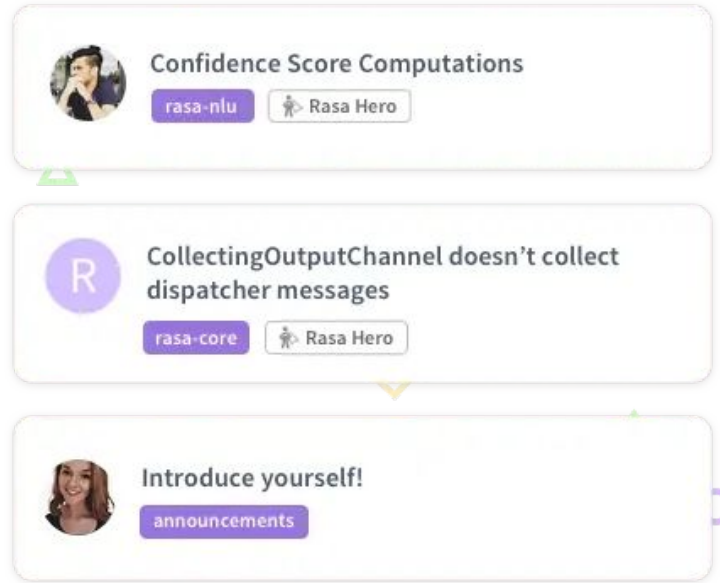
		Intent	Entities
Fine-tuned BERT	F1	89.67±0.48	85.73±0.91
	R	89.67±0.48	84.71±1.28
	P	89.67±0.48	86.78±1.02
sparse + ConveRT†	F1	**89.89±0.43**	**87.38±0.64**
	R	**89.89±0.43**	**87.15±0.97**
	P	**89.89±0.43**	**87.62±0.94**

Which featurizer is best depends on your dataset, so try different ones!

We don't believe in "one size fits all" machine learning

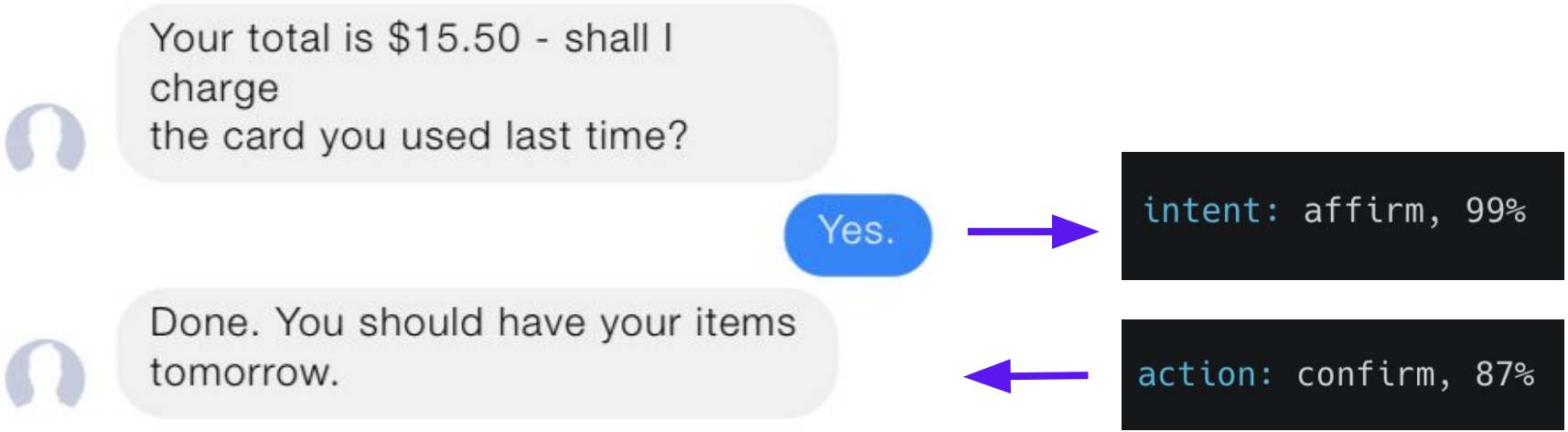
- We aim to provide sensible defaults and suggestions
- BUT even more important that these models are easy to customize

Share your results and compare notes with 8000+ developers at **forum.rasa.com**



Transformer Embedding Dialogue policy (TED)

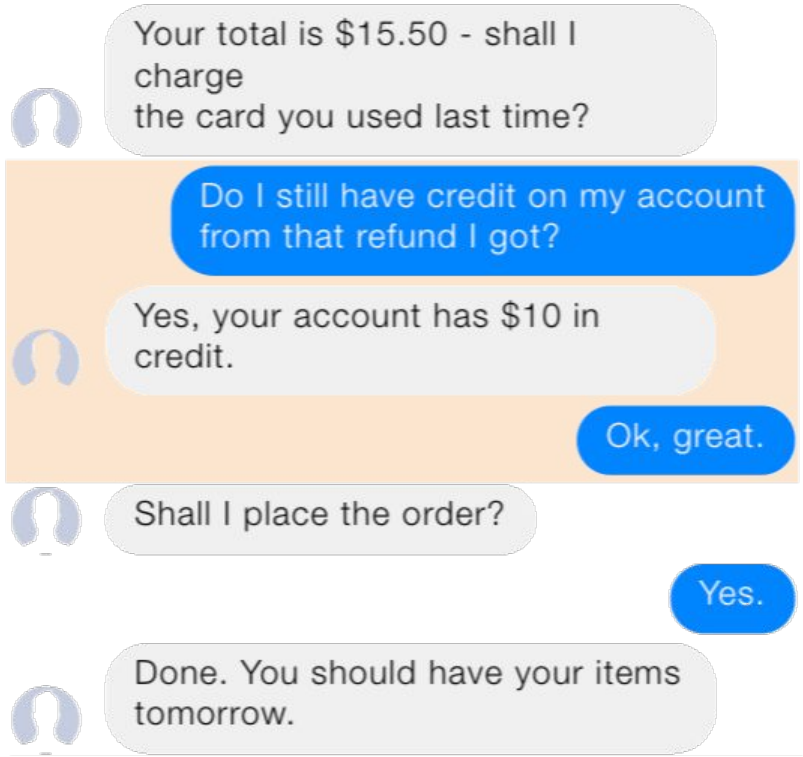
Conversational AI requires NLU and Dialogue management



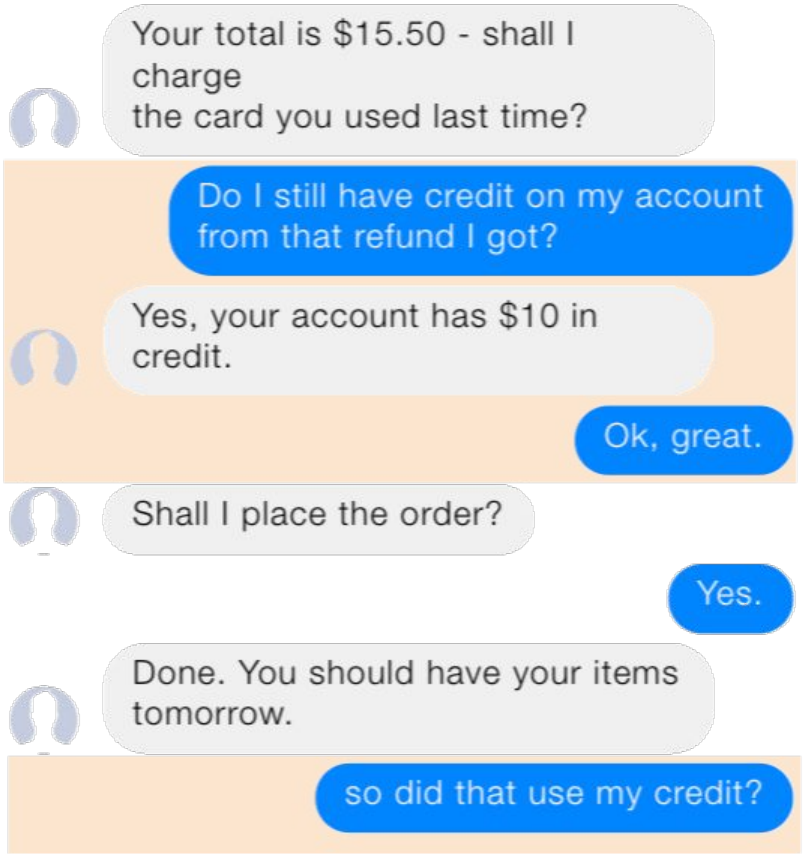
Happy paths are already solved

```python
class CheckoutForm(FormAction):
    """Perform steps for checkout"""

    def name(self):
        return "checkout_form"

    @staticmethod
    def required_slots(tracker):
        return ["address", "card_number", "shipping_method"]
```

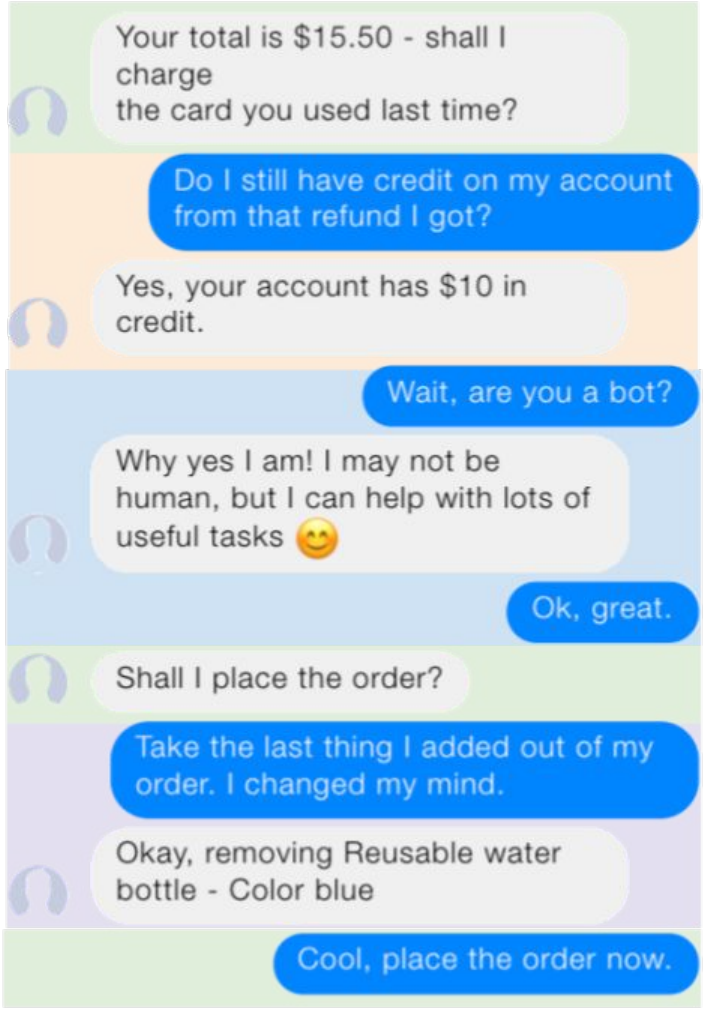
You can’t predict user behavior



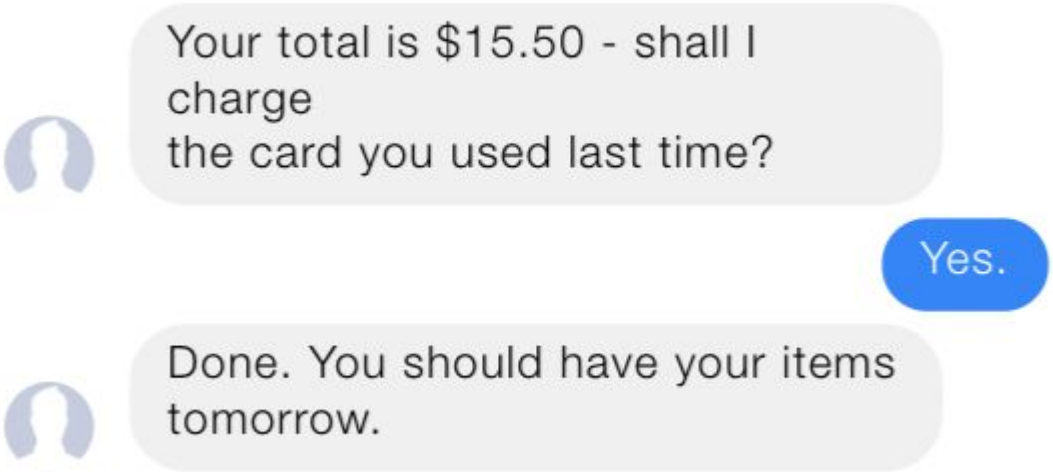
Users will interject and loop back to earlier topics

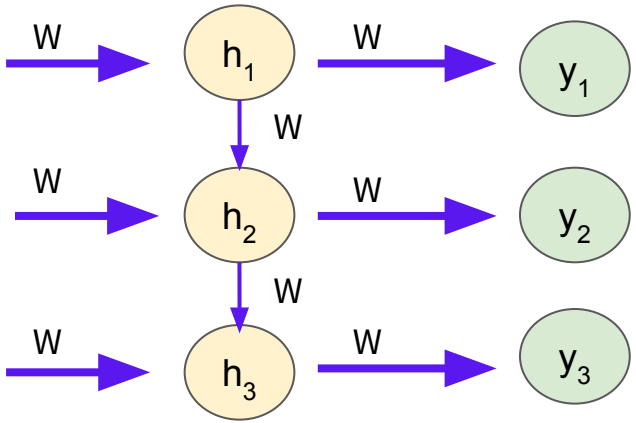


Real conversations don’t follow the happy path

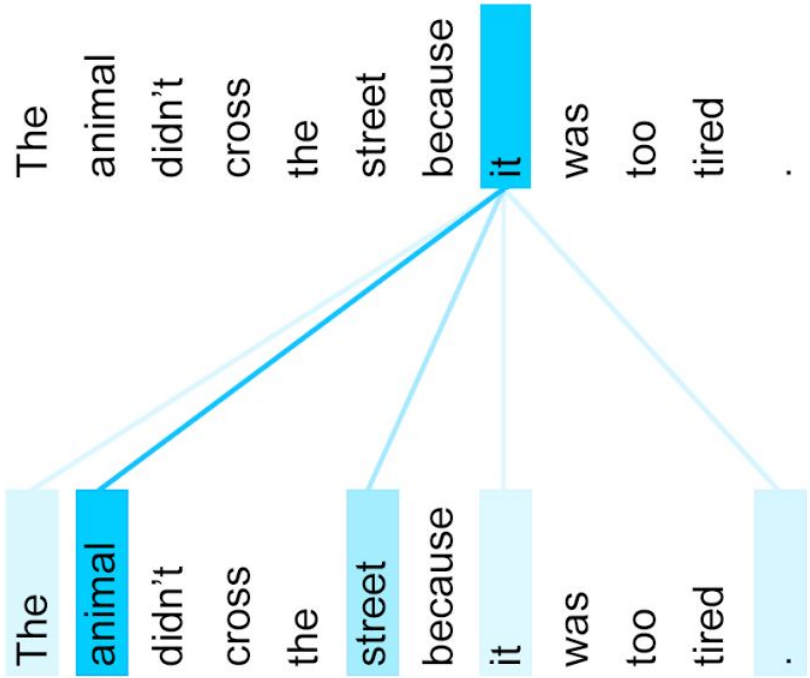


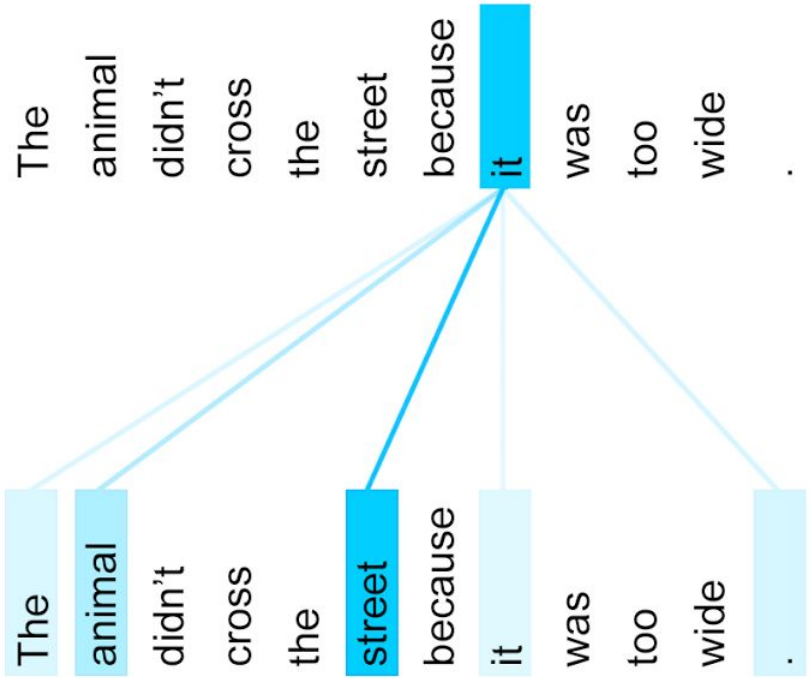
People typically use a recurrent neural net (RNN) to model dialogue





But not all input should be treated equally





https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Transformers (AKA self-attention) are now state of the art for many tasks

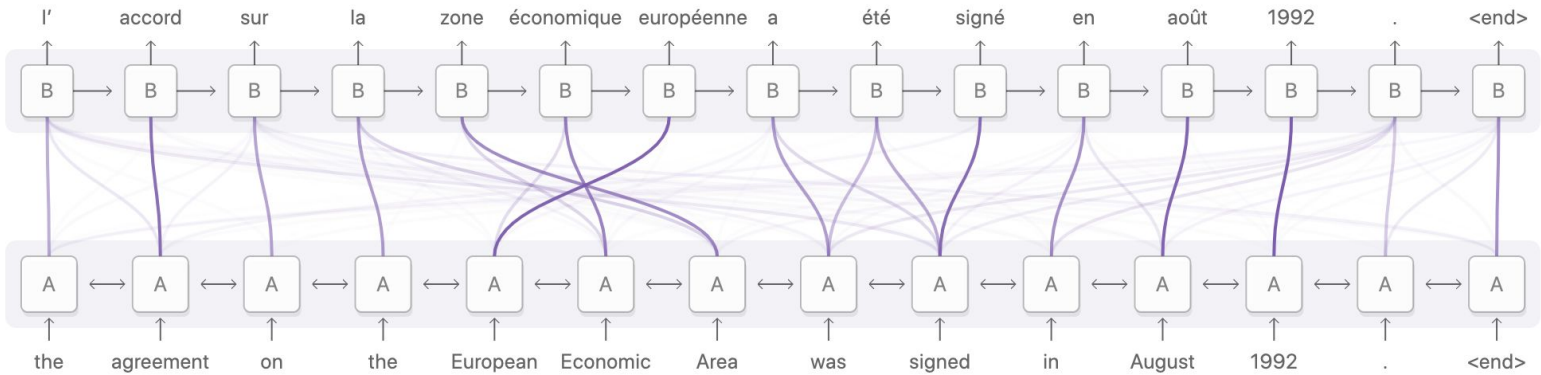
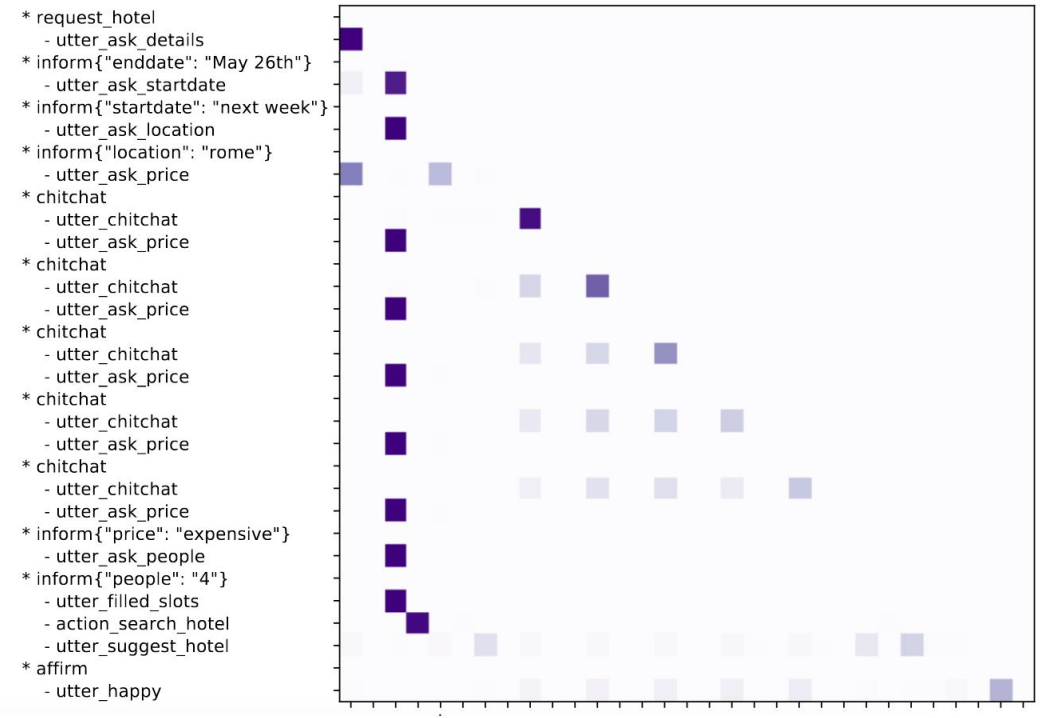


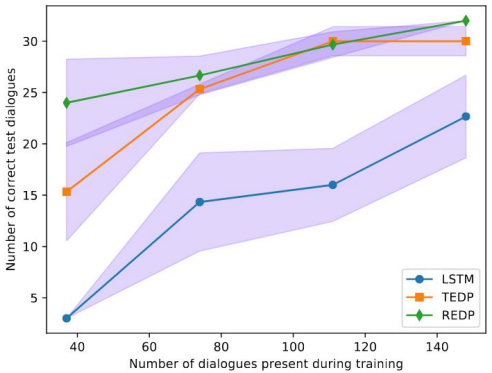
Diagram derived from Fig. 3 of Bahdanau, *et al.* 2014

https://distill.pub/2016/augmented-rnns/

We found out that the Transformer Embedding Dialogue policy can untangle sub-dialogues



paper



Conversation Driven Development (CDD)

The challenge

When developing assistants, it's impossible to anticipate all of the things your users might say.

The approach

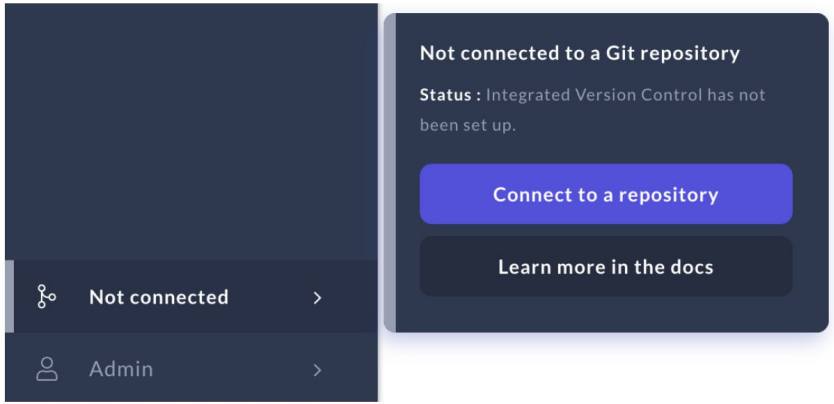
A user-centric process: listening to your users and using those insights to improve your AI assistant.

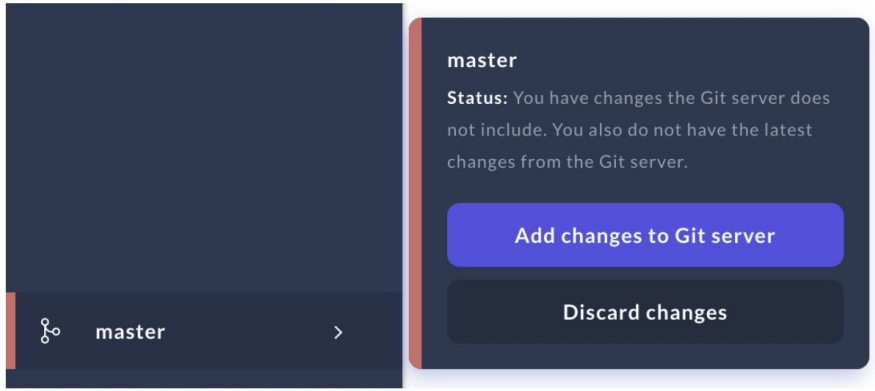
Blog post: Conversation-Driven Development

Conversation Driven Development

Rasa X

Successful production-ready bots follow engineering best practices





TIME TO VALUE

From zero to production-ready deployment in under 5 minutes

1. Create a VM
2. curl -s http://get-rasa-x.rasa.com | sudo bash

script takes ~4 mins and you have a full containerised deployment of rasa open source and rasa x 🚀https://lnkd.in/dsS-s9h
What it does:
- installs k3s
- installs helm
- installs a helm chart with everything

THE APPROACH

Continually improve your assistant using Rasa X

Ensure your new assistant passes tests using **continuous integration (CI)** and redeploy it to users using **continuous deployment (CD)**

Collect conversations between users and your assistant

Review conversations and **improve your assistant** based on what you learn

Testing

Testing

Use the Rasa CLI to test your assistant

End to End Evaluation

Run through test conversations to make sure that both NLU and Core make correct predictions.

```
$ rasa test
```

NLU Evaluation

Split data into a test set or estimate how well your model generalizes using cross-validation.

```
$ rasa test nlu -u
data/nlu.md --config
config.yml
--cross-validation
```

Core Evaluation

Evaluate your trained model on a set of test stories and generate a confusion matrix.

```
$ rasa test core
--stories
test_stories.md --out
results
```

Interactive Learning: Talk to your bot yourself

- **Correct** your bot's predictions as you go

- Save conversations as **training stories** or **E2E stories**

Run `rasa test` locally when building a minimum viable assistant

- Test your model after training to make development more productive and reliable

```
#### This file contains tests to evaluate that your bot behaves as expected.
#### If you want to learn more, please see the docs: https://rasa.com/docs/ra

## happy path 1
* greet: hello there!
  - utter_greet
* mood_great: amazing
  - utter_happy

## happy path 2
* greet: hello there!
  - utter_greet
* mood_great: amazing
  - utter_happy
* goodbye: bye-bye!
  - utter_goodbye

## sad path 1
* greet: hello
  - utter_greet
* mood_unhappy: not good
  - utter_cheer_up
  - utter_did_that_help
* affirm: yes
  - utter_happy
```

```
Your Rasa model is trained and saved at '/Users/ty/Documents/product_mgmt/docs/rasa/tem
(rasa_env) BERMB00017:temp ty$ rasa test
Processed Story Blocks: 100%|
2020-03-24 14:24:31 INFO     rasa.core.test  - Evaluating 7 stories
Progress:
100%|
2020-03-24 14:24:32 INFO     rasa.core.test  - Finished collecting predictions.
2020-03-24 14:24:32 INFO     rasa.core.test  - Evaluation Results on END-TO-END level:
2020-03-24 14:24:32 INFO     rasa.core.test  -  Correct:          7 / 7
2020-03-24 14:24:32 INFO     rasa.core.test  -  F1-Score:         1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  Precision:        1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  Accuracy:         1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  In-data fraction: 0.943
2020-03-24 14:24:32 INFO     rasa.core.test  - Evaluation Results on ACTION level:
2020-03-24 14:24:32 INFO     rasa.core.test  -  Correct:          35 / 35
2020-03-24 14:24:32 INFO     rasa.core.test  -  F1-Score:         1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  Precision:        1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  Accuracy:         1.000
2020-03-24 14:24:32 INFO     rasa.core.test  -  In-data fraction: 0.943
2020-03-24 14:24:32 INFO     rasa.core.test  -  Classification report:
                    precision    recall  f1-score   support
```

Adding custom actions and implementing forms

RASA CERTIFICATION WORKSHOP

Agenda

Day 1: Welcome and deep dive into Rasa

- Welcome! 👋
- Intro to Rasa
- Conversation Design
- Deep dive into NLU
- Deep dive into dialogue policies
- CDD and testing
- Demo

Day 2: Adding custom actions, forms and improving your assistant using Rasa X

- Events and Slots
- Custom actions and Forms
- Improving the assistant using Rasa X
- Connecting the assistant to the external messaging channels
- What's next

Day 2: Roadmap for the first half of the day

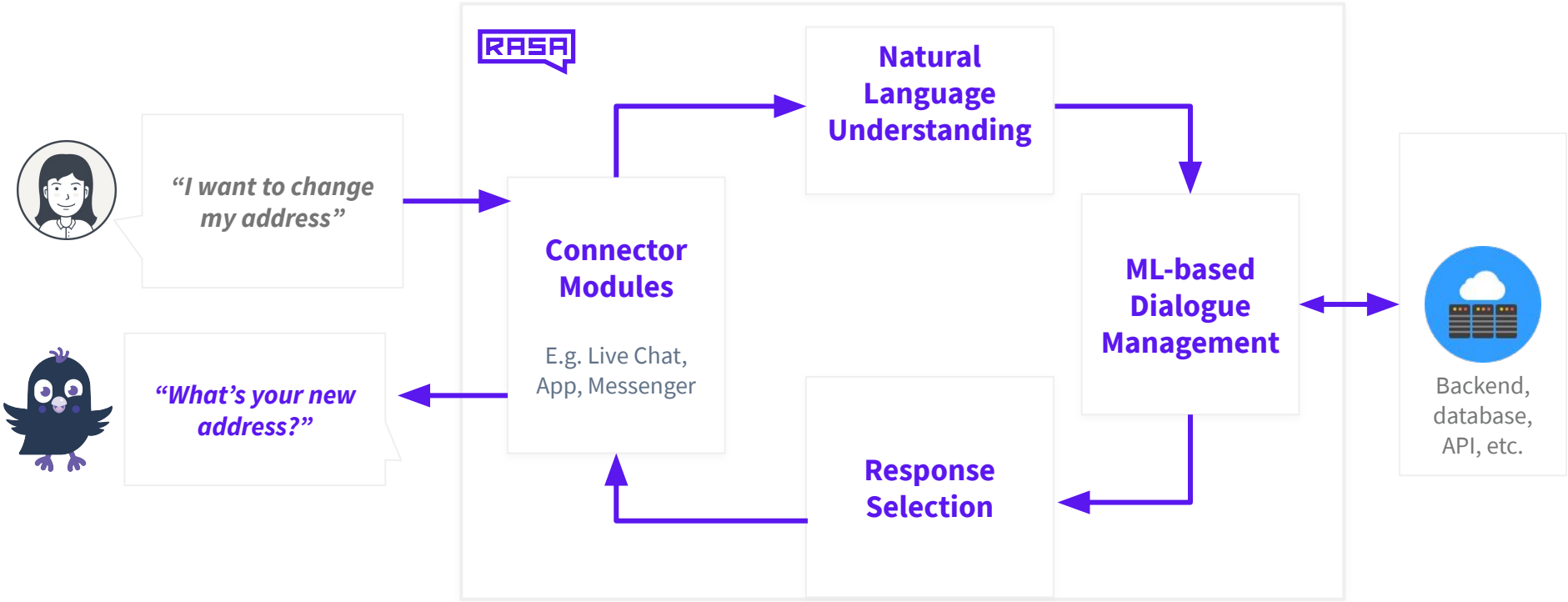
Part 1:

- Rasa events and slots: talk and coding together
- Custom actions: talk and coding together
- Short break 🧁

Part 2:

- Forms: talk and coding together
- Recap and Q&A

Rasa Open Source



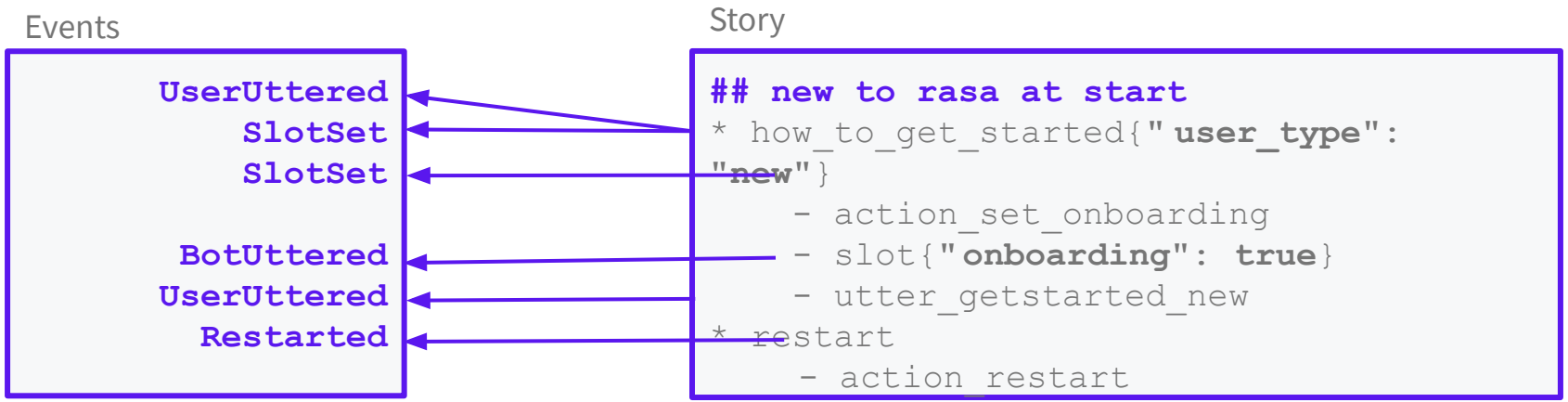
Project setup: Files

`__init__.py`	an empty file that helps python find your actions
`actions.py`	code for your custom actions
`config.yml`	configuration of your NLU and Core models
`credentials.yml`	details for connecting to other services
`data/`**`nlu.md`**	your NLU training data
`data/`**`stories.md`**	your stories
`domain.yml`	your assistant's domain
`endpoints.yml`	details for connecting to channels like fb messenger
`models/<timestamp>.tar.gz`	your initial model

Rasa events and slots

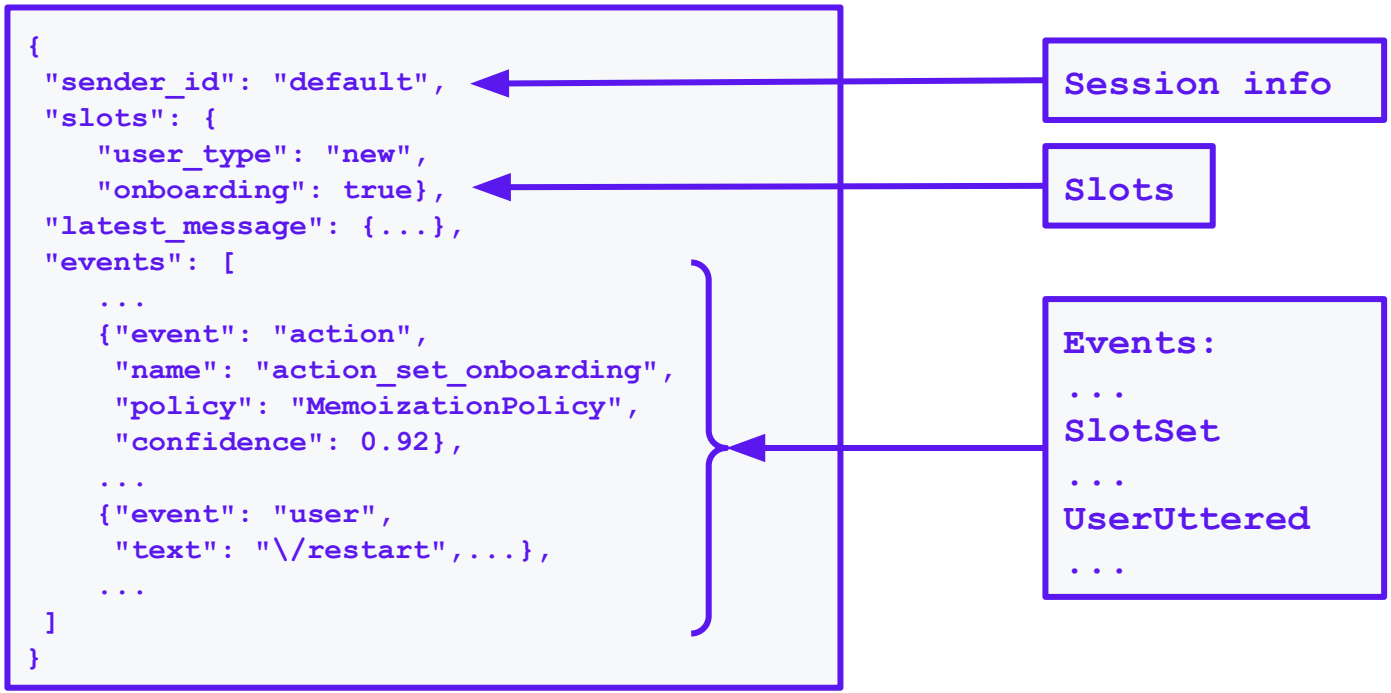
Events

- **Internally**, conversations are represented as a sequence of **events.**
- Some events are automatically tracked



Tracker

- Trackers maintain the **state of a dialogue** between the assistant and the user.
- It keeps track of events, slots, session info etc.



Core Deep Dive

Slots

Your bot's memory

Hi, my name is Tom

Hello! How can I help?

I would like to check my account balance

Sure. Can you tell me your surname?

Hi

Hello! How can I help?

I would like to check my account balance

What is your name?

Slots

Your bot's memory

- Can store:
  - user-provided info
  - info from the outside world
- Can be set by:
  - NLU (from extracted entities, or buttons)
  - Custom Actions
- Can be configured to affect or not affect the dialogue progression

Core Deep Dive

Slots types

Text and List slots influence conversation path based on **whether it is set or not.**

Text Slot

`text`

Use For: User preferences where you only care whether or not they've been specified.

Example:

```
slots:
   location:
      type: text
```

"I am based in London"

List Slot

`list`

Use For: Lists of values

Example:

```
slots:
   appointment_times:
      type: list
```

"Are there available appointments on Monday or Wednesday this week?"

Core Deep Dive

Slots types

Categorical, Boolean and Float slots influence conversation path based on **the value of the slot.**

Categorical Slot

```
categorical
```

Use For: Slots which can take one of N values

Example:

```
slots:
   price_level:
      type: categorical
      values:
      - low
      - medium
      - high
```

"I am looking for a restaurant in a low price range."

Boolean Slot

```
bool
```

Use For: True or False

Example:

```
slots:
   is_authenticated:
      type: bool
```

Float Slot

```
float
```

Use For: Continuous values

Example:

```
radius:
   type: float
   min_value:  0.0
   max_value:  50.0
```

"Find me a restaurant within 2 mile radius."

Core Deep Dive

Slots types

Unfeaturized slots don’t have any influence on the dialogue.

Unfeaturized Slot

```
unfeaturized
```

Use For: Continuous values

Example:

```
slots:
   patient_name:
      type: unfeaturized
```

“My name is Sarah.”

Slots

There are a few different ways how slots can be set

- Slots set from NLU

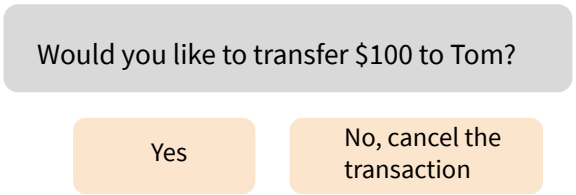
```
## intent:search_transactions
- how much did I spend at [Target](vendor_name) this week?
- what is my typical spending at [Amazon](vendor_name)?
```

Slots

There are a few different ways slots can be set

- Slots set by clicking buttons

```
utter_ask_transfer_form_confirm:
 -   text: "Would you like to transfer $100 to Tom?"
     buttons:
       -   title: Yes
           payload: /affirm
       -   title: No, cancel the transaction
           payload: /deny
```



Slots

There are a few different ways how slots can be set

- Slots set by custom actions

```
## account balance story
* check_account_balance
    - action_account_balance
    - slot{"account_balance": 1000}
    - slot{"amount_transferred": None}
```

Livecoding

Let’s define the slots for our Financial Bot assistant!

1. Let’s use a sandbox branch for today’s exercise. First, make sure your are in a financial bot directory:
   cd financial-demo

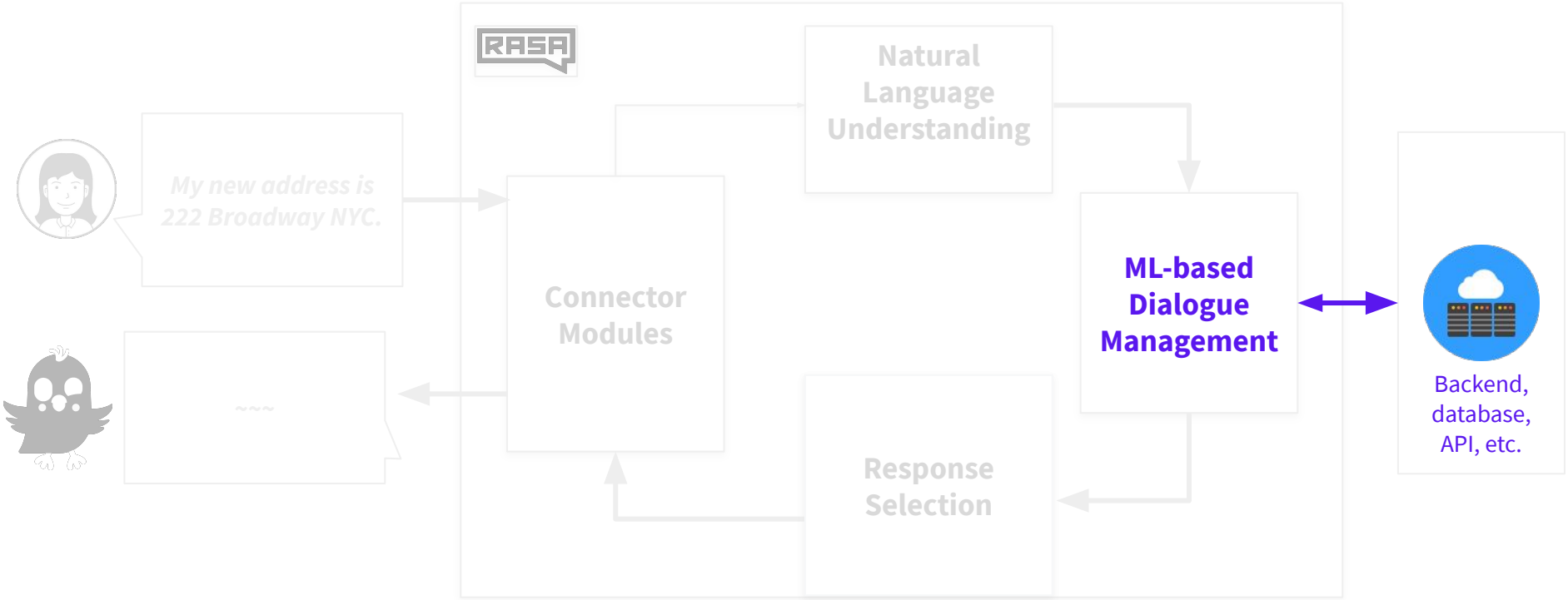
2. Pull the latest updates pushed to the repo:
   git pull

3. Checkout the sandbox branch named “workshop_day_2”:
   git checkout workshop_day_2

Custom actions

Core Deep Dive

Custom Actions: Connecting to the outside world



Actions

Functions your bot runs in response to user input.

Four different action types:

- **Utterance actions**: `utter_`
  - send a specific message to the user
  - specified in `responses` section of the domain
- **Retrieval actions**: `respond_`
  - send a message selected by a retrieval model
- **Custom actions**: `action_`
  - run arbitrary code and send any number of messages (or none).
  - return events
- **Default actions**:
  - built-in implementations available but can be overridden
  - E.g. action_listen, action_restart, action_default_fallback

Custom Actions: The Action Server

Custom action code is run by a **webserver** called the **action server**

- **How custom actions get run:**
  - When a custom action is predicted, Rasa will call the **endpoint** to your action server
  - The action server:
    - runs the code for your custom action
    - [optionally] returns information to modify the dialogue state
- **How to create an action server:**
  - You can create an action server in any language you want!
  - You can use the **Rasa-SDK** for easy development in Python

Custom Actions: Examples

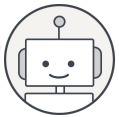
Custom actions can do whatever you want them to! e.g.

- **Send multiple messages**
- **Query a database**
- **Make an API call to another service**
- **Return events e.g.**
  - Set a slot
    - e.g. based on a database query)
  - Force a follow up action
    - force a specific action to be executed next
  - Revert a user utterance
    - I.e. remove a user utterance from the tracker

Custom Action Example: Query a Database

using the Rasa SDK

"Should I also update your mailing address?"



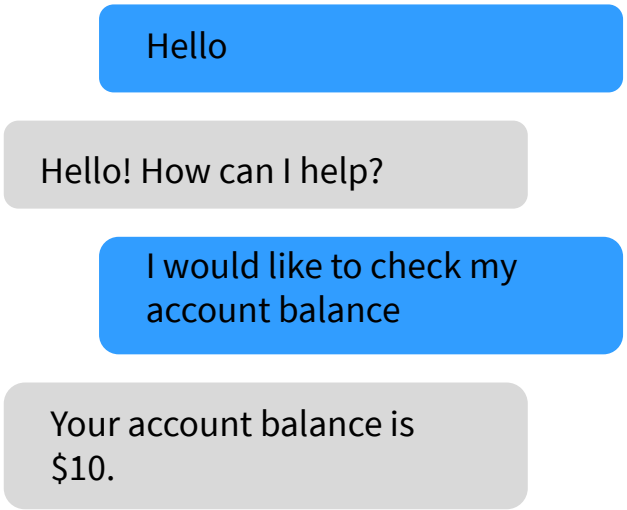
"What do you have on file?"

```
class ActionCheckAddress(Action):
   def name(self) -> Text:
      return "action_check_address"

   def run(self,
           dispatcher: CollectingDispatcher,
           tracker: Tracker,
           domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

      idnum = tracker.get_slot('person_id')
      q = "SELECT Address FROM Customers \
          \WHERE CustomerID=1;'{0}'".format(idnum)
      result = db.query(q)

      return [SlotSet(
          "address",
          result if result is not None else "NotOnFile")]
```

Let's Code

Let's implement a custom for our financial assistant

We will implement a custom action which will allow users to check their account balance:



Forms

Forms

Rasa Forms allow you to describe all happy paths with a single story

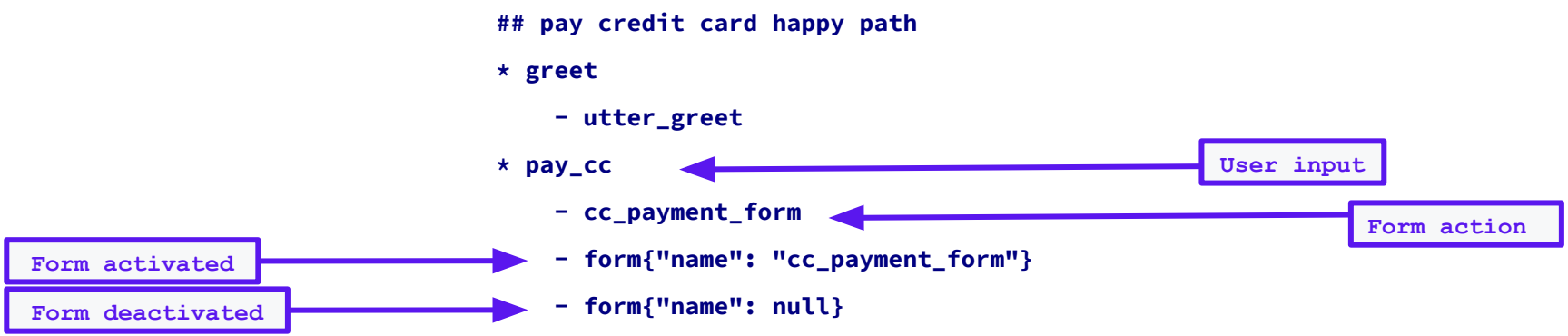
Happy path 1:
U: I would like to make a money transfer
B: Towards which credit card would you like to make a payment?
U: Towards my justice bank credit card
B: How much do you want to pay?
U: $100
B: The transaction has been scheduled.

Happy path 2:
U: I would like to make a 100$ money transfer
B: Towards which credit card would you like to make a payment?
U: Towards my justice bank credit card
B: The transaction has been scheduled.

```
## pay credit card happy path
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
    - form{"name": null}
```

Forms

The structure of the form



Core Deep Dive

Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- name - defines the name of the form actions

```
def name(self) -> Text:
    " " "Unique identifier of the form" " "
    return "cc_payment_form"
```

Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- required slots - sets the list of required slots

```
def required_slots(tracker: Tracker) -> List[Text]:
    " " "A list of required slots to fill in" " "
    return [“credit_card”, “amount_of_money”]
```

Core Deep Dive

Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- submit - defines the output of the form action once all slots are filled in

```
def submit(self, args) -> List[Dict]:
    " " "Defines what the form has to do after all slots are filled in" " "
    dispatcher.utter_message("The payment is confirmed")
    return [AllSlotsReset()]
```

Core Deep Dive

Forms: Custom slot mappings

The slot mappings define how to extract slot values from user inputs.

With slot mappings you can define how certain slots can be extracted from:

- Entities
- Intents
- Support free text input
- Support yes/no inputs

```
def slot_mappings(self) -> List[Text, Union[Dict, List[Dict]]]:
    " " "Defines what the form has to do after all slots are filled in" " "
    return {
        “confirm”: [
                self.from_intent(value:True, intent=”affirm”),
                self.from_intent(value:False, intent=”deny”)
                ]
            }
```

RASA

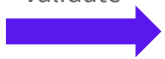
Forms: Validating user input

Validate method in form action allows you to check the value of the slot against a set of values.

I would like to make a $100 payment.

slot: amout_of_money

validate



“Minimum balance”: 85
“Current balance”: 550

Forms

Custom Form Action

Custom Form Action allows you to add new methods to a regular FormAction.

```
class CustomFormAction(FormAction):

    def name(self):
        return “”


    def custom_method(self, args):
            ...
        return
```

Forms

Handling unhappy paths

Users are not always cooperative - they can change their mind or interrupt the form.

- Create stories to handle the interruptions:

```
## pay credit card unhappy path
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
* chitchat
    - utter_chitchat
    - cc_payment_form
    - form{"name": null}
```

Forms

Handling unhappy paths

Users are not always cooperative - they can change their mind or interrupt the form.

- Use action `action_deactivate_form` to handle the situation where users decide not to proceed with the form:

```
## chitchat
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
* stop
    - utter_ask_continue
* deny
    - action_deactivate_form
    - form{"name": null}
```

Forms

How this affect domain and policy configuration?

Domain

```
forms:
  cc_payment_form
```

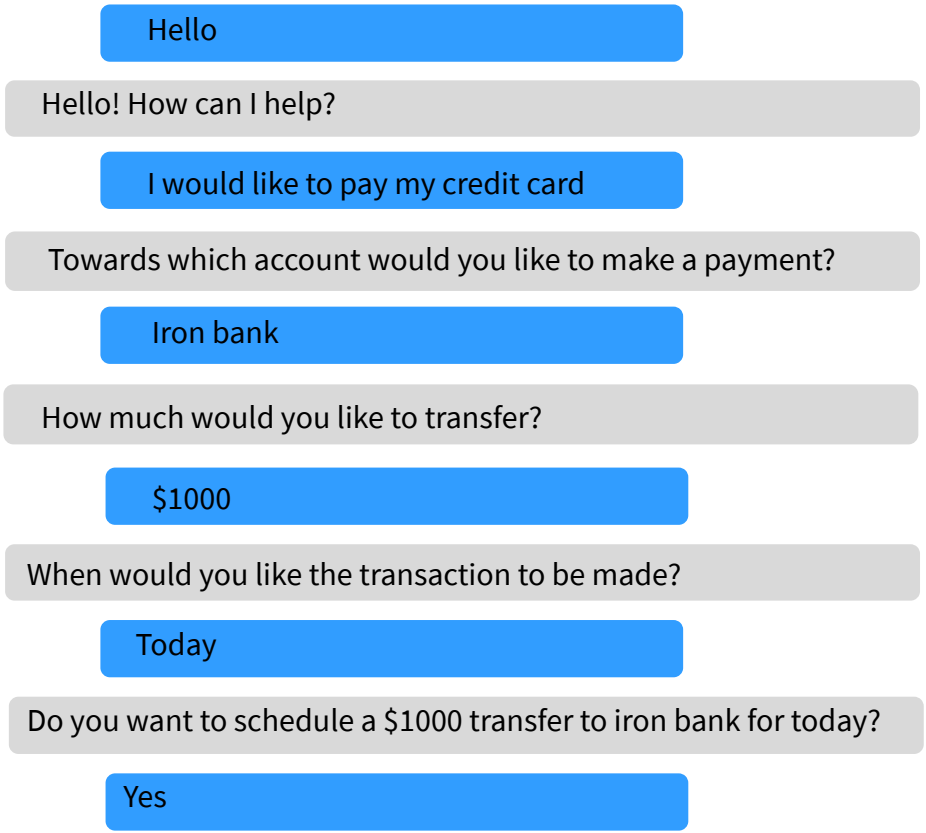
Policy configuration

```
policies:
  -   name: FormPolicy
```

Let's Code

Let's implement a for paying out a credit card

We will implement a form which will enable our assistant to collect necessary information to pay a credit card:



RASA

Part 4: Messaging Channels, Conversation Driven Development, and Rasa X

Agenda

Theme for this section: Connecting with your users

I. Conversation Driven Development
   A. Combining user input and software engineering best practices at every stage of the development cycle
   B. Conversations = training data
II. Rasa X
   A. A tool for Conversation Driven Development
   B. Launching Rasa X locally
   C. Reviewing conversations
III. Messaging Channels
   A. Live coding: connecting Telegram
IV. Review

Conversation Driven Development

CDD captures the lessons we've learned as a community

If you've built conversational AI before, you know that:

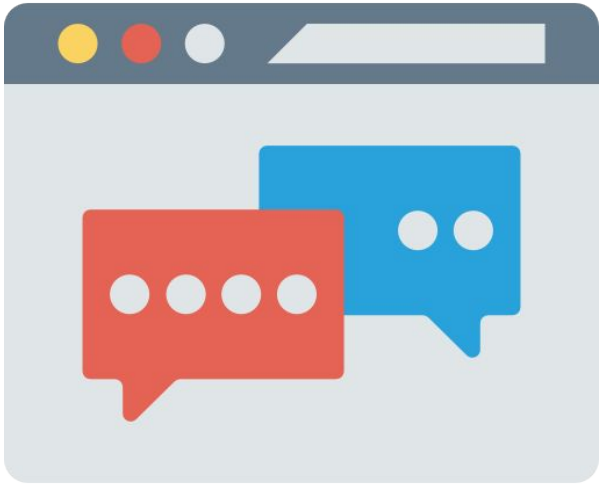
- It is very hard.
- Building a prototype is not the hard part.
- The hard parts all show up when you want to go from a prototype to something you'd want to ship.

Conversation-Driven Development should:

- Help all of us build better conversational AI.
- **Save newcomers from having to learn this the hard way**.

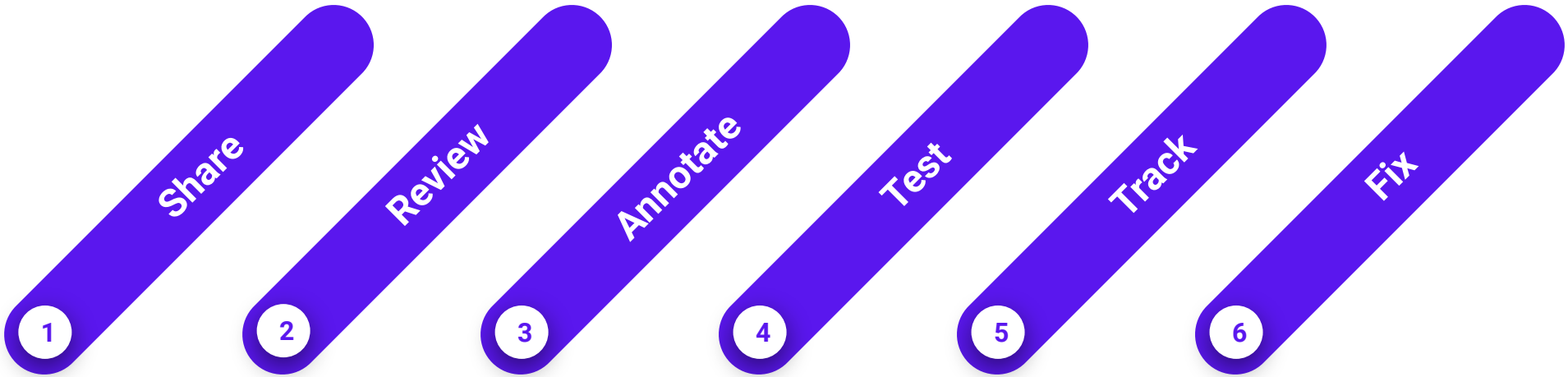
What is the opposite of conversation driven development?

- Developing your assistant for months in isolation before testing it with users
- Only looking at top-level metrics, not conversations themselves
- Autogenerating training data that doesn’t reflect ways users really interact
- Pushing changes to production without running tests



RASA

Conversation-Driven Development



Share

Users will always surprise you.

So get some test users to try your prototype **as early as possible**.

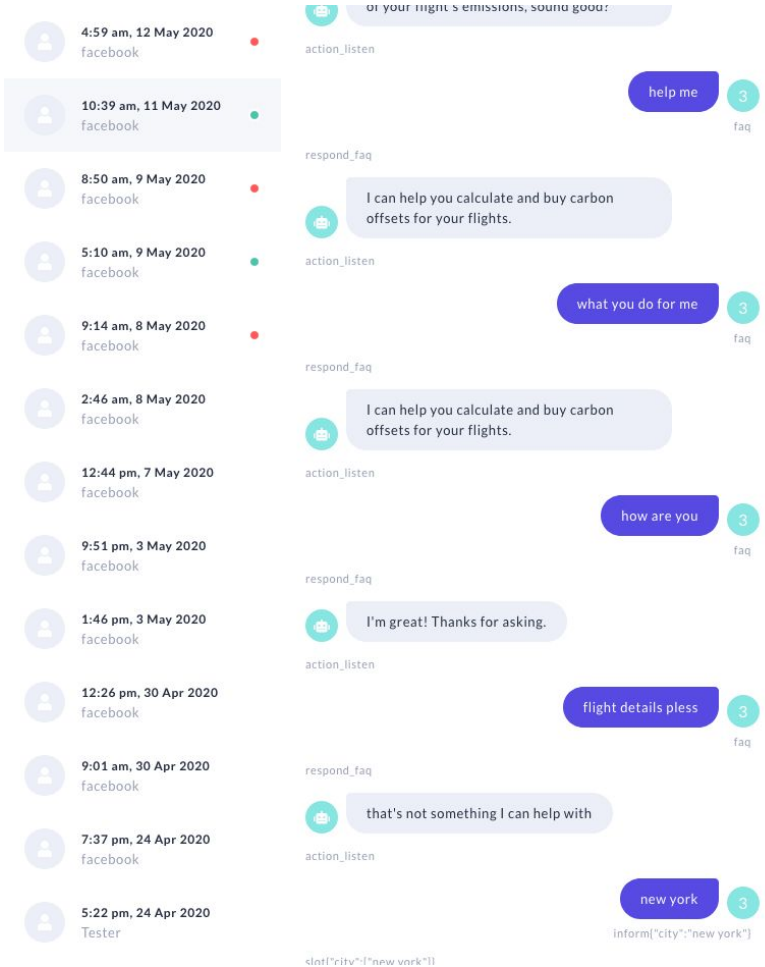
Projects that ship without a bunch of test users are bound to fall down in production.

RASA

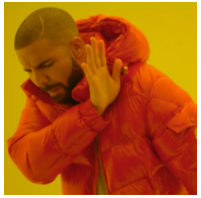
Review

At every stage of a project, it is worth reading what users are saying.

Avoid getting caught up in metrics right away. Conversations are valuable data.

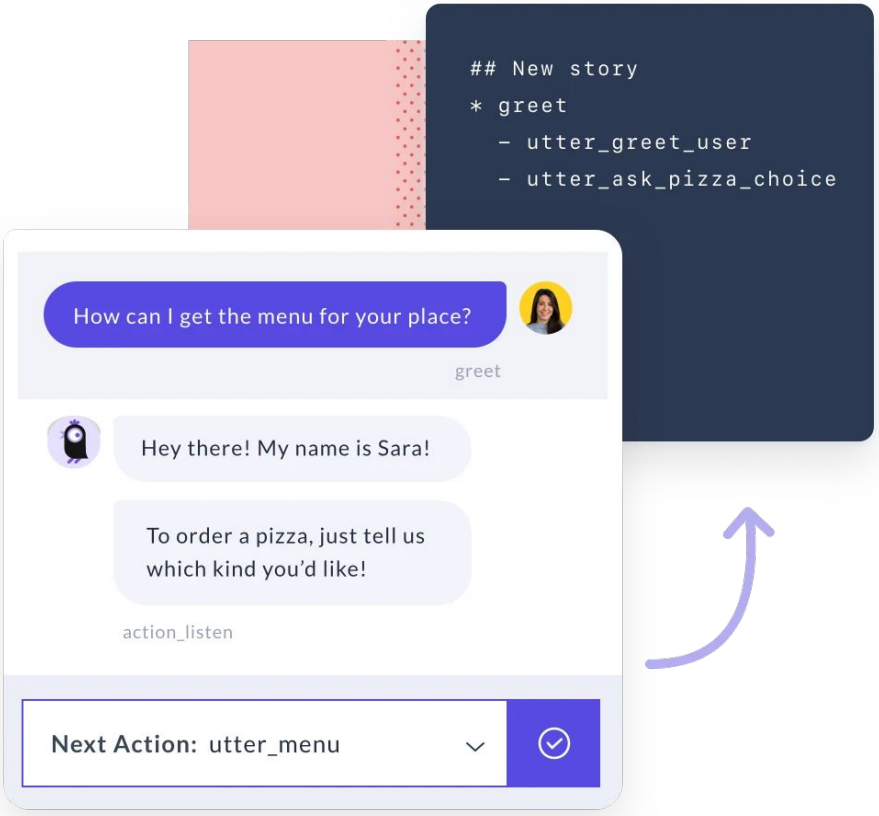


Annotate



Using a script to generate synthetic training data

Turning real messages into training examples



Real Conversations > Synthetic Data

During development, training data is created by:

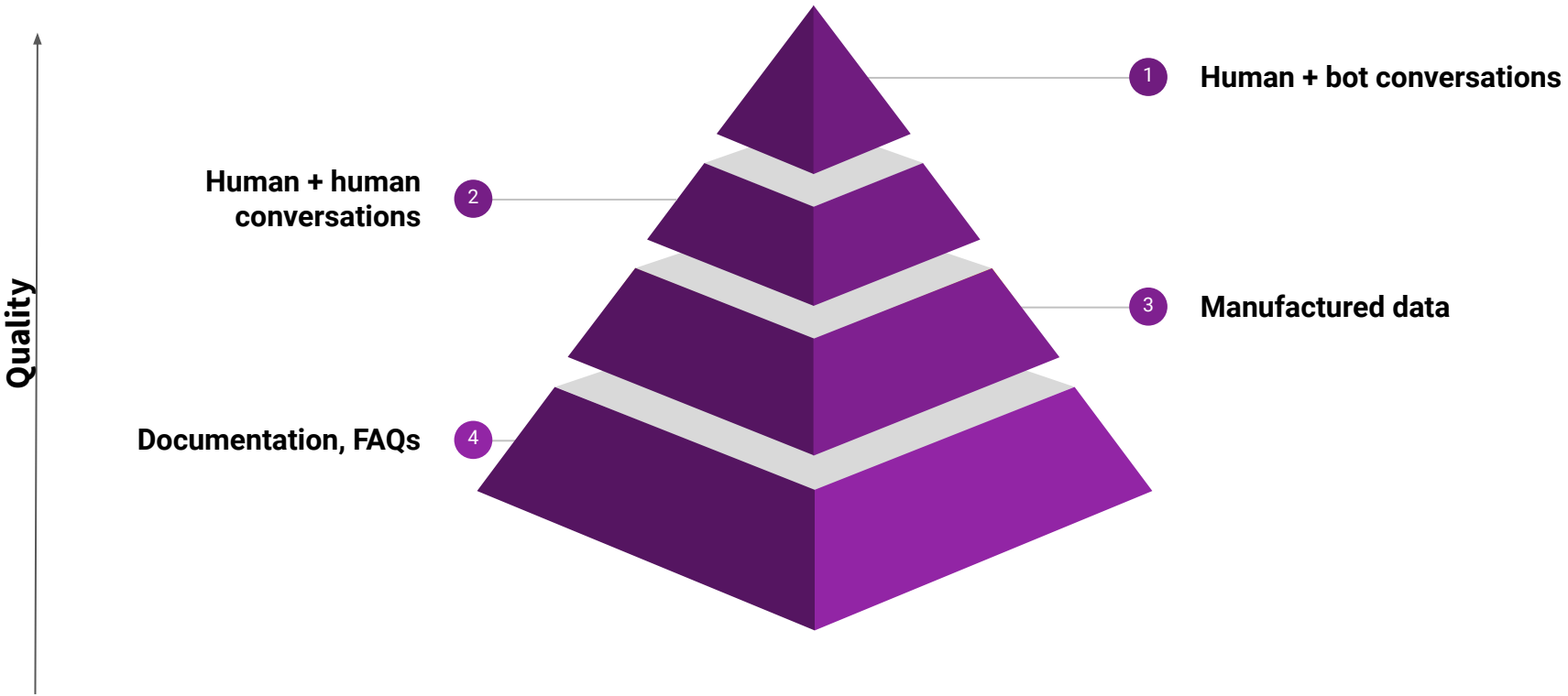
- Writing down hypothetical conversations and training examples
- Studying existing chat transcripts or knowledge bases
- Consulting subject matter experts

That's a great start, but all of those scenarios are based on **human:human** conversations.

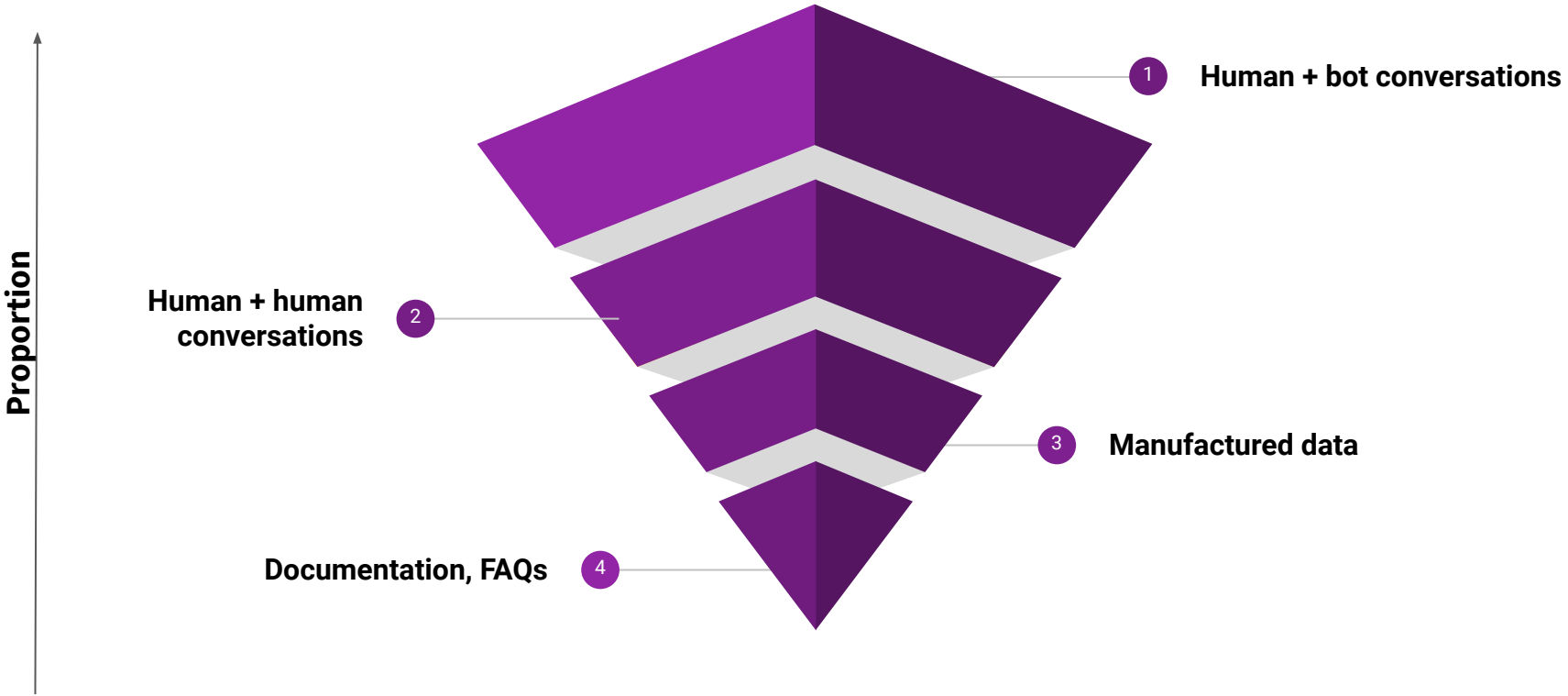
Human:bot conversations have different patterns - users talk to a bot differently than they do a human

The best training data is generated from the assistant's
actual conversations

Quality of data, by source



By the time you go to production, most of your training data should come from human:bot interactions



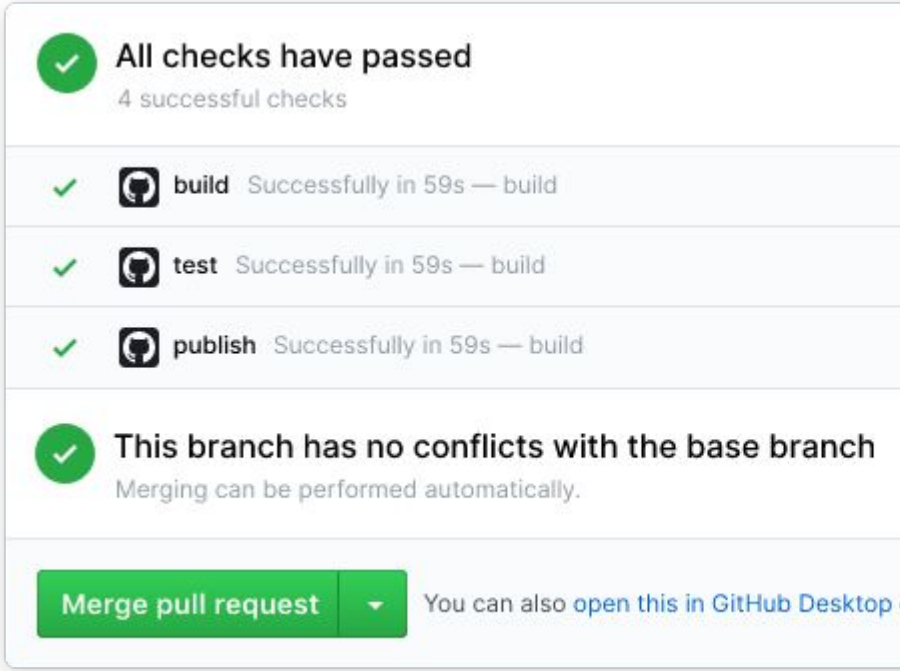
RASA

Test

Professional teams don’t ship applications without tests.

Use whole conversations as end-to-end tests

Run them on a continuous integration (CI) server.



Track

Use proxy measures to track which conversations are successful and which ones failed.

‘Negative’ signals are useful too, e.g. users **not** getting back in touch with support.

Fix

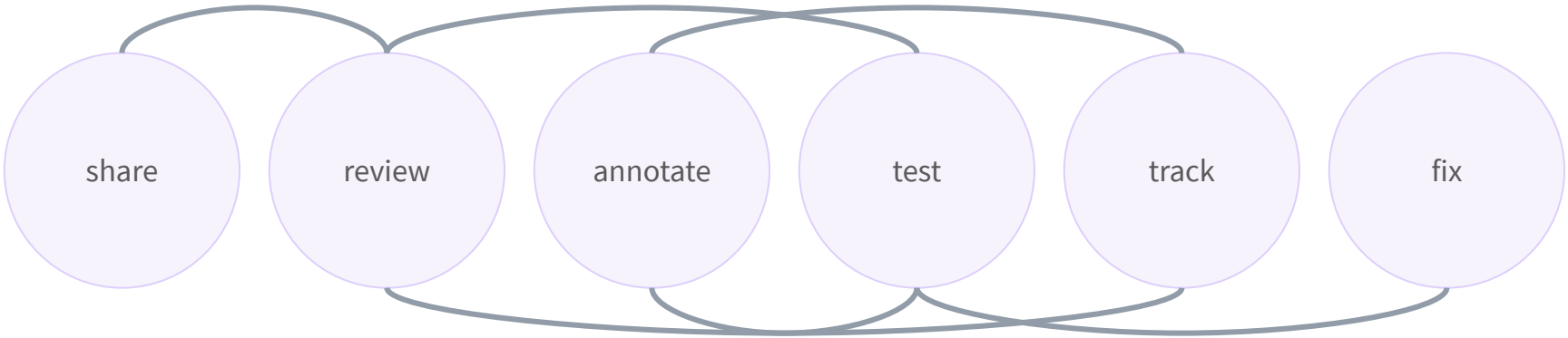
Study conversations that went smoothly and ones that failed.

Successful conversations can become new tests 🎉

Fix issues by annotating more data and/or fixing your code 🔧

CDD in practice

It’s not a linear process: you’ll find yourself jumping between these actions



Some actions require software skills, others a deep understanding of the user

Image credit: Freepik

CDD for teams

Product

- Product Managers
- Innovation and Business Leaders

Design

- UX Designers
- Conversation Designers
- Content Creators

Development

- Engineers
- Data Scientists
- Annotators

Rasa X is a tool to help teams do CDD

Rasa X turns conversations into training data

Review 👁 Annotate ✏️ Improve 💪

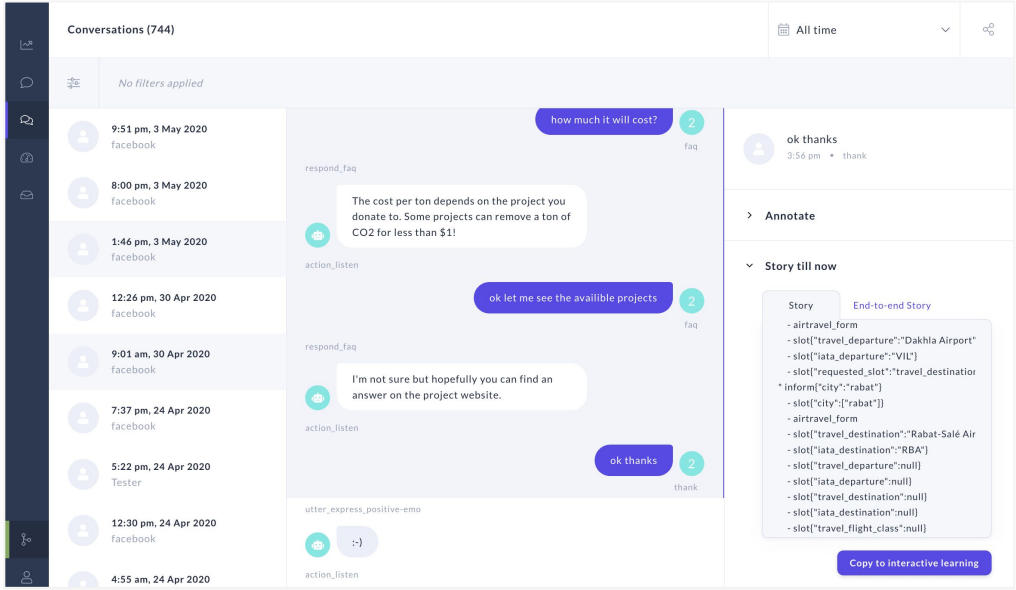
Rasa X layers on top of Rasa Open Source

Rasa Open Source: Framework for building AI assistants

```
language: en
pipeline:
- name: HFTransformersNLP
  model_weights: "gpt2"
  model_name: "gpt2"
- name: LanguageModelTokenizer
- name: LanguageModelFeaturizer
- name: DIETClassifier
  intent_classification: True
  entity_recognition: True
  epochs: 50
```

```
$ rasa train
$ rasa run
```

Rasa X: Tool for Conversation-Driven Development



Rasa X Local mode vs Server mode

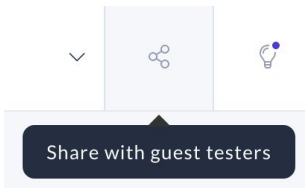
Local Mode:

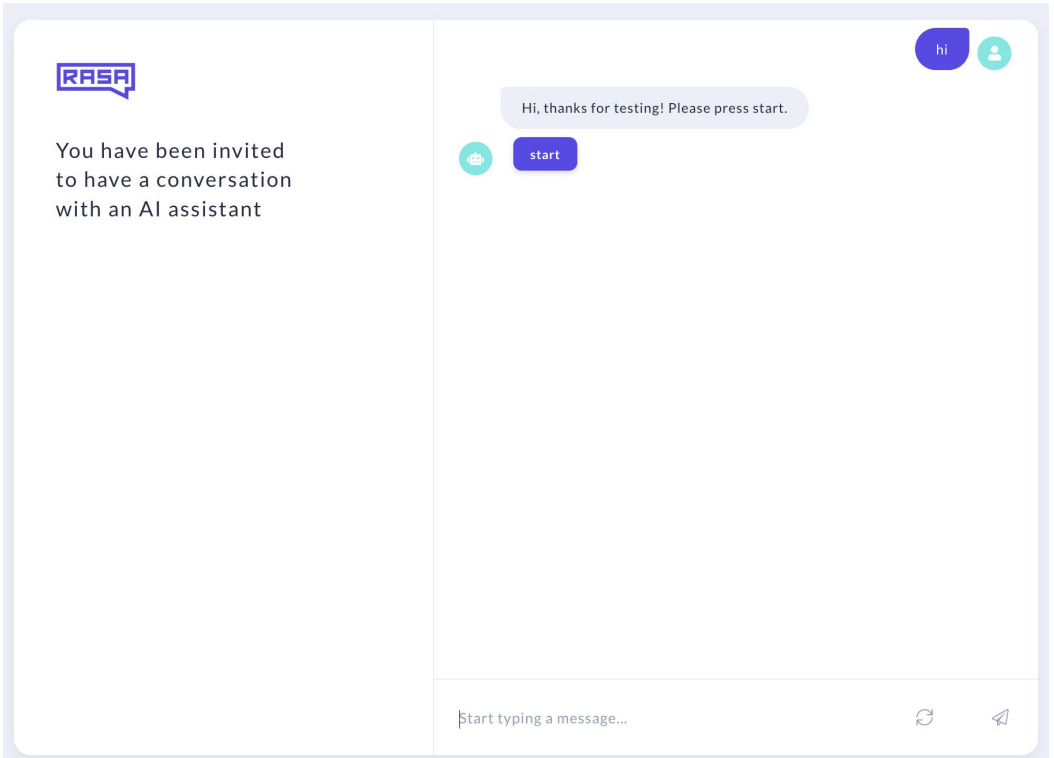
- Great for initial testing, familiarizing yourself with Rasa X
- Conversations get saved to a local database

Server Mode:

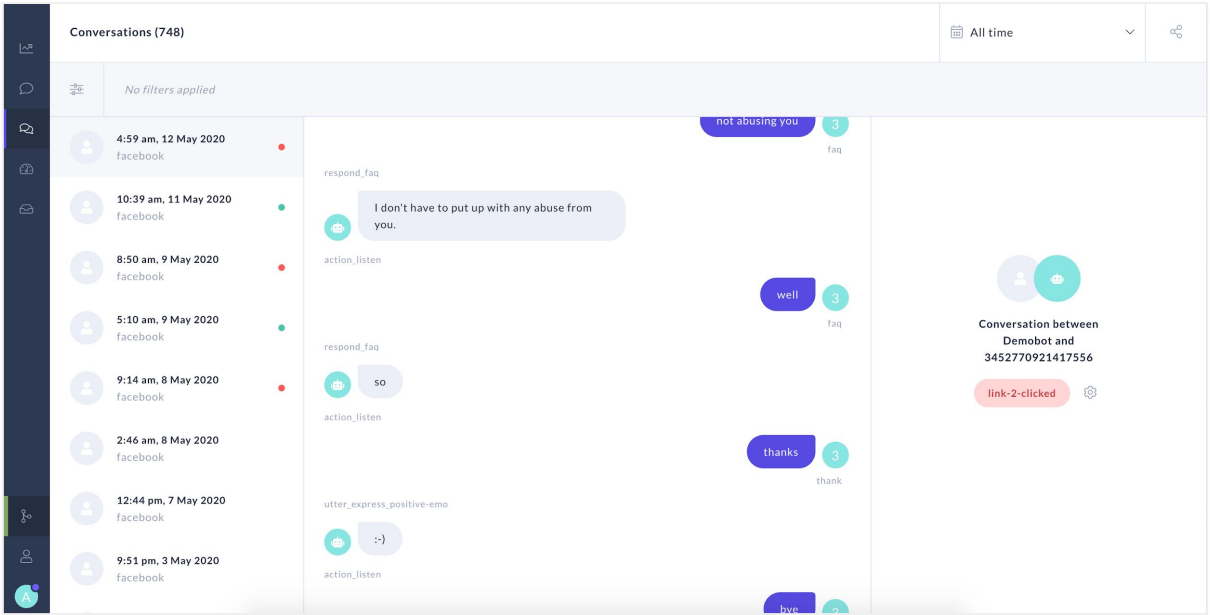
- Great for serving your bot and collecting conversations from many testers or users, 24/7
- Production-ready & scalable
- Deploys Rasa X (and your assistant) using Docker Compose or Kubernetes
- Conversations get saved to a production database
- Includes a Git integration (integrated version control)

Share your bot with testers using just a link

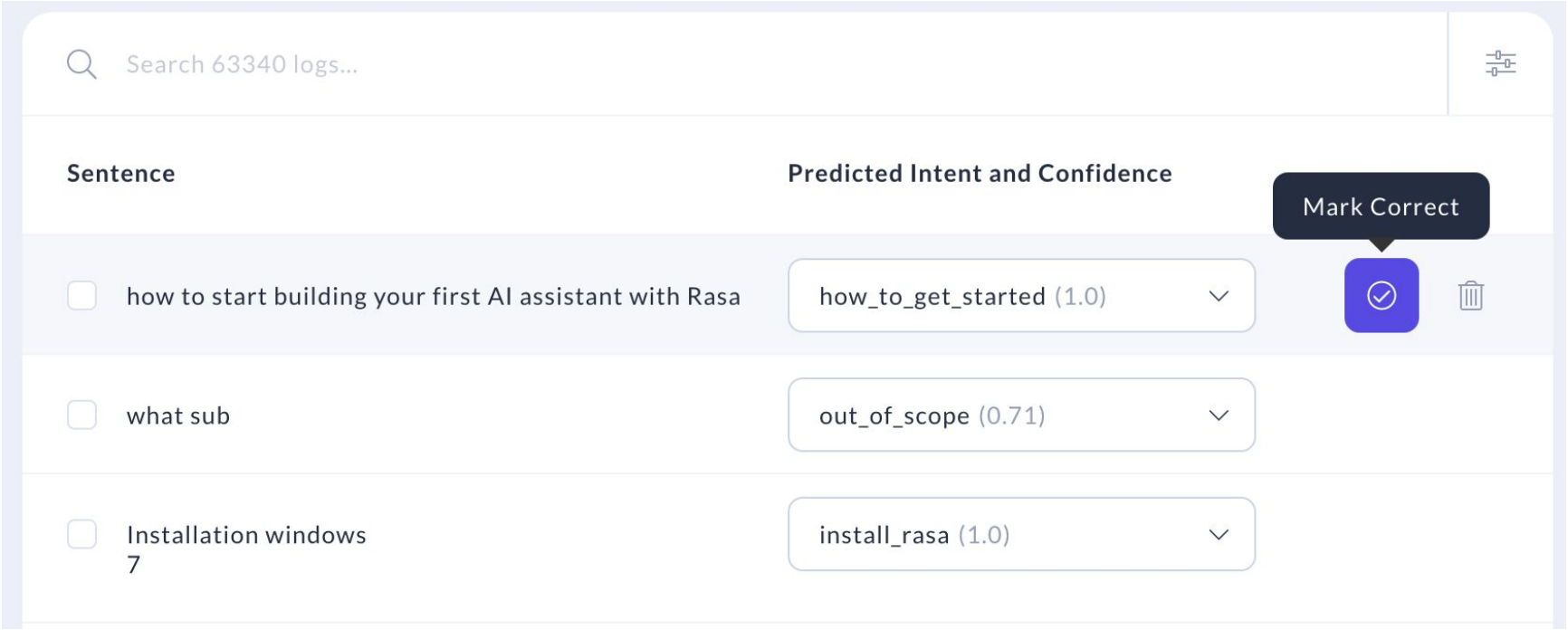




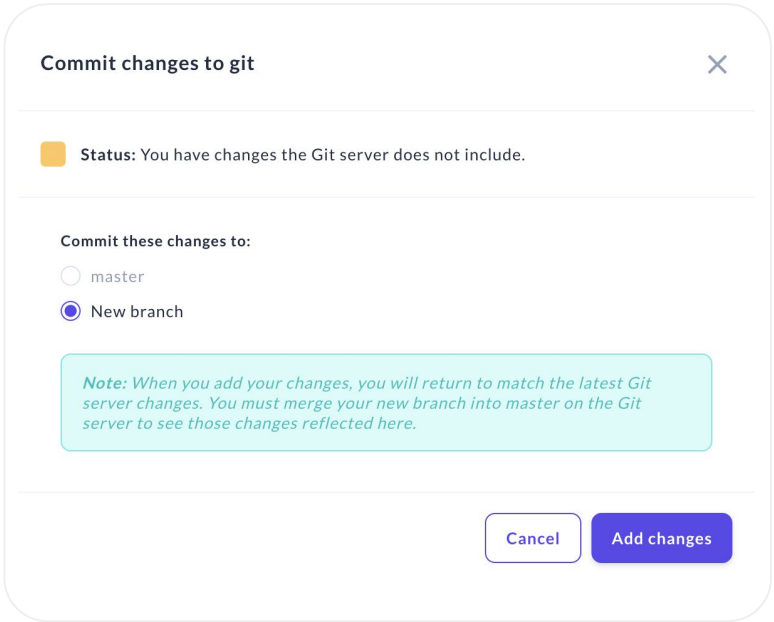
Review conversations coming in from every channel

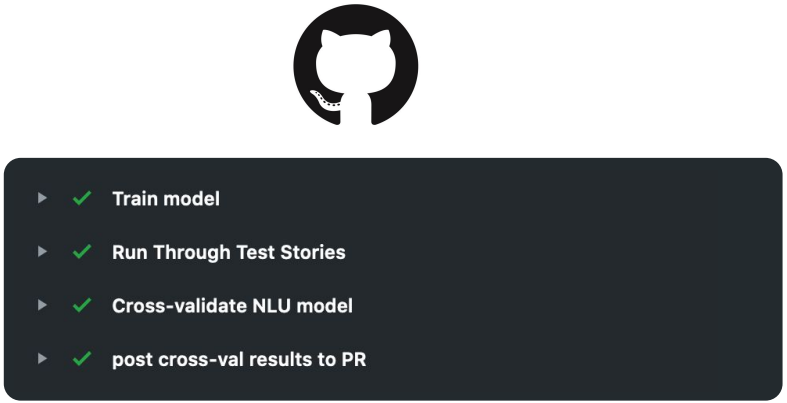


Annotate the messages coming in



Push new training data to git and trigger your CI pipeline



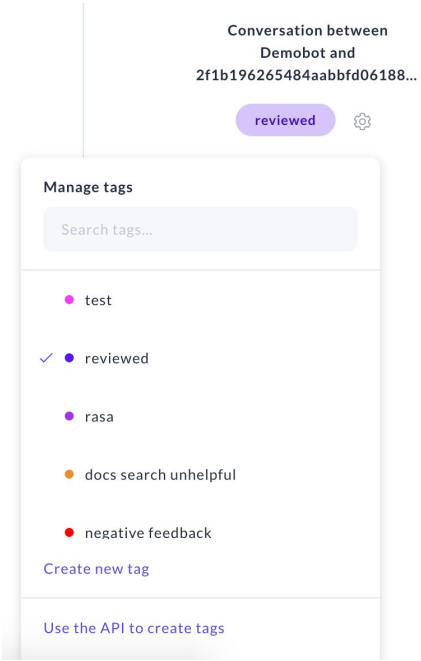


Intent Cross-Validation Results (5 folds)

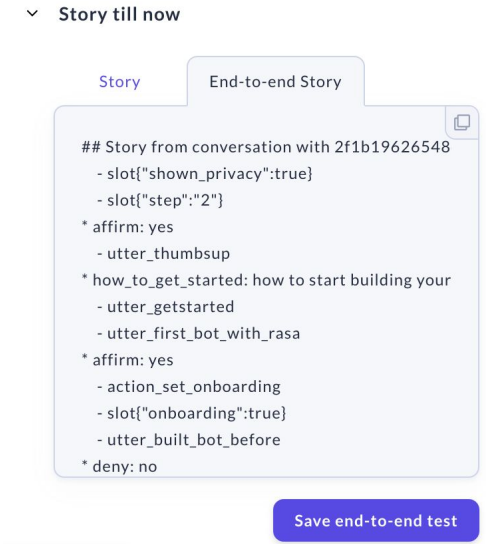
class	support	f1-score	confused_with
macro avg	2541	0.7999	N/A
weighted avg	2541	0.8392	N/A
faq	753	0.8172	inform(29), inquire-ask_clarification-offsets(17)
inform	619	0.9367	faq(24), estimate_emissions(6)

Track failures and successes

Use the API to automatically tag conversations, or add tags manually as you read



Turn successful conversations into new end-to-end tests



Installing Rasa X Locally

Checkout the master branch

```
git checkout master
```

Install Rasa X in your virtual environment:

```
pip3 install rasa-x --extra-index-url https://pypi.rasa.com/simple
```

Launch Rasa X in the browser by running the following command in your project directory:

```
rasa x
```

RASA

Testing Locally: ngrok

Creates a secure, publicly accessible tunnel URL connected to a process running on localhost.

Allows you to develop as though your application were running on a server instead of locally.

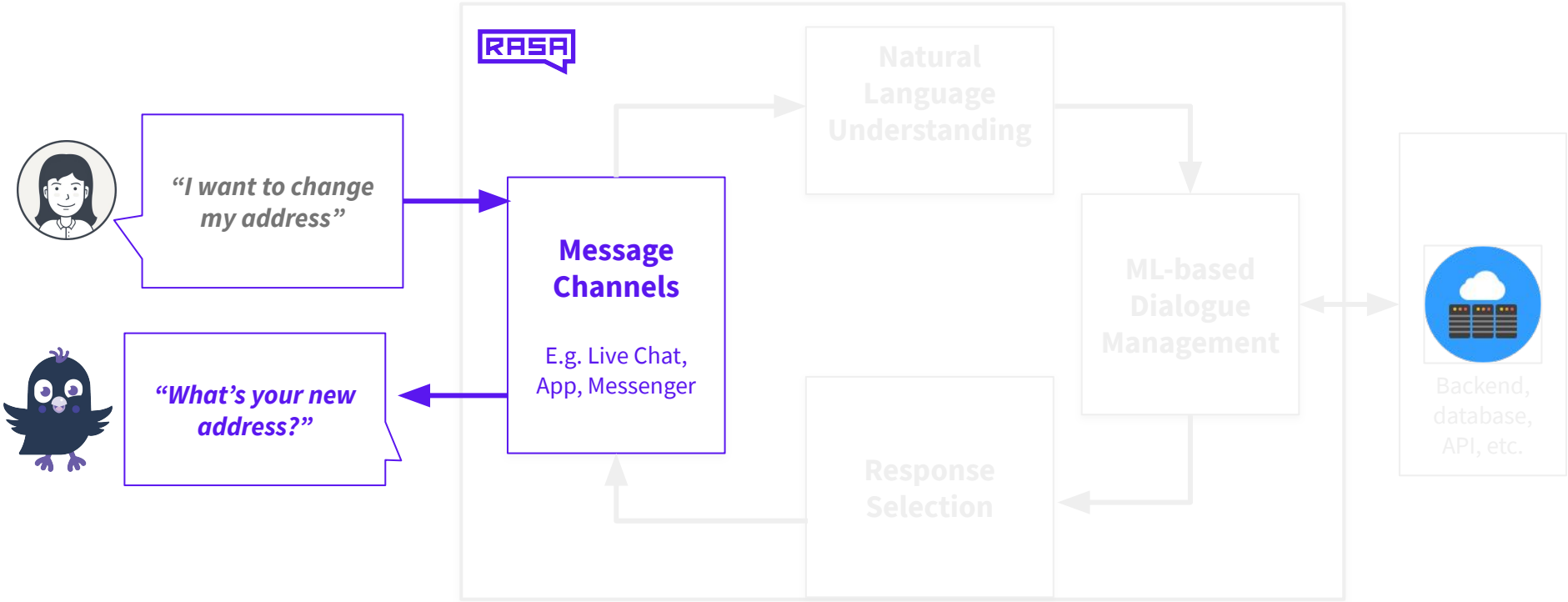
```
ngrok by @inconshreveable

Session Status                online
Session Expires               7 hours, 59 minutes
Version                       2.3.35
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://87ee9a2c9b3b.ngrok.io -> http://localhost:5005
Forwarding                    https://87ee9a2c9b3b.ngrok.io -> http://localhost:5005

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Starting ngrok

1. Start Rasa X. When you start Rasa X locally, it runs on localhost:5002
2. Open a new terminal window and start ngrok. Specify that ngrok should listen for the process running on port 5002
   a. ngrok http 5002
3. Save the URL generated by ngrok
4. Generate a Share your Bot link in Rasa X.
5. In the Share Your Bot URL, replace localhost:5002 with your ngrok URL
   http://92099d45e813.ngrok.io/guest/conversations/production/76a25a7880ac40f5bda7b42d650bca75

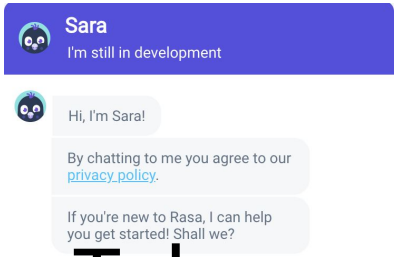
Messaging Channels

Rasa Open Source



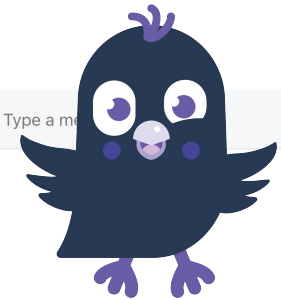
MESSAGING CHANNELS

Built in Channels



slack

Telegram



twilio

RASA

Messaging Channels

- Channel connection details are kept in the **credentials.yml** file
- When you move to production, you'll want to re-create these credentials securely on your server
- After you update credentials.yml, start (or restart) the server with `rasa run`

config.yml | domain.yml ● | credentials.yml ×

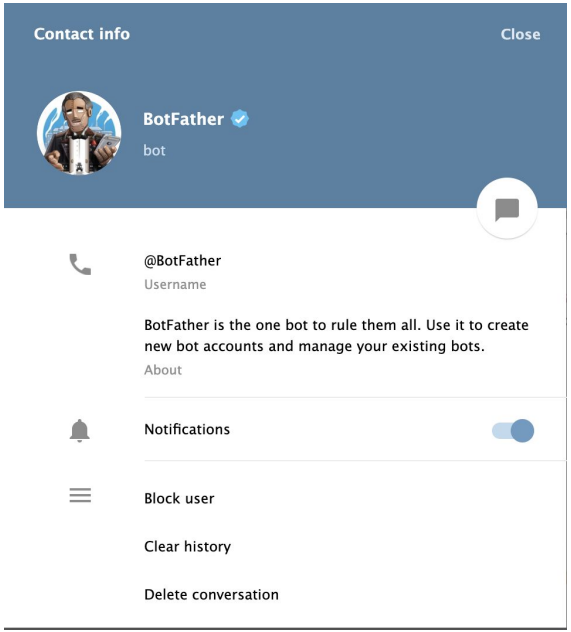
financial-demo > credentials.yml

```
28  #  pw: "<bot token>"
29  #  webhook_url: "<callback URL>"
30
31  rasa:
32    url: "http://localhost:5002/api"
33
34  telegram:
35    access_token: "1292722675:AAGwNrGWTrIvOeZxjItqwZLhZxakDuGLVvY"
36    verify: "rasaworkshop_bot"
37    webhook_url: "https://e6e218a18360.ngrok.io/webhooks/telegram/webhook"
38
```

Telegram

Register a new bot:

- Go to https://web.telegram.org/#/im?p=@BotFather
- Type /newbot
- Name shows up next to the bot's profile icon
- Username is the bot's handle, and ends in _bot



credentials.yml

```
Telegram:
   access_token: "your API token"
   verify: "bot's username (ends in _bot)"
   webhook_url: "https://<ngrok-url>/webhooks/telegram/webhook"
```

Testing Telegram locally

1. Start the rasa server (this command starts the server without a chat session on the command line)
   a. rasa run
2. Start ngrok on port 5005
   a. ngrok http 5005
3. Add the ngrok URL to your Telegram webhook URL - save the file
4. Restart the Rasa server (so changes to credentials.yml get picked up)
   a. /stop
   b. rasa run
5. Make sure your action server is running!
6. Try talking to your bot on Telegram

What’s Next?

Get people to chat with your assistant and keep improving it!

Where to go from here?

1. Look at conversations and ask: how does your assistant struggle?
2. Learn how to solve problem from Rasa Docs and Community Forum
3. Improve assistant
4. Test and deploy your update
5. Repeat 😎

THE APPROACH

Continually improve your assistant using Rasa X

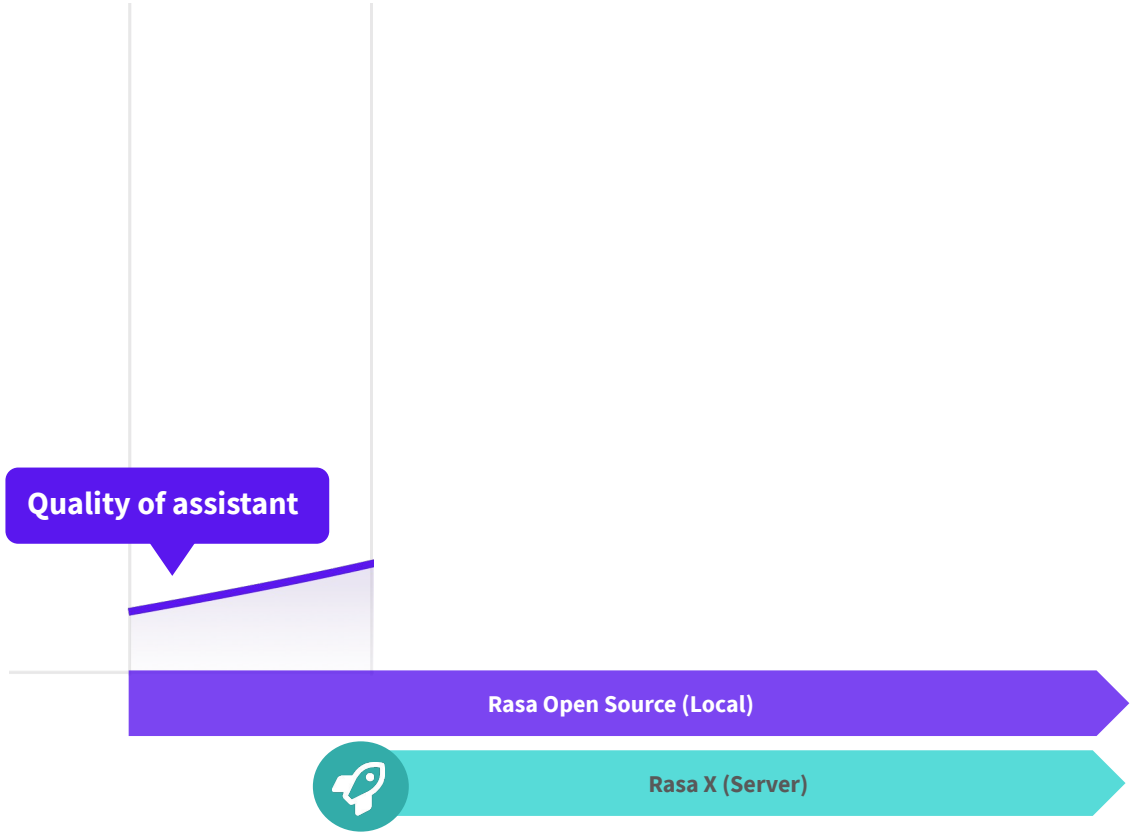
Ensure your new assistant passes tests using **continuous integration (CI)** and redeploy it to users using **continuous deployment (CD)**

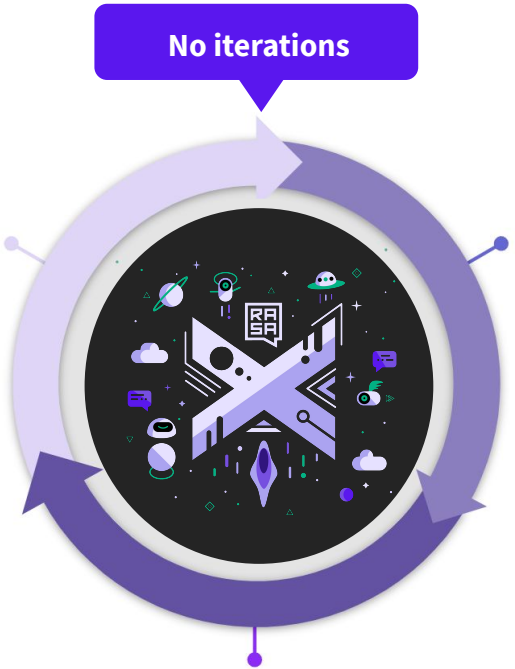
Collect conversations between users and your assistant

Review conversations and **improve your assistant** based on what you learn

THE APPROACH

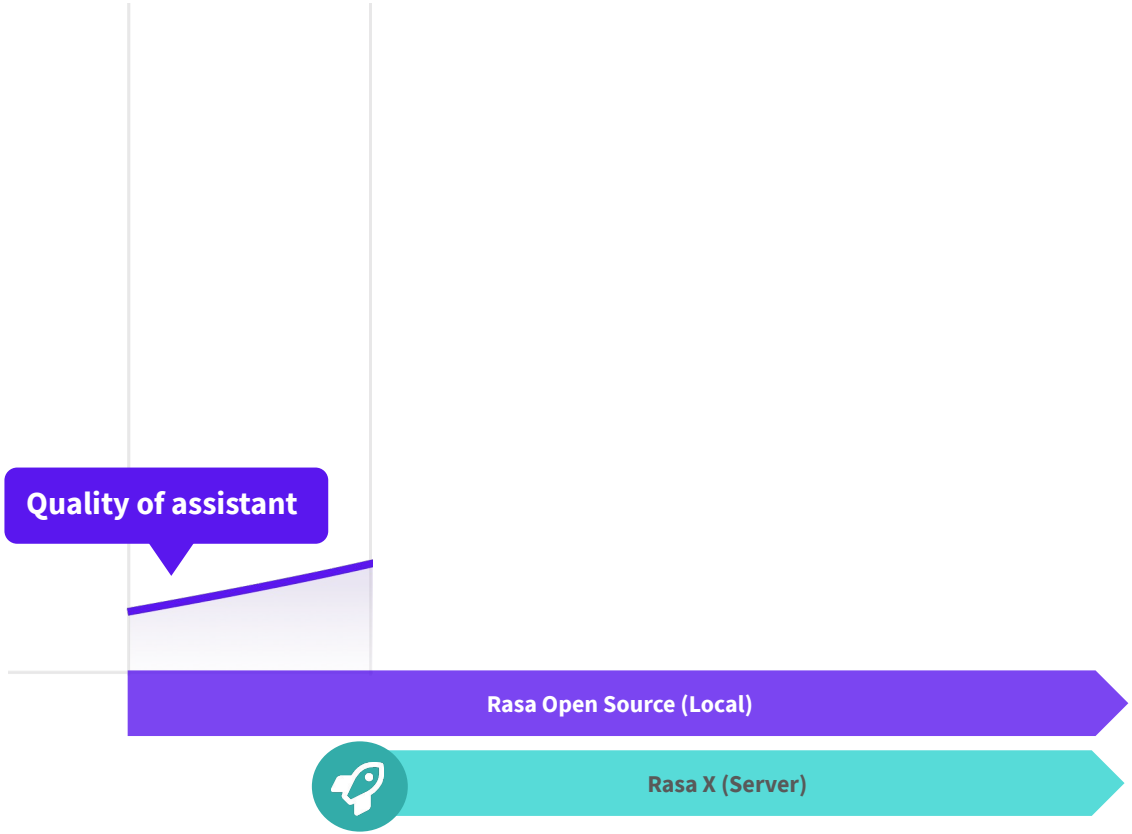
The path to a contextual assistant





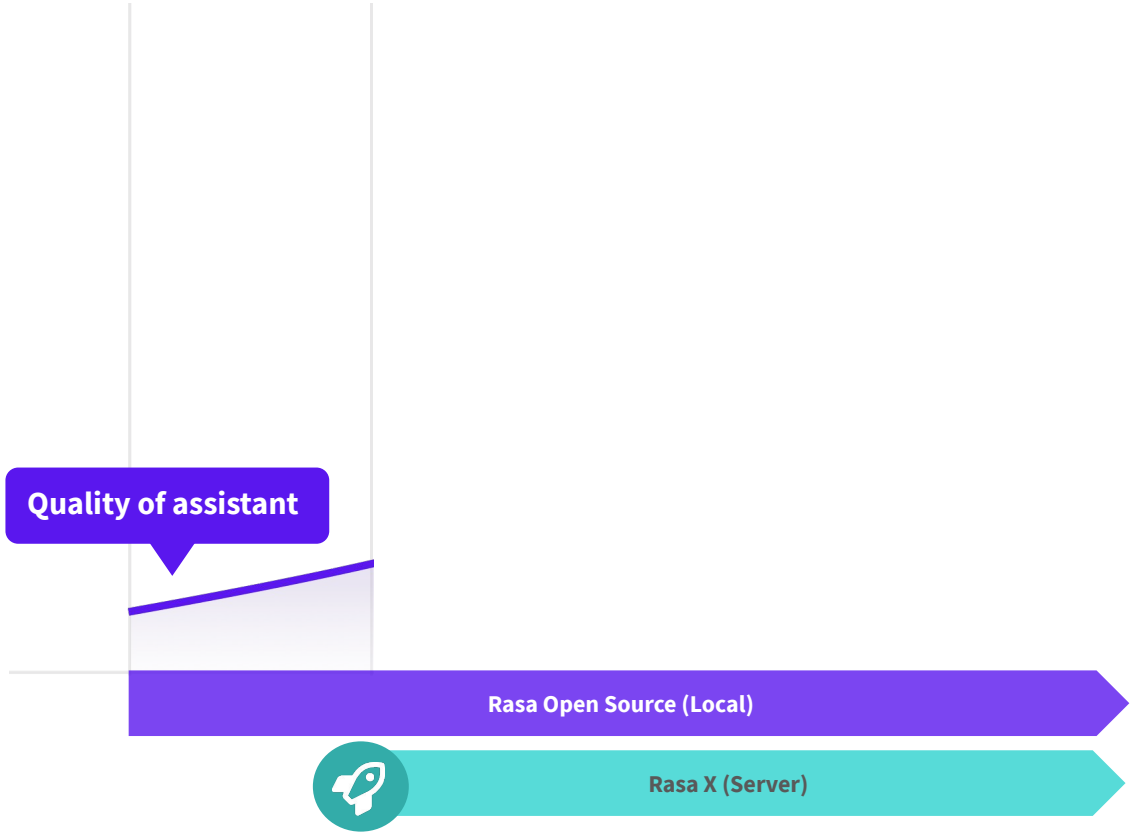
THE APPROACH

The path to a contextual assistant



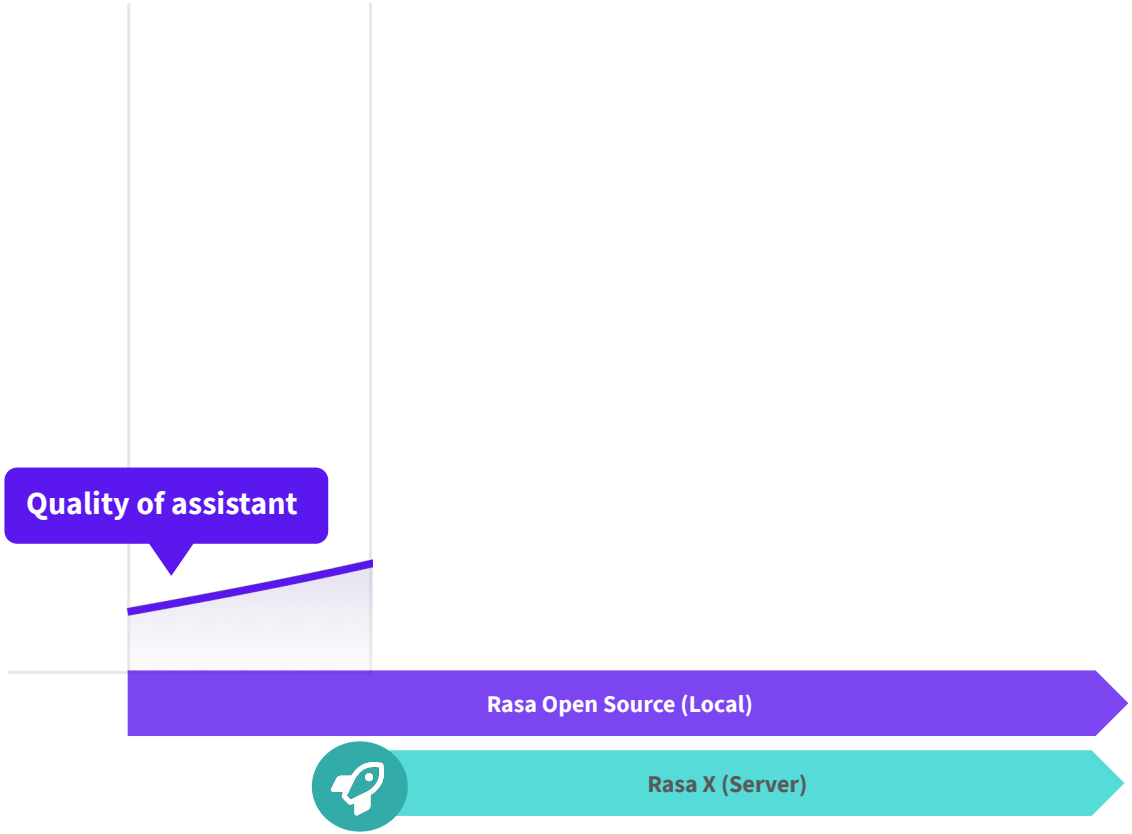
THE APPROACH

The path to a contextual assistant



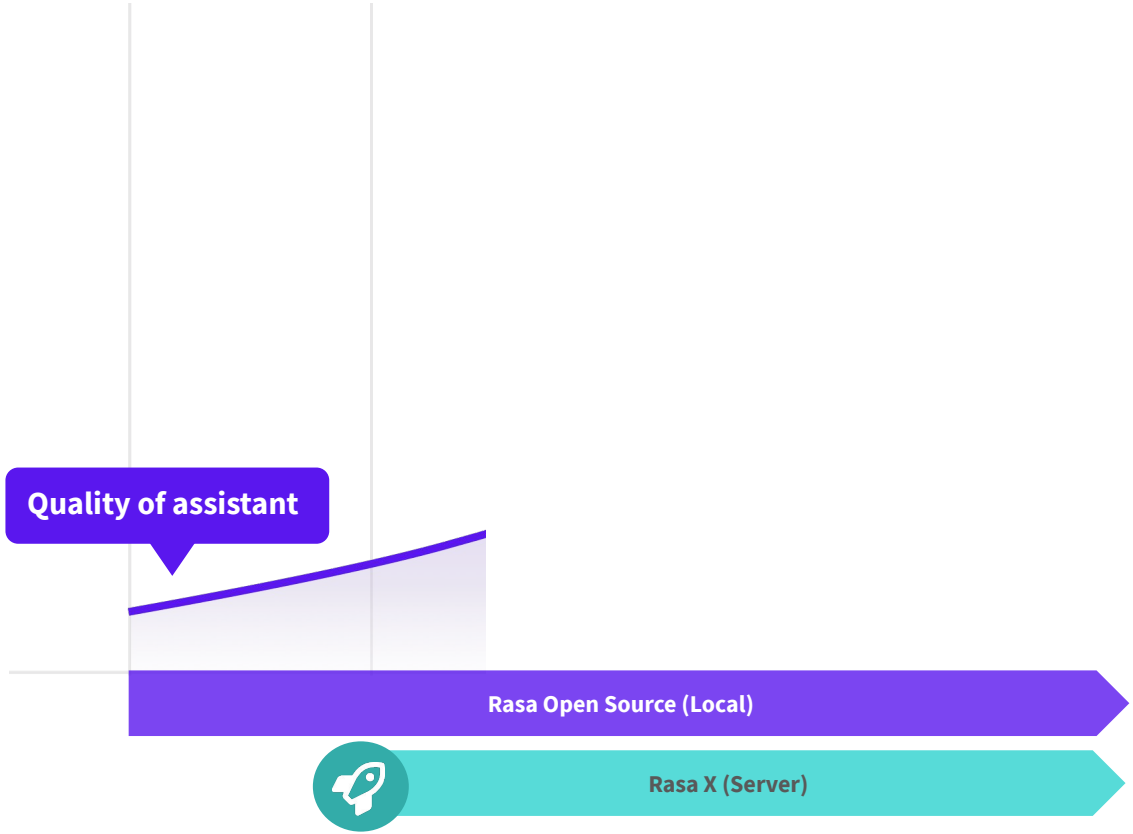
THE APPROACH

The path to a contextual assistant



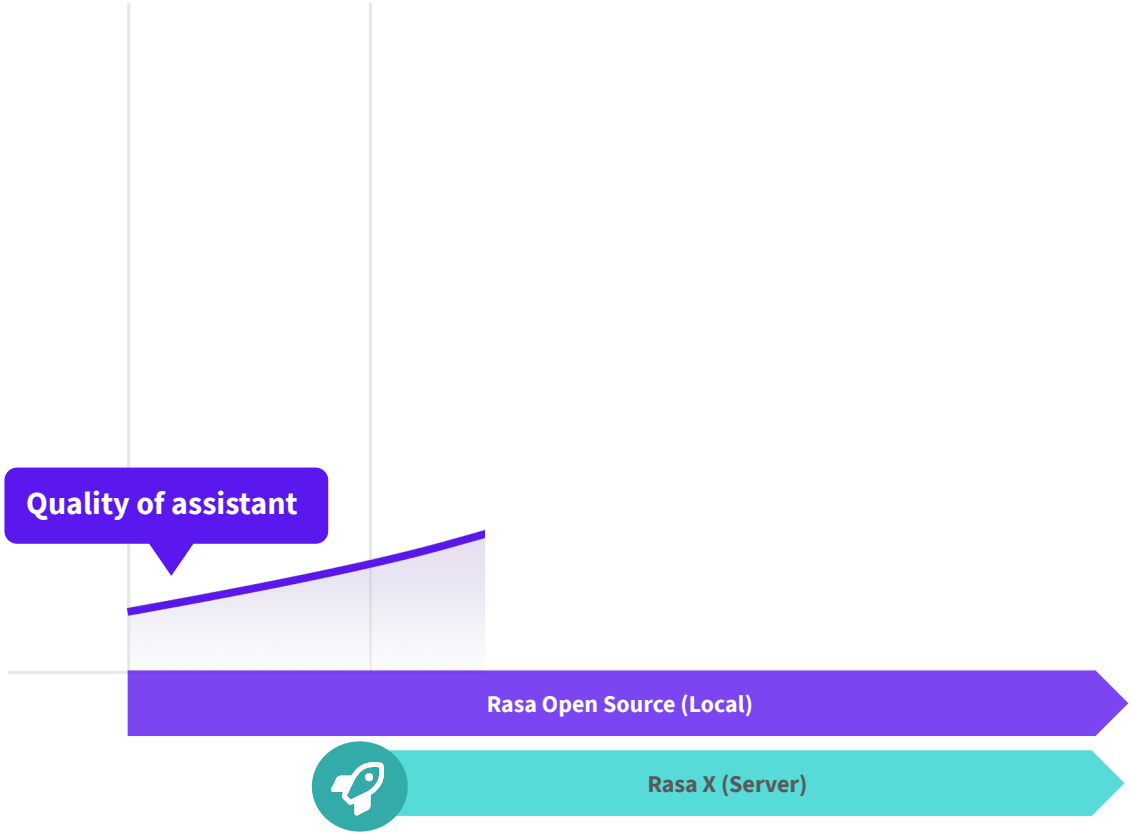
THE APPROACH

The path to a contextual assistant



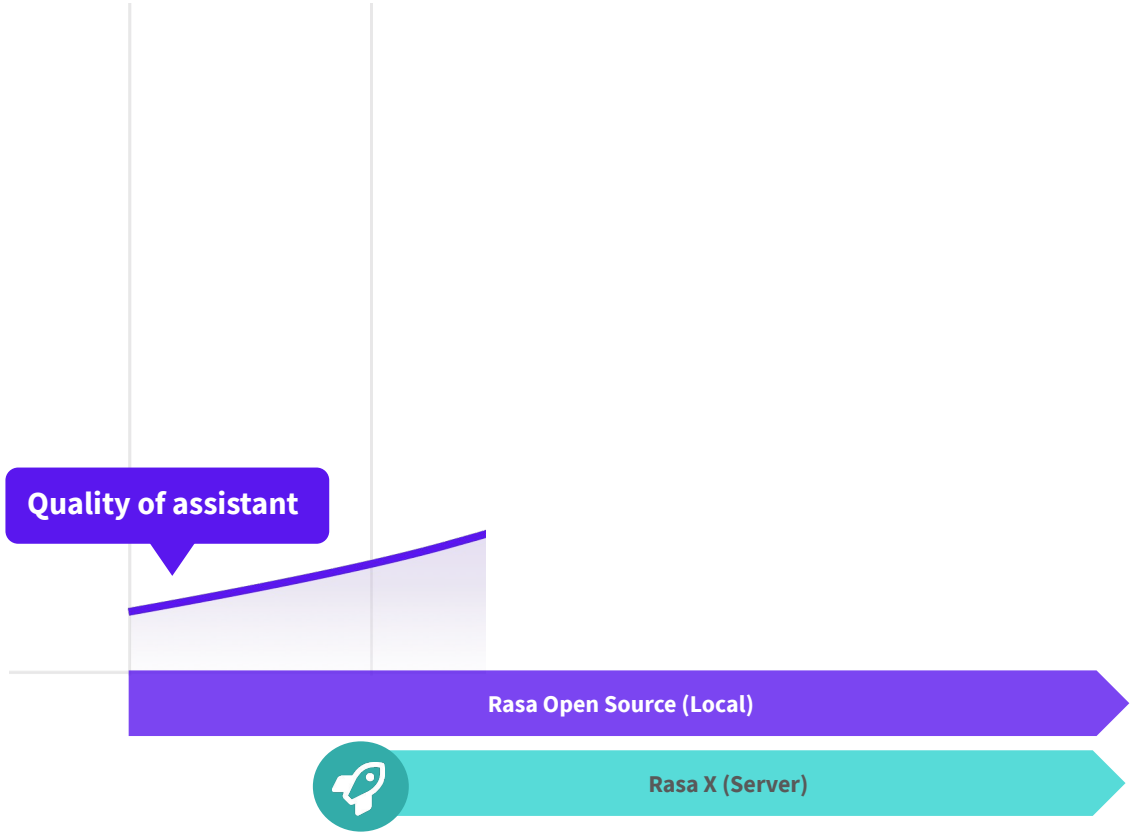
THE APPROACH

The path to a contextual assistant



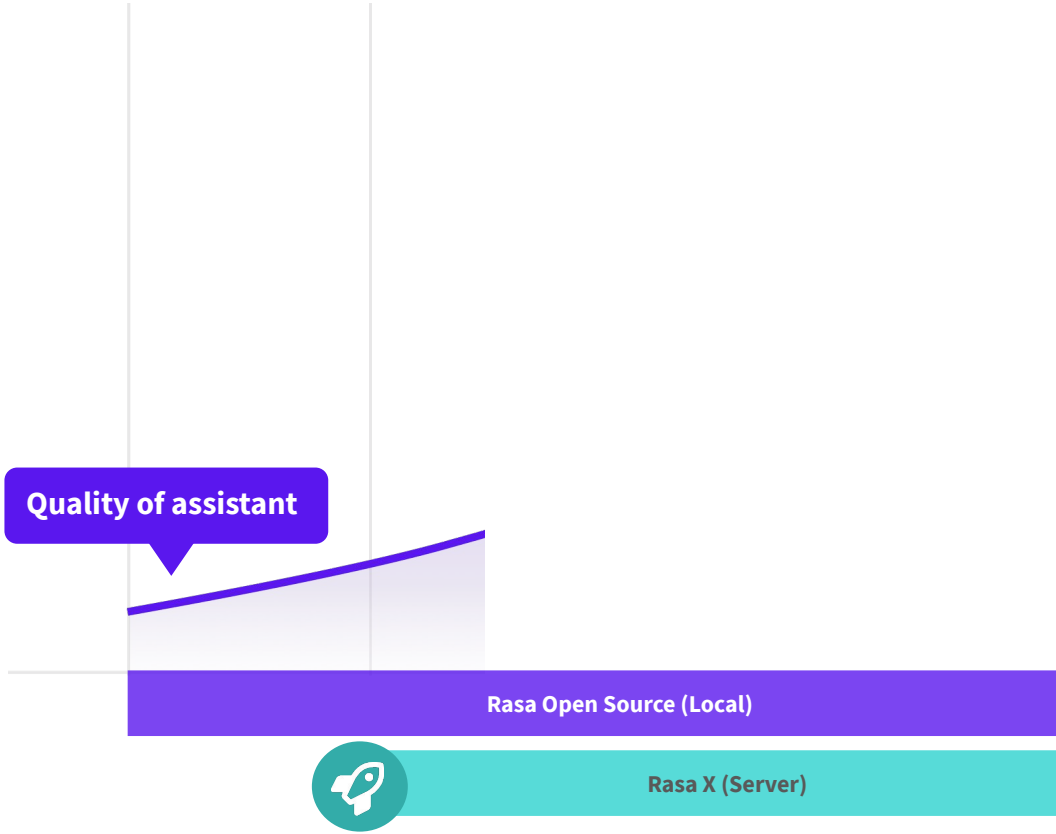
THE APPROACH

The path to a contextual assistant



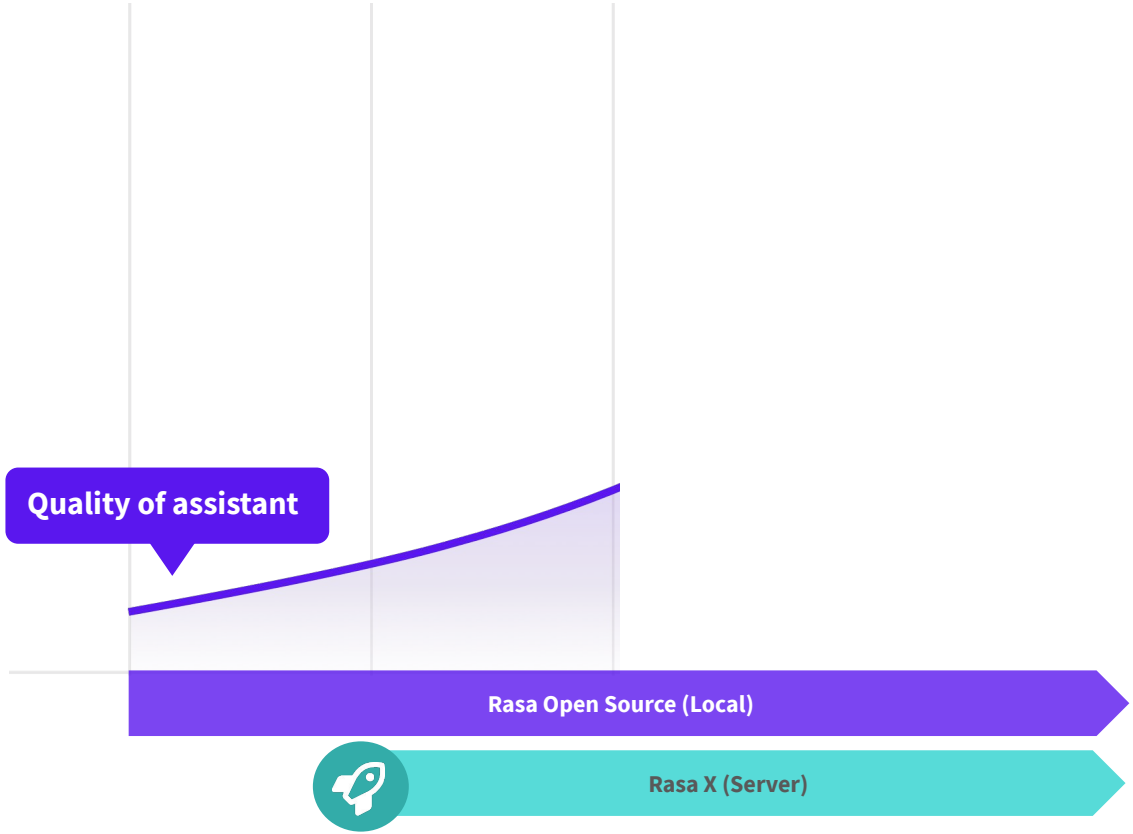
THE APPROACH

The path to a contextual assistant



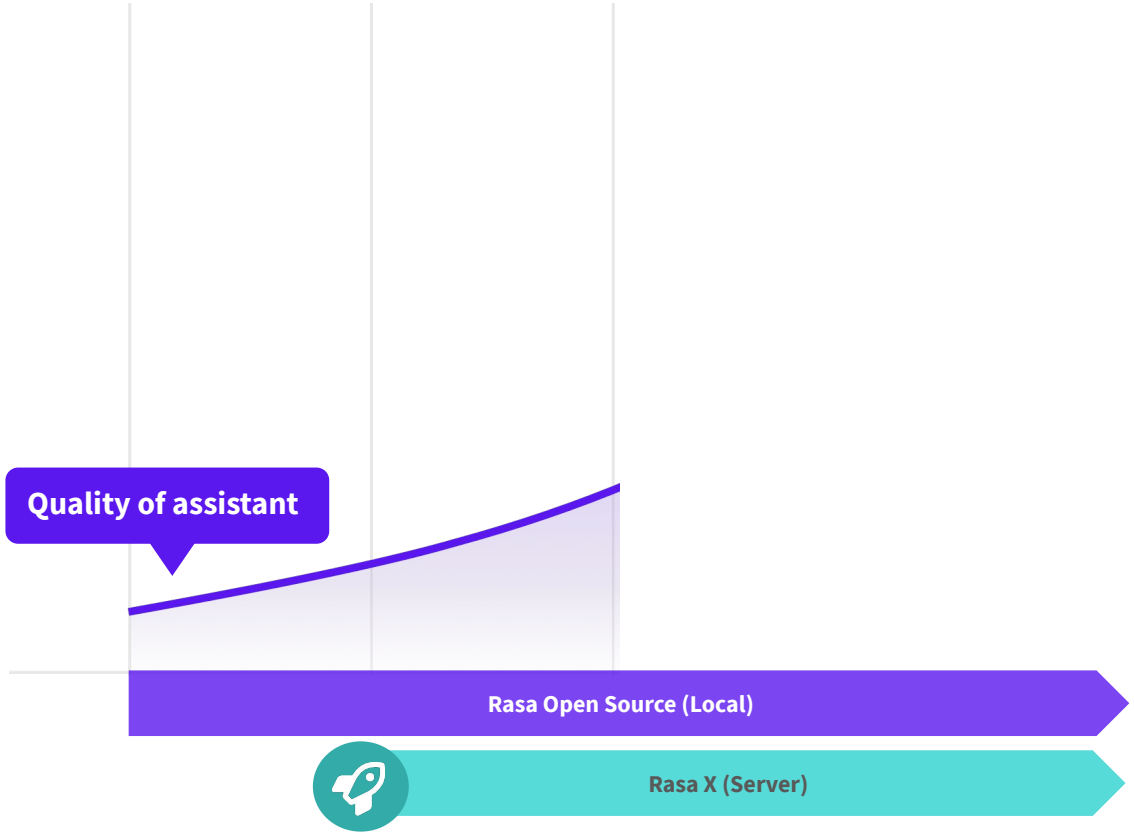
THE APPROACH

The path to a contextual assistant



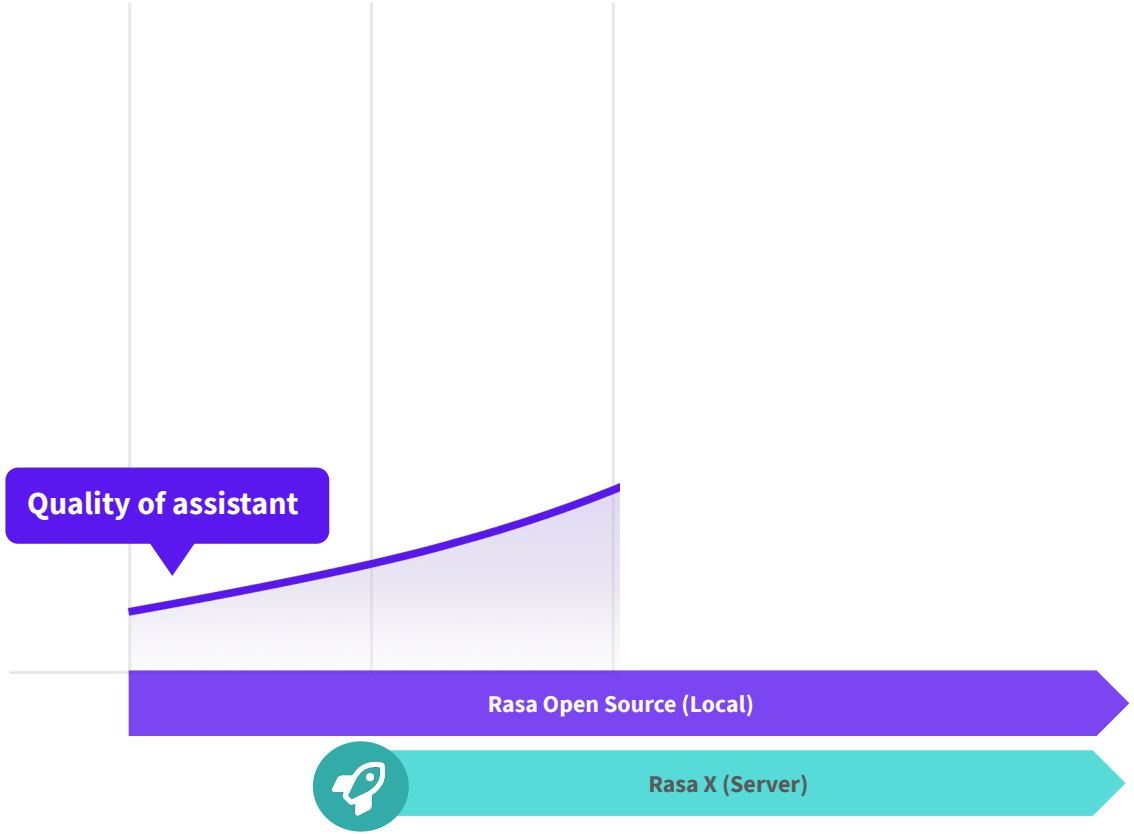
THE APPROACH

The path to a contextual assistant



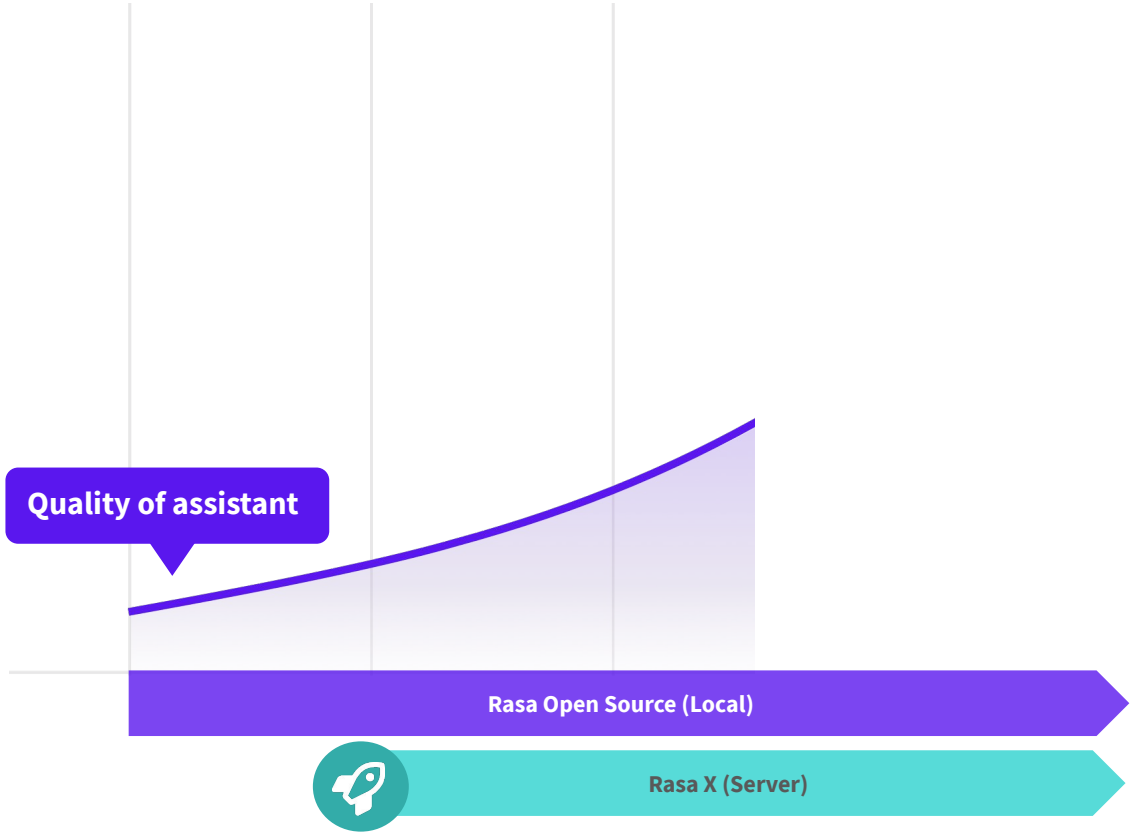
THE APPROACH

The path to a contextual assistant



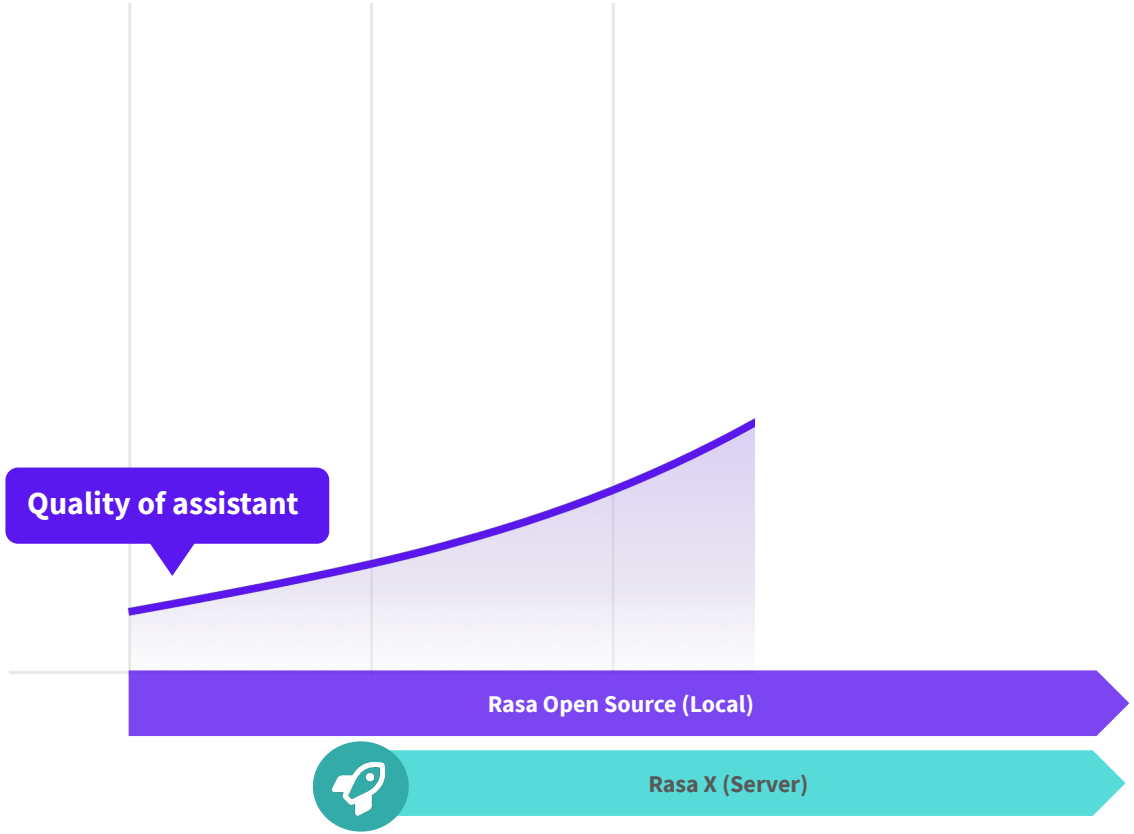
THE APPROACH

The path to a contextual assistant



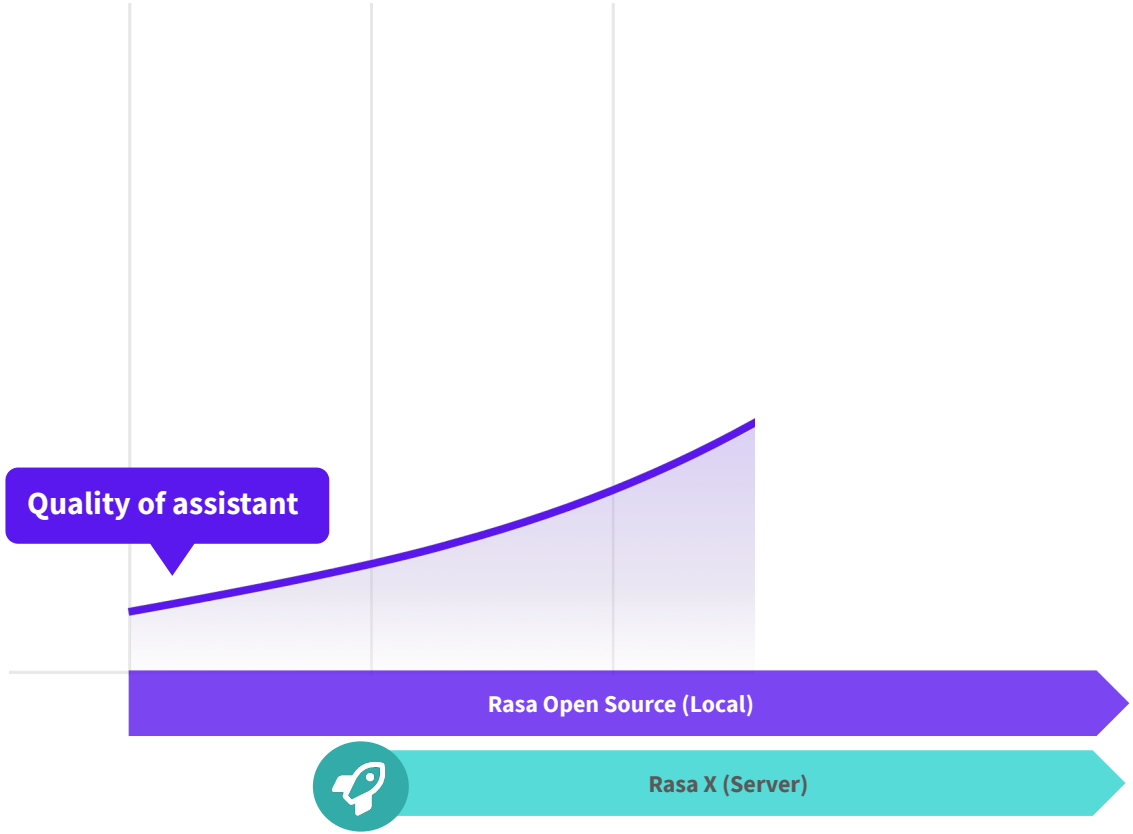
THE APPROACH

The path to a contextual assistant



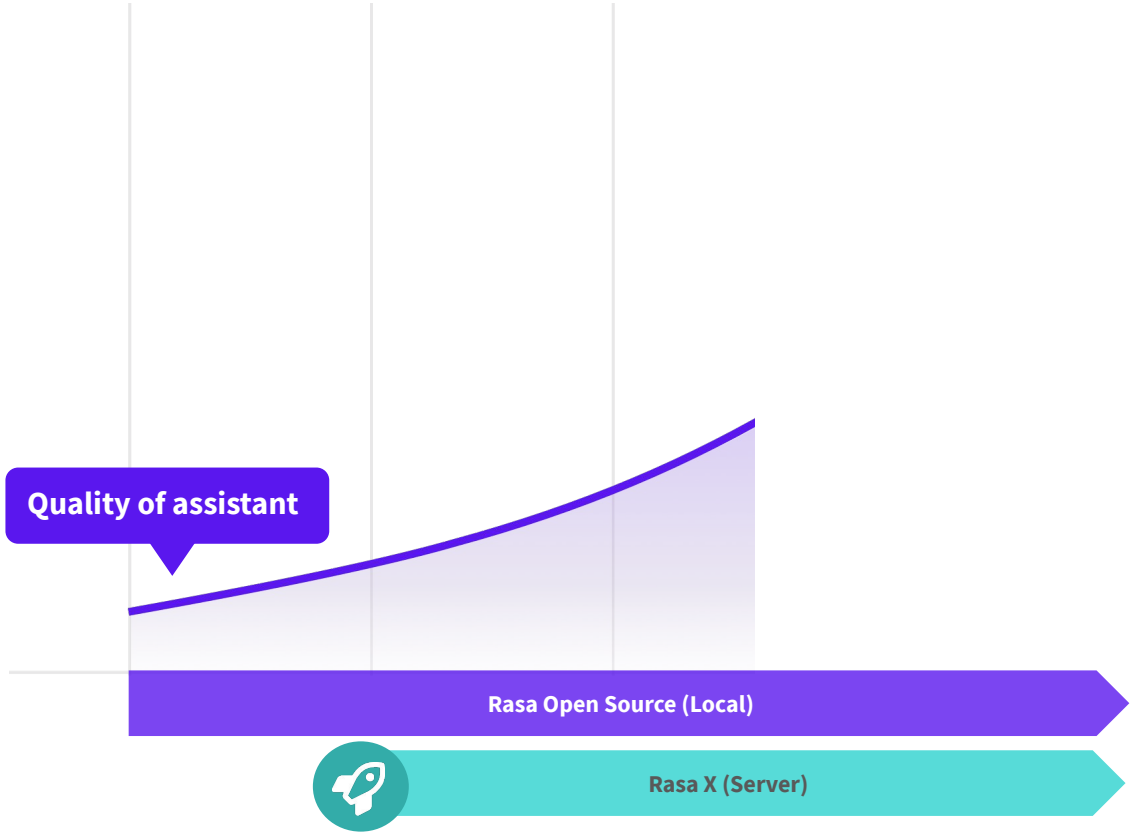
THE APPROACH

The path to a contextual assistant



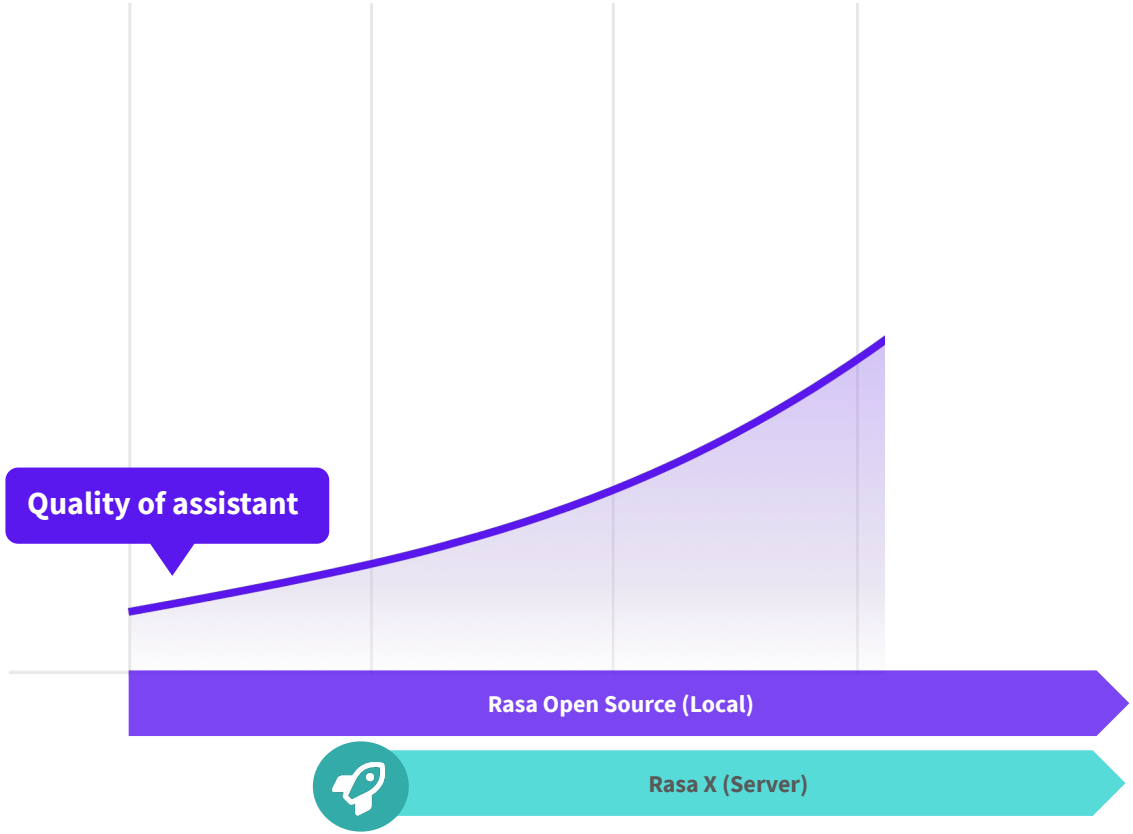
THE APPROACH

The path to a contextual assistant



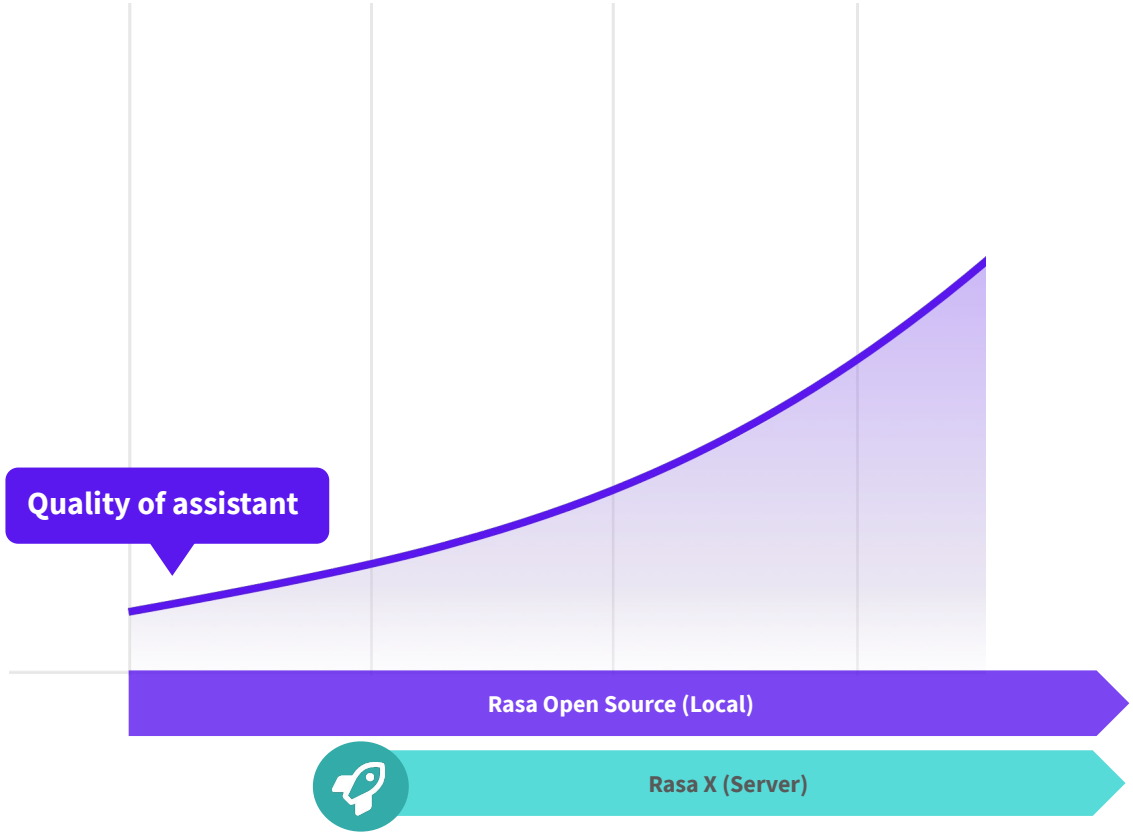
THE APPROACH

The path to a contextual assistant



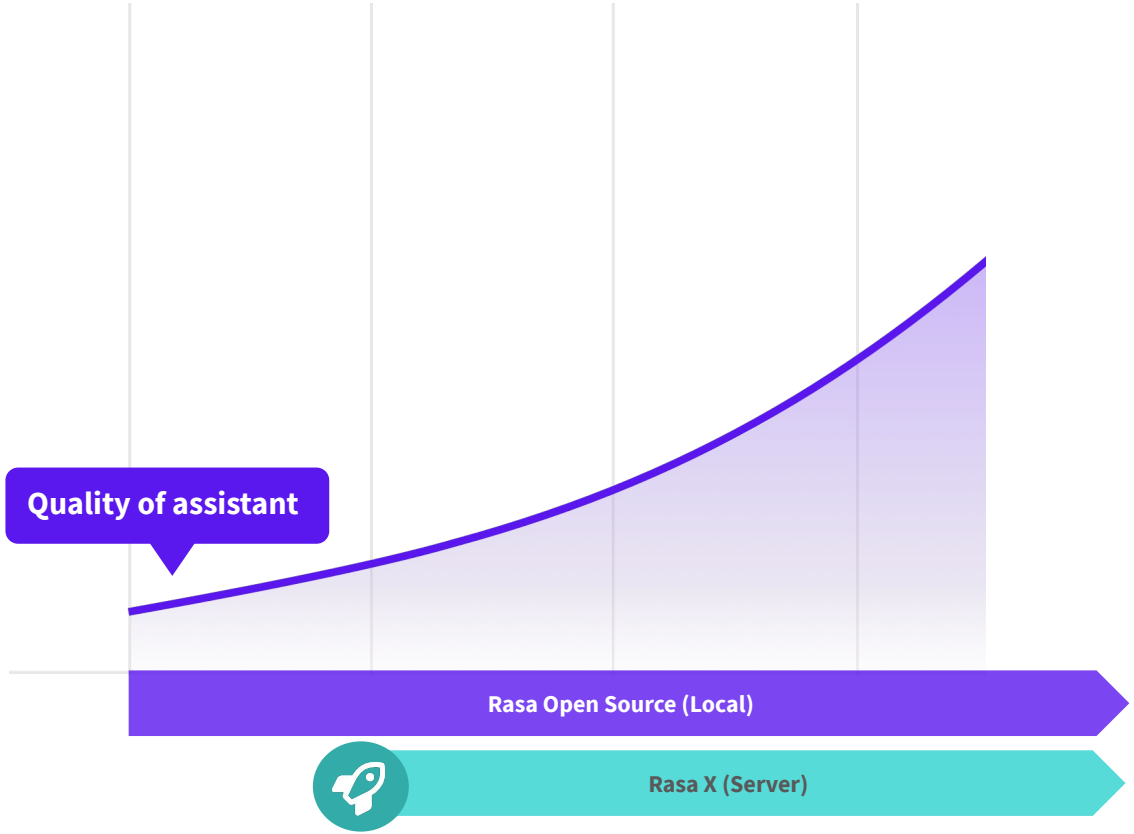
THE APPROACH

The path to a contextual assistant



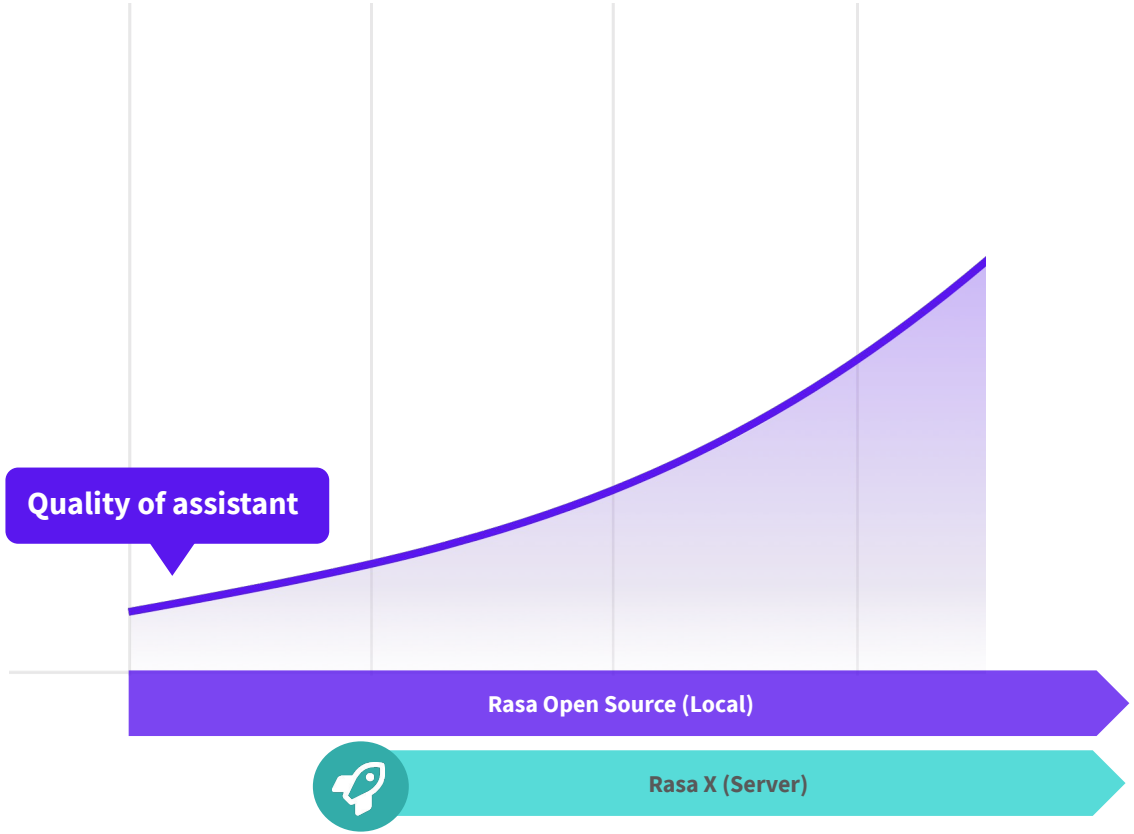
THE APPROACH

The path to a contextual assistant



THE APPROACH

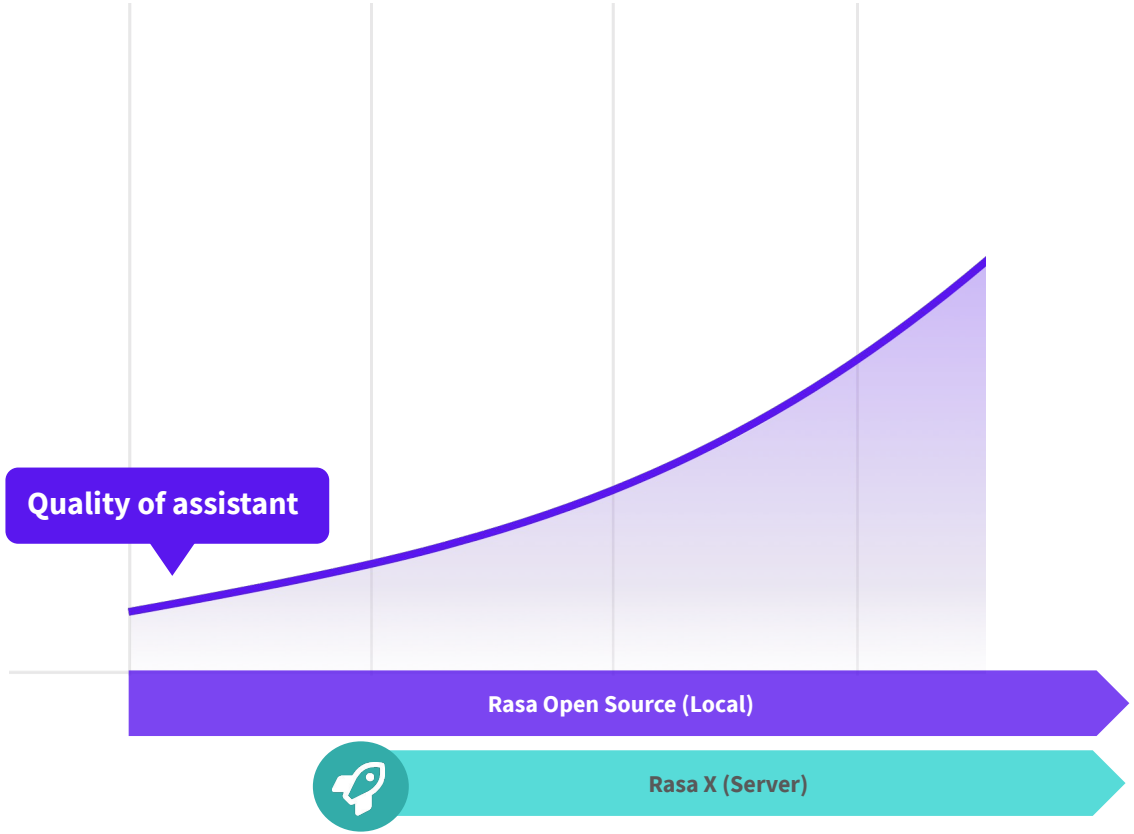
The path to a contextual assistant



RASA

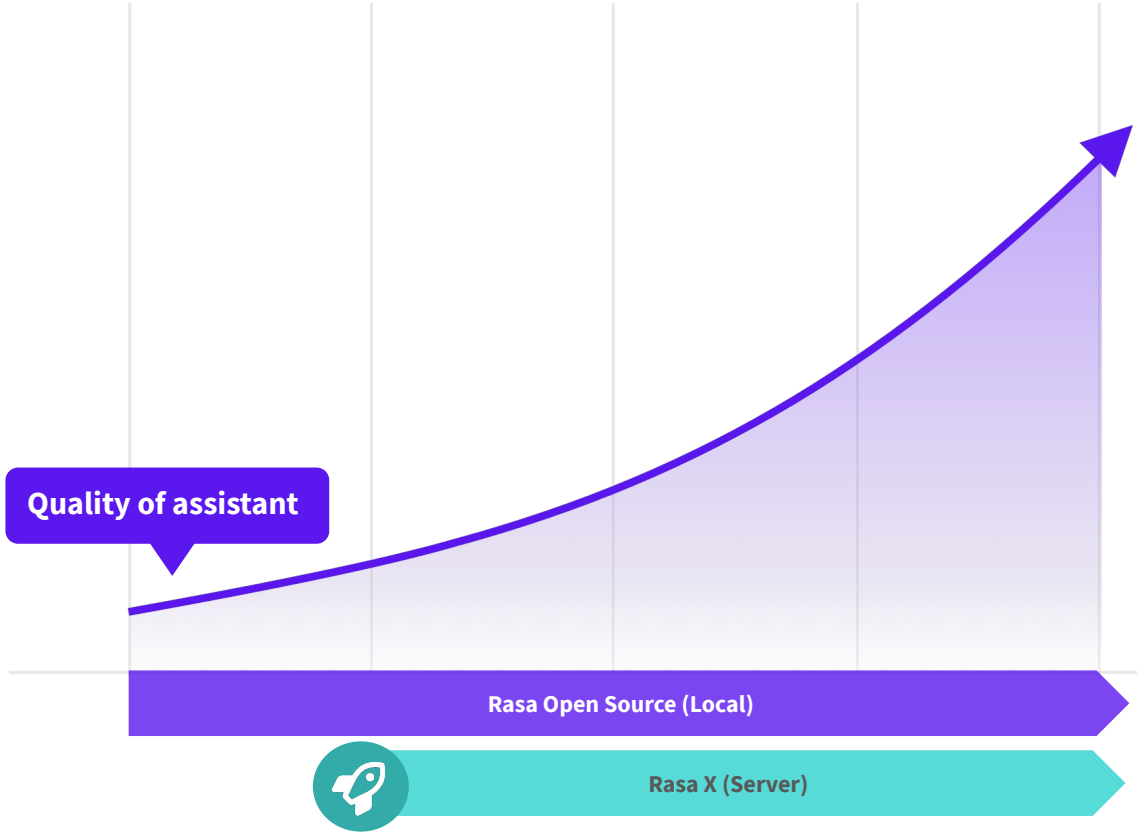
THE APPROACH

The path to a contextual assistant



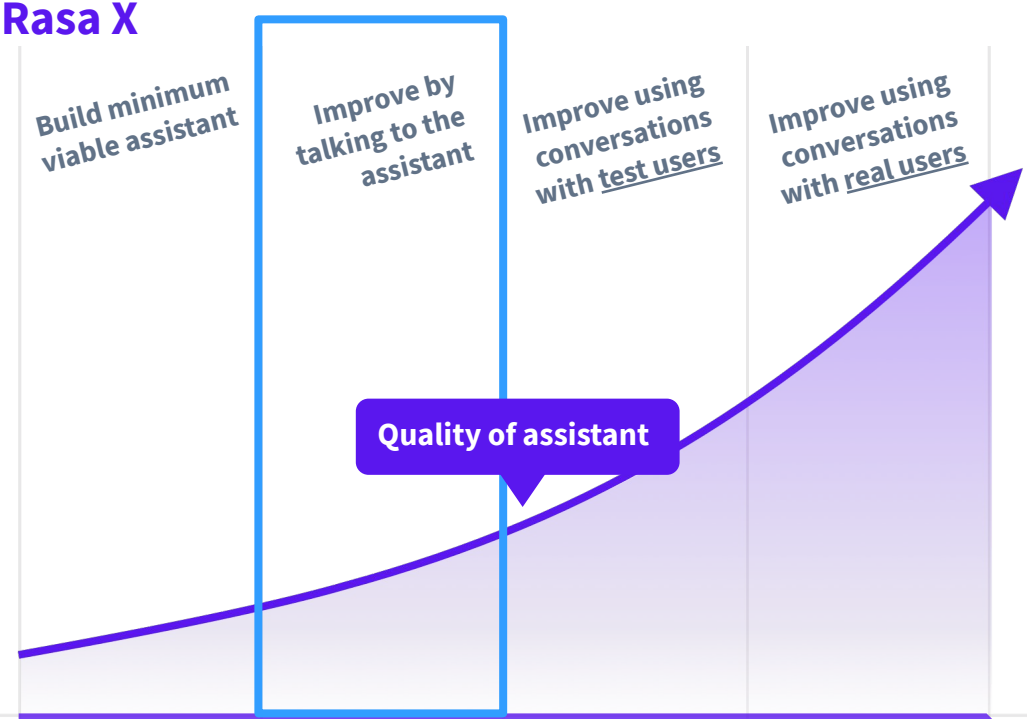
THE APPROACH

The path to a contextual assistant



Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X

Talk to your bot



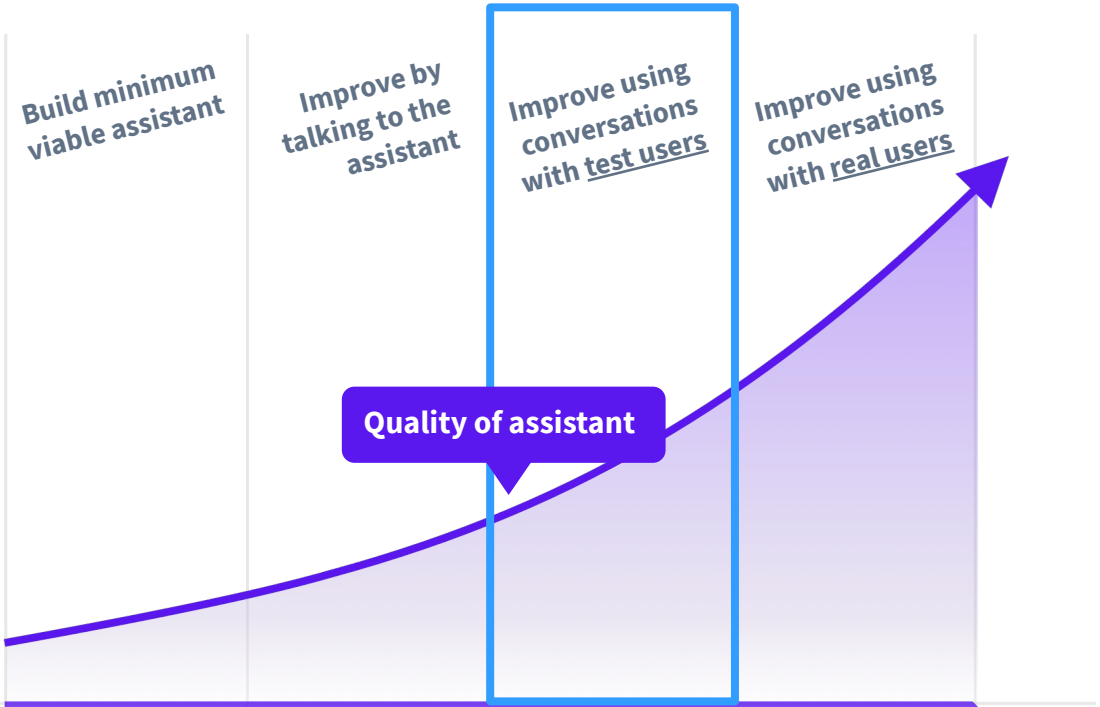
Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

RASA

Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X

Share your bot

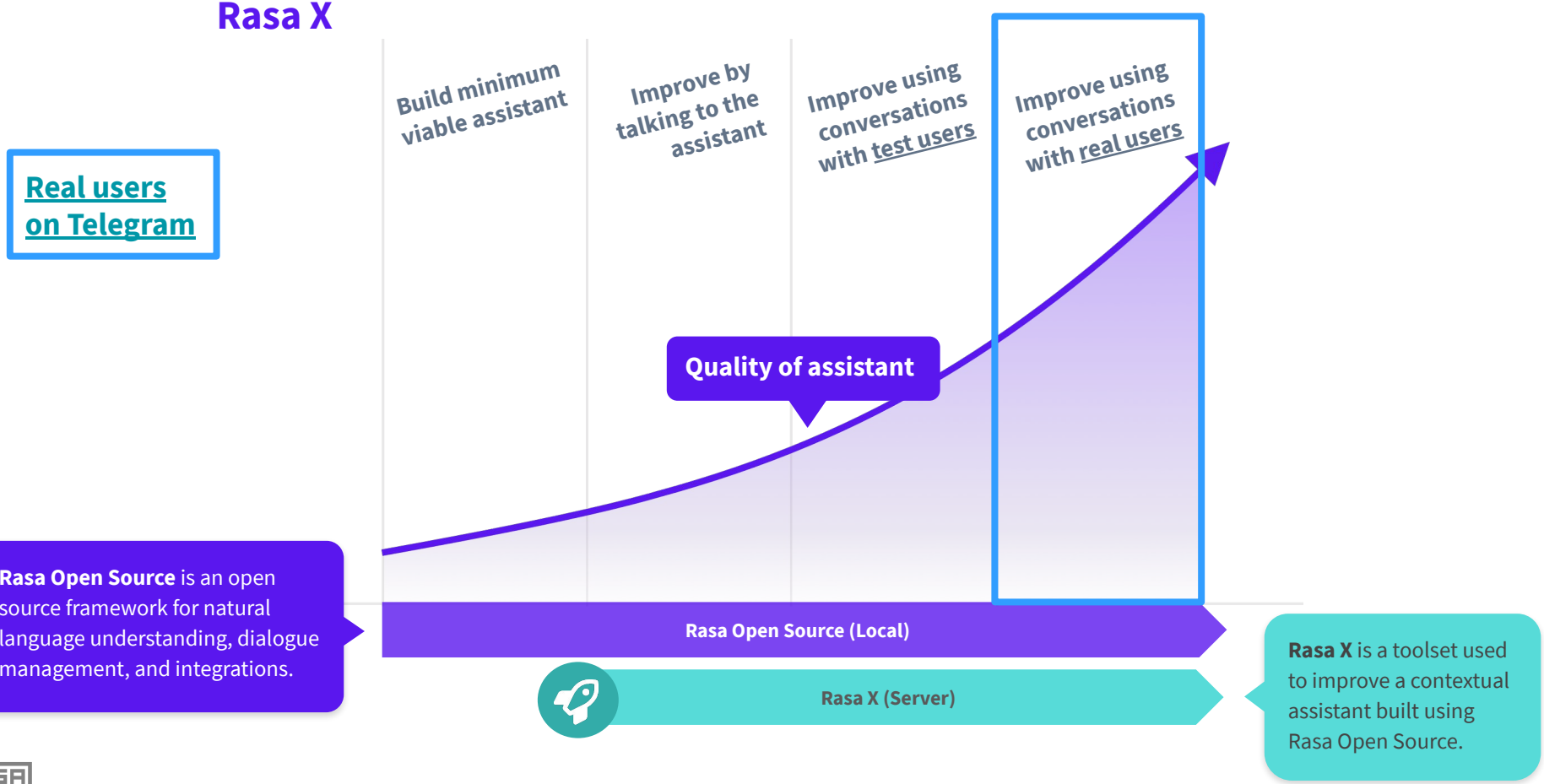


Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

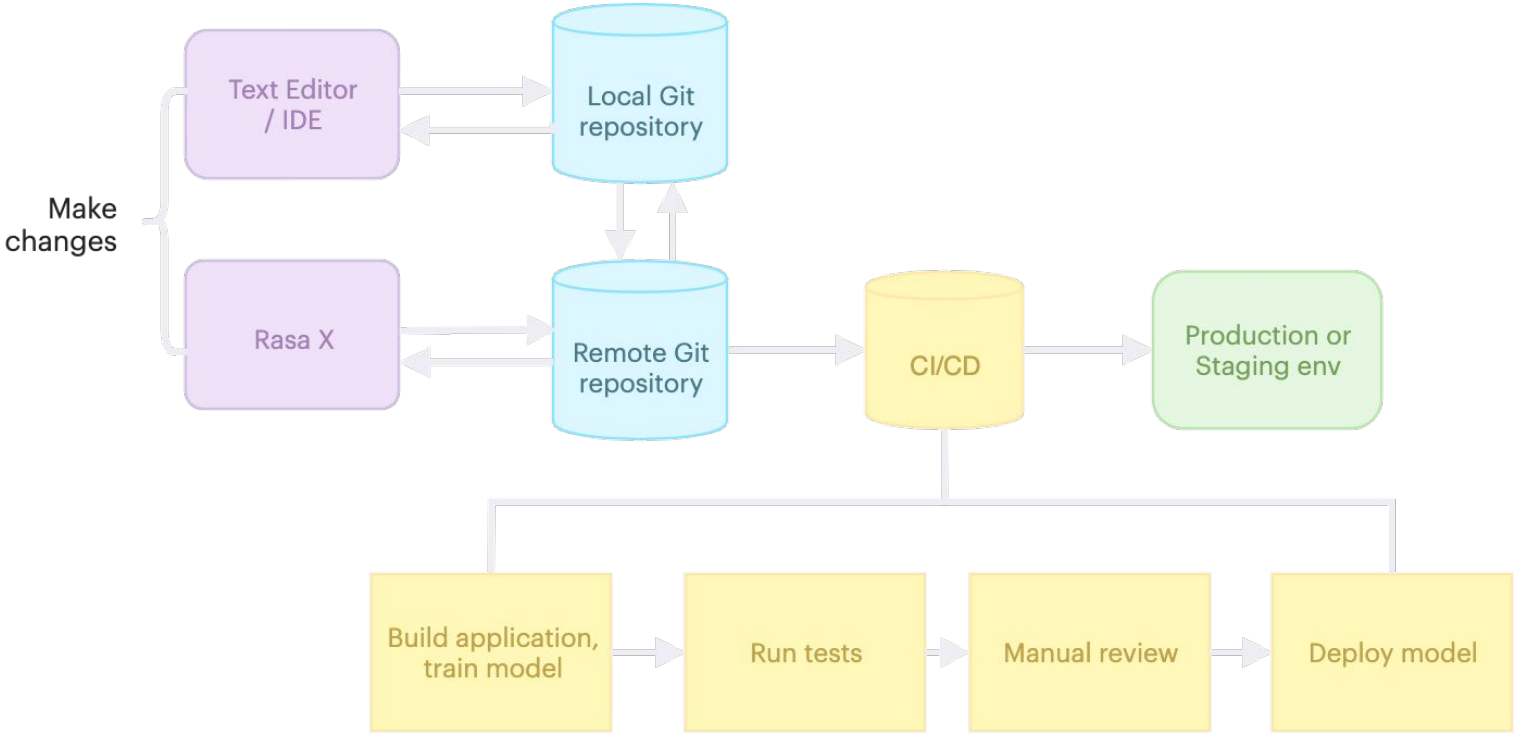
RASA

Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X



DEVELOPMENT WORKFLOW

End-to-end workflow for managing assistant updates



Recap

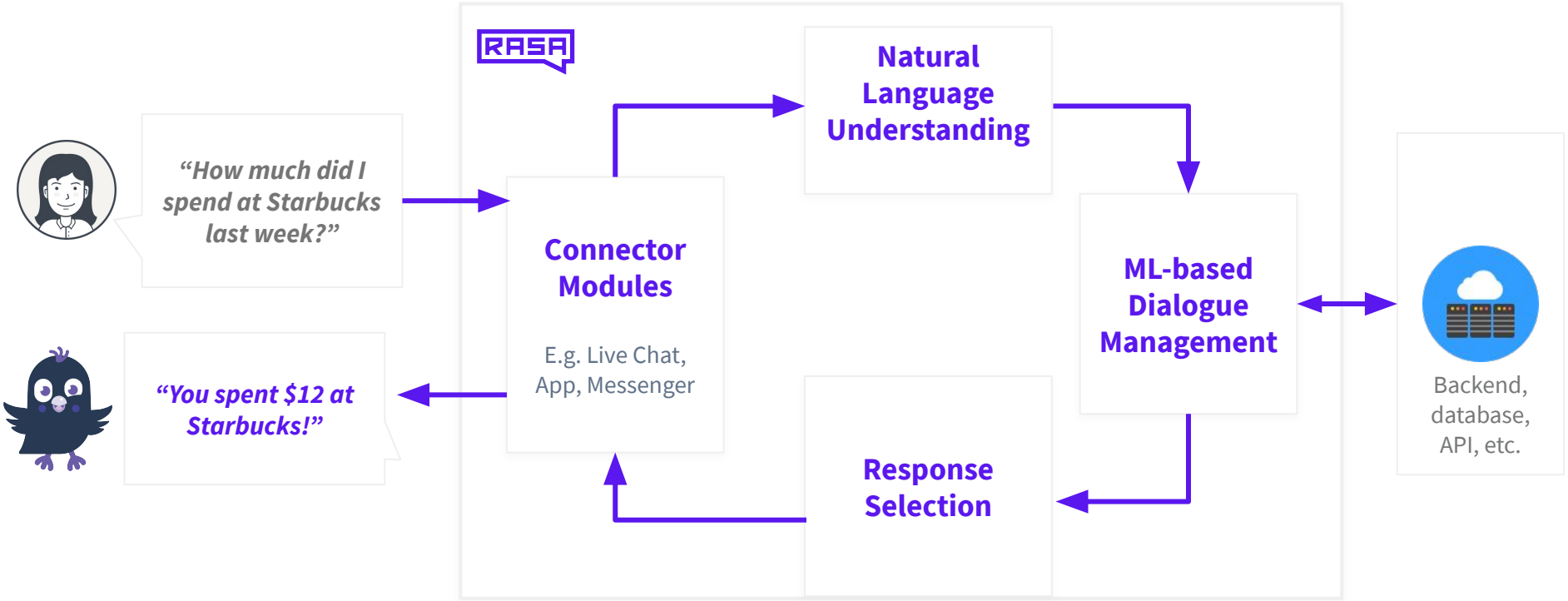
RECAP

What we've learned so far

- Set up
- Deep dive of NLU and dialogue management
- Build an MVP assistant
- Share your assistant with the outside world
- Make iterative improvements and take your assistant to the next level

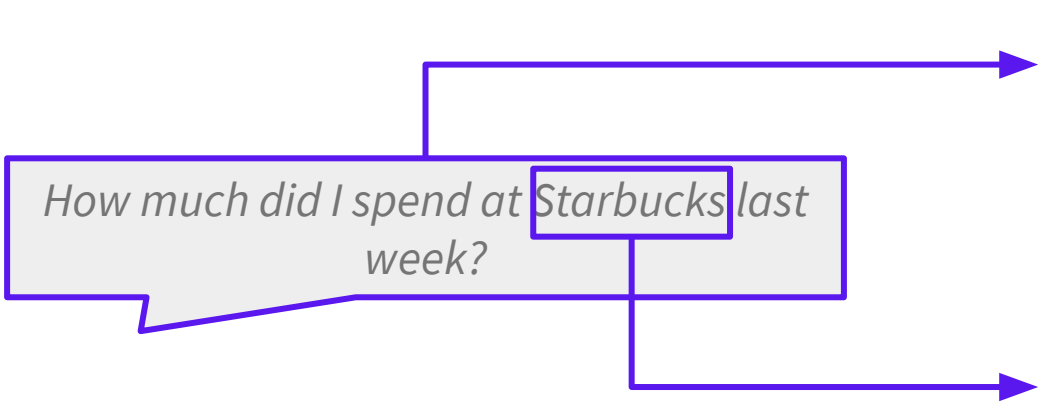
RASA OPEN SOURCE

Rasa Open Source



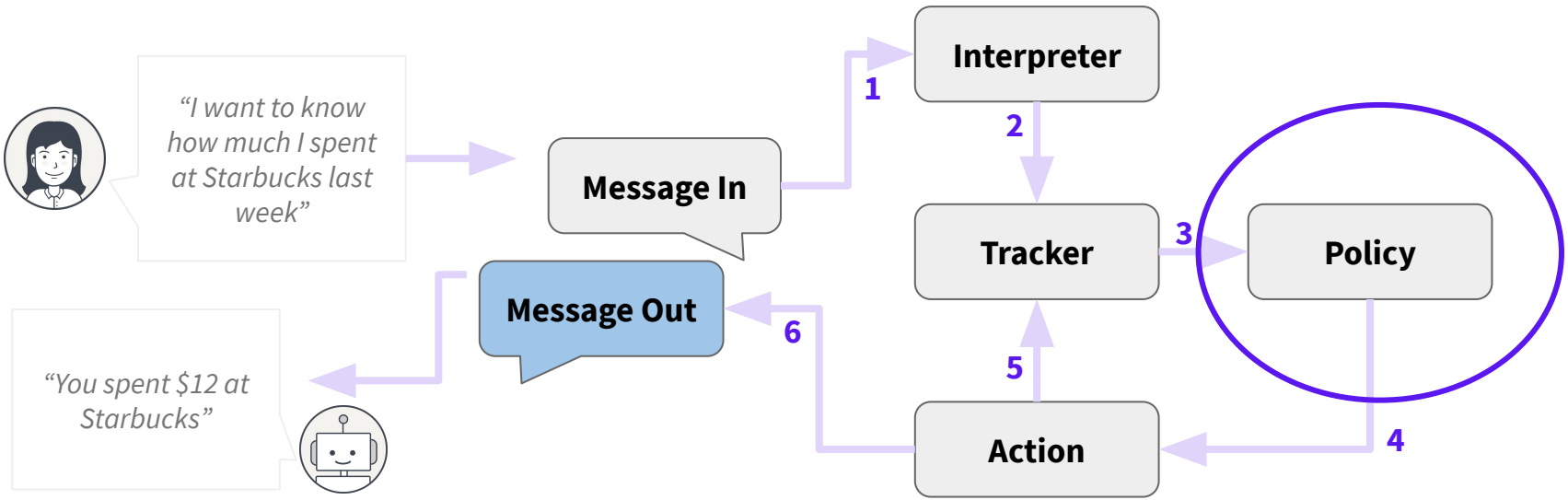
Intents and Entities

Two of the most common and necessary types of information to extract from a message



Policies

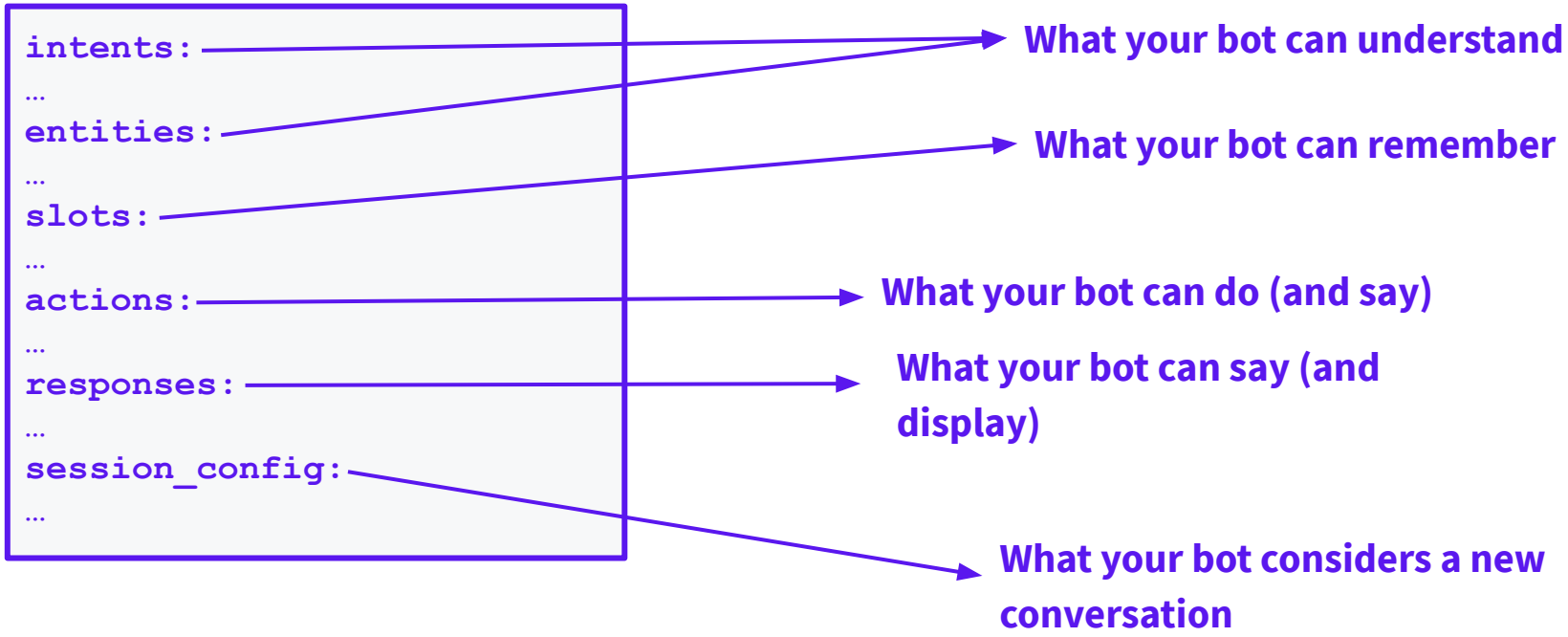
- Decide which action to take at every step in the conversation
- Each policy predicts an **action** with some **probability.** This is called **core confidence**.



Core Deep Dive

Domain

Defines the “world” of your assistant - what it knows, can understand, and can do



Multiple Policies

- The policy with the **highest confidence** wins.
- If the confidence of two policies is equal, the policy with the **highest priority** wins.

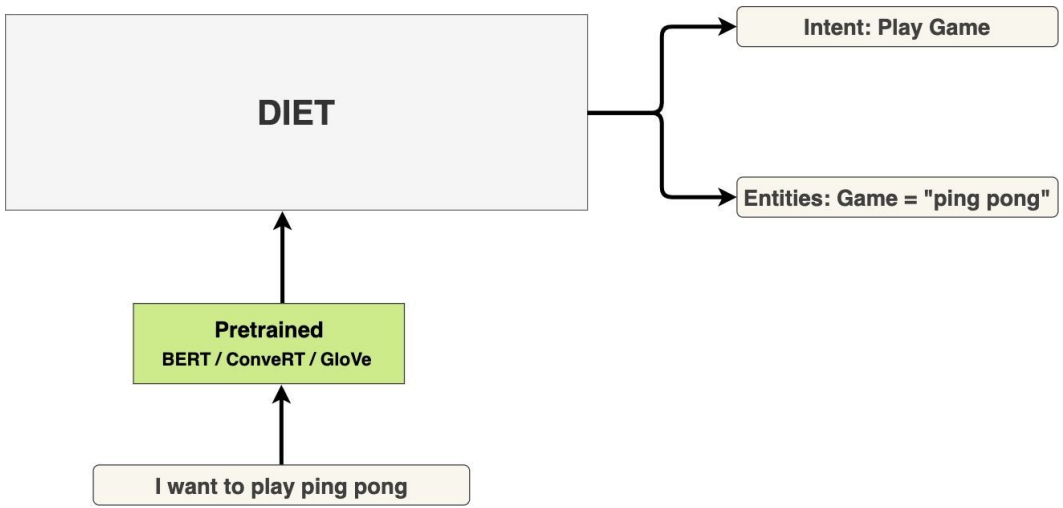
Rule-based policies have higher priority than ML-based policies

```
Policy priorities
(higher numbers = higher priority)

5. FormPolicy
4. FallbackPolicy, TwoStageFallbackPolicy
3. MemoizationPolicy, AugmentedMemoizationPolicy
2. MappingPolicy
1. EmbeddingPolicy, TEDPolicy
```

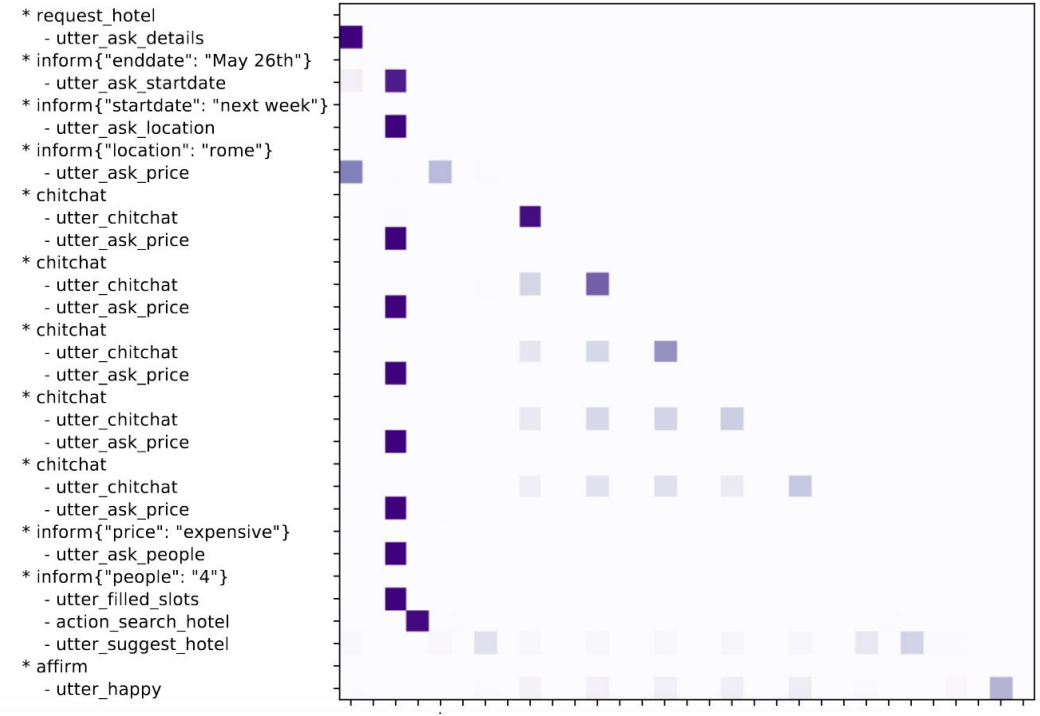
DIET is our new neural network architecture for NLU

What is DIET?

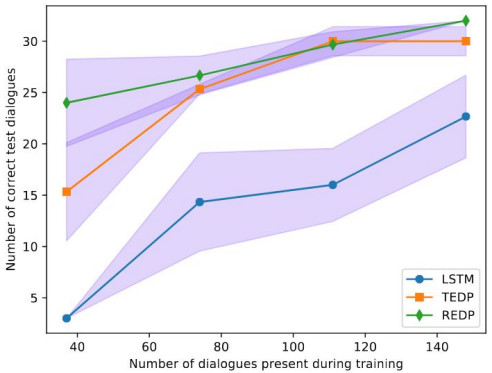
- New state of the art neural network architecture for NLU
- Predicts intents and entities together
- Plug and play pretrained language models



We found out that the Transformer Embedding Dialogue policy can untangle sub-dialogues



paper



Scope Conversation

How to get started with conversation design

The assistant’s purpose

Leverage the knowledge of domain experts

Common search queries

FAQs and wikis

Machine learning models require training data that the models can generalize from

NLU needs data in the form of examples for intents

```
## intent:bot_challenge
- are you a bot?
- are you a human?
- am I talking to a bot?
- am I talking to a human?

## intent:i_like_food
- I like [apples](food)
- my friend likes [oranges](food)
- I'm a fan of [pears](food)
- do you like [coffee](food)
```

Dialogue management model needs data in the form of stories

```
## new to rasa at start
* how_to_get_started{"user_type":
"new"}
    - action_set_onboarding
    - slot{"onboarding": true}
    - utter_getstarted_new
    - utter_built_bot_before
* deny
    - utter_explain_rasa_components
    - utter_rasa_components_details
    - utter_ask_explain_nlucorex
```

What is a Minimum Viable Assistant (MVA)?

- A basic assistant that can handle the most important **happy path** stories.
  - **Happy path**: If your assistant asks a user for some information and the user provides it, we call that a happy path.
  - **Unhappy path:** All the possible edge cases of the bot

Testing

Use the Rasa CLI to test your assistant

End to End Evaluation

Run through test conversations to make sure that both NLU and Core make correct predictions.

```
$ rasa test
```

NLU Evaluation

Split data into a test set or estimate how well your model generalizes using cross-validation.

```
$ rasa test nlu -u
data/nlu.md --config
config.yml
--cross-validation
```

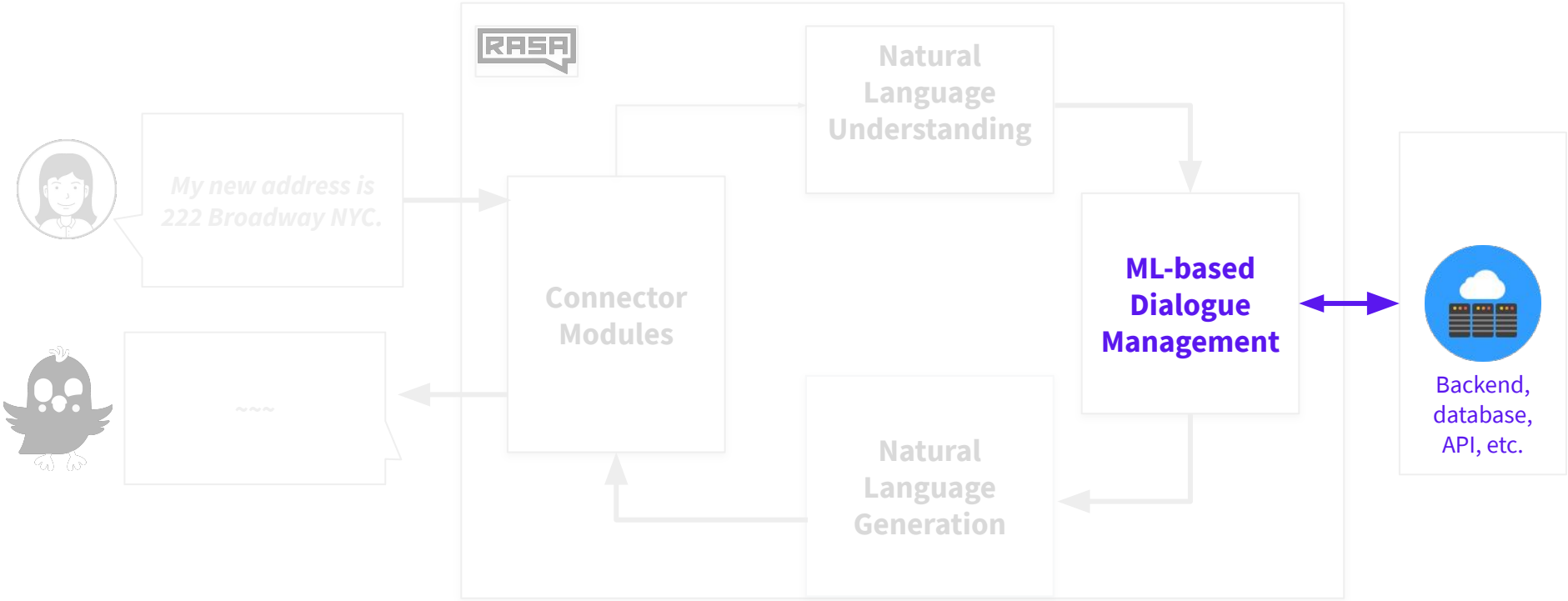
Core Evaluation

Evaluate your trained model on a set of test stories and generate a confusion matrix.

```
$ rasa test core
--stories
test_stories.md --out
results
```

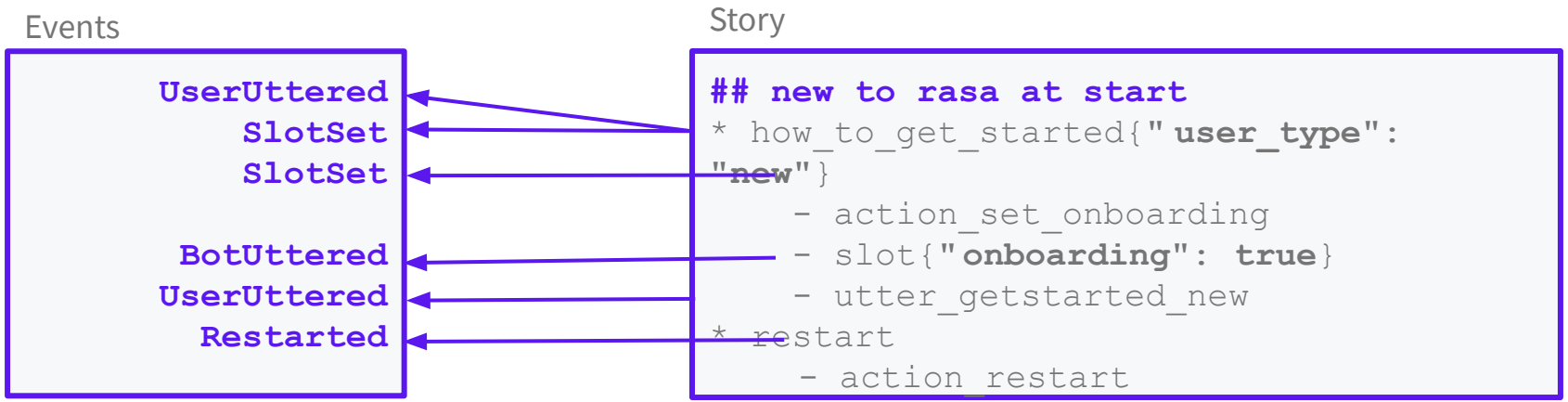
Core Deep Dive

Custom Actions: Connecting to the outside world



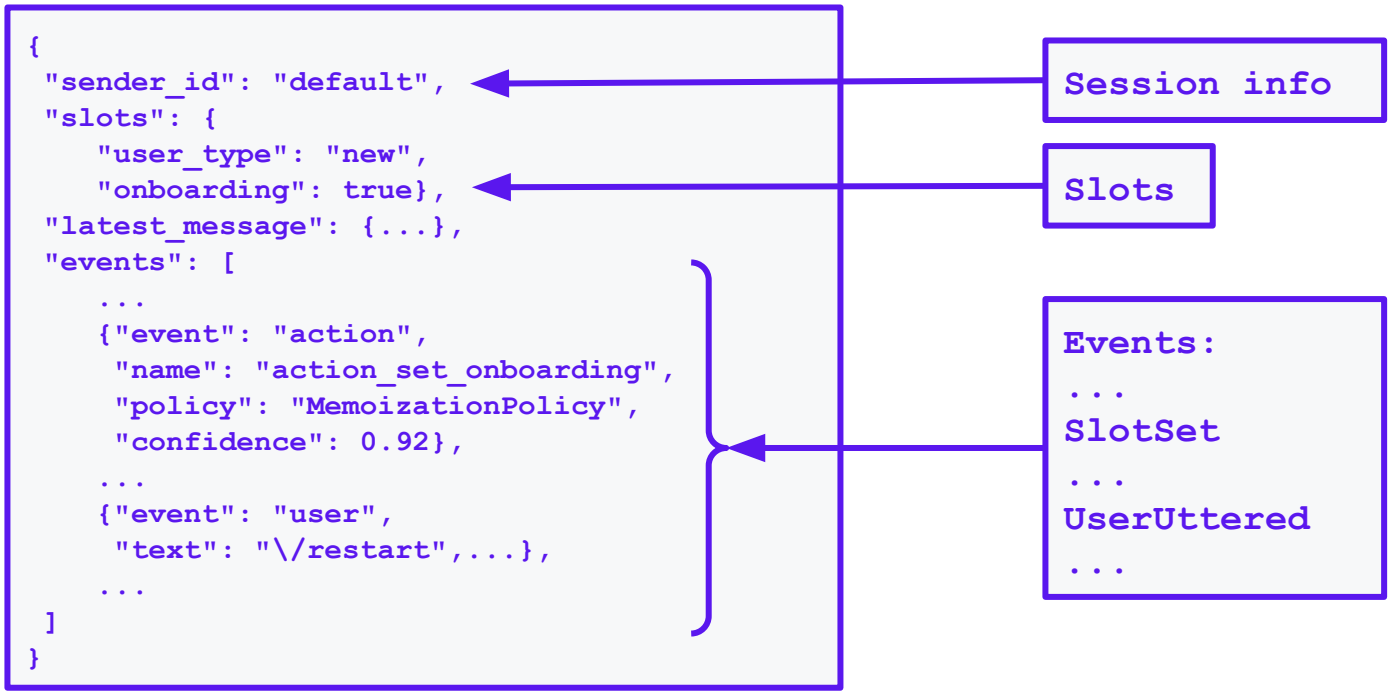
Events

- **Internally**, all conversations are represented as a sequence of **events.**
- Some events are automatically tracked



Tracker

- Trackers maintain the **state of a dialogue** between the assistant and the user.
- It keeps track of events, slots, session info etc.



Slots

Your bot's memory

- Can store:
  - user-provided info
  - info from the outside world
- Can be set by:
  - NLU (from extracted entities, or buttons)
  - Custom Actions
- Can be configured to affect or not affect the dialogue progression

Actions

Things your bot runs in response to user input.

Four different action types:

- **Utterance actions**: `utter_`
  - send a specific message to the user
  - specified in `responses` section of the domain
- **Retrieval actions**: `respond_`
  - send a message selected by a retrieval model
- **Custom actions**: `action_`
  - run arbitrary code and send any number of messages (or none).
  - return events
- **Default actions**:
  - built-in implementations available but can be overridden
  - E.g. action_listen, action_restart, action_default_fallback

Custom Actions: Examples

Custom actions can do whatever you want them to! e.g.

- **Send multiple messages**
- **Query a database**
- **Make an API call to another service**
- **Return events e.g.**
  - Set a slot
    - e.g. based on a database query)
  - Force a follow up action
    - force a specific action to be executed next
  - Revert a user utterance
    - I.e. remove a user utterance from the tracker

Forms

The structure of the form

```
## pay credit card happy path
* greet
    - utter_greet
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
    - form{"name": null}
```

User input

Form action

Form activated

Form deactivated

Forms

Handling unhappy paths

Users are not always cooperative - they can change their mind or interrupt the form.

- Use action `action_deactivate_form` to handle the situation where users decide not to proceed with the form:

```
## chitchat
* request_restaurant
    - restaurant_form
    - form{"name": "restaurant_form"}
* stop
    - utter_ask_continue
* deny
    - action_deactivate_form
    - form{"name": null}
```

Rasa X turns conversations into training data

Review 👁 Annotate ✏️ Improve 💪

Real Conversations > Synthetic Data

During development, training data is created by:

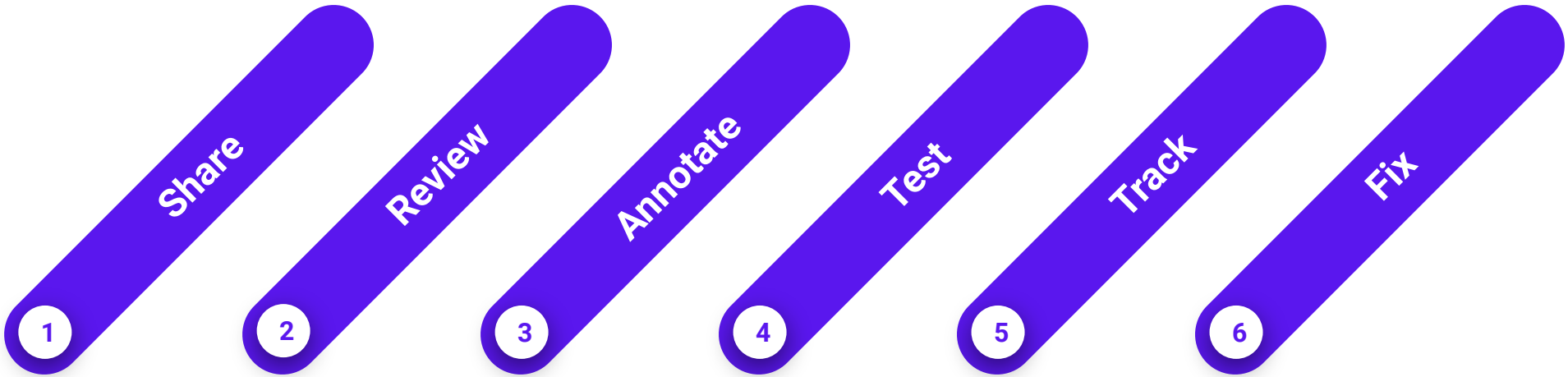
- Writing down hypothetical conversations and training examples
- Studying existing chat transcripts or knowledge bases
- Consulting subject matter experts

That's a great start, but all of those scenarios are based on human:human conversations.

Human:bot conversations have different patterns - users say different things when they are talking to a bot.

The best training data is generated from the assistant's
actual conversations

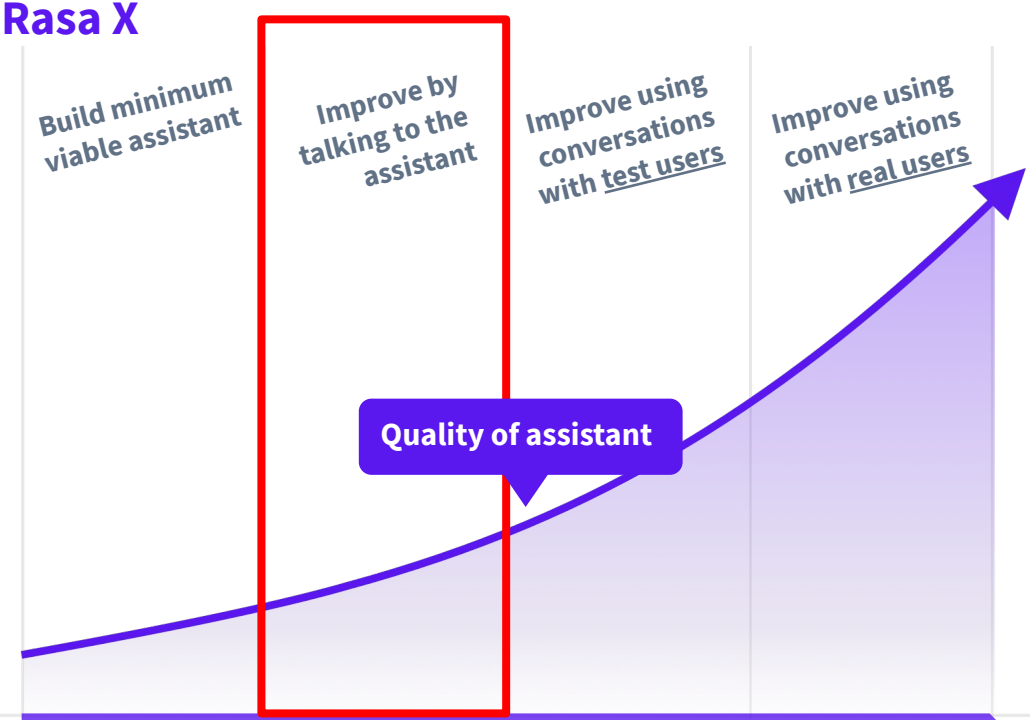
Conversation-Driven Development



RASA

Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X

Talk to your bot



Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

RASA

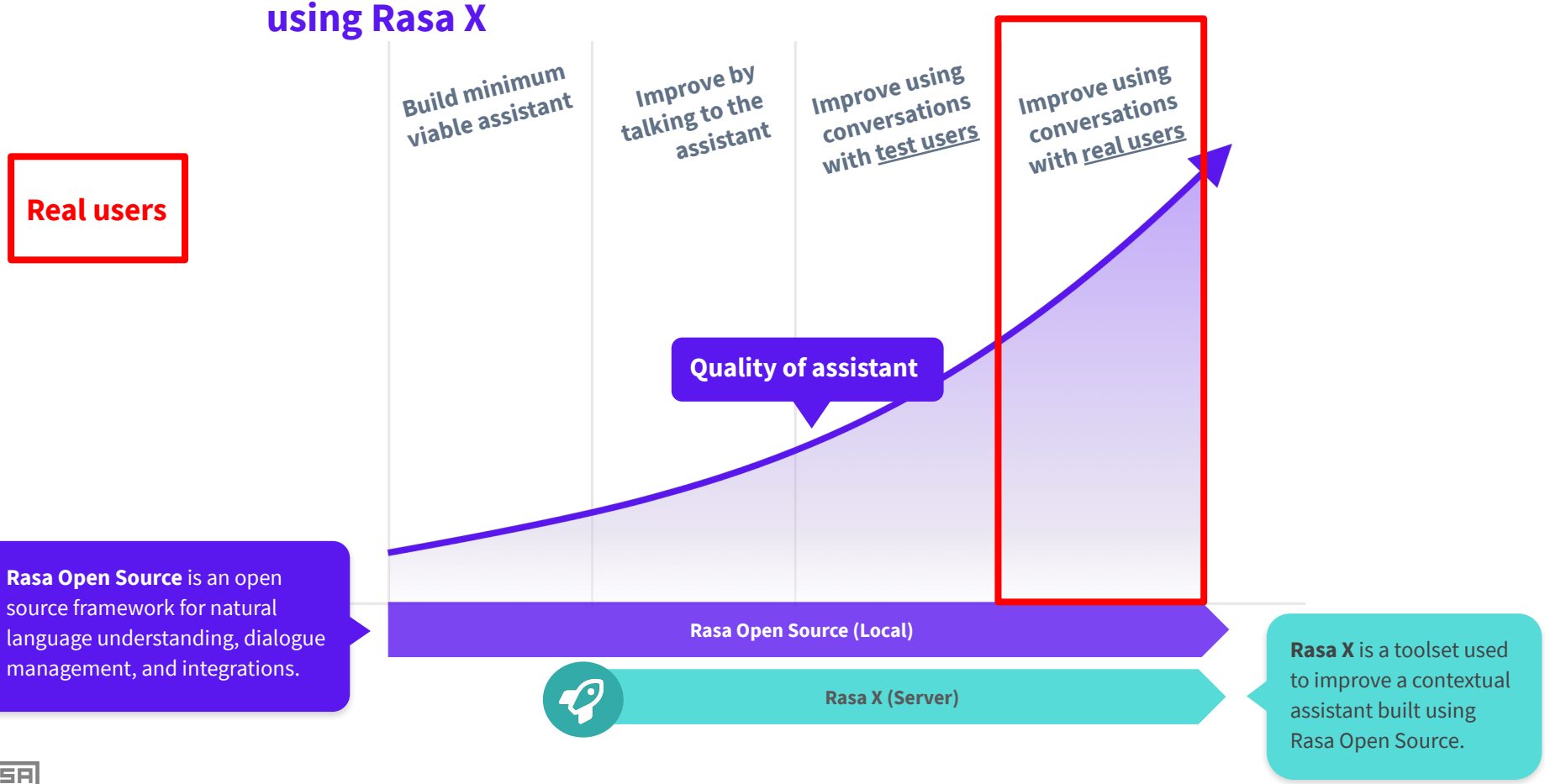
Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X

Share your bot

Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

Build a minimum viable assistant with Rasa Open Source + improve it using Rasa X



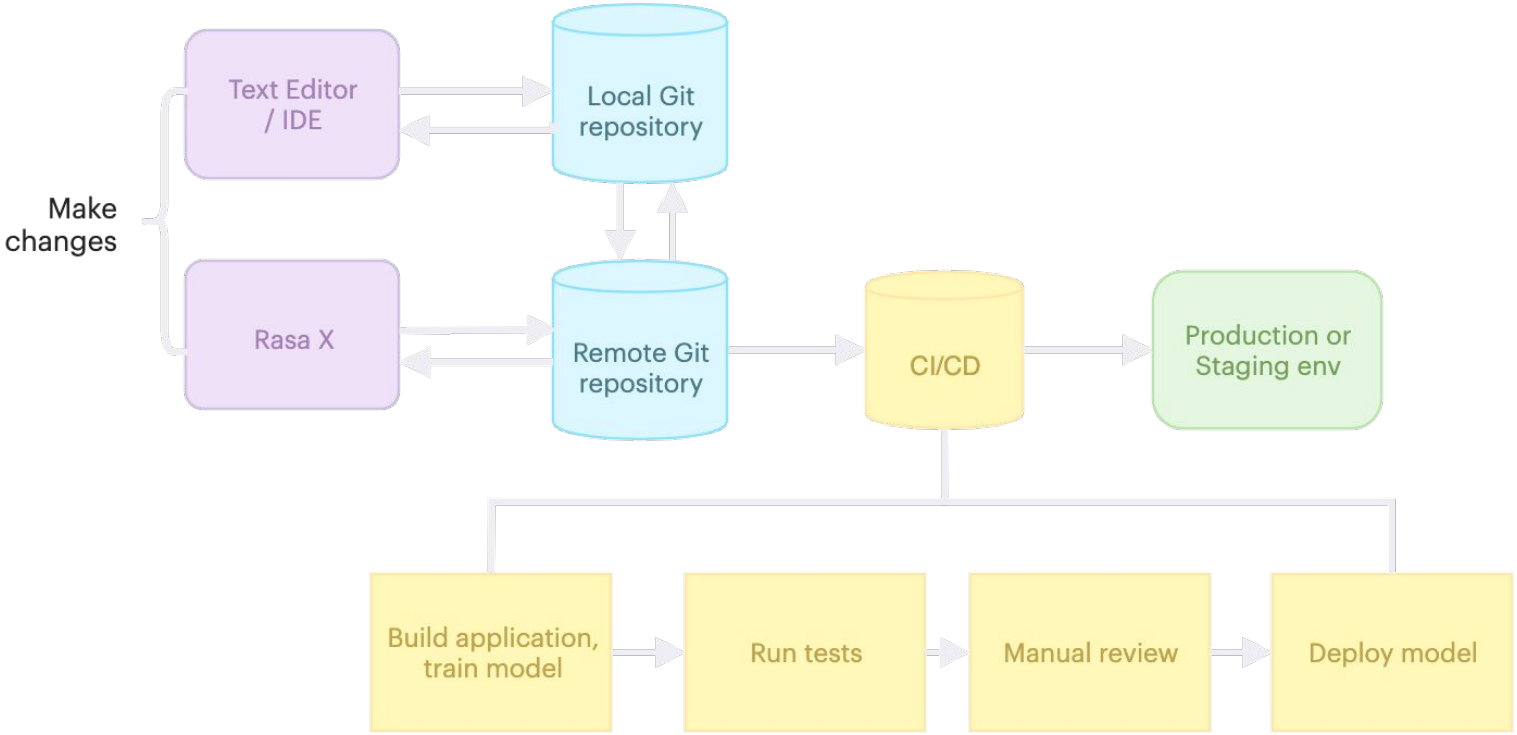
Rasa Open Source is an open source framework for natural language understanding, dialogue management, and integrations.

Rasa X is a toolset used to improve a contextual assistant built using Rasa Open Source.

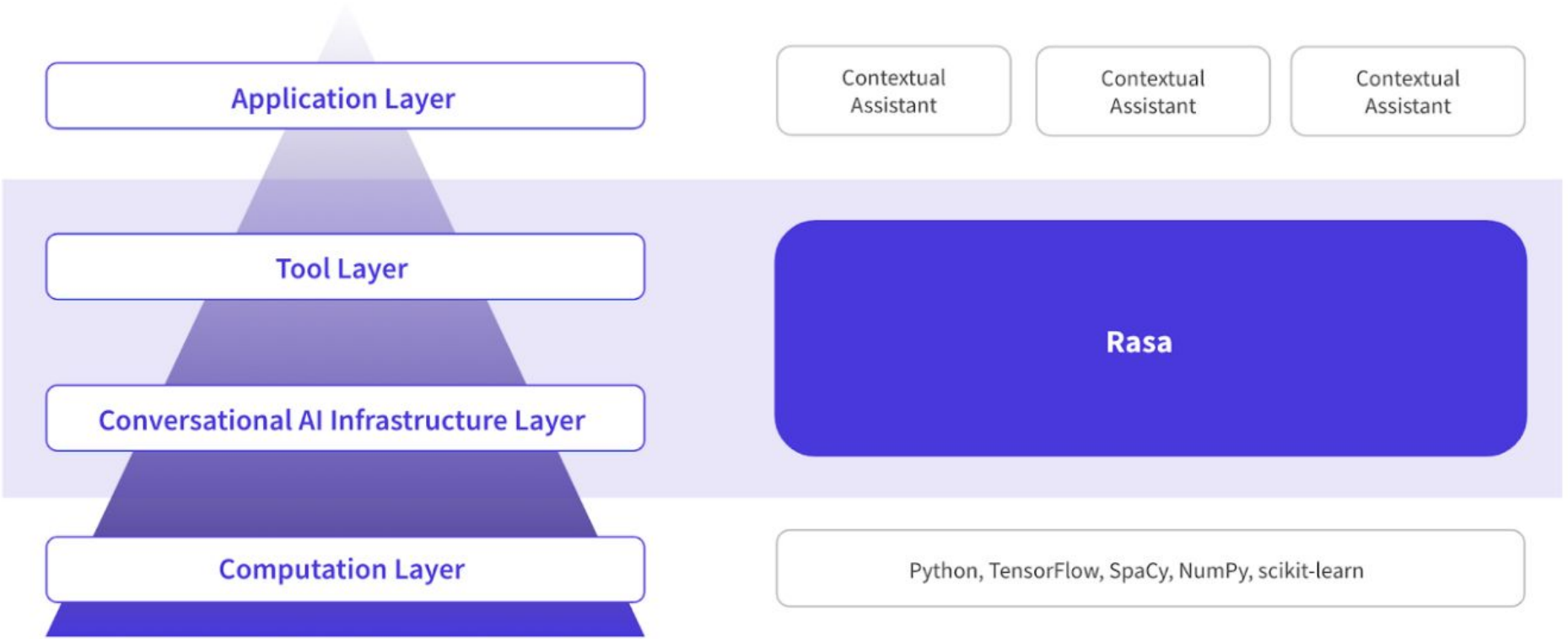
RASA

DEVELOPMENT WORKFLOW

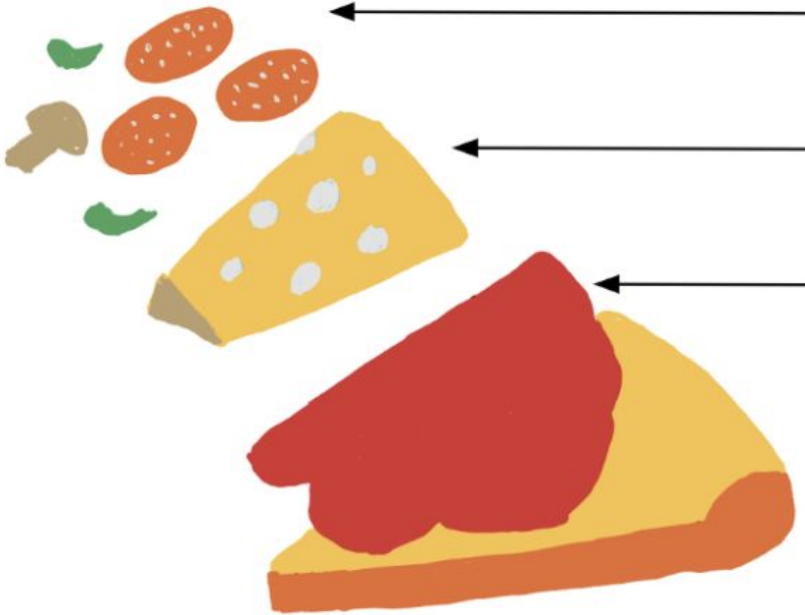
End-to-end workflow for managing assistant updates



How Rasa fits into your stack



How Rasa fits into your stack



Application layer

Tools/services layer,
ex. Rasa X

Conversational AI
infrastructure layer,
ex. Rasa Open
Source

Computation layer,
ex. Python,
TensorFlow