Microsoft Federal Developer Summit
Building AI Solutions

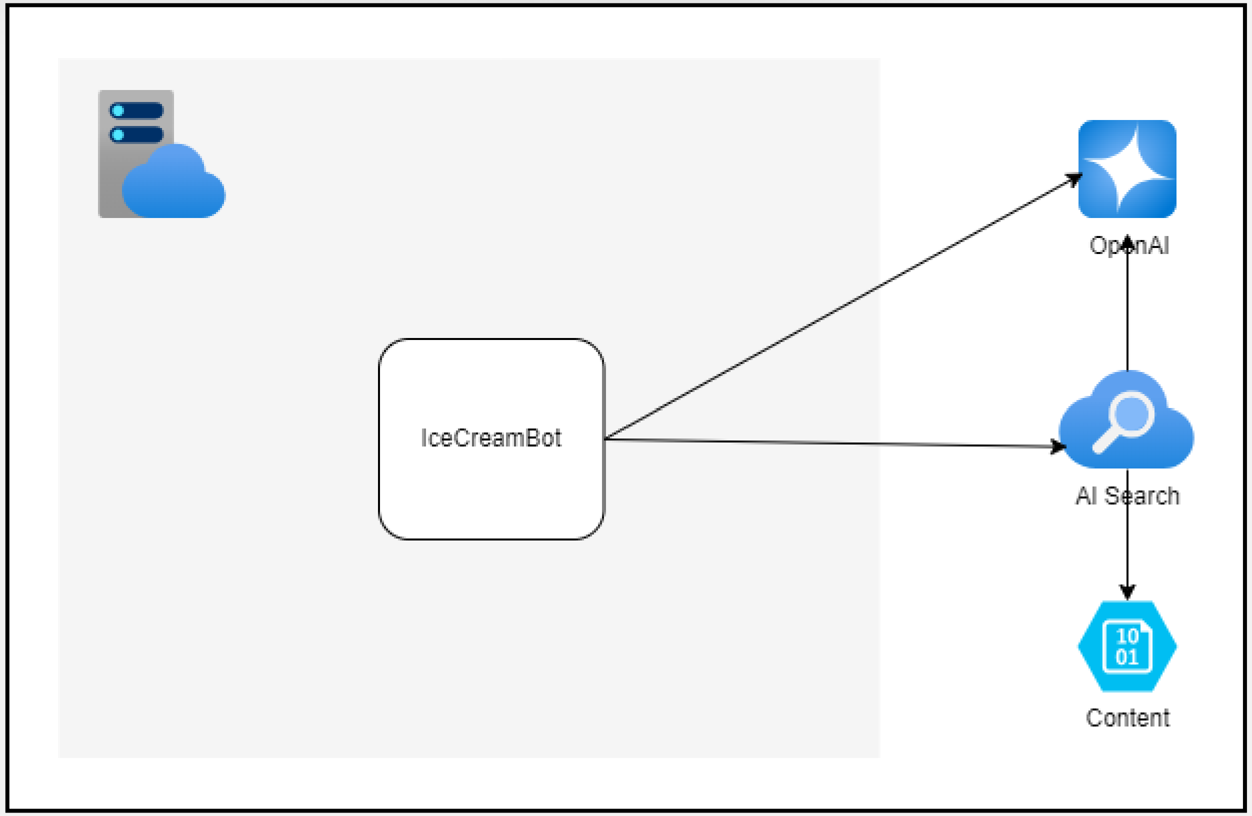
Scaling AI Apps: Things to know before production

Dan Biscup – Principal Cloud Solution Architect
Daniel Setlock – Principal Cloud Solution Architect

AGENDA

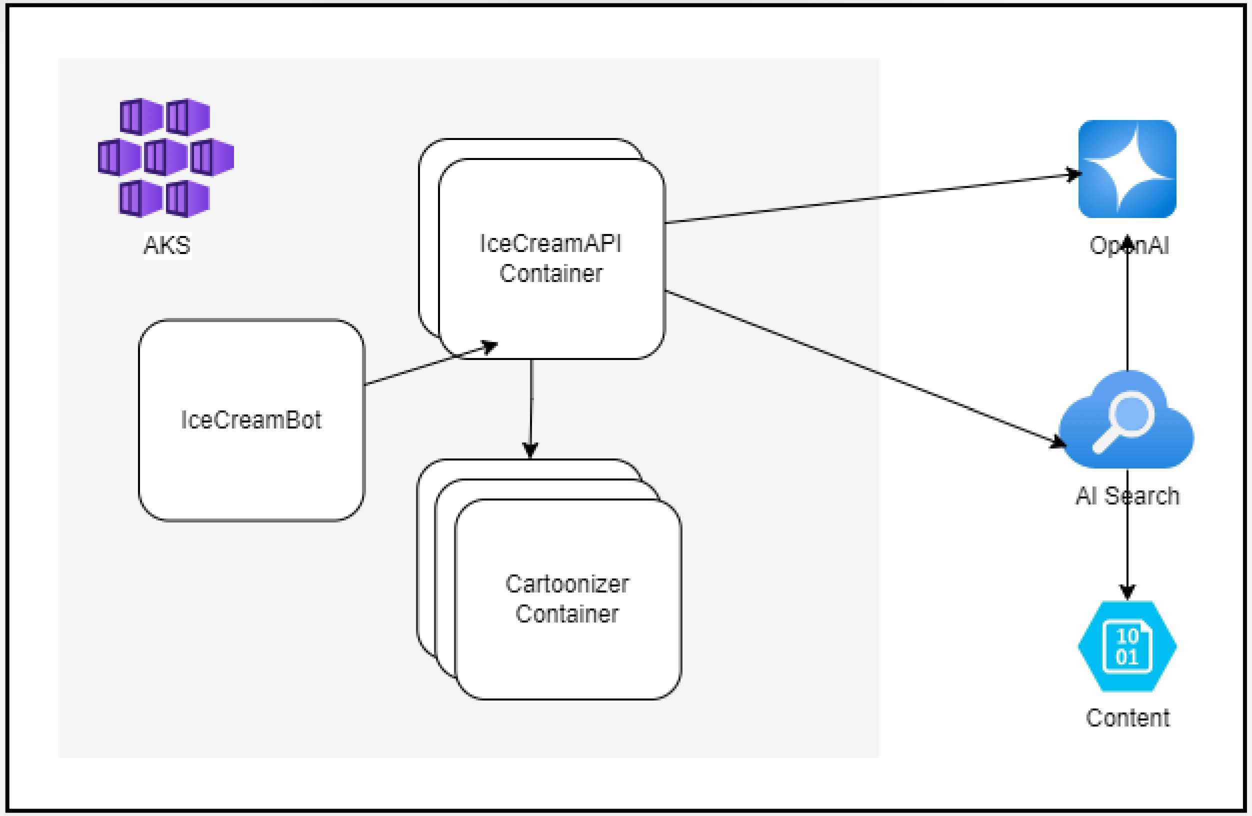
- What is an AI application at scale?
- Review architecture
- Communication
- How do they handle scale?
  - Infrastructure scaling
  - Token Optimization Strategy

What is an AI application at scale?



- Ice-cream Chatbot
- V1 app architecture
- Many apps start off like this
- Some python running in the cloud
- Cartoonizer
- Many Bottlenecks

What is an AI application at scale?



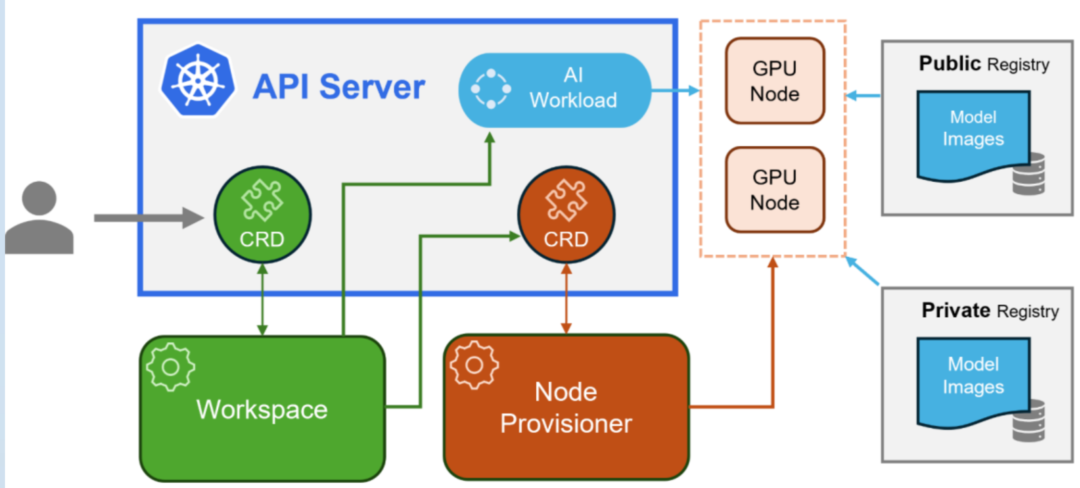
- V2 app architecture
- Many more containers
  - Scale separately as needed
  - GPU
- AKS runs any workload
  - AI favors Python
  - Python favors containers
  - AKS favors containers!
- APIs can scale as needed
- AI processing Containers
- PaaS external
- All roads lead to AKS!

AKS Expanded

AKS Expanded: Running Models in AKS

Kaito

```yaml
apiVersion: kaito.sh/v1alpha1
kind: Workspace
metadata:
  name: workspace-falcon-7b
resource:
  instanceType: "Standard_NC12s_v3"
  labelSelector:
    matchLabels:
      apps: falcon-7b
inference:
  preset:
    name: "falcon-7b"
```



OpenLLM

```
openllm build dolly-v2 --model-id databricks/dolly-v2-3b
```

```
bentoml containerize customdolly:v2 -t dolly-v2-3b:latest --opt
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dolly-v2-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dolly-v2
  template:
    metadata:
      labels:
        app: dolly-v2
    spec:
      containers:
        - name: dolly-v2
          image: dolly-v2-3b:latest
          ports:
            - containerPort: 3000
```

AKS Expanded: Expanding capabilities

- Add-ons
  - Azure Policy
  - AGIC
  - KEDA
  - Key Vault
  - Virtual Nodes

- Extensions
  - Dapr
  - Azure Machine Learning
  - Flux
  - Azure Storage

- Integrations
  - Helm
  - Grafana
  - Prometheus
  - Istio
  - Linkerd

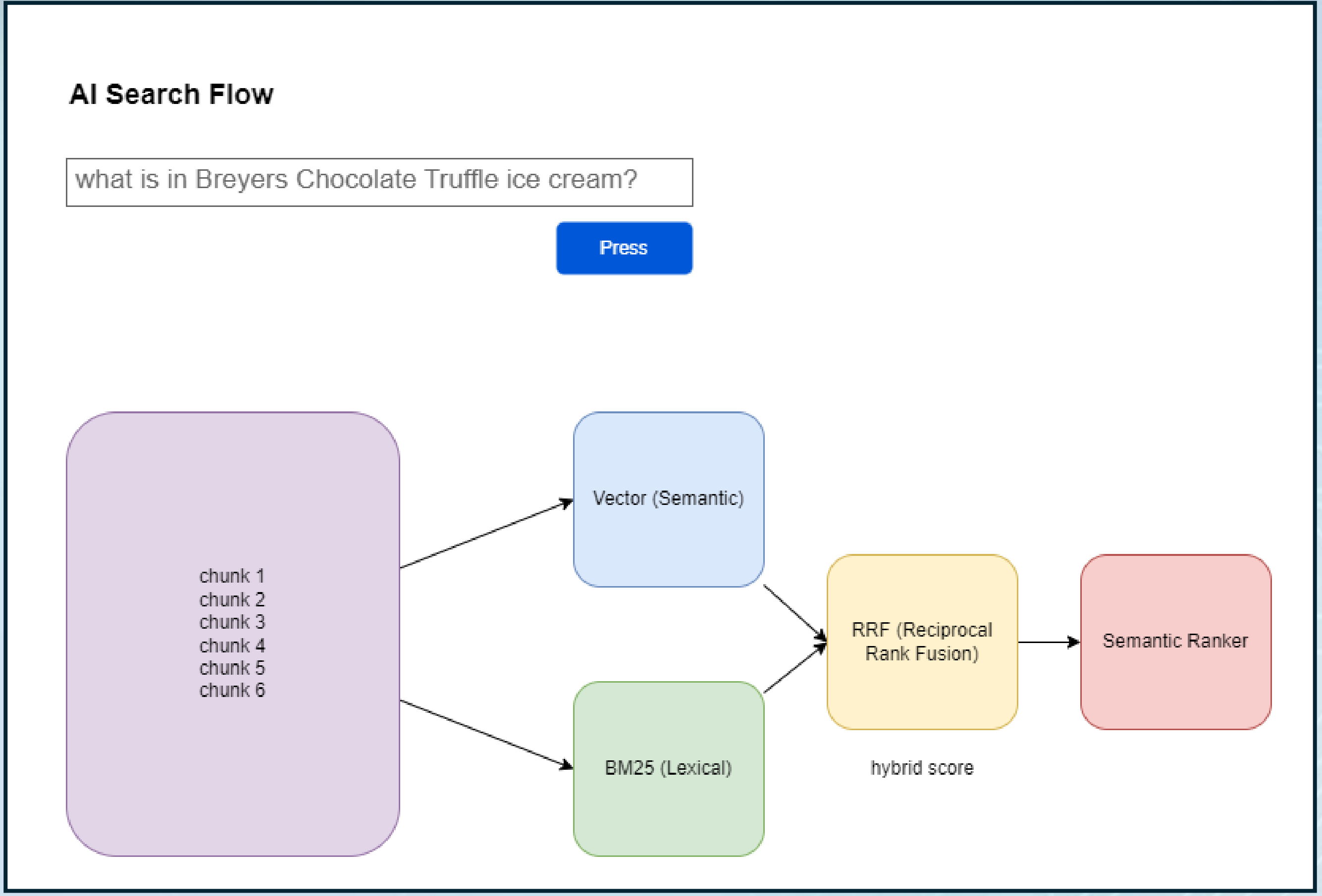
All roads lead to AKS

AI Search Expanded

- Overview
  - Core RAG service
  - Many government use cases
  - The ‘R’ in retrieval

Schematic Ranker – Main Take-Away!

- Higher re-ranker score
- RAG top take-away
  - Top score
  - Less results



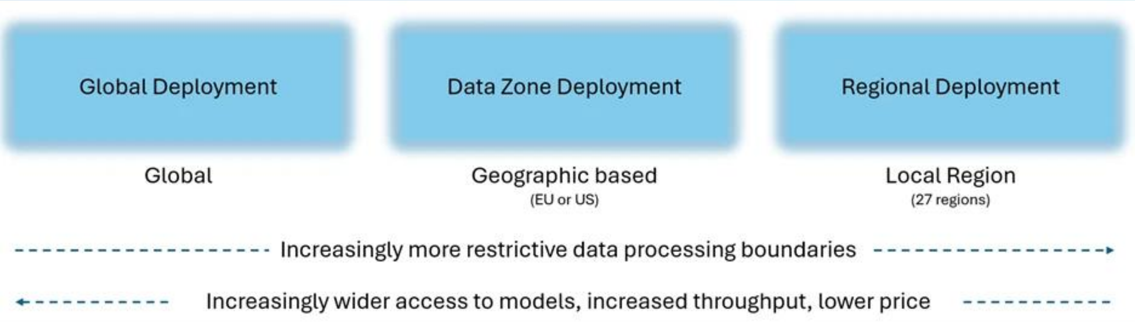
Deeper

- Growing datasets create larger indexes to traverse
  - Schema architecture
- Semantic Search configurations
  - What the data? bm25
- Vector profiles and algorithms
  - Data science
- Configuring efficient vectorizers
  - Enable logging
  - Vector compression
    - Scalar and binary quantization

Token Strategy Expanded

- Explicit token limits for models - All GPT-4 models have a max input of 128,000 tokens, output varies by model. (GPT-3x 4k-16k input/4k output)

- Azure OpenAI Data Zones



Global Deployment | Data Zone Deployment | Regional Deployment

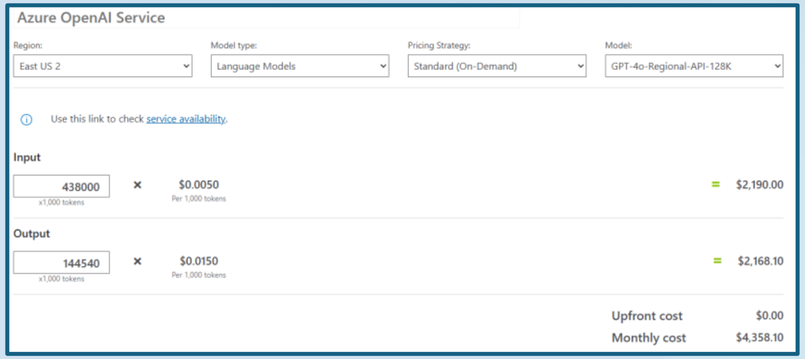
Global | Geographic based (EU or US) | Local Region (27 regions)

Increasingly more restrictive data processing boundaries →

← Increasingly wider access to models, increased throughput, lower price

Pricing (Scaling)

Calculating Token Cost



Azure OpenAI Service

Region: East US 2

Model type: Language Models

Pricing Strategy: Standard (On-Demand)

Model: GPT-4o-Regional-API-128K

ⓘ Use this link to check service availability.

Input

438000 x1,000 tokens	×	$0.0050 Per 1,000 tokens	=	$2,190.00

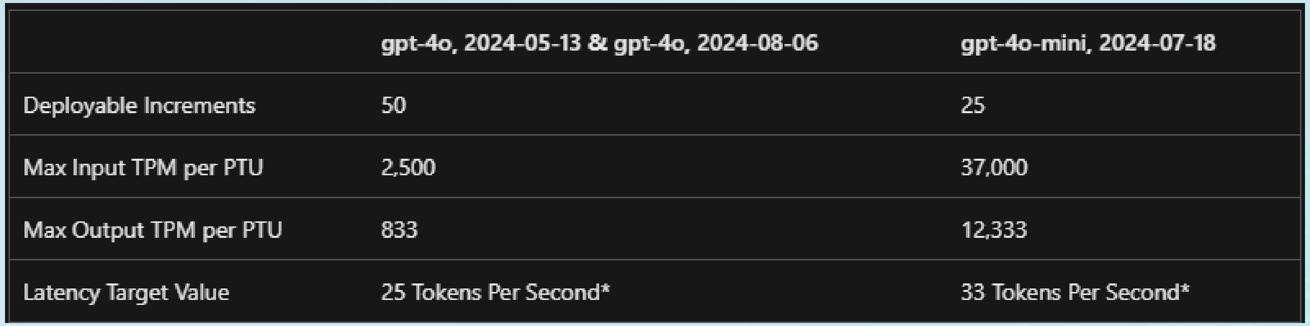
Output

144540 x1,000 tokens	×	$0.0150 Per 1,000 tokens	=	$2,168.10

Upfront cost	$0.00
Monthly cost	$4,358.10

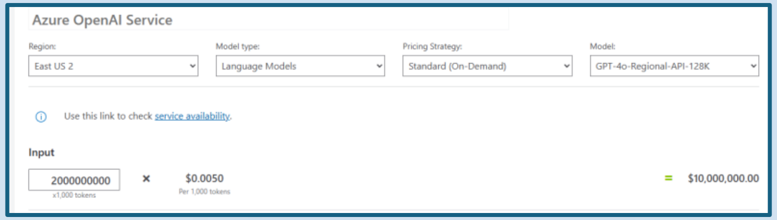
10x60x730 = 438000/token/month

3.3x60x730 = 144540/token/month



	gpt-4o, 2024-05-13 & gpt-4o, 2024-08-06	gpt-4o-mini, 2024-07-18
Deployable Increments	50	25
Max Input TPM per PTU	2,500	37,000
Max Output TPM per PTU	833	12,333
Latency Target Value	25 Tokens Per Second*	33 Tokens Per Second*

Pay-As-You-Go



Azure OpenAI Service

Region: East US 2

Model type: Language Models

Pricing Strategy: Standard (On-Demand)

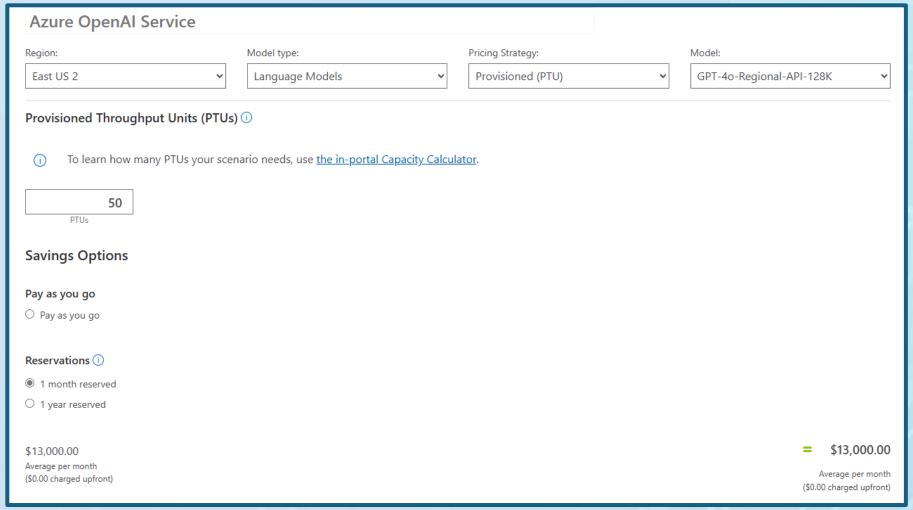
Model: GPT-4o-Regional-API-128K

ⓘ Use this link to check service availability.

Input

2000000000 x1,000 tokens	×	$0.0050 Per 1,000 tokens	=	$10,000,000.00

Provisioned Throughput



Azure OpenAI Service

Region: East US 2

Model type: Language Models

Pricing Strategy: Provisioned (PTU)

Model: GPT-4o-Regional-API-128K

Provisioned Throughput Units (PTUs) ⓘ

ⓘ To learn how many PTUs your scenario needs, use the in-portal Capacity Calculator.

50
PTUs

Savings Options

Pay as you go

○ Pay as you go

Reservations ⓘ

● 1 month reserved

○ 1 year reserved

$13,000.00
Average per month
($0.00 charged upfront)

= $13,000.00
Average per month
($0.00 charged upfront)

Prompt Engineering – Taking power away from the users

Sanitize User Inputs

```
def sanitize_input(user_input):
    # Remove potentially harmful characters or sequences
    user_input = re.sub(r'[^\w\s]', '', user_input)
    # Collapse multiple spaces to a single space
    user_input = re.sub(r'\s+', ' ', user_input).strip()
    return user_input
```

Use Template-based prompts

```
def create_prompt(user_input):
    sanitized_input = sanitize_input(user_input)
    return f"Please provide a concise and accurate summary for the following query: '{sanitized_input}'"
```

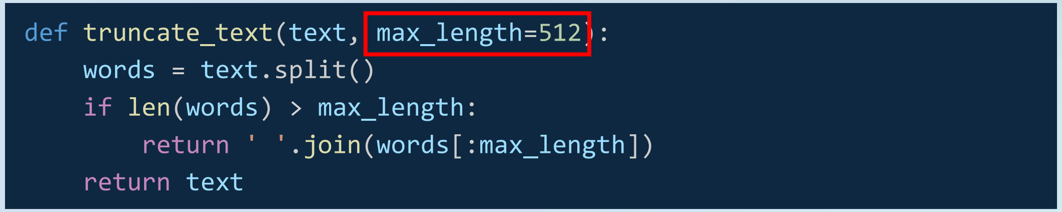
Dynamic Context Injection

```
def dynamic_prompt(user_query):
    if 'summary' in user_query:
        return f"Summarize the following information concisely: {sanitize_input(user_query)}"
    elif 'explain' in user_query:
        return f"Explain the concept: {sanitize_input(user_query)} in detail."
    else:
        return f"Answer the following question: {sanitize_input(user_query)}"
```

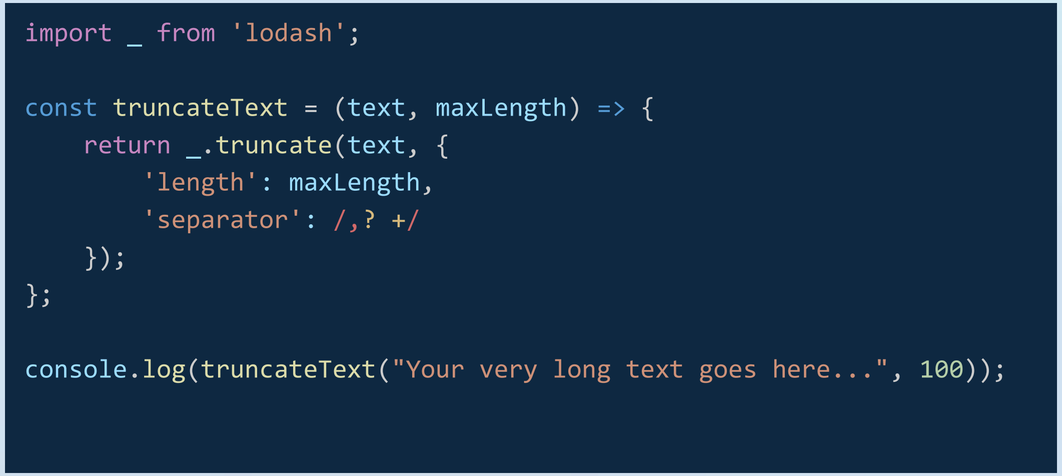
Everything together

```
user_input = "How does quantum computing work?"
prompt = dynamic_prompt(user_input)
response = openai.Completion.create(engine="gpt-4o", prompt=prompt, max_tokens=150)
print(response.choices[0].text.strip())
```

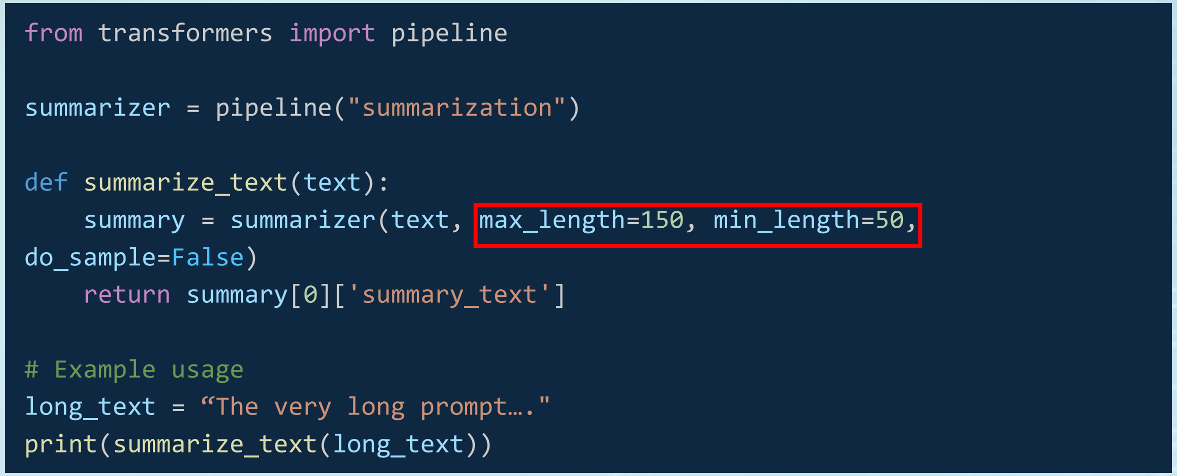
Truncation and Summarization



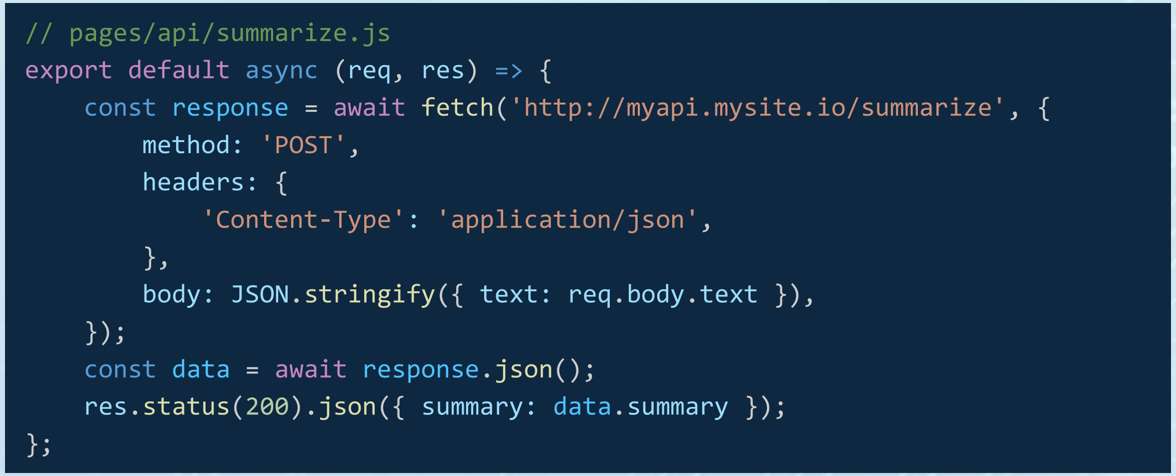
```python
def truncate_text(text, max_length=512):
    words = text.split()
    if len(words) > max_length:
        return ' '.join(words[:max_length])
    return text
```



```javascript
import _ from 'lodash';

const truncateText = (text, maxLength) => {
    return _.truncate(text, {
        'length': maxLength,
        'separator': /,? +/
    });
};

console.log(truncateText("Your very long text goes here...", 100));
```

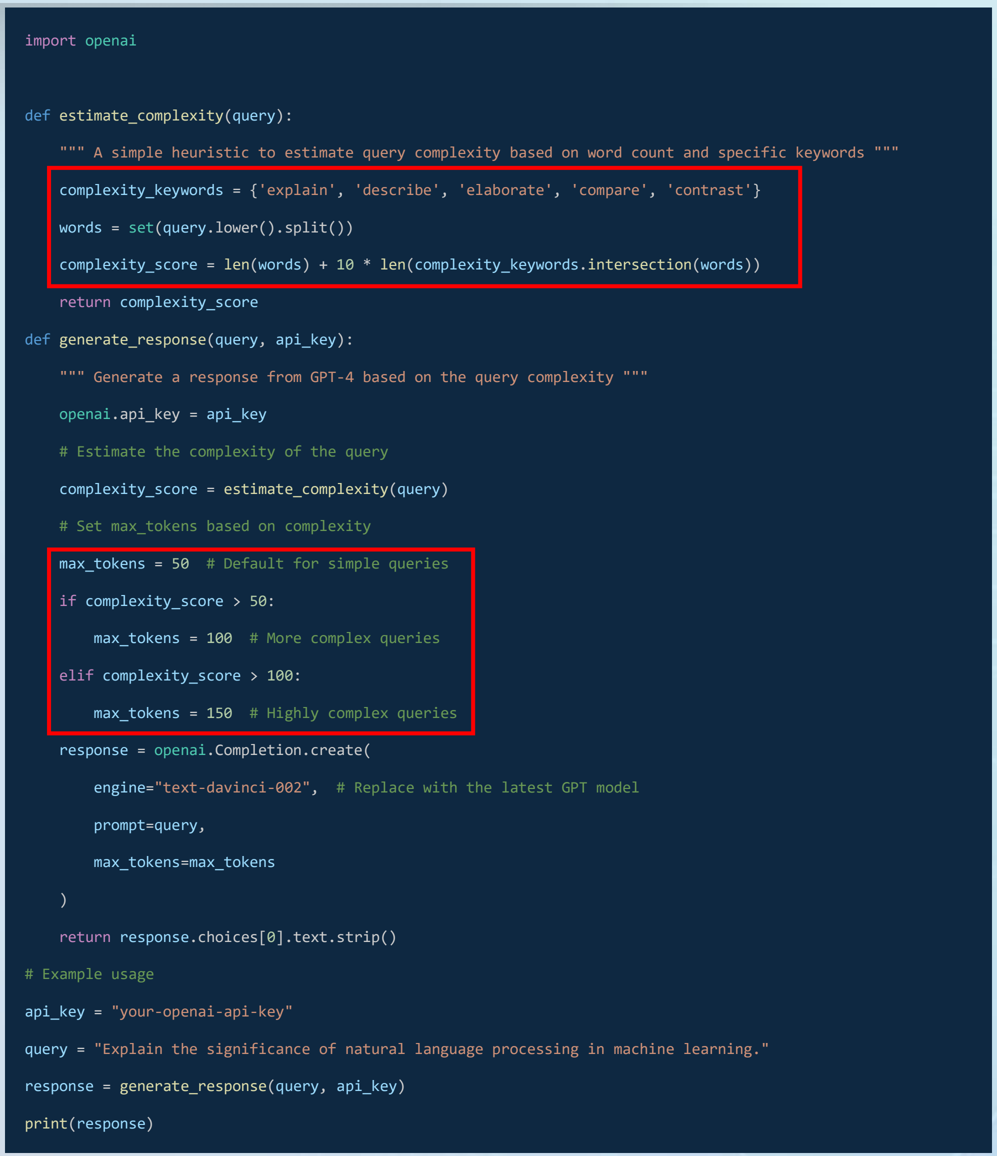


```python
from transformers import pipeline

summarizer = pipeline("summarization")

def summarize_text(text):
    summary = summarizer(text, max_length=150, min_length=50,
do_sample=False)
    return summary[0]['summary_text']

# Example usage
long_text = “The very long prompt…."
print(summarize_text(long_text))
```



```javascript
// pages/api/summarize.js
export default async (req, res) => {
    const response = await fetch('http://myapi.mysite.io/summarize', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ text: req.body.text }),
    });
    const data = await response.json();
    res.status(200).json({ summary: data.summary });
};
```

Smart Token Allocation



```python
import openai


def estimate_complexity(query):
    """ A simple heuristic to estimate query complexity based on word count and specific keywords """
    complexity_keywords = {'explain', 'describe', 'elaborate', 'compare', 'contrast'}
    words = set(query.lower().split())
    complexity_score = len(words) + 10 * len(complexity_keywords.intersection(words))
    return complexity_score
def generate_response(query, api_key):
    """ Generate a response from GPT-4 based on the query complexity """
    openai.api_key = api_key
    # Estimate the complexity of the query
    complexity_score = estimate_complexity(query)
    # Set max_tokens based on complexity
    max_tokens = 50  # Default for simple queries
    if complexity_score > 50:
        max_tokens = 100  # More complex queries
    elif complexity_score > 100:
        max_tokens = 150  # Highly complex queries
    response = openai.Completion.create(
        engine="text-davinci-002",  # Replace with the latest GPT model
        prompt=query,
        max_tokens=max_tokens
    )
    return response.choices[0].text.strip()
# Example usage
api_key = "your-openai-api-key"
query = "Explain the significance of natural language processing in machine learning."
response = generate_response(query, api_key)
print(response)
```

Pre and Post Prompt/Request Processing

Pre-Processing

```
def sanitize_input(text):
    """ Sanitize input by removing unwanted characters and simplifying text. """
    text = re.sub(r'[^\w\s]', '', text)  # Remove non-alphanumeric characters
    text = re.sub(r'\s+', ' ', text).strip()  # Replace multiple spaces with single
space
    return text

def add_context(user_input, context):
    """ Add necessary context to the user's input to form a complete prompt. """
    return f"{context}\n\n{user_input}"

def preprocess_input(user_input, context="Please provide a detailed explanation:"):
    """ Full preprocessing pipeline for user input. """
    sanitized_input = sanitize_input(user_input)
    complete_prompt = add_context(sanitized_input, context)
    return complete_prompt

# Usage example
user_input = "Explain the significance of E=mc^2."
context = "Context: Provide a detailed educational explanation suitable for a high
school physics class."
refined_prompt = preprocess_input(user_input, context)
```

Post-Processing

```
def extract_key_points(response):
    """ Extract key points from a lengthy response. """
    sentences = response.split('. ')
    key_points = [sentence for sentence in sentences if 'important' in sentence]
    return ' '.join(key_points)
def enhance_output(raw_output):
    """ Enhance output by correcting grammar, refining tone, and adding proprietary information.""
    # Assuming a function 'correct_grammar' that fixes grammatical errors
    # and 'refine_tone' that adjusts the tone of the output
    output = correct_grammar(raw_output)
    output = refine_tone(output, desired_tone='formal')
    output += "\n\nNote: This explanation is provided based on the latest scientific research."
    return output
def postprocess_response(raw_response):
    """ Full postprocessing pipeline for model output. """
    key_points = extract_key_points(raw_response)
    enhanced_response = enhance_output(key_points)
    return enhanced_response
# Example usage
raw_response = "The formula E=mc^2, introduced by Einstein, is important because it shows that
energy and mass are interchangeable."
processed_response = postprocess_response(raw_response)
```

Intelligent Layering and Smart Query Management

Caching and smart routing

- Cache common or routine prompts and responses to limit necessary queries and consumption
- Smart routing would dynamically determine the best way to handle requests based on predefined criteria

Caching

```python
from functools import lru_cache
@lru_cache(maxsize=100)
def get_response(query):
    return handle_query(query)
# Example usage
query = "What is the weather today in New York?"
response = get_response(query)  # First call, goes through full processing
response = get_response(query)  # Subsequent call, fetched from cache
```

Smart Routing

```python
def smart_route_query(query):
    if "lookup" in query.lower():
        return handle_simple_query(query)  # Assumes lookup queries are better handled by Azure
AI Search
    else:
        return handle_complex_query(query)  # More analytical, interpretive queries go to GPT-4
# Example usage
query = "Lookup population data for 2020."
response = smart_route_query(query)
```

Utilize a tiered query handling to more effectively consume resources

- First layer – Azure AI Search for structured queries or fetching date from predefined dataset
- Second layer – For complex language understanding, generation tasks, or when first layer results are not satisfactory

Tiered Query Handling

```python
def is_complex_query(query):
    # Simple heuristic to determine query complexity
    return len(query.split()) > 10  # Consider a query complex if more than 10 words
def handle_query(query):
    if is_complex_query(query):
        response = handle_complex_query(query)
    else:
        response = handle_simple_query(query)
    return response
def handle_simple_query(query):
    # Placeholder for Azure AI Search integration
    # Here you would query Azure AI Search and return the results
    return "Results from Azure AI Search for simple query."
def handle_complex_query(query):
    # Using OpenAI's GPT-4 for complex queries
    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=query,
        max_tokens=150
    )
    return response.choices[0].text.strip()
```

Feedback

Do you want us to follow up after the event? Do you have feedback?

https://aka.ms/summit/feedback

Microsoft Federal Developer Summit
Building AI Solutions