



A large smartphone screen is the central focus, set against a black background. On the screen, the words "TEDx Language" are displayed in large red letters. Below the title, three names and IDs are listed: "Biscaro Alessandro - 1087892", "Fabbris Thomas - 1086063", and "Gambirasio Lorenzo Umberto - 1087441". A cartoon illustration of a man with brown hair, wearing a blue t-shirt and brown pants, stands to the left of the screen, pointing his right index finger towards it. The overall design is clean and modern.

HOMEWORK 3

Piattaforme Cloud e  
Applicazioni Mobili

A.A 2024-2025

# API Get\_Watch\_Next\_By\_Id



L'API **Get\_Watch\_Next\_By\_Id** restituisce un array (in formato JSON) di video suggeriti dall'applicazione sulla base del talk che l'utente sta attualmente guardando. Il meccanismo di suggerimento dei talk è stato descritto all'interno della seconda presentazione.

L'API si aspetta di ricevere una **richiesta HTTP** con un body strutturato come segue:

## Body richiesta HTTP

```
1  {
2    "id": "66302"
3 }
```

## Risposta ottenuta dall'API

Request /Get_Watch_Next_By_Id	Latency ms 4033	Status 200
<b>Response body</b> <pre>{   "mainTalkTitle": "all your devices can be hacked",   "relatedTalks": [     {       "_id": "10459",       "description": "robin chase founded zipcar, the world's biggest car-sharing business. that was one of her smaller ideas. here she travels much farther, contemplating road-pricing schemes that will shake up our driving habits and a mesh network vast as the interstate.",       "duration": 805,       "slug": "robin_chase_the_idea_behind_zipcar_and_what_comes_next",       "speakers": "robin chase",       "tags": [         "technology",         "business",         "transportation",         "cities"       ],       "title": "the idea behind zipcar (and what comes next)",       "url": "https://www.ted.com/talks/robin_chase_the_idea_behind_zipcar_and_what_comes_next",       "publishedAt": "2008-01-31T03:23:00.000Z"     },     {       "_id": "38308",       "description": "the founder of 4chan, a controversial, uncensored online imageboard, describes its subculture, some of the internet \"memes\" it has launched, and the incident in which its users managed a very public, precision hack of a mainstream media website. the talk raises questions about the power -- and price -- of anonymity.",       "duration": 667,       "slug": "christopher_moot_poole_the_case_for_anonymity_online",       "speakers": "christopher \"moot\" poole",       "tags": [         "culture",         "computers",         "collaboration",         "activism",         "law",         "internet",         "government"       ],       "title": "the case for anonymity online",       "url": "https://www.ted.com/talks/christopher_moot_poole_the_case_for_anonymity_online",     }   ] }</pre>		

# UTILIZZO API *Get\_Watch\_Next\_By\_Id*



Questa API potrebbe essere sfruttata per mostrare sotto al video in esecuzione una **lista di talk collegati**, in modo da consentire all'utente di approfondire la sua conoscenza in un determinato ambito mentre sta raffinando l'apprendimento di una lingua straniera. In generale, un video non sempre condivide gli stessi tag con i talk ad esso relativi: questo potrebbe essere un vantaggio per analizzare una tematica sotto diversi punti di vista.

Il codice della Lambda Function (LF) è una variante di quello dell'API Get\_Talks\_By\_Tag ed i principali cambiamenti hanno interessato il suo handler. In particolare, viene **controllato se l'id specificato nel body è valido** e in caso affermativo prima viene interrogata la **collezione MongoDB tedx\_data\_cleaned** per recuperare i **related\_videos\_id** e successivamente vengono estratti i dettagli per ogni watch next trovato. Abbiamo inoltre notato che ogni video possiede al massimo 5/6 talk ad esso relativi per cui abbiamo abbandonato la funzionalità di paginazione dell'output offerta da **Get\_Talks\_By\_Tag**, attivata specificando i campi **doc\_per\_page** e **page** nel body della richiesta HTTP.

Qui è illustrato il codice dell'API descritta

```
try {
    await connect_to_db();
    console.log(`=> Attempting to fetch watch next for talk ID: ${body.id}`);

    const mainTalk = await talk.findById(body.id);

    if (!mainTalk) {
        console.log(`Main talk with ID '${body.id}' not found.`);
        return callback(null, {
            statusCode: 404,
            headers: { 'Content-Type': 'text/plain' },
            body: `Main talk with ID '${body.id}' not found.`
        });
    }

    const relatedIds = mainTalk.related_video_ids;
    let relatedTalksDocuments = [];

    if (!relatedIds || relatedIds.length === 0) {
        console.log(`No related video IDs found for talk '${mainTalk.title || body.id}'.
        Returning empty array for related talks.`);
    } else {
        console.log(`Found related video IDs: ${relatedIds.join(', ')}. Fetching related talks...`);
        relatedTalksDocuments = await talk.find({
            _id: { $in: relatedIds }
        }).select('-related_video_ids');
        console.log(`Found ${relatedTalksDocuments.length} related talk document(s.)`);
    }

    const responsePayload = {
        mainTalkTitle: mainTalk.title || 'N/A',
        relatedTalks: relatedTalksDocuments
    };

    return callback(null, {
        statusCode: 200,
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(responsePayload, null, 2)
    });
}

} catch (err) {
    console.error(`Error processing request for get_watch_next_by_id: ${err}`);
    return callback(null, {
        statusCode: err.statusCode || 500,
        headers: { 'Content-Type': 'text/plain' },
        body: `Could not fetch related talks. Error: ${err.message || 'Internal server error.'}`
    });
}
```

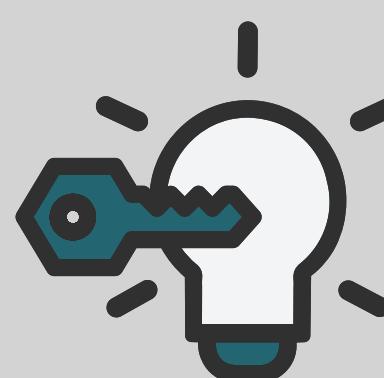
# API Generate\_Exercises\_TedxLanguage



L'API **Generate\_Exercises\_TedxLanguage** ha l'obiettivo di generare un'insieme di esercizi di completamento sulla base del transcript di un talk dato, per poi inserirli all'interno di una **collezione dedicata in MongoDB Atlas**. L'API si aspetta di ricevere una **richiesta HTTP** con un body strutturato come segue

## Body richiesta HTTP

```
1  {  
2  |   "talk_id": "66302"  
3  }
```



```
In [3]: # Modello QG-E2E  
  
hub = {  
    'HF_MODEL_ID': 'valhalla/t5-base-e2e-qg',  
    'HF_TASK': 'text2text-generation'  
}  
  
qg_model = HuggingFaceModel(  
    transformers_version='4.26',  
    pytorch_version='1.13',  
    py_version='py39',  
    env=hub,  
    role=role  
)  
  
qg_predictor = qg_model.deploy(  
    initial_instance_count=1,  
    instance_type='ml.m5.large',  
    endpoint_name='QG-E2E'  
)
```

```
In [ ]:  
  
from sagemaker.huggingface import HuggingFaceModel  
from sagemaker import get_execution_role  
import sagemaker  
  
role = get_execution_role()  
sess = sagemaker.Session()  
  
# Modello fill-mask  
hub = {  
    'HF_MODEL_ID': 'bert-base-uncased',  
    'HF_TASK': 'fill-mask'  
}  
  
huggingface_model = HuggingFaceModel(  
    transformers_version='4.26',  
    pytorch_version='1.13',  
    py_version='py39',  
    env=hub,  
    role=role  
)  
  
predictor = huggingface_model.deploy(  
    initial_instance_count=1,  
    instance_type='ml.m5.large',  
    endpoint_name='NLP-Exercise-Generator'  
)
```

Le domande vengono generate attraverso **due modelli NLP di Hugging Face** pre-addestrati su Amazon SageMaker, per cui il primo passo è stata la scelta dei modelli da utilizzare ed il loro deploy. Per il suggerimento delle migliori alternative per completare frasi che presentano parole mancanti, abbiamo scelto il modello bert-base-uncased, mentre per la generazione di domande a partire da un input testuale abbiamo ripiegato sul modello **valhalla/t5-base-e2e-qg**. Entrambi i modelli sono stati **deployati su un'istanza EC2**, creando degli opportuni endpoint.

# GENERATE EXERCISE



```
export default async function fetchTranscript(baseUrl) {
  const transcriptUrl = baseUrl.endsWith('/') ? `${baseUrl}transcript` : `${baseUrl}/transcript`;

  console.log(`Fetching transcript from: ${transcriptUrl}`);
  const response = await fetch(transcriptUrl);
  if (!response.ok) {
    throw new Error(`Failed to fetch transcript. Status: ${response.status}`);
  }
  const html = await response.text();
  const $ = cheerio.load(html);

  let transcriptText = '';

  const scriptTag = `$('script[type="application/ld+json"]');
  if (scriptTag.length > 0) {
    const jsonData = scriptTag.html();
    if (jsonData) {
      try {
        const data = JSON.parse(jsonData);
        if (data && data.transcript) {
          transcriptText = data.transcript;
        }
      }
    }
  }
  if (!transcriptText) {
    transcriptText = $('div[role="button"] [aria-disabled="false"] > span').map((i, el) =>
      $(el).text()).get().join(' ').replace(/\s\s+/g, ' ').trim();
  }
  if (!transcriptText) {
    throw new Error("Transcript content is empty or could not be found on the page.");
  }
  return transcriptText.slice(0,8192);
}
```

Il secondo passo è stato recuperare il transcript completo di un talk a partire dal suo URL registrato nel dataset dell'applicazione.

Abbiamo scoperto che l'indirizzo Web ottenuto concatenando "/transcript" all'URL di un talk punta ad una pagina HTML riportante la trascrizione integrale del video. La pagina HTML ricevuta in risposta viene passata a Cheerio per procedere con lo scraping. Innanzitutto, viene controllata la presenza del tag `<script type="application/ld+json">` poichè potrebbe contenere al suo interno i dati strutturati relativi al un talk (incluso il suo transcript); in caso non venga trovato il tag specificato sopra, viene prelevato il transcript navigando la struttura della pagina HTML.

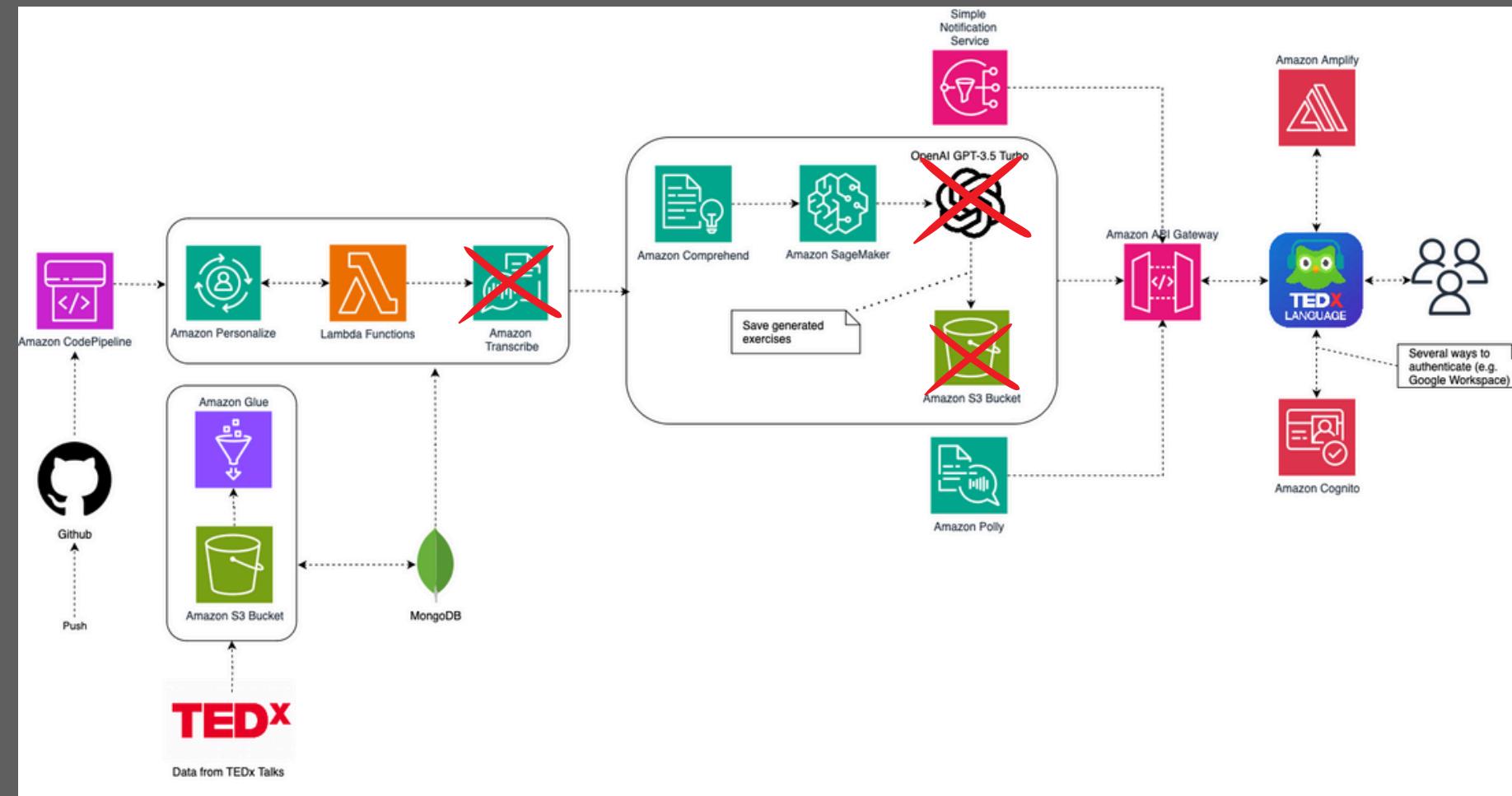
Infine, il testo estratto viene troncato prima di essere mandato in input ai modelli NLP per evitare che l'API Gateway vada in timeout durante l'esecuzione della LF una volta trascorsi 29 secondi dalla richiesta.

# MODIFICA DELL'ARCHITETTURA



Rispetto all'architettura che avevamo previsto inizialmente, abbiamo deciso di:

- Salvare gli esercizi generati in una collezione MongoDB Atlas, e non più su un bucket S3, in modo da avere un'unica sorgente dei dati per l'applicazione;
- Usare i modelli di NLP disponibili su SageMaker adatti al nostro caso d'uso (e non impiegare un modello personalizzato basato su OpenAI GPT-3.5 Turbo) per accelerare i tempi di sviluppo, al contempo non compromettendo la qualità del risultato finale;
- recuperare le trascrizioni dei talk direttamente dal sito di TED, e non attraverso un servizio come Amazon Transcribe, in quanto più congeniale alla nostra situazione in cui la trascrizione deve essere recuperata dall'URL del talk (e non dal video del talk stesso)





# CRITICITÀ TECNICHE

## COSTI SU SAGEMAKER

Elevati costi associati agli endpoint attivi su SageMaker

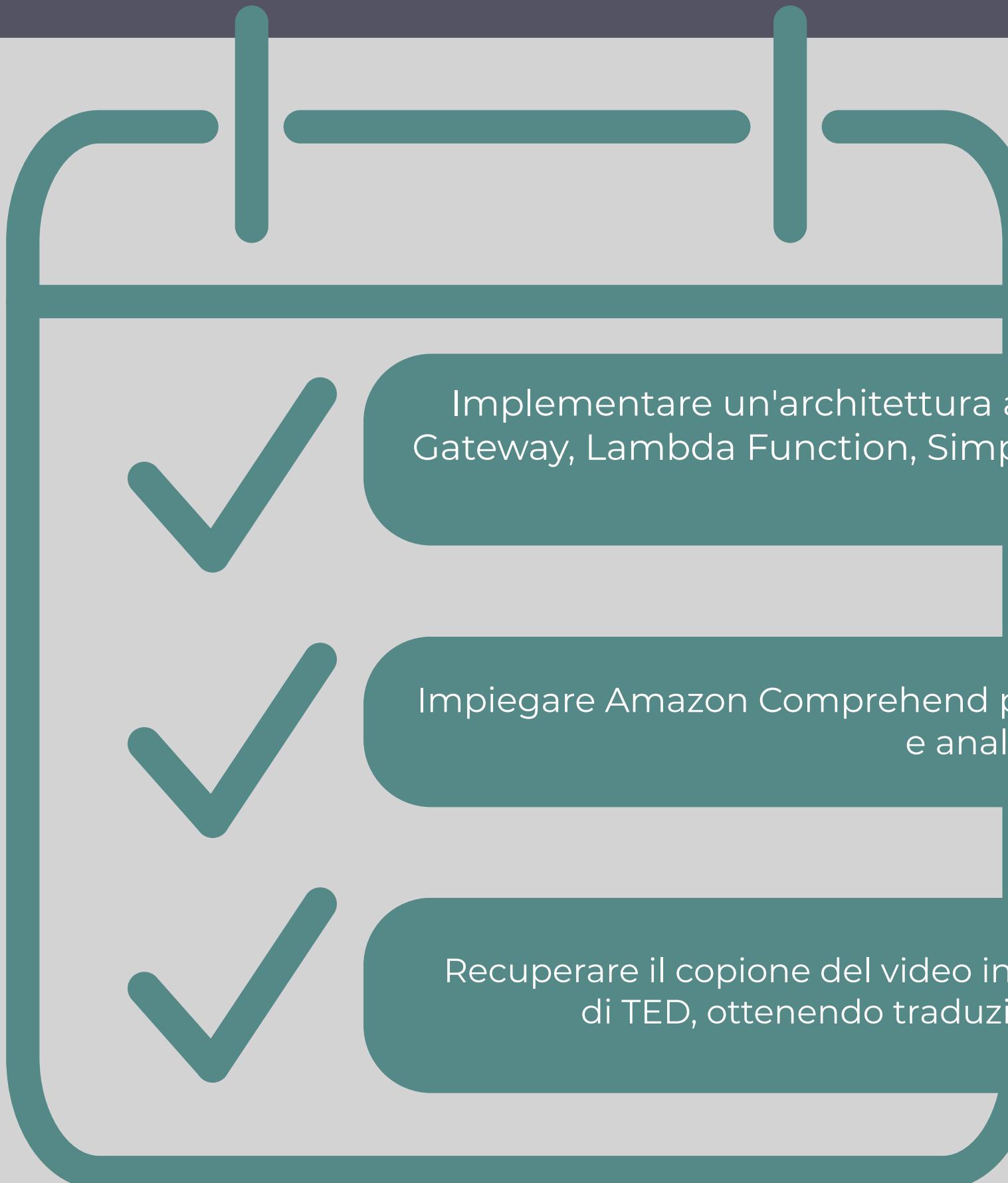
## FRAGILITÀ SCRAPING

Forte dipendenza dalla struttura attuale della pagina HTML di TED (che potrebbe variare senza preavviso)

## MODELLI NLP & LF

Rischio di oltrepassare il timeout previsto da API Gateway se si impiegano modelli di NLP in una Lambda Function

# POSSIBILI EVOLUZIONI



Implementare un'architettura asincrona per Generate\_Exercises\_TedxLanguage utilizzando API Gateway, Lambda Function, Simple Queue Service e DynamoDB disaccoppiando la richiesta iniziale dalla risposta finale

Impiegare Amazon Comprehend per individuare i momenti salienti di ogni video, rilevando frasi chiave e analizzando il "sentimento" della trascrizione

Recuperare il copione del video in un lingue diverse da quella in cui è tenuto il talk direttamente dal sito di TED, ottenendo traduzioni affidabili, senza ricorrere a servizi come Amazon Translate

# LINK utili



[GitHub](#)



[Trello](#)

