UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea Magistrale

Progetto, sviluppo e orchestrazione di pipeline ETL tramite Apache Airflow in esecuzione su servizi Cloud

Relatore
prof. Riccardo Martoglia

Correlatore:
Andrea Giannetti

Laureando
Francesco Barbanti

Anno accademico 2021/2022

Ringraziamenti

In primis, ringrazio il Professore Riccardo Martoglia, relatore di questa tesi, per la competenza e la disponibilità mostrata durante tutto il mio percorso Universitario.
Un grazie di cuore è rivolto ad Ammagamma, per avermi accolto e dato l'opportunità di sviluppare questo progetto.
Un ringraziamento speciale al correlatore di questo elaborato, Andrea Giannetti, per le indicazioni, il supporto e i preziosi consigli senza i quali questo progetto non sarebbe stato possibile.
Una dedica speciale alla mia famiglia e i miei amici, che ogni giorno mi hanno sostenuto e incoraggiato al fine di raggiungere questo traguardo.

Parole chiave

Apache Airflow
Pipeline ETL
Cloud Computing
Amazon Web Services
AWS Glue

Indice

Elenco delle figure x

Elenco delle tabelle xii

Elenco dei listati xiii

I Studio delle tecnologie utili allo sviluppo e l'orchestrazione di pipeline ETL con servizi cloud 1

1 Apache Airflow 2
- 1.1 Pipeline ETL 2
- 1.2 Airflow 4
  - 1.2.1 Come nasce Airflow 4
  - 1.2.2 Concetti chiave 5
  - 1.2.3 Architettura di Airflow 6
  - 1.2.4 Esecutori di Airflow 8
  - 1.2.5 Operatori di Airflow 11
  - 1.2.6 Control flow 13
  - 1.2.7 Dynamic Task Mapping 14
  - 1.2.8 Airflow Webserver 15
  - 1.2.9 Quando usare Airflow 15

2 Cloud Computing 17
- 2.1 Le fondamenta del cloud computing 17
- 2.2 Cloud actor 18
- 2.3 Service model 19
- 2.4 Deployment model 20
- 2.5 Cloud provider 22
- 2.6 Tecnologie abilitanti del cloud computing 23
  - 2.6.1 Virtualizzazione 23
  - 2.6.2 Web 2.0 24
  - 2.6.3 Service Oriented Architecture 25
- 2.7 Amazon Web Services 25
  - 2.7.1 Infrastruttura globale di AWS 26
  - 2.7.2 Introduzione ai servizi di AWS 27

2.7.3 Amazon Virtual Private Cloud (Amazon VPC) 27
2.7.4 AWS Identity and Access Management (IAM) 28
2.7.5 AWS Secrets Manager 30
2.7.6 Amazon Elastic Container Registry (Amazon ECR) 31
2.7.7 Amazon Auto Scaling 31
2.7.8 Amazon Elastic Compute Cloud (Amazon EC2) 31
2.7.9 Amazon Elastic Container Service (Amazon ECS) 32
2.7.10 Amazon EC2 vs. Amazon Fargate 34
2.7.11 Amazon CloudFormation 35
2.7.12 Amazon Relational Database Service (Amazon RDS) 35
2.7.13 Amazon S3 36
2.7.14 Amazon Athena 37
2.7.15 Amazon CloudWatch 38
2.7.16 Amazon Glue 38
2.7.17 Amazon Managed Workflows for Apache Airflow (MWAA) 39

II Progetto e sviluppo delle pipeline ETL 42

3 Progettazione 43
3.1 Lo stato dell'arte 43
3.1.1 Descrizione delle pipeline ETL 44
3.1.2 Ambiente di esecuzione delle pipeline ETL 46
3.2 Code Refactoring 46
3.2.1 Creazione di asset comuni 47
3.2.2 Aggregazione delle pipeline ETL 48
3.3 Creazione dell'ambiente di sviluppo e di produzione 48
3.4 Valutazioni sui motori di gestione di workflow 49
3.4.1 Analisi sul costo dei servizi 50
3.4.2 Scelta del servizio 50
3.5 Architettura delle pipeline rivisitate 51

4 Implementazione 53
4.1 Infrastruttura 53
4.1.1 Cluster ECS 54
4.1.2 Airflow control plane 55
4.1.3 Task Fargate 56
4.2 DAG 58

5 Valutazione dei risultati ottenuti 61
5.1 Code refactoring 61
5.1.1 Halstead complexity 61
5.1.2 Cyclomatic complexity 62
5.1.3 Cognitive Complexity 63
5.1.4 Confronto della qualità del codice tra le diverse pipeline ELT 63
5.2 Orchestrazione tramite Apache Airflow 69

5.2.1 Costo computazionale 69
5.3 Metriche non quantitative 74

Riferimenti bibliografici 78

Elenco delle figure

1.1 Struttura di una generica pipeline ETL . . . 3
1.2 Deadlock in un grado diretto ciclico . . . 6
1.3 Architettura di Apache Airflow. Immagine tratta da [8] . . . 7
1.4 Panoramica di caricamento ed esecuzione di un DAG su Airflow . . . 8
1.5 Diagramma di sequenza di esecuzione di un DAG con l'esecutore Kubernetes. Immagine tratta da [11]. . . . 10
1.6 Diagramma di sequenza di esecuzione di un task con l'esecutore Celery. Immagine tratta da [13]. . . . 12
1.7 Esempio di branching in un DAG . . . 14
1.8 Screenshot che riprende una lista di DAGs dal webserver. Immagine tratta da [14] . . . 15
1.9 Screenshot che riprende un workflow dalla UI. Immagine tratta da [14] . . 16

2.1 Modelli di distribuzione del cloud computing . . . 21
2.2 Panoramica del cloud computing. Immagine ispirata alla Figura 1 di [17] . 21
2.3 Magic quadrant dei cloud provider nel 2021. Immagine tratta da [18] . . . 22
2.4 Google trends dei principali cloud provider . . . 23
2.5 Componenti principali della virtualizzazione . . . 24
2.6 Infrastruttura globale di AWS . . . 27
2.7 Servizi AWS utilizzati dalle pipeline ETL del progetto . . . 28
2.8 Differenza di costo per diverse opzioni di acquisto di una macchina EC2 . . 33
2.9 Esempio di Auto Scaling Group con un Capacity Provider in un cluster ECS 34
2.10 Architettura di AWS Glue. Immagine che prende spunto da [26] . . . 39
2.11 Architettura di AWS MWAA. Immagine tratta da [27] . . . 40

3.1 Struttura delle pipeline ETL mensili . . . 45
3.2 Struttura delle pipeline ETL annuali . . . 45
3.3 Ambiente di esecuzione delle pipeline ETL . . . 46
3.4 Struttura della pipeline ETL mensile dopo il refactoring . . . 48
3.5 Struttura della pipeline ETL annuale dopo il refactoring . . . 49
3.6 Architettura AWS delle pipeline rivisitate . . . 51

5.1 Halstead complexity delle pipeline ETL . . . 65
5.2 Halstead complexity dei primi 10 script delle pipeline ETL . . . 66
5.3 Complessità ciclomatica delle pipeline ETL . . . 66

5.4 Distribuzione delle classi di complessità ciclomatica delle pipeline ETL . . 67
5.5 Complessità cognitiva delle pipeline ETL 68
5.6 Distribuzione del costo dei job della pipeline in esecuzione su Airflow . . . 72
5.7 Costo dell'architettura di Airflow e di Glue all'aumentare del numero di pipeline ETL mensili 73
5.8 Costo dell'architettura di Airflow e di MWAA all'aumentare del numero di pipeline 74

Elenco delle tabelle

1.1 Panoramica di alcuni gestori di flussi di lavoro e delle loro caratteristiche principali . . . 5
1.2 Trigger rules di Airflow . . . 14

2.1 Numero di servizi AWS suddivisi per categoria . . . 29
2.2 Gestione di un database su infrastruttura on-premises e su Amazon EC2. Tabella tratta da [24]. . . . 36
2.3 Gestione di un database su Amazon EC2 e su Amazon RDS. Tabella tratta da [24]. . . . 37

3.1 Gestione di un database su Amazon EC2 e su Amazon RDS . . . 50

4.1 Capacità computazionali delle classi di task Fargate . . . 56
4.2 Parametri dell'operatore ECS di Airflow . . . 60

5.1 Classi di complessità nella Cyclomatic complexity . . . 63
5.2 . . . 64
5.3 Configurazioni di capacità computazionale dei job delle pipeline ETL . . . 69
5.4 Confronto dei tempi e costi di esecuzione della pipeline mensile in esecuzione su Fargate e Glue . . . 70
5.5 Confronto dei tempi e costi di esecuzione della pipeline annuale in esecuzione su Fargate e Glue . . . 71
5.6 Tempi di esecuzione del job *calculate_summable_kpi* parallelizzato . . . 71
5.7 Costo dei servizi dell'architettura di Airflow . . . 73

Elenco dei listati

4.1 Creazione di puntatori alla VPC e alla rispettiva subnet 54
4.2 Creazione cluster ECS 54
4.3 Creazione Auto Scaling Group e Capacity Provider 55
4.4 Definizione delle classi di tasks Fargate 56
4.5 Definizione del task elaborate_navigation_data 58
4.6 Definizione delle dipendenze della pipeline annuale 59

Introduzione

Da qualche anno a questa parte, il Cloud Computing non è più solamente una soluzione teorica al problema dell'enorme aumento dei dati da processare e dei servizi Internet da gestire, bensì è diventato un modello di elaborazione basato su risorse condivise e utilizzabili dinamicamente.
Proprio per questo motivo, e grazie anche alla nascita di piattaforme di cloud computing (come per esempio Amazon Web Services), ad oggi la stragrande maggioranza delle aziende fa uso dei servizi e delle infrastrutture messe a disposizione dalle piattaforme Cloud a discapito delle soluzioni *on-premise* gestite e controllate privatamente.
Lo scopo di questa trattazione è di descrivere il progetto sviluppato in collaborazione con Ammagamma S.r.l [1], una società che costruisce le architetture e gli applicativi necessari alla gestione e all'estrazione del valore dai dati attraverso soluzioni matematiche. In particolare, tale progetto si pone come obiettivo quello di progettare, orchestrare ed ridistribuire delle pipeline ETL che in origine eseguivano sul servizio AWS Glue su un'istanza di Apache Airflow rilasciata sull'architettura cloud di AWS.
Questa trattazione è articolata in cinque capitoli ed è suddivisa in due parti principali.
La parte I ha il compito di descrivere le tecnologie utili allo sviluppo e all'orchestrazione delle pipeline ETL prese in esame in questo elaborato tramite servizi Cloud. Più nello specifico, nel primo capitolo si affronta Apache Airflow, una piattaforma di schedulazione e monitoraggio di workflows batch-oriented. Il secondo capitolo approfondisce il tema del cloud computing esponendo i principali servizi di AWS utili allo sviluppo delle pipeline ETL.
La parte II si concentra maggiormente sul progetto e sullo sviluppo effettivo delle pipeline ETL. In particolare, il terzo capitolo descrive la fase di progettazione dell'architettura cloud e della struttura delle pipeline. La quarta sezione espone la fase di implementazione allegando il codice sorgente più significativo. A conclusione, il quinto capitolo ha come obiettivo quello di analizzare e descrivere la qualità dei risultati ottenuti.

Parte I

Studio delle tecnologie utili allo sviluppo e l'orchestrazione di pipeline ETL con servizi cloud

Capitolo 1

Apache Airflow

Questo primo capitolo si pone come obiettivo quello di analizzare Apache Airflow, una piattaforma di schedulazione e monitoraggio di flussi di lavoro. Inoltre, oltre ad una sua descrizione generale, si vuole anche esporre in maniera approfondita gli elementi che compongono la sua architettura, non prima di aver affrontato il concetto di pipeline ETL.

1.1 Pipeline ETL

Prima di descrivere Apache Airflow, è necessario introdurre il concetto di pipeline.
Al giorno d'oggi i dati sono di importanza fondamentale in tutti i processi aziendali, e in questo contesto i sistemi ETL trovano applicazione in diversi casi d'uso, come per esempio l'inserimento dei dati in un data warehouse. Questo perché, se opportunamente implementate, le pipeline ETL permettono di estrarre i dati dalle sorgenti dati e di elaborarli al fine di creare un data layer su cui costruire future applicazioni.
Quindi, possiamo definire i sistemi ETL come l'insieme dei processi che compongono una data pipeline che hanno lo scopo di estrarre (*Extract*), trasformare (*Transform*) e caricare (*Load*) i dati in un data warehouse, dal quale possono essere utilizzati senza ulteriori trasformazioni. Anche se tecnicamente una pipeline ETL è anche una data pipeline, spesso non vale la relazione inversa. Questo perché, con data pipeline si descrive qualsiasi insieme di processi che muove dati da un sistema ad un altro, non è necessario che queste informazioni subiscano delle trasformazioni. Invece, come detto precedentemente, una pipeline ETL si compone di un insieme ordinato di processi di estrazione, trasformazione e caricamento: questo tipo di pipeline include sempre una fase di elaborazione. Inoltre, un ulteriore differenza tra i due flussi di processi riguarda la modalità di esecuzione: se da un lato le pipeline ETL spesso elaborano chunks di dati ad intervalli regolari (esecuzione batch), dall'altro lato la maggior parte delle volte le data pipeline eseguono come processi real-time per elaborare gli eventi nel momento in cui sono registrati.
Anche se tipicamente i sistemi ETL possono consumare molte risorse computazionali del data warehouse, essi apportano un valore significativo ai dati grezzi. Infatti:

- rimuovono dati duplicati e integrano dati mancanti;
- strutturano i dati per essere utilizzati da applicazioni per l'utente finale;

- forniscono delle misure di confidenza dei dati;
- assicurano la qualità dei dati;
- adattano i dati provenienti da più fonti per utilizzarli insieme.

Oltre agli stage di estrazione trasformazione e caricamento, spesso si fa riferimento ad un ulteriore fase della pipeline, quella di *cleaning*. In Figura 1.1 è mostrata la struttura di una generica pipeline ETL, da cui si possono fare alcune considerazioni. In particolare, la fase

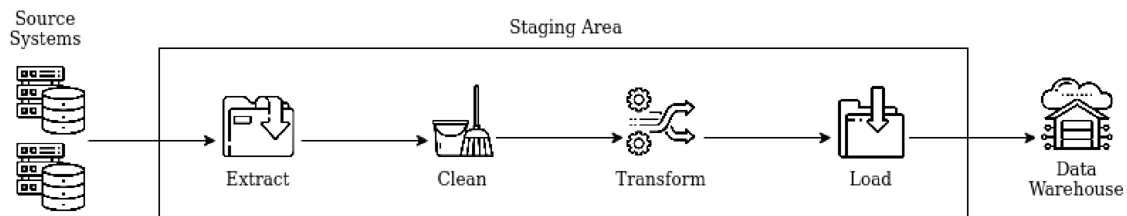


Figura 1.1: Struttura di una generica pipeline ETL

di estrazione ottiene i dati grezzi da una o più fonti eterogenee, come per esempio APIs, sensori o database. Esistono tre modalità di estrazione dati:

- estrazione parziale senza notifiche di aggiornamento: vengono estratti i dati seguendo delle condizioni di caricamento. Per esempio, avendo uno scheduler giornaliero verrebbero estratti solamente i dati generati lo stesso giorno;
- estrazione parziale con notifiche di aggiornamento: questo metodo si basa su una strategia di notifica in cui la sorgente fornisce una notifica di aggiornamento per eseguire il processo di estrazione quando un record o un set di dati viene modificato. Questo è il metodo di estrazione più semplice;
- estrazione completa: vengono estratti tutti i dati dalla sorgente senza applicare alcun tipo di condizione.

Una volta conclusa l'estrazione i dati grezzi vengono solitamente memorizzati in flat files o in tabelle relazionali. Anche se questa fase è dedicata esclusivamente all'acquisizione dei dati, può capitare che a tali dati vengano applicate delle semplici trasformazioni. Infatti può essere conveniente risolvere alcuni problemi di legacy derivanti dal formato dei dati già in questa fase della pipeline, lasciando alle fasi successive le trasformazioni più complesse e importanti. Alcuni esempi di trasformazioni applicate nella fase di estrazione sono la conversione da decimale a intero e la conversione da codifica binaria a codice ASCII.
In molti casi il livello di qualità dei dati accettabile per le sorgenti è diverso dalla qualità richiesta dal data warehouse. Per questo è necessaria anche una fase di pulizia, la quale ha il compito di rimuovere il rumore dai dati. Alcune delle problematiche risolte in questo step sono:

- dati duplicati;
- dati mancanti;

- dati inconsistenti;
- dati non necessari;
- dati con formato errato.

Il risultato della fase di cleaning spesso è anche memorizzato permanentemente, dal momento che la fase di trasformazione successiva è la parte più complessa della pipeline e a volte anche irreversibile.
Lo step di trasformazione ha come obiettivo quello di convertire i dati nel formato richiesto dalle applicazioni che fanno uso del data warehouse. Questo processo è complicato dalla presenza di multiple fonti dati eterogenee, che quindi richiedono l'implementazione di una logica di integrazione non banale. Inoltre a questo stadio si applicano dei filtri, in modo da mantenere solamente le informazioni utili.
La finalità dei sistemi ETL è di rendere i dati pronti ad essere interrogati. Per raggiungere questo scopo è necessario caricare i dati nel data warehouse finale, e questo è reso possibile dall'ultimo stadio della pipeline.
Sempre in Figura 1.1 si può notare come tutti gli step della ETL sono racchiusi all'interno della cosiddetta *staging area*. Infatti qualsiasi sistema ETL deve mantenere i dati in diversi storage permanenti o semi-permanenti; con staging area si identificano zone di archiviazione localizzate tra la sorgente dati e il data warehouse finale. Questo tipo di storage è fondamentale in quanto consente di effettuare backup schedulati e di conseguenza anche operazioni di roll-back in caso di problemi. Per tale ragione, è consigliato creare quattro staging area (una per ogni fase della pipeline), in modo da memorizzare semi-permanentemente i dati elaborati ad ogni stadio.
Progettare una pipeline ETL può essere estremamente complesso e non privo di errori, come affermano gli autori del paper "QoX-driven ETL design: Reducing the cost of ETL consulting engagements" [2], nel quale è affermato che il design e l'implementazione di sistemi ETL costituiscono il 70% del lavoro nei progetti di data warehouses.
Con il passare del tempo i sistemi ETL sono diventati sempre più complessi a causa dell'aumento del numero di dati a disposizione e delle richieste applicative. Per risolvere questo problema negli anni sono nati diversi strumenti di organizzazione dei job ETL, i quali organizzano questi ultimi in flussi di lavoro (in inglese *workflows*). Attualmente in commercio si trovano molti tool, come mostrato in tabella 1.1. Sicuramente uno dei più utilizzati e più supportati dalla community open source è proprio Apache Airflow.

1.2 Airflow

Lo scopo di questa sezione è elencare le caratteristiche più importanti di Apache Airflow partendo dalla sua storia, descrivendo i suoi componenti nello specifico ed esaminando i casi d'uso migliori.

1.2.1 Come nasce Airflow

Airflow [3] nasce nel 2014 come progetto Open Source avviato da Maxime Beauchemin, un dipendente di Airbnb. Infatti in questi anni il noto portale statunitense stava crescendo

Nome	Originato da	Implementata in	Workflow definiti in	Scheduling	Interfaccia utente	Piattaforma di installazione	Scalabilità orizzontale
Airflow	Airbnb	Python	Python	Si	Si	Ovunque	Si
Argo	Applatix	Go	YAML	Si	Si	Kubernetes	Si
Azkaban	Linkedin	Java	YAML	Si	Si	Ovunque	
Conductor	Netflix	Java	JSON	No	Si	Ovunque	Si
Luigi	Spotify	Python	Python	No	Si	Ovunque	Si
Nifi	NSA	Java	UI	Si	Si	Ovunque	Si
Oozie		Java	XML	Si	Si	Hadoop	Si

Tabella 1.1: Panoramica di alcuni gestori di flussi di lavoro e delle loro caratteristiche principali

molto rapidamente e si trovava ad affrontare ogni giorno una maggiore quantità di dati interni. Per realizzare l'ambizione di diventare un'organizzazione completamente *data-driven*, ossia una società che basa le proprie azioni a seguito dell'analisi dei dati raccolti, aveva necessità di automatizzare regolarmente i processi implementando batch di lavori programmati e aumentando il numero di data scientist e analisti. Per supportare questa transizione, Airbnb necessitava di uno strumento di orchestrazione di workflows robusto e particolarmente efficiente.
Airflow prende vita come progetto open source e il primo commit su GitHub risale al 2015. Dopodichè il progetto è entrato a far parte dell'incubatore ufficiale della Fondazione Apache [4] nell'aprile 2016, il quale lo ha annunciato come progetto di tipo *top-level* nel 2019 [5].

1.2.2 Concetti chiave

In primis è necessario dare una vera e propria definizione di Airflow. In particolare, Apache Airflow è una piattaforma di schedulazione e monitoraggio di workflows batch-oriented e ad oggi è diventato uno stardard de facto per l'orchestrazione di data pipelines. La piattaforma si propone di essere:

- **altamente scalabile**: la sua architettura è modulare e fa uso di una coda di messaggi per orchestrare un numero arbitrario di worker. Grazie a questa caratteristica, a livello teorico, è garantita la scalabilità orizzontale senza limiti; non ci sono vincoli alla complessità delle pipelines;

- **estensibile**: essendo un progetto open source, qualsiasi utilizzatore può implementare i propri componenti custom o estendere le librerie già presenti per raggiungere il livello di astrazione desiderato. Grazie alla sua popolarità ad oggi si può fare affidamento su una community numerosa, la quale fornisce periodicamente operatori e worker non ufficiali;

- **dinamico**: in Airflow le pipeline sono definite come codice Python, consentendo la generazione dinamica delle stesse;

- **agnostico rispetto agli strumenti**: dal momento che l'implementazione Airflow contiene al proprio interno varie REST API [6], si può connettere con ogni altro tool che permetta una interazione mediante APIs;

- **elegante**: essendo i flussi di lavoro definiti tramite codice python, essi sono anche molto chiari e espliciti. Inoltre tramite un motore di template di nome Jinja [7] è possibile parametrizzare i diversi job del workflow, consentendo di riutilizzare script in workflow differenti.

Come qualche altro orchestratore, Airflow permette di definire i workflow come DAGs di task. In particolare DAG è l'acronimo di *Directed Acyclic Graph*, ossia un grafo diretto che non presenta cicli. Ogni DAG rappresenta un workflow che si vuole eseguire, in cui i nodi corrispondono ai task mentre gli archi sono le dipendenze dei vari task. Condizione necessaria alla schedulazione del flusso è che il grafo sia, oltre che diretto, anche aciclico; quest'ultima condizione è molto importante dal momento che impedisce di incorrere in dipendenze circolari tra i task e che porterebbe ad un deadlock. In Figura 1.2 è mostrato un esempio di quanto appena detto, in cui i task 1, 2 e 3 hanno terminato l'esecuzione, mentre il task 4 e il 5 non saranno mai schedulati dal momento che le loro dipendenze creano un ciclo: entrambi i tasks aspettano la terminazione dell'altro. Come già anticipato, in Airflow

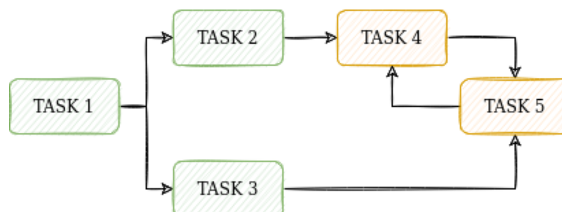


Figura 1.2: Deadlock in un grado diretto ciclico

i DAG sono definiti tramite codice Python in opportuni file. Oltre alla definizione dei task e delle loro relazioni, questi file tipicamente contengono anche metadati aggiuntivi che informano Airflow di come dovrebbe essere schedulato il grafo, la modalità di esecuzione, i parametri dei job e così via. Ogni DAG non si preoccupa di ciò che accade all'interno dei task che lo compongono, bensì solo di come eseguirli: l'ordine in cui eseguirli, quante volte ritentare in caso di fallimento, se hanno dei timeout e così via.
Un vantaggio di definire i DAG di Airflow come codice Python è che questo approccio programmatico offre una grande flessibilità nella costruzione dei flussi di lavoro. Ad esempio è possibile utilizzare costrutti Python per generare dinamicamente task opzionali in base a determinate condizioni o addirittura generare interi DAG basati su metadati file di configurazione esterni. Inoltre, volendo, è possibile anche creare DAG come composizione di uno o più grafi.

1.2.3 Architettura di Airflow

Ad alto livello, Airflow è organizzato in cinque componenti principali (come mostrato in Figura 1.3):

- **Airflow scheduler**: rappresenta il cuore di Airflow, dove risiede tutta la logica. Infatti, come ci si può aspettare, lo scheduler è un demone con il compito di schedulare i job del flusso di lavoro. Per fare questo analizza il DAG controllando i suoi intervalli di scheduling e avvia l'esecuzione del workflow assegnando i task all'esecutore. Di default, lo scheduler controlla se i DAG devono essere eseguiti una volta al minuto, ma è possibile modificare tale intervallo agendo sul file di configurazione di Airflow;
- **Airflow executor**: l'esecutore si occupa di eseguire i task. In particolare è il meccanismo che definisce come le risorse di calcolo disponibili devono essere usate per eseguire i task;
- **Airflow worker**: processo che esegue un task del workflow nelle modalità definite dall'esecutore. A seconda dell'esecutore scelto, è possibile che i worker facciano o meno parte dell'infrastruttura Airflow. Infatti è possibile eseguire i task direttamente tramite l'esecutore localmente (quindi senza distribuire il lavoro ai worker) o assegnare i task a uno o più worker remoti;
- **Airflow webserver**: un server Flask che permette di visualizzare i DAG, la loro storia di esecuzione, i trigger associati e molto altro;
- **Airflow Database**: un database utilizzato dal webserver, scheduler e esecutore per memorizzare i metadati di tutti i DAG e dei relativi task. In genere si utilizza di un database Postgres, ma sono supportati anche MySQL, MsSQL e SQLite.

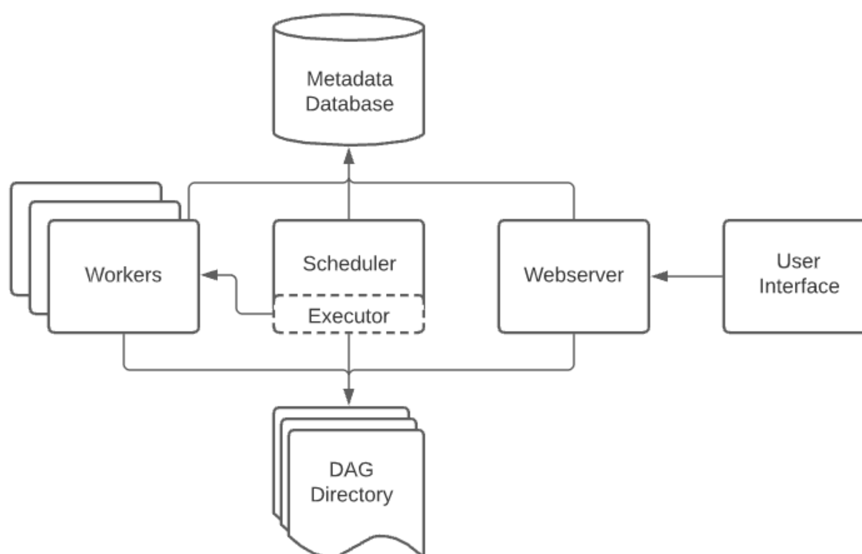


Figura 1.3: Architettura di Apache Airflow. Immagine tratta da [8]

La Figura 1.4 rappresenta il processo di caricamento ed esecuzione di un DAG su Airflow. In particolare, una volta che gli utenti hanno definito i flussi di lavoro come DAG in file Python, essi vengono letti dallo scheduler per estrarre i task corrispondenti, le dipendenze e gli intervalli di scheduling. Dopodiché lo scheduler controlla se il workflow deve essere

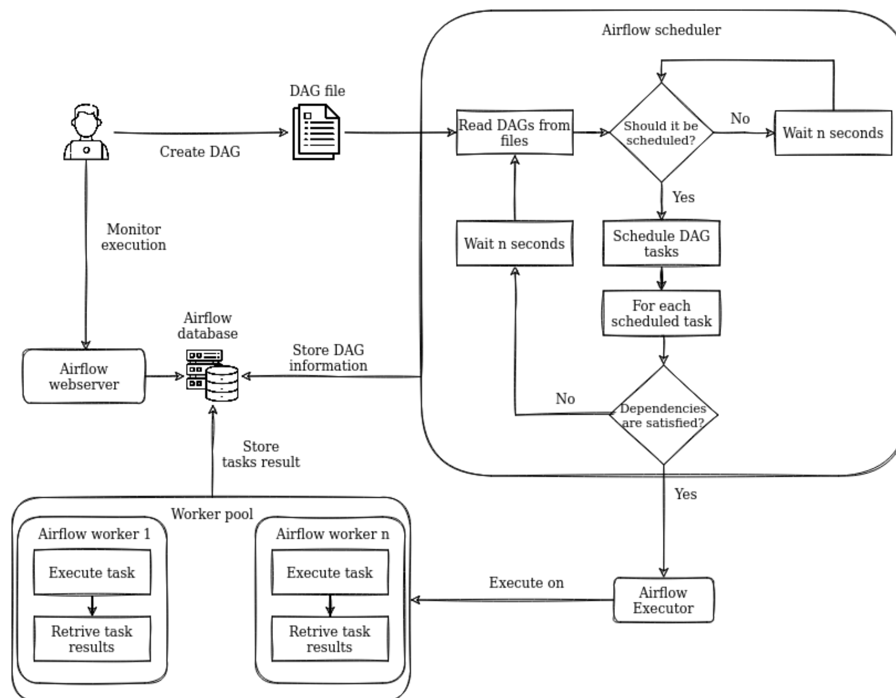


Figura 1.4: Panoramica di caricamento ed esecuzione di un DAG su Airflow

eseguito e, se sì, inoltra all'esecutore i task che hanno le dipendenze soddisfatte. A questo punto lo scheduler inoltra i task all'esecutore che, sulla base delle risorse a disposizione e seguendo una logica spesso di tipo FIFO (*First In First Out*), assegna i task ai worker. L'utente è in grado di monitorare lo stato dei singoli task e dell'intero workflow tramite il webserver, il quale ottiene le informazioni interrogando il database condiviso con lo scheduler e i worker.

1.2.4 Esecutori di Airflow

Come già accennato in precedenza, l'esecutore è il responsabile dell'esecuzione dei tasks. Infatti dopo che riceve un segnale dallo scheduler inizia ad allocare risorse e mette i tasks in una coda. Nel momento in cui una risorsa è disponibile, l'esecutore estrae un task e lo affida ad un worker, la cui risorsa sarà deallocata nel momento in cui il task termina l'esecuzione.
In particolare essi possono essere suddivisi in due categorie principali: esecutori locali ed esecutori remoti. La differenza è che i primi eseguono i tasks localmente (ovvero internamente allo scheduler), mentre i secondi delegano il lavoro ad un pool di worker remoti. Come ci si può immaginare, gli esecutori locali non sono adatti ad ambienti di produzione bensì sono spesso utilizzati per debuggare e creare ambienti di testing.
La conseguenza di essere una piattaforma open-source, flessibile e altamente estendibile si ritrova nella possibilità di scelta dell'esecutore da un elevato numero di possibilità. Infatti, ad oggi, è possibile scegliere l'esecutore che meglio si adatta alle richieste e all'infrastruttura dell'ambiente di produzione.

Questa sezione ha lo scopo di descrivere ad alto livello gli esecutori più utilizzati in Airflow.

Debug executor

Alle volte può essere molto complesso investigare l'errore di un task che esegue in un container, dal momento che spesso è necessario accedere all'interno dello stesso e modificare lo script Python per aggiungere eventuali punti di breakpoint. Questa procedura è facilitata dall'esecutore chiamato *Airflow debug executor*, il quale consente di debuggare il codice direttamente dall'IDE di programmazione.
Per abilitare questo esecutore è necessario aggiungere il blocco *main* alla fine del file DAG per renderlo eseguibile e settare la configurazione di avvio del debugger all'interno dell'IDE.

Local executor

Come dice il nome, l'esecutore locale opera sullo stesso nodo dello scheduler esemplificando l'architettura a singolo nodo. Tale esecutore avvia i task in parallelo sulla stessa macchina, abilitando quindi un tipo di elaborazione parallela ma non distribuita (proprio come un comune PC); è compito del sistema operativo sottostante gestire la concorrenza dei processi del workflow. Nel momento in cui si utilizza questo tipo di executor è necessario scegliere tra due strategie implementate:

- parallelismo illimitato: con questa strategia l'esecutore crea un processo ogni volta che riceve un task da eseguire. Con questo tipo di approccio non è neanche necessario istanziare una coda in cui inserire i task non ancora eseguiti;

- parallelismo limitato: con questa strategia l'esecutore crea un numero di processi equivalente ad un valore specificato all'avvio del workflow. Quindi, una volta che il worker è pronto per eseguire, informa l'esecutore il quale inoltra il task. Allo stesso modo, nel momento in cui l'esecutore ha terminato i task da eseguire, invia ai worker un token di terminazione. In questo caso è necessario istanziare una coda.

L'esecutore locale è ideale in scenari di testing in quanto è molto veloce e semplice da impostare, fornisce parallelismo e non è pesante. Tuttavia è altamente sconsigliato in ambienti di produzione in quanto è un componente *single point of failure* e riduce la scalabilità orizzontale propria di Airflow.

Sequential executor

Il sequential executor è l'esecutore di default di Airflow. Come il nome suggerisce, con questo componente i task sono eseguiti seguendo una logica sequenziale. Per questo motivo, anche se si progetta il DAG per fare eseguire più task in parallelo, con questo esecutore i task sono eseguiti uno dopo l'altro. Inoltre, come riporta la documentazione ufficiale di Airflow [9], il sequential executor è l'unico esecutore che permette di utilizzare un database di tipo sqlite, dal momento che sqlite non supporta connessioni multiple.
È chiaro che questo tipo di esecutore è utilizzato per apprendere e testare il funzionamento di Airflow.

Kubernetes Executor

Anche se in tale trattazione non verranno discussi i diversi servizi di orchestrazione dei container, per comprendere questo tipo di esecutore è necessario introdurre Kubernetes. In particolare, Kubernetes è una piattaforma open-source sviluppata da Google per la gestione di carichi di lavoro e servizi containerizzati in un ambiente clusterizzato [10]. Un concetto fondamentale di questa tecnologia è chiamato *pod*, il quale definisce un gruppo di uno o più container che condividono memoria, risorse di rete e le specifiche su come eseguire i container.
L'esecutore Kubernetes esegue come un processo all'interno dello scheduler di Airflow e consente di eseguire ogni task nel proprio pod su un cluster Kubernetes. Per interagire lo scheduler fa uso delle API proprie di Kubernetes, mentre quest'ultimo comunica con il database per memorizzare le informazioni riguardo lo stato di avanzamento del flusso di lavoro. Non è infatti necessario che il database e lo scheduler siano all'interno del cluster, ma è indispensabile che siano in grado di comunicare tramite chiamate API.
In Figura 1.5 è rappresentato il diagramma di sequenza relativo all'esecuzione di un DAG sfruttando un cluster e l'esecutore Kubernetes. Questo tipo di esecutore, per una serie di

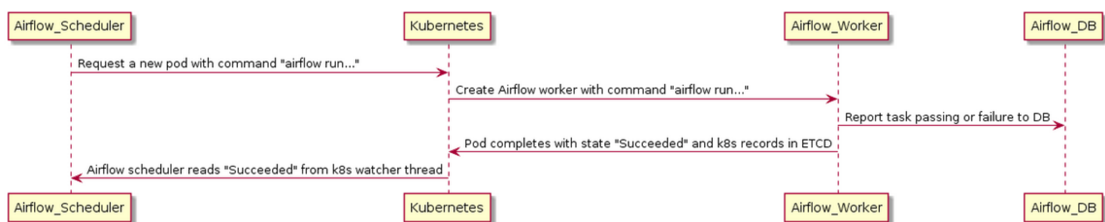


Figura 1.5: Diagramma di sequenza di esecuzione di un DAG con l'esecutore Kubernetes. Immagine tratta da [11].

cause, ha portato una considerevole crescita degli utilizzatori di Airflow. Una di queste cause è sicuramente la popolarità di Kubernetes, diventato oggigiorno una delle piattaforme di orchestrazione dei container più adottata. Oltre ai vantaggi di Kubernetes, in questo esecutore lo scheduler non richiederà costantemente lo stato di ogni task bensì saranno gli stessi pod che, una volta terminata l'esecuzione, comunicheranno direttamente con lo scheduler tramite le API. Uno svantaggio è rappresentato dai requisiti fondamentali per l'adozione di questo esecutore, dal momento che è necessaria una conoscenza approfondita di Kubernetes e la gestione del cluster associato, tuttavia offre elevata scalabilità e affidabilità ai guasti.

Celery executor

Celery è un sistema distribuito e open-source che implementa una coda asincrona di attività basata sullo scambio di messaggi distribuiti. Il suo focus principale è di processare i task in real-time, per di più supporta anche la schedulazione di questi ultimi. L'idea alla base è che tutte le risorse dell'applicazione possono inserire task all'interno della coda, ed in modo totalmente trasparente al programmatore tali task vengono inviati ai worker. Uno dei maggiori benefici che porta Celery è che forza all'utilizzo di un'architettura distribuita

rendendo l'applicazione più scalabile. Celery è interamente scritto in Python, anche se il protocollo può essere implementato in un qualsiasi linguaggio di programmazione.
Per istanziare un sistema Celery, è necessario configurare i seguenti componenti:

- **Celery workers**: unità computazionali che si occupano di elaborare i job della coda in background. Essi possono essere sia remoti che locali, ma per rendere l'applicazione scalabile spesso si preferisce la prima opzione.;
- **Message broker**: servizio che permette l'integrazione asincrona tramite scambio di messaggi. Questo tipo di interazione è fortemente disaccoppiata in quanto l'invio dei messaggi avviene su un canale dedicato solamente al broker, ed è responsabilità di quest'ultimo consegnarli ai diretti interessati. Oltre a questo compito, il broker si occupa anche di aspetti legati alla sicurezza, priorità dei messaggi e l'inoltro ordinato.

Celery si propone per essere:

- semplice: è relativamente semplice installare, utilizzare e mantenere questo servizio;
- flessibile: essendo open-source, è possibile estendere qualsiasi parte di Celery o addirittura utilizzare la propria implementazione del servizio;
- veloce: un singolo processo Celery può processare milioni di task al minuto, come riportato in [12];
- supporta nativamente diversi broker, come per esempio RabbitMQ e Redis.

Il Celery executor è l'esecutore più maturo presente all'interno di Airflow e sblocca la possibilità di scalare orizzontalmente permettendo di creare un cluster di worker Celery, in cui ogni macchina può contenere uno o più Airflow worker, mentre la comunicazione tra gli Airflow workers e lo scheduler avviene tramite l'interfaccia del broker Celery.
La Figura 1.6 descrive la sequenza di esecuzione di un task con l'esecutore Celery. In particolare, nel momento in cui lo scheduler trova un task che deve essere eseguito lo inoltra al broker, oltre ad interrogare periodicamente il database di Airflow per controllare lo stato dei task. Nel momento in cui il broker riceve un messaggio dallo scheduler allora lo inoltra al worker Celery opportuno, il quale elabora il task su un processo dedicato. Quando termina l'esecuzione, il worker salva questo evento nel database, in modo tale che lo scheduler se ne possa accorgere.
Celery supporta diverse implementazioni di broker. In questo elaborato sarà successivamente utilizzato **Redis**, anche detto *Remote Dictionary Server*: si tratta di un archivio di strutture dati in-memory, utilizzato come un database chiave-valore, una cache e anche come broker di messaggi. Infatti, grazie alla sua capacità di mantenere i dati in memoria centrale, Redis minimizza il tempo di risposta eliminando l'accesso ai dati su disco. Questa sua caratteristica lo rende perfetto per diversi casi d'uso, come il caching, analisi dati real time e gestore di messaggi.

1.2.5 Operatori di Airflow

Un altro concetto fondamentale di Airflow sono i suoi operatori. In particolare un operatore non è altro che un template di un task predefinito che può essere esplicitato in modo

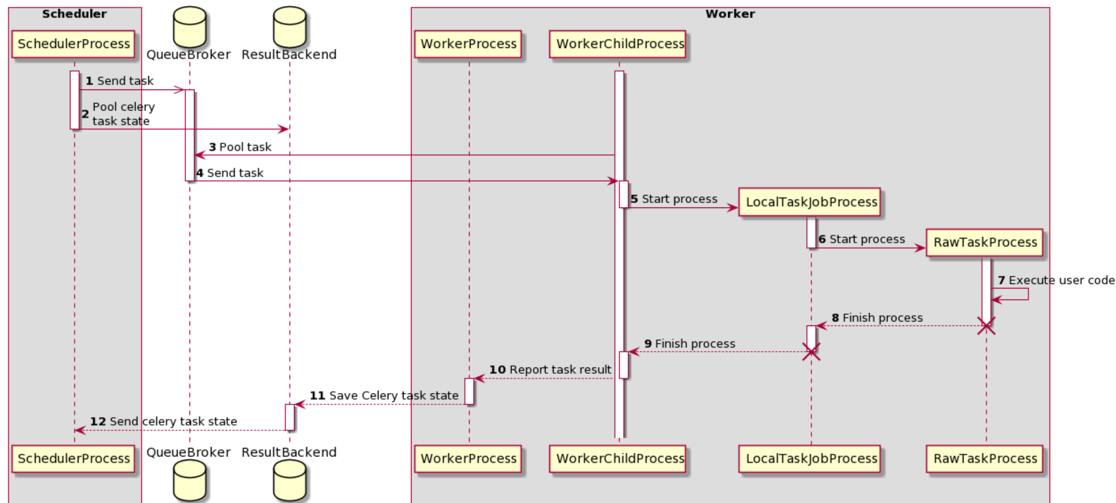


Figura 1.6: Diagramma di sequenza di esecuzione di un task con l'esecutore Celery. Immagine tratta da [13].

dichiarativo all'interno del codice Python del DAG. Essi contengono la logica di come processare i dati della pipeline e per questo motivo spesso si implementano operatori custom definendo una classe Python che implementa il metodo *execute*.
Ad alto livello, gli operatori possono essere suddivisi in tre classi distinte:

- **action operators**: operatori utili ad eseguire qualsiasi tipo di operazione. Alcuni esempi sono rappresentati dal *PythonOperator* (che esegue una funzione Python) e dal *BashOperator* (che esegue un comando bash);
- **transfer operators**: operatori utili al trasferimento dei dati da una sorgente ad una destinazione. Un esempio è *S3FileTransformOperator*, il quale copia i dati da una sorgente S3 (servizio di storage in cloud di AWS) ad una locazione temporanea nel filesystem locale;
- **sensors**: operatori che valutano, ad intervalli di tempo regolari, se una condizione è verificata o no. Quindi, quando l'evento accade, il loro stato diventa *succeed* ed è eseguito il task associato. Al contrario, se la condizione non è ancora verificata, il sensore aspetta un altro intervallo prima di rivalutarla. Poiché sono principalmente inattivi, i sensori hanno tre diverse modalità di funzionamento, che consentono di utilizzarli in modo più efficiente:
  - *poke*: il sensore mantiene le risorse computazionali del worker associato per l'intero tempo di esecuzione. Questa modalità è il default dei sensori di Airflow ed è indicata per casi d'uso in cui i sensori hanno intervalli di attesa piccoli, ovvero controllano il criterio molto frequentemente;
  - *reschedule*: se il criterio non è verificato allora il sensore rilascia il worker associato e sarà rischedulato al prossimo intervallo. Al contrario del poke, questo funzionamento è ottimo nel caso in cui il criterio è esaminato ad intervalli di tempo grandi;

- *smart sensor*: consolida più istanze di sensori in un unico processo con lo scopo di ridurre il costo dell'infrastruttura di Airflow. Infatti, invece di utilizzare un processo per ogni task, l'idea principale è quella di utilizzare un singolo processo centralizzato per eseguire tali task in batch. Tuttavia questo tipo di sensore è ancora in early-access e potrebbe cambiare in future versioni di Airflow.

Un esempio di sensore è il *SqlSensor*, il quale attende che i dati siano presenti all'interno di una tabella SQL. Questo sensore è utilizzato spesso se si vuole triggerare l'esecuzione del flusso solo dopo aver collezionato i dati all'interno del database.

Operatori AWS

Dal momento che questo elaborato tratta della progettazione e rilascio di un workflow con Airflow sull'infrastruttura cloud di AWS, è stato necessario l'adozione di operatori in grado di istanziare dei worker su un cluster in cloud. Per fare questo si è fatto uso degli operatori *ECSOperator* e *GlueOperator*, i quali rispettivamente permettono di eseguire un task su un cluster ECS di AWS e su un job Glue. Più nello specifico, ECS è un servizio di orchestrazione che consente di gestire e distribuire containers su un cluster (vedi sezione 2.7.9), mentre Glue è un servizio serverless di ETL (vedi sezione 2.7.16).

1.2.6 Control flow

Di default Airflow esegue un task quando tutte le sue dipendenze sono rispettate. Tuttavia, in fase di progettazione delle pipeline può capitare di avere necessità di creare flussi di lavoro più complessi di quelli visti fino ad adesso. Un esempio è il seguente: si immagini di decidere di eseguire un task a partire da un gruppo di essi sulla base dei risultati del task precedente. Fortunatamente Airflow fornisce building block per costruire condizioni logiche di esecuzione; questa sezione ha lo scopo di descrivere le principali funzionalità di controllo del flusso: il branching e le trigger rules.

Branching

Il *branching* è una funzionalità di Airflow che permette di non eseguire tutti i task che dipendono dal task corrente, bensì di selezionarne solo alcuni (come evidenziato in Figura 1.7). Per fare questo sono disponibili diversi operatori come per esempio *BranchPythonOperator*, *BranchSQLOperator* e *BranchDayOfWeekOperator*. L'operatore più flessibile di tutti è sicuramente il BranchPythonOperator, che come il PythonOperator prende una funzione Python in ingresso e, sulla base della logica di tale funzione, restituisce in output una lista di ID dei tasks che lo scheduler deve schedulare.

Trigger rules

Le regole di trigger definiscono come e perchè un task viene attivato, in quali condizioni. Anche se il comportamento standard di Airflow prevede di eseguire un task quando tutte le sue dipendenze sono rispettate, queste regole permettono di definire un sistema di dipendenze più complesso. Tutti gli operatori hanno il parametro *trigger_rule* che definisce quanto appena detto, in accordo con la tabella 1.2 che specifica le varie regole.

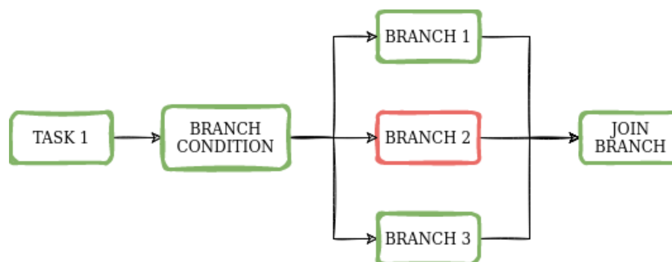


Figura 1.7: Esempio di branching in un DAG

Nome	Descrizione
all_success	Tutti i task a monte devono essere eseguiti correttamente
all_failed	Tutti i task a monte sono in uno stato failed o upstream_failed
all_done	Tutti i task a monte hanno terminato l'esecuzione
all_skipped	Tutti i task a monte sono stati saltati
one_failed	Almeno un task a monte è fallito
one_success	Almeno un task a monte ha avuto successo
none_failed	Tutti i task a monte non sono in uno stato failed o upstream_failed. Ciò significa che hanno avuto successo o sono stati saltati
none_failed_min_one_success	Tutti i task a monte non sono in uno stato failed o upstream_failed e almeno un task a monte ha avuto successo
none_skipped	Nessun task a monte è stato saltato
always	Nessuna dipendenza, esegui il task sempre

Tabella 1.2: Trigger rules di Airflow

1.2.7 Dynamic Task Mapping

Con il termine *dynamic task mapping* si intende la capacità di Airflow di generare dinamicamente task paralleli a runtime. Questa caratteristica permette al workflow di creare un numero di task in base all'ambiente senza che l'utente ne definisca un numero in fase di progettazione. Il dynamic task mapping è stato introdotto con il rilascio di Airflow 2.3 mentre prima di questa versione per ottenere un risultato simile era necessario definire i task all'interno di un ciclo for.
Possiamo dire quindi che il dynamic task mapping segue il paradigma di programmazione *MapReduce*, il quale si riferisce a due compiti separati e distinti: mappatura di un insieme di dati e raggruppamento degli stessi in coppie chiave-valore. Calato nel contesto di gestione di workflow, la procedura di mapping permette di creare un task per ogni insieme di input, mentre la procedura di reduce consente ad un task specifico di operare su tutti gli output dei task precedenti.

1.2.8 Airflow Webserver

L'ultimo componente di Airflow che rimane da commentare è il suo webserver. Grazie alla sua interfaccia utente è possibile monitorare lo stato delle pipeline, innescare manualmente i workflow, visualizzare i log, il tempo di esecuzione dei task e molto altro ancora. Di default è implementato in flask, ma tramite le REST API proprie di Airflow è possibile integrarlo all'interno della propria dashboard.
In Figura 1.8 riporta la pagina che mostra la lista dei DAG dal webserver, mentre in Figura 1.9 è presente un esempio di workflow visto dall'interfaccia utente.

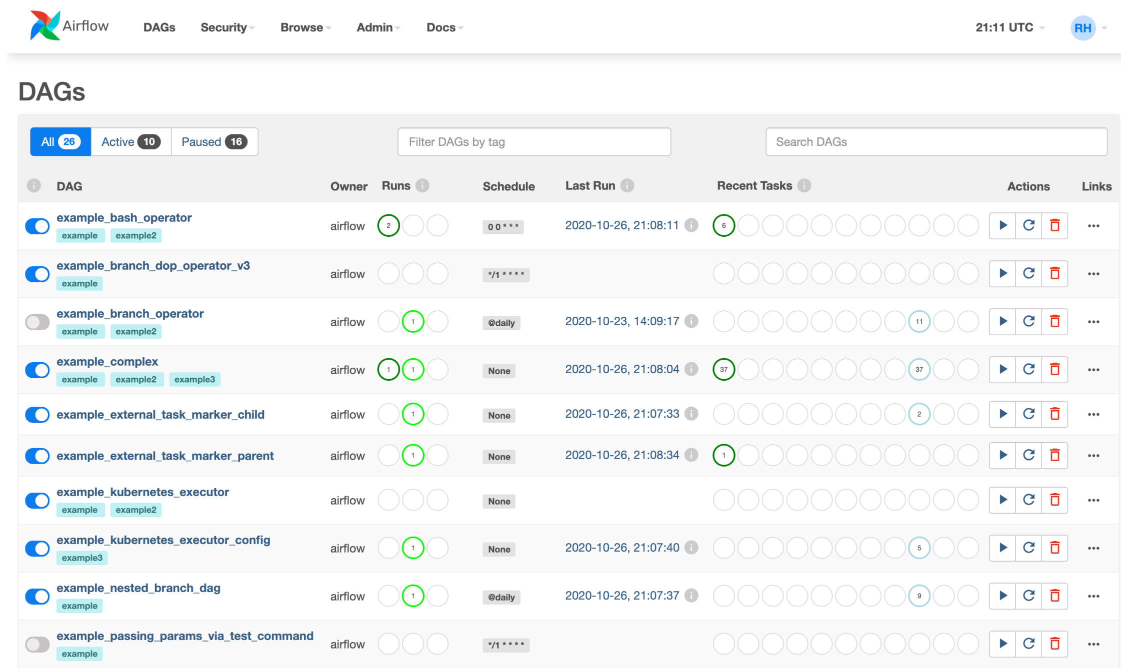


Figura 1.8: Screenshot che riprende una lista di DAGs dal webserver. Immagine tratta da [14]

1.2.9 Quando usare Airflow

Come tutti gli strumenti, anche Airflow non è adatto a ogni contesto applicativo. I vantaggi che questa piattaforma offre possono essere riassunti in questi punti:

- implementare pipeline tramite codice Python permette di creare workflow arbitrariamente complessi usando ogni costrutto di questo linguaggio;

- semplice da utilizzare: nella sua installazione di default sono compresi tutti i componenti di cui ha bisogno ed è possibile cambiarli agendo direttamente sul file di configurazione. Anche l'adozione di un linguaggio ad alto livello come Python permette di interfacciarsi con Airflow tramite un livello di astrazione maggiore;

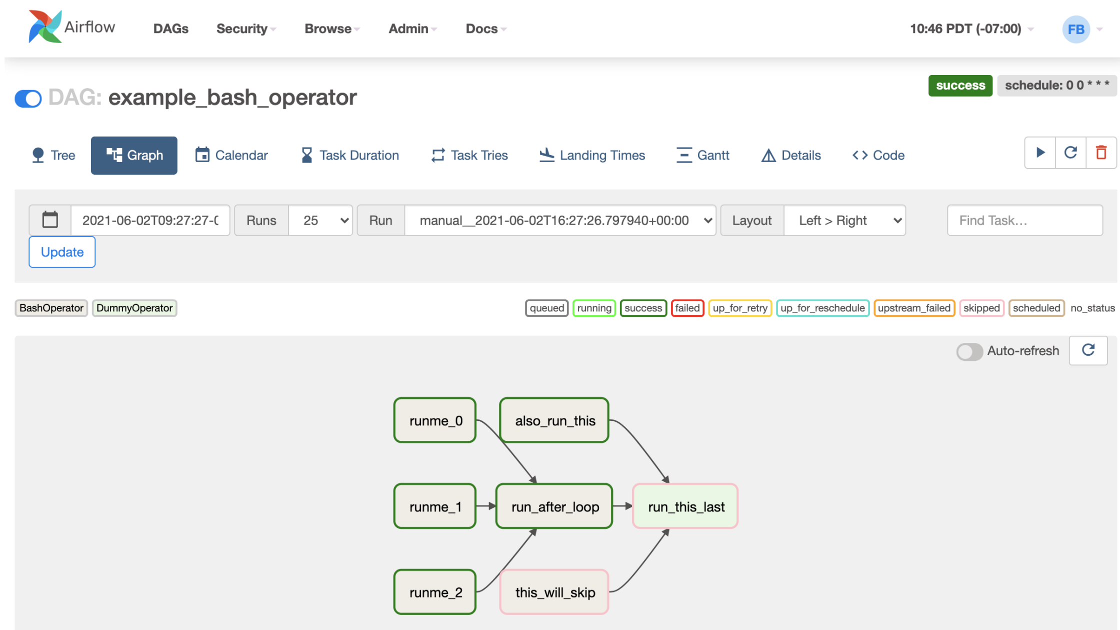


Figura 1.9: Screenshot che riprende un workflow dalla UI. Immagine tratta da [14]

- ha una community estesa che sviluppa costantemente nuove estensioni per integrare Airflow con altri tipi di sistemi cloud, altri database ecc;
- elevata scalabilità e flessibilità;
- possiede una dashboard di default che permette di monitorare i risultati delle pipeline;
- altamente estendibile. È possibile connettere Airflow a qualsiasi altro tipo di sistema per creare flussi di lavoro più complessi. Questo lo rende uno strumento ottimo per molte aziende che hanno la necessità di estendere le funzionalità dei loro sistemi attuali;
- supporta CICD (*Continuous integration Continuous delivery* grazie all'utilizzo di Python.

Apache Airflow è adatto a contesti in cui è necessario gestire un numero di workflow elevato. Infatti questo strumento fornisce uno scheduling, un database, un webserver e degli esecutori e quindi richiede un minimo di risorse computazionali: se si hanno pochi task e altamente sporadici Airflow rappresenta un overkill, mentre se si deve gestire un numero elevato di workflow questa piattaforma è in grado di farlo in maniera ordinata e pulita.
Non è da considerare un'opzione valida nel caso in cui è richiesta la gestione di *streaming pipelines*: Airflow nasce per eseguire task in modalità *batch* piuttosto che per carichi di lavoro in streaming. Inoltre i DAG possono diventare molto complessi e difficili da mantenere: è quindi richiesto un monitoraggio costante e una buona esperienza in merito.

Capitolo 2

Cloud Computing

2.1 Le fondamenta del cloud computing

Come ogni altra tecnologia di successo, anche il cloud computing è nato per risolvere un problema comune a livello globale e da qualche anno a questa parte è diventato un elemento cruciale nelle strategie di innovazione di molte imprese. È stato ideato dai service provider con lo scopo di supportare un numero sempre maggiore di utenti tramite un servizio elastico con il minimo delle risorse.
Facendo uso della definizione fornita dal NIST (*National Institute of Standards and Technology*) nel paper "*The NIST Definition of Cloud Computing*" [15], il cloud computing è un modello che consente di accedere in rete in modo ubiquo, conveniente e su richiesta a un pool condiviso di risorse informatiche configurabili (ad esempio, reti, server, storage, applicazioni e servizi) che possono essere rapidamente messe a disposizione e rilasciate con uno sforzo minimo di gestione o di interazione con il service provider.
Anche se questa definizione risale al 2011, l'idea alla base del cloud computing era già diffusa ben prima degli anni 2000. In particolare nel 1961, alla celebrazione del centenario dell'MIT, il Professore John McCarthy affermò come l'informatica un giorno sarebbe potuta essere organizzata come un servizio pubblico, proprio come il sistema telefonico o l'energia elettrica, ed il suo costo sarebbe direttamente proporzionale all'utilizzo [16]. Queste parole hanno anticipato quello che sarebbe poi successo nel 2006, quando Amazon ha rilasciato al pubblico la prima versione di AWS, anche detto *Amazon Web Services*.
L'introduzione del cloud computing ha permesso alle aziende di condividere il tempo di esecuzione su un'infrastruttura informatica comune, costituita da dispositivi intercambiabili che forniscono calcolo, archiviazione dati e comunicazione. Si possono riassumere le sue caratteristiche principali nei seguenti punti:

- **on-demand self-service**: è il consumatore che, al momento del bisogno, può ottenere il sevizio senza dover contattare esplicitamente il fornitore;

- **rapid elasticity and scalability**: i consumatori possono facilmente aumentare o diminuire le risorse richieste al bisogno. Questo consente di mitigare i rischi legati a un dimensionamento non corretto delle infrastrutture necessarie evitando che picchi di carico blocchino il sistema. Per il consumatore del servizio queste capacità di calcolo

spesso appaiono illimitate e possono essere sia acquisite tramite una richiesta esplicita o in automatico;

- **resource pooling**: il provider mette a disposizione un insieme di risorse che possono essere fornite a più di un cliente alla volta tramite un'architettura multi-tenant e alla virtualizzazione. Inoltre questi servizi possono essere adattati alle esigenze di ciascun cliente senza che ne siano evidenti i cambiamenti, la complessità dell'architettura è trasparente al consumatore finale;

- **broad network access**: i servizi offerti dai provider sono disponibili in rete e accessibili mediante meccanismi standard che ne favoriscono l'uso da parte di piattaforme client eterogenee;

- **secure**: malfunzionamenti o smarrimenti di dispositivi vari non precludono l'accesso ai dati, che rimangono sempre e comunque raggiungibili;

- **economical**: il cloud computing segue un modello di pagamento *pay-per-use* che, unito alle sue caratteristiche di scalabilità ed elasticità, riduce i costi rispetto ad un'architettura on-premise. Inoltre non è richiesto nessun iniziale investimento sull'hardware;

- **guaranteed Quality of Service**: gli ambienti di elaborazione offerti dai cloud providers devono garantire la qualità dei servizi, come ad esempio la velocità di CPU, la larghezza di banda e la dimensione della memoria. La QoS viene definita tramite il *Service Level Agrement* (SLA), un accordo tra il cloud provider e il cliente che garantisce il mantenimento di un livello minimo di servizio.

Tutto questo è racchiuso nel cloud computing, in cui il termine cloud è utilizzato per rappresentare l'hardware e il software presente nei data center del provider. Ovviamente la metafora della nuvola non è casuale: essa infatti è per sua natura lontana, opaca e cambia forma dinamicamente.

2.2 Cloud actor

In questa sezione si vogliono descrivere tutti gli attori presenti nel cloud computing e le loro interazioni:

- **cloud consumer**: rappresenta una persona o un'organizzazione che mantiene relazioni di business e usa i servizi messi a disposizione dai cloud providers. Necessitano di definire una SLA in accordo con il provider per specificare i requisiti minimi dei servizi che il fornitore è tenuto a disporre;

- **cloud provider**: entità responsabile di fornire un servizio a terze parti. Si incarica di gestire e acquistare l'infrastruttura richiesta, eseguire il software e fornire il servizio al consumer tramite la rete;

- **cloud auditor**: soggetto di terze parti in grado di eseguire un esame indipendente sui servizi offerti dai providers con l'intento di esprimere un parere in merito. Questi controlli vengono eseguiti per verificare la conformità agli standard riguardo la sicurezza, la privacy dei dati, le prestazioni, i costi e cosi via;

- **cloud broker**: entità che gestisce la negoziazione tra provider e consumer, oltre che l'uso e la consegna dei servizi ai consumatori. Alle volte il broker può essere anche un software, che indica al customer il provider che più gli conviene sulla base delle esigenze;
- **cloud carrier**: intermediario che fornisce la connettività ed il trasporto di servizi cloud tra il cloud consumer e il cloud provider.

2.3 Service model

Dal momento che il cloud computing è diventata una tecnologia molto utilizzata dalle aziende, con il tempo sono nati numerosi modelli in grado di soddisfare le specifiche esigenze di utenti diversi. Infatti non tutti i servizi cloud forniscono lo stesso livello di astrazione bensì offrono diversi livelli di controllo, flessibilità e gestione. Tuttavia è bene precisare che non sempre esiste una vera e propria distinzione netta, può capire che un servizio appartenga ad una combinazione di due modelli. In particolare, gli utenti possono interfacciarsi principalmente con tre diversi service model:

- **Infrastructure as a Service (IaaS)**: questo tipo di servizi fornisce l'infrastruttura su cui eseguire un'applicazione. Il consumatore ha a disposizione risorse di computazione, di archiviazione e di rete con cui distribuire ed eseguire software arbitrario, che può includere sia interi sistemi operativi che applicazioni. Questo modello dipende fortemente dalla capacità di virtualizzazione del provider ossia la sua abilità di emulare componenti hardware, mappate sulle componenti fisiche, al fine di renderle disponibili al software in forma di risorsa virtuale. Il vantaggio di questa soluzione è che l'utente è sollevato dalla complessa gestione dell'infrastruttura e può aumentare e diminuire le risorse di calcolo su richiesta in modo dinamico. Questo garantisce una scalabilità del servizio su richiesta. Sarà il provider a concentrarsi sulle performance e sulla virtualizzazione delle risorse mentre il focus dell'utente è sulla logica applicativa. Alcuni esempi di Iaas sono Amazon EC2 e Microsoft Azure Platform.
- **Platform as a Service (PaaS)**: è simile a IaaS, ma oltre alle risorse hardware include anche il sistema operativo e i servizi richiesti per eseguire una particolare applicazione. Questo modello è rivolto agli sviluppatori e offre un ciclo di vita del software completo dal momento che permette a questi ultimi di implementare applicazioni direttamente sul cloud. Il provider fornisce lo storage, la rete e unità di elaborazione per eseguire un'applicazione, mentre l'utente supervisiona la distribuzione del software e le impostazioni di configurazione dell'ambiente che ospita l'applicazione. Con questo tipo di modello le aziende sono facilitate nello sviluppare applicazioni e spesso vengono ridotti i tempi di sviluppo grazie alla presenza di componenti pre-codificate all'interno della piattaforma. Tuttavia rispetto a IaaS l'utente non ha controllo sul sistema operativo sottostante, e per tale ragione non è possibile gestire aspetti legati all'ambiente su cui esegue l'applicazione. Un esempio PaaS è Google App Engine, il quale offre la possibilità alle applicazione di eseguire sull'infrastruttura di Google.
- **Software as a Service (SaaS)**: rappresenta la forma di servizi di cloud computing più completa. Infatti viene fornita all'utente un insieme di applicazioni in esecuzione

su una piattaforma e un'infrastruttura di proprietà del service provider. Queste applicazioni sono progettate per essere accedute simultaneamente da diversi client tramite Internet. Il consumatore non gestisce o controlla né l'infrastruttura sottostante né le funzionalità delle singole applicazioni. I vantaggi di SaaS sono svariati: l'utente non si deve preoccupare di aggiornare le applicazioni, correggere bug o installare il software sulle proprie macchine. Questo modello è ottimo per le piccole aziende che non hanno il personale sufficiente per gestire installazioni e aggiornamenti software o per applicazioni utilizzate solo sporadicamente. Un esempio si SaaS è la suite Microsoft Office 365.

2.4 Deployment model

Non tutti i sistemi cloud seguono lo stesso modello di distribuzione. Ogni modello ha la propria capacità di personalizzazione, requisiti di sicurezza, condivisione dei servizi cloud e l'ubicazione dei servizi ospitati. In particolare si possono identificare quattro diversi modelli di distribuzione sulla base del possessore del data center:

- **public cloud**: è costituito da servizi gestiti da terze parti che vengono esposti ai consumatori tramite connessioni Internet. In questo tipo di distribuzione il provider pubblico ha pieno controllo dell'architettura e gli utenti finali possono accedervi ed usufruirne con un modello *pay-as-you-go*. È il modello cloud più diffuso in quanto permette di scalare molto rapidamente e a costi che dipendono dall'utilizzo del servizio, non richiede manutenzione, è affidabile e forniscono strategia di data recovery. Tuttavia presenta anche diverse criticità legate alla privacy, alla sicurezza e all'assenza di controllo sull'infrastruttura sottostante;

- **private cloud**: l'infrastruttura è dedicata ad una sola azienda, anche se le organizzazioni di terze parti hanno accesso alla gestione del cloud per conto del proprietario. Rispetto al cloud pubblico, il cloud privato offre maggiore sicurezza dal momento che è accessibile solamente a utenti certificati, e per questo è utilizzato per memorizzare e gestire dati sensibili. Come contro si ha un costo tipicamente maggiore legato sia all'utilizzo energetico sia al personale incaricato di gestire l'architettura;

- **community cloud**: è simile al cloud privato ma le risorse sono condivise da una comunità, ovvero un'insieme di più organizzazioni che hanno caratteristiche comuni come ad esempio stessi requisiti e vincoli di sicurezza. Il vantaggio di questa soluzione rispetto al cloud privato deriva dalla condivisione dei costi tra tutti i membri della comunità. Un esempio di questo modello di distribuzione è l'infrastruttura cloud delle Università che viene condivisa agli studenti con lo scopo comune di apprendimento;

- **hybrid cloud**: deriva dalla composizione di due o più architetture cloud, che possono essere sia private, community o public, le quali rimangono entità uniche ma sono legate da una tecnologia standardizzata o proprietaria che consente la portabilità dei dati e delle applicazioni. Il cloud ibrido ha il vantaggio di unire tutti i benefici dei diversi modelli di distribuzione, per esempio è facile gestire i problemi di sicurezza dei dati salvando i dati sensibili in uno cloud privato mentre quelli non sensibili in un cloud pubblico.

In Figura 2.1 sono raffigurati i quattro diversi modelli di distribuzione del cloud computing.
In aggiunta a questi modelli di distribuzione negli ultimi anni è stato introdotto il con-

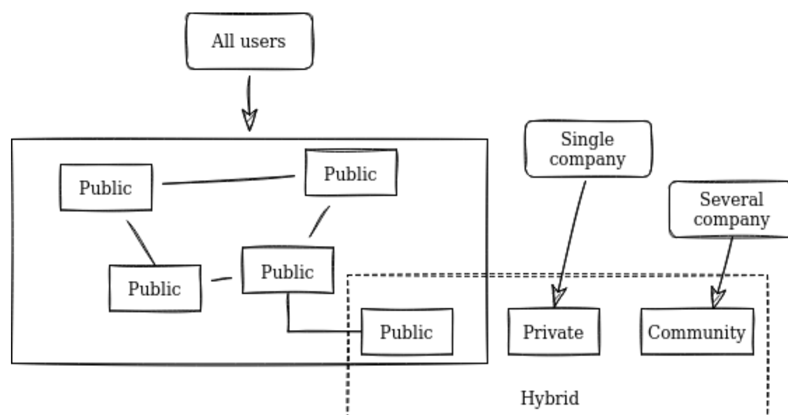


Figura 2.1: Modelli di distribuzione del cloud computing

cetto di *virtual private cloud* (VPC), il quale permette alle organizzazioni di ottenere una sezione isolata dell'infrastruttura del cloud pubblico creando una rete virtuale sulla quale avere controllo completo. Una VPC ha caratteristiche proprie sia del cloud pubblico sia di quello privato. Infatti utilizza le risorse di calcolo messe a disposizione da un provider condividendole con tutti gli altri consumatori anche se il collegamento a tali servizi avviene tramite una rete privata virtuale alla quale il provider dedica risorse private.
La Figura 2.2 rappresenta una panoramica del Cloud computing, in cui sono raffigurati sia gli autori sia la sua architettura.

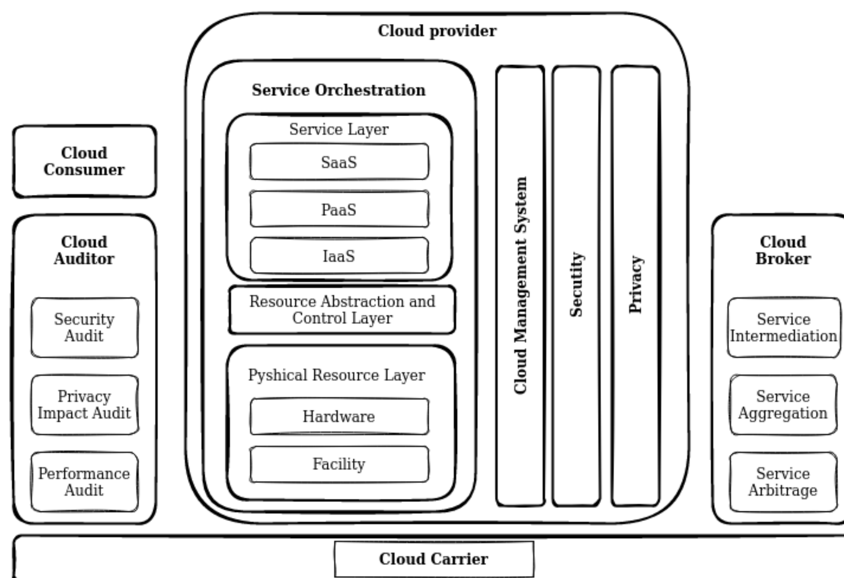


Figura 2.2: Panoramica del cloud computing. Immagine ispirata alla Figura 1 di [17]

2.5 Cloud provider

Negli ultimi anni le aziende hanno aumentato la richiesta di servizi cloud con l'obiettivo di ridimensionare la propria struttura ed essere più scalabili. Grazie ai benefici di questa tecnologia e l'adesione a questi servizi in costante crescita, il numero di cloud provider è aumentato notevolmente, portando così concorrenza nel mercato. Le compagnie come Amazon, Google e Microsoft hanno ridotto sempre più i costi dei servizi offerti rendendo sempre più accessibile e appetibile l'utilizzo di architettura cloud diventando in assoluto i fornitori cloud più utilizzati. Questo è confermato anche dagli analisti di Gartner, che nel Settembre 2021 hanno pubblicato il report "*Magic Quadrant for Cloud Infrastructure and Platform Services* [18]" nel quale hanno comparato diversi cloud provider generando il grafico mostrato in Figura 2.3.
Amazon Web Services (AWS) è il provider di servizi più consolidato, infatti è nato

Figura 2.3: Magic quadrant dei cloud provider nel 2021. Immagine tratta da [18]

nel 2006 e da allora è sempre stato uno dei leader del mercato. Il suo focus principale è cercare di possedere porzioni sempre più ampie della *supply chain* [18], dove per supply chain si intende il processo che permette di portare sul mercato un prodotto o servizio, trasferendolo dal fornitore fino al cliente.
Il secondo leader a parere di Gartner è **Microsoft Azure**, annunciato nel 2008 in occasione della *Professinal Developers Conference* [19]. Si distingue per l'attenzione alla sicurezza dei dati pur fornendo un elevato numero di servizi ma richiede un cliente esperto in ambito commerciale, in quanto dispone di licenze e contratti molto complessi. Si sta avvicinando ad AWS in termini di offerta soprattutto grazie all'ecosistema Office365. Inoltre ha il vantaggio che Microsoft è già fortemente integrato all'interno dell'infrastruttura di molte aziende, aumentando così il bacino di utenti a cui si rivolge.
L'ultimo colosso è Google che offre **Google Cloud Platform**. Nel Magic Quadrant di Garter del 2014 si trovava nel quadrante degli innovatori ma, grazie alla sua velocità di

innovazione e ai continui investimenti sull'infrastruttura, è diventato il terzo leader nel mondo del cloud computing. Sebbene disponga di alcune tecnologie di punta è ancora in fase rudimentale per i servizi IaaS, e per questo fonda il suo business su servizi PaaS e SaaS.
Una menzione d'onore riguarda **Alibaba Cloud** che viene considerato un visionario da Gartner. Supporta i clienti con sede nel Sud-est Asiatico e si concentra sull'aumento delle offerte di database PaaS.
La Figura 2.4 mostra i Google Trends dei maggiori cloud provider, e dimostra come AWS e Microsoft Azure siano sicuramente i fornitori di servizi più popolari e ricercati negli ultimi dieci anni.

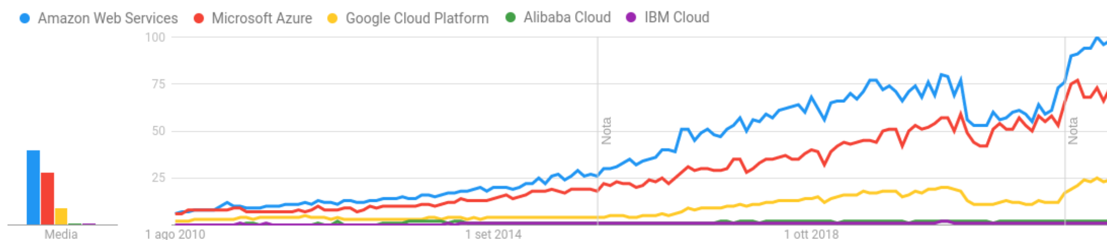


Figura 2.4: Google trends dei principali cloud provider

2.6 Tecnologie abilitanti del cloud computing

Le tecnologie a supporto del cloud computing sono molte , ma le più importanti sono la virtualizzazione, il Web 2.0, i Web service e SOA.

2.6.1 Virtualizzazione

La virtualizzazione è una tecnologia che consente di creare più ambienti simulati o risorse dedicate da un unico pool di risorse di un sistema fisico. Questo modello ha rappresentato un'importante svolta nel mondo dell'informatica; grazie alla virtualizzazione il costo per la capacità di calcolo è diminuito considerevolmente. Anche se la tecnologia e i casi d'uso si sono evoluti nel tempo, l'idea alla base della virtualizzazione resta la stessa: permettere ad un sistema di elaborazione di eseguire più applicazioni indipendenti ed eterogenee allo stesso tempo.
I principali componenti della virtualizzazione sono mostrati in Figura 2.5 e sono:

- **host**: server fisico che ospiterà le macchine virtuali. Questa macchina esegue il software di virtualizzazione che permette a più applicazioni eterogenee di eseguire sullo stesso sistema fisico. Tipicamente le caratteristiche dell'host sono trasparenti alle macchine virtuali che eseguiranno su di esso;

- **guest**: software che viene eseguito sull'host all'interno di un ambiente virtuale. Su un host può essere in esecuzione anche più di un solo guest, che tipicamente vengono chiamati anche macchine virtuali;

- **hypervisor**: software di gestione delle machine virtuali in esecuzione su un host. Questo componente è il cuore della virtualizzazione, dal momento che isola il sistema operativo e le risorse dell'host dalle macchine virtuali. Permette di eseguire più sistemi operativi allo stesso tempo sullo stesso hardware e si possono categorizzare in due tipologie:
  - hypervisor di tipo 1: anche detto *hypervisor bare metal*, gira direttamente sull'hardware dell'host per gestire i sistemi operativi guest. Opera da un livello più alto di privilegi rispetto ad un normale sistema operativo, infatti si interfaccia direttamente con l'hardware e la pianificazione delle risorse dei guest avviene direttamente sullo stesso hardware. Sono gli hypervisor più comuni e prestazionali; un esempio molto utilizzato è Microsoft Hyper-V;
  - hypervisor di tipo 2: componente software in esecuzione sul sistema operativo dell'host come una normale applicazione. Ha il compito di astrarre i sistemi operativi guest dal sistema operativo host. Questo tipo di hypervisor è meno performante rispetto a quello di tipo 1 in quanto è gestito dal SO dell'host, ma è ideale per utenti che desiderano eseguire più sistemi operativi sullo stesso computer. Un esempio di hypervisor di tipo 2 molto comune è VirtualBox o anche QEMU.

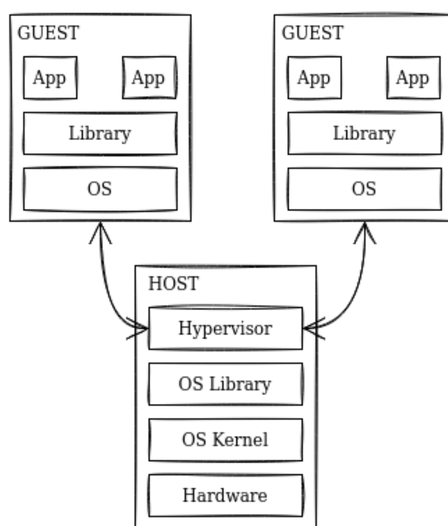


Figura 2.5: Componenti principali della virtualizzazione

Le tecniche di virtualizzazione permettono di creare sia delle macchine virtuali sia delle reti virtuali, come per esempio le VPN.

2.6.2 Web 2.0

Il termine Web 2.0 è stato coniato nel 2015 e ancora ad oggi è un concetto controverso. Efthymios Constantinides e Stefan Fountain nel paper "*Web 2.0: Conceptual foundations and marketing issues*" lo definiscono come un insieme di applicazioni online open-source,

interattive e controllate dagli utenti che ampliano le esperienze, le conoscenze e il potere di mercato degli utenti in veste di partecipanti ai processi aziendali e sociali. Le applicazioni del Web 2.0 supportano la creazione di reti informali di utenti che facilitano il flusso di idee e conoscenze consentendo la generazione, la diffusione, la condivisione e la modifica/-raffinazione efficiente di contenuti informativi [20].
L'idea alla base del Web 2.0 è di migliorare l'interconnettività e l'interattività delle applicazioni Web. I servizi di cloud computing sono applicazioni Web che forniscono servizi su richiesta: è quindi una naturale evoluzione per il cloud computing adottare le tecniche del Web 2.0.

2.6.3 Service Oriented Architecture

Con Service Oriented Architecture, anche detto SOA, si intende un approccio architetturale basato sul riutilizzo di *software building blocks* attraverso interfacce di servizio che usano linguaggi di comunicazione comuni in una rete. Questa metodologia ha come obiettivo quello di evitare l'implementazione di software monolitico e di riutilizzare codice già testato per costruire nuovi servizi. Seguendo questo approccio si aumenta anche la virtualizzazione, dal momento che non si considerano più le caratteristiche dell'hardware ma ci si concentra sul software.
Bisogna tuttavia differenziare SOA dai Web services: il primo è un approccio mentre i secondi sono una tecnologia che ha permesso di integrare applicazioni distribuite, e quindi di utilizzare SOA.

2.7 Amazon Web Services

In questa sezione si vogliono descrivere i servizi di AWS che verranno poi utilizzati nella seconda parte di questa trattazione.
Amazon Web Services è una piattaforma di cloud computing proprietaria di Amazon altamente affidabile, scalabile e a basso costo che serve centinaia di migliaia di aziende in 190 paesi di tutto il mondo. La sua nascita è la conseguenza dell'intuizione di sfruttare le risorse inutilizzare generare profitto, offrendole come servizi al pubblico. Nel 2006 AWS ha iniziato ad offrire servizi di infrastruttura IT alle aziende sotto forma di servizi web. Ad oggi Amazon Web Services mette a disposizione un numero molto elevato di servizi che spaziano dalle infrastrutture per il calcolo, l'archiviazione e i database fino alle nuove tecnologie, quali il machine learning, l'intelligenza artificiale, i data lake, analytics e Internet of Things [21]. Questo rende molto appetibile il suo utilizzo, dal momento che risulta abbastanza semplice, veloce ed efficiente migrare le applicazioni in esecuzione su sistemi on-premises verso il cloud di AWS.
Questa piattaforma può essere considerata un pioniere del paradigma *pay-as-you-go* e fornisce tutti i vantaggi che presenta il cloud computing (vedi sezione 2.1). In particolare i benefici economici che introduce sono svariati, quali per esempio la mancanza di investimenti iniziali, l'utilizzo efficiente delle risorse, il pagamento in base all'uso e la riduzione del tempo necessario a presentare il prodotto sul mercato. Oltre a sostenere il bilancio delle aziende si possono identificare anche dei benefici tecnici, quali la scalabilità automatica e proattiva, capacità di recupero dai guasti, elevata flessibilità e automatizzazione grazie

all'infrastruttura programmabile.
Una caratteristica che non tutti gli altri cloud provider possiedono è l'elevata globalizzazione dell'infrastruttura di AWS; questo permette di servire clienti in 190 paesi del mondo, di ottenere una latenza bassa con un throughput elevato e di garantire che i dati risiedano solo nella regione specificata dall'utente.

2.7.1 Infrastruttura globale di AWS

Anni dopo il rilascio del primo servizio, ad oggi le risorse di AWS sono ospitate in più zone in tutto il mondo. Infatti i servizi offerti da AWS eseguono su data-center ad alta efficienza e altamente disponibili ma, in rare occasioni, si possono verificare dei fallimenti. In questo contesto la sua infrastruttura globale di permette di risolvere il problema: infatti, se si ospitassero tutte le istanze di un servizio in un'unica zona interessata dal guasto, nessuna delle istanze sarebbe disponibile. Al contrario distribuire l'istanza di un servizio in più zone permette di aumentare notevolmente la sua disponibilità e affidabilità.
In particolare, queste zone sono composte da:

- **availability Zone** (AZ): una zona distinta all'interno di una regione AWS isolata dai guasti delle altre AZ che dispone di uno o più data centers con capacità di calcolo, rete e connettività con le altre AZ della stessa regione. Esse offrono la possibilità ai clienti di gestire le applicazioni con una disponibilità, una tolleranza agli errori e una scalabilità maggiori rispetto a quelle ottenibili con un singolo data center. L'idea è infatti di partizionare un'applicazione tra AZ differenti in modo da aumentare la protezione da eventuali fallimenti dovuti per esempio a interruzioni di corrente o altro. Tutte le AZ di una regione AWS sono interconnessi con una rete ad alta larghezza di banda e comunicano tramite una connessione criptata. Per mantenere bassa latenza di comunicazione tra le availability zones la distanza massima che li separa è di 100km;

- **region**: locazione fisica in cui vengono raggruppati data centers, in cui ogni regione è isolata e indipendente dalle altre ed è composta da due o più availability zone. Questo design permette di raggiungere il massimo livello di tolleranza agli errori e stabilità. Ad oggi le regioni AWS disponibili sono 26 e sono sparse in tutto il mondo in modo da garantire la massima copertura possibile. Come logico pensare, all'utente è consigliato di scegliere la più vicina.
  Le risorse che un utente istanzia in una regione non vengono create in automatico nelle altre, a meno che non si utilizzi esplicitamente una funzione di replica offerta dalla risorsa stessa. È importante specificare che, ad oggi, non tutte le regioni supportano tutti i servizi di AWS. Per esempio, il servizio Amplify è disponibile in Virginia Settentrionale ma non lo è per la regione di Città del Capo in Africa;

- **local zone**: viene definita come un'estensione di una regione AWS che è geograficamente vicina all'utente. L'idea è infatti di avvicinare i servizi di calcolo, storage e molti altri ai clienti finali in modo da eseguire facilmente applicazioni che richiedono latenze nell'ordine dei millisecondi con la tecnologia cloud. L'utente può estendere qualsiasi VPC di una regione AWS creando una nuova subnet e assegnandola ad una zona locale. Quando si crea una subnet in una zona locale, la VPC viene estesa a questa local zone.

In Figura 2.6 è riassunto quanto detto fino ad adesso riguardo l'infrastruttura globale di AWS.

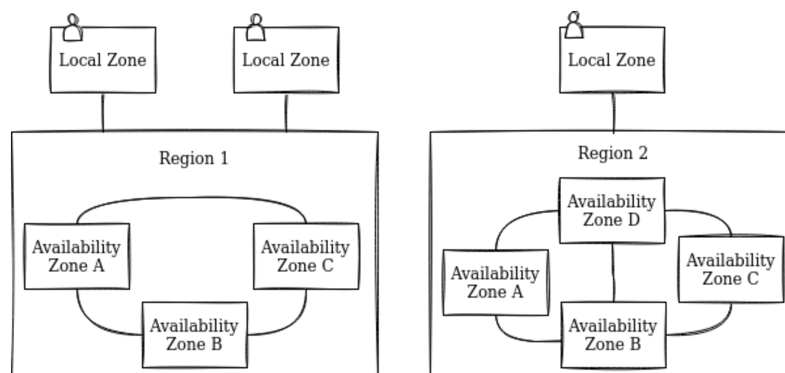


Figura 2.6: Infrastruttura globale di AWS

2.7.2 Introduzione ai servizi di AWS

Come detto precedentemente, AWS è uno tra i cloud provider che in assoluto ha il maggior numero di servizi offerti. In Tabella 2.1 è evidenziato il numero di servizi offerti suddivisi per categoria (aggiornato al Settembre 2022), con un totale di 219 prodotti.
Inoltre questi servizi sono fortemente interconnessi e per questo motivo possono essere facilmente utilizzati in combinazione, lasciando all'utente la possibilità di creare applicazioni di qualsiasi tipo e di qualsiasi complessità.
L'ambiente AWS di esecuzione delle pipeline ETL prese in carico nella seconda parte di questo elaborato è un esempio di combinazione di servizi di natura diversa, come mostrato in Figura 2.7. Infatti, a partire da diverse fonti dati e da pipeline ETL opportunamente definite, l'obiettivo del progetto è quello di restituire delle predizioni riguardo il livello di abbandono di un utente sfruttando la potenza di calcolo e la sicurezza offerta da AWS.
Nelle prossime sottosezioni saranno descritte le caratteristiche principali di ogni servizio utilizzato.

2.7.3 Amazon Virtual Private Cloud (Amazon VPC)

Prima di descrivere questo servizio è opportuno definire il concetto di VPC. In particolare, una virtual private cloud è un cloud privato sicuro e isolato ospitato all'interno di un cloud pubblico. Questo permette di combinare la scalabilità e la convenienza del cloud computing pubblico con l'isolamento dei dati offerto tipicamente dalle infrastrutture basate sul cloud privato.
Come si può notare in Figura 2.7, nell'ambiente di esecuzione delle pipeline ETL si è fatto uso di Amazon VPC, un servizio che permette di avviare risorse AWS all'interno di una rete virtuale isolata logicamente definita dall'utente. Il cliente può gestire la quasi totalità delle caratteristiche della rete, come per esempio configurare l'intervallo dei propri indirizzi IP, creare sottoreti, gestire gateway di rete, selezionare le ACL e impostare opportunamente le tabelle di routing. In particolare, componenti di questo servizio sono:

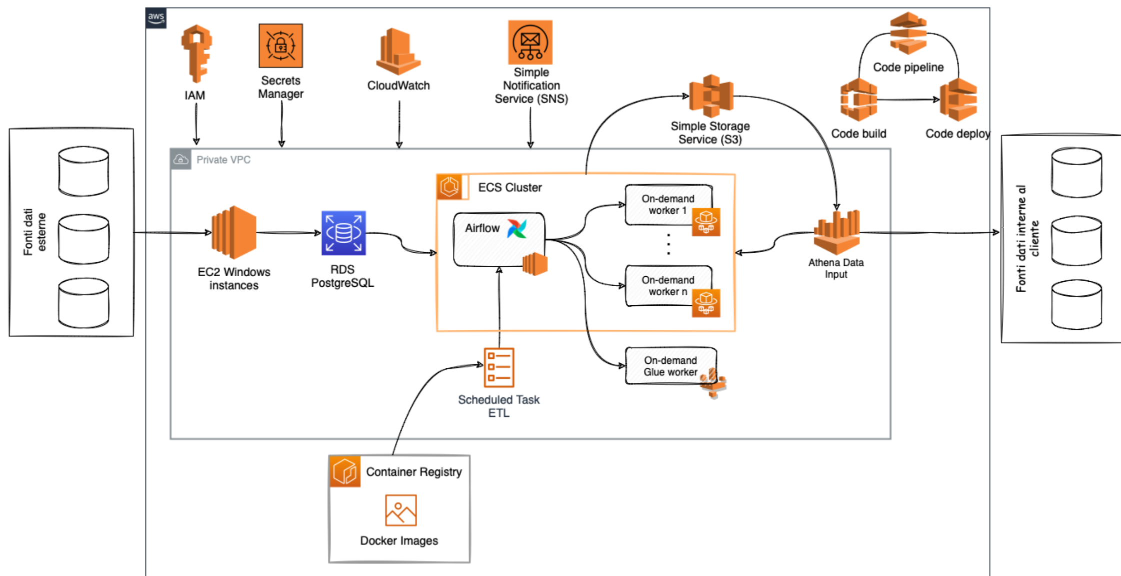


Figura 2.7: Servizi AWS utilizzati dalle pipeline ETL del progetto

- una virtual private cloud
- una o più subnet, dove per subnet si intende un segmento dello spazio di indirizzamento della VPC utile ad isolare risorse
- Internet e NAT gateway
- VPC endpoints. Essi consentono di stabilire connessioni tra un cloud privato virtuale e i servizi supportati senza richiedere un gateway Internet, un dispositivo NAT o una connnessione VPN.
- Connessioni peering con altre VPC. Esse permettono di instradare il traffico tra le reti utilizzando indirizzi IPv4 o IPv6 privati.

Come riporta la documentazione ufficiale di AWS [22], la creazione di una VPC non richiede costi aggiuntivi. Tuttavia le funzionalità opzionali, come per esempio l'utilizzo di NAT gateway o l'analisi del traffico sono a pagamento.

2.7.4 AWS Identity and Access Management (IAM)

La sicurezza è un requisito fondamentale per i sistemi cloud e alle volte può risultare anche una limitazione. Negli ambienti di public cloud la sicurezza e la conformità seguono un modello di responsabilità condivisa: il cloud provider si occupa di proteggere l'infrastruttura su cui vengono eseguiti i servizi cloud, mentre la responsabilità del cliente è determinata dai servizi scelti. Ad esempio, per i servizi IaaS, il cliente deve occuparsi personalmente degli aggiornamenti e patch di sicurezza del sistema operativo, del software applicativo e delle configurazioni del firewall.
Da Figura 2.7 si può notare che nell'ambiente di esecuzione delle pipeline ETL è stato

Categoria	Numero di servizi
Analisi	13
Integrazione di applicazioni	7
Blockchain	2
Gestione finanziaria del cloud	6
Calcolo	17
Database	11
Strumenti per sviluppatori	17
Elaborazione degli utenti finali	3
Front-end Web e dispositivi mobili	7
Internet of Things	16
Machine Learning	32
Gestione e governance	25
Servizi multimediali	11
Migrazione e trasferimento	9
Reti e distribuzione di contenuti	14
Tecnologie quantistiche	1
Robotica	1
Satellite	1
Sicurezza, identità e conformità	20
Archiviazione	9

Tabella 2.1: Numero di servizi AWS suddivisi per categoria

attivato IAM, un servizio che permette di controllare gli accessi alle risorse AWS associando i permessi degli utenti autenticati ai servizi attivati. Nelle organizzazioni accade spesso di avere più utenti che richiedono l'accesso allo stesso account AWS. Prima di questo servizio era comune condividere le stesse credenziali spesso anche in modo non sicuro, per esempio tramite mail. Questo comportamento ha forti ripercussioni sulla sicurezza cloud, per questo motivo AWS tramite IAM permette di creare credenziali uniche per ogni utente e definire chi ha accesso a quale risorsa, non è più necessario condividerle.
I principali componenti su cui si basa IAM sono:

- **user**: entità che rappresenta una persona o un'applicazione a cui sono associati un nome, delle credenziali e dei permessi. Ogni volta che viene creato un nuovo user,

IAM genera tre identificativi per riconoscere un utente: un *friendly name* (quello specificato dal customer), un ID unico e un Amazon Resource Name (ARN), ossia una stringa che identifica in modo univoco una risorsa in tutto AWS. Di default, un utente appena creato non è autorizzato ad eseguire alcuna azione in AWS. Definire i permessi per ogni utente permette di gestire le risorse cloud ad una granularità molto fine;

- **group**: un gruppo è semplicemente l'aggregazione logica di più utenti che condividono gli stessi permessi. L'introduzione di questo componente ha permesso a IAM di semplificare la gestione degli utenti, raggruppando gli utenti per risorse accessibili simulando la struttura aziendale. Un utente può appartenere a più gruppi e un gruppo non può essere innestato all'interno di un altro;

- **role**: un ruolo è un'identità di IAM con permessi specifici che ne definiscono le azioni permesse e negate. Un ruolo IAM è simile ad un utente IAM con la differenza che il primo non è associato ad un utente o ad un gruppo specifico, bensì sono le entità che possono assumerlo. Inoltre un ruolo non ha credenziali a lungo termine come password o access key associate: nel momento in cui è necessario assumere un ruolo vengono fornite credenziali di sicurezza temporanee per la sessione corrente. Per questo motivo i ruoli sono molto utili per delegare accessi a utenti, applicazioni o servizi che normalmente non avrebbero accesso alle risorse AWS;

- **policy**: stabilisce le autorizzazioni e controlla l'accesso alle risorse AWS a cui viene associata. Una policy può essere connessa sia a un utente, a un gruppo o a un ruolo e possono essere di sei tipi differenti. Le più comuni sono quelle *identity-based* che gestiscono i permessi delle entità, mentre le *resource-based* sono associabili alle risorse e concedono al soggetto il permesso di eseguire azioni specifiche su quella risorsa. Le *permission boundaries* definiscono il numero massimo di permessi che una policy identity-based può concedere ad una entità. Un altro tipo di policy è chiamata *organizations service control policy* ed è utilizzata per definire il numero massimo di permessi associabili ad ogni account dell'organizzazione, mentre le *ACL policy* controllano quali soggetti di altri account possono accedere alla risorsa a cui è collegata l'ACL. In ultimo ci sono le *session policy* che limitano i permessi che un ruolo o un utente concedono alla sessione.

IAM è un servizio di AWS che è disponibile senza costi aggiuntivi.

2.7.5 AWS Secrets Manager

Per evitare di memorizzare credenziali e token all'interno del codice delle pipeline ETL si è fatto uso di AWS Secrets Manager, come mostrato in Figura 2.7.
AWS Secrets Manager è un servizio che permette di archiviare credenziali, chiavi API, token e altri segreti di sicurezza. Gli utenti e le applicazioni possono recuperare un segreto mediante una chiamata alle API Secrets Manager senza dover memorizzare tale segreto all'interno del codice dell'applicazione. È possibile combinare questo servizio con IAM, in modo da controllare l'accesso alla chiave segreta utilizzando le policy di IAM. Come molti altri servizi, anche Secrets Manager supporta la replica automatica in più regioni AWS:

in questo modo in caso di fallimento dei data center di una regione la chiave sarà sempre recuperabile da un altra regione.

2.7.6 Amazon Elastic Container Registry (Amazon ECR)

Per raggruppare in modo sicuro, scalabile e affidabile le immagini dei container associati alle pipeline ETL si è fatto uso di Amazon Elastic Container Registry, come mostrato in Figura 2.7. Questo servizio semplifica la gestione del ciclo di vita delle immagini agli sviluppatori, dal momento che è connesso con gli altri servizi Cloud di AWS, come per esempio Amazon ECS. Dal punto di vista puramente teorico, ECR è l'analogo del più conosciuto DockerHub. Supporta gli standard Open Container Initiative e Docker Registry HTTP API V2, e per questo è possibile utilizzare gli stessi comandi di Docker per caricare, estrarre e taggare le immagini.
Una caratteristica interessante di Amazon ECR è la possibilità di gestire il ciclo di vita delle immagini tramite le cosiddette *lifecycle policies*, le quali permettono di definire delle regole che portano alla rimozione di immagini inutilizzate e , dal momento che il costo di ECR è direttamente proporzionale al numero di immagini memorizzate, consente di diminuire i costi. Inoltre è possibile configurare ogni repository in modalità *scan on push*, con la quale viene analizzata ogni immagine prima di essere caricata per identificare eventuali vulnerabilità software. In particolare AWS offre due tipi di scanning, *enhanced scanning* e *basic scanning*. La prima tipologia si integra con Amazon Inspector per fornire scansioni continue e automatizzate dei repository e, quando appaiono nuove vulnerabilità, i risultati della scansione vengono aggiornati e viene notificato all'utente. Diversamente il basic scanning sfrutta il *Common Vulnerabilities and Exposures*, un database open source rilasciato nel 1999 dalla MITRE corporation per identificare e classificare le vulnerabilità di software e firmware [23].

2.7.7 Amazon Auto Scaling

AWS Auto Scaling è un servizio che, monitorando l'utilizzo delle risorse infrastrutturali, è in grado di regolare automaticamente le capacità computazionali per mantenere le prestazioni costanti e affidabili al minor costo possibile. È un servizio molto utilizzato in quanto permette di scalare l'applicazione in modo semplice e veloce, oltre che ottimizzare l'utilizzo e l'efficienza dei costi dei servizi AWS, in modo da pagare solo le risorse effettivamente necessarie. Quando la domanda diminuisce, AWS Auto Scaling rimuove automaticamente la capacità di risorse in eccesso per evitare spese eccessive. Agli occhi dell'utente finale questo servizio è totalmente trasparente; infatti non dovrebbe notare eventuali picchi di richiesta.

2.7.8 Amazon Elastic Compute Cloud (Amazon EC2)

Amazon EC2 è un servizio di cloud computing che permette di eseguire applicazioni e servizi web su macchine virtuali nell'ecosistema scalabile di AWS. È adatto per gli sviluppatori o le aziende che non hanno o non possono disporre dell'infrastruttura fisica necessaria per la propria applicazione o non sono in grado di gestire carichi computazionali improvvisi.
Essendo un servizio cloud, offre la possibilità di riservare le risorse in base alle proprie esigenze. Proprio per questo motivo si dice che EC2 è un servizio elastico, in quanto è

possibile aumentare e diminuire le prestazioni hardware e software dinamicamente.
Amazon EC2 è anche altamente flessibile, in quanto è possibile scegliere tra diversi tipi di istanze e sistemi operativi. Infatti sono disponibili istanze general purpose, compute optimized, memory optimized, storage optimized e molte altre. In fase di definizione dell'istanza, il programmatore seleziona le dimensioni di storage, la CPU e il software applicativo più consoni ai propri obiettivi.
Le informazioni che riguardano il sistema operativo, i software applicativi e la configurazione di lancio dell'istanza sono contenuti all'interno delle cosiddette *Amazon Machine Image* (AMI). Di default AWS mette a disposizione immagini gestite e supportate gratuitamente, ma è possibile anche creare la propria AMI custom. Se più istanze EC2 richiedono le stesse configurazioni è possibile anche lanciarle usando la stessa AMI.
Oltre a definire le caratteristiche hardware della macchina, è possibile definire anche la rete dentro la quale risiederà la macchina virtuale e volendo anche dei gruppi di sicurezza, i quali consentono di limitare e controllare il traffico in entrata ed in uscita di una qualsiasi risorsa.
Al fine di ottimizzare i costi, AWS fornisce diverse opzioni di acquisto delle istanze:

- **on-demand instances**: il pagamento avviene in base all'utilizzo della risorsa ad ogni secondo;
- **savings plans**: permette di ridurre il costo delle istanze EC2 utilizzando macchine con un carico costante per un periodo di 1 o 3 anni. Il costo è in dollari all'ora;
- **reserved instances**: permette di ridurre il costo delle istanze EC2 utilizzando macchine con una configurazione costante, incluso il tipo di istanza e la regione AWS per un periodo di 1 o 3 anni;
- **spot instances**: sono richieste istanze inutilizzate con lo scopo di ridurre i costi;
- **dedicated hosts**: viene dedicato interamente un host fisico per eseguire le istanze;
- **dedicated instances**: pagamento ad ora per istanze eseguite su un hardware a tenant singolo;
- **capacity reservations**: possibilità di prenotare risorse per le istanze per qualsiasi durata.

In Figura 2.8 è mostrata la differenza di prezzo di una macchina EC2 di tipo *t3 medium* (2 VPCU e 4 GB di memoria) con diversi piano di acquisto.

2.7.9 Amazon Elastic Container Service (Amazon ECS)

Amazon ECS è un servizio di orchestrazione altamente scalabile che permette di rilasciare, eseguire, gestire e distribuire containers su un cluster. Supporta nativamente Docker, e questo è molto utile in quanto è possibile gestire le applicazioni containerizzate nello stesso modo nel quale si gestiscono i container Docker in locale. AWS ECS esonera il programmatore dalla gestione dei nodi e del control plane del cluster, permettendo di concentrarsi solamente sul codice applicativo.
Facendo parte dell'ecosistema AWS, anche ECS è in grado di integrarsi con gli altri servizi

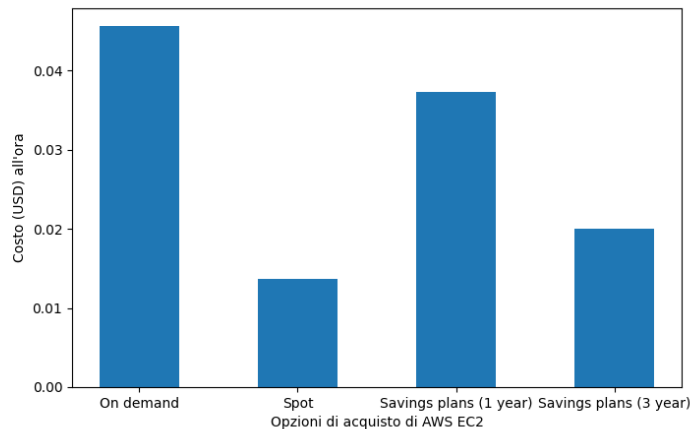


Figura 2.8: Differenza di costo per diverse opzioni di acquisto di una macchina EC2

di sicurezza messi a disposizione dal provider, quali per esempio IAM. Questo fa si che sia possibile assegnare permessi specifici ad ogni container garantendo un elevato livello di isolamento nella creazione delle applicazioni.
Questo servizio può essere utilizzato con un repository di immagini Docker mantenuto da terze parti o accedere direttamente alle immagini contenute all'interno di Amazon Elastic Container Registry (ECR).
Un costrutto fondamentale di ECS sono i cosiddetti *task*, descritti come l'istanziazione di una *task definition* all'interno di un cluster. In particolare, una task definition è un file JSON che descrive uno o più containers che formano un'applicazione. All'interno di questo file è possibile specificare diversi parametri, come per esempio quali porte aprire o anche quali dati deve usare un container. Un altro componente fondamentale di ECS è il cosiddetto *container agent*, il quale esegue su ogni container contenuto in un cluster ECS con lo scopo di inviare informazioni riguardo i task in esecuzione e le risorse utilizzate dal container.
Amazon ECS fornisce anche un *capacity provider*, il quale permette di determinare e gestire l'infrastruttura che un task necessita per eseguire su un cluster. Nel momento in cui viene eseguito un task, se non specificato, viene utilizzato il capacity provider di default del cluster. Questo componente astrae la gestione delle risorse al programmatore che non si deve più preoccupare dell'architettura ma solo del codice applicativo. Interessante è il caso di utilizzo di un capacity provider con un Auto Scaling group per gestire le istanze EC2 associate al cluster. Si immagini per esempio lo scenario descritto in Figura 2.9, in cui è presente un cluster ECS su cui devono essere eseguiti cinque task su macchine EC2. Tramite il capacity provider il cluster è in grado di determinare quante risorse sono necessarie e, fornendo questa informazione all'Auto Scaling Group, quest'ultimo istanzierà il numero minimo di macchine EC2 utili ad eseguire tutti i tasks.
Amazon ECS supporta sia l'infrastruttura di rete di Docker sia il servizio di VPC di aws. Entrambe le reti permettono di isolare e definire il modo in cui i container interagiscono tra di loro, anche se la rete messa a disposizione da AWS consente di utilizzare quattro

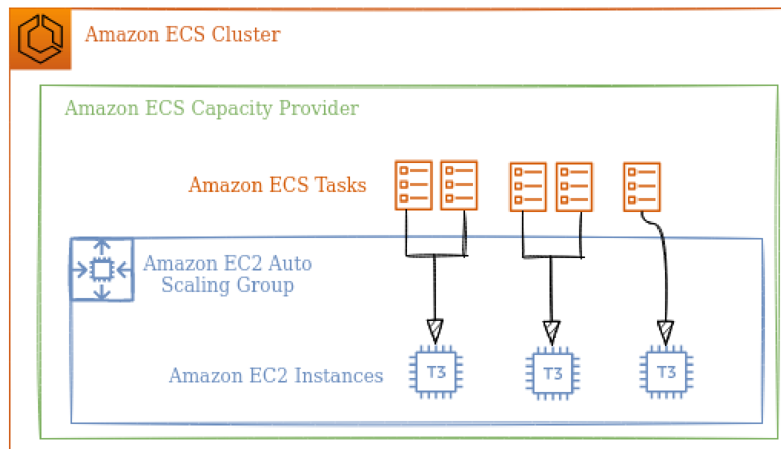


Figura 2.9: Esempio di Auto Scaling Group con un Capacity Provider in un cluster ECS

diverse metodologie:

- **awsvpc**: questa modalità assegna ad ogni task ECS un'interfaccia di rete dedicata;
- **bridge**: questa modalità crea un bridge che connette tutti i container che eseguono su una macchina all'interno di una VPC, alla quale si può accedere tramite la sua interfaccia di rete;
- **host**: questa modalità aggiunge i container direttamente allo stack di rete dell'host, esponendo i container sulla rete senza isolarli;
- **none**: questa modalità non definisce alcuna rete.

Amazon ECS supporta due modalità di avvio dei container, tramite EC2 e tramite Fargate. In particolare, questa seconda tecnologia è stata introdotta per supportare l'elaborazione *serveless*. Questo significa che Fargate permette di eseguire containers senza dover gestire server o cluster di EC2: il programmatore è sollevato dalla scelta del tipo di server, quando scalare il cluster e le varie configurazioni di ottimizzazione. Infatti per eseguire un task Fargate è necessario specificare solamente i requisiti di CPU e memoria, la rete e le policy IAM. Concettualmente il servizio è analogo a EC2, con la sola differenza che è serverless.

2.7.10 Amazon EC2 vs. Amazon Fargate

Dopo aver definito i due tipi di servizi, in questa breve sezione si vogliono descrivere i casi d'uso delle due tecnologie. Infatti i costi dei due servizi non solo uguali, dal momento che EC2 delega al programmatore la gestione e la definizione dell'architettura sottostante. Il costo di EC2 è basato sul tipo di istanze utilizzate; questo permette di ottimizzare i costi scegliendo le istanze che più fanno al caso dell'utente. Inoltre è possibile anche riservare delle istanze per periodi medio/lunghi in modo da ottenere delle promozioni. Tuttavia è responsabilità dell'utente assicurarsi di scegliere le istanze migliori sulla base dei container che devono essere eseguiti, con il rischio di uno spreco di denaro. D'altro canto, la metrica di costo di Fargate si basa su quanta CPU e memoria richiedono i task al secondo: questo fa

si che si paga solamente per quello che si usa. Ovviamente il costo al secondo per lo stesso hardware di Fargate è maggiore di quello per EC2. In particolare Fargate è più adatto per:

- piccoli carichi di lavoro con sequenza occasionale;
- carichi di lavoro ridotti;
- lavori in cui il programmatore non è in grado di gestire l'infrastruttura

Al contrario EC2 è ottimo per:

- carichi di lavoro che richiedono un utilizzo costante di core e memoria della CPU
- lavori in cui il programmatore è in grado di gestire in autonomia l'infrastruttura

2.7.11 Amazon CloudFormation

AWS CloudFormation è un servizio di IaC (Infrastructure as Code) che permette la gestione e il provisioning delle risorse infrastrutturali tramite codice anziché con processi manuali. Questo servizio sfrutta la nozione di modello, ovvero un'entità in grado di descrivere tutte le risorse e le relative proprietà. In seguito, a partire da uno o più modelli, CloudFormation si occuperà di allestire e configurare tutte le dipendenze tra le risorse, che altrimenti sarebbero abbastanza complesse da gestire.
Un altro costrutto fondamentale di CloudFormation è lo *stack*, ovvero una raccolta di risorse AWS che è possibile gestire come una singola unità. Questo permette di creare, aggiornare o eliminare l'intera infrastruttura di un progetto semplicemente agendo sullo stack. Oltre a permettere di gestire più risorse contemporaneamente, l'utilizzo di uno stack facilita il programmatore nel tracciamento delle modifiche all'infrastruttura; questo significa che, nel caso si verificassero problemi dopo un aggiornamento dello stack, sarà sempre possibile effettuare un roll-back alla versione precedente.

2.7.12 Amazon Relational Database Service (Amazon RDS)

La gestione di basi di dati in cloud (per esempio su macchine EC2) o su architetture on-premise è complessa, dal momento che ci sono più aspetti da considerare quali la sicurezza, la disponibilità, le prestazioni e la scalabilità. Per questo motivo è nato Amazon Relational Database Service (più in breve RDS), un servizio web che permette di configurare, utilizzare e ridimensionare le risorse di database relazionali nel cloud di AWS. Questo servizio permette di replicare la base di dati in più availability zones per migliorarne la disponibilità e l'affidabilità, oltre che attivare tecniche di *failover* automatiche per commutare ad un database secondario ridondante in modo sincrono nel caso di interruzioni nel funzionamento del database primario.
Amazon RDS solleva l'utente dalla gestione del software in esecuzione sul database relazionale, sarà lo stesso servizio che si assicura l'installazione e l'aggiornamento delle ultime patch, anche se è possibile specificare se e quando installare gli aggiornamenti sulla base di dati.
Attualmente Amazon RDS supporta diversi engine, quali per esempio MySQL, Oracl, SQL Server e PostreSQL.
Le Tabelle 2.2 e 2.3 confrontano la gestione delle basi di dati rispettivamente su architetture on-premises contro AWS EC2 e tra EC2 contro RDS.

Feature	On-premises management	Amazon EC2 management
Application optimization	Customer	Customer
Scaling	Customer	Customer
High availability	Customer	Customer
Database backups	Customer	Customer
Database software patching	Customer	Customer
Database software install	Customer	Customer
Operating system (OS) patching	Customer	Customer
OS installation	Customer	Customer
Server maintenance	Customer	AWS
Hardware lifecycle	Customer	AWS
Power, network, and cooling	Customer	AWS

Tabella 2.2: Gestione di un database su infrastruttura on-premises e su Amazon EC2. Tabella tratta da [24].

2.7.13 Amazon S3

Amazon Simple Storage Service è un servizio di storage in cloud che offre scalabilità, disponibilità dei dati e sicurezza. I dati sono memorizzati come oggetti nei cosiddetti *bucket*, ovvero un container per oggetti o file non più grandi di 5 TB. Ogni oggetto contiene una chiave che corrisponde all'identificatore univoco dell'oggetto all'interno del bucket, quindi la combinazione di bucket, chiave oggetto e, se il versionamento è abilitato per il bucket anche l'ID della versione, identificherà in modo univoco ogni oggetto.
Per gestire e monitorare gli accessi ad un bucket ci sono diverse alternative: si possono specificare direttamente le policy di accesso, si può utilizzare il servizio IAM o in alternativa si possono generare delle liste di controllo degli accessi (ACL).
Inoltre Amazon S3 mette a disposizione diverse classi di storage concepite per i diversi casi d'uso [25]:

- **S3 standard**: la classe di storage predefinita. Offre un'archiviazione di oggetti con durabilità, disponibilità e prestazioni elevate per i dati di accesso più frequenti. Fornisce bassa latenza e throughput elevato, e quindi è adatto per una vasta gamma di casi d'uso, tra cui applicazioni cloud e analisi di big data;

- **S3 Intelligent-Tiering**: questa classe trasferisce automaticamente i dati tra due livelli di accesso per sfruttare il livello di accesso più conveniente sulla base della frequenza di accesso ai dati. È ideale per dati con modelli di accesso sconosciuti o mutevoli;

Feature	Amazon EC2 management	Amazon RDS management
Application optimization	Customer	Customer
Scaling	Customer	AWS
High availability	Customer	AWS
Database backups	Customer	AWS
Database software patching	Customer	AWS
Database software install	Customer	AWS
Operating system (OS) patching	Customer	AWS
OS installation	Customer	AWS
Server maintenance	AWS	AWS
Hardware lifecycle	AWS	AWS
Power, network, and cooling	AWS	AWS

Tabella 2.3: Gestione di un database su Amazon EC2 e su Amazon RDS. Tabella tratta da [24].

- **S3 Standard-Infrequent Access**: questa classe consente accessi rapidi in casi di necessità a dati pensati per non essere acceduti di frequente. Offre le stesse prestazioni della classe S3 standard ed è la soluzione ideale per archiviazione a lungo termine, come per esempio i backup;
- **S3 One Zone-Infrequent Access**: questa classe è concepita per dati ad accesso poco frequente, ma a cui non serve il modello multiplo di resilienza per zona di disponibilità, che invece la classe Standard-Infrequent Access ha;
- **S3 Glacier**: è un insieme di classi che sono dedicate all'archiviazione dei dati nel lungo periodo. Ovviamente il costo di questo tipo di storage è minore di tutti gli altri, in quanto i tempi di recupero dei dati possono andare dai 5 minuti fino alle 12 ore, in base alla tipologia di archiviazione scelta.

Un altra caratteristica interessante di Amazon S3 è la possibilità di hostare un sito web statico a partire da un insieme di file contenuti in un singolo bucket.

2.7.14 Amazon Athena

Amazon Athena è un servizio serverless di query interattivo che semplifica l'analisi dei dati contenuti in Amazon S3 tramite espressioni SQL standard. Proprio per il fatto che è un servizio serverless, Athena permette di eseguire rapidamente query sui bucket S3 senza configurare e gestire server o data warehouse pur mantenendo ottime prestazioni.

Come ogni altro servizio cloud di AWS, anche Athena segue un modello di pagamento pay-per-use; in particolare i costi dipendono dalla quantità di dati scansionati per ciascuna interrogazione.
In Athena, tabelle e database sono contenitori definiti da metadati che definiscono lo schema dei dati di origine sottostanti, cioè il nome e il tipo delle colonne, il nome della tabella e la locazione di S3 dove sono memorizzati i dati.

2.7.15 Amazon CloudWatch

Amazon CloudWatch è un servizio web che permette di monitorare l'utilizzo delle risorse infrastrutturali e per applicazioni on-premises, AWS e ibride. Viene spesso utilizzato sia per tenere traccia dei parametri legati alle macchine EC2 per raccogliere e archiviare i logs delle applicazioni in tempo reale. Un altra caratteristica interessante è la possibilità di creare degli allarmi su qualsiasi parametro di una risorsa; ad esempio si può impostare un allarme sul tempo di esecuzione di una macchina Fargate in modo che, scaduto questo tempo, viene inviata una notifica e stoppata l'esecuzione.

2.7.16 Amazon Glue

Amazon Glue è un serizio serverless di ETL che permette di estrarre i dati da una sorgente, trasformarli e in ultimo caricarli su un data warehouse in modo semplice ed astraendo la componente infrastrutturale di questo processo. Infatti una delle parti più difficili in un progetto di data warehousing è la gestione e il mantenimento di un processo ETL affidabile. In questo contesto si colloca Glue, infatti tramite questo servizio il programmatore non si deve preoccupare più della ricerca di tutte le sorgenti dati e dei loro schemi, oltre che della scalabilità della soluzione.
AWS Glue si compone di:

- **data catalog**: componente che archivia i metadati e le strutture dati organizzandole in tabelle all'interno di databases. Il suo scopo è quello di centralizzare dati strutturati o semi strutturati derivanti da diverse fonti dati eterogenee per poterli interrogare e trasformare in modo coerente da una vasta gamma di applicazioni;
- **crawler**: analizza i contenuti delle sorgenti dati e crea o aggiorna una o più tabelle nel data catalog;
- **ETL engine**: motore che, usando i metadati contenuti nel data catalog, genera automaticamente gli scripts PySpark o Scala utili all'utente per eseguire le varie operazioni ETL;
- **job**: implementa la *buisness logic* di un task della ETL usando i metadati del data catalog. Supporta i linguaggi Python e Scala, oltre che fornire nativamente supporto per PySpark;
- **console**: permette di definire e orchestrare il flusso di lavoro ETL.

La Figura 2.10 mostra l'architettura di AWS Glue, che presenta i componenti appena descritti. AWS Glue supporta anche un'interfaccia grafica, chiamata *AWS Glue Studio*, che

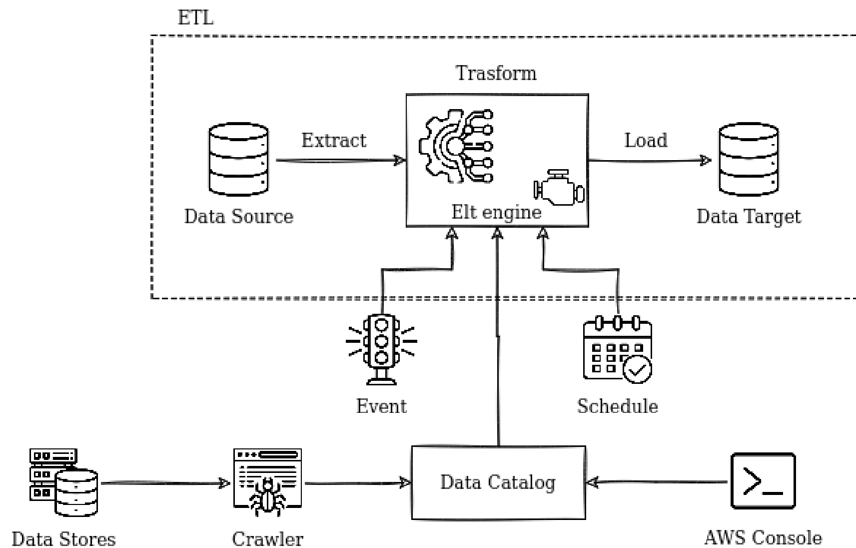


Figura 2.10: Architettura di AWS Glue. Immagine che prende spunto da [26]

permette di creare, eseguire e monitorare job ETL in cloud. Grazie a questa UI è possibile generare un job suddividendolo nei diversi step, e per ogni step è possibile monitorare il suo risultato.
Glue è un servizio che si differenzia da quelli attualmente in commercio in quanto è in grado automaticamente di determinare la posizione dei dati, il loro schema e costruire un data catalog centralizzato. In secondo luogo genera automaticamente il codice ETL per estrarre, trasformare e caricare i dati in Python o Scala ed in ultimo fornisce anche tutti i vantaggi dei servizi serverless.

2.7.17 Amazon Managed Workflows for Apache Airflow (MWAA)

Amazon Managed Workflows for Apache Airflow è un servizio di AWS che permette di utilizzare Apache Airflow sull'infrastruttura cloud di Amazon. Grazie a MWAA è possibile creare flussi di lavoro senza dover gestire l'infrastruttura sottostante sfruttando le caratteristiche di scalabilità, disponibilità e sicurezza proprie di AWS. In Figura 2.11 è mostrata l'architettura di MWAA; si può notare come lo scheduler e i workers Airflow sono applicazioni containerizzate che eseguono in AWS Fargate, i quali si connettono alle subnet private in ogni Amazon VPC. Ogni enviorment MWAA contiene il proprio Apache Airflow Metadatabase, gestito direttamente da AWS, che è accessibile al container dello scheduler e ai container dei workers tramite un endpoint VPC sicuro.
MWAA supporta l'integrazione con strumenti di terze parti come Apache Hadoop, Presto, Hive e Spark oltre che ovviamente con i servizi AWS; per esempio i DAG devono essere caricati in appositi bucket S3.

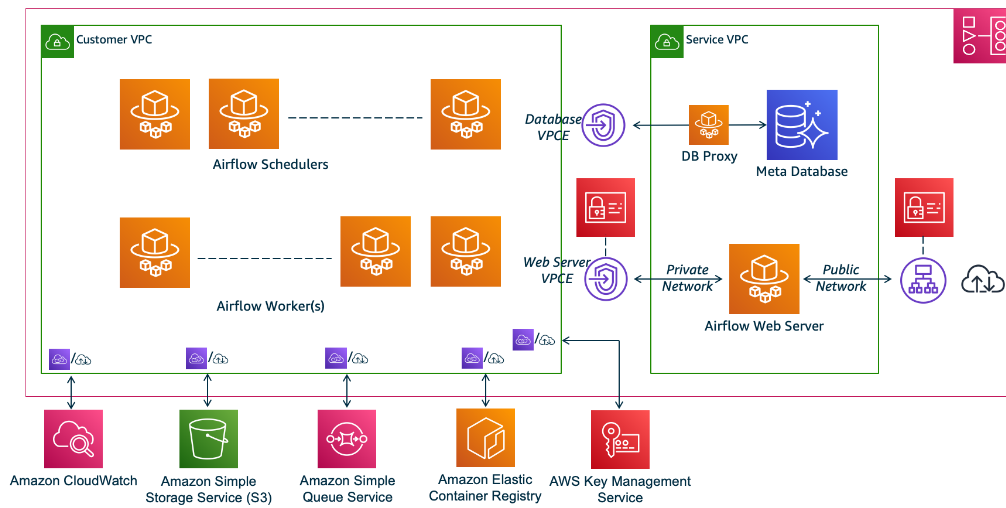


Figura 2.11: Architettura di AWS MWAA. Immagine tratta da [27]

Parte II

Progetto e sviluppo delle pipeline ETL

Capitolo 3

Progettazione

Il presente capitolo ha lo scopo di descrivere il progetto nato per merito di un tirocinio curricolare presso Ammagamma [1], una società che tramite lo sviluppo di soluzioni di intelligenza artificiale punta a contribuire allo sviluppo di una società consapevole delle potenzialità, delle implicazioni e degli impatti della tecnologia. In seguito saranno descritti gli obiettivi posti ad inizio collaborazione e il modo in cui sono stati ottenuti. Più nello specifico saranno motivate le scelte progettuali riguardo l'architettura cloud utilizzata e le problematiche riscontrate.

3.1 Lo stato dell'arte

Se ben addestrati, i modelli di intelligenza artificiale sono in grado di fornire predizioni molto accurate e utili. Per questo motivo, da alcuni anni a questa parte, le aziende si sono concentrate sempre più nella raccolta dei dati, nella loro trasformazione e nell'estrazione di valore anche tramite l'utilizzo di modelli di intelligenza artificiale.
All'interno della collaborazione in corso da svariati anni con un noto quotidiano economico politico finanziario italiano, Ammagamma ha sviluppato una serie di modelli di predizione del tasso di abbandono (in inglese *churn rate*) sui loro principali abbonamenti. Questi modelli prendono in ingresso una serie di dati e, sulla base della storia commerciale e di navigazione di un cliente in un determinato momento prima della fine dell'abbonamento, prevedono la probabilità di abbandono di quest'ultimo. L'output dei modelli viene aggiunto ai KPI (*key performance indicator*) e passato nuovamente ai sistemi aziendali.
Al fine di estrarre, trasformare, predire e fornire i risultati in output sono in produzione quattro pipeline ETL, divise in coppie, che valutano rispettivamente il tasso di abbandono mensile ed annuale. Tuttavia la mole di dati in aumento ha richiesto alcune modifiche alle pipeline attualmente in essere, che hanno difficoltà a scalare oltre un certo limite.
Gli obiettivi del progetto di tirocinio definiti sono principalmente due:

- costruzione e schedulazione di una pipeline ETL scalabile che estragga, aggreghi e prepari il dato per modelli predittivi e ne appenda il risultato all'output finale;
- confronto delle prestazioni tra la pipeline attualmente in produzione e quella rimaneggiata.

3.1.1 Descrizione delle pipeline ETL

Come già detto in precedenza le pipeline attualmente in produzione sono quattro, a cui possiamo simbolicamente assegnare i seguenti nomi:

- *etl_annual*
- *etl_annual_qv*
- *etl_monthly*
- *etl_monthly_qv*

Come si può notare dalla terminologia, le pipeline si possono suddividere in due famiglie: ETL mensili ed ETL annuale. Ad alto livello, la differenza fra le due classi è che le prime sono eseguite all'inizio di ogni settimana considerando i dati della settimana precedente, mentre le seconde sono schedulate per eseguire il primo di ogni mese considerando i dati del mese precedente. Inoltre per ogni spazio temporale è presente una pipeline con suffisso *qv* che, senza andare nello specifico, serve per gestire un prodotto del quotidiano.
Tutte e quattro le pipeline condividono la stessa struttura, ovvero sono composte da tre layer:

- **trasformation**: in questo stage sono preparati i dati per essere dati in pasto al modello di intelligenza artificiale. Questa fase si compone di task abbastanza pesanti che richiedono risorse di calcolo non banali;
- **modeling**: viene calcolato il churn e l'engagement di un utente. Il risultato di questa operazione dipende direttamente dalla bontà dei modelli di machine learning;
- **presentation**: sono elaborati i dati per rispettare gli schemi predefiniti e vengono eseguiti dei controlli sulle predizioni per verificarne l'accuratezza

In Figura 3.1 è evidenziata la struttura delle due pipeline mensili, dove i blocchi di colore blu sono relativi alla pipeline di tipo qv, i blocchi di colore arancione sono della pipeline standard mentre quelli verdi corrispondono a step comuni alle due pipeline. In modo analogo la Figura 3.2 mostra la struttura delle pipeline annuali. A partire da queste due immagini si può fare una prima considerazione fondamentale per il proseguo del progetto: anche se le ETL annuali presentano più step di quelle mensili, entrambe condividono più o meno la stessa struttura e la stessa logica di fondo. Questo si può ritrovare anche nel codice implementativo delle stesse, e per questo motivo uno dei primi obiettivi del tirocinio è stato quello di eliminare il codice duplicato creando funzioni comuni alle quattro pipeline.

Una nota importante riguarda il primo step di entrambe le pipeline, ovvero quello dal nome *create_users_tables*. Questo script ha il compito di estrarre i dati presenti all'interno di un database PostgreSQL istanziato e gestito tramite AWS RDS in diversi bucket S3 per essere poi interrogati tramite Athena. Anche se questo processo può sembrare del tutto inutile e computazionalmente oneroso, è necessario a causa dell'inadeguatezza dell'RDS. Infatti il sistema di archiviazione dei dati è stato progettato ed implementato da un'altra società, e solo in seguito è stato passato il testimone ad Ammagamma. La scelta di RDS non si è rivelata adeguata alle esigenze di progetto. La mole di dati inserita nel database

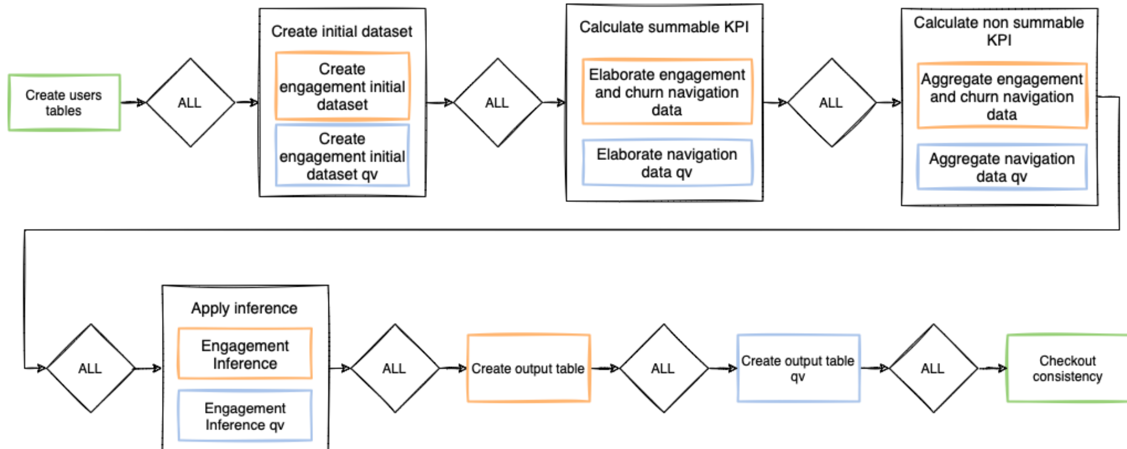


Figura 3.1: Struttura delle pipeline ETL mensili

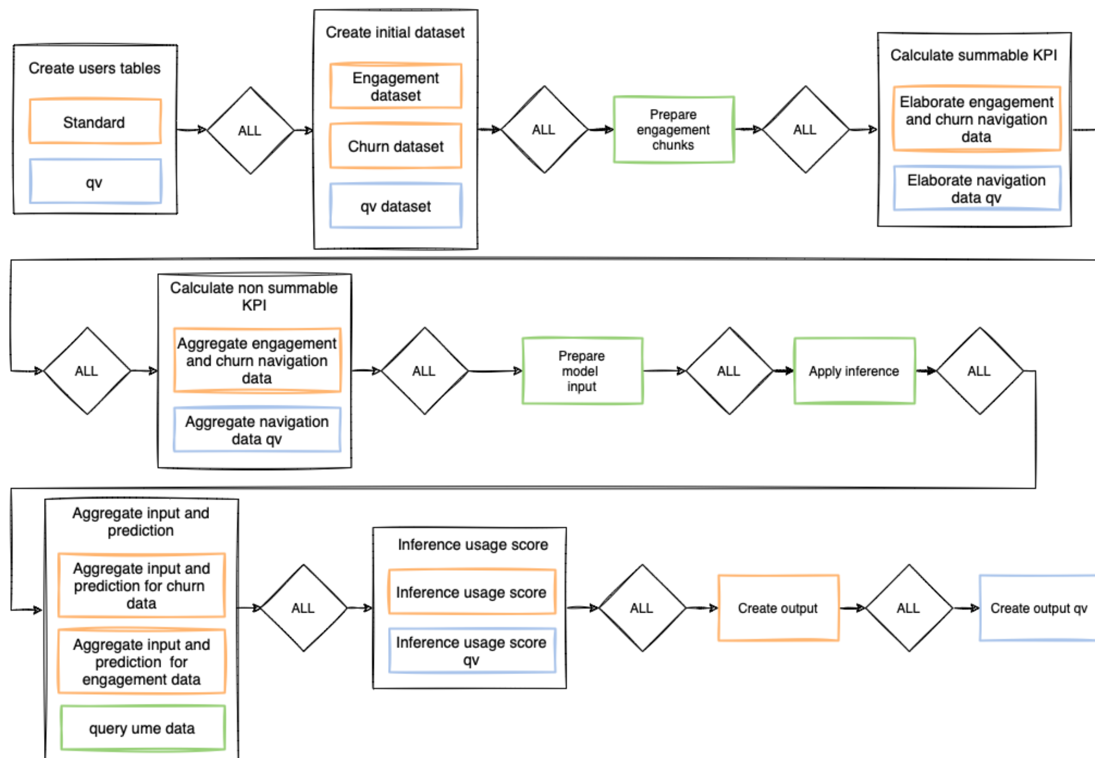


Figura 3.2: Struttura delle pipeline ETL annuali

PostgreSQL ne rende necessario un upscaling periodico e comporta tempi molto lunghi per il completamento delle query necessarie alla creazione delle basi dati per i modelli di churn. Per ovviare in parte al problema, il team di Ammagamma ha memorizzato direttamente i dati più pesanti (ovvero quelli relativi alla navigazione) direttamente su S3 e resi accessibili tramite Athena, in modo da rimuovere il collo di bottiglia dopo la prima lettura. Tuttavia

questa soluzione non risolve tutti i problemi, dal momento che l'estrazione dei dati dal database relazione è comunque lenta e viene generata una ridondanza informativa.
Le pipeline sono implementate in Python e le librerie sfruttate in maggioranza sono *pandas* per elaborare i dati caricandoli in dataframe e *boto3*, il software development kit (SDK) di AWS per Python.

3.1.2 Ambiente di esecuzione delle pipeline ETL

In questa sezione si vuole descrivere l'ambiente entro cui le pipeline vengono schedulate. Più nello specifico, si fa uso dei servizi cloud di AWS sia per memorizzare i dati, per schedulare le pipeline, per definire i flussi di lavoro e per eseguire effettivamente i job.
In Figura 3.3 è mostrato l'ambiente di esecuzione delle pipeline, in cui è ovviamente presente AWS Glue per la gestione e l'esecuzione dei job ETL. Il servizio Simple Storage (S3) (descritto in sezione 2.7.13) è utilizzato sia per memorizzare gli scripts delle pipeline sia per contenere i dati interrogabili mediante AWS Athena. Per garantire un buon livello di sicurezza si fa uso di servizi quali IAM e Secrets Manager, mentre per monitorare i log è stato attivato AWS CloudWatch. Inoltre viene utilizzato anche Simple Notification Service, il quale inoltra una email in caso di errore o completamento delle pipeline.
All'interno dell'ambiente AWS sono istanziate anche delle macchine EC2 Windows che hanno lo scopo di estrarre i dati presenti all'interno di fonti dati esterne per memorizzarli all'interno del database PostgreSQL.
Infine vi è una VPC privata, che permette di definire una rete virtuale isolata logicamente in cui vengono schedulate ed eseguite le pipeline.

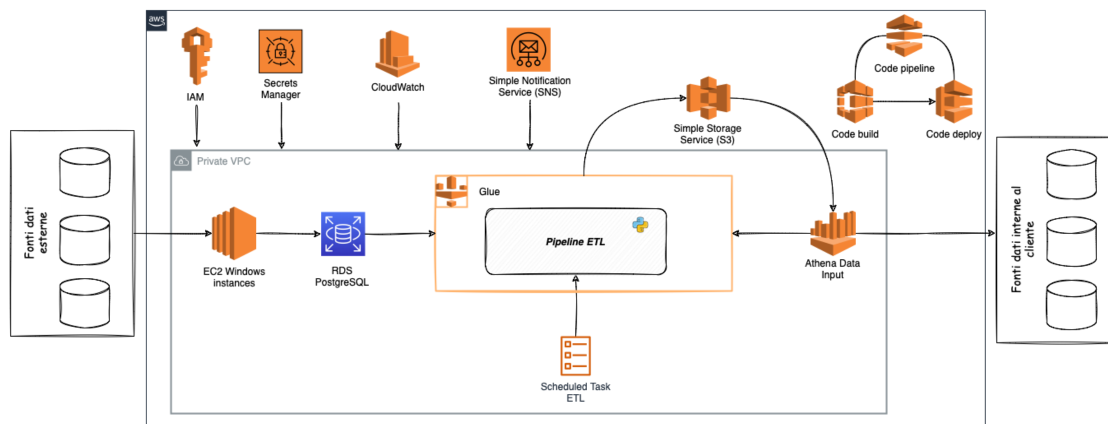


Figura 3.3: Ambiente di esecuzione delle pipeline ETL

3.2 Code Refactoring

Prima ancora di definire la nuova architettura cloud di esecuzione delle pipeline, è stato necessario revisionare il codice per migliorarne le qualità non funzionali. L'obiettivo di

questa fase è quella di aumentare la leggibilità, la manutenibilità, le prestazioni e l'estensibilità del codice oltre che ridurne la complessità e il codice replicato senza modificarne le funzionalità.

3.2.1 Creazione di asset comuni

Dopo aver studiato il codice della pipeline mensile e annuale è stato chiaro che entrambe potevano essere riscritte parzialmente utilizzando un grado di astrazione maggiore. Infatti, come si può notare anche dalla loro struttura mostrata in Figura 3.1 e 3.2, esse hanno una logica di base simile. Quindi, al fine di aumentare la manutenibilità e ridurre la dimensione del codice, si è pensato di creare asset comuni alle pipeline per raggruppare funzioni e costanti utili ad entrambi i flussi ETL.
In particolare sono stati creati i seguenti file Python:

- *commons_aws_etl.py*: contiene le funzioni che, tramite il client Python di aws, interagiscono con i servizi cloud;
- *commons_date_etl.py*: include tutte le funzioni che processano date;
- *commons_processing_etl.py*: contiene funzioni che processano dataframe pandas;
- *commons_database_etl.py*: funzioni che interagiscono con il database PostgreSQL in esecuzione sul servizio RDS;
- *commons_etl.py*: comprende il resto delle funzioni e costanti comuni.

Una nota interessante riguarda il file *commons_database_etl.py*; può sembrare strano creare un asset per includere le funzioni che interagiscono con il database PostgreSQL, dal momento che le pipeline estraggono una sola volta i dati dal database relazionale e li inseriscono in bucket S3 per interrogarli tramite Athena. Infatti questo file contiene solamente una funzione che, dopo aver scaricato i parametri di connessione da AWS Secrets Manager, ritorna una connessione con il database. Tuttavia è stato utile includere tale funzione all'interno di un file indipendente per ovviare ad una limitazione di AWS Glue. Più nello specifico, per importare una libreria esterna all'interno di un job spark di Glue è necessario creare un file .zip della libreria stessa e caricarlo in un bucket S3. Questo comporta che ogni volta che viene fatta una modifica ad una funzione degli asset è necessario ricreare il file compresso e ricaricarlo nel bucket; dal momento che dopo la definizione della nuova architettura le pipeline eseguiranno su Airflow ad eccezione di un singolo job che rimarrà su Glue, e che tale job non sfrutta una connessione con il database PostgreSQL, allora inserire commons_database_etl.py nel file compresso sarebbe risultato inutile e avrebbe aumentato le dimensioni dello stesso (e di conseguenza aumentando il tempo di startup).
Come già detto, entrambe le pipeline interrogano i dati presenti all'interno dei bucket S3 tramite delle query SQL in esecuzione su Athena. Per questo motivo si è cercato di generalizzare quanto più possibile queste interrogazioni tramite l'utilizzo di parametri: in questo modo può capitare che job di pipeline differenti richiamino la stessa query SQL ma con parametri diversi per soddisfare i loro bisogni.

3.2.2 Aggregazione delle pipeline ETL

Per ridurre la dimensione del codice e semplificare anche la sua manutenibilità è stato deciso di aggregare le pipeline con i due livelli temporali che le definiscono. Questo si traduce nel fondere la pipeline di tipo qv a quella standard relativa allo stesso lasso temporale (mensile o annuale). In altre parole, alla fine di questo processo si vogliono ottenere solamente due pipeline:

- *etl_monthly*;
- *etl_annual*.

Il raggiungimento di questo obiettivo è stato semplificato fortemente dalla fase di astrazione e creazione di asset comuni descritti in sottosezione 3.2.1. Inoltre si sono parametrizzati anche i job stessi: l'idea è infatti quella di avere job più generici possibile che, in base alle variabili d'ambiente impostate o ai parametri di ingresso degli script, siano in grado di eseguire o no determinate operazioni.
A seguito di questa trasformazione sia la pipeline mensile che quella annuale avranno una struttura più semplice, come evidenziato rispettivamente in Figura 3.4 e 3.5, nelle quali si può notare la dicitura "*flag*" laddove lo script sia parametrizzato. Si noti come alcuni

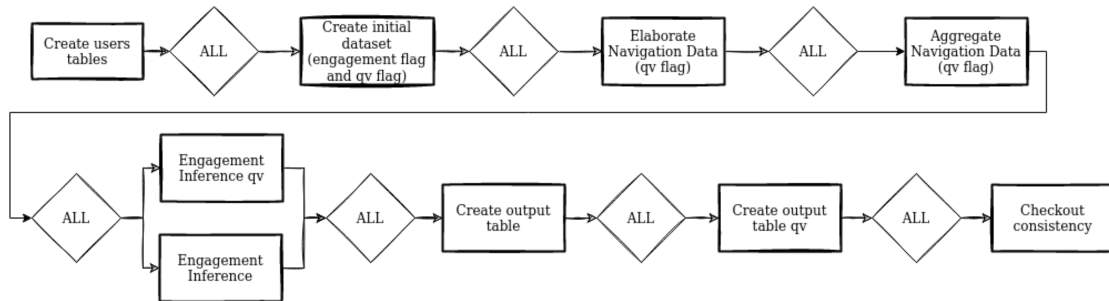


Figura 3.4: Struttura della pipeline ETL mensile dopo il refactoring

job del flusso ETL non sono stati generalizzati e parametrizzati, questo perchè lo sforzo richiesto avrebbe superato abbondantemente il beneficio ottenuto.
Dopo questa fase sarà possibile schedulare la pipeline annuale standard o quella di tipo qv semplicemente passando i parametri desiderati al flusso ETL, ed entrambe avranno la stessa implementazione; lo stesso discorso si può fare anche per la pipeline mensile.

3.3 Creazione dell'ambiente di sviluppo e di produzione

Nella maggior parte dei casi, lo sviluppo di applicazioni di una certa complessità richiede una distinzione tra ambiente di sviluppo, ambiente di test e ambiente di produzione. Lo scopo di questa separazione è di permettere ai programmatori di sviluppare e rilasciare aggiornamenti dell'applicazione senza intaccare l'attuale versione in uso sugli utenti finali. La creazione di un ambiente di *developement* e un ambiente di *production* è una necessità

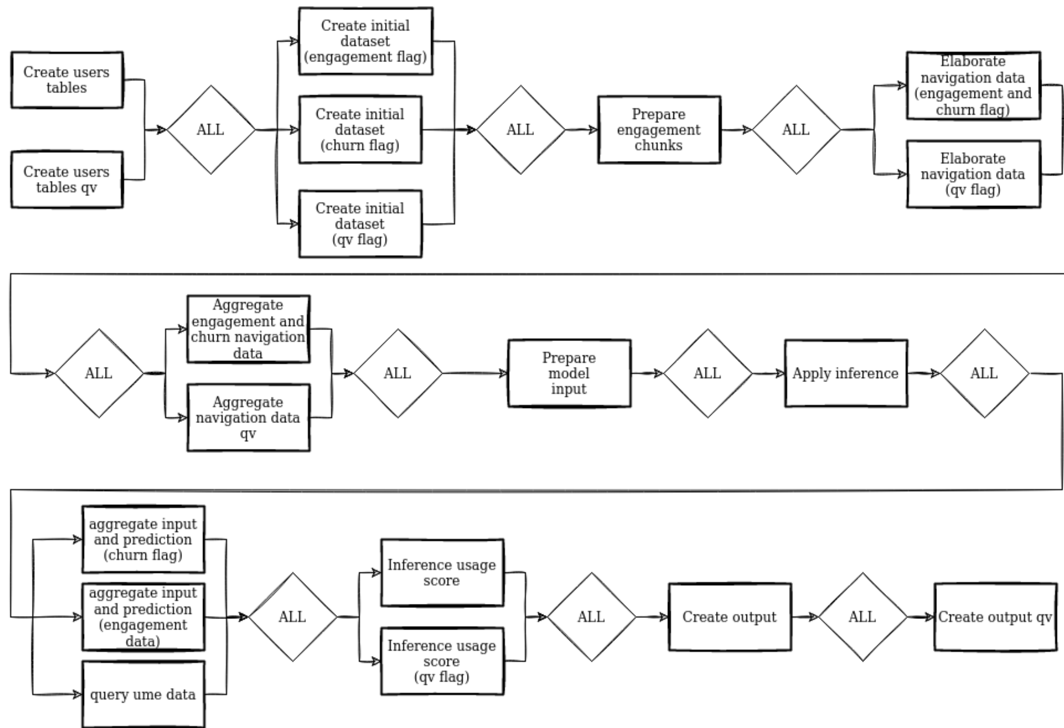


Figura 3.5: Struttura della pipeline ETL annuale dopo il refactoring

nata in corso d'opera a questo progetto, in quanto si aveva l'esigenza di testare la pipeline revisionata senza modificare i risultati delle precedenti esecuzioni della stessa. Infatti memorizzare l'output delle predizioni e delle interrogazioni negli stessi bucket S3 avrebbe portato alla sostituzione degli stessi, perdendo quindi la possibilità di compararli con i risultati della pipeline non revisionata.
Per risolvere questo problema si è fatto uso di una semplice variabile d'ambiente: se infatti viene definita la variabile di nome *production_run* allora le costanti all'intero dei commons punteranno a servizi e risorse utilizzate in produzione, altrimenti punteranno a risorse di default utilizzate per l'ambiente di sviluppo. Questo approccio, anche se spartano, permette di trattare i job della pipeline allo stesso modo in entrambi gli ambienti, i quali sfrutteranno solamente risorse differenti.

3.4 Valutazioni sui motori di gestione di workflow

Il panorama tecnologico attuale propone svariate tecnologie per schedulare e monitorare flussi di lavoro, ognuna delle quali è maggiormente indicata per ambiti applicativi diversi. In Tabella 1.1 di Sezione 1.1 ne viene fornita una panoramica. Come già anticipato, il framework utilizzato per monitorare le pipeline ETL del progetto è Apache Airflow, il quale è stato descritto approfonditamente nel Capitolo 1. Questa scelta deriva da uno studio preliminare che ha messo a confronto AWS Glue, AWS MWAA e Apache Airflow.

Il seguente paragrafo ha lo scopo di motivare la scelta tecnologica sulla base di diverse metriche di valutazione.

3.4.1 Analisi sul costo dei servizi

La prima metrica di valutazione è il costo all'ora dei servizi, dal momento che le pipeline devono essere schedulate su un ambiente cloud di AWS e quindi si segue un modello di pagamento pay-per-use. A supporto di questa analisi è stato utilizzato *AWS Pricing Calculator* [28], un calcolatore di prezzi messo a disposizione da AWS che fornisce una stima di costo per diverse configurazioni dei servizi cloud. In Tabella 3.1 ne viene fornita una stima, in cui sono studiate quattro possibilità:

- eseguire le pipeline con Glue;
- eseguire le pipeline con MWAA;
- eseguire i job delle pipeline su Fargate e il core di Airflow su una macchina EC2 accesa e spenta all'occorrenza;
- eseguire i job delle pipeline su Fargate e il core di Airflow su una macchina EC2 sempre accesa

Servizio	**Costo (usd) per ora**
Aws Glue	4.41
AWS MWAA	361.15
AWS Fargate + EC2 t3 medium spot	1.77 + 3.35 = 5.12
AWS Fargate + EC2 t3 medium always running	1.77 + 36.59 = 38.36

Tabella 3.1: Gestione di un database su Amazon EC2 e su Amazon RDS

Questa analisi è stata fatta cercando di fornire quanto più possibile ai servizi le stesse capacità computazionali e di storage. Si noti inoltre che, nella terza e quarta soluzione non è stata presa in considerazione anche l'adozione del database RDS necessario ad Airflow; questo perché si è pensato di utilizzare un'istanza già presente nell'ambiente di AWS.
Il costo proibitivo di MWAA è stato la discriminante che ne ha fatto scartare la possibilità di utilizzo in questo progetto. Infatti il suo prezzo può essere giustificato ed ammortizzato in caso di un elevato numero di workflow da gestire e, dal momento che dopo la fase di refactoring le pipeline sono diventate solamente due, è stato scelto di non considerarlo. Al contrario installare ed eseguire Airflow su una macchina EC2 e i job relativi alle pipeline su Fargate sembra essere una soluzione relativamente economica (in base al tipo di EC2 scelta e al tempo di utilizzo) e che incorpora tutti i benefici introdotti da Apache Airflow.

3.4.2 Scelta del servizio

Dopo averne stimato i prezzi, in questa sottosezione si vuole motivare la scelta che ha portato ad utilizzare Airflow in esecuzione su una macchina EC2.

Anche se l'analisi di Tabella 3.1 sembra prediligere l'utilizzo di Glue, è necessario considerare altri aspetti. Una delle caratteristiche fondamentali di Airflow che non si ritrova in Glue è la sua elevata dinamicità, derivata dalla possibilità di definire le pipeline come codice Python e di conseguenza consentendo la generazione dinamica delle stesse. Inoltre essendo un framework open source è anche altamente estendibile, fornendo la possibilità di creare flussi di lavoro molto complessi. Infine Airflow mette a disposizione un visualizzatore di più semplice utilizzo e più immediato rispetto a quello di Glue, oltre che fornire anche più informazioni riguardo lo stato dei job e della storia di esecuzione delle pipeline.
Tuttavia la scelta di eseguire Airflow su una macchina EC2 delega allo sviluppatore la gestione di tutta l'infrastruttura, e quindi richiede una buona conoscenza del mondo AWS oltre che di un impegno costante nel mantenimento dei servizi.

3.5 Architettura delle pipeline rivisitate

Come già descritto in sottosezione 1.2.3, Airflow è composto da uno scheduler, un esecutore, un database, uno o più worker ed un web server. È possibile configurare Airflow scegliendo il database relazionale, il tipo di esecutore e i worker. La scelta del database è stata obbligata dalla presenza di una istanza RDS già presente all'interno dell'ambiente AWS, che è di tipo PostgreSQL. Per quanto riguarda l'esecutore si è scelto di utilizzare Celery (descritto in sottosezione 1.2.4) e Redis come suo broker, dal momento che risulta l'esecutore più maturo presente all'interno di Airflow.
Inoltre, al fine di mantenere la scalabilità di Airflow, si è scelto di utilizzare l'operatore ECSOperator, descritto in sottosezione 1.2.5.
In Figura 3.6 è mostrata l'architettura AWS delle pipeline rivisitate. Si può notare come

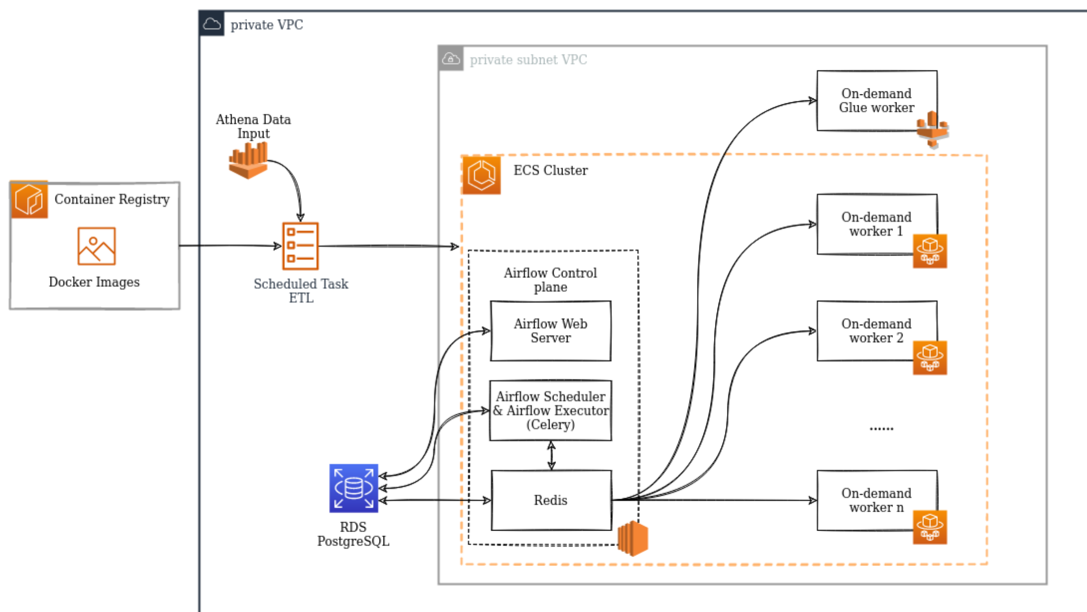


Figura 3.6: Architettura AWS delle pipeline rivisitate

il web server, lo scheduler, Celery e Redis eseguono all'interno della stessa macchina EC2. Nello stesso cluster ECS vengono istanziate macchine Fargate al bisogno, ovvero ogni volta che Celery deve eseguire un job della pipeline crea una macchina Fargate per eseguire il lavoro. Si noti che all'interno della subnet privata è presente anche un worker Glue; questa limitazione deriva dal fatto che lo script di nome *prepare_engagement_chunks* della pipeline annuale è scritto in *pyspark*, l'interfaccia Python di Apache Spark. È stato necessario in quanto ad oggi Glue non permette di eseguire job di tipo Python shell con più di 4 CPU virtuali e con 16 GB di memoria. Dal momento che il numero di dati da elaborare negli script successivi era elevato, è stato indispensabile creare questo script in pyspark per suddividere i dati in 20 batch. Eseguire uno script pyspark significa anche creare un cluster di risorse distribuite, e a seguito di qualche semplice calcolo è risultato più economico e più mantenibile sfruttare il servizio Glue (che istanzia in automatico il cluster distribuito) invece che gestire autonomamente questa architettura. Per questo motivo tale script è rimasto un job Glue, il quale viene lanciato dall'esecutore di Airflow.
Il tutto esegue all'interno di una subnet di una VPC privata, la quale al proprio interno contiene anche un database PostgreSQL utilizzato per memorizzare i log e le informazioni di schedulazione delle pipeline.
Ovviamente sia il DAG che rappresenta le due pipeline sia i job che le compongono sono stati containerizzati, e le due immagini (una per il DAG e una che contiene tutti i job) sono state memorizzate nel Container Registry di AWS. In questo modo, una volta che un worker Fargate deve eseguire un particolare task, sfrutta l'immagine dei job contenuta nel registry e richiama l'esecuzione del corretto script python tramite l'operazione di *command override* propria dei container docker.
La scelta di inserire tutti i componenti di Airflow all'interno di una macchina EC2 e non di una o più macchine Fargate deriva dal fatto che il web server e gli altri componenti richiedono un utilizzo più o meno costante di core e memoria della CPU. Questo non si può dire però per i job delle due ETL: infatti ci sono task che richiedono molta più potenza computazionale di altri, e per questo motivo il servizio serverless Fargate di AWS è perfetto.

Capitolo 4

Implementazione

I capitoli precedenti sono stati utili per descrivere a grandi linee il progetto, il framework Apache Airflow, i servizi utilizzati e gli obiettivi prefissati. Questo capitolo descrive i punti chiave della soluzione implementata, a partire dall'infrastruttura cloud e dei principali componenti che vi partecipano, nella sezione 4.1. Nella sezione 4.2 viene esposta l'implementazione dei DAG associati alle pipeline ETL tramite Airflow.

4.1 Infrastruttura

La seguente sezione vuole presentare la modalità con la quale sono state rilasciate le pipeline ETL sull'infrastruttura cloud di AWS.
I servizi cloud messi a disposizione da Amazon possono essere rilasciati con diverse modalità: tramite interfaccia grafica, da linea di comando o mediante il Kit di sviluppo per il cloud AWS (CDK). In particolare, CDK è un framework di sviluppo software open source rilasciato nel 2019 da Amazon che consente di definire risorse di applicazioni cloud tramite linguaggi di programmazione noti.
L'intera infrastruttura delle pipeline presa in considerazione in questo elaborato è stata rilasciata sfruttando proprio CDK. Questa scelta deriva dai numerosi vantaggi introdotti dall'approccio Infrastructure as Code (IaC):

- Versione e tracciabilità: il codice può essere versionato in un repository di codice sorgente. In questo modo, nel caso in cui fosse necessaria una modifica allo stack dell'infrastruttura, è possibile monitorare e memorizzare i cambiamenti;
- flessibilità e scalabilità: si possono suddividere le risorse dell'infrastruttura in componenti modulari combinabili a seconda dei bisogni, favorendo la disponibilità e scalabilità;
- modifiche rapide: dal momento che l'intera infrastruttura è definita a livello di codice, è possibile creare, aggiornare e distruggere l'infrastruttura eseguendo nuovamente lo script;
- rilasci più veloci: si possono creare ambienti di test e di sviluppo con la stessa configurazione dell'ambiente di produzione eseguendo lo stesso comando con parametri differenti.

Attraverso CDK è possibile preconfigurare e definire stack di risorse cloud grazie ad un insieme di costrutti software. In particolare, AWS CDK suporta nativamente TypeScript, JavaScript, Python, Java, C# e Go. Per mantenere coerenza rispetto alle pipeline ETL, anche il codice di rilascio dell'infrastruttura cloud è stato scritto in Python.
I seguenti sottocapitoli descrivono l'implementazione dei diversi componenti dell'architettura proposta in Figura 3.6, in cui è presente un cluster ECS nel quale sono eseguite diverse macchine Fargate istanziate al bisogno. Di particolare importanza è la macchina virtuale che contiene i componenti principali di Airflow, quali il Web Server, lo schedueler e l' esecutore.

4.1.1 Cluster ECS

Il primo servizio necessario al rilascio delle pipeline ETL sull'architettura di Airflow è il cluster ECS. Infatti non è sufficiente eseguire i container associati ai job delle pipeline, bensì è fondamentale gestire numerose attività tra cui per esempio l'assegnazione delle risorse, il monitoraggio dell'integrità dei container e il routing del traffico. Per questo motivo i container associati alle diverse applicazioni del progetto sono stati rilasciati su un servizio di orchestrazione scalabile quale ECS.
Come evidenziato in Figura 3.6, tutte le risorse orchestrate dal cluster ECS devono essere istanziate all'interno di una subnet VPC privata: nel Listato 4.1 sono mostrate le istruzioni CDK per ottenere i puntatori a queste risorse.

```
from aws_cdk import aws_ec2 as ec2

# Get VPC
vpc = ec2.Vpc.from_lookup(self, 'vpc', vpc_id='###')

# Get private subnet
vpc_subnet_private_a = ec2.SubnetSelection(
    subnet_filters=[ec2.SubnetFilter.by_ids(subnet_ids=['###'])]
    )
```

Listato 4.1: Creazione di puntatori alla VPC e alla rispettiva subnet

Dopodiché, in fase di creazione del cluster ECS è possibile specificare la VPC desiderata (come mostrato nel Listato 4.2).

```
from aws_cdk import aws_ecs as ecs

ecs_cluster = ecs.Cluster(
                self,
                'airflow_cluster',
                cluster_name='airflow_cluster',
                vpc=vpc
        )
```

Listato 4.2: Creazione cluster ECS

4.1.2 Airflow control plane

Come evidenziato in Figura 3.6, i componenti principali di Airflow sono stati inseriti all'interno della stessa macchina EC2. Questa scelta deriva principalmente dalla volontà di ridurre quanto più possibile i costi; attualmente lo scheduler deve gestire solamente due pipeline ETL e, per questo motivo, non avrebbe senso istanziare una macchina virtuale unicamente per questo servizio. Inoltre, mantenere tutti i componenti di Airflow all'interno della stessa macchina virtuale semplifica notevolmente la gestione dell'intera infrastruttura.
Tuttavia, un grande svantaggio di questo approccio architetturale è che non risolve il problema del *single point of failure*: infatti, se la macchina EC2 non fosse disponibile, le pipeline ETL non potrebbero essere schedulate. Per evitare questo singolo punto di vulnerabilità si è fatto uso di Amazon EC2 Auto Scaling, una metodologia di cloud computing grazie alla quale la quantità di risorse computazionali associate ad un servizio scala automaticamente in base al carico di lavoro. Per fare questo è necessario definire un Auto Scaling Group, ovvero una collezione di macchine EC2. Successivamente, specificando il numero minimo e massimo di istanze in ogni gruppo, AWS Auto Scaling è in grado di assicurare che la cardinalità del gruppo rimanga sempre all'interno di questo intervallo. Inoltre, per determinare il numero di risorse necessarie si è istanziato un Capacity Provider, il quale è in grado di comunicare questa informazione direttamente all'Auto Scaling Group.
Dal momento che il control plane di Airflow contiene anche un web server, è necessario associare un Security Group all'istanza EC2 per controllare il traffico di rete in entrata e in uscita.
Il Listato 4.3 mostra l'allocazione di un AutoScaling Group con capacità massima e minima pari a una macchina EC2 ti tipo *t3.medium*, che presenta 2 vCPU e 4 GiB di RAM.

```
from aws_cdk import (
    aws_ec2 as ec2,
    aws_autoscaling as autoscaling,
    aws_ecs as ecs,
)

# Create security group for ec2 instance in ecs
ec2_security_group = ec2.SecurityGroup(self,
                           'airflow-security-group',
                           vpc=vpc,
                           allow_all_outbound=True)

ec2_security_group.add_ingress_rule(
    peer=ec2.Peer.ipv4('10.0.0.0/8'),
    description='inbound tcp',
    connection=ec2.Port.tcp(8080)
)

# Create autoscaling group for ec2 instance in ecs
autoscaling_group = autoscaling.AutoScalingGroup(
    self,
    'autoscaling_group_airflow',
    auto_scaling_group_name='autoscaling_group_airflow',
```

```
    instance_type=ec2.InstanceType("t3.medium"),
    machine_image=ecs.EcsOptimizedImage.amazon_linux2(),
    key_name='airflow-key-pair',
    max_capacity=1,
    min_capacity=1,
    vpc=vpc,
    vpc_subnets=vpc_subnet_private_a,
    role=iam_ec2_role,
    security_group=ec2_security_group,
)

# Create capacity provider for ec2 instance in ecs
capacity_provider = ecs.AsgCapacityProvider(
                        self, "AsgCapacityProvider",
                        auto_scaling_group=autoscaling_group)
ecs_cluster.add_asg_capacity_provider(capacity_provider)
```

Listato 4.3: Creazione Auto Scaling Group e Capacity Provider

Creare un Autoscaling Group con capacità minima e massima pari a uno assicura di avere sempre e solo una istanza EC2 in esecuzione, ma nulla vieta di aumentare il numero massimo nel caso si vogliano gestire più pipeline ETL in contemporanea, assicurando un buon livello di scalabilità.

4.1.3 Task Fargate

Componente fondamentale dell'architettura proposta in Figura 3.6 sono i tasks Fargate lanciati on-demand dall'esecutore di Airflow. Dal momento che non tutti i job delle pipeline ETL richiedono le stesse capacità computazionali, si è pensato di creare tre differenti classi di tasks Fargate con rispettivamente capacità di calcolo limitate, standard ed elevate.
In Tabella 4.1 sono esposte le capacità computazionali dei task relativi alle tre classi, mentre nel Listato 4.4 è mostrato il codice CDK utile a definire tali classi.

Classe	cVPU	Memory	Cost per hour
Low	1	2 GB	0.0494 $
Standard	2	6 GB	0.117 $
High	4	16 GB	0.395 $

Tabella 4.1: Capacità computazionali delle classi di task Fargate

```
from aws_cdk import (
    aws_ecs as ecs,
    aws_ecr as ecr
)

task_repository = ecr.Repository.from_repository_name(
                    self,
```

```
                    'repo',
                    repository_name='etl_airflow_tasks')

task_image = ecs.ContainerImage.from_ecr_repository(
                    repository=task_repository)

port_mapping = ecs.PortMapping(
            container_port=8080,
            protocol=ecs.Protocol.TCP
        )

# Limited capabilities
task_def_low = ecs.FargateTaskDefinition(
                        self,
                        'etl-airflow-task-low',
                        cpu=1024,  # 1 vCPU
                        memory_limit_mib=2048,  # 2GB
                        family='etl_airflow_task_low',
                        execution_role=iam_task_role,
                        task_role=iam_task_role)

task_container_low = task_def_low.add_container(
                        'etl-airflow-task-container-low',
                        image=task_image,
                        logging=ecs.LogDrivers().aws_logs(
                            stream_prefix="etl_airflow_task_low",
                            log_retention=logs.RetentionDays.THREE_DAYS
                        ),
                        port_mappings=[port_mapping])


# Standard capabilities
task_def = ecs.FargateTaskDefinition(
                        self,
                        'etl-airflow-task',
                        cpu=2048,  # 2 vCPU
                        memory_limit_mib=8192,  # 8GB
                        family='etl-airflow-task',
                        execution_role=iam_task_role,
                        task_role=iam_task_role)

task_container = task_def.add_container(
                        'etl-airflow-task-container',
                        image=task_image,
                        logging=ecs.LogDrivers().aws_logs(
                            stream_prefix="etl_airflow_task",
                            log_retention=logs.RetentionDays.THREE_DAYS
                        ),
                        port_mappings=[port_mapping])
```

```
# High capabilities
task_def_high = ecs.FargateTaskDefinition(
                        self,
                        'etl-airflow-task-high',
                        cpu=4096,  # 4 vCPU
                        memory_limit_mib=16384,  # 16GB
                        family='etl-airflow-task-high',
                        execution_role=iam_task_role,
                        task_role=iam_task_role)

task_container_high = task_def_high.add_container(
                        'etl-airflow-task-container-high',
                        image=task_image,
                        logging=ecs.LogDrivers().aws_logs(
                            stream_prefix="etl_airflow_task_high",
                            log_retention=logs.RetentionDays.THREE_DAYS
                        ),
                        port_mappings=[port_mapping])
```

Listato 4.4: Definizione delle classi di tasks Fargate

L'immagine del container associato ai task Fargate è stata versionata e memorizzata su Amazon ECR.

4.2 DAG

Uno dei grandi vantaggi di Apache Airflow è la possibilità di definire workflow arbitrariamente complessi come Directed Acyclic Graph in diversi linguaggi di programmazione. L'implementazione delle pipeline prese in esame in questo elaborato fa uso di due operatori propri di Airflow, quali l'ECS e il Glue Operator. Infatti, avendo preventivamente creato i task Fargate e i job Glue, questi operatori permettono di richiamarne l'esecuzione specificando diversi parametri di lancio, quali per esempio il *command override* per sostituire il comando di esecuzione dei container Docker associati. In tabella 4.2 sono descritti tutti i parametri di ingresso che questo operatore accetta.
Il Listato 4.5 mostra la creazione del task di nome "*elaborate_navigation_data*" della pipeline annuale (vedi Figura 3.5), in cui sono stati passati i flag che abilitano il calcolo dei KPI per l'engagement, il churn sia per il perimetro standard che il perimetro dei qv. Inoltre, dal momento che questa elaborazione richiede elevate capacità di calcolo, il task viene lanciato con la configurazione "task_definition_high", ovvero su una macchina virtuale con 4 vCPU e 16 GB di memoria.

```
CHURN_DATASET = 'create_churn_dataset'
ENGAGEMENT_DATASET = 'create_engagement_dataset'
USAGE_QV_DATASET = 'create_usage_qv_dataset'

TASK_DEFINITION_HIGH = 'etl-airflow-task-high'
CONTAINER_TASK_HIGH = 'etl-airflow-task-container-high'
CLUSTER = 'airflow_cluster'
```

```
NETWORK_CONFIGURATION = {
    "awsvpcConfiguration": {
        "securityGroups": [os.environ.get("SECURITY_GROUP_ID",
           "##")],
        "subnets": [os.environ.get("SUBNET_ID", "##")],
    },
}

default_params = {
    "current_month": True,
    "rebuild": False,
    "reset_tables": False,
    "upload_to_s3_exchange": False,
    CHURN_ELABORATION: True,
    ENGAGEMENT_ELABORATION: True,
    QV_ELABORATION: True,
}

elaborate_navigation_data = ECSOperator(
    task_id='elaborate_navigation_data',
    task_definition=TASK_DEFINITION_HIGH,
    cluster=CLUSTER,
    launch_type=LAUNCH_TYPE,
    region_name=REGION_NAME,
    overrides=change_command_override_python_filename(
        'containerOverrides': [
            {
                'name': CONTAINER_TASK_HIGH,
                'command': ["python",
                   f'trn_elaborate_navigation_data.py',
                json.dumps(default_params)]
            }
        ]
    )
    network_configuration=NETWORK_CONFIGURATION
)
```

Listato 4.5: Definizione del task elaborate_navigation_data

Dopo aver definito tutti i task della pipeline, è necessario esplicitare le dipendenze tra di essi. Per fare questo Airflow raccomanda di utilizzare gli operatori » e «, i quali possono essere concatenati per creare dipendenze multiple. Il Listato 4.6 mostra le dipendenze della pipeline annuale, mostrata anche in Figura 3.5.

```
create_user_tables_editoria >>
create_initial_dataset >>
prepare_engagement_data_chunks >>
elaborate_navigation_data >>
[aggregate_navigation_data, aggregate_navigation_data_qv] >>
prepare_model_input >>
```

```
churn_train_and_inference >>
[churn_aggregate_input_and_prediction_churn,
 churn_aggregate_input_and_prediction_engagement,
    query_ume_tracciato] >>
inference_usage_score >>
create_output >>
create_output_qv
```

Listato 4.6: Definizione delle dipendenze della pipeline annuale

Nome	**Richiesto**	**Descrizione**
task_definition	Si	Nome della definizione del task su ECS
cluster	Si	Nome del cluster su ECS
overrides	Si	Parametro di override utilizzato anche dal client python di AWS boto3
aws_conn_id	No	Credenziali AWS di connessione
region_name	No	Nome della regione AWS
launch_type	Si	Tipo di servizio da istanziare (EC2, Fargate o Esterno)
capacity_provider_strategy	No	Strategia del capacity provider utilizzato per il task
group	No	Nome del gruppo di task associato al task da eseguire
placement_constraints	No	Regola da considerare durante il collocamento del task
placement_strategy	No	Algoritmo per selezionare le istanze dove collocare il task
platform_version	No	Versione della piattaforma su cui eseguire il task
network_configuration	No	Configurazione di rete del task
tags	No	Dizionario di tag
awslogs_group	No	Gruppo CloudWatch in cui memorizzare i log del container associato al task
awslogs_region	No	Regione AWS in cui memorizzare i log di CloudWatch
awslogs_stream_prefix	No	Prefisso utilizzato dai log di CloudWatch
awslogs_fetch_interval	Si	Intervallo che il fetcher del task aspetta prima di inserire i log in CloudWatch. Di default è 30 secondi
quota_retry	No	Determina se e come riprovare l'avvio del task se è fallito
reattach	Si	Se True controlla se il task lanciato precedentemente è ancora in esecuzione. In caso affermativo, l'operatore si collega ad esso invece di avviare un nuovo task. Questo è utile per evitare di rilanciare un nuovo task quando la connessione tra Airflow ed ECS cade mentre il task è ancora in esecuzione. Di default è False
number_logs_exception	Si	Numero di linee del log di CloudWatch da ritornare in caso di eccezione Airflow. Di default è 10
wait_for_completion	Si	Se True aspetta che la creazione del cluster sia completa. Di default è True

Tabella 4.2: Parametri dell'operatore ECS di Airflow

Capitolo 5

Valutazione dei risultati ottenuti

Dopo aver descritto le logiche di progettazione ed implementazione del progetto, in questo capitolo verranno trattati i risultati ottenuti sulla base di diverse metriche di valutazione.

5.1 Code refactoring

La qualità del codice è tipicamente un concetto difficile da definire e da misurare. Essa è influenzata da tre fattori principali: la leggibilità, la manutenibilità e la testabilità. Con leggibilità ci si riferisce alla facilità di comprensione del codice sorgente da parte di un programmatore che non sia l'autore originale, la testabilità identifica il grado di difficoltà con cui un artefatto software può essere testato mentre la manutenibilità misura l'efficienza delle modifiche che possono essere apportate al programma per migliorarlo, correggerlo o modificarlo.
Nel corso degli anni la letteratura ha prodotto diverse metriche di valutazione della qualità del codice sorgente. Oltre alle misure standard, come per esempio il numero di linee di codice, in questo elaborato saranno valutati tre algoritmi di complessità specifici, quali la *Halstead complexity*, la *Cyclomatic complexity* e la *Cognitive complexity*.

5.1.1 Halstead complexity

Nel 1977 Maurice Howard Halsted ha proposto un algoritmo per misurare della complessità del codice, che prende il nome di Halstead complexity [29]. Tale metrica è statica, in quanto nasce dalla premessa che le metriche del software dovrebbero riflettere l'implementazione o l'espressione di algoritmi in diversi linguaggi, ma essere indipendenti dalla loro esecuzione su una specifica piattaforma.
La complessità di Halstead si basa sull'interpretazione del codice sorgente come una sequenza di *token*, dove ogni token viene classificato per essere un operatore o un operando. In particolare vengono definite le seguenti grandezze:

- η_1: numero di operatori distinti;

- η_2: numero di operandi distinti;
- N_1: numero totale di operatori;
- N_2: numero totale di operandi;

A partire da questi valori si possono calcolare diverse metriche:

- vocabolario: $\eta = \eta_1 + \eta_2$
- dimensione: $N = N_1 + N_2$
- volume: $V = N \cdot \log_2 \eta$
- difficoltà: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$. Questa metrica si riferisce alla difficoltà di implementare l'algoritmo
- sforzo: $E = D \cdot V$. Lo sforzo si riferisce al tempo effettivo di codifica
- numero di potenziali bug: $B = \frac{V}{3000}$
- tempo di testing: $T = \frac{E}{k}$, dove di default $k = 18$

5.1.2 Cyclomatic complexity

Sviluppata da Thomas McCabe nel 1976, la cyclomatic complexity è una metrica quantitativa in grado di valutare la complessità e la stabilità del codice sorgente di un software. La logica alla base di questo algoritmo è la seguente: quanto più è elevato il numero di percorsi linearmente indipendenti all'interno di un modulo del programma quanto più il codice è complesso. Infatti i programmi con una complessità ciclomatica inferiore sono più semplici da comprendere e meno rischiosi da modificare rispetto a quelli con complessità elevata.
Dal punto di vista matematico questo tipo di complessità deriva dal *control flow graph* del programma, dove:

- E: numero di archi del grafo
- N: numero di nodi del grafo
- P: numero di componenti connesse

In questo modo si può calcolare la cyclomatic complexity come:

$$M = E - N + 2P$$

In tabella 5.1 sono mostrate le classi di complessità relative a questa metrica.

Valore	Complessità
1 - 10	Semplice
11 - 20	Moderata
21 - 50	Alta
> 50	Molto alta

Tabella 5.1: Classi di complessità nella Cyclomatic complexity

5.1.3 Cognitive Complexity

Nel paper dal nome "*Cognitive Complexity. A new way of measuring understandability*" [30] del 2017, SonarSource ha introdotto la complessità cognitiva come nuova metrica per misurare la comprensibilità del codice sorgente. Analogamente alla complessità ciclomatica, la complessità cognitiva basa il calcolo della complessità del codice sorgente sulle strutture presenti nel control flow graph. SonarSource afferma di aver introdotto questo nuovo tipo di complessità per risolvere alcuni problemi critici relativi alla complessità ciclomatica, i quali non permettono di misurare effettivamente la comprensibilità del codice bensì solo la sua testabilità.
In generale, la complessità cognitiva si basa su tre regole fondamentali:

1. ignora le strutture che consentono di abbreviare più dichiarazioni in una sola;
2. incrementa di uno la complessità per ogni interruzione del flusso lineare del codice;
3. incrementa la complessità quando le strutture di interruzione del flusso sono annidate

5.1.4 Confronto della qualità del codice tra le diverse pipeline ELT

In questa sottosezione si vogliono confrontare le diverse pipeline ETL sulla base di metriche standard, quali per esempio il numero di linee di codice, oltre che a metriche di complessità come Halsted, Cyclomatic e Cognitive complexity (descritte rispettivamente in sezione 5.1.1, 5.1.2 e 5.1.3).
Questo confronto ha lo scopo di quantificare se e quanto il processo di rielaborazione del codice delle pipeline (descritto in sezione 3.2) ha portato beneficio. Dal momento che le due pipeline ETL risultanti dopo la fase di rielaborazione sfruttano molto codice comune, sono stati sommati i valori delle metriche. La pipeline che deriva dalla somma di queste due ETL prende il nome di ETL rielaborata.

Metriche standard

In tabella 5.2 sono mostrate una serie di metriche standard per le ETL prese in analisi in questo elaborato. In particolare, con ETL rielaborata si intende la pipeline dopo il processo di code refactoring e di aggregazione del perimetro standard e quello dei qv; per questo

motivo, è interessante confrontare la somma dei valori delle ETL non rielaborate con i valori della ETL rielaborata.

Metriche	**Etl mensile standard**	**Etl mensile qv**	**Etl annuale standard**	**Etl annuale qv**	**Somma**	**Etl rielaborata**
Linee di codice	4837	3403	15067	3279	26586	9264
Blocchi di istruzioni	2775	1762	6618	1636	12791	4328
Funzioni	233	139	546	124	1042	316
Classi	2	2	8	2	14	8
Numero di files	28	13	48	12	101	64
Linee di commento	396	315	852	294	1857	1193
Percentuale di linee commentate	8.2 %	9.3 %	5.6 %	8.9 %	6,9 %	12.9 %
Blocchi duplicati	37	79	221	81	418	40
Percentuale di blocchi duplicati	1.3 %	4.5 %	3.3 %	4.9 %	3.3 %	0.92 %

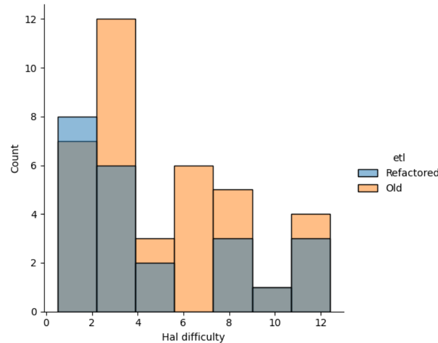
Tabella 5.2

Una prima considerazione importante riguarda il numero di righe di codice: si è passati da 26586 a 9264, riducendo di un fattore pari a 2.87. Questo ha come diretta conseguenza una maggiore manutenibilità e una diminuzione della complessità del codice.
Grazie al processo di code refactoring è stata diminuita anche la percentuale di blocchi duplicati, passando dal 3.3% allo 0.92%.
Anche se è diminuito il numero di linee commentate, ne è quasi raddoppiata la percentuale. Infatti, se la percentuale di commenti nelle ETL non rielaborate è pari al 6.9%, nella pipeline rielaborata si è prossimi al 13%. Anche questo dato conferma la tendenza ad una maggiore manutenibilità del codice, semplificandone la comprensione e l'approccio a futuri sviluppatori.

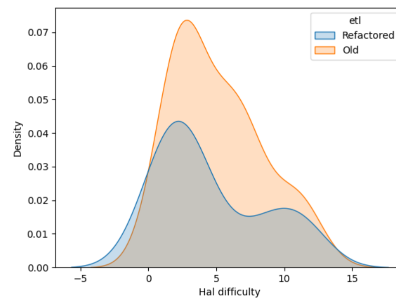
Halstead complexity

In Figura 5.2 sono evidenziate le due metriche più interessanti relative alla complessità di Halstead. In particolare, in Figura 5.1a e Figura 5.1b sono mostrati rispettivamente la distribuzione e il kernel density estimation della difficoltà di Halstead, mentre in Figura 5.1c e 5.1d è stata riportata la distribuzione e il kernel density estimation dei potenziali bug di Halstead. Come ci si poteva aspettare, la pipeline rivisitata ha meno script con difficoltà elevate e il valore massimo che assume la funzione di densità corrisponde a valori di difficoltà uguali a 2.17. Al contrario, la difficoltà più probabile secondo il KDE della pipeline non rivisitata è pari a 2.86.

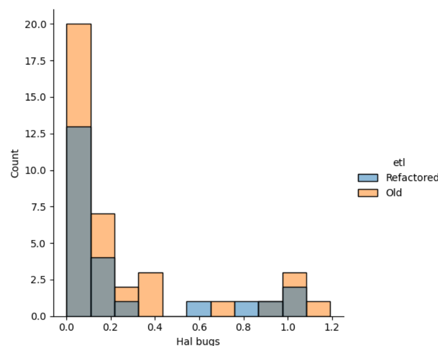
La diminuzione del numero di script durante la fase di rielaborazione ha permesso di ridurre di conseguenza anche il numero di script a difficoltà di Halstead più elevate. Un'ulteriore osservazione può essere fatta a partire dalla Figura 5.2a, la quale mostra la distribuzione di difficoltà di Halstead dei dieci script a difficoltà più elevata per le due



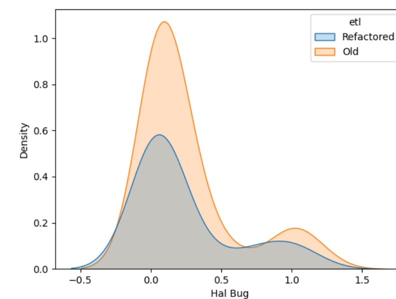
(a) Distribuzione del valore di difficoltà di Halstead delle pipeline ETL



(b) Kernel density estimation del valore difficoltà di Halstead delle pipeline ETL



(c) Distribuzione del valore di potenziali bug di Halstead delle pipeline ETL



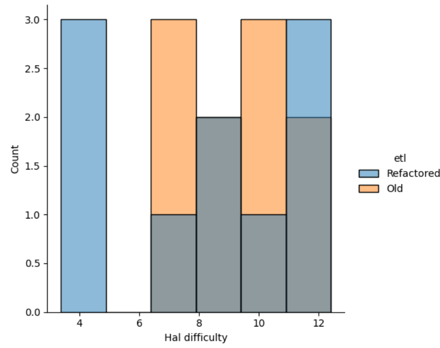
(d) Kernel density estimation del valore di potenziali bug di Halstead delle pipeline ETL

Figura 5.1: Halstead complexity delle pipeline ETL

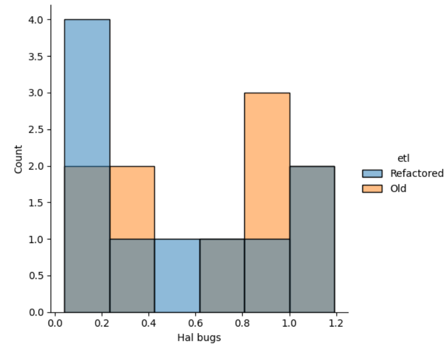
ETL. Analogamente, la Figura 5.2b evidenzia la distribuzione del numero di probabili bug di Halstead per i dieci script aventi tale metrica più elevata. In particolare, non si notano sostanziali differenze per la metrica di difficoltà tra le due pipeline, mentre il numero di potenziali bug è leggermente diminuito.

Cyclomatic complexity

In Figura 5.3 è confrontata la complessità ciclomatica della pipeline rivisitata con la pipeline non rivisitata. In particolare, in Figura 5.3a e Figura 5.3b sono mostrati rispettivamente la distribuzione e il kernel density estimation della complessità ciclomatica, mentre in Figura 5.3c e 5.3d sono state riportate le stesse valutazioni solamente per i dieci script di ogni pipeline aventi valore maggiore.
Da questi grafici si può notare che la complessità ciclomatica della pipeline rivisitata è mediamente minore rispetto alla pipeline non rivisitata, soprattutto per gli script più complessi.

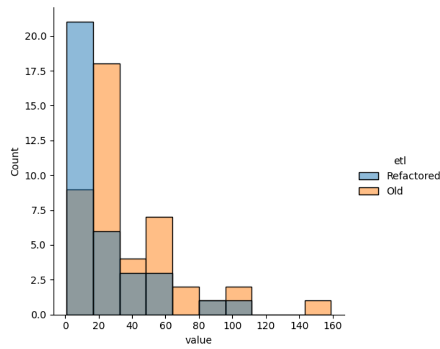


(a) Distribuzione del valore di difficoltà di Halstead dei dieci script delle pipeline ETL

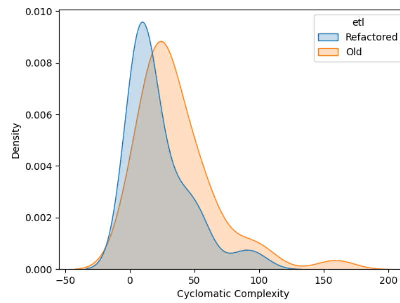


(b) Distribuzione del valore di potenziali bug di Halstead dei dieci script delle pipeline ETL

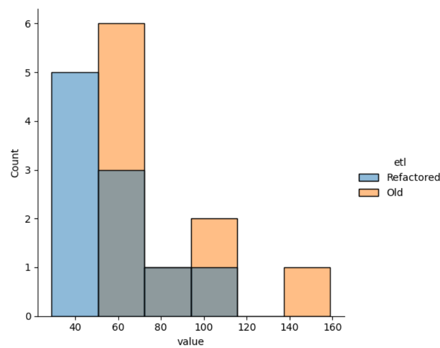
Figura 5.2: Halstead complexity dei primi 10 script delle pipeline ETL



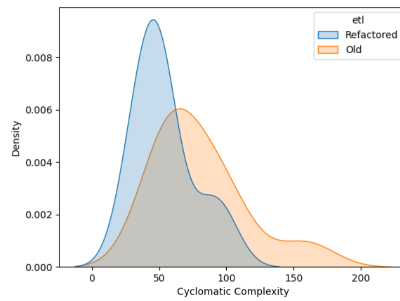
(a) Distribuzione della complessità ciclomatica delle pipeline ETL



(b) Kernel density estimation della complessità ciclomatica delle pipeline ETL



(c) Distribuzione della complessità ciclomatica dei dieci script più complessi delle pipeline ETL



(d) Kernel density estimation della complessità ciclomatica dei dieci script più complessi delle pipeline ETL

Figura 5.3: Complessità ciclomatica delle pipeline ETL

Sulla base delle classi di complessità ciclomatica descritte in tabella 5.1, in Figura 5.4 è stata graficata la distribuzione delle classi delle pipeline ETL. Si può notare una netta distinzione tra le due pipeline; infatti quella rivisitata mappa la maggior parte della complessità nella classe semplice, a differenza della ETL non rivisitata in cui sono presenti molti job a complessità alta.

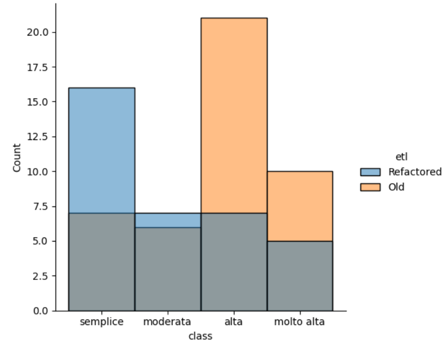
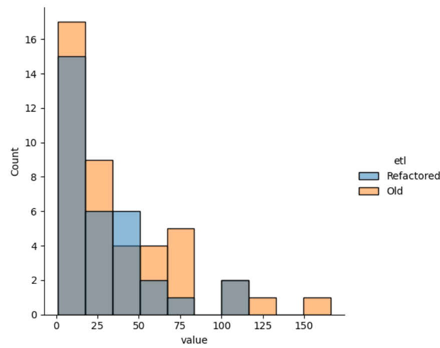


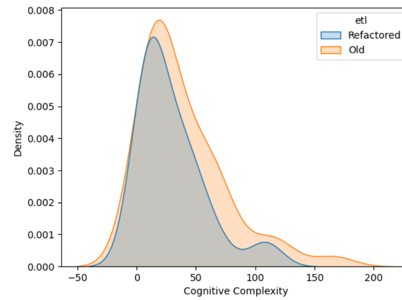
Figura 5.4: Distribuzione delle classi di complessità ciclomatica delle pipeline ETL

Cognitive complexity

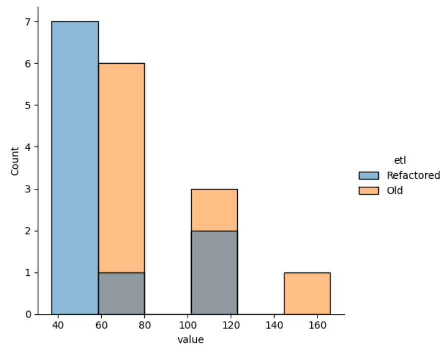
I valori di complessità cognitiva raggiunti dalla pipeline rivisitata confermano le considerazioni sostenute per la complessità ciclomatica. In Figura 5.5 sono evidenziate le misurazioni di complessità cognitiva delle due pipeline.



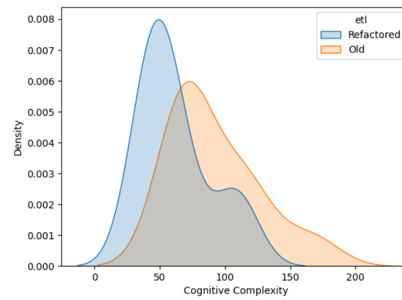
(a) Distribuzione della complessità cognitiva delle pipeline ETL



(b) Kernel density estimation della complessità cognitiva delle pipeline ETL



(c) Distribuzione della complessità cognitiva dei dieci script più complessi delle pipeline ETL



(d) Kernel density estimation della complessità cognitiva dei dieci script più complessi delle pipeline ETL

Figura 5.5: Complessità cognitiva delle pipeline ETL

5.2 Orchestrazione tramite Apache Airflow

Come descritto in Capitolo 3, parte centrale di questo progetto è ridistribuire delle pipeline ETL attualmente in esecuzione su AWS Glue su una istanza di Apache Airflow. In particolare, l'architettura AWS su cui esegue Airflow è descritta accuratamente in sezione 3.5; l'idea di fondo è di istanziare macchine Fargate al bisogno, ovvero ogni volta in cui deve essere eseguito un job della pipeline viene avviata una macchina Fargate per svolgere il lavoro. In Figura 3.6 ne è mostrata l'architettura.
La seguente sezione si pone come obiettivo quello di confrontare la soluzione trattata in questo elaborato con il contesto di esecuzione attuale delle pipeline (AWS Glue) dal punto di vista del costo computazionale.

5.2.1 Costo computazionale

Per analizzare il costo computazionale sono stati monitorati i tempi di esecuzione dei job delle pipeline etl in entrambe le configurazioni cloud: sia nel contesto di AWS Glue sia con Apache Airflow.
Le capacità computazionali dei servizi presi in analisi in questi due contesti non sono liberamente configurabili ma devono essere scelte a partire da una serie di configurazioni di default. In Tabella 5.3 sono mostrate le capacità computazionali dei job delle pipeline ETL per AWS Fargate e AWS Glue, con anche il loro costo associato.

AWS Fargate			**AWS Glue**		
vCPU	Memoria	Costo all'ora	vCPU	Memoria	Costo all'ora
1	2 GB	0.0494 $	0.25	1 GB	0.0275 $
2	6 GB	0.117 $	4	16 GB	0.44 $
4	16 GB	0.395 $	8	32 GB	0.88 $

Tabella 5.3: Configurazioni di capacità computazionale dei job delle pipeline ETL

ETL mensile

L'ETL mensile viene schedulata una volta a settimana e ha il compito di estrarre, trasformare e caricare una quantità limitata di dati. Per questo motivo, nell'architettura proposta in questo elaborato, i job di elaborazione sono eseguiti da Fargate con macchine virtuali molto piccole, tipicamente con 1 vCPU e 2 GB di memoria. Questo ha come conseguenza una notevole diminuzione dei costi, pari circa a quattro volte il costo di esecuzione della stessa pipeline ETL su AWS Glue. In tabella 5.4 sono evidenziati i tempi di esecuzione e le capacità computazionali associate ai job della pipeline mensile nelle due architetture prese in analisi in questo elaborato. Come si può notare, il tempo di esecuzione della pipeline è fortemente influenzato dal job di nome *create_user_table*, il quale è responsabile di estrarre i dati dal database del cliente per poi inserirli in AWS Athena. Proprio per questa

ragione tale job dipende fortemente dalle prestazioni del database del cliente, sul quale non è possibile intervenire.

	Apache Airflow con AWS Fargate			AWS Glue		
Job	vCPU	Memoria	Tempo di esecuzione	vCPU	Memoria	Tempo di esecuzione
Create user table	2	8 GB	1853 sec	4	16 GB	1729 sec
Create initial dataset	1	2 GB	145 sec	4	16 GB	97 sec
Calculate summable KPI	1	2 GB	167 sec	4	16 GB	186 sec
Calculate non summable KPI	1	2 GB	166 sec	4	16 GB	227 sec
Apply inference	1	2 GB	220 sec	4	16 GB	197 sec
Create output table	1	2 GB	353 sec	4	16 GB	298 sec
Checkout consistency	1	2 GB	91 sec	4	16 GB	40 sec
			Costo totale: 0.07564 $			Costo totale: 0.3390 $

Tabella 5.4: Confronto dei tempi e costi di esecuzione della pipeline mensile in esecuzione su Fargate e Glue

ETL annuale

Se l'ETL mensile risulta poco ottimizzabile, l'ETL annuale viene schedulata una volta al mese elaborando una grande mole di dati; su questa si ha maggiore possibilità di miglioramento. Anche in questo caso sono stati analizzati i tempi di esecuzione di ogni job dell'ETL sulla base delle capacità computazionali fornite.
A differenza della pipeline mensile, quella annuale contiene un job di nome *prepare_data_chunks* implementato in pyspark, l'interfaccia Python di Apache Spark. L'utilizzo di un motore di elaborazione dati distribuito ha come scopo quello di suddividere i dati in 20 batch differenti, in modo da poterli elaborare nei job successivi della pipeline. Come descritto nel capitolo 3.5, a seguito di una breve analisi, si è ritenuto più opportuno fare ancora uso di tale job Glue anche per l'ETL rielaborata e in esecuzione su Airflow.
In tabella 5.5 sono evidenziati i tempi di esecuzione e le capacità computazionali associate ai job della pipeline annuale nelle due architetture prese in analisi in questo elaborato. Una prima osservazione riguarda i tempi di esecuzione: anche se la capacità computazionale è stata dimezzata, i tempi di esecuzione non sono aumentati bensì sono mediamente diminuiti. Questo fenomeno è dovuto a due fattori principali: dal processo di code refactoring (descritto in sezione 3.2) e dalle troppe capacità computazionali assegnate ai job della pipeline di Glue. Infatti, una grande limitazione di Glue è la possibilità di scegliere le capacità computazionali a partire da poche configurazioni diverse; ad oggi, per i processi di tipo *python shell*, sono disponibili due configurazioni: 4 vCPU e 16 GB di memoria o 0.25 vCPU e 1 GB di memoria.

Come detto precedentemente, il job *prepare_data_chunks* utilizza pyspark ed esegue su Glue in entrambe le pipeline. Dal momento che ha come unico scopo la suddivisione dei

	Apache Airflow con AWS Fargate			AWS Glue		
Job	vCPU	Memoria	Tempo di esecuzione	vCPU	Memoria	Tempo di esecuzione
Create user table	2	8 GB	847 sec	4	16 GB	831 sec
Create initial dataset	2	8 GB	883 sec	4	16 GB	806 sec
Prepare data chunks	4	16 GB	1378 sec	4	16 GB	1234 sec
Calculate summable KPI	2	8 GB	1189 sec	4	16 GB	1703 sec
Calculate non summable KPI	2	8 GB	244 sec	4	16 GB	186 sec
Prepare model input	2	8 GB	119 sec	4	16 GB	71 sec
Apply inference	2	8 GB	86 sec	4	16 GB	126 sec
Aggregate input and prediction	2	8 GB	440 sec	4	16 GB	482 sec
Inference usage score	2	8 GB	208 sec	4	16 GB	351 sec
Create output	2	8 GB	462 sec	4	16 GB	386 sec
			Costo totale: 5.197 $			Costo totale: 5.129 $

Tabella 5.5: Confronto dei tempi e costi di esecuzione della pipeline annuale in esecuzione su Fargate e Glue

dati in batch indipendenti e il suo codice sorgente risulta già ottimizzato, non è possibile diminuirne il tempo di esecuzione. Allora, per ridurre il costo della pipeline in esecuzione su Airflow ci si è concentrati sull'ottimizzazione del job *calculate_summable_kpi*, che è il secondo job più oneroso. Tale script ha come obiettivo quello di elaborare i 20 chunks di dati generati dal job precedente per calcolare dei KPI sommabili. Allo stato dell'arte del progetto, questo job trasforma un solo chunk di dati alla volta anche se questi ultimi risultano indipendenti l'uni dagli altri. Preso atto di questo, si è pensato di renderlo parallelo; l'idea è di elaborare più chunks nello stesso momento partizionando tale carico a più processi differenti. Per fare questo si è utilizzato il pacchetto *multiprocessing* di Python [31], il quale fornisce un'astrazione per creare processi figli e controllarne l'esecuzione. In tabella 5.6 sono evidenziati i tempi di esecuzione del job *calculate_summable_kpi* parallelizzato con due, tre e quattro processi.

Numero di processi	**vCPU**	**Memoria**	**Tempo di esecuzione**
1	2	8 GB	1889 sec
2	2	8 GB	1008 sec
3	2	8 GB	896 sec
4	2	8 GB	924 sec

Tabella 5.6: Tempi di esecuzione del job *calculate_summable_kpi* parallelizzato

Quindi, a seguito di questa ottimizzazione, il costo della pipeline in esecuzione su Airflow passa da 5.197$ a 5.187$. Il motivo per cui la differenza di costo tra queste due versioni della pipeline è quasi impercettibile deriva dalle metriche di costo utilizzate da AWS. Infatti, il job spark *prepare_data_chunks* esegue su un cluster di 30 workers, e il suo costo deriva dalla seguente formula:

$$glue_spark_cost = n_{workers} \cdot 0.44 \cdot DPU$$

Dove una DPU corrisponde ad una macchina con 4 vCPUs e 16 GB di memoria.
In Figura 5.6 mostra la distruzione del costo dei job della pipeline in esecuzione su Airflow. Si nota bene che la maggior parte dei job hanno un costo molto vicino allo zero, ad eccezione di un solo script (che è proprio *prepare_data_chunks*): questo spiega il motivo per il quale, anche se è stato dimezzato il tempo di esecuzione del secondo job più lento, il costo della pipeline è diminuito molto poco.

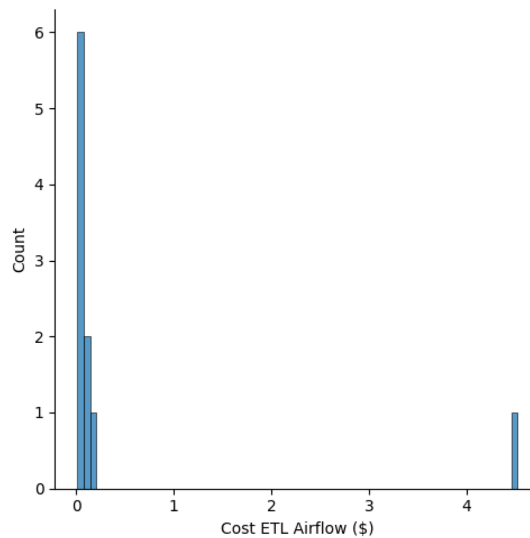


Figura 5.6: Distribuzione del costo dei job della pipeline in esecuzione su Airflow

Quando conviene Airflow?

In questa sottosezione si vuole determinare il numero di pipeline dopo il quale si ha un risparmio monetario utilizzando l'architettura di Airflow su un cluster ECS implementata in questo elaborato.
In particolare, l'architettura proposta in Figura 3.6 fa uso di una macchina EC2 (per il web server e il control plane di Airflow), un database RDS per mantenere lo storico delle schedulazioni e delle macchine Fargate istanziate al bisogno. In tabella 5.7 sono evidenziati i costi dei servizi che sono indipendenti dal numero di pipeline da schedulare. Per ammortizzare tali spese è necessario che il costo delle pipeline eseguite su Airflow sia minore delle stesse eseguite su Glue. Questo è possibile dal momento che Fargate mette a disposizione molte più configurazioni di potenza computazionale tra cui scegliere, mentre Glue solamente due.

Di conseguenza, una volta determinate le capacità computazionali che i job delle pipeline richiedono, l'utente può sceglierne la configurazione più economica.

Servizio	Istanza	Costo all'ora
AWS RDS	1 vCPU e 1 GB di RAM	0.076 $
AWS EC2	4 vCPU e 16 GB di RAM	0.1664 $

Tabella 5.7: Costo dei servizi dell'architettura di Airflow

A partire da questa ipotesi, in Figura 5.7 è stato confrontato il costo dei servizi sfruttati da Airflow e Glue all'aumentare del numero di pipeline ETL di tipo mensile eseguite, dove il costo delle singole macchine Fargate è già stato descritto in tabella 5.4. Il numero di pipeline dopo il quale conviene dal punto di vista economico l'utilizzo di Airflow è pari a 220.

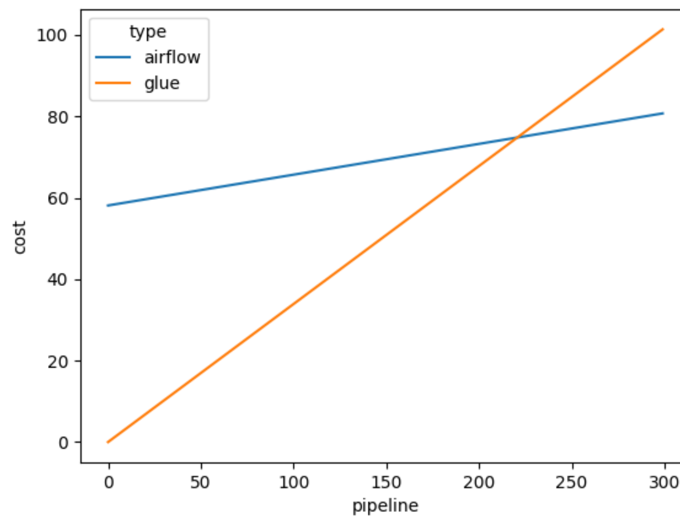


Figura 5.7: Costo dell'architettura di Airflow e di Glue all'aumentare del numero di pipeline ETL mensili

Relativamente alla pipeline ETL annuale, è impossibile trovare il numero di pipeline dopo il quale si ha un risparmio economico. Questo perchè una singola schedulazione della pipeline annuale su Airflow ha un costo apri a 5.187$, contro i 5.129$ di Glue.

Amazon MWAA

MWAA, acronimo di Amazon Managed Workflows for Apache Airflow, è un servizio cloud di AWS che fornisce l'architettura e la logica di esecuzione propria di Apache Airflow. A seguito di una breve analisi, descritta in sezione 3.4.1, si è scelto di non adottare questo servizio bensì istanziare Airflow su un cluster ECS. In questa sottosezione si vogliono

confrontare i costi ad alto livello tra l'architettura descritta in questo elaborato e MWAA. In particolare, supponendo di avere:

- n pipeline con 20 job ciascuna;
- ogni job con tempo di esecuzione pari a 5 minuti;
- un ambiente MWAA di medie dimensioni e 20 worker con 2 vCPUs e 4 GB di RAM;
- l'architettura presa in esame in questo elaborato con i servizi descritti in tabella 5.7 e macchine Fargate al bisogno con 2 vCPUs e 6 GB di RAM

Allora, Figura 5.8 dimostra che sono necessarie più di 3000 pipeline per far si che il servizio MWAA sia più economico della soluzione proposta in questo elaborato. Questo deriva principalmente dal costo molto elevato dell'ambiente di MWAA, che impatta oltre il 98% del totale; il restante 2% rappresenta il costo di esecuzione vera e propria delle pipeline.

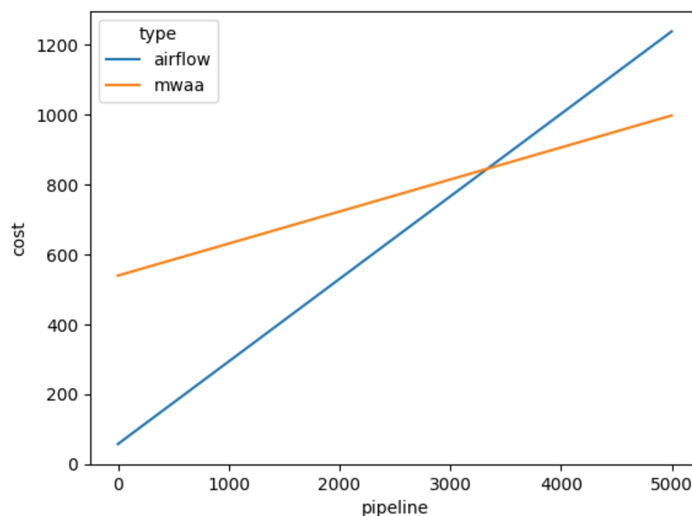


Figura 5.8: Costo dell'architettura di Airflow e di MWAA all'aumentare del numero di pipeline

5.3 Metriche non quantitative

Quelle descritte nelle sezioni precedenti erano metriche quantitative. Tuttavia, l'adozione di Airflow ha portato una serie di benefici difficilmente descrivibili mediante numeri. In particolare, l'obiettivo di questa sezione è proprio quello di elencare tutti i pregi che la soluzione trattata in questo elaborato ha portato.

Benefici del code refactoring

Come già dimostrato in sezione 5.1, il code refactoring ha permesso di diminuire mediamente la complessità del codice sorgente. Oltre a questo, durante la fase di rielaborazione delle pipeline, sono stati raggiunti i seguenti risultati:

- creazione di un ambiente di sviluppo e uno di produzione;
- aggregazione delle pipeline su due livelli temporali. In questo modo si è passati da gestire quattro pipeline differenti a due;
- creazione di asset comuni alle due pipeline.

Benefici di Apache Airflow

Adottare Apache Airflow come piattaforma di schedulazione e monitoraggio si è rilevata una buona scelta, soprattutto nel caso di gestione di un numero elevato di pipeline. Questo perchè:

- è altamente dinamico: i workflow sono definiti tramite DAG (Directed Acyclic Graph) in python. Questo permette di utilizzare i costrutti propri di python per generare dinamicamente task opzionali in base a determinate condizioni;
- è altamente estendibile: è possibile connettere Airflow a qualsiasi altro tipo di sistema per create flussi di lavoro più complessi;
- è un progetto open source;
- permette il versionamento dei dag. Questo porta una serie di vantaggi, quali per esempio la tracciabilità, supporto CI/CD e molti altri;
- a differenza di Glue, fornisce un'interfaccia grafica ricca di informazione e molto chiara;
- è altamente scalabile, allo stesso modo di Glue;
- è agnostico rispetto gli strumenti: questo permette di scegliere i componenti che si ritengono più opportuni all'interno dell'architettura;
- in Glue era necessario definire un trigger per ogni job di una pipeline. In Airflow questo non è richiesto, dal momento che le pipeline sono definite tramite dei grafi;
- l'architettura di Airflow adottata in questo elaborato consente di aumentare le capacità computazionali ai worker scegliendo tra tante configurazioni possibili. Infatti, sulla base del tipo di job, è possibile istanziare esecutori *memory oriented* o *CPU oriented*: Glue non offre questa opportunità;
- rispetto a Glue, in Airflow è più semplice l'operazione di debugging;
- monitoraggio delle schedulazioni più granulare rispetto a Glue.

Benefici Infrastructure as code

Tutta l'architettura descritta in questo elaborato è stata istanziata su AWS, solo dopo essere stata definita a codice. Questo ha portato i seguenti benefici:

- versionamento dell'architettura;
- configurazione dell'ambiente più rapida;
- maggiore trasparenza;
- automatizzazione dei processi manuali;
- semplifica la creazione di ambienti di test e di produzione

Conclusioni

Si conclude il presente elaborato con alcune considerazioni derivate delle osservazioni maturate durante la fase di stesura di questa tesi.

In prima battuta, grazie allo studio di numerose fonti e articoli scientifici, è stato analizzato Apache Airflow. Successivamente nel secondo capitolo, dopo una breve introduzione al cloud computing, sono stati descritti i servizi cloud di AWS utili alla realizzazione di pipeline ETL. Questa parte di elaborato contiene le fondamenta su cui è stato implementato l'intero progetto.

Dopo una fase di analisi delle pipeline in esecuzione su Glue allo stato dell'arte, è stato portato a termine un'importante operazione di code refactoring che ha compreso la creazione di asset comuni, una fase di aggregazione delle pipeline ed è stato creato un ambiente di sviluppo e di produzione. Dopodiché, è stata progettata e descritta tramite IAC l'architettura cloud a supporto di Airflow.

Infine, con l'obiettivo di valutare la bontà della soluzione proposta in questo elaborato, sono state descritte e misurate una serie di metriche di qualità del codice sorgente delle pipeline rivisitate. Oltre a questo, sono stati valutati e confrontati i costi computazionali della soluzione proposta con le pipeline ETL in esecuzione su Glue e MWAA.

La soluzione proposta in questo elaborato rappresenta un'alternativa molto più economica rispetto al servizio MWAA di AWS. Inoltre, se comparata con la soluzione Glue di partenza, presenta molti aspetti derivanti dall'introduzione di Airflow.

Bibliografia

[1] *Ammagamma website*. URL: https://ammagamma.com/.

[2] Alkis Simitsis et al. «QoX-driven ETL design: Reducing the cost of ETL consulting engagements». In: vol. 29. Giu. 2009.

[3] *Apache airflow home page*. URL: https://airflow.apache.org/.

[4] *Apache Software Foundation home page*. URL: https://www.apache.org/.

[5] *The Apache Software Foundation Announces Apache Airflow as a Top-Level Project*. URL: https://news.apache.org/foundation/entry/the-apache-software-foundation-announces44.

[6] *Airflow Rest API*. URL: https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html.

[7] *Jinja Documentation*. URL: https://jinja.palletsprojects.com/en/3.1.x/.

[8] Airflow Documentation. *Architecture Overview*. URL: https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html.

[9] Airflow Documentation. *Sequential Executor*. URL: https://airflow.apache.org/docs/apache-airflow/stable/executor/sequential.html.

[10] Kubernetes Documentation. *Cos'è Kubernetes?* URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/.

[11] Airflow Documentation. *Kubernetes Executor*. URL: https://airflow.apache.org/docs/apache-airflow/stable/executor/kubernetes.html.

[12] Celery Documentation. *Introduction to Celery*. URL: https://docs.celeryq.dev/en/stable/getting-started/introduction.html.

[13] Airflow Documentation. *Celery Executor*. URL: https://airflow.apache.org/docs/apache-airflow/stable/executor/celery.html.

[14] Airflow Documentation. *UI / Screenshots*. URL: https://airflow.apache.org/docs/apache-airflow/stable/ui.html.

[15] Peter Mell e Timothy Grance. «The NIST Definition of Cloud Computing». In: set. 2011.

[16] Simson L. Garfinkel. «Architects of the Information Society». In: MIT Press, 1999.

[17] Fang Liu et al. «NIST Cloud Computing Reference Architecture». In: set. 2011.

[18] Raj Bala et al. *Magic Quadrant for Cloud Infrastructure and Platform Services*. 2021. URL: https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb.

[19] Microsoft. *Professional Developers Conference*. 2008. URL: https://news.microsoft.com/speeches/professional-developers-conference-2008-day-1-keynote-ray-ozzie-amitabh-srivastava-bob-muglia-dave-thompson/.

[20] Efthymios Constantinides e Stefan J Fountain. *Web 2.0: Conceptual foundations and marketing issues*. 2008. URL: https://doi.org/10.1057/palgrave.dddmp.4350098.

[21] AWS Documentation. *What is aws?* URL: https://aws.amazon.com/it/what-is-aws/?nc1=f_cc.

[22] AWS Documentation. *Amazon VPC princing*. URL: https://aws.amazon.com/vpc/pricing/.

[23] *CVE website*. URL: https://cve.mitre.org/.

[24] AWS Documentation. *What is Amazon Relational Database Service (Amazon RDS)?* URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html.

[25] AWS Documentation. *Amazon S3 Storage Classes*. URL: https://aws.amazon.com/s3/storage-classes/?nc=sn&loc=3.

[26] AWS Documentation. *AWS Glue concepts*. URL: https://docs.aws.amazon.com/glue/latest/dg/components-key-concepts.html.

[27] AWS Documentation. *What Is Amazon Managed Workflows for Apache Airflow (MWAA)?* URL: https://docs.aws.amazon.com/mwaa/latest/userguide/what-is-mwaa.html.

[28] *AWS Pricing Calculator mainpage*. URL: https://calculator.aws/#/.

[29] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[30] Ann Campbell. *Cognitive Complexity. A new way of measuring understandability*. 2017. URL: https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[31] *Documentazione pacchetto Python multiprocessing*. URL: https://docs.python.org/3/library/multiprocessing.html.