Generative AI:

text - text:

eg: chatGPT

text - to- image :

eg: DALL-E

openAI:

- Code Cushman 001

- COde Davinci 002

Bigcode

- Starcoder

- StarCodeBase

- StarCoder+

Replit

Salesforce - CodeGen

Microsoft - CoPilot

AWS CodeWhisperer

uses:

requirement gathering

design and architecture

code generation

code review and refractoring

testing and quality assurance

documentation

bug detection and resolution

prompt types :

1. Zero-shot prompt:

providing a prompt/question to the AI without any prior examples. It is one of the ways to test models reasoning ability and generate

coherant answers for unfamiliar questions.

2. one-shot prompt: this technique leverages the model's ability to generalize from a single instance to generate coherent and contextually appropriate responses.

3. Few-shot prompting:When a prompt with a limited number of examples or demonstrations related to the desired task is given to an AI language model then that prompt is called as few-shot prompt.

generalize from a small amount of information and adapt its understanding to new or unseen inputs.

Benefits:

-> quickly accomplish basic coding tasks

-> more easily use unfamiliar APIs and frameworks

-> get high-quality suggestions for AWS services

-> use AI responsibly

-> improve application security

-> use your favourite tools.

top use cases:

-> accelerate time-consuming development tasks

-> quickly adopt new technologies to build complex solutions

-> adopt new services more easily and with confidence.

-> improve application security.

-------------------------------------------------------------------------

languages:

-> python, java,javascript,typescript,C#

-> C, C++,SQL,go,kotlin,Rust,Scala,PHP,Shell,Ruby

------------------------------------------------------------------------------------------------------------------------------------------------

PROMPT ENGINEERING:

-> two types of LLMs :

Base LLMs :

1. predicts next word based on text training data.

**EG 1 :Once upon a time, there was a unicorn,** that lived in a magical forest with all her unicorn friends.

**EG2: what is capital of France?**

What is France’s largest city?

What is France’s currency?

Instruction tuned LLM :

* tries to follow instructions
* fine-tune on instructions and good attempts at following those instructions.
* RLHF : Reinforcement Learning with Human Feedback.
* Eg: What is capital of France?The capital of France is Paris.

22269

16811

**Principles:**

* Pip install openai

import openai

openai.api\_key=”ak-“

Using delimeters helps to avoid prompt injections.

Prompt injection – if the user is allowed to give input to the prompt,they might give conflicting instructions to the model that might make it follow the users instructions rather than doing what you wanted it to do.

* **Write clear and specific instructions**

**Tactics:**

1. Use delimeters

Triple quotes “””

Triple backticks ‘’’

Triple dashes ---

Angle brackets <>

XML tags : <tag> </tag>

1. Ask for structured output : HTML,JSON
2. Check whether conditions are satisfied. Check assumptions required to do the task.
3. Few-shot prompting- give successful examples of completing tasks.

* **Give model time to think**

**Tactics:**

1. Specify the steps to complete a task.

Step 1

Step2

Step 3 …….

1. Instruct the model to work out its own solution before rushing to a conclusion.

**Model Limitations:**

**Hallucination:**

Makes statements that sound plausible but are not true**.**

**Reducing hallucinations:**

First find relevant information,

Then answer the question based on the relevant information.

**Iterative Prompt Development:**

**Prompt guidelines:**

* Be clear and specific.
* Analyse why result doesnot give desired output.
* Refine the idea and the prompt
* Repeat

**SUMMARIZING:**

**INFERRING:**

* Sentiment analysis.
* Zero shot prompting

**TRANSFORMING:**

* Translating one lang to another.

**EXPANDING:**

System (sets behavior of assistant) 🡪 assistant 🡪 user

(chat model) you

Chatgpt messages are assistant messages. Guiding assistant. User doesnot know.

**RESPONSIBLE AI:**

With Azure OpenAI, users can summarize text, get code suggestions, generate images for a web site, and much more.

**Generating natural language** - Such as: summarizing complex text for different reading levels, suggesting alternative wording for sentences, and much more.

Text competition: generate and edit text.

Embeddings: search, classify, and compare text.

**Generating code-** Such as: translating code from one programming language into another, identifying and troubleshooting bugs in code, and much more.

**Generating images -** Such as: generating images for publications from text descriptions and much more.

In machine learning, the algorithm needs to be told how to make an accurate prediction by consuming more information (for example, by performing feature extraction). In deep learning, the algorithm can learn how to make an accurate prediction through its own data processing, thanks to the artificial neural network structure.

Training deep learning models often requires large amounts of training data, high-end compute resources (GPU, TPU), and a longer training time. In scenarios when you don't have any of these available to you, you can shortcut the training process using a technique known as transfer learning.

Transfer learning is a technique that applies knowledge gained from solving one problem to a different but related problem.

Azure OpenAI is available for Azure users and consists of four components:

* Pre-trained generative AI models
* Customization capabilities; the ability to fine-tune AI models with your own data
* Built-in tools to detect and mitigate harmful use cases so users can implement AI responsibly.
* Enterprise-grade security with role-based access control (RBAC) and private networks

Azures AI services 🡪 1. Azure machine learning 2. Cognitive services 3. Applied ai services

AzureAI services have 5 pillars : 1. Voice 2. Speech 3. Language 4. Decision 5. Azure OpenAI service.

Azure's existing Language service can be used for widely known use-cases that require minimal tuning (the process of optimizing a model's performance). Azure OpenAI's service may be more beneficial for use-cases that require highly customized generative models, or for exploratory research.

Azure Open AI models:

**GPT-4** models that represent the latest generative models for natural language and code.

**GPT-3.5** models that can generate natural language and code responses based on prompts.

**Embeddings** models that convert text to numeric vectors for analysis - for example comparing sources of text for similarity.

**DALL-E** models that generate images based on natural language descriptions.

* Azure OpenAI's natural language models are able to take in natural language and generate responses.
* Natural language learning models are trained on words or chunks of characters known as tokens.

These tokens are mapped into vectors for a machine learning model to use for training. Generative pre-trained transformer(GPT)

|  |  |
| --- | --- |
| Summarizing text | "Summarize this text into a short blurb" |
| Classifying text | "What genre of book is this?" |
| Generating names or phrases | "Write a tagline for my flower company" |
| Translation | "Translate 'How are you' to French" |
| Answering questions | "What does Azure OpenAI do?" |
| Suggesting content | "Give me the five best weddings songs" |

* **Fairness**: AI systems shouldn't make decisions that discriminate against or support bias of a group or individual.
* **Reliability and Safety**: AI systems should respond safely to new situations and potential manipulation.
* **Privacy and Security**: AI systems should be secure and respect data privacy.
* **Inclusiveness**: AI systems should empower everyone and engage people.
* **Accountability**: People must be accountable for how AI systems operate.
* **Transparency**: AI systems should have explanations so users can understand how they're built and used.

It defines a four stage process to develop and implement a plan for responsible AI when using generative models. The four stages in the process are:

1. *Identify* potential harms that are relevant to your planned solution.
2. *Measure* the presence of these harms in the outputs generated by your solution.
3. *Mitigate* the harms at multiple layers in your solution to minimize their presence and impact, and ensure transparent communication about potential risks to users.
4. *Operate* the solution responsibly by defining and following a deployment and operational readiness plan.

***IDENTIFY:***

1. Identify potential harms –

Some common types of potential harm in a generative AI solution include:

->Generating content that is offensive, pejorative, or discriminatory.

->Generating content that contains factual inaccuracies.

->Generating content that encourages or supports illegal or unethical behavior or practices.

1. Prioritize identified harms

* prioritize the harms with the most likely and impactful harms first
* The prioritization must take into account the intended use of the solution as well as the potential for misuse; and can be subjective. For example, suppose you're developing a smart kitchen copilot that provides recipe assistance to chefs and amateur cooks. Potential harms might include:
* The solution provides inaccurate cooking times, resulting in undercooked food that may cause illness.
* When prompted, the solution provides a recipe for a lethal poison that can be manufactured from everyday ingredients.

1. Test and verify the prioritized harms-

Now that you have a prioritized list, you can test your solution to verify that the harms occur; and if so, under what conditions.

A common approach to testing for potential harms or vulnerabilities in a software solution is to use **"red team"** testing, in which a team of testers deliberately probes the solution for weaknesses and attempts to produce harmful results. Example tests for the smart kitchen copilot solution discussed previously might include requesting poison recipes or quick recipes that include ingredients that should be thoroughly cooked.

1. Document and share the verified harms.

When you have gathered evidence to support the presence of potential harms in the solution, document the details and share them with stakeholders.

* Red teaming is a strategy that is often used to find security vulnerabilities or other weaknesses that can compromise the integrity of a software solution.

**MEASURE**

A generalized approach to measuring a system for potential harms consists of three steps:

1. Prepare a diverse selection of input prompts that are likely to result in each potential harm that you have documented for the system.
2. Submit the prompts to the system and retrieve the generated output.
3. Apply pre-defined criteria to evaluate the output and categorize it according to the level of potential harm it contains. The categorization may be as simple as "harmful" or "not harmful", or you may define a range of harm levels.

**MITIGATE**

Mitigation of potential harms in a generative AI solution involves a layered approach:

* **Model** Selecting a model that is appropriate for the intended solution use. For example, while GPT-4 may be a powerful and versatile model, in a solution that is required only to classify small, specific text inputs, a simpler model might provide the required functionality with lower risk of harmful content generation.
* ***Fine-tuning*** a foundational model with your own training data so that the responses it generates it makes are more likely to be relevant and scoped to your solution scenario.
* **Safety System:**

The safety system layer includes platform-level configurations and capabilities that help mitigate harm. For example, Azure OpenAI Service includes support for content filters that apply criteria to suppress prompts and responses based on classification of content into four severity levels (safe, low, medium, and high) for four categories of potential harm (hate, sexual, violence, and self-harm).

1. **Application**

The application layer is the software application through which users interact with the generative AI model. Harm mitigation techniques at this layer include:

* Specifying ***metaprompts***or system inputs that define behavioral parameters for the model.
* Applying prompt engineering techniques to add context to input prompts,
  + Citing sources of information in the generated output (or explicitly noting that no citation has been provided)

1. **Positioning:**

->The positioning layer includes any documentation or other user collateral that describes the use of the solution to its users and stakeholders.

-> Documentation and other descriptions of a generative AI solution should be appropriately transparent about the capabilities and limitations of the system, the models on which it's based, and any potential harms that may not always be addressed by the mitigation measures you have put in place.

Common compliance reviews include:

* Legal
* Privacy
* Security
* Accessibility
* phased delivery plan
* incident response plan
* rollback plan

CODY :

Sourcegraph large language model

Your editor

**Look up context Cody**

Code intelligence graph

* Cody today lives in your editor. It will also soon live in source graph UI.User will ask cody a question, and then cady take that query and then query Sourcegraph, called as Code intelligence graph-> summation of all source graphAPIs build over the years,🡪 feed the knowledge to LLM ,also feed it with user query. LLM will give a response.
* Valid and verify
* Synthesize+response.

Katalon’s NEW GPT-powered Manual Test Generator for Jira

* This new feature enables Jira users to generate test steps from a single button click within Jira issues.
* [SmartWait](https://docs.katalon.com/docs/create-tests/record-and-spy/webui-record-and-spy-utilities/smart-wait-function#description) and [Self-healing](https://docs.katalon.com/docs/maintain/self-healing-tests-in-katalon-studio#ariaid-title1) - reduce the maintenance overhead of automated testing with automatic run-time error prevention and mediation.
* [Visual Testing](https://katalon.com/visual-testing): specify if screenshot will be captured as part of a test execution using KS then review the results using TO. We use AI to identify critical UI changes to the layout and text content, reduce false positives and focus on changes that matter to humans.
* [Test Failure Analysis](https://docs.katalon.com/docs/analyze/reports/view-test-reports/view-test-reports-in-katalon-testops/failed-test-results): automatically categorize failed test cases by root cause and recommend actions.
* [Test Flakiness](https://docs.katalon.com/katalon-analytics/docs/view-test-cases.html#flaky-test-cases): learn the status changing pattern from a test execution history and calculate its test flakiness.
* [Image Locator](https://docs.katalon.com/katalon-studio/docs/web-image-based-testing.html) for Web and Mobile app tests (KS): locate UI elements by how it is rendered instead of using the object’s attributes.
* [Web Service Anomalies Detection](https://docs.katalon.com/docs/katalon-testops/reporting/detect-abnormal-web-services#ariaid-title1) (TO): detect APIs with abnormal performance

Katalon TestOps provides a seamless Jira integration that allows you to:

* Link Jira requirements and Jira defects.
* Link test runs to Jira releases.
* Write BDD scenarios in Jira tickets.
* View test results from linked Katalon Studio test cases in Jira.
* We use Jira webhook to notify Katalon Platform when specific events happen in Jira.
* By using a webhook, Jira issue lifecycle changes can be reflected on Katalon Platform in real-time without any manual intervention from users.

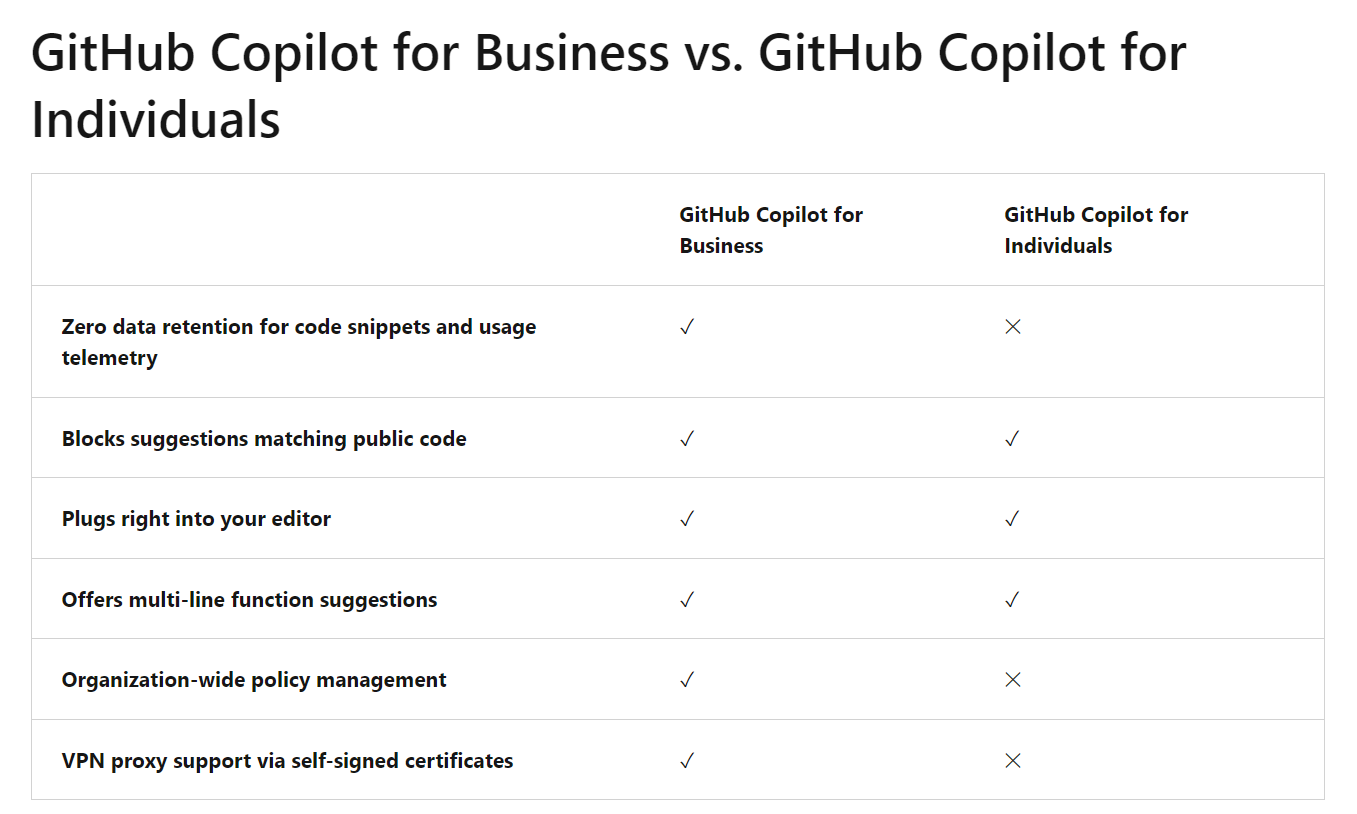
StudioAssist requires you to provide an open.ai key in order to function.

GITHUB COPILOT:

* 46% of new code is now written by AI
* 55% faster overall developer productivity
* 74% of developers feel more focused on satisfying work

Advancements:

* Generate code based on comments
* Generate unit tests
* Provides fixes to bugs
* Automatically add the tags based on changed code for pull requests.
* GitHub is launching GitHub Copilot for Docs, an experimental tool that uses a chat interface to provide users with AI-generated responses to questions about documentation—including questions developers have about the languages, frameworks, and technologies they’re using.
* GitHub Copilot CLI can compose commands and loops, and throw around obscure find flags to satisfy your query.
* The GitHub Copilot extension stores the log files in the standard log location for Visual Studio Code extensions.



**AZURE OPENAI:**

Microsoft has partnered with OpenAI to deliver on three main goals:

* To utilize Azure's infrastructure, including security, compliance, and regional availability, to help users build enterprise-grade applications.
* To deploy OpenAI AI model capabilities across Microsoft products, including and beyond Azure AI products.
* To use Azure to power all of OpenAI's workloads.

You can find the regions available for a service through the CLI command az account list-locations. To see how to sign into Azure and create an Azure group via the CLI, you can refer to the [**documentation here**](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/how-to/create-resource?pivots=cli#sign-in-to-the-cli?azure-portal=true).

## **Deploy using Azure CLI**

You can also deploy a model using the console. Using this example, replace the following variables with your own resource values:

* myResourceGroupName: *replace with your resource group name*
* myResourceName: *replace with your resource name*
* MyModel: *replace with a unique name for your model*
* gpt-35-turbo: *replace with the base model you wish to deploy*

Once the model is deployed, you can test how it completes prompts. A prompt is the text portion of a request that is sent to the deployed model's completions endpoint. Responses are referred to as completions, which can come in form of text, code, or other formats.

## **Prompt types:**

Prompts can be grouped into types of requests based on task.

* 1. Classifying content – sentimental analysis is an example
  2. Generating new content- list ways of travelling(prompt) – response is completion ( bike,car….)
  3. Holding a conversation – a friendly AI assistant
  4. Transformation – eng – hello French:?
  5. Summarizing content- provide summary of the content
  6. Picking up where you left off – one way to grow tomatoes – is to plant seeds.
  7. Giving factual responses – how many moons does earth have? – One
* Azure OpenAI Service lets you tailor our models to your personal datasets using a process known as fine-tuning.

## **Completions playground:**

* Temperature: temp decreases- repetitive and deterministic responses.

Temp increases- unexpected or creative responses.

Adjust temp ot top p -> not both.

* Max length: Set a limit on the number of tokens per model response. The API supports a maximum of 4000 tokens shared between the prompt (including system message, examples, message history, and user query) and the model's response. One token is roughly four characters for typical English text.
* Stop sequences: Make responses stop at a desired point, such as the end of a sentence or list. **Specify up to four sequences where the model will stop generating further tokens in a response.**
* Top probabilities:  Lowering Top P narrows the model’s token selection to likelier tokens. Increasing Top P lets the model choose from tokens with both high and low likelihood
* Frequency penalty:

Reduce the chance of repeating a token proportionally based on how often it has appeared in the text so far. This decreases the likelihood of repeating the exact same text in a response.

* Presence penalty: This increases the likelihood of introducing new topics in a response.
* Pre -response text: Insert text after the user’s input and before the model’s response. This can help prepare the model for a response.
* Post-response text: Insert text after the model’s generated response to encourage further user input, as when modeling a conversation.

The Chat playground, like the Completions playground, also includes the Temperature parameter. The Chat playground also supports other parameters not available in the Completions playground. These include:

Max response

Top p

Past messages included

**Integrate Azure OpenAi into your app:**

1. **Create azure openAI resource – similar to other cognitive services creation in azure portal**
2. **Choose and deploy a model.**

Model families break down into three main families:

**Text** or **Generative Pre-trained Transformer (GPT)** - understand and generate natural language and some code

**Code -** built on top of GPT models, and trained on millions of lines of code. These models can understand and generate code, including interpreting comments or natural language to generate code.

**Embeddings** - These models can understand and use embeddings, which are a special format of data that can be used by machine learning models and algorithms.

1. **Authentication and specification of deployed model –**

AOAI can be accessed via a REST API or an SDK currently available for Python and C#. The endpoints available for interacting with a deployed model are used differently, and certain endpoints can only use certain models. The available endpoints are:

* **Completion** - model takes an input prompt, and generates one or more predicted completions
* **ChatCompletion** - model takes input in the form of a chat conversation (where roles are specified with the message they send), and the next chat completion is generated
* **Embeddings** - model takes input and returns a vector representation of that input.

Completion is available for all gpt-3 generation models. ChatCompletion is the only supported option for gpt-4 models and is the preferred endpoint when using the gpt-35-turbo model.

# Use Azure OpenAI REST API

For each call to the REST API, you need the endpoint and a key from your Azure OpenAI resource, and the name you gave for your deployed model.

Completions:

**curl https://YOUR\_ENDPOINT\_NAME.openai.azure.com/openai/deployments/YOUR\_DEPLOYMENT\_NAME/completions?api-version=2022-12-01\**

**-H "Content-Type: application/json" \**

**-H "api-key: YOUR\_API\_KEY" \**

**-d "{**

**\"prompt\": \"Your favorite Shakespeare is\",**

**\"max\_tokens\": 5**

**}"**

**Response:**

**{**

**"id": "<id>",**

**"object": "text\_completion",**

**"created": 1679001781,**

**"model": "text-davinci-003",**

**"choices": [**

**{**

**"text": "Macbeth",**

**"index": 0,**

**"logprobs": null,**

**"finish\_reason": "stop"**

**}**

**]**

**}**

# Write more effective prompts

## 1.**Provide clear instructions**

2. **format of instructions** - Recency bias can affect models, where information located towards the end of the prompt can have more influence on the output than information at the beginning. This recency bias can also come into play when using ChatCompletion in a chat scenario, where more recent messages in the conversation included in the prompt have a greater impact on the response.

**3. use section markers -** A specific technique for formatting instructions is to split the instructions at the beginning or end of the prompt, and have the user content contained within --- or ### blocks.

## **Primary, supporting, and grounding content**

This content can be thought of in two ways: primary and supporting content.

**PRIMARY:** Primary content refers to content that is the subject of the query, such a sentence to translate or an article to summarize. This content is often included at the beginning or end of the prompt (as an instruction and differentiated by --- blocks), with instructions explaining what to do with it.

Supporting content is content that may alter the response, but isn't the focus or subject of the prompt. Examples of supporting content include things like names, preferences, future date to include in the response, and so on. Providing supporting content allows the model to respond more completely, accurately, and be more likely to include the desired information.

**---**

**<insert full email here, as primary content>**

**---**

**<the next line is the supporting content>**

**Topics I'm very interested in: AI, webinar dates, submission deadlines**

**Extract the key points from the above email, and put them in a bulleted list:**

* **This grounding data allows the model to give more accurate and informed answers that may not be part of the dataset it was trained on.**

## **Cues**

Cues are leading words for the model to build upon, and often help shape the response in the right direction. They often are used with instructions, but don't always. Cues are particularly helpful if prompting the model for code generation.

**Write a join query to get customer names with purchases in the past 30 days between tables named orders and customer on customer ID.**

**SELECT**

## Configure app to access Azure OpenAI resource

Configuration for each language varies slightly, but both require the same parameters to be set. The necessary parameters are endpoint, key, and the name of your deployment, which is called the engine when sending your prompt to the model.

By providing context to the AI model, it allows the model to better understand what you are asking for or what it should know to provide the best answer. Context can be provided in several ways.

**SYSTEM MESSAGE:**

* The system message is included at the beginning of a prompt and is designed to give the model instructions, perspective to answer from, or other information helpful to guide the model's response. This system message might include tone or personality, topics that shouldn't be included, or specifics (like formatting) of how to answer.
* The ChatCompletion endpoint enables including the system message by using the System chat role.
* If using the Completion endpoint, similar functionality can be achieved by including the system message at the start of the prompt. This is called a meta prompt, and serves as a base prompt for the rest of the prompt content.

**CONVERSATION HISTORY:**

* Conversation history enables the model to continue responding in a similar way (such as tone or formatting) and allow the user to reference previous content in subsequent queries. This history can be provided in two ways: from an actual chat history, or from a user defined example conversation.
* More conversation history included in the prompt means a larger number of input tokens are used. You will have to determine what the correct balance is for your use case, considering the token limit of the model you are using.

**FEW SHOT LEARNING :**

Using a user defined example conversation is what is called few shot learning, which provides the model examples of how it should respond to a given query. These examples serve to train the model how to respond.

## **Break down a complex task:**

Another technique for improved interaction is to divide complex prompts into multiple queries.

### **Chain of thought:**

Asking a model to respond with the step by step process by which it determined the response is a helpful way to understand how the model is interpreting the prompt. By doing so, you can see where the model made an incorrect logical turn and better understand how to change your prompt to avoid the error.

[**Develop Generative AI solutions with Azure OpenAI Service**](https://learn.microsoft.com/en-us/training/paths/develop-ai-solutions-azure-openai/)

* base GPT-3 family, the standard text model (such as text-davinci-002) has a good base understanding of code. Codex (such as code-davinci-002) has expanded coding capabilities on top of the standard text model.
* This improvement results in just a single model for more recent generations (such as gpt-35-turbo and gpt-4) that can be used for both natural language and code.

When using the playground, you can adjust the **settings** to specify:

* **The number of images to be generated in response to a prompt (the default number of images is one).**
* **The resolution (size) of the generated image(s). Available sizes are *256x256*, *512x512*, or *1024x1024* (which is the default value).**

To make a REST call to the service, you need the endpoint and authorization key for the Azure OpenAI Service resource you have provisioned in Azure. You initiate the image generation process by submitting a POST request to the service endpoint with the authorization key in the header. The request must contain the following parameters in a JSON body:

* **prompt**: The description of the image to be generated.
* **n**: The number of images to be generated.
* **size**: The resolution of the image(s) to be generated (*256x256*, *512x512*, or *1024x1024*).

Request Header:

{

"prompt": "A badger wearing a tuxedo",

"n": 1,

"size": "512x512"

}

**Response:**

{

"created": 1686780744,

"expires": 1686867152,

"id": "6d765598-eeee-4f49-885d-03ee1c8f9b02",

"result": {

"created": 1686780744,

"data": [

{

"url": "https://dalleproduse.....png"

}

]

},

"status": "succeeded"

}

**If you're uploading or using files already in a storage account, Azure OpenAI on your data supports .md, .txt, .html, .pdf, and Microsoft Word or PowerPoint files.**

MONGO DB :

* Mongo and Mongo D:

Mongo – command line shell that connects to a specific instance of mongod. Gives commands to mongoD

Mongo D – Mongo Daemon basically the host process for the database. take action based on commands given.

Mongodb counterparts to relational databases

**Database -> database**

**Tables - > collections --**

{

Name:”all”,

Age:”18”

}

**Rows -> documents(BSON)**

{

Name:”sure”, 🡨 field:value

Age:26, 🡨 field:value

Status:”A”, 🡨 field:value

Groups:[“news”,”sports”] 🡨 field:value

}

**columns -> fields**

**DOCKER:**

**Container:**

->A way to package application with all the necessary dependencies and configuration.

-> portable artifact, easily shared and moved around

-> makes development and deployment more efficient.

->containers live in container repository.

-> private repository or public repository for docker.

**Before containers:**

* Installation process different on each OS environment.
* Many steps where something could go wrong.

After

**REDUX**

* Redux is a third party general purpose javascript library.
* Maintaining state more efficiently in redux.
* Key words:
* Store: it is a centralized location where global state of application will be stored. Store is created using reducer.
* Data in store is immutable.
* {
* isUserLoggedIn :true
* }

**Reducers:** Reducers are responsible for changing state in store.

Reducers are pure functions which take state and action as i/p parameters and returns new state.

Function (state,action)

{

Switch(action.type)

Case “Logout” :

Return{

isUserLoggedIn:false

}

Case “Login”

Return{

isUserLoggedIn: true

}

}

Var counter=100

Function add(a,b)

{

return a+b

}

**Actions:**

They are plain js objects which will have “type” using actions we will just specify intention

<button onCLick=()=>{dispatch({type:”Login”,UserName:””})

{Provider} from react/redux

createStore function\*\*

shortcomings of redux:

* too much of boiler plate code.
* Create own action objects.

ReduxToolKit: easy for configuring store.

Header links and UserProfile contents should be displayed only when user is logged in.

In redux store we can have one state:

{

isUserLoggedIn: true/false

}

In app.js we provide access to store using provider tag.

useSelector is used to subscribe to store.

Reducers:

Pure functions : they are not dependent on external factors.

They don’t have knowledge of API/NW calls etc

They can respond to action generated by slice.

Reducers can handle only synchronous actions.

To handle async actions use extra reducers.

Redux\_thunk in redux

CreateAsyncThunk – create async actions

Eg: axios call etc

It will generated few action creators/action types:

Now in our slice using extraReducers we will handle actions generated by CreateAsyncThunk

* Axios/fetch return promise object
* Pending,fulfilled,rejected.

Header auth/login userprofile

* createAsyncThunk : inside this function we will write async logic.
* Eg: axios call etc
* It will generate few action creatores/action types
* Now in our slice using extraReducers we will handle actions generated by CreateAsyncThunk.
* **For class based components: connect(mapStateToProps, mapDispatchToProps)**
* // similar to useSelector
* mapStateToProps(){

//it will pass state from store as props to a given class component

}

//similar to useDispatch()

mapDispatchToProps()

{

//it will pass actions as props to a given class comp

}

Export default connect(mapStateToProps, mapDispatchToProps)(ClassComp); //it will connect CLassComp to redux store.