Politecnico di Torino
Laurea magistrale in
Ingegneria Informatica (Computer Engineering) - Cybersecurity
A.A. 2023-2024
Aprile 2024

Sviluppo di automazioni per discovery e gestione di risorse attraverso Infrastructure as Code in ambienti multicloud

Relatore
Prof. Riccardo Sisto

Tutor aziendale Liquid Reply
Dott. Danilo Abbaldo
Dott. Matteo D’Amore
Dott. Davide Sarais

Candidato
Lorenzo Buompane

Abstract

Il cloud computing è una tecnologia sempre più utilizzata tra le aziende poichè ha permesso di far affidamento a servizi remoti eliminando la necessità di costruire e mantenere la propria infrastruttura IT locale. Con l'evoluzione di questa tecnologia e il nascere di nuove esigenze aziendali, molte organizzazioni sono passate ad un approccio multicloud.

Il multicloud è l'utilizzo di più Cloud Service Provider per la creazione della propria infrastruttura IT, fornendo all'azienda una maggiore flessibilità, resilienza, riduzione dei costi e indipendenza da cloud provider specifici. L'utilizzo di risorse in provider differenti rende però più complessa la gestione tra esse.

L'adozione dell'approccio Infrastructure as Code (IaC) ha un ruolo fondamentale in questo contesto. IaC è una pratica appartenente alla filosofia DevOps, che abilita la gestione e la configurazione delle risorse cloud, trattando l'infrastruttura come se fosse codice, anche in ambienti multicloud.

Il progetto di tesi svolto si propone di affrontare il problema di gestione delle risorse in cloud service provider diversi, partendo da uno scenario in cui un'azienda possiede già un'infrastruttura basata sul cloud ma non gestisce le risorse con l'approccio Infrastructure as Code. Lo sviluppo comprende una prima fase di discovery delle risorse, seguita da un'importazione di esse in un progetto Pulumi, strumento che utilizza IaC, permettendone così la gestione tramite codice.

Gli script per l'individuazione delle risorse, i meccanismi di import e la definizione di blueprint per la distribuzione dell'infrastruttura sono stati sviluppati con l'ausilio di SDK per Python forniti da Microsoft Azure e Amazon Web Services (AWS), Cloud Service Provider presi in considerazione durante il lavoro di tesi.

Il progetto realizzato mira alla gestione ottimizzata ed efficace delle risorse e alla riduzione di tempo per la creazione e la configurazione

di esse, diminuendo inoltre l'errore umano. Si sono valutati casi d'uso diversi per soddisfare esigenze differenti: l'importazione delle risorse in un progetto Pulumi per la gestione tramite IaC, la personalizzazione del file IaC per la creazione di una nuova infrastruttura (blueprint), la preparazione di un file IaC pronto per essere eseguito in una regione differente in caso di disaster recovery. Ogni scenario è stato automatizzato tramite pipeline CI/CD in Jenkins che scaricano gli script e caricano i file generati in opportune repository Git.

I risultati ottenuti in questo progetto portano ad un aumento significativo delle prestazioni. Sono stati confrontati i tempi di esecuzione necessari per iac-izzare N risorse con la soluzione sviluppata rispetto a eseguire l'attività in modo manuale. È stato automatizzato il processo di acquisizione dei dati necessari all'import delle risorse, che richiede pochi secondi, rispetto a effettuare l'accesso al portale per il recupero dei dati e la creazione del file necessario, il quale necessita di un tempo stimato di almeno qualche minuto a risorsa. Inoltre, risultati simili sono visibili nel momento in cui il blueprint IaC creato è utilizzato per il deploy di una nuova infrastruttura, evitando la configurazione manuale di ogni risorsa.

Altri vantaggi garantiti sono controllo di versione, elasticità, scalabilità, automazione, riduzione dell'errore umano.

Inoltre, questo progetto permette futuri studi e miglioramenti per rendere più efficiente la gestione delle risorse in contesti aziendali specifici, aggiornando le automazioni già create o aggiungendone altre.

Indice

Lista delle figure 10

Introduzione 12

Concetti base 15

1.1 Cloud computing 15

1.1.1 Tipologie di cloud computing 16

1.1.2 Tipologie di servizi di cloud computing 16

1.1.3 Cloud computing datacenter 18

1.1.4 Principali cloud service provider 20

1.2 Multicloud 21

1.3 DevOps 23

1.3.1 Pratiche DevOps 24

1.4 Infrastructure-as-Code 26

1.4.1 Pulumi 27

1.5 Pipeline CI/CD 30

1.5.1 Jenkins 31

Progetto 34

2.1 Preparazione dell'ambiente e primi test 35

2.2 Discovery 37

2.2.1. Azure 38

2.2.2 Amazon Web Services (AWS) 40

2.3 Import 42

2.4 Parsing 45

2.4.1 Microsoft Azure 46

2.4.2 Amazon Web Services (AWS) 50

2.5 Customization 56
2.5.1 Microsoft Azure 56
2.5.2 Amazon Web Services (AWS) 56
2.6 Automation 58
Use cases 60
3.1 Requisiti 60
3.2 Importazione delle risorse in un progetto Pulumi per la gestione tramite IaC 62
3.2.1 Pipeline 62
3.2.2 Stage 1: Parameter check 63
3.2.3 Stage 2: Clone 64
3.2.4 Stage 3: Azure import 64
3.2.5 Stage 4: AWS import 66
3.2.6 Stage 5: Upload 67
3.2.7 Ultime operazioni 67
3.3 Personalizzazione del file IaC per la creazione di una nuova infrastruttura: blueprint 68
3.3.1 Pipeline 68
3.3.2 Stage 1: Parameter check 68
3.3.3 Stage 2: Clone 68
3.3.4 Stage 3: Azure import 69
3.3.5 Stage 4: AWS import 70
3.3.6 Stage 5: Upload 71
3.3.7 Ultime operazioni 71
3.4 Preparazione di un file IaC pronto per essere eseguito in una regione differente in caso di Disaster Recovery 73

3.4.1 Pipeline 73
3.4.2 Stage 1: Parameter check 74
3.4.3 Stage 2: Clone 74
3.4.4 Stage 3: Azure recovery 75
3.4.5 Stage 4: AWS recovery 75
3.4.6 Stage 5: Azure deploy 76
3.4.7 Stage 6: AWS deploy 77
3.4.8 Stage 7: Upload 77
3.4.9 Ultime operazioni 78
Validazione 80
4.1 Tempi di esecuzione script 80
4.2 Tempi di esecuzione pipeline 85
4.3 Stime e confronti con operazioni manuali 89
4.4 Ulteriori vantaggi 91
Limitazioni e sviluppi futuri 94
Conclusioni 97
Bibliografia e Sitografia 100

Lista delle figure

Figura 1 - Servizi di Cloud Computing 17
Figura 2 - Regioni e zone 18
Figura 3 - Cloud Service Provider: percentuali di utilizzo 20
Figura 4 - DevOps 23
Figura 5 - Infrastructure as Code 26
Figura 6 - Struttura Pulumi 28
Figura 7 - Traduzione tipo risorsa 39
Figura 8 - Comunicazione con repository 61
Figura 9 - Parametri input primo scenario 63
Figura 10 - Parametri input terzo scenario 74
Figura 11 - Tempi di esecuzione script di discovery 81
Figura 12 - Tempi di esecuzione importazione 82
Figura 13 - Tempi di esecuzione script di parsing 83
Figura 14 - Tempi di esecuzione script di customization 84
Figura 15 - Tempi di esecuzione pipeline di importazione e creazione dello stack 85
Figura 16 - Tempi di esecuzione pipeline di importazione e creazione del blueprint 86
Figura 17 - Tempi di esecuzione pipeline di personalizzazione: disaster recovery 87
Figura 18 - Tempi di esecuzione pipeline di personalizzazione e distribuzione: disaster recovery 88
Figura 19 - Tempi di esecuzione script discovery con 6 risorse vs processo manuale 90
Figura 20 - Tempi di import con set da 1,5 e 10 risorse 92

Introduzione

L'approccio DevOps e, più in particolare, Infrastructure as Code (IaC), sono considerati metodologie innovative per la gestione delle risorse in ambienti di cloud computing e multicloud. L'approccio DevOps deriva dai termini sviluppo (Development) e operazioni (Operations): dal punto di vista lavorativo, i team di questi due ambiti devono collaborare e condividere responsabilità legate al progetto su cui operano. Questo approccio porta con sé alcune pratiche che, se utilizzate, garantiscono maggiore efficienza in tutte le fasi di un progetto: tra questi accorgimenti troviamo l'implementazione di pipeline e l'utilizzo dell'approccio Infrastructure as Code.

Infrastructure as Code è una metodologia che permette di gestire le risorse cloud in modo più semplice ed efficiente; viene utilizzato per lo sviluppo delle infrastrutture cloud tramite codice. Può portare diversi benefici soprattutto per le infrastrutture multicloud poiché, essendo ogni cloud gestito tramite codice, garantisce un maggiore controllo su ogni componente. Negli ultimi anni è stato sviluppato un nuovo strumento IaC chiamato Pulumi. La sua caratteristica principale è la possibilità di scrivere il codice in diversi linguaggi di programmazione, come Python, Javascript, Typescript e Go. Pipeline CI/CD (Continuous Integration/Continuous Delivery) permette, invece, la gestione dei programmi con costanti e più piccoli aggiornamenti, i quali vengono distribuiti nell'ambiente di produzione dopo essere stati testati in ambienti di test. Queste tecnologie innovative vengono sviluppate nel corso del progetto.

Il progetto è stato svolto durante un tirocinio presso l'azienda Liquid Reply, specializzata in ambienti cloud e multicloud, offrendo automazioni dell'infrastruttura, piattaforme per microservizi, soluzioni containerizzate e sicurezza nelle soluzioni cloud proposte.

Il progetto sviluppato fa parte di un piano più esteso che ha come obiettivo quello di aiutare i clienti e le aziende a scegliere un approccio

DevOps, automatizzato per la gestione dell'intera infrastruttura.

La tesi è suddivisa in cinque capitoli; nel primo vengono introdotti i concetti sulle tecnologie e metodologie utilizzate, analizzando gli ambienti di cloud computing con la loro struttura dei datacenter e di ambienti multicloud, l'approccio DevOps e Infrastructure as Code con la presentazione dello strumento Pulumi, la metodologia CI/CD con la descrizione dello strumento Jenkins.

Successivamente, nel secondo capitolo, viene introdotto il progetto di tesi, il quale consiste nell'automatizzare la ricerca e la gestione di risorse cloud già esistenti tramite l'approccio IaC. Verranno descritti gli script sviluppati e le pipeline create per soddisfare alcune esigenze di un possibile cliente o azienda. I processi sviluppati si dividono in tre fasi: discovery, importazione e personalizzazione. La prima parte cerca le risorse che sono online nei vari Cloud Service Provider e che soddisfano determinati filtri. Nella seconda fase, queste risorse vengono importate e viene creato in automatico un file IaC. Infine, questo file viene personalizzato per rispondere a esigenze specifiche.

Nel terzo capitolo sono stati individuati tre scenari di utilizzo in cui possono essere utilizzati gli script sviluppati. Per ognuno di essi viene creata una pipeline che automatizza le operazioni dopo aver inserito in input i parametri necessari. Vengono, nel quarto capitolo, valutati i risultati ottenuti dai test eseguiti sui singoli script e sulle pipeline, in modo da individuare, nell'ultimo capitolo, i limiti e i possibili sviluppi futuri del progetto.

Capitolo 1

Concetti base

Prima di presentare il lavoro sostenuto è fondamentale stabilire una solida comprensione di diversi concetti chiave: il cloud computing, le sue caratteristiche, i suoi vantaggi, l'infrastruttura multicloud, il suo funzionamento, l'importanza dell'approccio DevOps e quali sono alcuni strumenti utilizzati (Infrastructure as Code), e infine come le funzioni e i processi possono essere automatizzati.

1.1 Cloud computing

> *"Il cloud computing è un modello che consente un accesso di rete onnipresente, conveniente e su richiesta a un pool condiviso di risorse IT configurabili (ad esempio, reti, server, storage, applicazioni e servizi) che possono essere rapidamente fornite e rilasciate con il minimo sforzo di gestione o interazione con il fornitore di servizi."* [1]

Un'azienda ha, quindi, la possibilità di erogare ai propri clienti tutti i servizi necessari senza creare e mantenere i propri server, ma utilizzando le risorse necessarie a sostituire l'infrastruttura IT tradizionale. Un'infrastruttura basata su cloud porta con sé diversi vantaggi[2]:

- *Costo*: con l'utilizzo del cloud computing vengono eliminati tutti i costi di acquisto e gestione dell'infrastruttura hardware, anche se si prevede un prezzo legato al tempo di utilizzo;
- *Velocità*: la creazione e la disponibilità di nuove risorse (anche in grandi quantità) è molto rapida, nell'ordine dei minuti;
- *Scalabilità*: le risorse vengono ridimensionate in modo elastico e possono essere create e distrutte quando necessario;

[1] Peter Mell, Timothy Grance (Settembre 2011), NIST SP 800-145, The NIST Definition of Cloud Computing.

- *Produttività*: i team IT possono concentrarsi sui propri obiettivi aziendali, in quanto tutte le attività legate alla gestione e configurazione dell'infrastruttura non sono più necessarie;
- *Affidabilità*: i backup possono essere gestiti e, in caso di guasto, possono essere ripristinati senza perdita di dati, tutto tramite cloud.

1.1.1 Tipologie di cloud computing

La gestione delle risorse cloud si basa su un approfondito studio delle esigenze dell'azienda: non tutti i cloud sono uguali.

Ci sono tre modelli principali di cloud computing[3]:

- *Cloud pubblico*: le risorse cloud appartengono ad un cloud service provider di terze parti e vengono acquistate dalle aziende in base alle loro necessità;
- *Cloud privato*: si opera seguendo il modello del cloud pubblico ma le risorse cloud sono esclusive di una singola azienda o organizzazione, solitamente localizzate nel datacenter locale;
- *Cloud ibrido*: si tratta di ambienti cloud derivati da più di un servizio cloud, cloud privato e pubblico, due cloud privati/pubblici.

1.1.2 Tipologie di servizi di cloud computing

I servizi cloud possono essere di diverse tipologie in base a ciò che l'utente necessita. Sono quattro le tipologie principali di servizi di cloud computing[3]:

- *Hardware-as-a-service (HaaS)*: l'utente affitta i componenti hardware da fornitori che gestiscono questo tipo di servizio. È molto simile ad avere un datacenter locale ma, in questo caso, l'utente non deve pensare a gestire un edificio apposito con corrente, condizionamento, connettività. Questo servizio è spesso utilizzato quando si necessita di alti requisiti di sicurezza[4];
- *Infrastructure-as-a-service (IaaS)*: il provider del servizio offre al cliente l'infrastruttura tramite la virtualizzazione delle risorse dei suoi server fisici. Il cliente è in grado di crearsi le proprie virtual

machine con la propria virtual network e collegare dischi virtuali. Può essere quindi replicato lo stesso ambiente che verrebbe usato con l'infrastruttura tradizionale;

- *Platform-as-a-service (PaaS)*: viene distribuito un set di servizi di base che permette la creazione di software e la gestione dei dati e del software stesso. L'utente, quindi, non deve più gestire l'infrastruttura sulla quale l'applicazione viene eseguita;
- *Software-as-a-service (SaaS)*: il provider offre al cliente software completi con tutte le funzionalità incluse. Spesso sono applicazioni web utilizzabili dall'utente tramite browser. La personalizzazione di queste applicazioni è molto più limitata ma non è più necessaria né la gestione dell'infrastruttura né lo sviluppo dell'applicazione.

Figura 1 - Servizi di Cloud Computing

1.1.3 Cloud computing datacenter

I provider cloud per garantire i propri servizi gestiscono grandi datacenter con tutte le risorse necessarie a soddisfare ogni tipo di cliente. I database cloud operano all'interno di una struttura informatica distribuita, sfruttando l'infrastruttura sottostante dei fornitori di cloud computing come Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP). L'architettura comprende diversi tipi di risorse, tra cui componenti di archiviazione, elaborazione, rete e gestione, tutti perfettamente integrati per lavorare coordinati tra loro. I database cloud sfruttano risorse di calcolo scalabili, consentendo agli utenti di regolare dinamicamente la capacità di elaborazione in base alle richieste del carico di lavoro, ottimizzando così l'utilizzo delle risorse e riducendo al minimo i costi.

I database dei cloud provider sono suddivisi in regioni e zone, per resistere il più possibile a guasti e garantire la disponibilità delle risorse ai clienti in ogni momento.

Le zone sono luoghi fisicamente separati dotati della propria alimentazione, raffreddamento e rete. Ogni zona è costituita da uno o più datacenter vicini tra loro; anche se i datacenter all'interno di una zona sono più di uno, viene considerato una singola entità logica.

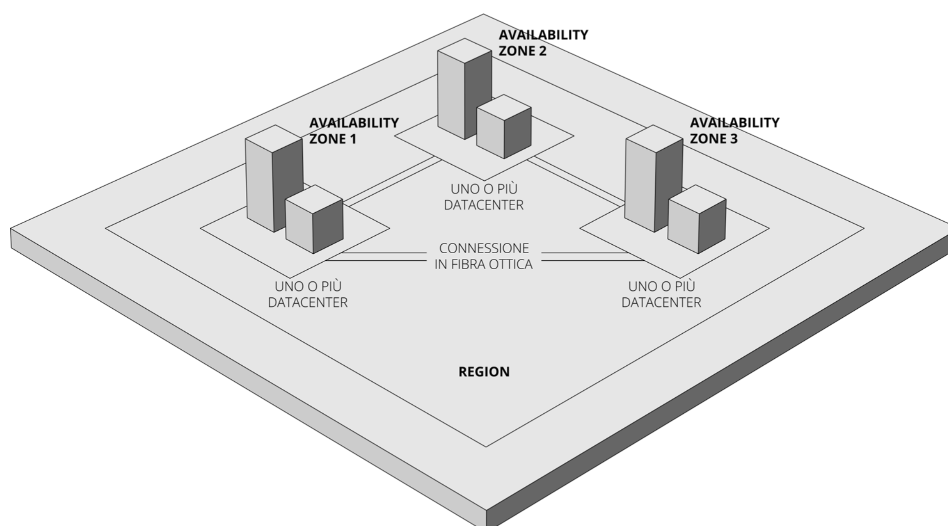

Figura 2 - Regioni e zone

La regione è costituita da una o più zone che sono geograficamente vicine, le quali sono collegate tra loro tramite fibra ottica, garantendo così bassa latenza e ridondanza evitando un disservizio in caso di guasto di rete[5].

Distribuendo le risorse su più zone di disponibilità all'interno di una regione, i fornitori di servizi cloud possono ottenere elevata disponibilità e funzionalità di disaster recovery, riducendo così al minimo il rischio di interruzioni del servizio dovute a guasti localizzati o disastri naturali. Per esempio, consideriamo un sito web distribuito in più zone all'interno di una regione. Nel momento in cui si verifica un guasto in una delle regioni, i servizi in quella zona vengono spostati in un'altra all'interno della regione, garantendo, nel minor tempo possibile, il ripristino del servizio.

Ogni cloud provider offre al cliente la possibilità di scegliere la regione specifica e le zone in cui si vogliono distribuire le risorse: ci sono vari fattori che influiscono nella scelta di esse, come la latenza, i costi e le strategie di recovery.

Per decidere in quale regione le risorse verranno distribuite bisogna considerare dove il servizio verrà maggiormente usato, per permettere una latenza minore nell'utilizzo di esso. Inoltre, i costi relativi a energia, tasse e gestione possono essere molto differenti in base allo Stato in cui i datacenter sono situati[6].

Successivamente alla scelta della regione, l'utente deve considerare se le risorse debbano essere distribuite tra le zone della regione o in una singola zona, per garantire o non garantire ridondanza e disponibilità in caso di guasto.

Ogni cloud provider inoltre può gestire come preferisce le interconnessioni tra regioni diverse. Ad esempio, in Amazon Web Services le regioni sono progettate per essere completamente isolate tra loro per garantire maggiore stabilità e sicurezza[7], a differenza invece di GPC che permette e consiglia il deployment tra più regioni[8], mentre Microsoft Azure permette l'utilizzo di soluzioni multi-region (solo alcune regioni Azure permetto ridondanza cross-region)[9].

1.1.4 Principali cloud service provider

Sono tre i principali cloud service provider: Amazon Web Services, Microsoft Azure e Google Cloud. Amazon Web Services è il cloud provider maggiormente utilizzato. Come riportato nel sito Canalys[10], nel terzo trimestre del 2023, AWS ha una quota di mercato del 31%. Microsoft Azure, invece, copre il 25% del mercato, piazzandosi al secondo posto tra i provider cloud principali. Infine, Google Cloud occupa il 10% del mercato cloud. Le restanti quote di mercato sono divise tra vari cloud service provider minori.

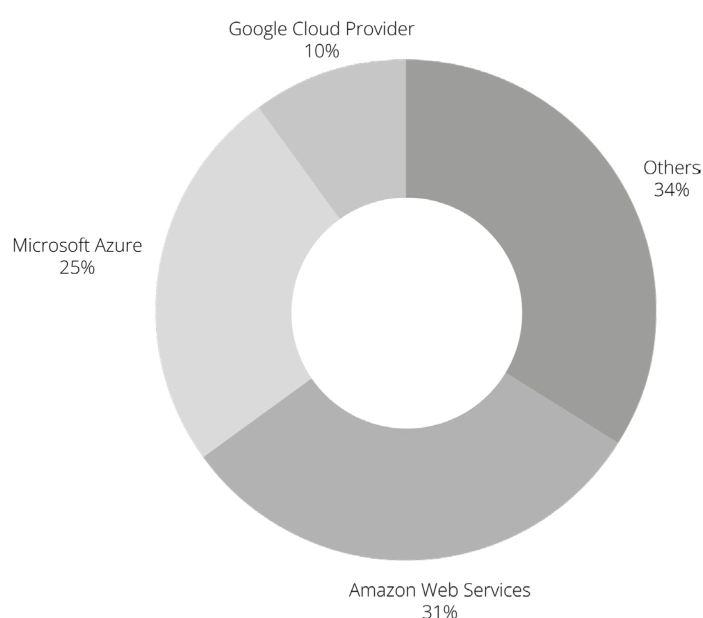

Figura 3 - Cloud Service Provider: percentuali di utilizzo

1.2 MultiCloud

Un ambiente multicloud utilizza servizi di cloud service provider differenti: due o più provider pubblici, due o più provider privati, o una combinazione tra le due soluzioni[11]. Un approccio di questo tipo permette a un'organizzazione di avere un ambiente più affidabile in grado di reggere problemi legati ad un singolo cloud service provider; inoltre, ogni provider offre servizi cloud più efficienti rispetto ad altri. L'azienda può, quindi, scegliere, in base alle funzionalità e applicazioni da sviluppare, su quale cloud service provider distribuire le proprie applicazioni. Come descritto nell'articolo pubblicato da VMware[11], l'intelligenza artificiale è ottimizzata in Google Cloud, mentre le virtual machine basate sul sistema operativo Windows sono più adatte ad essere in Microsoft Azure. Grazie al multicloud, però, tutte queste funzionalità possono interagire tra loro in modo integrato e coerente.

Secondo l'analisi proposta da Gartner[12], si prospetta un cambiamento nell'approccio multicloud: si stima che nel 2025 le aziende passeranno da un approccio "unintentional" ad uno "intentional", basato su cloud distribuiti rispetto a quelli privati, quindi un utilizzo maggiore dei cloud pubblici.

Questo cambiamento è portato dai diversi vantaggi di un approccio multicloud:

- *Flessibilità*: le quantità di risorse utilizzabile è maggiore e rende più semplice la scelta della risorsa più adatta al carico di lavoro da eseguire;
- *Risparmio economico*: i costi sono ridotti, grazie alla possibilità di scegliere la risorsa economicamente più vantaggiosa tra i diversi provider;
- *Affidabilità*: i tempi di inattività dovuti a problemi di un provider sono limitati, in quanto non si ha più un single point of failure;
- *Indipendenza*: non si è più legati a uno specifico fornitore.

L'infrastruttura che si può creare è quindi più facilmente adattabile alle esigenze di un'azienda, ma aumenta la complessità di essa, rendendone più difficile la gestione. Dietro ai numerosi vantaggi di un'infrastruttura

multicloud, vi sono nuove sfide da superare: la gestione delle risorse risulta più complessa, l'interconnessione tra essere è più complicata e lo scambio di dati deve essere ben progettato per non intercorrere in problemi di sicurezza.

Secondo il report di Flexera[13], attualmente la strategia multicloud è la più utilizzata tra le aziende (87%), il restante utilizza un'infrastruttura basata su cloud singolo, l'11% cloud pubblico, il 2% cloud privato. La soluzione multicloud più adottata è quella di un cloud ibrido sia pubblico che privato (72%), mentre il 13% utilizza solo cloud pubblici, il 2% solo privati.

1.3 DevOps

Il termine DevOps è l'unione dei termini "Development" (sviluppo) e "Operations" (operazioni), concetti che aumentano l'efficienza, la velocità, e la sicurezza dei prodotti sviluppati e distribuiti. DevOps rappresenta un grande cambiamento rispetto all'approccio tradizionale. La collaborazione e la condivisione delle responsabilità sono alla base della cultura DevOps. Il lavoro da eseguire è ben pianificato, a partire dall'architettura fino al rilascio e manutenzione. I team di sviluppo e quelli delle operazioni sono responsabili in egual modo del fallimento o successo del prodotto: esso deve essere gestito da entrambi i team in ogni fase, il team di operazioni deve comunque supervisionare lo sviluppo e viceversa. La conoscenza reciproca del lavoro dell'altro team permette la definizione di obiettivi ed esigenze che facilitano il lavoro di entrambi[14].

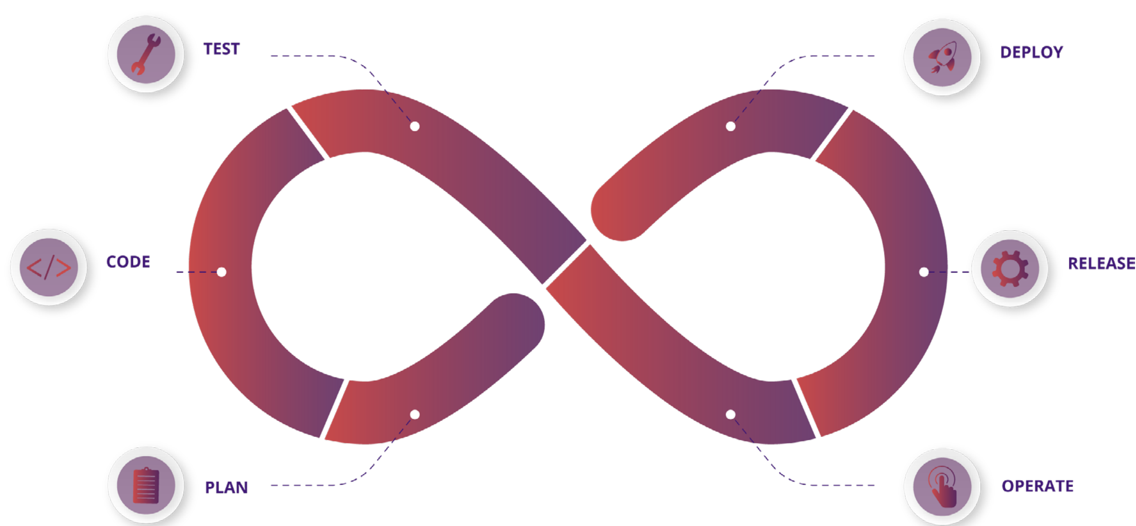

Figura 4 - DevOps

1.3.1 Pratiche DevOps

DevOps porta con sé alcune pratiche che devono essere utilizzate per garantire efficienza elevata durante tutte le fasi di sviluppo e operazione.

Come riportato da Atlassian[15], la prima tra le pratiche fondamentali è la gestione del progetto. Questo deve essere diviso in task e milestone piccoli per permettere dei rilasci più frequenti e di minor grandezza, garantendo una più bassa percentuale di errore e, di conseguenza, una veloce individuazione e correzione di essi. Ogni team è tenuto a valutare requisiti, obiettivi e risultati in modo continuo e ad interagire con gli altri team comunicando feedback e possibili cambiamenti.

Utilizzando un approccio DevOps, è necessario integrare automazioni che riducono l'intervento manuale nella maggior numero di processi possibili. Pipeline CI/CD (Continuous Integration/ Continuous Delivery) sono molto utilizzate. Implementare nel processo integrazione continua (CI) significa che ogni sviluppatore unisce le modifiche apportate al programma principale il più spesso possibile. La consegna continua (CD) è il passaggio successivo all'integrazione continua, la quale permette di distribuire le modifiche in un ambiente di test e successivamente di produzione. Prima di essere rilasciate, un processo automatizzato di test convalida le modifiche effettuate, per cercare di evitare il più possibile problemi nel prodotto finale; anche la distribuzione delle modifiche dopo averle convalidate avviene in modo automatizzato.

Il controllo della versione è una procedura che permette la gestione di applicazioni per conoscere le modifiche apportate, e utile nel momento in cui è necessario ripristinare una versione precedente. Spesso questo tipo di controllo è implementato con l'utilizzo di piattaforme Git, che oltre a tenerne traccia è in grado di aiutarti anche con la gestione dei conflitti a livello di codice[16].

Tutte le pipeline che sono create devono essere monitorate per valutare e correggere eventuali errori durante l'esecuzione di esse. Una seconda fase di monitoraggio deve avvenire direttamente sull'applicazione per evitare bug e guasti, prevenendo un malfunzionamento nel momento in cui il cliente la sta utilizzando. Un sistema di monitoraggio tramite log

e metriche deve essere configurato in modo opportuno, soprattutto al giorno d'oggi, dove la maggior parte delle applicazioni sono composte da microservizi che comunicano tra loro rispetto alle precedenti applicazioni monolitiche[15].

Una pratica molto utilizzata dai team di DevOps è Infrastructure as Code (IaC) ovvero la gestione dell'infrastruttura attraverso lo sviluppo di codice. Questo ne permette la gestione anche attraverso automazioni, evitando la configurazione manuale. La gestione di nuove versioni e la configurazione del sistema avviene tutta tramite codice rendendo le modifiche standardizzate.

1.4 Infrastructure as Code

La gestione dell'infrastruttura cloud viene spesso creata e gestita dalle aziende con processi manuali, aumentando così i tempi di esecuzione e, di conseguenza, la possibilità di errore. Infrastructure as Code (IaC) è un approccio al provisioning e alla gestione di un'infrastruttura tramite lo sviluppo di codice, evitando quindi i possibili problemi sopra elencati.

Il file Infrastructure as Code definisce al suo interno tutte le informazioni necessarie per il provisioning dell'infrastruttura desiderata.

La gestione di quest'ultima attraverso codice permette di avere diversi vantaggi[17]:

- Configurazione e modifica delle risorse semplificati grazie alla gestione tramite codice;
- Controllo di versione, come avviene con il codice di qualsiasi applicazione, con la possibilità di rollback;
- Scalabilità e maggior velocità nella creazione di risorse e nella distribuzione di modifiche nell'infrastruttura;
- Ripetibilità, consentendo la creazione di più ambienti uguali data la presenza di un file con la descrizione delle risorse necessarie;
- Creazione di automazioni per evitare l'intervento manuale ed il possibile errore umano.

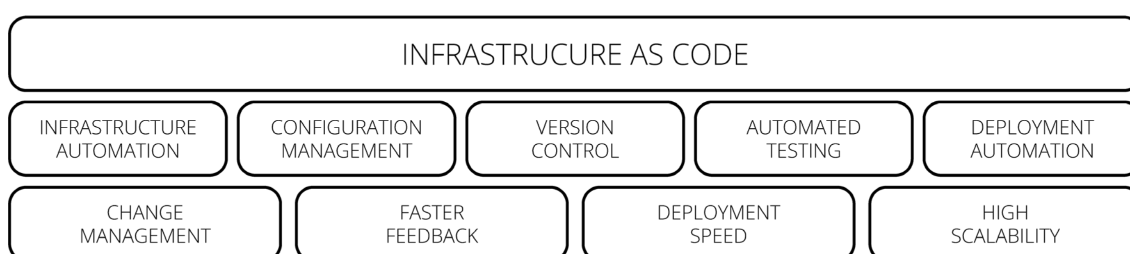

Figura 5 - Infrastructure as Code

L'Infrastructure as Code definisce due metodologie: dichiarativa e imperativa[18]. La prima richiede la configurazione dello stato desiderato del sistema, ovvero le risorse necessarie e le relative proprietà e uno strumento che permetterà l'esecuzione di processi per garantire che le modifiche siano apportate correttamente. Nel secondo caso invece vengono definiti i comandi necessari per il raggiungimento di una configurazione desiderata.

Gli approcci descritti sono più efficaci e vantaggiosi in contesti diversi: in una situazione dove l'infrastruttura si basa sullo stato corrente ed è più soggetta a modifiche future, si preferisce un approccio dichiarativo; d'altra parte, quando le modifiche future sono poche o nulle, la scelta migliore è un approccio imperativo.

Gli strumenti per utilizzare questa metodologia sono diversi, come ad esempio Ansible, Pupper, Chef, Terraform e Pulumi. Per il progetto di tesi è stato scelto quest'ultimo, poiché offre le funzionalità richieste ed è compatibile con più cloud service provider.

1.4.1 Pulumi

Pulumi è uno strumento open-source di Infrastructure as Code che consente lo sviluppo con moderni linguaggi di programmazione, come Python, Go, JavaScript, TypeScript, Java, .NET e YAML, permettendo così l'utilizzo di strutture intrinseche dei linguaggi come cicli, condizioni e funzioni, oltre alla possibilità di utilizzare un IDE a propria scelta con possibilità di controllo di errori e auto completamento[19].

Pulumi permette la creazione, la gestione e la distruzione di un'infrastruttura cloud composta da diverse risorse come quelle di calcolo (virtual machine), di rete (network, subnet, interfacce), di archiviazione (dischi, spazio riservato) attraverso un file di codice IaC. Questa modalità di creazione e gestione dell'infrastruttura è più accessibile, intuitiva e comoda per gli sviluppatori. Sono supportati più di 150 Cloud Service Provider, tra i quali vi sono i più usati: Microsoft Azure, Amazon Web Services e Google Cloud Provider.

Pulumi ha il compito di eseguire il file IaC generato, validando la correttezza di esso. Questi processi sono eseguiti sulla macchina in

cui è installato il programma che può essere un server, una macchina virtuale, un container o un computer tramite comandi lanciati da riga di comando (CLI). Validata la correttezza del codice, lo strumento procederà con la distribuzione delle risorse: viene controllato quali sono quelle da creare, da modificare, da sostituire o da distruggere contattando il Cloud Provider scelto.

Pulumi è composto da un *deployment engine* (motore di distribuzione) e un *language host* (host della lingua). L'host della lingua è composto da un'esecutore del linguaggio, ovvero un programma binario chiamato *pulumi-language-languageName*, che lancia il runtime del linguaggio di programmazione scelto, il quale prepara l'esecuzione del programma Pulumi e controlla l'esecuzione per comprendere la registrazione delle risorse[19].

Il motore di deployment di Pulumi deve comparare lo stato corrente ed effettuare le azioni necessarie per raggiungere lo stato desiderato (approccio dichiarativo). Questo controlla lo stato dell'infrastruttura in gestione, e decide per ogni risorsa se deve crearla (nel caso non esista), aggiornarla con nuovi parametri, o sostituirla con una nuova. Il motore deve collaborare con il provider di risorse per permettere la lettura o l'aggiornamento di quelle già presenti nel cloud provider e la creazione di quelle non esistenti. Questo avviene grazie a provider di risorse che sono composti da plugin, responsabili della gestione della risorsa, e l'SDK, che associa i tipi di risorsa del provider che si utilizza[19].

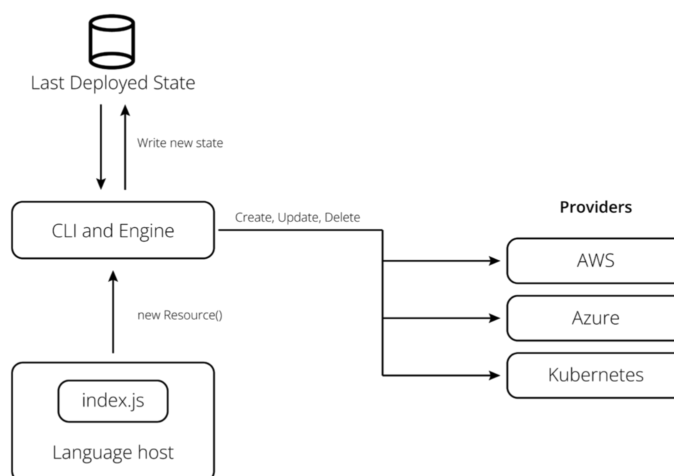

Figura 6 - Struttura Pulumi

Dopo la scelta del linguaggio, che viene scelto dallo sviluppatore, si crea un nuovo progetto nel quale verranno descritte le risorse che si desiderano avere nell'infrastruttura, con le relative proprietà. Nel progetto sono contenuti il codice sorgente e tutti i metadata necessari al funzionamento.

I programmi Pulumi possono essere divisi in diversi stack, i quali sono delle istanze isolate e configurabili. Spesso, in ogni fase di sviluppo, viene creato un apposito stack: development, testing, production ecc[20].

Pulumi Cloud è un servizio cloud offerto dallo strumento in questione che permette di avere una panoramica delle infrastrutture create: se viene utilizzato contiene tutte le informazioni relative ai progetti creati, alle risorse gestite e ai secret che sono stati salvati in modo sicuro. Può essere utilizzato in ogni progetto, ma è necessario prima effettuare l'accesso. L'utilizzo del cloud permette di avere diversi vantaggi: migliora la collaborazione tra i team e permette l'integrazione con pipeline CI/CD.

Quando è necessario creare un nuovo progetto Pulumi deve essere invocato il comando *pulumi new*: viene creato un file *Pulumi.yaml* che contiene le informazioni relative a quel progetto, come il nome, il linguaggio, e una descrizione che sono scelti durante l'esecuzione del comando. All'interno del progetto viene richiesto almeno uno stack, creato con il comando *pulumi stack init stackName*: questo è vuoto e settato come attivo. Nella cartella del progetto sarà ora presente un file *Pulumi.stackName.yaml* con le proprietà del progetto: esse sono configurabili mediante il comando *pulumi config set propertyName propertyValue* oppure modificando e salvando il file dello stack.

Dopo aver configurato il progetto e lo stack è possibile lanciare il comando *pulumi up* per eseguire il programma Pulumi e vedere apportate le creazioni e le modifiche delle risorse che sono state descritte nel file *__main__* del progetto. La distruzione di esse può avvenire tramite il comando *pulumi destroy*, e, successivamente anche lo stack può essere eliminato mediante il *comando pulumi stack rm*.

1.5 Pipeline CI/CD

L’approccio CI/CD è fondamentale in ambito DevOps: è l’insieme delle fasi di integrazione continua (CI), distribuzione continua e/o deployment continuo (CD).

L’integrazione continua comprende le fasi di build, test e merge del codice sorgente di un software in maniera frequente e automatica, a differenza delle altre fasi che comprendono il rilascio in un repository e la distribuzione del software in un ambiente di produzione[21].

L’automazione di queste procedure permette una migliore distribuzione e un efficace monitoraggio del software durante tutto il suo ciclo di vita: questo è ciò che viene denominato pipeline CI/CD[22]. L’approccio CI/CD viene eseguito come una metodologia ciclica: una volta completata una parte, viene eseguita quella successiva. In questo modo, un prodotto è sempre aggiornato, fornendo costantemente nuove versioni.

L’Integrazione Continua (CI) è un processo di automazione utilizzato dagli sviluppatori per facilitare l’integrazione delle modifiche al codice in un unico repository. Questo processo consente di eseguire in modo più efficace e frequente le modifiche, avviando test automatici per verificare l’affidabilità delle modifiche che sono state eseguite al codice. L’esecuzione dei test permette agli sviluppatori di avere un feedback immediato sulle modifiche effettuate, permettendo loro di intervenire e correggere i problemi riscontrati. I test solitamente sono di unità e di integrazione: vengono valutati diversi aspetti dell’app, delle classi, dei moduli e la compatibilità tra essi. Questo approccio è utile soprattutto quando gli sviluppatori lavorano in maniera simultanea su diverse funzioni della stessa applicazione.

La fase di distribuzione continua (Continuous Delivery) rende automatico il rilascio di un prodotto in un repository dopo che sono stati superati i test di unità e integrazione. Questa fase permette di avere uno spazio con tutto il codice in continuo aggiornamento.

Il deployment continuo permette, infine, il rilascio automatico delle modifiche dal repository all’ambiente di produzione. Questo processo riduce il carico di lavoro dei team di Ops e accelera la distribuzione

dell'app. Una modifica che supera tutti i test automatizzati, verrà automaticamente rilasciata e sarà pronta per essere utilizzata. Tuttavia, il deployment continuo necessita di test automatici adeguati, in quanto, non essendoci operazioni manuali, bisogna controllare e validare il completo funzionamento.

L'applicazione nell'approccio DevOps permette di interconnettere i team di Development e di Operations, aumentandone la produttività e la qualità del prodotto sviluppato, oltre alla diminuzione della probabilità di errore che può esserci con processi manuali.

Sono diversi gli strumenti che permettono l'implementazione CI/CD, ad esempio: GitHub, GitLab, Jenkins, Bitbucket, Azure pipelines, AWS CodePiepline. Per il lavoro di tesi svolto è stato scelto Jenkins come strumento per la creazione di pipeline CI/CD che interagisce con i repository Git opportunamente creati.

1.5.1 Jenkins

Jenkins è un software open-source per automazioni CI/CD scritto in Java. Jenkins rende possibile l'implementazione di automazioni CI/CD nel software con la creazione di pipeline. L'esecuzione di comandi in sequenza e la convalida del codice tramite test automatici sono alcune delle operazioni che possono essere programmate in una pipeline. Inoltre, è possibile creare dei trigger che attivano la pipeline, ad esempio quando viene eseguito un commit in un repository.

In Jenkins sono presenti diversi plugin che permettono l'estensione di alcune funzionalità, come ad esempio l'integrazione con Git, oppure anche con cloud service provider (Azure, AWS) che consentono funzionalità utili.

Jenkins utilizza un'architettura master-slave, i quali comunicano tra loro grazie al protocollo TCP/IP. Il master è il server principale che ha il compito di pianificare i lavori da distribuire tra i vari slave e monitorare l'esecuzione di questi. Gli slave sono principalmente macchine remote comandate dal master. Il master può essere configurato in modo che un lavoro venga sempre eseguito su un particolare slave, altrimenti la scelta è automatica[23].

Come anticipato in precedenza, in Jenkins si possono creare delle pipeline che permettono l'esecuzione automatica di step, ovvero una lista di comandi da eseguire in sequenza. Una pipeline è composta da diversi stage: l'intera esecuzione viene divisa in varie fasi ed ognuna di esse crea uno stage.

L'esecuzione di uno stage può fallire o passare: quando uno stage fallisce tutta la pipeline viene interrotta.

Capitolo 2

Progetto

Il progetto realizzato in collaborazione con l'azienda Liquid Reply mira a semplificare e automatizzare la gestione delle risorse cloud attraverso l'implementazione di un approccio Infrastructure as Code (IaC). Questo approccio risulta particolarmente utile per le aziende che utilizzano servizi cloud e multicloud, le quali spesso si trovano a dover gestire un gran numero di risorse attraverso il portale del provider di servizi cloud.

Il progetto è composto da cinque fasi principali:

- Discovery: in questa fase, vengono identificate le risorse online da importare nel file IaC. Vengono applicati uno o più filtri per selezionare solo le risorse necessarie;
- Import: viene creato un progetto Pulumi, all'interno del quale vengono importate le risorse identificate nella fase di Discovery. Queste risorse saranno quindi gestibili attraverso il file IaC generato;
- Parsing: il file IaC viene modellato per creare dei blueprint riutilizzabili in vari scenari;
- Customization: i blueprint vengono personalizzati per adattarsi al caso d'uso specifico del disaster recovery;
- Automation: vengono create delle pipeline per gli scenari selezionati, che verranno descritti nel capitolo successivo.

In sintesi, questo progetto offre alle aziende un modo più efficiente e controllabile per gestire le loro risorse cloud.

2.1 Preparazione dell'ambiente e primi test

I primi passi sono caratterizzati dallo studio degli strumenti da utilizzare in questo progetto.

La funzionalità di importazione di Pulumi consente di importare una risorsa cloud esistente in un progetto Pulumi, in modo che possa essere gestita tramite il file IaC. Sono due i modi in cui si possono importare le risorse:

- Comando di importazione CLI: attraverso il comando *pulumi import* viene cercata la risorsa nel cloud service provider specificato nello stack del progetto Pulumi con i parametri *tipo, name, id*.

```
Pulumi import <type> <name> <id>
```

- Tramite codice: viene inserita nel codice l'opzione *import* nella dichiarazione della risorsa, in egual modo sono presenti i parametri *tipo, name, id* per trovare la risorsa nel CSP.

```
Variable = aws.ec2.Instance("my-instance",
      Name="instance_name",
      Opts=ResourceOptions(import_="instance_id"))
```

Lo scenario che andiamo a trattare implica l'importazione di più di una risorsa: viene quindi utilizzato il comando di importazione da CLI senza però specificare la risorsa nella linea di comando, ma utilizzando un file che permette a Pulumi l'importazione in blocco di più risorse.

Vengono quindi creati degli script per permettere il funzionamento automatico di questa funzionalità e, successivamente, la personalizzazione per la creazione di un blueprint delle risorse importate utile in molteplici casi d'uso.

I Cloud Service Provider presi in considerazione durante lo sviluppo degli script sono Microsoft Azure e Amazon Web Services. Ad inizio progetto è stato scelto un set di risorse per entrambi i CSP su cui testare gli script creati.

Per quanto riguarda Microsoft Azure le risorse considerate sono:

- Resource Group;
- Storage account;
- Virtual Network;
- Subnet;
- Network Interface;
- Virtual Machine.

In modo simile anche per AWS sono state scelte le risorse:

- Bucket S3;
- Virtual Private Cloud (VPC);
- Subnet;
- Security Group;
- Elastic Compute Cloud Instance (EC2);
- Network Interface.

Queste risorse sono state scelte per creare un ambiente in cui ci fosse una macchina virtuale (virtual machine in Azure e EC2 in AWS), con uno storage (Storage Account e Bucket) e le risorse relative al collegamento della vm a una network per ogni Cloud Provider.

Sono stati creati per entrambi i CSP due file IaC per la creazione delle risorse appena descritte, con le dipendenze tra risorse e tag specifici, in modo da differenziare le risorse di questo progetto da altri già esistenti.

Tutte le operazioni sono state eseguite sulle subscription AWS e Azure di Liquid Reply.

2.2 Discovery

L'importazione in blocco necessita di un file con estensione JSON (JavaScript Object Notation), che verrà utilizzato come input del comando *pulumi import*. Il primo passo del progetto, quindi, è stato quello di automatizzare la creazione del file utilizzando le API dei cloud provider usati. Gli script creati sono diversi per Azure e AWS poiché le API utilizzate ritornano risultati diversi. Questi devono essere interpretati e modificati per renderli ad hoc per il file di cui necessitiamo. Gli script creati hanno un filtro che permette di scegliere quali sono le risorse da importare. Per Azure è stato scelto di creare un filtro per avere le risorse appartenenti ad un resource group oppure che hanno un tag specifico (coppia chiave-valore). Invece per AWS il filtro è stato fatto sulla regione e un tag specifico.

La struttura del file richiesto è composta da due oggetti: *nameTable* e *resources*. La prima è una mappa con, in ogni riga, la coppia <*name*> *:* <*id*> per ogni risorsa, la seconda, invece, è costituita da un oggetto per ogni risorsa contente *name, id, type*. Di seguito viene riportato un esempio:

```
{
      “nameTable”: {
            “name_1”:”id_1”,
            “name_2”:”id_2”,
            …
      },
      “resources”: [
            {
                  “name_1”: …,
                  “id_1”: …,
                  “type_1”: …
            },
            {
                  …
            }
      ]
}
```

La *nameTable* è necessaria quando sono presenti *parent* o quando il *provider* è diverso da quello del progetto Pulumi, ovvero *azure-native* o *aws-classic* che sono stati scelti per risorse da importare per questa tesi.

In questo lavoro, le risorse considerate non necessitano dei due campi, ma la tabella viene comunque creata nel caso di necessità futura.

I tre campi presenti in *resources* sono obbligatori per ogni risorsa e permettono a Pulumi di individuare e importare correttamente la risorsa presente nel Cloud Service Provider.

Lo sviluppo del progetto è stato eseguito su una macchina virtuale Ubuntu, utilizzando il linguaggio di programmazione Python con Visual Studio Code.

2.2.1 Azure

Lo script di discovery delle risorse in Microsoft Azure è sviluppato in Python utilizzando l'SDK Azure per Python. In particolare, è stato utilizzato il pacchetto *azure.identity* per l'accesso all'account del Cloud Service Provider: durante la fase di sviluppo, l'accesso all'account Azure è stato eseguito tramite CLI con il comando *az login* dopo l'installazione di Azure CLI. Per la ricerca delle risorse nel Cloud Provider si sono utilizzate le funzioni del pacchetto *azure.mgmt.resource*.

La prima fase di sviluppo prevede il recupero delle informazioni necessarie delle risorse appartenenti ad un Resource Group passato come input; il filtro relativo ai tag, è stato implementato in un secondo momento.

L'esecuzione delle funzioni richiede l'accesso all'account che avviene tramite la classe *DefaultAzureCredential()*, in grado di acquisire le credenziali necessarie dopo aver effettuato il login con Azure CLI. Esse sono utilizzate come input nella classe *ResourceManagementClient(credential, subscription)*: anche la subscription viene inserita in input all'esecuzione dello script.

Le informazioni delle risorse vengono acquisite con la funzione *resources.list()* della classe *ResourceManagementClient* in cui può essere scelto un

filtro: in questo caso viene confrontato il nome del Resource Group con quello scelto in input. I dati ritornati da questa funzione sono quelli necessari per la creazione del file ma è stato notato che il Resource Group non era presente, poiché esso non rispetta il filtro: il Resource Group non dipende da sé stesso. Per ovviare a questo problema, viene chiamata la funzione *resource_groups.list()* della medesima classe che ritorna l'elenco di tutti i Resource Group; successivamente viene cercato quello scelto in input e viene salvato nella lista delle risorse.

Per ogni risorsa è stato creato un vettore di dizionari contenente i valori richiesti: *name, id, type* per ogni risorsa. Per quanto riguarda *name* e *id* non vi sono problemi in quanto sono specificati all'interno dei dati delle risorse ritornati dalla funzione precedentemente descritta. Non è così semplice, invece, recuperare il tipo della risorsa, in quanto quello descritto dai dati acquisiti è quello relativo alla denominazione ARM, Azure Resource Manager. È stato necessario quindi creare una funzione che convertisse il tipo ARM nel tipo richiesto da Pulumi: sono stati analizzati i tipi delle risorse prese in considerazione, sia in formato ARM che Pulumi, per trovare un algoritmo che risolvesse il problema. Di seguito troviamo un esempio dei due tipi di formato:

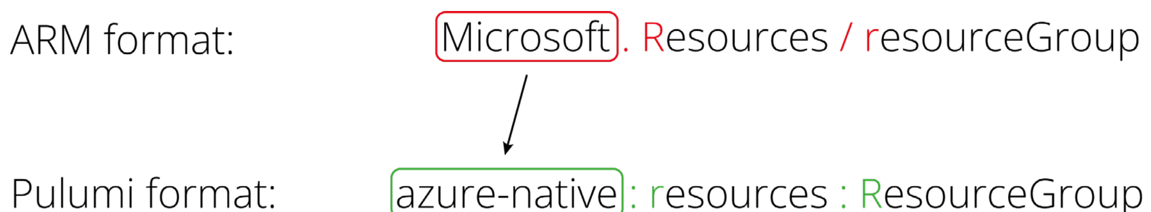

Figura 7 - Traduzione tipo risorsa

La funzione sviluppata è funzionante per il set di risorse utilizzate e tutte quelle che hanno uno schema simile. La funzione divide la stringa del tipo ARM passata in input in parti utilizzando il carattere "/" come separatore.

Successivamente, divide ulteriormente la prima parte del tipo di risorsa ARM utilizzando il carattere "." come separatore. La stringa di output viene creata partendo dal "azure-native:" (indica il provider Pulumi utilizzato) e concatenando la seconda parte del secondo split

convertita in minuscolo. Infine, la funzione aggiunge l'ultimo elemento della prima divisione, con il primo carattere convertito in maiuscolo e privato dell'ultimo carattere.

Durante la prima fase di sviluppo è stato usato anche il pacchetto *NetworkManagementClient* per trovare i dettagli della subnet presente nel set di risorse utilizzate, in quanto non appariva nella lista delle risorse. Dopo l'aggiunta al file di output viene invocato il comando d'importazione. L'esecuzione termina in errore, poiché la subnet non è una vera e propria risorsa ma una proprietà della Virtual Network: è sufficiente quindi avere la rete virtuale tra le risorse e verrà automaticamente impostata anche la sua subnet.

A questo punto tutte le informazioni sono salvate nel vettore: viene generato un ulteriore dizionario per la creazione della *nameTable* e infine viene creato il file JSON con la struttura richiesta utilizzando la funzione *json.dumps* del pacchetto JSON per Python.

Il secondo script, che implementa il filtro tramite la coppia chiave valore di un tag, è stato sviluppato nel medesimo modo cambiando il filtro delle risorse.

2.2.2 Amazon Web Services (AWS)

La sequenza logica di operazione è molto simile a quella seguita per il caso di Microsoft Azure. È stato sviluppato in Python con l'implementazione dell'SDK Boto3 di AWS. Per eseguire tutte le operazioni che verranno descritte è necessaria la creazione di una sessione tramite la classe Session, la quale richiede i seguenti parametri: *aws_access_key_id*, *aws_secret_access_key*, *aws_session_token*, *region_name*. Questi sono passati in input quando viene eseguito lo script. Come possiamo notare, la sessione dipende dalla regione, poiché le regioni in AWS sono considerate isolate: vedremo anche successivamente che durante l'importazione delle risorse è necessario specificare la regione presa in considerazione.

Inizialmente si è creato un client *ResourceGroupTaggingAPI* ma è stato difficile estrapolare i dati necessari, in particolar modo il tipo della risorsa. Si è scelta, quindi, una strada alternativa: è stato creato un client

resource-group che va a rappresentare gruppi di risorse utilizzando criteri definiti da tag. In questo caso il criterio utilizzato è una coppia chiave-valore: viene creata una specifica query che cerca tra tutte le risorse, e ritorna solo quelle che rispettano il criterio scelto.

Il risultato di ritorno dalla funzione viene analizzato e per ogni risorsa estrapolato ResourceARN e ResourceType: mediante questi due valori sono state sviluppate due funzioni che estrapolano *name, id, type*.

La funzione *ARN_to_id_name(arn, type)* utilizza questi due valori in input e ritorna il nome e l'id della risorsa. Come per il caso di Azure, il tipo delle risorse deve essere convertito da quello di AWS a quello di Pulumi. La funzione *aws_to_pulumi_type(aws_type)* esegue questa traduzione: dal tipo AWS che viene passato in input alla funzione, vengono eseguiti degli split per la creazione di una nuova stringa relativa al tipo Pulumi corretto. Nella prima versione, lo script a questo punto creava il file JSON necessario per l'importazione nel medesimo modo in cui è stato creato con Microsoft Azure. Dopo diversi test si è notato che le EC2 Instance terminate pochi istanti prima, durante la fase di discovery venivano lette e salvate sul file, poiché AWS, prima di eliminare completamente una Instance, modifica lo stato di essa in "*terminated*"; solo dopo qualche minuto la elimina. Questo causava un problema durante la fase successiva di importazione poiché Pulumi, leggendo il file JSON, cercava di recuperare informazioni su Instance terminate, non riuscendo ad ottenerle, causando quindi un errore nell'importazione. È stato necessario, perciò, creare un nuovo client di tipo EC2 (*describe_intance_status*), che permettesse di avere tutte le informazioni di ogni Instance, comprese quelle terminate. Dopo la lettura dello stato dell'istanza è stato creato un vettore in cui venivano inserite solo le instance con stato diverso da "*terminated*", permettendo così di filtrare le risorse terminate e non esportarle nel file JSON.

2.3 Import

A questo punto, è stato creato il file con i dettagli delle risorse da importare e "iac-izzare" grazie a Pulumi, il quale permette di automatizzare questo processo. Il comando da utilizzare in questo contesto è *pulumi import*: per questo è necessaria la creazione di un nuovo progetto Pulumi specificandone il nome e il provider delle risorse.

Durante lo sviluppo del progetto è stato effettuato il login in Pulumi tramite CLI ed è stato utilizzato Pulumi Cloud per permettere di avere lo stato di ogni risorsa salvato non solo localmente. Facendo accesso con il proprio account Pulumi, si ha una panoramica di tutte le risorse gestite nei vari cloud provider e i vari progetti e stack che sono stati creati.

I provider di risorse utilizzati sono azure-native alla versione 2.28.0 e *aws-classic* alla versione 6.20.1. Il progetto Pulumi, in cui è presente lo stack con le risorse importate, viene creato con il comando:

Azure: `pulumi new azure-python --name ${PROJECT_NAME} --stack ${STACK_NAME}`

AWS: `pulumi new aws-python --name ${PROJECT_NAME} --stack ${STACK_NAME}`

Nel caso di AWS è necessario specificare, come anticipato nel capitolo precedente, la regione in cui le risorse si trovano con il comando:

```
pulumi config set aws:region ${REGION_NAME}
```

Dopo aver concluso i passaggi precedenti è possibile procedere con l'importazione delle risorse. Il comando da invocare è *pulumi import*, a cui passiamo come parametro il nome del file creato nella fase precedente e il nome del file IaC che verrà generato automaticamente in output.

```
pulumi import -f ${FILENAME_RESOURCES}.json -o ${FILENAME_IAC}.py
```

Il parametro -o è opzionale: nel caso non fosse presente l'output viene stampato a terminale. Il progetto è stato sviluppato in Python, che è stato scelto anche come linguaggio per Pulumi: per questo motivo sono stati usati i pacchetti *azure-PYTHON* e *aws-PYTHON* e l'estensione del file IaC in output è *.py*.

Quando l'esecuzione del comando d'importazione termina avremo il file IaC, su cui possiamo eseguire le modifiche, e uno stack Pulumi, in cui sono salvati gli stati delle risorse. Le risorse sono così state "iac-izzate" e sono ora gestibili e configurabili modificando solo le linee di codice del file creato.

A differenza di Azure, il caso di AWS ha riscontrato degli ulteriori problemi legati al provider scelto: *aws-classic*. Il provider AWS Classic di Pulumi è creato attorno a AWS CloudFormation, che è il servizio AWS che gestisce le risorse come codice. Quando si crea una risorsa con il provider AWS in Pulumi, si crea indirettamente uno stack CloudFormation che gestisce la risorsa. D'altra parte, il provider AWS Native di Pulumi lavora direttamente con l'API di AWS, senza passare attraverso CloudFormation. Questo approccio può offrire aggiornamenti più immediati e talvolta supportare nuove funzionalità AWS più rapidamente rispetto al provider AWS basato su CloudFormation, poiché non deve attendere che tali funzionalità siano implementate in CloudFormation. Il provider *aws-native* è ancora in fase di preview: non tutte le risorse AWS sono già supportate in questo provider, solo quelle compatibili con le API AWS Cloud Control. Per questo motivo si è scelto di utilizzare il provider classic per tutte le risorse.

Questa scelta ha portato però ad alcuni problemi: in particolare, alcune proprietà delle risorse possono essere definite da più di un parametro. Nel caso di creazione manuale sta allo sviluppatore scegliere quali proprietà utilizzare, ma nel caso di importazione automatica vengono importate entrambe. I parametri duplicati sono semplici da trovare in quanto appaiono come warning alla fine dell'importazione, lasciando il compito allo sviluppatore di effettuare un controllo. Inoltre, anche i tag

delle istanze s3 Bucket non vengono importati.

A questo punto le risorse sono state importate correttamente:

- In Microsoft Azure, le risorse non presentano problemi e possono essere gestite dal file IaC;
- In AWS, a causa dell'errore descritto, necessitano di una modifica che verrà automatizzata nella prossima fase. Il problema verrà probabilmente risolto con l'utilizzo del provider *aws-native* quando non sarà più in fase di preview.

2.4 Parsing

Il file generato automaticamente può essere utilizzato per la gestione delle risorse che sono online, ma se si vuole un file IaC in grado di distribuire o modificare risorse indipendentemente dal fatto che siano già state create, è necessario apportare alcune modifiche al file generato.

Se si analizza questo file, si può notare che l'interconnessione tra le varie risorse è descritta mediante l'id o il nome della risorsa da cui dipende: questo però è sotto forma di stringa, non permettendo a Pulumi di capire in automatico quali siano le dipendenze tra risorse. Nel momento in cui le risorse sono già state create e sono online non vi è nessun problema, ma in una fase di creazione di esse, Pulumi le crea con un ordine casuale e in parallelo: se una risorsa X dipende da una risorsa Y, nel momento in cui Pulumi cerca di creare X, non è garantito che Y sia stata già creata. Nel caso in cui la risorsa non è stata creata, Pulumi termina in errore in quanto non riesce a trovare l'id o il nome della risorsa in quanto non esiste. È necessario quindi trovare un modo per permettere a Pulumi di riconoscere le dipendenze tra le risorse. È possibile ottenere questo risultato in due modi:

- Sostituire la stringa con nome o id con, rispettivamente nome_variabile.name, nome_variabile.id;
- Utilizzare l'opzione di Pulumi *depends_on*: in ogni risorsa viene aggiunta una riga tra i parametri `opts=pulumi.ResourceOptions(depends_on=[…])`, inserendo tra le parentesi quadre tutti i nome_variabile da cui dipende.

In questo modo Pulumi, nel momento in cui deve creare una risorsa, conosce i dettagli delle sue dipendenze e schedula la creazione di esse in modo intelligente: la risorsa viene creata solo quando tutte le sue dipendenze sono già state generate.

In entrambi i casi viene utilizzato il nome della variabile: ciò significa che questa deve essere dichiarata prima del suo utilizzo. È necessario quindi una modifica del file con riorganizzazione delle righe di codice. È stata, inoltre, aggiunta una funzione di modifiche "manuali", ovvero

modifiche che sono necessarie per il corretto funzionamento del file e che possono essere modificate o aggiunte per rendere lo script personalizzabile.

Per sviluppare gli script, è stato necessario condurre un'analisi del codice per trovare un algoritmo che permettesse la risoluzione di questi problemi. Sia nel caso di Microsoft Azure, sia di Amazon Web Services è stato sviluppato un programma che legge riga per riga il file generato nel punto precedente, e apporta le dovute modifiche creando un nuovo file IaC in output. Per svolgere tutte le operazioni è stato creato un oggetto *Code* in cui viene salvato il nome della risorsa: viene così istanziata una lista con i nomi delle risorse dipendenti e una lista di linee di codice. Inoltre, sono state programmate delle funzioni interne alla classe, per inserire ed eliminare le risorse o le linee di codice, e una funzione di debug, che permette di stampare l'oggetto *Code*.

Inoltre, viene utilizzato il file creato nella fase di discovery delle risorse, dal quale vengono salvate in due variabili la *nameTable* e le *resources*, rendendo così più facile il riconoscimento di *id* e *name* delle varie risorse.

2.4.1 Microsoft Azure

Sono state sviluppate quattro funzioni:

- Input: lettura riga per riga, individuazione di ogni risorsa;
- Personalizzazione: modifica di alcune linee di codice per rendere il file eseguibile anche in situazioni in cui le risorse non sono online;
- Gestione dipendenze: creazione della stringa con le dipendenze da inserire nel codice;
- Output: creazione del file di output seguendo un algoritmo per avere le dipendenze gestite correttamente.

<u>Input</u>

La funzione opera riga per riga e si basa su un serie di espressioni regolari che identificano le righe principali. Viene dichiarata una classe *Code* iniziale con nome "start" in cui vengono inserite tutte le righe

precedenti alla prima dichiarazione di una risorsa; successivamente, ogni volta che viene trovata la dichiarazione di una risorsa, viene istanziato un oggetto *Code* (chiamato *codeStruct*) con il nome della risorsa stessa. Queste strutture *Code* vengono inserite in una lista chiamata *codeVec*.

La prima condizione va a determinare se la riga rappresenta l'inizio di una dichiarazione di una nuova risorsa attraverso la regex:

```
re.search(" = azure_native\.", x) ≠ None and re.search("\(\"",
x) ≠ None
```

In questo caso, l'oggetto *codeStruct* creato nelle iterazioni precedenti viene inserito nella lista di oggetti *Code*; successivamente poi viene ridefinito l'oggetto *codeStruct* con il nome della nuova risorsa, e le liste due liste all'interno vuote. Infine, viene inserita la prima riga opportunamente modificata per essere compatibile con il linguaggio Python.

Nella fase di importazione delle risorse, ognuna di esse viene automaticamente importata con l'opzione di Pulumi di protezione, che non permette l'eliminazione della risorsa fino a quando non viene inviato il comando opportuno per eliminare questa protezione. Dal momento in cui in questo scenario viene creato un blueprint che verrà utilizzato per la creazione di nuove risorse, l'opzione di protezione viene eliminata: in questo caso non ci sono conflitti con risorse create precedentemente. Per cui se nella riga è presente "`opts=pulumi.ResourceOptions(protect=True)`", essa viene eliminata.

Se la riga non soddisfa né *if* della prima condizione né *else if* della seconda, significa che seguirà le operazioni della condizione *else*. In essa vi sono tre condizioni da controllare:

- Se è presente una dipendenza legata al nome della risorsa;
- Se è presente una dipendenza legata all'id della risorsa;
- Se è una riga di codice standard.

Per effettuare i primi due check è stato necessario utilizzare la *nameTable* che è stata importata ad inizio esecuzione, poiché permette di trovare il nome della risorsa, nonché anche il nome della variabile della risorsa nel file IaC, dal suo id. Nel primo caso vengono iterati i *values* della tabella, che corrispondono agli id delle risorse. Se uno di questi è presente nella riga in considerazione, viene ricavato il nome della risorsa e successivamente vengono inserite sia la riga nella lista di righe di codice di *codeStruct*, sia il nome della risorsa nella lista delle dipendenze dell'oggetto *codeStruct*. Inoltre, viene settato un flag al valore per ottimizzare l'esecuzione dello script, non procedendo con le condizioni successive in quanto non è possibile avere sulla stessa riga sia un id, che un nome di una risorsa.

Successivamente, se il flag è diverso da 1, ovvero non è stato trovato alcun id nella riga, viene controllato se è presente un nome di una risorsa, verificando se nella riga è presente un *key* della tabella. In caso affermativo, viene inserita la riga e il nome nell'oggetto *codeStruct* come nel caso precedente. Infine, il flag viene impostato a valore 2.

Se la riga non soddisfa nessuna delle condizioni precedenti, significa che è una riga di codice senza essere la prima riga di una risorsa o una dipendenza: per questo motivo viene semplicemente inserita la riga in *codeStruct*.

Durante alcuni test che sono stati eseguiti, si è notato che nello scenario in cui il file IaC con le modifiche descritte in questa fase viene eseguito con il comando *pulumi up*, la creazione di macchine virtuali fallisce poiché viene importato anche il disco di OS (nel set di risorse utilizzato non è stato creato nessun disco di dati, il problema persiste anche per questo tipo di dischi): Microsoft Azure non permette la connessione di un disco creato in precedenza ad una vm. È necessario quindi che il disco venga generato durante la creazione della macchina virtuale. Per risolvere questo problema è necessario eliminare le risorse di tipo disco dal file IaC ed eliminare la riga nella dichiarazione della virtual machine in cui è presente l'id del disco già creato. Queste operazioni verranno effettuate nelle funzioni successive: in questa viene controllato quali sono i dischi collegati alla macchina virtuale e vengono inseriti in una lista (*os_disk_name*) che verrà utilizzata successivamente. Infine, prima

di ritornare la lista *codeVec*, viene inserito l'ultimo oggetto *codeStruct*.

Personalizzazione

In questa funzione viene risolto il problema descritto precedentemente eliminando dalla lista *codeVec* gli oggetti con nome presente nella lista *os_disk_name* e commentando la riga nella dichiarazione della virtual machine con l'id del disco. Inoltre, con la creazione di una virtual machine è necessario settare una password: questa durante l'importazione non viene inserita nel file IaC per questioni di sicurezza. Nel momento in cui viene utilizzato questo script si sta creando un blueprint che può essere personalizzato prima della sua esecuzione: è stata quindi aggiunta una password di default che deve essere necessariamente sostituita.

Un ultimo errore durante la fase di creazione di una virtual machine è legato alla proprietà *require_guest_provision_signal* che viene in automatico importata a *True*. La subscription utilizzata per effettuare i test non ha la feature *Microsoft.Compute/Agentless* abilitata, non permettendo l'utilizzo di questa opzione. In questa funzione viene perciò commentata la riga in cui è presente.

Gestione delle dipendenze

In questo momento abbiamo il file IaC generato dall'importazione salvato in una lista di oggetti *Code*: ognuno di essi rappresenta una risorsa, tranne il primo che ha salvate solo le righe precedenti alla dichiarazione della prima risorsa. In questa funzione viene scansionato ogni oggetto e aggiunta la riga con le dipendenze. Questa viene strutturata in questo modo:

```
opts=pulumi.ResourceOptions(depends_on=[ dep1, dep2, … ],
```

Le risorse da cui dipende sono salvate nella lista all'interno dell'oggetto *Code*. In questo modo ogni risorsa ha il riferimento alla risorsa da cui dipende, e Pulumi può schedulare la creazione di esse.

Output

L'ultima fase riguarda la creazione del file finale di output. Nella stringa d'esempio nel punto precedente abbiamo i valori *dep1*, *dep2*: questi

sono i nomi delle variabili delle dipendenze. Essendo il file IaC scritto in Python, le variabili presenti nella stringa devono essere dichiarate precedentemente al momento in cui vengono usate.

È stato sviluppato, perciò, un algoritmo che decide l'ordine di scrittura nel file di output in modo che una risorsa possa essere scritta solo se il suo vettore delle dipendenze è vuoto. Il primo oggetto della lista di *Code* sono le righe di codice iniziali, quindi possono essere scritte subito.

Successivamente, un ciclo *while* itera fino a quando la lista *codeVec* non è vuota. Viene utilizzata una lista temporanea (*resource_outputted*) dove vengono salvati i nomi delle risorse già scritte nel file di output. Un ciclo *for* itera su ogni oggetto *Code*: si verifica se nella lista delle dipendenze è presente almeno un nome, in caso affermativo viene verificato se le dipendenze al suo interno sia presente nel vettore *resource_outputted* e in questo caso eliminata dalla lista delle dipendenze.

Un secondo *if* controlla se la lista delle dipendenze è vuota e in caso affermativo le linee di codice di quella risorsa vengono scritte sul file di output, il nome della risorsa aggiunto a *resource_outputted* e l'oggetto eliminato dalla lista delle risorse.

2.4.2 Amazon Web Services (AWS)

Lo sviluppo dello script in Amazon Web Services è stato diviso in quattro funzioni, come nel caso precedente:

- Input: lettura riga per riga, individuazione di ogni risorsa;
- Gestione dipendenze: creazione della stringa con le dipendenze da inserire nel codice;
- Personalizzazione: modifica di alcune linee di codice per rendere il file eseguibile in ogni situazione, correzione di bug;
- Output: creazione del file di output seguendo un algoritmo per avere le dipendenze gestite correttamente.

Input

La funzione di input legge, come nel caso precedente, riga per riga il file

generato dall'import tramite Pulumi. Le operazioni che sono eseguite sono molto simili a quanto fatto con Microsoft Azure: viene riutilizzata la classe creata precedentemente come struttura in cui salvare tutti i dettagli di ogni risorsa e la lista in cui contenere tutti gli oggetti *Code*.

Con un'espressione regolare viene trovata la prima riga di ogni risorsa:

```
re.search(" = aws\.", x) ≠ None and re.search("\(\"", x) ≠ None
```

Viene creato un oggetto *Code* (*codeStruct*) inizializzato con il nome della risorsa e con le liste delle dipendenze e del codice vuote. Viene poi inserita la prima riga nella lista dell'oggetto.

Una seconda condizione verifica se nella riga è presente "`opts=pulumi.ResourceOptions(protect=True)`": nel caso fosse presente, essa viene eliminata.

Le verifiche successive riguardano le dipendenze. Il primo controllo avviene sugli id delle risorse, verificando se nella riga in esame è presente uno dei *values* presenti nella *nameTable* che è stata importata all'inizio dello script: se viene soddisfatta questa condizione oltre ad inserire la riga di codice nell'oggetto *codeStruct*, viene inserito il nome della risorsa da cui dipende nella lista delle dipendenze. Il nome di questa è ottenuto dalla tabella *nameTable*. Allo stesso modo viene aggiornato *codeStruct* se nella riga è presente un id di una risorsa, ovvero una delle *keys* presenti nella tabella *nameTable*.

Se nessuna delle condizioni elencate è soddisfatta significa che in quella riga non ci sono dipendenze e può essere salvata direttamente nella lista di righe di codice in *codeStruct*.

Gestione delle dipendenze

Come nel caso di Azure, in questa funzione viene scansionato ogni oggetto e aggiunta la riga con le dipendenze. Questa viene strutturata in questo modo:

```
opts=pulumi.ResourceOptions(depends_on=[ dep1, dep2, … ],
```

Le risorse da cui dipende sono salvate nella lista all'interno dell'oggetto *Code*. In questo modo ogni risorsa ha il riferimento alla risorsa da cui dipende, e Pulumi può schedulare la creazione di esse.

Personalizzazione

È necessario introdurre alcune modifiche al codice, necessarie a risolvere i problemi descritti nella parte introduttiva causati dal provider *aws-classic* di Pulumi. Alcune risorse vengono importate con alcuni parametri duplicati: il provider classic, in alcuni casi, configura una stessa proprietà con diversi parametri che, durante l'importazione, vengono salvati sul file IaC. A terminale si può notare che sono presenti dei warning: non si tratta di errori poiché Pulumi riesce comunque a portare a termine l'importazione. È necessaria, però, una modifica perché, altrimenti, nel momento in cui il file IaC viene eseguito, Pulumi cerca di configurare la stessa proprietà con più di un parametro: questa operazione non viene permessa per evitare incongruenze tra valori di parametri diversi. In questo caso Pulumi blocca il processo di creazione e scaturisce un errore. In particolare, le modifiche principali sono state effettuate su Instance, Subnet e Network Interface nella funzione *aws_bug_bypass*: nella prima vengono duplicate le informazioni relative alla quantità di core e di thread della CPU, nella seconda quelle relative ad *availability zone*, mentre nella terza ci sono informazioni duplicate per l'IP privato.

Di seguito sono riportati i messaggi di warning riscontrati:

```
aws:ec2:Instance (tesi_Instance):

    warning: One or more imported inputs failed to validate. This is almost certainly a bug in the `aws` provider. The import will still proceed, but you will need to edit the generated code after copying it into your program.

    warning: aws:ec2/instance:Instance resource 'tesi_Instance' has a problem: Conflicting configuration arguments: "cpu_options.0.core_count": conflicts with cpu_core_count. Examine values at 'tesi_Instance.cpuOptions.0.coreCount'.

    warning: aws:ec2/instance:Instance resource 'tesi_Instance' has a problem: Conflicting configuration arguments: "cpu_options.0.threads_per_core": conflicts with cpu_threads_per_core. Examine values at 'tesi_Instance.cpuOptions.0.threadsPerCore'.

    warning: aws:ec2/instance:Instance resource 'tesi_Instance'
```

```
has a problem: Conflicting configuration arguments: “cpu_core_count”: conflicts with cpu_options.0.core_count. Examine values at ‘tesi_Instance.cpuCoreCount’.

    warning: aws:ec2/instance:Instance resource ‘tesi_Instance’ has a problem: Conflicting configuration arguments: “cpu_threads_per_core”: conflicts with cpu_options.0.threads_per_core. Examine values at ‘tesi_Instance.cpuThreadsPerCore’.


aws:ec2:Subnet (tesi_Subnet):

    warning: One or more imported inputs failed to validate. This is almost certainly a bug in the `aws` provider. The import will still proceed, but you will need to edit the generated code after copying it into your program.

    warning: aws:ec2/subnet:Subnet resource ‘tesi_Subnet’ has a problem: Conflicting configuration arguments: “availability_zone_id”: conflicts with availability_zone. Examine values at ‘tesi_Subnet.availabilityZoneId’.

    warning: aws:ec2/subnet:Subnet resource ‘tesi_Subnet’ has a problem: Conflicting configuration arguments: “availability_zone”: conflicts with availability_zone_id. Examine values at ‘tesi_Subnet.availabilityZone’.


aws:ec2:NetworkInterface (tesi_NetworkInterface):

    warning: One or more imported inputs failed to validate. This is almost certainly a bug in the `aws` provider. The import will still proceed, but you will need to edit the generated code after copying it into your program.

    warning: aws:ec2/networkInterface:NetworkInterface resource ‘tesi_NetworkInterface’ has a problem: Conflicting configuration arguments: “private_ips”: conflicts with private_ip_list. Examine values at ‘tesi_NetworkInterface.privateIps’.

    warning: aws:ec2/networkInterface:NetworkInterface resource ‘tesi_NetworkInterface’ has a problem: Conflicting configuration arguments: “private_ip_list”: conflicts with private_ips. Examine values at ‘tesi_NetworkInterface.privateIpLists’.

    warning: aws:ec2/networkInterface:NetworkInterface resource ‘tesi_NetworkInterface’ has a problem: expected interface_type to be one of [“efa” “branch” “trunk”], got interface. Examine values at ‘tesi_NetworkInterface.interfaceType’.
```

Vengono perciò eliminate alcune righe di codice. Per quanto riguarda le Instance EC2 viene mantenuto il parametro *cpu_options* che contiene i parametri *core_count* e *threads_per_core*: le altre due opzioni sono deprecate. Nella dichiarazione della Subnet rimane solo il parametro

availability_zone, perché *availability_zone_id* non è supportato in ogni regione e, quindi, non è sempre presente. Nell'ultimo caso viene mantenuto il parametro *private_ips*, che è sufficiente per il corretto funzionamento.

Con i test effettuati è stato notato che vi è un secondo problema che non viene visualizzato a terminale: durante la fase di import di un Bucket S3, non vengono salvati nel file IaC i tag che la risorsa possiede. È stato quindi necessario reperire le informazioni su di essi in questa fase: si è creato un client boto3 con configurazione personalizzata per impostare la regione in cui il Bucket è presente e, tramite la funzione *get_bucket_tagging*, è stato possibile recuperare i dati necessari, interpretarli e aggiungere una nuova riga nella dichiarazione della risorsa con i tag corretti.

<u>Output</u>

L'ultima fase riguarda la creazione del file finale di output. L'algoritmo alla base è molto simile a quello precedente: si scrive la risorsa nel file di output solo quando tutte le sue dipendenze sono già state scritte. Inoltre, non è più possibile lasciare nel codice le stringhe relative a id delle risorse in quanto queste sono composte da una stringa casuale, sempre diversa ad ogni creazione della risorsa. Perciò, per avere un blueprint che dia la possibilità di essere utilizzato più volte, è necessario sostituire le stringhe con *nome_variabile.id*, per permettere a Pulumi di acquisire direttamente l'id corretto. Per far comprendere a Pulumi le dipendenze tra risorse può bastare questa sostituzione, ma è stata comunque inserita la stringa con l'opzione *depends_on* (descritta nel paragrafo della gestione delle dipendenze) per permettere una più facile lettura delle dipendenze agli sviluppatori che gestiranno i file IaC.

Dopo la scrittura del primo oggetto della lista *codeVec* con le prime linee di codice Python, inizia il ciclo *while* che termina quando la lista *codeVec* è vuota. Viene utilizzata una lista temporanea (*resource_outputted*) dove vengono salvati i nomi delle risorse già scritte nel file di output. Un ciclo *for* itera su ogni oggetto *Code*: si controlla che la lista delle dipendenze non sia vuota e, in caso essa presenti dei nomi, viene verificato se questi sono presenti nel vettore *resource_outputted*.

In questo caso eliminate dalla lista delle dipendenze.

Un secondo *if* controlla se la lista delle dipendenze è vuota e in caso affermativo le linee di codice di quella risorsa vengono scritte sul file di output, il nome della risorsa aggiunto a *resource_outputted* e l'oggetto eliminato dalla lista delle risorse. Durante la scrittura delle righe di codice viene verificata la presenza di una stringa contenente un id di una risorsa, se è presente viene sostituito con *nome_varibile.id*. Unica eccezione per le risorse di tipo *Bucket* in cui è presente il parametro *bucket* in cui vi è segnato il suo stesso id: essendo la stringa, l'id di sé stesso non causa nessun problema, perciò non viene effettuata alcuna modifica. Questo inoltre causerebbe una referenza a sé stesso causando un errore nell'esecuzione dello script.

2.5 Customization

L'esecuzione dello script di parsing crea un file IaC che è eseguibile anche nel caso in cui le risorse non sono online. Se questo viene inserito in un progetto Pulumi e viene lanciato il comando *pulumi up*, vengono create tutte le risorse evitando i problemi riscontrati con il file IaC creato automaticamente con l'importazione.

Questo file è chiamato blueprint poiché è un punto di partenza: può essere modificato a dovere per soddisfare diverse esigenze. Per esempio, consideriamo che le risorse del file IaC definiscono un ambiente in cui sono eseguiti diverse applicazioni: uno sviluppatore potrebbe aver bisogno dello stesso ambiente per eseguire dei test, senza, però, causare problemi, fallimenti o rallentamenti nell'ambiente di produzione. Avendo un file IaC che ne descrive l'ambiente può modificare qualche parametro e duplicare tutte le risorse per eseguire i suoi test. Un secondo esempio è quello di considerare il caso di Disaster Recovery: avendo un file IaC con le risorse importate, nel caso queste non fossero più disponibili per problemi esterni, è possibile modificare la regione su cui le risorse devono essere distribuite e ricreare lo stesso ambiente in una seconda regione.

Sono stati sviluppati due script che soddisfano lo scenario del secondo esempio.

2.5.1 Microsoft Azure

È stato sviluppato un programma Python che prendesse in input un file IaC, il nome della regione in cui vi erano le risorse e il nome della nuova regione. Lo script, quando viene eseguito, legge riga per riga il file in input e sostituisce il nome della vecchia regione con quello della nuova. Il risultato finale è un file con le risorse configurate per essere create nella nuova regione.

2.5.2 Amazon Web Services (AWS)

Con AWS lo script è stato sviluppato in modo differente: il file in output non avrà la nuova regione come parametro in ogni risorsa ma vengono

solo eliminate le proprietà che fanno riferimento a zone e regioni, in quanto è pensato per essere eseguito in un progetto Pulumi che necessita della configurazione della regione AWS con il comando:

```
pulumi config set aws:region ${REGION_NAME}
```

Con l'esecuzione di alcuni test con il file IaC personalizzato si è riscontrato un errore legato all'AMI (Amazon Machine Images) delle risorse di tipo Instance: l'id dell'immagine non viene trovato dopo il cambio della regione. Dopo aver analizzato varie AMI uguali in regioni diverse si è notato che gli id differiscono; perciò, è necessario trovare l'id corretto per la nuova regione e sostituirlo.

Per la risoluzione di questo problema è stato necessario capire quale fosse la vecchia regione in quanto questa non viene passata in input come nel caso precedente: è stata quindi recuperata dal nome dell'*availability_zone* dell'Instance. Viene creata una nuova *Session* (come nella fase di Discovery) utilizzando come regione quella appena ricavata. Dopo aver creato un client EC2, la funzione *describe_images*, avente come parametro l'id dell'AMI, ricava tutti i dettagli dell'immagine.

Successivamente, una nuova *Session* viene creata, in questo caso però con la nuova regione. Viene nuovamente richiamata la funzione *describe_images*, senza inserire l'id della vecchia immagine ma piuttosto alcune informazioni ricavate, come architettura, tipo, stato e descrizione. Questi sono utilizzati come filtro e viene ricavata un'unica AMI avente un id differente dal precedente ma compatibile con la nuova regione. L'id viene sostituito dal vecchio e viene salvata la riga aggiornata nel file di output.

2.6 Automation

Tutti gli script sviluppati possono essere eseguiti in serie ad altre operazioni per soddisfare esigenze diverse che un cliente può avere. Per rendere il processo ancora più ottimizzato e veloce possono essere create delle automazioni, che eseguono una serie di operazioni, tra cui l'esecuzione degli script, per ottenere il risultato voluto dal cliente. Sono stati immaginati tre diversi scenari che si focalizzano su tre diverse necessità che un'azienda o un cliente può avere. Basandosi su questi sono state create delle pipeline in Jenkins, che, dopo aver inserito i parametri necessari in input, automatizzano le operazioni.

Le automazioni create verranno spiegate nel prossimo capitolo dopo aver descritto lo scenario preso in esame.

Capitolo 3

Use cases

Il progetto sviluppato comprende le varie fasi descritte nel capitolo precedente. Queste spesso possono essere eseguite in serie tra loro oppure singolarmente per soddisfare esigenze differenti. Sono stati quindi immaginati tre scenari principali: questi indicano una situazione particolare in cui un cliente può trarre vantaggi usando gli script sviluppati.

Per ognuno di essi sono state sviluppate delle pipeline di Jenkins che soddisfano gli obiettivi di questi scenari in modo completamente automatico.

3.1 Requisiti

Per permettere un corretto funzionamento di ogni automazione sono necessari alcuni programmi installati e la configurazione di alcune impostazioni. I programmi che devono essere necessariamente installati nel terminale per eseguire le automazioni sono:

- Pulumi versione 3.103.1 o successive;
- Jenkins;
  - Git plugin;
  - Active Choice plugin;
  - Azure plugin;
  - SshAgent plugin;
- Python3 versione 3.10.12 o successive;
- Git;
- Configurazione *Service Principal* per consentire l'autenticazione in Azure.

Inoltre, in Jenkins è necessario impostare le credenziali di accesso nella sezione *Credentials*, per salvarle in modo sicuro. Le credenziali necessarie sono il token di accesso di Pulumi, la chiave SSH per permettere la gestione delle repository git utilizzate, le credenziali di accesso di Microsoft Azure e/o quelle di Amazon Web Services.

L'utilizzo di repository Git ha permesso un'organizzazione efficace dei file IaC e dei progetti che sono stati creati. È stato progettato un repository per ogni use case che si è valutato, e uno per tutti gli script sviluppati. Ognuno di questi presenta al suo interno due cartelle per differenziare i file e progetti di Microsoft Azure da quelli di AWS. Inoltre, per quanto riguarda Azure vi è al suo interno un'ulteriore suddivisione legata alla subscription in cui risiedono le risorse. Queste sono le caratteristiche comuni a tutti i repository: in ogni caso d'uso verrà poi descritta l'organizzazione interna.

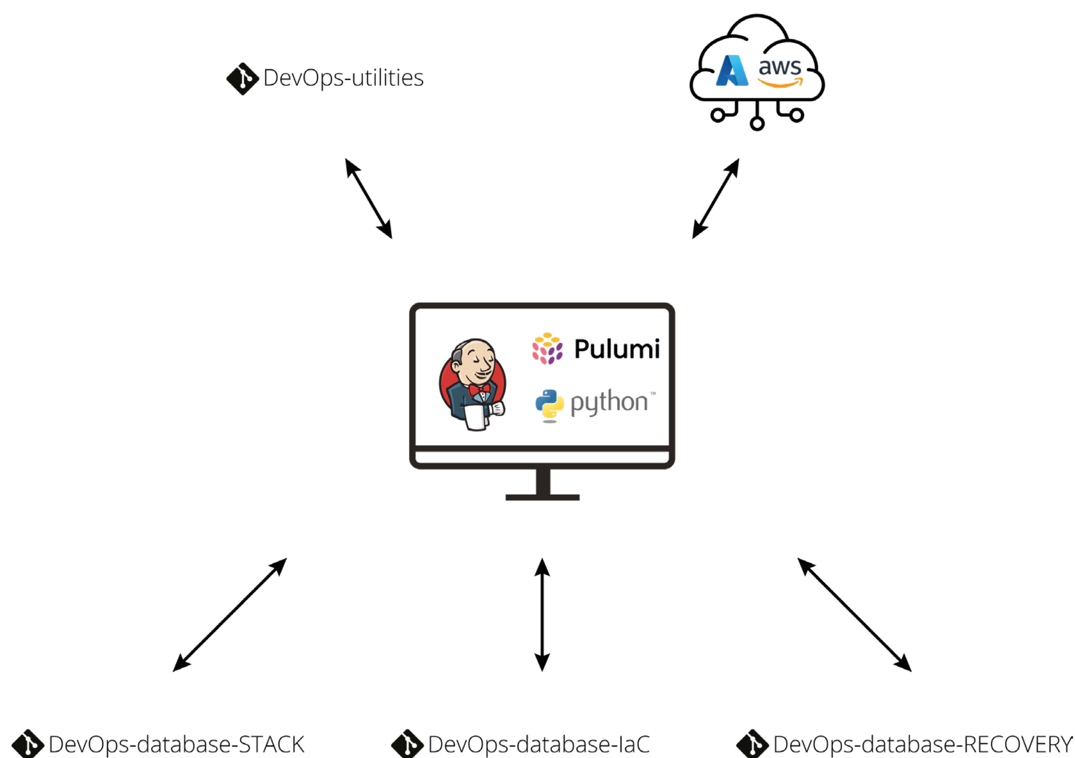

Figura 8 - Comunicazione con repository

3.2 Importazione delle risorse in un progetto Pulumi per la gestione tramite IaC

Si immagina la presenza di un'azienda o un cliente che possiede un'infrastruttura cloud, più o meno complessa, gestita però tramite il portale del cloud provider scelto. L'azienda ha la necessità di ottimizzare la gestione delle risorse tramite l'approccio Infrastructure as Code per apportare tutti i vantaggi che sono stati descritti nel capitolo iniziale.

Questo permetterebbe anche di aver un controllo migliore sulla creazione delle risorse, ad esempio, avendo la possibilità di accedere al file IaC in un repository, chi necessita di una nuova risorsa può modificare il file e creare una *Merge Request* che deve essere approvata. L'approvazione può essere data solo dal personale che ha le conoscenze necessarie per capire se la risorsa può essere creata senza causare problemi. Questo permette anche di avere una cronologia delle modifiche effettuate e la possibilità di rollback.

3.2.1 Pipeline

La pipeline Jenkins ha come obiettivo la creazione di un progetto Pulumi con le risorse online di uno specifico Cloud Service Provider. La pipeline viene eseguita in automatico dopo aver scelto alcuni parametri in input necessari per il corretto funzionamento.

L'implementazione della selezione dei parametri è stata realizzata attraverso l'uso del plugin Active Choice di Jenkins, il quale consente di variare le opzioni dei parametri in funzione della selezione precedente.

La prima scelta è caratterizzata da un checkbox relativo al cloud provider da cui si vogliono importare le risorse: Microsoft Azure o Amazon Web Services. Successivamente, viene scelto con un checkbox quale filtro utilizzare: per AWS la scelta è obbligatoriamente quella della coppia chiave-valore di un tag, altrimenti, con Azure è possibile scegliere il filtro sul Resource Group. A questo punto è necessario inserire i valori nei campi di testo: per entrambi i casi sono necessari i nomi del progetto Pulumi da creare e il nome del relativo stack; nel caso si tratti di AWS sono necessari anche i dati relativi alla regione, la chiave e il

valore del tag, in caso contrario (Azure) il nome del resource group o la coppia chiave-valore del tag.

CloudProvider
Select the correct cloud provider
AWS
AZURE

Filter
Select the filter type
ResourceGroup
Tag

InputParameter
Add parameters
TagKey
TagValue
PulumiProjectName
PulumiStackName

Figura 9 - Parametri input primo scenario

Se la pipeline si conclude senza errori, si ottiene nel repository di database un progetto Pulumi con tutte le risorse importate con il file IaC per la loro gestione.

Le pipeline di Jenkins vengono divise in vari stage: in caso di fallimento di uno di questi, viene bloccata l'esecuzione degli stage successivi e visualizzato nei log il problema riscontrato.

3.2.2 Stage 1: Parameter check

L'esecuzione è organizzata in cinque stage, di cui due dipende dalla scelta dal provider. Il primo stage, chiamato *PARAMETER CHECK*, contiene le operazioni necessarie per verificare i parametri che sono stati inseriti all'avvio della pipeline. È necessario controllare che i parametri *CloudProvider* e *Filter* non siano vuoti: sono selezionati tramite checkbox, quindi non è necessario controllare la correttezza del valore scelto. I dati da inserire nelle textbox sono salvati in una singola

variabile che è stata denominata *InputParameter*, la quale contiene tutte le informazioni concatenate e separate da una virgola. Per questo motivo, la funzione *split(",")* è utilizzata per dividere una stringa iniziale in diverse parti. Viene, successivamente, effettuato un controllo sulla presenza del valore e non sul suo contenuto. Infine, in una variabile vengono salvate data e ora dell'esecuzione, per permettere di salvare queste informazioni nel nome dei file che vengono creati. In questo modo, è possibile avere i file relativi a importazioni delle medesime risorse in momenti diversi.

3.2.3 Stage 2: Clone

La seconda fase, *CLONE*, crea le cartelle e scarica i file necessari per le prossime operazioni: è necessario avere installato in Jenkins il plugin Git e avere configurato le credenziali per la lettura e scrittura dei repository che vengono utilizzati, in quanto questi sono privati. Vengono create le cartelle *utilities* e *database-stack* in cui vengono scaricati i rispettivi repository:

- *DevOps-utilities*
- *DevOps-database-STACK*

3.2.4 Stage 3: Azure import

Le due fasi successive sono molto simili tra loro e dipendono dal tipo di Cloud Provider: viene eseguita solo quella relativa al provider selezionato. In entrambe sono necessarie le credenziali di Pulumi e del Cloud Provider, che devono essere precedentemente configurate nella sezione Credentials di Jenkins: viene utilizzato il plugin Azure per il CSP di Microsoft, e le credenziali standard per AWS.

Considerando lo stage *AZURE IMPORT*, innanzitutto viene controllato qual è il filtro utilizzato per permettere l'utilizzo dello script corretto. Viene eseguito uno script shell chiamando la funzione *sh*. Con il comando export tutte le credenziali vengono salvate nelle variabili della shell. Gli script si trovano nella cartella del Cloud Provider all'interno di *utilities*, cartella creata durante lo stage *CLONE*.

All'interno della directory, viene eseguito lo script di discovery: *discovery_rg.py* se il filtro è il resource group, altrimenti *discovery_tag.py*. I due comandi completi con i relativi parametri sono:

```
python3 discovery_rg.py -s ${AZURE_SUBSCRIPTION_ID} -r ${RESOURCE_
GROUP_NAME}
python3 discovery_tag.py -s ${AZURE_SUBSCRIPTION_ID} -k ${TAGKEY}
-v ${TAGVALUE}
```

Questi creano un file *resources.json* con l'elenco delle risorse da importare. Per quest'operazione, dopo il cambio di directory nella cartella *PULUMI* generata dalla fase *CLONE*, è necessario creare un progetto Pulumi con il comando:

```
pulumi new azure-python --name ${PROJECT} --stack ${STACK} -yes
```

e successivamente:

```
pulumi import --skip-preview -f resources.json -o __main__.py
```

Dopo che l'esecuzione di questi due comandi termina, si trovano, nella cartella *PULUMI*, tutti i file del progetto che sono stati creati, compreso il file IaC generato automaticamente. Per rendere possibile la gestione del progetto da un altro terminale è necessario esportare le informazioni relative allo stack, viene, quindi, eseguito:

```
pulumi stack export --stack ${STACK} --file stack.json
```

Viene utilizzato ora un secondo repository come database. In esso saranno contenuti tutti i file relativi ai progetti importati. Viene quindi creata una cartella specifica per l'importazione corrente che deve rispettare un pattern facilmente riconoscibile:

```
${CloudProvider} / ${AZURE_SUBSCRIPTION_ID} / ${CloudProvider}--
${RESOURCE_GROUP_NAME}--${IMPORT_DATE}
```

Nel caso in cui il filtro è quello relativo al tag non ci sarà `${RESOURCE_GROUP_NAME}` ma `${TAGKEY}:${TAGVALUE}`.

All'interno di questa cartella sono presenti tutti i file del progetto, esclusi quelli dell'ambiente Python che verranno rigenerati durante l'importazione dello stack in un altro terminale.

3.2.5 Stage 4: AWS import

La sequenza di operazioni nel caso di AWS è molto simile a quella di Azure: lo stage viene denominato *AWS IMPORT*. Gli script si trovano nella cartella AWS: all'interno viene eseguito il comando:

```
python3 discovery.py -k ${TAGKEY} -v ${TAGVALUE} -i ${AWS_ACCESS_
KEY_ID} -s ${AWS_SECRET_ACCESS_KEY} -r ${REGION} -t ${AWS_SESSION_
TOKEN}
```

La creazione del progetto ora richiede il provider delle risorse Pulumi AWS Classic; quindi, la creazione del progetto viene completata dal comando:

```
pulumi new aws-python --name ${PROJECT} --stack ${STACK} --yes
```

Prima dell'importazione delle risorse con il comando *pulumi import*, identico al caso di Azure, è necessario configurare la regione AWS:

```
pulumi config set aws:region ${REGION}
```

Successivamente all'importazione delle risorse, come nel caso precedente viene esportato lo stack.

Come riportato nel capitolo 2, il provider AWS Classic presenta alcune problematiche che rendono impossibile l'utilizzo del file IaC generato senza apportarci alcune modifiche: è perciò necessario eseguire lo script di parsing del file. Viene, perciò, lanciato il comando:

```
python3 formatting.py -r ${REGION}
```

Il file IaC, rinominato in *input.py*, viene trasformato e corretto per permetterne l'esecuzione. Viene creata ora la cartella specifica per il progetto, la quale ha lo stesso pattern descritto nel paragrafo precedente in cui vi sono tutti i file necessari al progetto.

3.2.6 Stage 5: Upload

I file del progetto sono pronti per essere caricati nel repository di database: grazie ai comandi Git, viene aggiornato il repository di database di questo scenario, ovvero DevOps-database-STACK.

Vengono eseguiti in serie i seguenti comandi all'interno della cartella database-stack, nella quale nella fase di CLONE è stato fatto un pull del repository:

```
git add .
git commit -m 'Import ${CloudProvider} stack database updated'
git push git@<git_URL>/DevOps-database-STACK.git
```

3.2.7 Ultime operazioni

A fine esecuzione degli stage, viene sempre eseguita la funzione *cleanWs()* che elimina i file che sono stati generati durante l'esecuzione della pipeline.

A questo punto tutti i file sono stati salvati su un repository: il cliente o l'azienda può gestire il progetto da qualsiasi computer in cui vi sono installati Python3 e Pulumi. La cartella del progetto in questione può essere scaricata con *git pull* e, all'interno di esse, viene eseguito il comando di importazione dello stack per rendere il progetto funzionante:

```
pulumi stack import --file stack.json
```

3.3 Personalizzazione del file IaC per la creazione di una nuova infrastruttura: blueprint

La possibilità di avere un blueprint IaC con un set di risorse particolari, scelte mediante un filtro, apre a diversi scenari di utilizzo: uno di questi è la possibilità di distribuire le risorse del file IaC più volte per avere ambienti uguali dove, ad esempio, eseguire dei test senza causare problemi all'ambiente di produzione, andando a modificare solo qualche linea di codice. Un secondo esempio può essere quello che andremo a sviluppare nel capitolo 3.4, ovvero avere un file personalizzato per riprodurre le risorse in regioni differenti da quella in cui sono state importate: può essere utile in caso di Disaster Recovery.

L'obiettivo finale di questo scenario è quello di avere un database con diversi file IaC, i quali possono essere scaricati e personalizzati. Non è necessario avere quindi un progetto Pulumi con lo stato delle risorse salvato in locale o sul cloud.

3.3.1 Pipeline

L'esecuzione di questa pipeline richiede in input alcuni parametri: sono gli stessi del caso precedente ma senza quelli relativi al progetto e allo stack Pulumi. Oltre ai parametri *CloudProvider* e *Filter*, sono necessari: per Azure, il nome del resource group o un tag specifico, per AWS, solo la coppia chiave-valore del tag.

3.3.2 Stage 1: Parameter check

La verifica sui parametri, nello stage di *PARAMETER CHECK*, è simile al caso precedente, a differenza dei dati relativi al progetto e stack Pulumi, che in questo scenario non sono necessari.

3.3.3 Stage 2: Clone

La seconda fase, ovvero quella di *CLONE*, crea le cartelle *utilities* e *database-iac* e scarica i file presenti nei rispettivi repository *DevOps-utilities* e *DevOps-database-IaC*.

3.3.4 Stage 3: Azure import

Sono due gli stage d'importazione, uno per Microsoft Azure e uno per Amazon Web Services. Lo stage *AZURE IMPORT* esegue uno script shell, nel quale a sua volta vengono eseguiti una serie di comandi: i primi export servono a salvare in variabili della shell le credenziali accesso sia di Azure che di Pulumi, mentre successivamente avviene un cambio di directory, per raggiungere la cartella in cui vi sono gli script. All'interno di essa si eseguono le operazioni per ricavare il file IaC. Il primo passo è quello di ricavare il file JSON con i dettagli delle risorse, per cui viene eseguito lo script di discovery: viene eseguito solo quello relativo al filtro scelto, resource group o tag, utilizzando gli stessi comandi del punto precedente.

A differenza del primo scenario, non vengono richiesti in input i parametri relativi al progetto, poiché lo scopo è quello di ottenere solamente il file IaC, ma per utilizzare il comando di importazione è richiesto un progetto e uno stack, i quali verranno poi eliminati. Nel repository delle utilities è già presente un file *Pulumi.yaml* con le proprietà del progetto: questo permette di evitare di utilizzare il comando *pulumi new*, ma solo di inizializzare lo stack con:

```
pulumi stack init dev
```

A questo punto è possibile eseguire il comando d'importazione:

```
pulumi import --skip-preview -f resources.json -o input.py
```

Il file *input.py* è il file IaC generato automaticamente, che deve essere necessariamente usato in input allo script di parsing, per generare il blueprint. Il comando da eseguire è:

```
python3 formatting.py
```

A questo punto non resta che rinominare il file di output dallo script di

parsing con un pattern specifico e copiarlo nel database dei file IaC. Il nome del file è rinominato con la seguente logica:

```
${CloudProvider}--${RESOURCE_GROUP_NAME}--${IMPORT_DATE}.py
```

La cartella di destinazione è nella directory in cui è stato clonato il database all'interno della cartella del Cloud Provider, in questo caso Azure, e all'interno della cartella della subscription in cui vi sono le risorse.

3.3.5 Stage 4: AWS import

Come nel caso di Azure, non è necessario richiedere i dati relativi al progetto in quanto questo verrà poi eliminato. Viene eseguito, quindi, uno script shell in cui vengono esportate al suo interno le credenziali di accesso di Pulumi e AWS, e poi eseguite una serie di operazioni molto simili al caso di Microsoft Azure.

Nella cartella delle utilities di AWS, viene eseguito lo script di discovery delle risorse (stesso comando del capitolo 3.2.5); successivamente viene inizializzato uno stack, in quanto è già presente nella directory il file con i dettagli del progetto *Pulumi.yaml*. Con Amazon Web Services è richiesta la configurazione della regione in cui vi sono le risorse da importare: il comando `pulumi config set aws:region ${REGION}` esegue questa operazione.

La configurazione del progetto è stata eseguita e la creazione dei file è stata completata; perciò, è possibile continuare con il comando d'importazione:

```
pulumi import --skip-preview -f resources.json -o input.py
```

Il file *input.py* necessita di essere formattato attraverso lo script di parsing per creare il blueprint finale. Il comando da eseguire è:

```
python3 formatting.py -r ${REGION}
```

La regione nello script di parsing è necessaria per la corretta esecuzione di alcune funzioni della libreria boto3 utilizzata nello script.

La creazione del blueprint è completata; questo viene copiato nella cartella AWS del repository di database clonata e rinominato seguendo lo schema:

```
${CloudProvider}--${TAGKEY}:${TAGVALUE}--${REGION}--${IMPORT_
DATE}
```

3.3.6 Stage 5: Upload

L'unico file che necessita di essere salvato in un repository è il blueprint creato, i file che sono stati generati con tutte le altre operazioni verranno eliminati. Vengono quindi eseguiti in serie nella directory del database i seguenti comandi:

```
git add .
git commit -m 'Import ${CloudProvider} IaC database updated'
git push git@<git_URL>/DevOps-database-IaC.git
```

3.3.7 Ultime operazioni

A fine esecuzione degli stage, viene sempre eseguita la funzione *cleanWs()*, la quale elimina i file che sono stati generati durante l'esecuzione dei vari stage. Inoltre, coma spiegato precedentemente, il progetto Pulumi creato è stato usato solo per la generazione del file: viene quindi forzata la sua eliminazione.

I comandi differiscono in base al Cloud Provider utilizzato:

Azure:
```
sh("pulumi stack rm <pulumi-org>/azure_import_jenkins/dev
-y --force")
```

AWS:
```
sh("pulumi stack rm <pulumi-org>/aws_import_jenkins/dev -y
--force")
```

Quando è necessario un blueprint importato precedentemente, è possibile consultare il repository e trovare il file IaC necessario valutando il Cloud Provider, i filtri che sono stati utilizzati e la data d’importazione. Questo avviene nello scenario proposto nel capitolo successivo: si nota che, in quel caso, viene cercato un file di input nel repository di database utilizzato in questo scenario e, se trovato, verrà personalizzato per soddisfare i requisiti del caso d’uso in esame, ovvero di disaster recovery.

3.4 Preparazione di un file IaC pronto per essere eseguito in una regione differente in caso di Disaster Recovery

Il risultato del punto precedente rende possibile l'utilizzo del file in scenari diversi; l'esecuzione di questo file è indipendente dallo stato delle risorse. Il file quindi risulta personalizzabile, in modi diversi, per rispondere alle esigenze del cliente. È stato considerato uno dei vari scenari disponibili: modificare la regione in cui le risorse vengono distribuite. Questo scenario permette di modellare il file per renderlo utile in caso di problemi nella regione in cui le risorse sono online, ovvero in caso di Disaster Recovery. Con l'esecuzione di una pipeline creata appositamente per questo scenario è possibile trovare, se esiste, il file IaC con le risorse precedentemente importate dalla regione che per qualche problema non è più disponibile, sceglierne una nuova e creare l'infrastruttura nella regione aggiornata.

3.4.1 Pipeline

La pipeline che è stata sviluppata, all'inizio della sua esecuzione, necessita l'inserimento di alcuni parametri. I primi sono relativi al Cloud Provider e al filtro, come nei casi precedenti. Successivamente viene richiesto l'inserimento della nuova regione per entrambi i provider, e il nome della regione da sostituire solo nel caso di Microsoft Azure; infatti, potrebbero esserci risorse di diverse regioni all'interno dello stesso file IaC, a differenza di AWS che, per come è stato sviluppato l'algoritmo di discovery, ne permette una per file.

Viene, inoltre, lasciata la possibilità di scegliere se le nuove risorse debbano essere distribuite nella nuova regione, oppure se creare solamente il file IaC pronto per essere usato in un secondo momento. Una checkbox con nome *DEPLOY* indica questa scelta, se la risposta è affermativa vengono ulteriormente richiesti i nomi del progetto e dello stack Pulumi che verranno creati.

CloudProvider
Select the correct cloud provider
AWS
AZURE

Filter
Select the filter type
ResourceGroup
Tag

InputParameter
Add parameters
TagKey
TagValue
NewRegion

Deploy
Do you want deploy in the new region?
YES
NO

Pulumi
Pulumi infos
ProjectName
StackName

Figura 10 - Parametri input terzo scenario

In questo scenario non verranno utilizzati gli script di discovery e di parsing ma solamente quello di customization creato ad hoc per questo scenario. La pipeline, però, è sviluppata a partire dal repository in cui sono stati salvati i blueprint del secondo scenario: le pipeline sono pensate per essere utilizzate in sequenza, utilizzando i repository come database.

3.4.2 Stage 1: Parameter check

I parametri scelti all'avvio della pipeline vengono verificati, come avviene anche negli scenari precedenti: viene controllato che non ci siano valori vuoti.

3.4.3 Stage 2: Clone

Lo stage di *CLONE* prepara le tre directory in cui clonare i seguenti repository:

- *DevOps-utilities*: contenente gli script e i file per l'esecuzione;
- *DevOps.database-IaC*: contenente i blueprint creati nel secondo scenario;
- *DevOps-database-RECOVERY*: in cui salvare i file IaC modificati.

3.4.4 Stage 3: Azure recovery

In questo stage viene personalizzato il file quando viene scelto il cloud provider Microsoft Azure. I parametri relativi al filtro sono necessari in quanto viene cercato nel database dei blueprint il file IaC che soddisfa le condizioni scelte: nome del resource group o tag. Queste informazioni sono presenti nel nome del file, compresa la data di acquisizione: si sceglie, quindi, dai file che soddisfano il filtro, quello con data e ora più recente.

Se precedentemente il file non è stato creato, la pipeline stampa un messaggio di errore e non esegue le operazioni successive. In caso il file fosse presente, viene copiato nella cartella delle utilities e viene eseguito lo script di personalizzazione con il comando:

```
python3 change_region.py -o ${OLDREGION} -r ${NEWREGION}
```

Il file di output di questo script viene copiato nella cartella in cui è stato clonato il repository di recovery e rinominato seguendo il pattern:

```
${CloudProvider}--${RESOURCE_GROUP_NAME}--from:${OLDREGION}-to:${NEWREGION}--${IMPORT_DATE}.py
```

Nel caso il filtro scelto consiste nella coppia chiave-valore di un tag, `${RESOURCE_GROUP_NAME}` è sostituito da `${TAGKEY}:${TAGVALUE}`.

3.4.5 Stage 4: AWS recovery

Le operazioni che sono state sviluppate per il cloud provider di Amazon sono molto simili al caso precedente: lo script di parsing presenta

alcune modifiche rispetto a quello per Azure, come descritto nel capitolo 2.5.2.

Viene ricercato il file nel repository di database dello scenario precedente, selezionando quello più recente e che soddisfa la chiave e il valore del tag scelto in input. Nel caso in cui il file non venisse trovato a causa di una mancata importazione, la pipeline non svolgerà le operazioni di personalizzazione.

Se il file viene trovato, esso viene utilizzato come input del programma *change_region.py*: l'output viene copiato nella cartella di recovery e rinominato:

```
${CloudProvider}--${TAGKEY}:${TAGVALUE}--${NEWREGION}--${IMPORT_
DATE}
```

3.4.6 Stage 5: Azure deploy

Questo stage viene eseguito solo quando si sceglie di distribuire le risorse nella nuova regione con Azure durante l'avvio della pipeline, nel momento in cui si inseriscono i parametri. In questo caso, viene creato un progetto Pulumi in una cartella vuota, con il comando:

```
pulumi new azure-python --name ${PROJECT} --stack ${STACK} --yes
```

Viene copiato e sovrascritto al suo interno il file *__main__.py* con quello appena creato, viene eseguito il file IaC e, infine, vengono create le risorse nella nuova regione grazie all'esecuzione di:

```
pulumi up --skip-preview --yes
```

Per permettere la gestione di queste risorse da altri terminali è necessario esportare lo stack:

```
pulumi stack export --stack ${STACK} --file stack.json
```

Il repository di recovery ha al suo interno la cartella di Azure con le sottocartelle con le varie subscription: all’interno di esse saranno salvati i file IaC creati. In questo scenario ed è presente un ulteriore cartella, *pulumiStack*, contente le sottocartelle relative ai progetti con le risorse già distribuite. Questo permette di gestire i relativi progetti in terminali diversi da quello in cui è stata eseguita la pipeline.

3.4.7 Stage 6: AWS deploy

Nel medesimo modo vengono eseguite le operazioni in Amazon Web Services: in questo caso, sarà utilizzato *aws-classic* come provider Pulumi e verrà configurata la regione con:

```
pulumi config set aws:region ${NEWREGION}
```

Le risorse verranno distribuite nella regione selezionata con il comando:

```
pulumi up --skip-preview --yes
```

Lo stack viene esportato, con lo stesso comando eseguito nel caso di Azure. Il file IaC viene copiato nella cartella AWS del repository di recovery e il progetto nella cartella *pulumiStack*.

3.4.8 Stage 7: Upload

I file che sono stati creati nelle fasi precedenti e che necessitano di essere salvati nei repository sono tutti relativi a quello di recovery. Infatti, vengono eseguiti i seguenti comandi nella cartella di recovery:

```
git add .
git commit -m ‘Import ${CloudProvider} recovery database updated’
git push git@<git_URL>/DevOps-database-RECOVERY.git
```

3.4.9 Ultime operazioni

A fine esecuzione degli stage, viene sempre eseguita la funzione *cleanWs()* che elimina i file che sono stati generati durante l'esecuzione della pipeline.

Nel repository di recovery sono, a questo punto, presenti i file IaC con la regione aggiornata e/o lo stack del progetto esportato.

Nel secondo caso è possibile scaricare la cartella relativa al progetto su un terminale con Python3 e Pulumi installati e gestire le risorse tramite Infrastructure as Code dopo l'esecuzione del comando:

```
pulumi stack import --file stack.json
```

Capitolo 4

Validazione

I test sono stati eseguiti, per valutare le prestazioni e i tempi di esecuzione, sia sui singoli script sviluppati sia sulle pipeline create. Nel primo caso, quando eseguo i test su uno script Python, l'ambiente virtuale (contenuto nella cartella *venv*) è già stato creato in precedenza, mentre quando avvio i test su una pipeline, l'ambiente virtuale viene creato a partire dal file *requirement.txt*.

L'esecuzione e il calcolo dei tempi sono effettuati in un ambiente virtualizzato. I componenti principali della macchina fisica sono:

- Processore: Intel i7-7500U, dual-core;
- RAM: 16GB;
- OS: Windows 10 64bit.

Nella macchina virtuale è installato il sistema operativo Ubuntu 20.04 ed è stata configurata con 2 CPU e 6GB di RAM.

Per conoscere i tempi di esecuzione degli script è stato utilizzato il comando *time* inserito prima della chiamata dello script da terminale.

Le risorse con cui sono stati eseguiti i vari calcoli sono quelle descritte nei capitoli precedenti: in totale ci sono sei risorse per Microsoft Azure e cinque per Amazon Web Services. Sono state prese in esame dieci esecuzioni per ogni tipologia di test.

4.1 Tempi di esecuzione script

<u>Discovery</u>

In questo script vengono trovate le risorse online che soddisfano il filtro selezionato in input e viene creato il file con le informazioni necessarie per la fase successiva. Lo script Python viene eseguito da terminale

dieci volte per ogni Cloud Provider: le risorse non cambiano di test in test. La media di esecuzione del programma è di 2.39 secondi per Microsoft Azure, mentre per AWS è di 0.80 secondi.

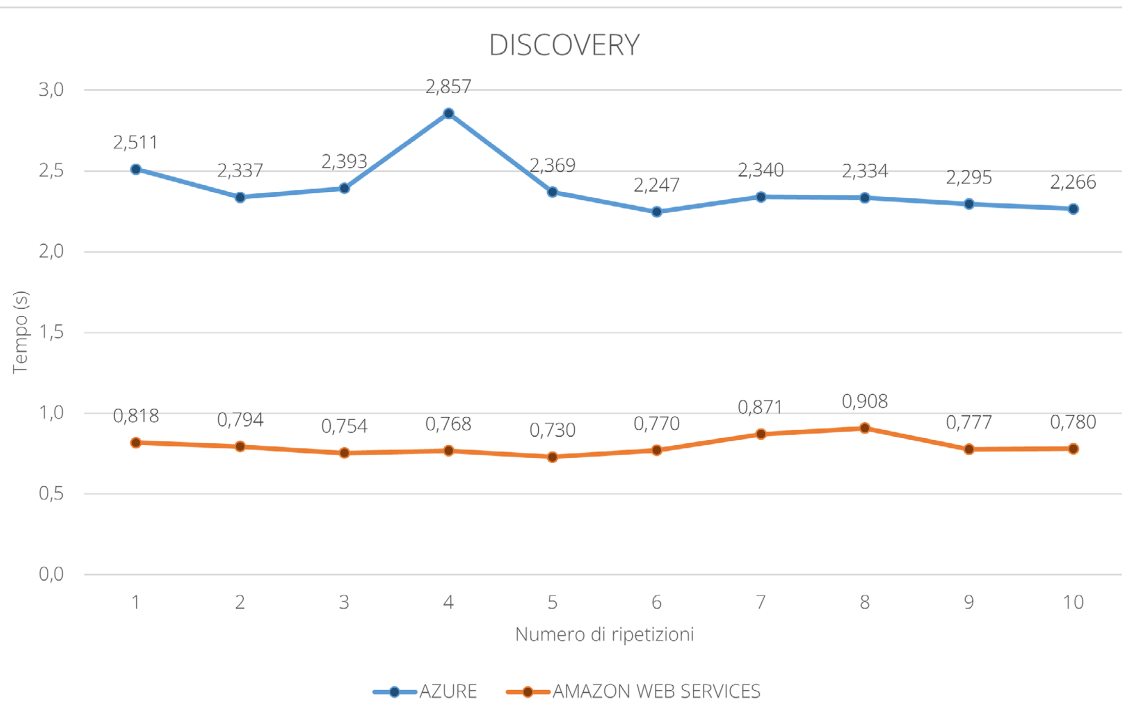

Figura 11 - Tempi di esecuzione script di discovery

L'algoritmo sviluppato nella fase di discovery è molto simile per entrambi i Cloud Provider: la differenza di tempo di esecuzione dei due script è dovuta al tempo necessario per gestire le richieste da parte dei due Cloud Service Provider.

Import

In questa fase le risorse che sono elencate nel file creato dalla fase di discovery vengono importate in un stack Pulumi e viene generato il file IaC in modo automatico. Viene ora utilizzata la funzionalità di importazione di Pulumi con le medesime risorse, utilizzando il file creato nella fase precedente. I tempi di importazione medi sono 33,90 per Microsoft Azure e 15,29 per Amazon Web Services. Si può notare a terminale che il comando d'importazione dopo 8 secondi circa (per entrambi i casi) trova tutte le risorse che sono descritte nel file: nel tempo rimanente Pulumi deve recuperare i dettagli delle risorse online

e creare il file di output. Notiamo che anche in questo caso i tempi per Azure sono più lunghi rispetto a quelli per AWS: il motivo è lo stesso del caso precedente.

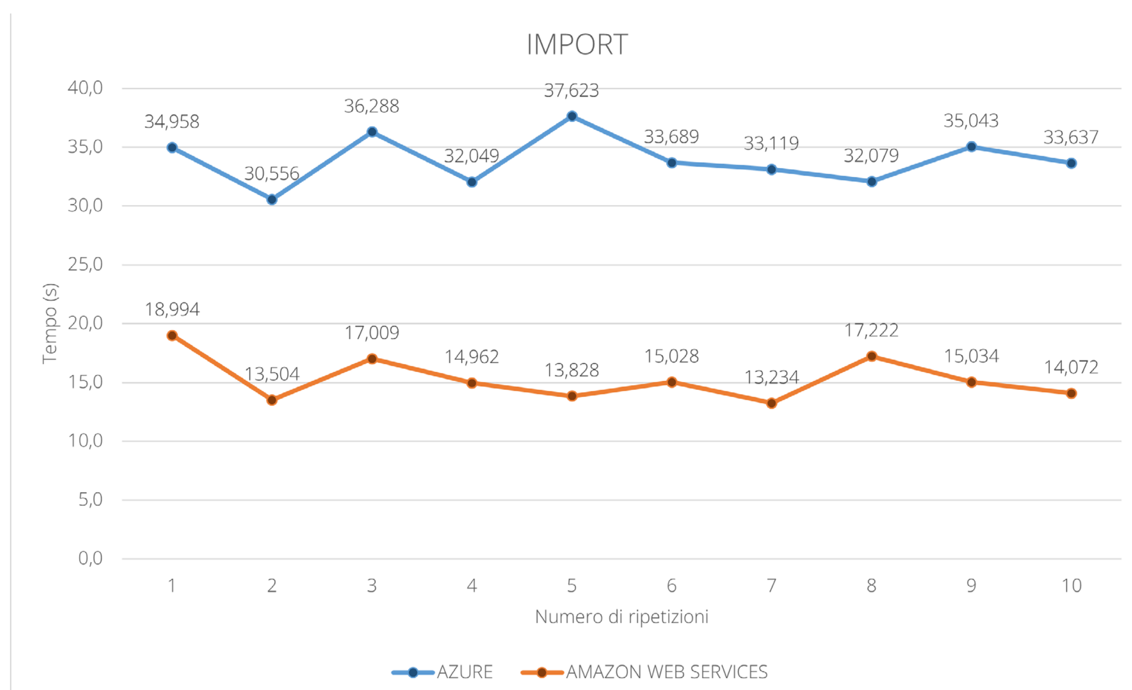

Figura 12 - Tempi di esecuzione importazione

<u>Parsing</u>

Lo script Python sviluppato è necessario per la creazione di blueprint IaC, eliminando i problemi riscontrati con il file generato in automatico della fase precedente. I tempi di esecuzione dello script di parsing sono notevolmente ridotti rispetto a quelli di discovery per lo script di Azure (2,39s) mentre sono simili per AWS (0,80s). In media per il primo Cloud Service Provider sono necessari 0.06 secondi per completare l'esecuzione dello scirpt, mentre per il secondo 0.50 secondi. I tempi ridotti per Microsoft Azure sono dovuti al fatto che non avvengono chiamate API, ma viene solamente letto in input un file, modificato e creato un nuovo file con le modifiche, a differenza di AWS che necessita di chiamate API per eseguire le operazioni di recupero dei tag non importati a causa del problema del provider *aws-classic*.

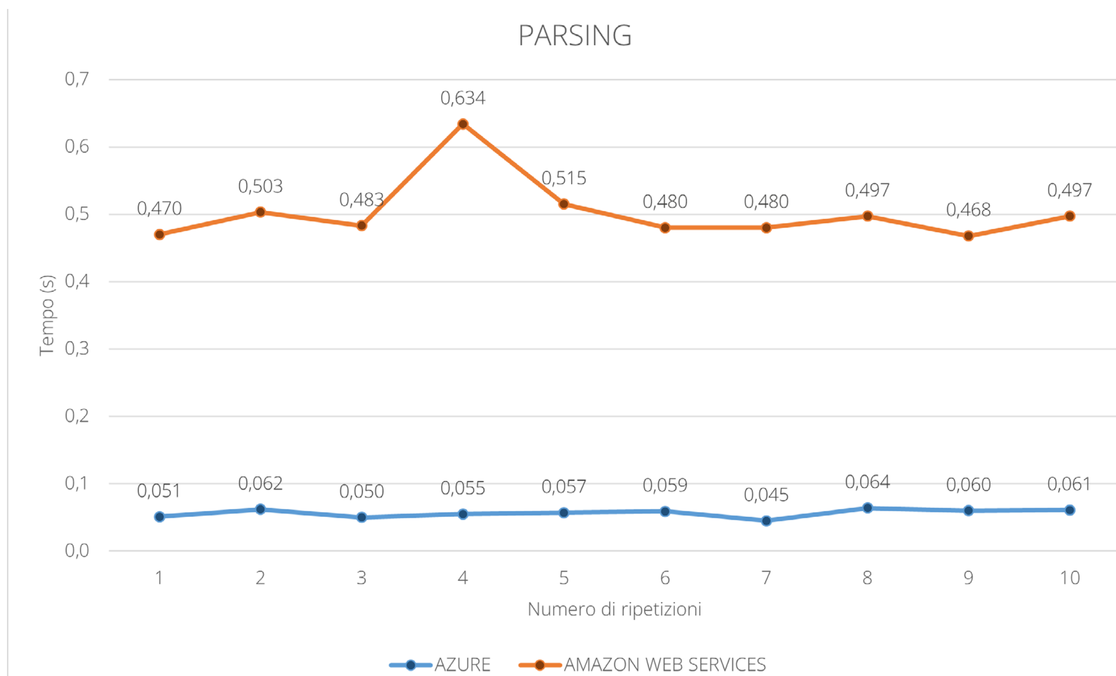

Figura 13 - Tempi di esecuzione script di parsing

<u>Customization</u>

Lo script Python personalizza il blueprint della fase precedente: viene preparato il file IaC per la distribuzione delle risorse in una nuova regione in caso di disaster recovery. Per lo stesso motivo descritto nel paragrafo precedente, abbiamo tempi di esecuzione medi per lo script di customization di 0.04 secondi per Microsoft Azure e 1.27 per Amazon Web Services. I tempi per AWS sono maggiori rispetto al caso precedente poiché vi sono più chiamate API.

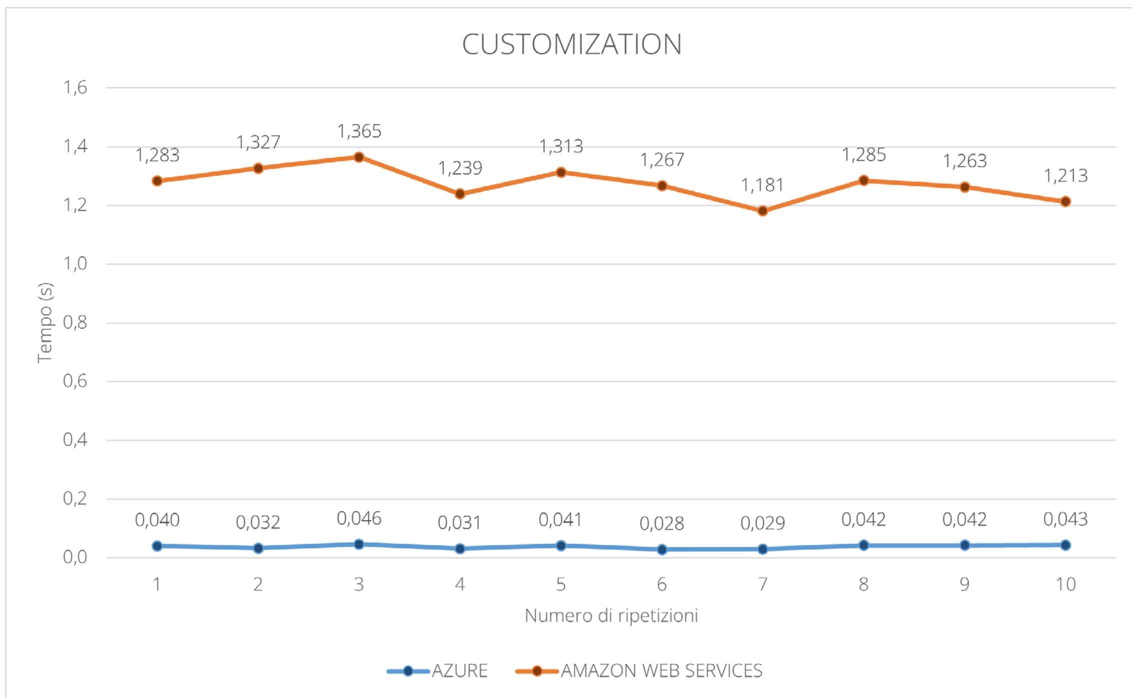

Figura 14 - Tempi di esecuzione script di customization

4.2 Tempi di esecuzione pipeline

Durante l'esecuzione delle pipeline, non solo vengono eseguiti gli script sviluppati, ma deve essere creato l'ambiente virtuale Python, nella cartella venv, e vengono installate le dipendenze necessarie per permettere un'esecuzione corretta di tutte le operazioni. Inoltre, in tutti test, i tempi di esecuzione prendono in considerazione anche le fasi di verifica dei parametri, di gestione delle repository necessarie e di pulizia del workspace utilizzato.

Importazione e creazione stack

La pipeline sviluppata esegue lo script di discovery e l'importazione delle risorse in uno stack che viene successivamente esportato. I tempi medi sono di 184 secondi per Microsoft Azure e 73,80 secondi per Amazon Web Services. La differenza di tempo tra i due Cloud Service Provider è in linea con quanto descritto per i singoli script.

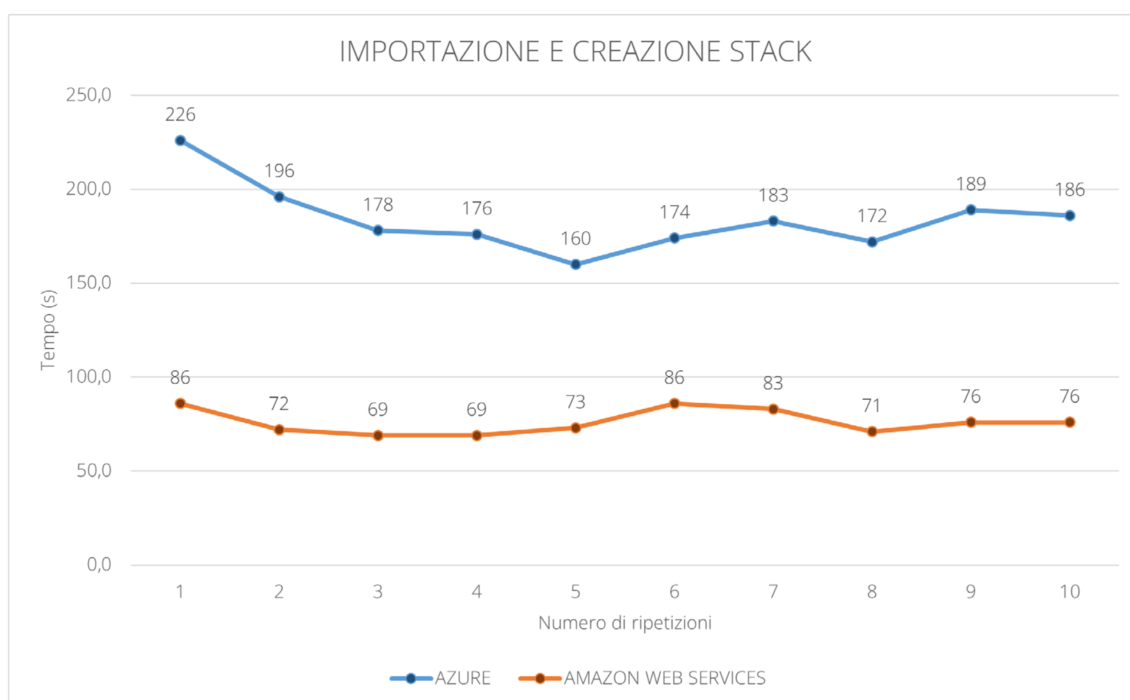

Figura 15 - Tempi di esecuzione pipeline di importazione e creazione dello stack

Importazione e creazione blueprint IaC

L'importazione delle risorse e la creazione del blueprint IaC richiede l'esecuzione dello script di discovery e parsing, e dell'importazione delle risorse. In questo caso lo stack Pulumi non viene esportato e quindi eliminato. I tempi di esecuzione medi per Azure sono di 78,40 secondi mentre per AWS 46 secondi. Questi sono proporzionali a quelli ottenuti con l'esecuzione dei singoli script.

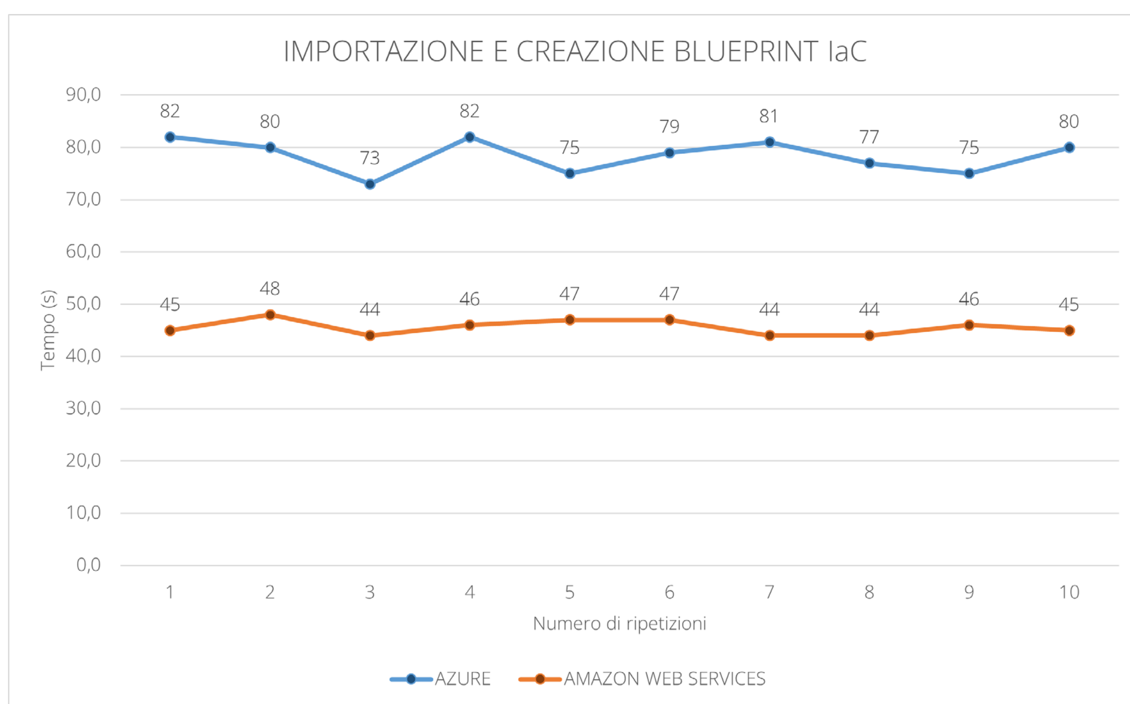

Figura 16 - Tempi di esecuzione pipeline di importazione e creazione del blueprint

Personalizzazione file IaC: cambio regione per problemi di disaster recovery

L'ultima pipeline riguarda il caso di personalizzazione dei file IaC: in particolare, il cambio della regione di distribuzione delle risorse in un caso di disaster recovery. È stato preso in esame sia il caso in cui si vuole creare il file IaC pronto per essere distribuito in un secondo momento, sia il caso di distribuzione immediata. Nei tempi ricavati da questi test sono compresi tutti gli stage della pipeline

Proporzionalmente ai risultati ottenuti dai test sui singoli script, il tempo medio necessario per la sola creazione del file è di 14,40 secondi per

Microsoft Azure e di 13,70 per Amazon Web Services.

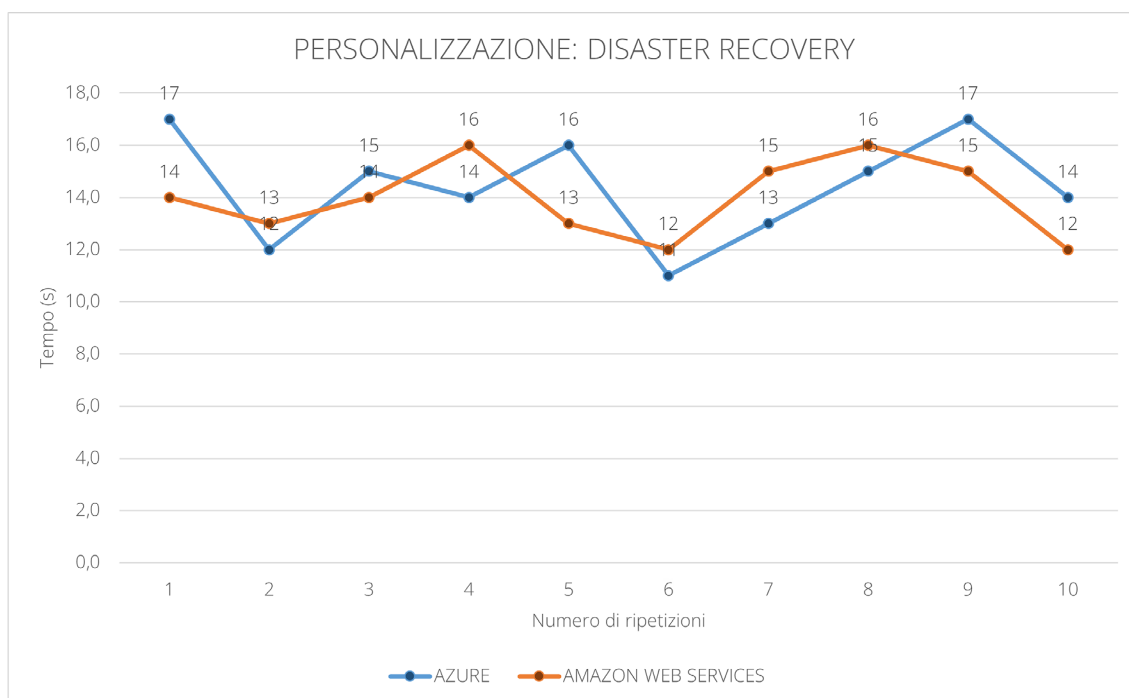

Figura 17 - Tempi di esecuzione pipeline di personalizzazione: disaster recovery

Nel caso in cui le risorse vengono distribuite nella nuova regione, i tempi si dilatano, in quanto è necessaria le creazione del progetto Pulumi: Microsoft Azure necessita in media di 188,40 secondi e Amazon Web Services di 80,80 secondi.

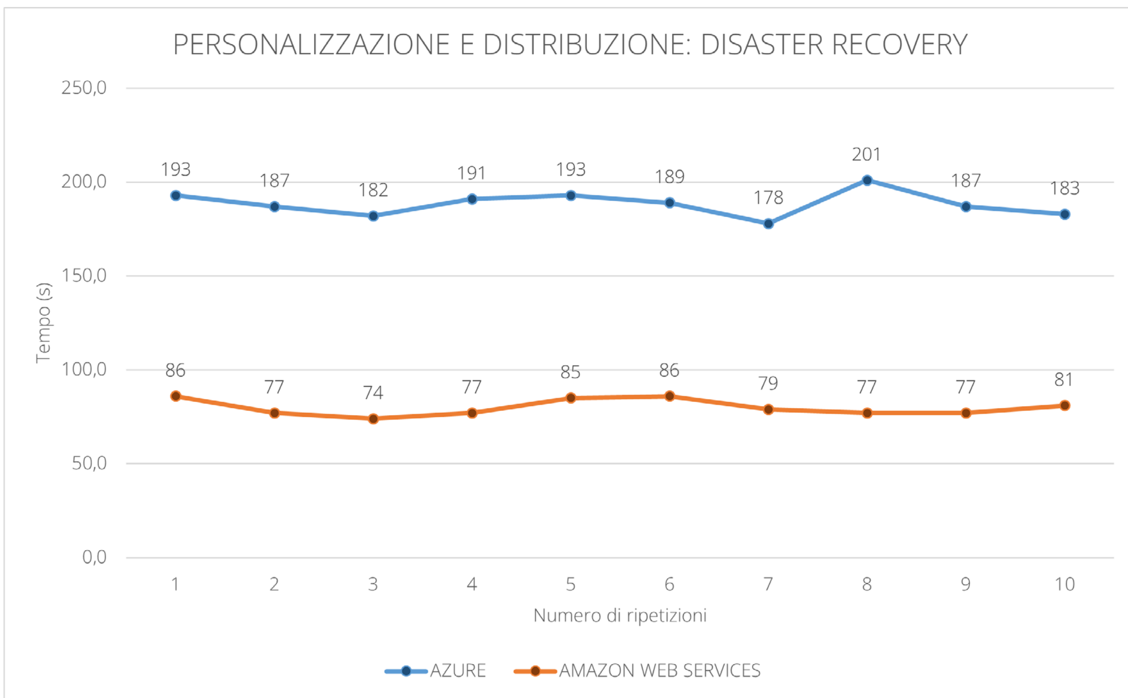

Figura 18 - Tempi di esecuzione pipeline di personalizzazione e distribuzione: disaster recovery

4.3 Stime e confronti con operazioni manuali

Per confermare l'importanza di automatizzare il processo d'importazione delle risorse e della loro gestione tramite Infrastructure as Code, è stato ritenuto necessario confrontare i tempi ottenuti dall'esecuzione delle pipeline create con i tempi di un processo manuale che ottiene gli stessi risultati. È necessario considerare che la fase di importazione in cui viene utilizzata la funzionalità di import di Pulumi, i tempi sono gli stessi, ma il file utilizzato in input al comando di importazione deve essere necessariamente creato: il tempo necessario per la creazione manuale del file è notevolmente maggiore. I vari step da eseguire per ogni risorsa sono i seguenti:

- Creazione della struttura del file JSON necessario per l'importazione
- Individuazione della risorsa dal portale del Cloud Service Provider
- Recupero *name, id, tipo*, per ogni risorsa
- Accesso alla pagina del *tipo* della risorsa nella documentazione Pulumi per il Cloud Provider scelto e individuazione e recupero del *tipo Pulumi* della risorsa

Viene stimato in media un tempo di 2 minuti per ogni risorsa, che può variare a seconda delle capacità e della velocità dell'operatore ad eseguire le operazioni indicate. Tenendo conto che durante la fase di discovery i tempi indicati riguardano 5/6 risorse, si notano miglioramenti notevoli con l'utilizzo degli script sviluppati. È stato simulato il processo manuale di creazione del file considerando le medesime risorse dei test effettuati nella fase di discovery: il tempo che è stato necessario per ottenere lo stesso risultato è di 12 minuti e 48 secondi, rispetto ai soli 2,39 secondi necessari per eseguire l'intero script.

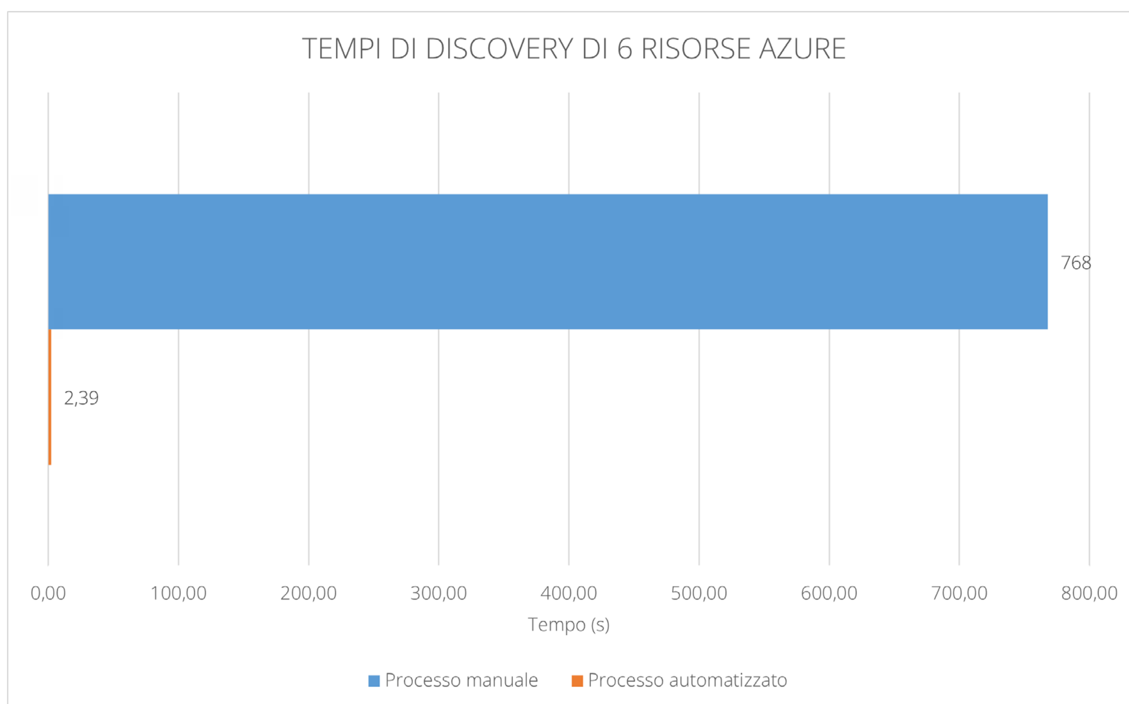

Figura 19 - Tempi di esecuzione script discovery con 6 risorse vs processo manuale

Inoltre, utilizzando le pipeline sviluppate è semplice avere a disposizione uno stack Pulumi con cui gestire le risorse: la modifica di una proprietà di una qualsiasi risorsa consiste nell'individuazione all'interno del file e la modifica del valore sulla riga in cui vi si trova il parametro relativo ad essa. Eseguire la stessa operazione senza l'utilizzo di un file IaC, richiederebbe più tempo poiché è necessario accedere al portale, individuare la risorsa e la proprietà specifica e modificarne il valore. È ancora più evidente se le modifiche da effettuare sono in numero maggiore, specialmente considerando risorse diverse: dal portale è necessario navigare tra le varie pagine per raggiungere la scheda con le proprietà da modificare, a differenze di gestirle tramite file IaC, individuando la risorsa nel codice stesso.

4.4 Ulteriori vantaggi

La possibilità di avere i file con le risorse "iac-izzate" storicizzate in un repository, permette una più semplice gestione dell'infrastruttura, garantendo la possibilità di eseguire rollback in caso di problemi o limitare la creazione o modifica di risorse a qualsiasi utente; è possibile, ad esempio, implementare un sistema di *Merge Request*: le richieste vengono validate da una singola persona permettendo la creazione di una risorsa o la sua modifica.

Grazie all'IaC, è possibile replicare rapidamente e facilmente l'intera infrastruttura in più ambienti, come sviluppo, test e produzione. Questo è particolarmente vantaggioso quando si vuole effettuare il deploy di una nuova versione di un'applicazione o testare nuove funzionalità.

Inoltre, un sistema automatizzato, permette una minore interazione dell'operatore, garantendo una percentuale di errore umano drasticamente inferiore.

La portabilità del progetto è un punto fondamentale, in quanto è possibile eseguire le operazioni richieste da qualsiasi terminale, il quale deve rispettare i requisiti descritti nel Capitolo 3.1, ovvero l'installazione e la configurazione di Pulumi, Jenkins, Python e Git. Inoltre, gli script sviluppati e i file IaC generati sono salvati su repository Git sempre online, con i quali è possibile comunicare da qualsiasi terminale che soddisfa i requisiti. Il peso dei file nei repository è ridotto in quanto la dimensione di essi è ridotta: basandosi sui test eseguiti, una cartella contenente i file di un intero progetto Pulumi non raggiunge 50 kilobytes di dimensione.

È stata effettuata un'ulteriore verifica delle prestazioni della pipeline relativa al primo scenario: discovery delle risorse e importazione. Per analizzare l'incremento dei tempi di esecuzione all'aumentare del numero di risorse sono stati eseguiti tre test su set di risorse diversi, composti da 1, 5 e 10 risorse. I dati ottenuti differiscono di circa 20 secondi tra un caso e l'altro: in media, per ogni risorsa aggiunta è necessario un tempo di circa 4 secondi. L'incremento dei tempi di esecuzione all'aumentare delle risorse da importare è, quindi, lineare.

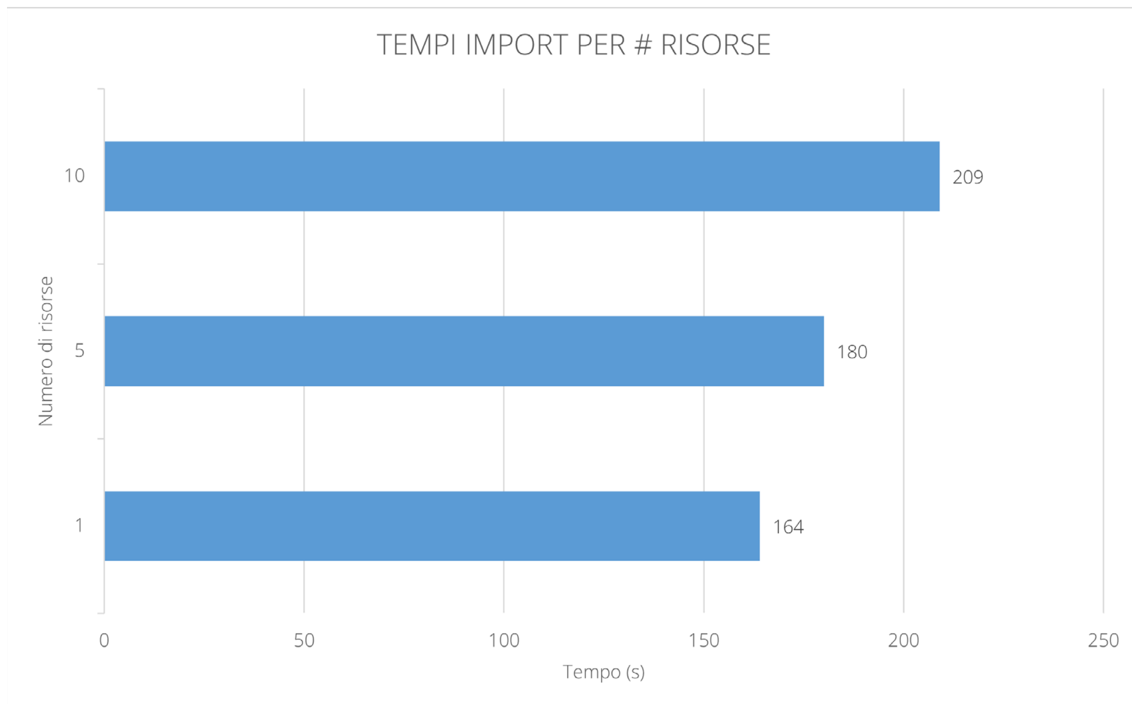

Figura 20 - Tempi di import con set da 1,5 e 10 risorse

Infine, con l'obiettivo di risolvere errori e ottimizzare l'esecuzione, sono stati effettuati dei test per ogni pipeline creata, in modo da validare la correttezza di ogni stage e salvare i dati ottenuti seguendo una struttura specifica nei repository; in questo modo è possibile rendere più semplice la consultazione dei relativi dati, sia da parte degli utilizzatori, sia con l'uso di automazioni, già presenti ed eventualmente future.

Capitolo 5

Limitazioni e sviluppi futuri

Lo sviluppo del progetto eseguito ha riscontrato alcune limitazioni e possibili sviluppi futuri.

Un punto da approfondire è il problema legato all'utilizzo del provider *aws-classic* di Pulumi: questo presenta alcuni problemi che devono essere controllati manualmente. Negli script che sono stati sviluppati, sono state automatizzate delle modifiche per risolvere questi problemi, ma essendo questi legati ai parametri delle risorse, nuovi tipi di risorsa potrebbero riscontrare problemi simili: quindi è necessario un aggiornamento dello script in uso. Un modo alternativo è quello di utilizzare come provider di Pulumi per AWS, la nuova versione *aws-native*, la quale però è ancora in fase di sviluppo e di conseguenza non tutte le risorse sono disponibili. È stato testato, inoltre, l'utilizzo di questo provider con le risorse che sono già funzionanti e, in particolare, il problema riscontrato con i tag non importati nelle risorse Bucket è stato corretto. In futuro, l'utilizzo di questo provider ottimizzerà le prestazioni degli script già sviluppati.

Il progetto realizzato utilizza come filtro i tag delle risorse: questo aspetto può essere una limitazione per le risorse che non sono state taggate precedentemente. Una possibile soluzione è quella sviluppata dall'Ing. Luca Castellana, con il Suo progetto di tesi sviluppato in Liquid Reply: questo permette di individuare e taggare le risorse seguendo un algoritmo da Lui sviluppato[24].

Sono state inoltre effettuate alcune prove con risorse PaaS, nello specifico con App Service, a cui è stato provato montare un File Share, per il salvataggio di dati, all'interno di uno Storage Account. È stato notato che in questo modo l'importazione avviene correttamente ma nel momento in cui si provano a distribuire le risorse, il File Share

non viene montato. Il progetto è in grado di ricostruire l'infrastruttura ma non i dati utilizzati e salvati nell'infrastruttura: è necessario quindi utilizzare un servizio di backup, che può essere fornito sia dal Cloud Provider stesso, sia da prodotti esterni, sia progettato internamente.

Conclusioni

In questo progetto di tesi sono stati studiati, analizzati e sviluppati processi e automazioni per la gestione efficiente delle risorse cloud. La metodologia Infrastructure as Code è un punto fondamentale per la realizzazione della soluzione proposta, permettendo la gestione delle risorse tramite codice.

Durante lo svolgimento del progetto sono state utilizzate diverse tecnologie: per applicare un approccio Infrastructure as Code è stato scelto Pulumi, un innovativo strumento che permette lo sviluppo dei processi mediante un linguaggio di programmazione a scelta tra quelli più famosi, tra cui Python, scelto come linguaggio di programmazione principale. Ulteriori strumenti sono stati utilizzati durante lo sviluppo del progetto, tra i quali Jenkins, per la creazione ed esecuzione delle pipeline, e Git, per avere repository in grado di salvare i dati necessari, ovvero script e file di database.

I risultati ottenuti portano a un notevole decremento del tempo necessario per "iac-izzare" le risorse presenti nei vari Cloud Service Provider: ad esempio, nella fase di acquisizione dei dati necessari a Pulumi per importare le risorse, si stima che in media un processo manuale necessita di circa due minuti per risorsa, rispetto ai pochi secondi necessari se si utilizza lo script sviluppato. È stato eseguito un test di discovery di 6 risorse sia manualmente, sia utilizzando lo script sviluppato: nel primo caso il tempo è di 12 minuti e 48 secondi, mentre il secondo metodo richiede soltanto 2,39 secondi.

L'utilizzo degli script e delle pipeline create portano con sè altri vantaggi, come controllo di versione, portabilità, elasticità, scalabilità, automazione e riduzione della percentuale di errore umano.

In conclusione, con la continua evoluzione di ambienti cloud e multicloud, la gestione delle risorse diventa sempre più difficile: è sempre più necessario adottare un approccio DevOps che aiuta le aziende nella gestione dell'intera infrastruttura. Il progetto realizzato è un supporto

concreto per le aziende per la creazione di un’infrastruttura “iac-izzata”, apportando ad essa tutti i vantaggi che ne conseguono.

Bibliografia e Sitografia

[1] Peter Mell, Timothy Grance, "NIST SP 800-145, The NIST Definition of Cloud Computing" (Settembre 2011), https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf (Consultato il 29/02/2024)

[2] "Cos'è il cloud computing?", https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-cloud-computing (Consultato il 29/02/2024)

[3] "Definizione e confronto tra i vari tipi di cloud" (4 agosto 2023), https://www.redhat.com/it/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud (Consultato il 29/02/2024)

[4] Yuval Shavit, "Hardware-as-a-service (in managed services)" (Giugno 2019), https://www.techtarget.com/searchitchannel/definition/Hardware-as-a-Service-in-managed-services (Consultato il 29/02/2024)

[5] Mary Zhang, "Cloud Regions and Availability Zones: Explained", (12 Ottobre 2023), https://dgtlinfra.com/cloud-regions-availability-zones/ (Consultato il 29/02/2024)

[6] Stephen J. Bigelow, "Select the right cloud regions, availability zones to optimize costs", (24 Marzo 2017), https://www.techtarget.com/searchcloudcomputing/tip/Learn-the-cost-implications-of-cloud-regions-and-availability-zones (Consultato il 29/02/2024)

[7] "Regions and Zones", https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (Consultato il 29/02/2024)

[8] "Aree geografiche e zone", https://cloud.google.com/compute/docs/regions-zones?hl=it (Consultato il 29/02/2024)

[9] "Azure cross-region replication" (24 Settembre 2023), https://learn.microsoft.com/en-us/azure/reliability/cross-region-replication-azure (Consultato il 29/02/2024)

[10] "Global cloud services spending up 16% in Q3 2023 as growth rate

stabilizes" (22 Novembre 2023), https://canalys.com/newsroom/global-cloud-services-q3-2023 (Consultato il 29/02/2024)

[11] "What is Multi-Cloud Infrastructure?", https://www.vmware.com/topics/glossary/content/multi-cloud-infrastructure.html (Consultato il 29/02/2024)

[12] Milind Govekar, "The Future of Cloud: Prepare for 2025", https://www.gartner.com/en/webinar/445864/1051166 (Consultato il 29/02/2024)

[13] Tanner Luxner, "Cloud computing trends and statistics: Flexera 2023 State of the Cloud Report" (5 Aprile 2023), https://www.flexera.com/blog/cloud/cloud-computing-trends-flexera-2023-state-of-the-cloud-report/ (Consultato il 29/02/2024)

[14] Tom Hall, "Cos'è la cultura DevOps?", https://www.atlassian.com/it/devops/what-is-devops/devops-culture (Consultato il 29/02/2024)

[15] Tom Hall, "DevOps Best Practices ", https://www.atlassian.com/devops/what-is-devops/devops-best-practices (Consultato il 29/02/2024)

[16] "Che cos'è DevOps?", https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-devops (Consultato il 29/02/2024)

[17] "What is Infrastructure as Code (IaC)?", https://www.vmware.com/topics/glossary/content/infrastructure-as-code.html (Consultato il 29/02/2024)

[18] "What is Infrastructure as Code (IaC)?" (11 Maggio 2022), https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (Consultato il 29/02/2024)

[19] "How Pulumi works", https://www.pulumi.com/docs/concepts/how-pulumi-works/ (Consultato il 29/02/2024)

[20] "Stacks", https://www.pulumi.com/docs/concepts/stack/ (Consultato il 29/02/2024)

[21] "Cosa si intende con CI/CD?" (27 ottobre 2023), https://www.redhat.com/it/topics/devops/what-is-ci-cd (Consultato il 29/02/2024)

[22] "Cos'è una pipeline?" (11 maggio 2022), https://www.redhat.com/it/

topics/devops/what-cicd-pipeline (Consultato il 29/02/2024)

[23] Kavya Tolety, "Jenkins Master and Slave Architecture" (12 Dicembre 2019), https://medium.com/edureka/jenkins-master-and-slave-architecture-e3d6c4728945 (Consultato il 29/02/2024)

[24] Ing. Luca Castellana, "Development and Analysis of FinOps processes for Multi-Cloud enviroment", (2023), https://webthesis.biblio.polito.it/secure/28530/ (Consultato il 29/02/2024)