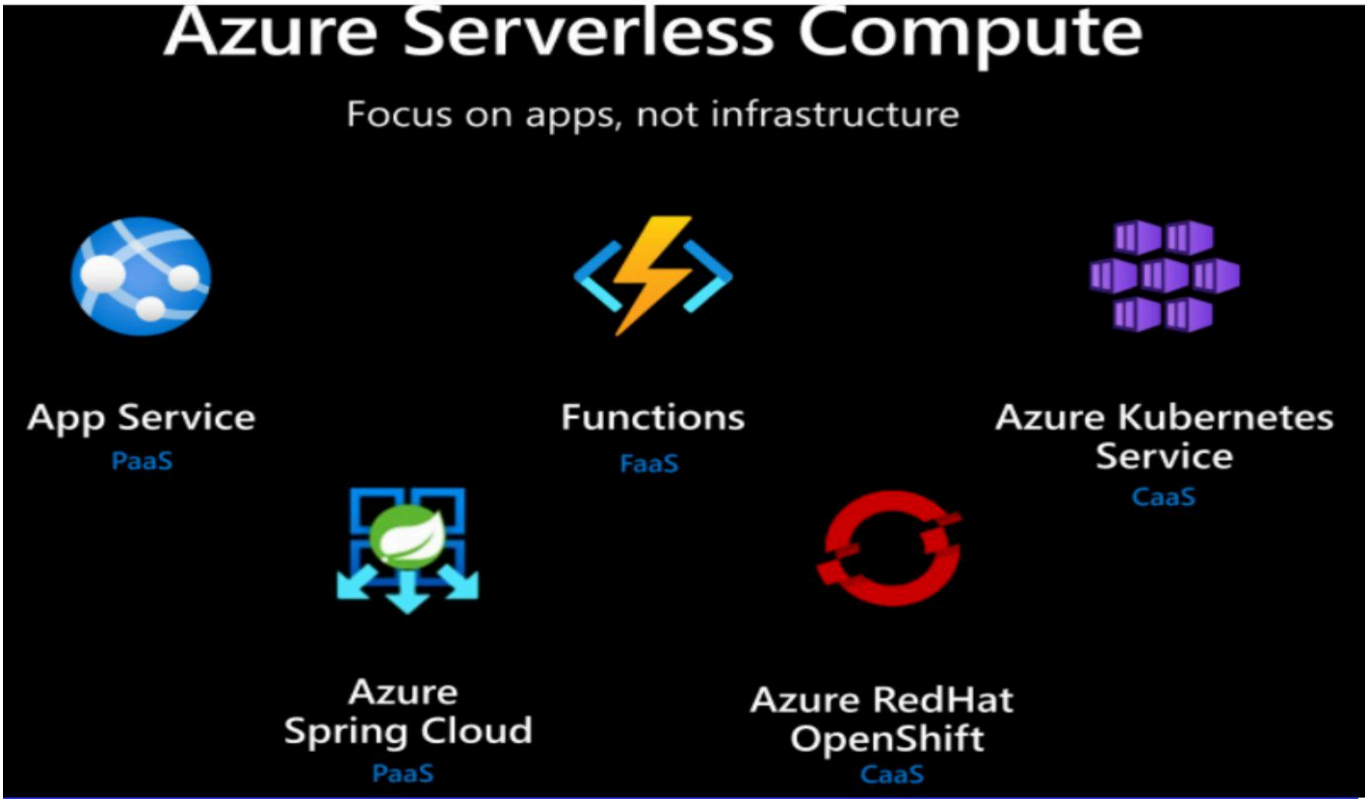
SERVERLESS COMPUTING IN AZURE

REALTIME PROJECTS

PART 4



500+ RealTime Projects platform

Project 1: Serverless Event-Driven Image Processing Pipeline

1. Project Scope

This project automates the process of resizing, watermarking, and optimizing images using Azure Functions and Azure Blob Storage. Whenever a new image is uploaded to a specific container, an event is triggered to process and store the transformed image in another container.

2. Tools Used

- **Azure Blob Storage** (For image storage)
- **Azure Functions** (Serverless compute for processing)
- **Azure Event Grid** (Event-driven architecture)
- **Azure Application Insights** (For monitoring)
- **Python** (Function code for image processing)

3. Analysis Approach

- When a user uploads an image to Blob Storage, **Azure Event Grid** triggers an Azure Function.
- The Function **resizes and watermarks the image** using the Python Pillow library.
- The processed image is then **saved in another container** for optimized delivery.

4. Step-by-Step Implementation

1. **Create an Azure Storage Account**
    - Navigate to **Azure Portal** → Create **Storage Account**.
    - Create **two containers**: `input-images` (for uploads) and `processed-images` (for processed files).
2. **Enable Event Grid Subscription**
    - In the Storage Account, enable **Event Grid**.
    - Create an event subscription that **triggers the Azure Function** whenever a new file is uploaded.
3. **Create an Azure Function**
    - Choose **Python runtime** and install dependencies (`pip install pillow`).
    - Implement function logic:

```python
from PIL import Image
import io
from azure.storage.blob import BlobServiceClient

def main(event: dict):
```

```
    storage_connection_string = "<Azure_Storage_Connection_String>"
    blob_service_client =
BlobServiceClient.from_connection_string(storage_connection_string)

    input_blob = event['data']['url']
    container_name = "processed-images"

    # Fetch the uploaded image
    blob_client =
blob_service_client.get_blob_client(container="input-images",
blob=input_blob.split("/")[-1])
    img_bytes = blob_client.download_blob().readall()

    # Process image
    image = Image.open(io.BytesIO(img_bytes))
    image = image.resize((500, 500))  # Resize to 500x500

    # Save the new image
    img_io = io.BytesIO()
    image.save(img_io, format='JPEG')
    img_io.seek(0)

    # Upload processed image
    output_blob_client =
blob_service_client.get_blob_client(container=container_name,
blob=input_blob.split("/")[-1])
    output_blob_client.upload_blob(img_io, overwrite=True)
```

4. **Deploy & Test**
    - Deploy function to **Azure Functions** using `Azure CLI`.
    - Upload an image to `input-images`, check if it's processed in `processed-images`.

5. Conclusion

This project demonstrated **event-driven image processing using serverless architecture**, eliminating the need for VM-based solutions. It improves efficiency and scalability.

Project 2: Serverless Video Transcription using Azure Cognitive Services

1. Project Scope

This project transcribes audio from uploaded videos in **real-time** using **Azure Functions** and **Azure Speech-to-Text API**.

2. Tools Used

- **Azure Blob Storage** (For video uploads)
- **Azure Event Grid** (For event-driven processing)
- **Azure Functions** (For executing transcription)
- **Azure Cognitive Services Speech API** (For speech-to-text conversion)
- **Python** (For integration)

3. Analysis Approach

1. A user uploads a video file to a **Blob Storage container**.
2. An **Azure Event Grid trigger** detects the file and calls **Azure Functions**.
3. The function extracts the **audio from video** and **sends it to the Azure Speech API**.
4. The transcribed text is stored in **Blob Storage or Cosmos DB**.

4. Step-by-Step Implementation

1. **Create a Storage Account & Container**
    - Containers: `videos` (raw files), `transcripts` (text output).
2. **Enable Event Grid**
    - Create an **event subscription** for blob uploads.
3. **Create an Azure Function**
    - Install dependencies: `pip install azure-cognitiveservices-speech moviepy`
    - Implement transcription function:

```python
import azure.cognitiveservices.speech as speechsdk
from moviepy.editor import AudioFileClip

def transcribe_audio(audio_path):
    speech_config = speechsdk.SpeechConfig(subscription="<API_KEY>",
region="eastus")
    audio_input = speechsdk.AudioConfig(filename=audio_path)

    speech_recognizer =
speechsdk.SpeechRecognizer(speech_config=speech_config,
audio_config=audio_input)
```

```
    result = speech_recognizer.recognize_once()

    return result.text

def main(event):
    # Extract audio from video
    video_path = event['data']['url']
    audio_path = "/tmp/audio.mp3"
    AudioFileClip(video_path).write_audiofile(audio_path)

    # Transcribe audio
    transcript = transcribe_audio(audio_path)

    # Save to Blob Storage
    save_transcription(transcript)
```

4. **Deploy and Test**
    - Deploy the function to **Azure Functions**.
    - Upload a **video file**, verify if transcription is generated.

5. Conclusion

This project showcases **real-time video-to-text conversion** without using servers, demonstrating how Azure Functions and Cognitive Services can be used effectively.

Project 3: Serverless Chatbot with Azure AI and Functions

1. Project Scope

Create a **serverless AI chatbot** using **Azure OpenAI** that responds to user queries in real-time.

2. Tools Used

- **Azure Functions** (For handling API requests)
- **Azure OpenAI** (For AI-powered responses)
- **Azure API Management** (For managing APIs)
- **Python & FastAPI** (For building API)

3. Analysis Approach

- A client sends a **query to the Azure Function** via an API request.

- The **Azure Function** forwards it to the **OpenAI API**.
- The response is sent back **as a real-time chatbot reply**.

4. Step-by-Step Implementation

1. **Create an Azure Function App**
   - Use **Python runtime** and enable HTTP triggers.
2. **Integrate OpenAI API**
   - Install: `pip install openai fastapi`
   - Function code:

```python
from fastapi import FastAPI
import openai

app = FastAPI()
openai.api_key = "<API_KEY>"

@app.post("/chatbot")
def chatbot(query: str):
    response = openai.Completion.create(
        model="gpt-3.5-turbo",
        prompt=query,
        max_tokens=50
    )
    return {"reply": response["choices"][0]["text"]}
```

3. **Deploy to Azure Functions**
   - Use `Azure CLI` to deploy the function.
   - Connect it to a chatbot UI for real-time interaction.
4. **Test the chatbot**
   - Send a POST request:

```
curl -X POST "https://<function-url>/chatbot" -d '{"query":"Hello"}'
```

5. Conclusion

This project provides an AI-powered **real-time chatbot** without requiring dedicated servers, making it scalable and cost-efficient.

Project 4: Automated Resume Screening with Azure AI & Cosmos DB

1. Project Scope

This project automates **resume screening** using **Azure Cognitive Services** and **Azure Cosmos DB**. When a **new resume (PDF/DOCX) is uploaded**, the system extracts key details like name, experience, and skills, stores them in **Cosmos DB**, and provides insights.

2. Tools Used

- **Azure Blob Storage** (For storing resumes)
- **Azure Cognitive Services (Text Analytics API)** (For extracting skills and experience)
- **Azure Functions** (For processing data)
- **Azure Cosmos DB** (For storing structured data)
- **Azure Logic Apps** (For automation)

3. Analysis Approach

1. **A user uploads a resume** to Blob Storage.
2. **Event Grid** triggers an **Azure Function**.
3. **Cognitive Services (Text Analytics API)** extracts **name, skills, and experience**.
4. The extracted details are **stored in Cosmos DB**.
5. **Azure Logic Apps** sends **email notifications** for qualified candidates.

4. Step-by-Step Implementation

1. **Create Azure Storage Container**
    - Containers: `resumes` (raw files) and `processed-resumes`.
2. **Enable Event Grid Trigger**
    - Set up an **event subscription** for blob uploads.
3. **Create Azure Function**

```python
import azure.cognitiveservices.speech as speechsdk

from azure.ai.textanalytics import TextAnalyticsClient
```

```
from azure.identity import DefaultAzureCredential

from azure.storage.blob import BlobServiceClient


def extract_text_from_resume(resume_url):

    credential = DefaultAzureCredential()

    client = TextAnalyticsClient(endpoint="<TEXT_ANALYTICS_ENDPOINT>",
credential=credential)

    document = [resume_url]


    response = client.extract_key_phrases(document)

    return response


def main(event: dict):

    resume_url = event['data']['url']

    extracted_data = extract_text_from_resume(resume_url)


    # Store in Cosmos DB

    save_to_cosmos_db(extracted_data)
```

4. **Deploy & Test**
    - Upload a resume and check if **data is stored in Cosmos DB.**

5. Conclusion

This project **automates resume processing**, helping HR teams shortlist candidates efficiently.

Project 5: Serverless IoT Data Processing with Azure Stream Analytics

1. Project Scope

This project **ingests IoT sensor data in real-time** and performs **analytics** using **Azure Stream Analytics** and **Azure Functions**.

2. Tools Used

- **Azure IoT Hub** (For collecting IoT data)
- **Azure Stream Analytics** (For real-time processing)
- **Azure Functions** (For data transformations)
- **Azure Cosmos DB** (For storing results)
- **Power BI** (For visualization)

3. Analysis Approach

1. **IoT devices** send telemetry data (temperature, humidity) to **Azure IoT Hub**.
2. **Azure Stream Analytics** processes and filters anomalies.
3. **Azure Functions** transforms the data.
4. **Processed data** is stored in **Cosmos DB**.
5. **Power BI Dashboard** visualizes live data.

4. Step-by-Step Implementation

1. **Set Up Azure IoT Hub**
   - Register an IoT device and generate connection string.
2. **Create Azure Stream Analytics Job**
   - Input: IoT Hub
   - Output: Azure Cosmos DB
   - Query:

```
Unset
SELECT deviceId, temperature, humidity, system.Timestamp AS time

INTO cosmosdb_output

FROM iot_hub_input

WHERE temperature > 80
```

3. **Create Azure Function for Data Transformation**

Python
```python
import json

def main(event: dict):

    data = json.loads(event)

    if data['temperature'] > 100:

        alert = "Critical temperature alert!"

    else:

        alert = "Normal temperature"

    return {"alert": alert}
```

4. **Deploy & Test**
    - Simulate **IoT data ingestion**.
    - Check **alerts in Cosmos DB**.

5. Conclusion

This project enables **real-time IoT data analytics**, useful for **predictive maintenance**.

Project 6: Real-time Twitter Sentiment Analysis using Azure Functions

1. Project Scope

This project **monitors Twitter** for a keyword and performs **sentiment analysis** using **Azure AI**.

2. Tools Used

- **Azure Logic Apps** (For Twitter API integration)
- **Azure Functions** (For text processing)
- **Azure Cognitive Services** (For Sentiment Analysis)
- **Azure Cosmos DB** (For storing results)
- **Power BI** (For visualization)

3. Analysis Approach

1. **Azure Logic Apps** fetches **real-time tweets**.
2. **Azure Function** analyzes tweet sentiment.
3. **Results are stored in Cosmos DB.**
4. **Power BI Dashboard** visualizes sentiment trends.

4. Step-by-Step Implementation

1. **Set Up Logic App**
    - Connect Twitter API to fetch tweets.
2. **Create Azure Function**

```python
from azure.ai.textanalytics import TextAnalyticsClient


def analyze_tweet(tweet):

    client = TextAnalyticsClient(endpoint="<API_ENDPOINT>",
credential="<KEY>")

    result = client.analyze_sentiment([tweet])

    return result[0].sentiment
```

3. **Store Results in Cosmos DB**
    - Insert **sentiment scores** into Cosmos DB.
4. **Deploy & Test**
    - Monitor **live sentiment trends** in Power BI.

5. Conclusion

This project enables **real-time brand monitoring** using **AI-powered sentiment analysis**.

Project 7: Automated Document Summarization with Azure AI

1. Project Scope

Automatically **summarizes large documents** using **Azure Cognitive Services.**

2. Tools Used

- **Azure Blob Storage** (For storing documents)
- **Azure Cognitive Services** (For text summarization)
- **Azure Functions** (For processing)
- **Cosmos DB** (For storing summaries)

3. Analysis Approach

1. **User uploads a document.**
2. **Azure Function extracts key insights.**
3. **Summary is stored in Cosmos DB.**

4. Step-by-Step Implementation

1. **Upload Document to Blob Storage**
    - Create a container `docs`.
2. **Create an Azure Function**

```python
def summarize_text(text):

    client = TextAnalyticsClient(endpoint="<ENDPOINT>",
credential="<KEY>")

    result = client.extract_summary([text])

    return result[0].summary
```

3. **Deploy & Test**
    - Upload a large document and retrieve **summary.**

5. Conclusion

This project **reduces reading time** using **automated text summarization.**

Project 8: Real-time Weather Data Processing with Azure Event Hub

1. Project Scope

Real-time weather data processing using **Azure Event Hub**.

2. Tools Used

- **Azure Event Hub** (For ingesting data)
- **Azure Functions** (For processing)
- **Cosmos DB** (For storing results)

3. Analysis Approach

1. **Weather sensors push data** to **Event Hub**.
2. **Azure Function transforms the data.**
3. **Processed data is stored in Cosmos DB.**

4. Step-by-Step Implementation

1. **Set Up Event Hub**
   - Configure **real-time streaming.**
2. **Create Azure Function**

```python
def process_weather(event):

    data = json.loads(event)

    return {"temperature": data["temp"], "humidity": data["humidity"]}
```

3. **Deploy & Test**
   - Simulate **weather events.**

5. Conclusion

This project enables **real-time weather insights**.

Project 9: AI-based Email Classification using Azure Logic Apps

1. Project Scope

This project automates **email classification** using **Azure AI**. Incoming emails are categorized as **Spam, Important, or General** using **Azure Cognitive Services**.

2. Tools Used

- **Azure Logic Apps** (For email processing)
- **Azure Cognitive Services** (For text classification)
- **Azure Functions** (For decision logic)
- **Cosmos DB** (For storing email metadata)

3. Analysis Approach

1. **Emails are received** via Outlook/Gmail.
2. **Logic Apps extracts email content.**
3. **Azure Function sends content to Azure AI** for classification.
4. **Classified emails** are stored in **Cosmos DB.**
5. **Spam emails** are automatically moved to the spam folder.

4. Step-by-Step Implementation

1. **Create Azure Logic Apps**
    - Connect Logic Apps to **Outlook API** to read emails.
2. **Implement Email Processing Workflow**
    - **Trigger**: When a new email is received.
    - **Action**: Extract **email body & subject.**
    - **Send data to Azure Function.**
3. **Create Azure Function for Classification**

```python
from azure.ai.textanalytics import TextAnalyticsClient

def classify_email(email_text):
```

```
    client = TextAnalyticsClient(endpoint="<ENDPOINT>",
credential="<KEY>")

    result = client.analyze_sentiment([email_text])

    if "urgent" in email_text.lower():

        return "Important"

    elif "discount" in email_text.lower():

        return "Spam"

    else:

        return "General"
```

4. **Store Results in Cosmos DB**
    - Categorized emails are stored with metadata.
5. **Deploy & Test**
    - Send emails with **different subjects** and verify classification.

5. Conclusion

This project automates **email classification**, saving time for users and improving inbox management.

Project 10: Serverless URL Shortener with Azure Durable Functions

1. Project Scope

Create a **serverless URL shortener** like **bit.ly** using **Azure Functions and Durable Functions**.

2. Tools Used

- **Azure Durable Functions** (For workflow automation)
- **Azure Table Storage** (For storing short URLs)
- **Azure API Management** (For exposing URL shortening service)

3. Analysis Approach

1. **Users submit long URLs** via API.
2. **Durable Function generates a unique short URL.**
3. **Short URL is stored in Azure Table Storage.**
4. **When accessed, the short URL redirects to the original URL.**

4. Step-by-Step Implementation

1. **Create Azure Function for Short URL Generation**

```python
import random, string

from azure.data.tables import TableServiceClient


def generate_short_url():

    return ''.join(random.choices(string.ascii_letters + string.digits, k=6))


def main(req):

    long_url = req.params.get("url")

    short_code = generate_short_url()


    # Store in Azure Table Storage

    table_service = TableServiceClient.from_connection_string("<CONNECTION_STRING>")

    table_client = table_service.get_table_client("shorturls")


    table_client.create_entity({"PartitionKey": "URL", "RowKey": short_code, "LongUrl": long_url})
```

```python
    return {"short_url": f"https://short.ly/{short_code}"}
```

2. **Create Azure Function for URL Redirection**

```python
def main(req):
    short_code = req.params.get("code")

    # Retrieve from Table Storage
    table_service =
TableServiceClient.from_connection_string("<CONNECTION_STRING>")
    table_client = table_service.get_table_client("shorturls")

    entity = table_client.get_entity(partition_key="URL",
row_key=short_code)
    long_url = entity["LongUrl"]

    return {"statusCode": 301, "headers": {"Location": long_url}}
```

3. **Deploy to Azure Functions**
    - Deploy both functions.
    - Expose them via **Azure API Management**.
4. **Test URL Shortening**
    - Call the API:

Unset
```
curl -X GET
"https://<function-url>/shorten?url=https://www.example.com"
```

5. **Test Redirection**
   - Open the shortened URL in a browser.

5. Conclusion

This **serverless URL shortener** eliminates the need for a dedicated backend, making it scalable and cost-effective.

→ CLOUD DEVOPS TRAINING

REALTIME HANDON 6 MONTHS

WEEK 1-2	1 Year Platform Subscription Azure Cloud
WEEK 3-6	Azure Cloud & Ansible & Docker
WEEK 7-10	Kubernetes & DevOps CI/CD
WEEK 11-12	Terraform & Automation
WEEK 13 - 16	Azure DevOps & Azure DevSecOps Assesment & Q&A
WEEK 17 - 22	Resume Writing 30 RealTime Projects Job Search
WEEK 23 - 24	Secure Job as an Leader

500+ RealTime Handson Usecases

→ TRAININGS

WE ARE DIFFERENT

At CareerByteCode, we redefine training by focusing on real-world, hands-on experience. Unlike traditional learning methods, we provide step-by-step implementation guides, 500+ real-time use cases, and industry-relevant projects across cutting-edge technologies like AWS, Azure, GCP, DevOps, AI, FullStack Development and more.

Our approach goes beyond theoretical knowledge—we offer expert mentorship, helping learners understand how to study effectively, close career gaps, and gain the practical skills that employers value.

16+
Years of operations

91+
Countries worldwide

241 K Happy clients

Our Usecases Platform
https://careerbytecode.substack.com

Our WebShop
https://careerbytecode.shop

CareerByteCode
Learning Made simple

CareerByteCode
All in One Platform

STAY IN TOUCH WITH US!

Website

Our WebShop https://careerbytecode.shop
Our Usecases Platform https://careerbytecode.substack.com

E-mail
careerbytec@gmail.com

HQ address
Belgium, Europe

Social Media
@careerbytecode

Phone
+32 471 40 8908

For any RealTime Handson Projects

And for more tips like this

Follow

Like & ReShare

@careerbytecode