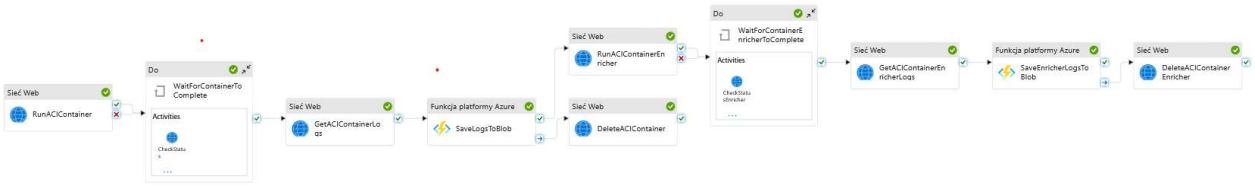
Pipeline do Dziennej Analizy Treści Podcastów

Kamil Małek



Cel projektu

Celem było stworzenie skalowalnego, modularnego i skonteneryzowanego pipeline'u do codziennego pozyskiwania, przetwarzania i analizy treści podcastów. Projekt miał także na celu integrację szerokiego zakresu zagadnień omawianych podczas ostatniego roku studiów. Choć pełen deployment w chmurze, ani ostatni etap pipelinu nie został ukończony, całość została przygotowana w sposób umożliwiający łatwą kontynuację.

Architektura Pipeline'u

Pipeline składa się z 4 głównych komponentów, oraz orkiestratora w postaci Azure Data Factory, a także pobocznych funkcji Azure zapisujących logi wykonanej pracy kontenerów efemerycznych. Proces uruchamiany jest codziennie o tej samej porze, zdefiniowanej na potrzeby projektu w github workflow na chwilę po deploymencie ADF pipeline.

1. Ingest – Pcaster

Stworzyłem skonteneryzowaną aplikację PCaster, która wykorzystuje bibliotekę Playwright do pozyskiwania rankingów podcastów z różnych źródeł. Pipeline codziennie zbiera dane o najpopularniejszych podcastach, a następnie standaryzuje je do jednolitego formatu. Dane są przetwarzane za pomocą Pandas i zapisywane w formacie Parquet na wcześniej skonfigurowanym storage, obsługującym zarówno Azure Blob Storage, jak i Amazon S3. PCaster opiera się na modularnej architekturze, gdzie różne scrapery dziedziczą po klasie abstrakcyjnej, co ułatwia rozbudowę systemu o kolejne źródła w przyszłości. Dzięki wykorzystaniu wielu źródeł aplikacja jest bardziej odporna na niedostępność poszczególnych serwisów oraz na zmiany w strukturze stron, które mogłyby wymagać modyfikacji scraperów.

Funkcje:

1. Skonteneryzowana aplikacja wykorzystująca Playwright do scrapowania rankingów podcastów z wielu źródeł.
2. Dane standaryzowane do jednego formatu i zapisywane jako pliki Parquet na storage (SeaweedFS, Azure Blob, S3).
3. Architektura oparta na klasie abstrakcyjnej umożliwia łatwą rozbudowę o kolejne źródła.
4. Wbudowana odporność na błędy dzięki redundantnym źródłom.

Technologie:

1. Playwright (kontener), używany do renderowania stron i pobierania danych.
2. Modularna architektura – każdy scraper dziedziczy po klasie abstrakcyjnej i może być łatwo dodany.
3. Redundancja źródeł – system działa nawet jeśli część źródeł jest niedostępna.
4. Pliki Parquet – format zoptymalizowany do dalszej analizy.

2. Enrichment - Enricher

Po pozyskaniu danych przez scraper następuje etap wzbogacenia danych o identyfikatory iTunes oraz PodcastIndex, których API wykorzystywane jest do pozyskania informacji o najnowszych odcinkach oraz linkach do źródeł audio w formacie mp3. Aplikacja Enricher również jest skonteneryzowana i składa się z kilku etapów przetwarzania danych. W pierwszej kolejności Enricher:

Funkcje:

1. Odczytuje pliki zapisane przez PCaster w storage.
2. Na podstawie różnych zescrapowanych rankingów tworzy jeden uśredniony Master Ranking z wykorzystaniem metody Bordy oraz odpowiednich wag przypisanych do źródeł, aby uniknąć dominacji pojedynczej platformy (np. Apple). Proces jest wysoko sparametryzowany, a jego parametry, takie jak limit pozycji w rankingu czy tematyka podcastów, można regulować za pomocą zmiennych środowiskowych kontenera. Do ułatwienia przetwarzania danych stworzono klasę dziedziczącą po pandasowym DataFrame, wykorzystując bibliotekę do rozmytego porównania fraz oraz normalizację i klasteryzację tytułów podcastów dla ich prawidłowej klasyfikacji.
3. Na podstawie master rankingu Enricher generuje kolejny DataFrame, przedstawiający listę ostatnich epizodów podcastów przypisanych do odpowiednich pozycji w rankingu, wraz z metadanymi i linkami do plików audio mp3. Zastosowano tu rozkład typu „round-robin", aby zapobiec faworyzowaniu epizodów z jednego źródła lub topowej części listy. Dane o odcinkach pozyskiwane są za pomocą API Podcasting Index oraz iTunes.
4. Dane tabelaryczne są ponownie zapisywane w formacie Parquet w storage, pod innym kluczem, oraz opatrzone stemplem czasowym w formacie JSON, zawierającym datę wykonania etapu.
5. Oprócz danych tabelarycznych Enricher tworzy plik wsadowy batch_job.json, zawierający linki do źródeł audio podcastów, docelowe lokalizacje do zapisu wyników transkrypcji oraz dodatkowe metadane ułatwiające identyfikację.

Technologie i funkcje:

1. Fuzzy matching – dopasowywanie nazw tytułów podcastów z wykorzystaniem rozmytego porównania tekstu.
2. Agregacja rankingów – redukcja biasu wynikającego z dominacji pojedynczego źródła.
3. Output JSON – plik sterujący dla kolejnych komponentów pipeline'u.

3. Transkrypcja – Whisperer

Do transkrypcji audio użyłem modelu whisper 'tiny' od OpenAI w wersji CPU, który okazał się w zupełności wystarczający. Tym razem aplikację, którą nazwałem Whisperer oparłem na bibliotece Celery, ze względu na to, że jest to najabardziej zasobożerny etap całego pipeline. Whisperer składa się z minimum 3 kontenerów: Submitter do przyjmowania zadań,
który może pracować w dwóch trybach: ciągłym nasłuchującym nadchodzących żądań z zadaniami zarówno przez komendy CLI jak i endpoint API (FastAPI), jak i tryb efemeryczny, w którym wywołuje się kontener w celu przetworzenia żądania zbioru zadań, po którym kontener zamyka się. Następnie jest Redis służący jako broker do kolejkowania zadań i przynajmniej jeden kontener Whisperer w trybie worker. Workery dzięki bibliotece Celery automatycznie pobierają nadchodzące zadania z brokera i biorąc pod uwagę, że trankrypcje są czasochłonne w porównaniu do reszty etapów pipeline - moża rozważyć skalowanie workerów na podstawie wielkości kolejki w brokerze.

Obraz dockera Whisperer zawiera wbudowany na etapie build model tiny whisper, dzięki czemu skrócony jest znacznie czas zimnego startu i unikamy dzięki temu każdorawowego pobierania dużego modelu podczas stawiania kontenerów, także Whisperer stworzony został z myślą o pracy w trybie efemerycznym.

Workery pobierają wewnętrznie do swojego kontenera plik audio z linku zadania, dzielą wewnętrznie na równej części "chunki", co znanie ogranicza i ujednolica zużycie procesora w procesie transkrypcji. Whisperer ma wbudowany mechanizm retry, który można dodatkowo sparametryzować za pomocą zmiennych środowiskowych na poziomie kontenera, takich jak limit czasu na wykonanie transkrypcji. W przypadku przerwanej pracy, o ile kontener nie zostanie zniszczony, po podjęciu ponownej próby worker zaczyna pracę od części pliku na któej zakończył transkrypcję. Po skończonej pracy wykonuje czyszczenie plików tymczasowych i ewentualnych pozostałości po nieudanych transkrypcjach.

Whisperer składa się z co najmniej trzech kontenerów:

Komponenty:

1. Submitter – przyjmuje zadania, pracując w trybie ciągłym (nasłuch CLI/API FastAPI) lub efemerycznym (uruchamiany on-demand do przetworzenia batchu i zamykany).

2. Broker – Redis do kolejkowania zadań.
3. Workery – kontenery z modelem Whisper, pobierające zadania z brokera i wykonujące transkrypcję.

Architektura i cechy:

1. Celery zarządza kolejką zadań i asynchronicznym przetwarzaniem.
2. Workery dzielą pliki audio na równe fragmenty (chunking), co stabilizuje zużycie CPU.
3. Wbudowany mechanizm retry pozwala na wznowienie pracy od miejsca przerwania, jeśli kontener pozostaje aktywny.
4. Możliwość skalowania liczby workerów w zależności od wielkości kolejki.
5. Tryb efemeryczny umożliwia uruchamianie kontenera tylko do wykonania konkretnego zadania, co optymalizuje zasoby.
6. Po zakończeniu pracy następuje czyszczenie plików tymczasowych i pozostałości po nieudanych transkrypcjach.
7. Model Whisper jest wbudowany podczas budowania obrazu, co znacznie skraca czas startu kontenera i eliminuje konieczność pobierania modelu w czasie uruchamiania.

Technologie:

1. Whisper (model tiny, CPU).
2. Celery (submitter, worker).
3. Redis (broker).
4. Docker (konteneryzacja, optymalizacja startu i skalowalność).

4. NLP – Spark NLP

Ostatnim etapem w pipelinie w założeniu była obróbka transkrybowanych danych z wykorzystaniem Spark NLP (John Snow Labs), jednak z powodu niewystarczającej ilości czasu zdążyłem zakreślić jedynie wstępny pipeline w skonteneryzowanym środowisku do tokenizacji, normalizacji i lematyzacji wynikowych tekstów w trybie lokalnym. W dalszej kolejności z możliwością rozszerzenia o Topic Discovery i głębszą analizę.

Funkcje:

1. Wstępna tokenizacja, normalizacja i lematyzacja w kontenerowym środowisku Spark NLP.
2. Gotowość do rozszerzenia o analizę semantyczną i wykrywanie tematów (Topic Discovery).

Stan: Proof-of-concept – napisane zostały proste skrypty do segmentacji i wstępnie skonteneryzowane środowisko SparkNLP. Powodzeniem zakończyły się testy pipe NLP na zsegmentowanym ówcześnie tekscie transkrypcji w formacie json (pliki w katalogu source). Skrypty do prostej segmentacji i NLP znajdują się w katalogu **libs/brick-process-nlp**

Kluczowe cechy

Główne zalety:

1. Modularność – każdy etap działa niezależnie, komunikując się plikami lub API.
2. Skalowalność – gotowość na duże wolumeny danych i elastyczne skalowanie.
3. Odporność na błędy – retry, redundantne źródła, transparentne logi.
4. Gotowość na rozwój – łatwe dodawanie nowych etapów analizy (np. topic modeling, sentyment, dashboard).

Środowisko lokalne – development stack

Projekt można uruchamiać częściowo, etapami pipeline'u, lokalnie w środowisku deweloperskim. Dostępne są kluczowe komponenty uruchamiane przez Docker Compose:

Komponenty:

1. SeaweedFS – storage kompatybilny z S3.
2. Redis – broker Celery.
3. Whisperer – submitter i worker do transkrypcji audio.
4. Spark NLP – kontener do przetwarzania NLP.

Skrypt development.sh umożliwia wygodne zarządzanie obrazami Docker (budowanie) oraz uruchamianie poszczególnych etapów pipeline'u z poziomu prostego menu tekstowego. Dane tymczasowe zapisywane są lokalnie w formatach Parquet i JSON, co ułatwia debugowanie. Architektura jest przygotowana do dalszego rozwoju i integracji kolejnych modułów.

Deployment w chmurze – Azure

Projekt został przygotowany pod deployment chmurowy, z wykorzystaniem Infrastructure-as-Code. Wdrożona automatyzacja obejmuje etap do enrichmentu z automatycznym zapisaniem logów w blob storage za pośrednictwem Azure Functions. Zarówno Pcaster jak i Enricher zaimplementowane są jako efemeryczne Azure Container Instances. Whisperer w zamierzeniu powinien być zdeployowany jako Azure Container Apps, z czego Submitter jako ACA Job, zaś workery skalowane do zera na podstawie długości kolejki w brokerze (Redis). Ostatni etap Spark NLP można wdrożyć jako kolejny, pojedynczy efemeryczny kontener, ponieważ nie jest mocno obciążony pracą, albo wykorzystać Databricks skonfigurowany ze sparkiem NLP i dostępem do storage.

Automatyzacja: GitHub Actions + Terraform

Funkcje automatyzacji:

1. Tworzenie infrastruktury (Resource Group, ACR, Blob Storage, ADF, App Services, ACI).
2. Obsługa danych wrażliwych przez GitHub Secrets.
3. CI/CD dla kodu i pipeline'ów danych.

Pipeline:

1. Azure Data Factory jako orchestrator (trigger harmonogramu).
2. Azure Container Instances dla efemerycznych workerów (np. Whisperer).
3. ACR jako rejestr obrazów kontenerów.

Orkiestrator Azure Data Factory

Zastosowany Pipeline Azure Data Factory realizuje automatyczne uruchamianie i zarządzanie kontenerami w Azure Container Instances (ACI) w sposób ephemeryczny, czyli tymczasowy.

Proces rozpoczyna się od uruchomienia kontenera ACI z zadaną konfiguracją obrazu "PCaster", zasobów oraz zmiennych środowiskowych, z wykorzystaniem Managed Service Identity (MSI) do uwierzytelniania. Po uruchomieniu kontenera pipeline czeka w pętli (Until) na jego zakończenie, okresowo sprawdzając status kontenera za pomocą żądań HTTP do API Azure.

Po zakończeniu pracy kontenera, pipeline pobiera logi wykonania kontenera i przekazuje je do zewnętrznej funkcji Azure Functions, która zapisuje je w bezpiecznym magazynie danych (np. Blob Storage). Następnie kontener jest automatycznie usuwany, co pozwala na optymalne wykorzystanie zasobów oraz unika zbędnych kosztów.

Analogiczny proces jest powtórzony dla drugiego kontenera o nazwie „Enricher", który wykonuje dodatkowe zadania wzbogacające dane. Cały pipeline wykorzystuje zaawansowane mechanizmy kontroli zależności między zadaniami, retry oraz timeout, co zapewnia stabilność i odporność na błędy.

Taki pipeline jest przykładem efektywnego wykorzystania ACI oraz Azure Data Factory do realizacji zautomatyzowanych, skalowalnych i bezstanowych procesów obliczeniowych w chmurze.

Terraform

Funkcje Terraform:

1. Tworzy Resource Group – logiczną grupę zasobów w Azure.
2. Tworzy Storage Account z włączonym Data Lake Gen2 do przechowywania danych.
3. Tworzy kilka kontenerów storage:
    - na pliki stanu Terraform (tfstate)
    - na dane projektu (Whisperer)
    - na logi z Azure Container Instances
4. Automatycznie ustawia sekrety GitHub Actions z nazwą resource group i storage account, aby workflow mógł się łączyć z Azure.
5. Rejestruje aplikację i service principal w Azure AD, nadaje im odpowiednie role (Contributor, Administrator) potrzebne do zarządzania zasobami.
6. Rejestruje dostawcę Microsoft.App, potrzebnego do Azure Container Apps.

7. Tworzy i konfiguruje Key Vault na sekrety i klucze do bezpiecznego przechowywania, w tym API kluczy i poświadczeń ACR.
8. Tworzy Azure Container Registry (ACR) do przechowywania obrazów kontenerów.

Całość pozwala na automatyzację i bezpieczne zarządzanie środowiskiem Azure w sposób powtarzalny i zintegrowany z procesami CI/CD w GitHub Actions.

Przed wykonaniem poleceń terraform plan i apply uruchamiam skrypty Bash, które automatycznie sprawdzają, czy kluczowe zasoby w chmurze (np. grupy zasobów, konta storage, rejestry kontenerów) już istnieją. Jeśli tak, skrypty importują je do stanu Terraform. Dzięki temu unikam ręcznego zarządzania istniejącą infrastrukturą i zapobiegam konfliktom podczas wdrożeń. Takie podejście zapewnia spójność i automatyzację, ułatwiając zarządzanie środowiskiem, nawet gdy część zasobów była tworzona niezależnie od Terraform.

Status projektu i dalszy rozwój

Z uwagi na ograniczenia czasowe, projekt nie został w pełni ukończony. Kluczowe komponenty (scraping, enrichment, transkrypcja) są gotowe i działają lokalnie oraz w środowisku Docker. Finalna faza NLP została rozpoczęta i przygotowana pod dalszy rozwój — wdrożony został szkielet NLP pipeline z Spark NLP (tokenizacja, normalizacja, lematyzacja).

Całość została zbudowana z myślą o łatwej rozbudowie oraz deploymentcie w chmurze (Azure), z wykorzystaniem Terraform i GitHub Actions. W razie potrzeby możliwe jest uruchomienie każdego etapu niezależnie (modularność). Projekt traktuję jako solidną bazę pod przyszłą kontynuację i produkcyjne wdrożenie.

Instrukcja obsługi repozytorium

Konfiguracja sekretów GitHub z lokalnej maszyny

Ten przewodnik opisuje, jak skonfigurować sekrety GitHub za pomocą lokalnych skryptów.

Wymagania wstępne

Upewnij się, że następujące narzędzia są **zainstalowane i skonfigurowane**:

1. GitHub CLI (`gh`)
2. Azure CLI (`az`)
3. W systemie **Windows** uruchamiaj skrypty za pomocą **Git Bash** lub kompatybilnego środowiska shell

Konfiguracja środowiska lokalnego

Krok 1: Sklonuj repozytorium na lokalną maszynę i przejdź do katalogu `[local]`

Krok 2: Utwórz i wypełnij plik `.env` zgodnie z wzorem zawartym w pliku `.example.env`

Krok 3: Uruchom skrypt `init.sh` w celu ustawienia sekretów i utworzenia service principal dla Azure

Uruchamianie skryptów Terraform do provisioning zasobów Azure

Uruchom skrypty w celu konfiguracji infrastruktury w następującej kolejności, używając GitBash. Skrypty te uruchamiają odpowiednie GitHub Workflows w repozytorium.

[UWAGA] Wszystkie skrypty są idempotentne i mogą być uruchamiane wielokrotnie (gh workflow sprawdza i importuje zasoby przed uruchomieniem skryptów terraform)

Kolejność wykonywania skryptów:

1. Konfiguracja grupy zasobów, OIDC dla logowania opartego na tokenach i storage z początkową przypisanymi rolami Ustala również zdalny backend dla stanu Terraform w nowo utworzonym Azure Storage

```
bash

./provision.sh --env bootstrap
```

2. Konfiguracja pozostałych wymaganych zasobów dla pipeline BigData

```
bash

./provision.sh --env main
```

3. Budowanie obrazów i wysyłanie ich do Azure Container Registry

```
bash

./provision.sh --env image_scraper
./provision.sh --env image_enricher
./provision.sh --env buildtest
# [opcjonalnie] dla celów testowych
#./provision.sh --env image_whisperer
```

4. Wdrożenie Azure Function do zapisywania logów efemerycznych Azure Container Instances do storage

```
bash

./provision.sh --env azfnlogs
```

5. Wdrożenie Azure Data Factory jako orkiestratora pipeline BigData do analizy podcastów

```
bash

./provision.sh --env adf
./provision.sh --env pipetest  # [opcjonalnie] test pipeline do debugowania
```

Skrypty dodatkowe:

```
bash

./provision.sh --env cleanacr  # Usuwa wszystkie obrazy z ACR oprócz ostatnio wysłanych dla każdego repozytorium
```

Lokalne środowisko deweloperskie i testowanie

Skrypt `./development.sh` w głównym katalogu repozytorium służy jako punkt wejścia do lokalnego uruchamiania poszczególnych usług pipeline:

[Local-S3-compatible-storage]

- Uruchom SeaweedFS z lokalnym storage S3

[Docker-images-builds]

- **Build image - PCaster** - scraper rankingów podcastów
- **Build image - Enricher** - wzbogacanie metadanych podcastów
- **Build image - Whisperer** - skalowalny transkryber audio

[Run-pipe-bricks]

- **Run PCaster with local S3 - Pipeline Stage 1 (Ingest)**
  - Apple Podcasts, Apple Platform, region US
- **Run PCaster with local S3 - Pipeline Stage 1 (Ingest)**
  - Pełen zakres
- **Run Enricher with local S3 - Pipeline Stage 2 (Process)**
  - Wzbogacanie metadanych podcastów i generowanie batch_job.json dla Whisperer

[Transcription-cluster]

- **Run Whisperer cluster** z lokalnym S3 i połączeniem Azure - Pipeline Stage 3 (Process)
  - Nasłuchiwanie na zgłaszane zadania
- **Run Whisperer Batch Job** - Pipeline Stage 4 (Process)
  - Uruchomienie zadania wsadowego Whisperer wygenerowanego przez Enricher

Podsumowanie etapów pipeline

Etap 1 (Ingest): PCaster - zbieranie rankingów podcastów
Etap 2 (Process): Enricher - wzbogacanie metadanych
Etap 3 (Process): Whisperer - transkrypcja audio
Etap 4 (Process): NLP - analiza treści (opcjonalnie)

Rozwiązywanie problemów

- Sprawdź, czy wszystkie wymagane narzędzia są poprawnie zainstalowane
- Upewnij się, że plik `.env` jest prawidłowo skonfigurowany
- W przypadku błędów Azure, sprawdź uprawnienia service principal
- Logi działania pipeline można znaleźć w Azure Data Factory i Azure Functions