# Build a conversational language understanding model

## 1. Introduction

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/1-introduction

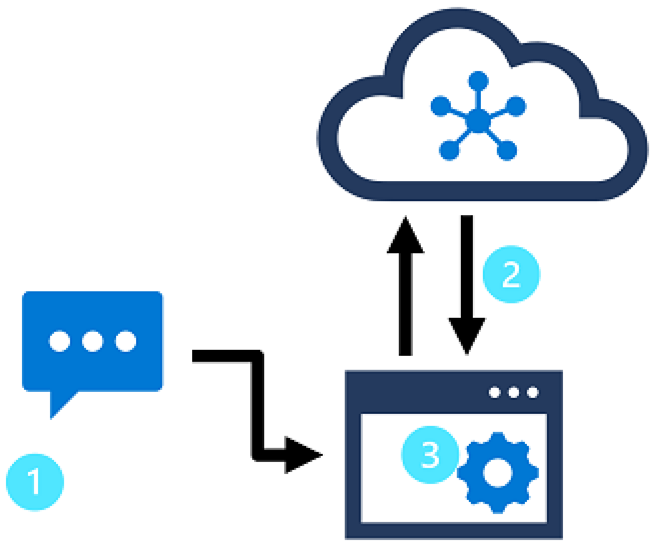
## Introduction

Completed

- 2 minutes

*Natural language processing* (NLP) is a common AI problem in which software must be able to work with text or speech in the natural language form that a human user would write or speak. Within the broader area of NLP, *natural language understanding* (NLU) deals with the problem of determining semantic meaning from natural language - usually by using a trained language model.

A common design pattern for a natural language understanding solution looks like this:



In this design pattern:

1. An app accepts natural language input from a user.
2. A language model is used to determine semantic meaning (the user's *intent*).
3. The app performs an appropriate action.

**Azure Language** enables developers to build apps based on language models that can be trained with a relatively small number of samples to discern a user's intended meaning.

In this module, you'll learn how to use the service to create a natural language understanding app using Azure Language.

After completing this module, you'll be able to:

- Provision an Azure Language resource.
- Define intents, entities, and utterances.
- Use patterns to differentiate similar utterances.
- Use pre-built entity components.
- Train, test, publish, and review a model.

## 2. Understand prebuilt capabilities of the Azure Language service

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/2a-understand-prebuilt-capabilities

# Understand prebuilt capabilities of the Azure Language service

Completed

- 6 minutes

The Azure Language service provides various features for understanding human language. You can use each feature to better communicate with users, better understand incoming communication, or use them together to provide more insight into what the user is saying, intending, and asking about.

Azure Language service features fall into two categories: Pre-configured features, and Learned features. Learned features require building and training a model to correctly predict appropriate labels, which is covered in upcoming units of this module.

This unit covers most of the capabilities of the Azure Language service, but head over to the Azure Language service documentation for a full list, including quickstarts and a full explanation of everything available.

Using these features in your app requires sending your query to the appropriate endpoint. The endpoint used to query a specific feature varies, but all of them are prefixed with the Azure Language resource you created in your Azure account, either when building your REST request or defining your client using an SDK. Examples of each can be found in the next unit.

## Pre-configured features

The Azure Language service provides certain features without any model labeling or training. Once you create your resource, you can send your data and use the returned results within your app.

The following features are all pre-configured.

### Summarization

Summarization is available for both documents and conversations, and will summarize the text into key sentences that are predicted to encapsulate the input's meaning.

### Named entity recognition

Named entity recognition can extract and identify entities, such as people, places, or companies, allowing your app to recognize different types of entities for improved natural language responses.

For example, given the text "The waterfront pier is my favorite Seattle attraction", *Seattle* would be identified and categorized as a location.

### Personally identifiable information (PII) detection

PII detection allows you to identify, categorize, and redact information that could be considered sensitive, such as email addresses, home addresses, IP addresses, names, and protected health information. For example, if the text "email@contoso.com" was included in the query, the entire email address can be identified and redacted.

### Key phrase extraction

Key phrase extraction is a feature that quickly pulls the main concepts out of the provided text. For example, given the text "Text Analytics is one of the features in Foundry Tools.", the service would extract "*Foundry Tools*" and "*Text Analytics*".

### Sentiment analysis

Sentiment analysis identifies how positive or negative a string or document is. For example, given the text "Great hotel. Close to plenty of food and attractions we could walk to", the service would identify that as *positive* with a relatively high confidence score.

### Language detection

Language detection takes one or more documents, and identifies the language for each. For example, if the text of one of the documents was "Bonjour", the service would identify that as *French*.

## Learned features

Learned features require you to label data, train, and deploy your model to make it available to use in your application. These features allow you to customize what information is predicted or extracted.

> **Note**
>
> Quality of data greatly impacts the model's accuracy. Be intentional about what data is used, how well it is tagged or labeled, and how varied the training data is. For details, see recommendations for labeling data, which includes valuable guidelines for tagging data. Also see the evaluation metrics that can assist in learning where your model needs improvement.

### Conversational language understanding (CLU)

CLU is one of the core custom features offered by Azure Language. CLU helps users to build custom natural language understanding models to predict overall intent and extract important information from incoming utterances. CLU does require data to be tagged by the user to teach it how to predict intents and entities accurately.

The exercise in this module will be building a CLU model and using it in your app.

### Custom named entity recognition

Custom entity recognition takes custom labeled data and extracts specified entities from unstructured text. For example, if you have various contract documents that you want to extract involved parties from, you can train a model to recognize how to predict them.

### Custom text classification

Custom text classification enables users to classify text or documents as custom defined groups. For example, you can train a model to look at news articles and identify the category they should fall into, such as *News* or *Entertainment*.

### Question answering

Question answering is a mostly pre-configured feature that provides answers to questions provided as input. The data to answer these questions comes from documents like FAQs or manuals.

For example, say you want to make a virtual chat assistant on your company website to answer common questions. You could use a company FAQ as the input document to create the question and answer pairs. Once deployed, your chat assistant can pass input questions to the service, and get the answers as a result.

For a complete list of capabilities and how to use them, see the Azure Language documentation.

## 3. Understand resources for building a conversational language understanding model

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/2-understand-resources-for-building

# Understand resources for building a conversational language understanding model

Completed

- 9 minutes

To use the Language Understanding service to develop a NLP solution, you'll need to create a Language resource in Azure. That resource will be used for both authoring your model and processing prediction requests from client applications.

**Tip**

This module's lab covers building a model for conversational language understanding. For more focused modules on custom text classification and custom named entity recognition, see the custom solution modules in the Develop natural language solutions learning path.

## Build your model

For features that require a model for prediction, you'll need to build, train and deploy that model before using it to make a prediction. This building and training will teach the Azure Language service what to look for.

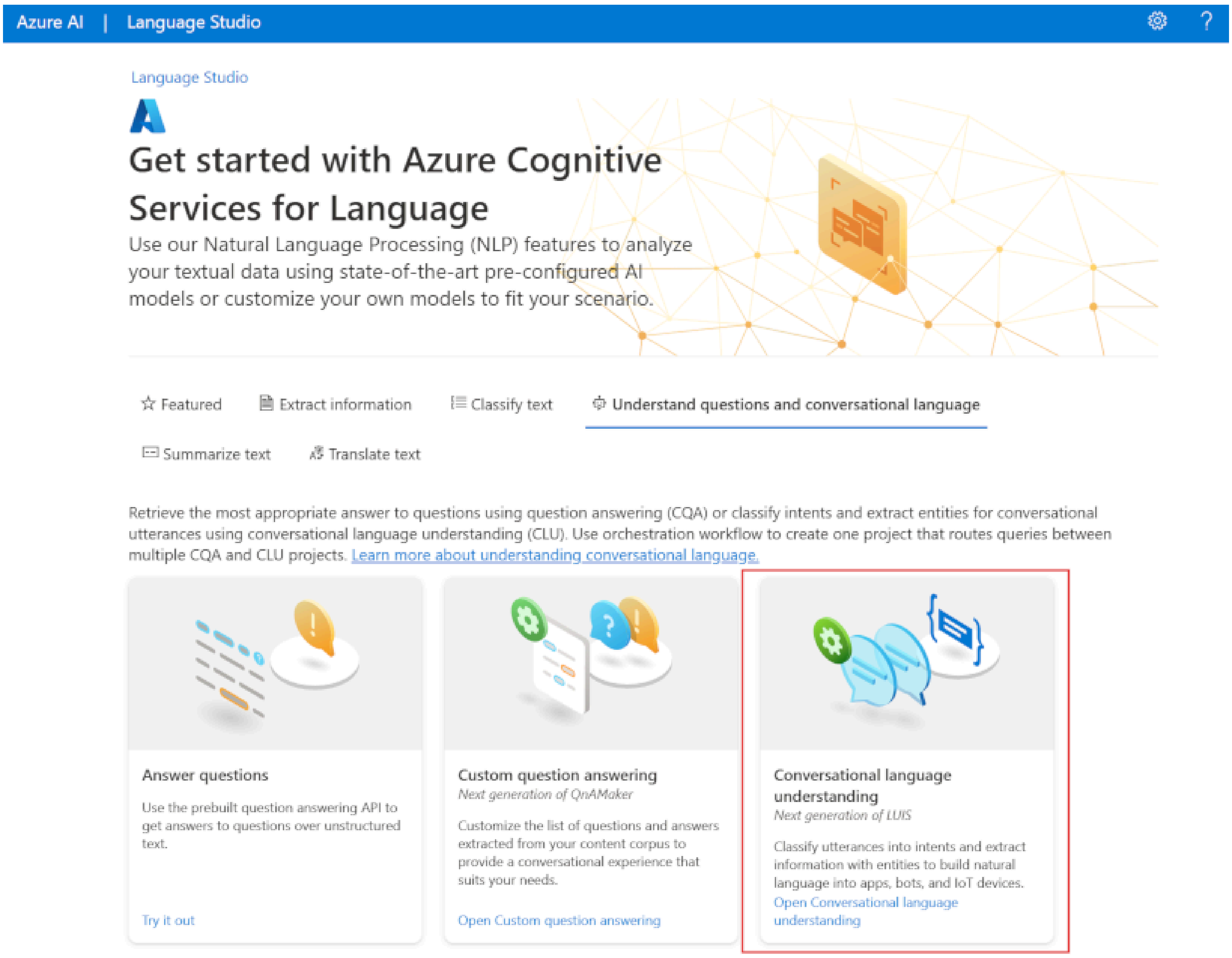
First, you'll need to create your Azure Language resource in the Azure portal. Then:

1. Search for **Foundry Tools**.
2. Find and select **Language Service**.
3. Select **Create** under the **Language Service**.
4. Fill out the necessary details, choosing the region closest to you geographically (for best performance) and giving it a unique name.

Once that resource has been created, you'll need a key and the endpoint. You can find that on the left side under **Keys and Endpoint** of the resource overview page.

### Use Language Studio

For a more visual method of building, training, and deploying your model, you can use Language Studio to achieve each of these steps. On the main page, you can choose to create a **Conversational language understanding** project. Once the project is created, then go through the same process as above to build, train, and deploy your model.


Azure AI | Language Studio

Language Studio

Get started with Azure Cognitive Services for Language

Use our Natural Language Processing (NLP) features to analyze your textual data using state-of-the-art pre-configured AI models or customize your own models to fit your scenario.

Featured | Extract information | Classify text | Understand questions and conversational language | Summarize text | Translate text

Retrieve the most appropriate answer to questions using question answering (CQA) or classify intents and extract entities for conversational utterances using conversational language understanding (CLU). Use orchestration workflow to create one project that routes queries between multiple CQA and CLU projects. Learn more about understanding conversational language.

Answer questions

Use the prebuilt question answering API to get answers to questions over unstructured text.

Try it out

Custom question answering

*Next generation of QnAMaker*

Customize the list of questions and answers extracted from your content corpus to provide a conversational experience that suits your needs.

Open Custom question answering

Conversational language understanding

*Next generation of LUIS*

Classify utterances into intents and extract information with entities to build natural language into apps, bots, and IoT devices.

Open Conversational language understanding

The lab in this module will walk through using Language Studio to build your model. If you'd like to learn more, see the [Language Studio quickstart](#)

## Use the REST API

One way to build your model is through the REST API. The pattern would be to create your project, import data, train, deploy, then use your model.

These tasks are done asynchronously; you'll need to submit a request to the appropriate URI for each step, and then send another request to get the status of that job.

For example, if you want to deploy a model for a conversational language understanding project, you'd submit the deployment job, and then check on the deployment job status.

### Authentication

For each call to your Azure Language resource, you authenticate the request by providing the following header.

| Key | Value |
|---|---|
| `Ocp-Apim-Subscription-Key` | The key to your resource |

**Request deployment**

Submit a **POST** request to the following endpoint.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMEN
```

| Placeholder | Value | Example |
|---|---|---|
| `{ENDPOINT}` | The endpoint of your Azure Language resource | `https://<your-subdomain>.cognitiveservices.azure.com` |
| `{PROJECT-NAME}` | The name for your project. This value is case-sensitive | `myProject` |
| `{DEPLOYMENT-NAME}` | The name for your deployment. This value is case-sensitive | `staging` |
| `{API-VERSION}` | The version of the API you're calling | `2022-05-01` |

Include the following `body` with your request.

```
{
  "trainedModelLabel": "{MODEL-NAME}",
}
```

| Placeholder | Value |
|---|---|
| `{MODEL-NAME}` | The model name that will be assigned to your deployment. This value is case-sensitive. |

Successfully submitting your request will receive a `202` response, with a response header of `operation-location`. This header will have a URL with which to request the status, formatted like this:

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMEN
```

**Get deployment status**

Submit a **GET** request to the URL from the response header above. The values will already be filled out based on the initial deployment request.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMEN
```

| Placeholder | Value |
|---|---|
| `{ENDPOINT}` | The endpoint for authenticating your API request |
| `{PROJECT-NAME}` | The name for your project (case-sensitive) |
| `{DEPLOYMENT-NAME}` | The name for your deployment (case-sensitive) |
| `{JOB-ID}` | The ID for locating your model's training status, found in the header value detailed above in the deployment request |
| `{API-VERSION}` | The version of the API you're calling |

The response body will give the deployment status details. The `status` field will have the value of *succeeded* when the deployment is complete.

```
{
    "jobId":"{JOB-ID}",
    "createdDateTime":"String",
    "lastUpdatedDateTime":"String",
    "expirationDateTime":"String",
    "status":"running"
}
```

For a full walkthrough of each step with example requests, see the conversational understanding quickstart.

## Query your model

To query your model for a prediction, you can use SDKs in C# or Python, or use the REST API.

### Query using SDKs

To query your model using an SDK, you first need to create your client. Once you have your client, you then use it to call the appropriate endpoint.

```
var languageClient = new TextAnalyticsClient(endpoint, credentials);
var response = languageClient.ExtractKeyPhrases(document);
```

```
language_client = TextAnalyticsClient(
            endpoint=endpoint,
            credential=credentials)
response = language_client.extract_key_phrases(documents = documents)[0]
```

Other language features, such as the conversational language understanding, require the request be built and sent differently.

```
var data = new
{
    analysisInput = new
    {
        conversationItem = new
        {
            text = userText,
            id = "1",
            participantId = "1",
        }
    },
    parameters = new
    {
        projectName,
        deploymentName,
        // Use Utf16CodeUnit for strings in .NET.
        stringIndexType = "Utf16CodeUnit",
    },
    kind = "Conversation",
};
Response response = await client.AnalyzeConversationAsync(RequestContent.Create(data));
```

```
result = client.analyze_conversation(
    task={
        "kind": "Conversation",
        "analysisInput": {
            "conversationItem": {
                "participantId": "1",
                "id": "1",
                "modality": "text",
                "language": "en",
                "text": query
            },
            "isLoggingEnabled": False
        },
        "parameters": {
            "projectName": cls_project,
            "deploymentName": deployment_slot,
            "verbose": True
        }
    }
)
```

### Query using the REST API

To query your model using REST, create a **POST** request to the appropriate URL with the appropriate body specified. For built in features such as language detection or sentiment analysis, you'll query the `analyze-text` endpoint.

**Tip**

Remember each request needs to be authenticated with your Azure Language resource key in the `Ocp-Apim-Subscription-Key` header

```
{ENDPOINT}/language/:analyze-text?api-version={API-VERSION}
```

| Placeholder | Value |
|---|---|
| `{ENDPOINT}` | The endpoint for authenticating your API request |
| `{API-VERSION}` | The version of the API you're calling |

Within the body of that request, you must specify the `kind` parameter, which tells the service what type of language understanding you're requesting.

If you want to detect the language, for example, the JSON body would look something like the following.

```
{
    "kind": "LanguageDetection",
    "parameters": {
        "modelVersion": "latest"
    },
    "analysisInput":{
        "documents":[
            {
                "id":"1",
                "text": "This is a document written in English."
            }
        ]
    }
}
```

Other language features, such as the conversational language understanding, require the request be routed to a different endpoint. For example, the conversational language understanding request would be sent to the following.

```
{ENDPOINT}/language/:analyze-conversations?api-version={API-VERSION}
```

| Placeholder | Value |
|---|---|
| `{ENDPOINT}` | The endpoint for authenticating your API request |
| `{API-VERSION}` | The version of the API you're calling |

That request would include a JSON body similar to the following.

```
{
  "kind": "Conversation",
  "analysisInput": {
    "conversationItem": {
      "id": "1",
      "participantId": "1",
      "text": "Sample text"
    }
  },
  "parameters": {
    "projectName": "{PROJECT-NAME}",
    "deploymentName": "{DEPLOYMENT-NAME}",
    "stringIndexType": "TextElement_V8"
  }
}
```

| Placeholder | Value |
|---|---|
| `{PROJECT-NAME}` | The name of the project where you built your model |
| `{DEPLOYMENT-NAME}` | The name of your deployment |

## Sample response

The query response from an SDK will in the object returned, which varies depending on the feature (such as in `response.key_phrases` or `response.Value`). The REST API will return JSON that would be similar to the following.

```json
{
    "kind": "KeyPhraseExtractionResults",
    "results": {
        "documents": [{
            "id": "1",
            "keyPhrases": ["modern medical office", "Dr. Smith", "great staff"],
            "warnings": []
        }],
        "errors": [],
        "modelVersion": "{VERSION}"
    }
}
```

For other models like conversational language understanding, a sample response to your query would be similar to the following.

```json
{
  "kind": "ConversationResult",
  "result": {
    "query": "String",
    "prediction": {
      "topIntent": "intent1",
      "projectKind": "Conversation",
      "intents": [
        {
          "category": "intent1",
          "confidenceScore": 1
        },
        {
          "category": "intent2",
          "confidenceScore": 0
        }
      ],
      "entities": [
        {
          "category": "entity1",
          "text": "text",
          "offset": 7,
          "length": 4,
          "confidenceScore": 1
        }
      ]
    }
  }
}
```

The SDKs for both Python and C# return JSON that is very similar to the REST response.

For full documentation on features, including examples and how-to guides, see the Azure Language documentation documentation pages.

## 4. Define intents, utterances, and entities

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/3-define-intents-utterances-entities

## Define intents, utterances, and entities

Completed

- 5 minutes

*Utterances* are the phrases that a user might enter when interacting with an application that uses your language model. An *intent* represents a task or action the user wants to perform, or more simply the *meaning* of an utterance. You create a model by defining intents and associating them with one or more utterances.

For example, consider the following list of intents and associated utterances:

- **GetTime**:
  - "What time is it?"
  - "What is the time?"
  - "Tell me the time"
- **GetWeather**:
  - "What is the weather forecast?"
  - "Do I need an umbrella?"
  - "Will it snow?"
- **TurnOnDevice**
  - "Turn the light on."
  - "Switch on the light."
  - "Turn on the fan"
- **None**:
  - "Hello"
  - "Goodbye"

In your model, you must define the intents that you want your model to understand, so spend some time considering the *domain* your model must support and the kinds of actions or information that users might request. In addition to the intents that you define, every model includes a **None** intent that you should use to explicitly identify utterances that a user might submit, but for which there is no specific action required (for example, conversational greetings like "hello") or that fall outside of the scope of the domain for this model.

After you've identified the intents your model must support, it's important to capture various different example utterances for each intent. Collect utterances that you think users will enter; including utterances meaning the same thing but that are constructed in different ways. Keep these guidelines in mind:

- Capture multiple different examples, or alternative ways of saying the same thing
- Vary the length of the utterances from short, to medium, to long
- Vary the location of the *noun* or *subject* of the utterance. Place it at the beginning, the end, or somewhere in between
- Use correct grammar and incorrect grammar in different utterances to offer good training data examples
- The precision, consistency and completeness of your labeled data are key factors to determining model performance.
    - Label **precisely**: Label each entity to its right type always. Only include what you want extracted, avoid unnecessary data in your labels.
    - Label **consistently**: The same entity should have the same label across all the utterances.
    - Label **completely**: Label all the instances of the entity in all your utterances.

*Entities* are used to add specific context to intents. For example, you might define a **TurnOnDevice** intent that can be applied to multiple devices, and use entities to define the different devices.

Consider the following utterances, intents, and entities:

| **Utterance** | **Intent** | **Entities** |
|---|---|---|
| What is the time? | GetTime | |
| What time is it in *London*? | GetTime | Location (London) |
| What's the weather forecast for *Paris*? | GetWeather | Location (Paris) |
| Will I need an umbrella *tonight*? | GetWeather | Time (tonight) |
| What's the forecast for *Seattle tomorrow*? | GetWeather | Location (Seattle), Time (tomorrow) |
| Turn the *light* on. | TurnOnDevice | Device (light) |
| Switch on the *fan*. | TurnOnDevice | Device (fan) |

You can split entities into a few different component types:

- **Learned** entities are the most flexible kind of entity, and should be used in most cases. You define a learned component with a suitable name, and then associate words or phrases with it in training utterances. When you train your model, it learns to match the appropriate elements in the utterances with the entity.
- **List** entities are useful when you need an entity with a specific set of possible values - for example, days of the week. You can include synonyms in a list entity definition, so you could define a **DayOfWeek** entity that includes the values "Sunday", "Monday", "Tuesday", and so on; each with synonyms like "Sun", "Mon", "Tue", and so on.
- **Prebuilt** entities are useful for common types such as numbers, datetimes, and names. For example, when prebuilt components are added, you will automatically detect values such as "6" or organizations such as "Microsoft". You can see this article for a list of supported prebuilt entities.

## 5. Use patterns to differentiate similar utterances

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/4-use-patterns-differentiate-similar-utterances

# Use patterns to differentiate similar utterances

Completed

- 2 minutes

In some cases, a model might contain multiple intents for which utterances are likely to be similar. You can use the pattern of utterances to disambiguate the intents while minimizing the number of sample utterances.

For example, consider the following utterances:

- "Turn on the kitchen light"
- "Is the kitchen light on?"
- "Turn off the kitchen light"

These utterances are syntactically similar, with only a few differences in words or punctuation. However, they represent three different intents (which could be named **TurnOnDevice**, **GetDeviceStatus**, and **TurnOffDevice**). Additionally, the intents could apply to a wide range of entity values. In addition to "kitchen light", the intent could apply to "living room light", television", or any other device that the model might need to support.

To correctly train your model, provide a handful of examples of each intent that specify the different formats of utterances.

- **TurnOnDevice**:
  - "Turn on the {DeviceName}"
  - "Switch on the {DeviceName}"
  - "Turn the {DeviceName} on"
- **GetDeviceStatus**:
  - "Is the {DeviceName} on[?]"
- **TurnOffDevice**:
  - "Turn the {DeviceName} off"
  - "Switch off the {DeviceName}"
  - "Turn off the {DeviceName}"

When you teach your model with each different type of utterance, the Azure Language service can learn how to categorize intents correctly based off format and punctuation.

## 6. Use pre-built entity components

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/5-use-pre-built-entity-components

## Use pre-built entity components

Completed

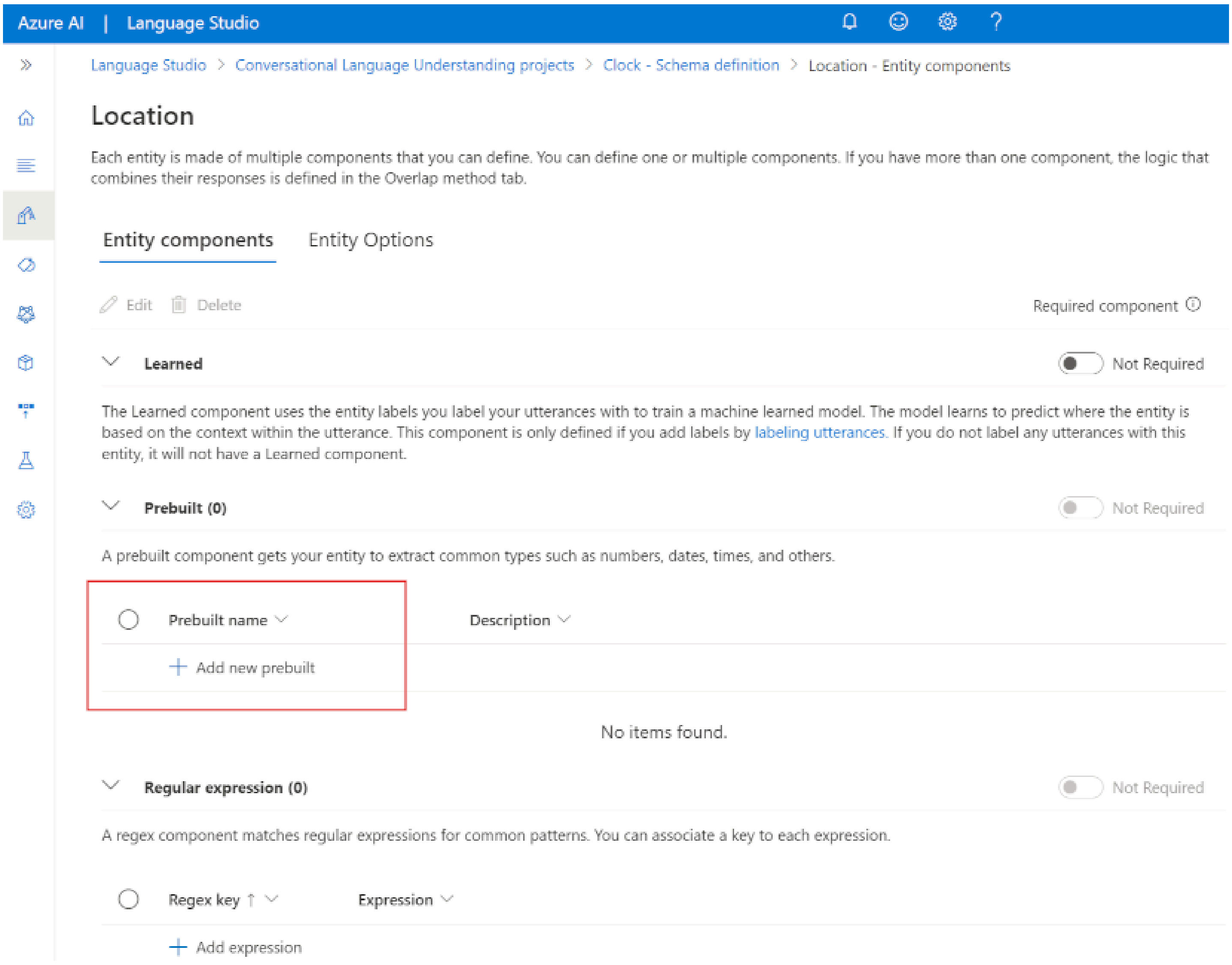
- 1 minute

You can create your own language models by defining all the intents and utterances it requires, but often you can use prebuilt components to detect common entities such as numbers, emails, URLs, or choices.

For a full list of prebuilt entities the Azure Language service can detect, see the list of supported prebuilt entity components.

Using prebuilt components allows you to let the Azure Language service automatically detect the specified type of entity, and not have to train your model with examples of that entity.

To add a prebuilt component, you can create an entity in your project, then select **Add new prebuilt** to that entity to detect certain entities.



Azure AI | Language Studio

Language Studio > Conversational Language Understanding projects > Clock - Schema definition > Location - Entity components

Location

Each entity is made of multiple components that you can define. You can define one or multiple components. If you have more than one component, the logic that combines their responses is defined in the Overlap method tab.

Entity components | Entity Options

Edit | Delete | Required component

Learned — Not Required

The Learned component uses the entity labels you label your utterances with to train a machine learned model. The model learns to predict where the entity is based on the context within the utterance. This component is only defined if you add labels by labeling utterances. If you do not label any utterances with this entity, it will not have a Learned component.

Prebuilt (0) — Not Required

A prebuilt component gets your entity to extract common types such as numbers, dates, times, and others.

| Prebuilt name | Description |
|---|---|
| + Add new prebuilt | |

No items found.

Regular expression (0) — Not Required

A regex component matches regular expressions for common patterns. You can associate a key to each expression.

| Regex key | Expression |
|---|---|
| + Add expression | |

You can have up to five prebuilt components per entity. Using prebuilt model elements can significantly reduce the time it takes to develop a conversational language understanding solution.

## 7. Train, test, publish, and review a conversational language understanding model

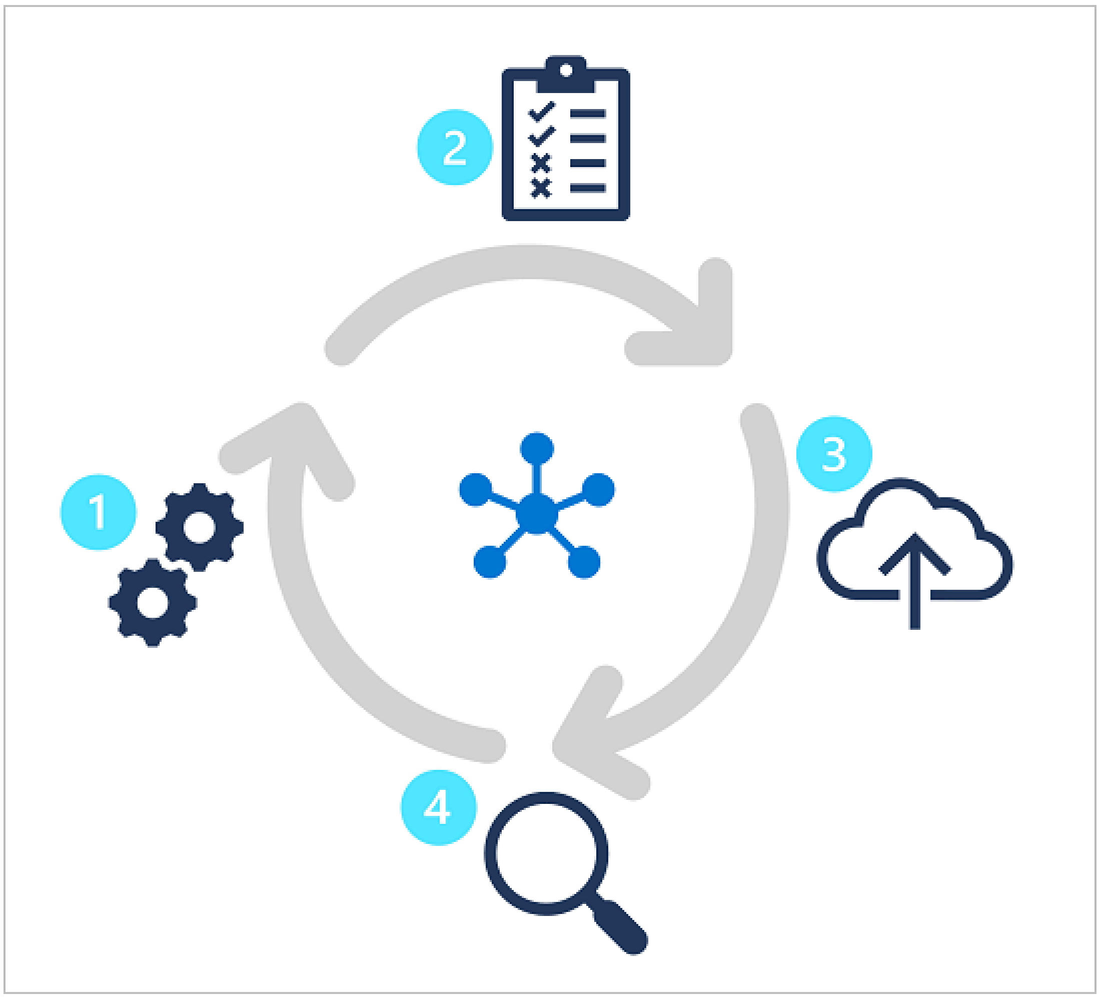
https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/6-train-test-publish-review

## Train, test, publish, and review a conversational language understanding model

Completed

- 1 minute

Creating a model is an iterative process with the following activities:



1. Train a model to learn intents and entities from sample utterances.
2. Test the model interactively or using a testing dataset with known labels
3. Deploy a trained model to a public endpoint so client apps can use it
4. Review predictions and iterate on utterances to train your model

By following this iterative approach, you can improve the language model over time based on user input, helping you develop solutions that reflect the way users indicate their intents using natural language.

## 8. Exercise - Build a conversational language understanding model

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/7-exercise

# Exercise - Build a conversational language understanding model

Completed

- 35 minutes

In this exercise, you use Azure Language to build a conversational language understanding model.

**Note**

To complete this lab, you need an **Azure subscription**.

Launch the exercise and follow the instructions.

Launch Exercise

**Tip**

After completing the exercise, if you've finished exploring Foundry Tools, delete the Azure resources that you created during the exercise.

## 9. Module assessment

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/8-knowledge-check

## Module assessment

Completed

- 3 minutes

### 10. Summary

https://learn.microsoft.com/en-us/training/modules/build-language-understanding-model/9-summary

## Summary

Completed

- 1 minute

In this module, you learned how to create a conversational language understanding model.

Now that you've completed this module, you can:

- Provision an Azure Language resource
- Define intents, entities, and utterances
- Use patterns to differentiate similar utterances
- Use pre-built entity components
- Train, test, publish, and review a model

To learn more about language understanding, see the Azure Language documentation.