POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica - Computer Networks and Cloud Computing

Tesi di Laurea Magistrale

Gestione delle API in ambienti cloud distribuiti: Configurazione e controllo di piattaforme Enterprise di API management in multi-hybrid-cloud

Relatore
Prof. Riccardo Sisto

Candidato
Bruno Accogli

Supervisore Aziendale
Dott. Danilo Abbaldo

Anno accademico 2021-2022

Sommario

Con l'inarrestabile crescita dell'ICT (Information and Communication Technologies) che ha inevitabilmente cambiato quasi ogni aspetto della nostra vita e il nostro modo di comunicare, le API hanno rappresentato (e rappresentano ancora) le basi di questa evoluzione digitale, costituendo un rigoroso e fondamentale sistema per uno scambio di informazioni controllato.

Le **APIs** (Application Programming Interfaces) sono delle interfacce che permettono ad un'applicazione di esporre i propri dati e/o funzionalità, ma anche di accedere a quelli di un'altra applicazione. Nell'ambiente Enterprise, le API, inizialmente usate per consentire la comunicazione tra applicativi e piattaforme interne alla compagnia stessa, rappresentano ora un importante e proficuo strumento di business, esponendole al mondo esterno e rendendole accessibili alle community di sviluppatori interessate. Controllare chi può accedere a tali risorse, quando e con quale frequenza rappresenta dunque una fase critica per il design delle API ma soprattutto durante lo sviluppo dell'applicazione stessa.

In un'epoca in cui il **Cloud**, oltre a costituire un insieme di tecnologie rivoluzionarie, si sta affermando sempre di più come strategia di business per le aziende, sta prendendo piede un approccio **multi-Cloud**, ovvero l'adozione di servizi di più Cloud provider in modo da soddisfare varie necessità e affrontare diversi problemi, che vanno dal *vendor lock-in* all'esclusività del servizio offerto.

In questi contesti non possono mancare soluzioni di gestione delle API, in grado di offrire un sistema di controllo in quelle situazioni in cui la proliferazione delle stesse in molteplici ambienti Cloud diventa un problema complesso da risolvere. Il mercato offre diverse soluzioni, ma solo alcune possono considerarsi valide per tale scopo: in questa tesi, dopo aver analizzato in modo approfondito il problema della gestione delle API, verranno presentati i più rilevanti prodotti di API Management con supporto ad ambienti multi-Cloud.

Sulla base dell'esperienza maturata all'interno dell'azienda **Blue Reply** verrà infine proposto un caso d'uso comparativo, volto a confrontare le performance dei gateway di due prodotti Enterprise di API Management presi in analisi e valutare, anche dal punto di vista delle funzionalità di security offerte, quale piattaforma

risulta essere la più flessibile ed "adatta" in uno scenario MultiCloud. Per ogni soluzione verranno evidenziati requisiti e costi.

Indice

1 Introduzione 8

1.1 Cos'è un'API . . . 8

1.1.1 REST . . . 10

1.2 La gestione delle API . . . 13

1.3 Il Cloud nel 2022 . . . 14

1.3.1 La disciplina FinOps . . . 15

1.4 Il Multi-cloud . . . 17

1.4.1 Casi d'uso . . . 18

1.4.2 Modelli di approccio ad architetture multi-cloud . . . 19

1.5 Obiettivi della tesi . . . 23

2 Strategie di gestione delle API su più piattaforme cloud 24

2.1 API Management in ambienti multi-hybrid-cloud e principali problematiche legate . . . 24

2.2 La soluzione . . . 25

2.2.1 Architettura di alto livello . . . 26

2.3 Attori coinvolti . . . 27

2.3.1 L'amministratore . . . 28

2.3.2 Lo sviluppatore . . . 28

2.4 Responsabilità in multi-hybrid-cloud e differenze con un ambiente single-cloud . . . 28

3 Introduzione ai principali sistemi di API Management 30

3.1 Azure API Management . . . 30

3.1.1 Control Plane . . . 30

3.1.2 Data Plane . . . 31

3.2 RedHat 3scale API Management . . . 32

3.2.1 Control Plane . . . 32

3.2.2 Data Plane . . . 33
3.3 Google Cloud Apigee API Platform . . . 34
3.3.1 Control Plane . . . 34
3.3.2 Data Plane . . . 35
3.4 Kong API Gateway . . . 37
3.4.1 Control Plane . . . 37
3.4.2 Data Plane . . . 38

4 Analisi e confronto di due solzioni di API management in uno scenario multi-cloud 42
4.1 Introduzione al problema . . . 42
4.2 Soluzione proposta . . . 43
4.2.1 Descrizione . . . 43
4.2.2 Requisiti . . . 45
4.2.3 Costi . . . 47
4.3 Implementazione . . . 49
4.3.1 Installazione dell'infrastruttura . . . 49
4.3.2 Installazione dei Data Plane . . . 53
4.3.3 Configurazione backend per i gateway . . . 56
4.3.4 Esposizione degli endpoint per i gateway . . . 56
4.4 Principali strumenti utilizzati . . . 60
4.4.1 Docker . . . 60
4.4.2 Kubernetes . . . 62
4.4.3 K3S . . . 65
4.4.4 OpenShift . . . 65
4.4.5 Prometheus . . . 66
4.5 Analisi di performance dei Data Plane . . . 68
4.5.1 Metriche considerate e risultati ottenuti . . . 68
4.5.2 Costi in termini di risorse . . . 69
4.6 Analisi dei security drivers . . . 70
4.6.1 Azure API Management . . . 71
4.6.2 RedHat 3scale API Management . . . 77
4.7 Espandibilità . . . 80
4.8 Confronto delle soluzioni . . . 82

5 Conclusioni 86

5.1 Considerazioni finali 86

5.2 Possibili sviluppi futuri 88

5.2.1 Disaster recovery 88

5.2.2 API Health check 90

5.2.3 API lifecycle DevOps 91

Bibliografia 93

Capitolo 1

Introduzione

1.1 Cos'è un'API

Le API (Application Programming Interfaces) sono delle interfacce che forniscono l'accesso a dei servizi, che possono essere dati oppure funzionalità. Attraverso tutta una serie di strumenti, definzioni e protocolli le API vengono integrate con i software e le applicazioni, consentendo ad un'entità esterna di accedere, tramite esse, a delle funzionalità o dati specifici.

Un'API crea un livello di astrazione tra un'applicazione e il mondo esterno, garantendo flessbilità agli sviluppatori e semplificando design, amministrazione e utilizzo della stessa. Esse inoltre facilitano l'integrazione di nuove funzionalità all'interno di un'architettura esistente, promuovendo la collaborazione dei team business e IT in ambienti Enterprise.
La scalabilità è il punto di forza di questo nuovo livello di astrazione: grazie alle API, infatti, le funzionalità e i dati di un'applicazione possono crescere nel tempo e quelli esistenti possono subire modifiche, o addirittura essere rimossi se obsoleti e sostituiti con una versione più aggiornata.
Una buona API, inoltre, funge da "blackbox", ovvero permette al fornitore non solo di "nascondere" la reale implementazione del servizio, ma anche di migliorarlo, mantenendo un'interfaccia consistente verso il client e lasciando immutata l'esperienza utente, il quale utilizzerà una nuova versione del servizio senza saperlo; l'utilizzo dell'API infatti non impone all'utente di conoscere le specifiche con cui le risorse vengono recuperate o la loro provenienza.

Con l'evolversi delle tecnologie e con la sempre più fondamentale necessità di automatizzazione dei processi, soprattutto in ambito web, le API hanno iniziato a rappresentare uno strumento di accesso ai servizi fondamentale non solo per utenti umani, ma anche per le applicazioni stesse, diventando così un punto di comunicazione essenziale tra due o più software, dove ognuno di essi può essere immaginato come un contenitore di servizi che vengono esposti verso l'esterno, ma allo stesso tempo un utilizzatore di quelli di un'altra applicazione.

Dal punto di vista di un'azienda, le API da essa gestite possono essere

- Private: destinate al solo uso interno, riguardano di solito servizi non accessibili al pubblico in quanto responsabili di materiale sensibile o funzionalità ancora in sviluppo

- Partnered: condivise con partner specifici, per incrementare i flussi di reddito

- Pubbliche: accessibili da chiunque, per consentire a sviluppatori terzi di realizzare applicazioni in grado di interagire con le API aziendali e di promuovere dunque l'innovazione

Le API condivise sono estremamente preziose poiché oltre a semplificare ed espandere il modo in cui l'azienda si collega con i partner e le community di sviluppatori, permettono anche di monetizzare i dati (basti pensare alle API di Google Maps [1]).

Negli anni, il concetto di API come interfaccia generica di connettività tra applicazioni si è evoluto. Recentemente, le API moderne hanno guadagnato delle caratteristiche che le rendono straordinariamente preziose e utili

- Le API moderne aderiscono a standard (tipicamente HTTP e REST) che semplificano il lavoro degli sviluppatori, e sono accessibili e facilmente comprensibili

- Vengono considerate più come prodotti che come codice; sono progettate per un pubblico specifico (basti pensare agli sviluppatori mobile o al mondo cloud), sono documentate e seguono uno schema di *versioning* in modo che gli utenti possano avere una chiara idea sulla manutenzione e sul loro ciclo di vita

- Dal momento che sono molto più standardizzate, hanno un controllo maggiore sulla sicurezza e sulla governance, e sono monitorate e gestite per alte prestazioni e scalabilità

- Come qualsiasi altro prodotto software, ogni API moderna ha il proprio ciclo di vita, che riguarda fasi di progettazione, test, creazione, gestione e controllo delle versioni. Inoltre, sono quasi sempre corredate di documentazione dettagliata sul loro utilizzo e sul versioning.

eBay è stata una delle prime a rilasciare il suo primo set di API, pubblicate nel 2000 con l'intento di aiutare i venditori a gestire e ampliare il loro business sulla piattaforma in modo scalabile. Dopo eBay, tante altre aziende come Netflix, Facebook e Spotify le diedero seguito, contribuendo alla nascita e allo sviluppo di ecostistemi digitali che vede da una parte le API, i **prodotti**, e dall'altra gli sviluppatori, i **clienti**. Negli ultimi decenni, infatti, la crescita inarrestabile della quantità di API pubbliche accessibili tramite il web dimostra come esse siano e saranno fondamentali non solo dal punto di vista tecnologico ma anche come strumento di crescita e di business, soprattutto per le piccole organizzazioni.

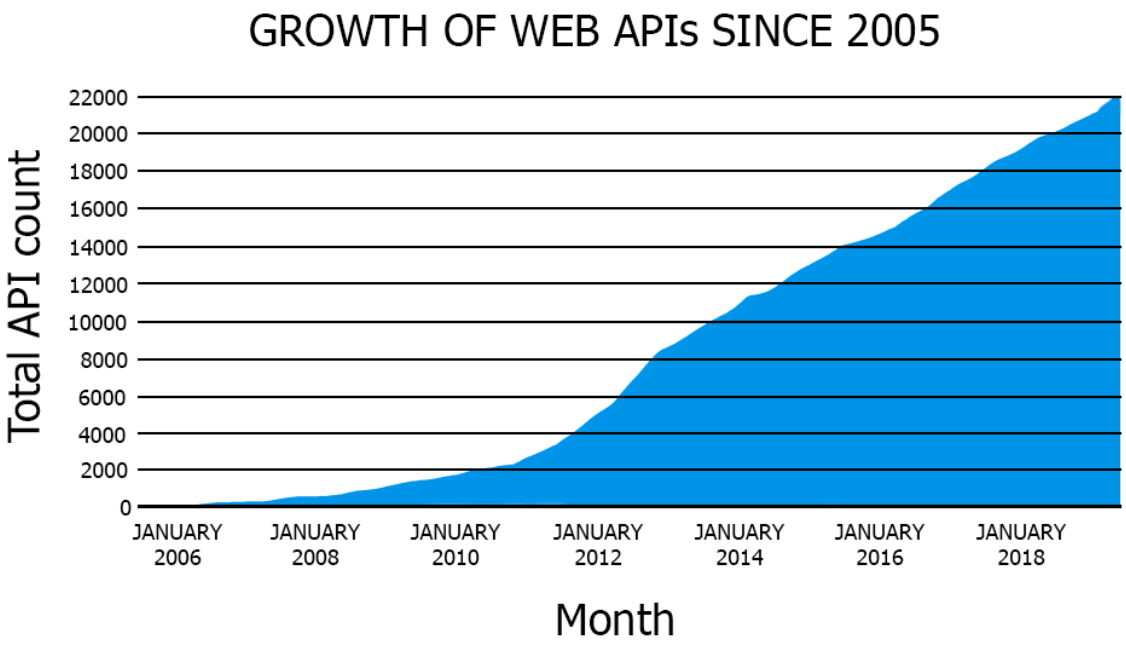

Figura 1.1. Crescita delle web APIs dal 2005. Fonte: programmableweb [2]

1.1.1 REST

REST è uno stile architetturale per la progettaione delle API. Il termine è stato introdotto dall'informatico Roy Fielding, uno dei principali autori delle specifiche dell'HyperText Transfer Protocol (HTTP), nella sua tesi di dottorato; è l'acronimo di REpresentational State Transfer e definisce 6 principi che un'API deve rispettare affinchè sia considerata RESTful

- **Client-server**: un'architettura client-server separata da interfacce uniformi e composta da client, server e risorse, con richieste gestite tramite HTTP. Questo implica una separazione di ruoli e compiti negli attori coinvolti, come ad esempio l'interfaccia grafica gestita dal client e il salvataggio delle informazioni lasciato al server.

- **Stateless (apolidia)**: il server non memorizza alcuno stato sulla sessione. Ogni richiesta che parte dal client è distinta e non connessa alle precedenti e contiene tutte le informazioni necessarie per utilizzare il servizio.

- **Cacheable**: i client possono mettere in cache le risposte, che devono definirsi cacheable o no in modo da prevenire l'utilizzo di stati vecchi o dati errati da parte del client; un corretto utilizzo della cache ottimizza l'interazione client-server migliorando prestazioni e scalabilità

- **Layered system**: la struttura del sistema su più livelli consente di avere più server e di organizzarli in base al ruolo (ad esempio uno reponsabile della sicurezza, uno della memorizzazione dei dati, uno del bilanciamento del carico e uno dell'autenticazione), creando una gerarchia che può vedere coinvolti diversi intermediari durante il recupero di informazioni; in questo modo nè il

client nè il server possono sapere se stanno comunicando con l'applicazione finale o con l'intermediario

- **Codice on demand** (opzionale): la possibilità da parte del server di estendere temporaneamente le funzionalità del client, inviando codice eseguibile, che può includere componenti compilati o degli script

- **Interfaccia uniforme**: un'interfaccia di comunicazione uniforme permette di trasmettere le informazioni in modo standard, disaccoppiando l'architettura tra client e server in modo da poter modificare i singoli componenti. Questo significa che

  - Le richieste devono identificare le risorse, con un identificatore; esse devono essere inoltre separate dalle rappresentazioni inviate al client
  - I client hanno informazioni sufficienti sulla rappresentazione di una risorsa per la modifica o cancellazione della stessa
  - Il server invia al client messaggi autodescrittivi che contengono le informazioni necessarie per l'ulteriore elaborazione della rappresentazione
  - Il server invia al client collegamenti ipertestuali nella rappresentazione in modo che il client possa scoprire ulteriori risorse correlate a completare un'attività

L'architettura REST si basa su HTTP. Il funzionamento prevede una struttura degli URL ben definita che identifica univocamente una risorsa o un insieme di risorse e l'utilizzo dei metodi HTTP specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE) e per altri scopi (OPTIONS, ecc.).
Lo stato di una risorsa in qualsiasi specifico istante è noto come rappresentazione della risorsa.

I vantaggi dell'utilizzo di un approccio REST nello sviluppo delle API includono

- Flessibilità: il disaccoppiamento client-server consente di dissociare e organizzare le varie componenti server così che ognuna possa evolversi in modo indipendente dalle altre; ciò consente agli sviluppatori di cambiare piattaforma o tecnologia lato server senza influire sull'applicazione client

- Scalabilità: REST ottimizza l'interazione client-server; l'apolidia (assenza di uno stato) e il caching ben gestito possono ridurre drasticamente il carico sul server, supportando la scalabilità senza rinunciare alle prestazioni

- Indipendenza: le API REST sono indipendenti dalla tecnolgia utilizzata; è possibile riscrivere le applicazioni client e server, eventualmente in linguaggi differenti, senza influire sul design delle API.

Una **richiesta** da parte di un client RESTful è costituita da

- Identificatore univoco della risorsa o gruppo di risorse: serve a identificare la risorsa

- Metodo HTTP: identifica il tipo di operazione da effettuare sulla risorsa; ad esempio GET, POST, PUT, DELETE

- Header HTTP: contiene dei metadati utili per l'interpretazione della richiesta, come ad esempio il suo formato. Possono essere specificati tramite Dati o Parametri

- Corpo: contiene la rappresentazione della risorsa; utile nel caso di una richiesta con metodo POST, PUT o PATCH

Figura 1.2. Esempio di richiesta POST

Una **risposta** da parte di un server RESTful è costituita da

- Codice di stato: contiene un codice di stato di 3 cifre che indica se la richiesta è andata a buon fine o meno

- Header HTTP: fornisce ulteriori informazioni sulla rispsota, come la codifica, la data e il tipo di contenuto

- Corpo: contiene la rappresentazione della risorsa (nel caso di una richiesta con metodo GET) o altre informazioni

```
HTTP/1.1 200
Content-Type: application/json

{
    "id": 10,
    "name": "shirt",
    "color": "red",
    "price": "$23"
}
```

Figura 1.3. Esempio di risposta ad una richiesta GET

1.2 La gestione delle API

Un'azienda il cui business ruota fortemente attorno ai propri servizi offerti tramite API si ritrova subito a dover affrontare il problema della loro gestione, necessaria in particolar modo in scenari in cui le molteplici applicazioni possiedute, alcune costruite con microservizi e altre gestite in modo *legacy*, sono responsabili dei servizi esposti con delle API che aumentano e subiscono modifiche continuamente. Per le organizzazioni è molto comune, inoltre, consumare le API di altre aziende, contribuendo sempre di più all'instaurarsi di rapporti produttore/consumatore grazie ad esse; **le aziende che offrono le loro API come prodotti possono quindi consumare quelle di altre aziende che fanno lo stesso**.

Con l'affermarsi del Cloud come strategia di business e la nascita dei **microservizi**, diverse realtà hanno dovuto scontrarsi presto con il fenomeno della proliferazione delle API e gestire ingenti quantità di traffico da un microservizio all'altro, evidenziando dunque quanto sia fondamentale per un'azienda la necessità di controllare le API che possiede e il modo in cui vengono utilizzate.
Sebbene la funzione di **API gateway** svolta da prodotti come Envoy, Traefik, Emissary Ingress e Gloo sia centrale ed essenziale per operazioni quali URL mapping & rewriting, rate limit, gestione delle API keys e cache, una piattaforma di API Management, oltre a includere la componente gateway, mira a offrire strumenti aggiuntivi che semplificano tutte le operazioni di progettazione, amministrazione e monitoraggio delle API stesse.

La gestione delle API riguarda in gran parte la centralizzazione del controllo delle stesse, includendo analytics, access control, monetizzazione e developer workflows. Una soluzione di API management ha come obiettivo migliorare la QoL (Quality of Life) degli attori coinvolti, fornendo ad essi gli strumenti necessari per svolgere tutte le operazioni in modo efficiente ed efficace. Per il raggiungimento di questi obiettivi, a supporto dunque del lifecycle di un'API, Gartner individua cinque aree funzionali fondamentali per un prodotto di API management

- **Developer Portal**: un catalogo self-service di API per abilitare, commercializzare e governare, tramite un sistema di controllo degli accessi, gli ecosistemi di sviluppatori che consumano API
- **API gateway**: gestione runtime della sicurezza e monitoraggio dell'utilizzo delle API
- **Gestione delle policies e analytics**: configurazione della sicurezza, mediazione delle API e analisi dell'utilizzo delle stesse
- **Progettazione e sviluppo delle API**: un'esperienza di sviluppo significativa e strumenti per la progettazione e la creazione di API e per l'abilitazione alle API di sistemi esistenti
- **API testing**: dai test di simulazione di base a test avanzati di funzionalità, prestazioni e sicurezza delle API

Alcuni fornitori si focalizzano su alcune delle precedenti aree, mentre altri supportano il lifecycle completo [6].

Figura 1.4. Magic Quadrant for Full Life Cycle API Management. Fonte: Gartner [6]

1.3 Il Cloud nel 2022

Fin dalla sua nascita, il cloud è stato sinonimo di innovazione. Agli albori del cloud pubblico, nuovi prodotti innovativi, iniziative di marketing e nuove aree geografiche erano possibili grazie a un'implementazione dell'infrastruttura IT più rapida e alla riduzione delle spese necessarie per testare nuove idee. L'innovazione continua tutt'ora, in quanto il cloud consente alle aziende di ottenere un accesso tempestivo e a basso rischio a nuove tecnologie come computer vision, AR/VR, serverless, e altro ancora.
Sebbene la migrazione rimanga un caso d'uso ancora molto utilizzato per il cloud pubblico, è bene ricordare che non è fine a se stessa [7].

L'esperienza di Netflix fa da padrone in quest'ambito: nata come azienda di noleggio DVD e videogiochi, nel 2008 ha avviato il suo servizio di streaming online on-demand, diventato presto il suo core business. Già qualche anno prima, però, in seguito a una corruzione dei propri database che impediva di noleggiare i DVD, Netflix aveva iniziato la migrazione verso il cloud, spostandosi da single-point-of-failure scalati verticalmente, come i database relazionali nel datacenter, verso sistemi distribuiti, altamente affidabili e scalabili orizzontalmente nel cloud, portandoli a spegnere l'ultimo datacenter nel 2016 e a dichiarare la migrazione

completa.
Ad oggi, Netflix usa risorse cloud (precisamente AWS) per qualsiasi necessità di scalabilità e storage, logica di business, big data processing/analytics, transcoding e tutte le funzioni che consentono alla piattaforma di streaming di funzionare [3].

Per restare competitive e rispondere ai costanti mutamenti dei mercati digitali, in cui nuovi concorrenti possono rivoluzionare un intero settore con una nuova app, le aziende devono adattarsi rapidamente e supportare lo sviluppo e il deployment di servizi innovativi. La creazione di applicazioni cloud-native, basato sul collegamento di un'architettura applicativa di microservizi attraverso le API, consente di accelerare tale processo; è sempre più frequente, infatti, l'abbandono dell'approccio *lift-and-shift* in favore di quello cloud-native, come previsto dal report *Predictions 2022* di Forrester [4].

1.3.1 La disciplina FinOps

Contemporaneamente, però, l'uso crescente del cloud pubblico si traduce per le organizzazioni di tutte le dimensioni in un aumento importante delle spese per il cloud, che è ora un elemento significativo nei budget IT.

Secondo il *Flexera 2022 State of the Cloud Report*, il 37% delle aziende ha dichiarato che la propria spesa annuale ha superato i 12 milioni di dollari e l'80% ha riferito che la spesa per il cloud supera 1,2 milioni di dollari all'anno. Le PMI eseguono carichi di lavoro sempre più piccoli; tuttavia, il 53% di esse spende più di 1,2 milioni di dollari, rispetto al 38% dello scorso anno. Risulta evidente che le organizzazioni continuano a spendere sempre di più per il cloud pubblico.
L'approccio multi cloud, inoltre, se da un lato rappresenta un vantaggio strategico di business, dall'altro necessita di una governance più puntuale e accurata: dover implementare e interconnettere i cloud di diversi fornitori implica, infatti, dover gestire e ottimizzare costi figli di logiche che, pur comuni sotto alcuni aspetti, sono il risultato di aziende diverse.
Secondo il report, inoltre, la spesa per il cloud pubblico da parte delle aziende è oltre il budget in media del 13% e si aspettano che la loro spesa per il cloud aumenterà del 29% nei prossimi dodici mesi. Questa tendenza indica che è più importante che mai gestire le **previsioni** e l'**ottimizzazione dei costi**.

È facile dunque capire perché la metodologia **FinOps** (crasi di Financial e Operations) si sta evolvendo e viene adottata come disciplina di gestione finanziaria: aiuta a evitare l'allocazione di risorse non utilizzate, a riuscire a dare valore ai dati e a elaborarli in rapporti che forniscano insights rilevanti, ad ottimizzare i costi e ridurre gli sprechi. In un framework FinOps, i gruppi IT e DevOps collaborano con i team di altri settori, come Finance, nella gestione dei costi del cloud e dell'intera impresa, con l'obiettivo di consentire ad essa di trarre dal cloud il massimo valore aziendale aiutando i team dei settori Tecnologia, Finance e Business a prendere decisioni di spesa condivise basate sui dati. Il fulcro di questa pratica gestionale consiste nell'applicare sistemi di controllo dei costi alle implementazioni cloud, preservando al contempo l'accesso alle risorse da parte degli sviluppatori .

L'implementazione di un framework FinOps prevede diverse fasi, le quali, per mantenere l'efficienza e consentire le innovazioni riducendo contemporaneamente i costi, devono essere ripetute periodicamente:

1. **Informare**: ovvero rendere ogni team consapevole dei costi; questa fase offre visibilità sui costi a tutti gli stakeholder riportando l'allocazione della spesa cloud totale all'azienda. È incentrata sul consentire ai team di gestire la spesa del cloud con l'obiettivo di raggiungere la responsabilità finanziaria.

2. **Ottimizzare**: per gestire al meglio le risorse cloud denza sprechi; in questa fase ci si concentra sulla riduzione dei costi del cloud per prendere decisioni intelligenti sull'utilizzo delle risorse e utilizzare efficacemente le leve finanziarie per ridurre le tariffe.

3. **Gestire**: ovvero integrare ogni giorno in azienda le pratiche FinOps a tutti i livelli; in questa fase si perfezionano gli obiettivi IT, finanziari e aziendali condivisi per concentrare e scalare gli sforzi operativi attraverso un miglioramento continuo.

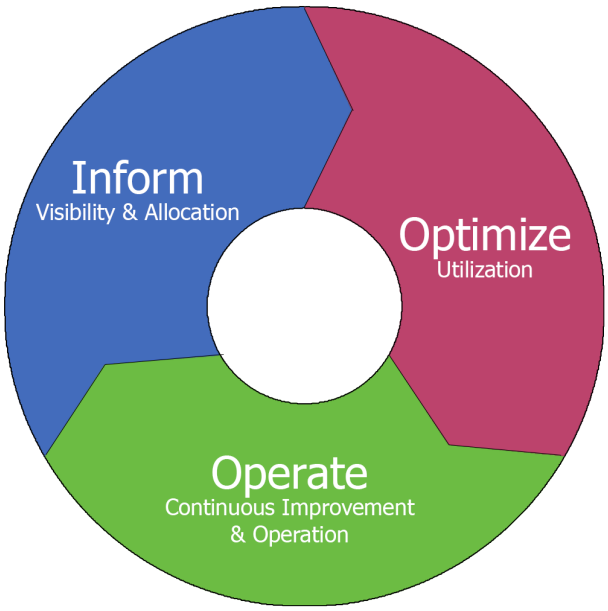

Figura 1.5. Fonte: FinOps Foundation [9]

L'aspetto positivo è che, come dimostrato anche dalle maggiori tendenze in ambito cloud nel 2022, sempre più aziende adottano la disciplina FinOps. Accanto all'ottimizzazione dell'utilizzo dei servizi dei cloud provider, e dunque alla riduzione dei costi, troviamo la sempre più frequente migrazione dei workload nel cloud e l'abbandono dei software on-premise in favore di quelli SaaS (Software-as-a-Service). [8].

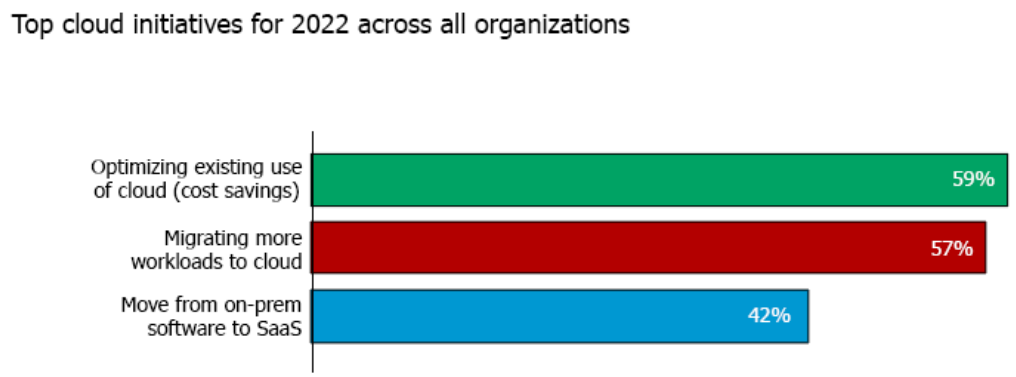

Figura 1.6. Fonte: Flexera [8]

1.4 Il Multi-cloud

La tendenza da parte delle aziende ad adottare i servizi di più piattaforme cloud è aumentata notevolmente negli ultimi due anni. Secondo il *Flexera 2022 State of the Cloud Report*, l'89% delle compagnie utilizzano un approccio multi cloud e l'80% utilizza il cloud ibrido, combinando l'utilizzo di cloud pubblico e ambiente onpremise.

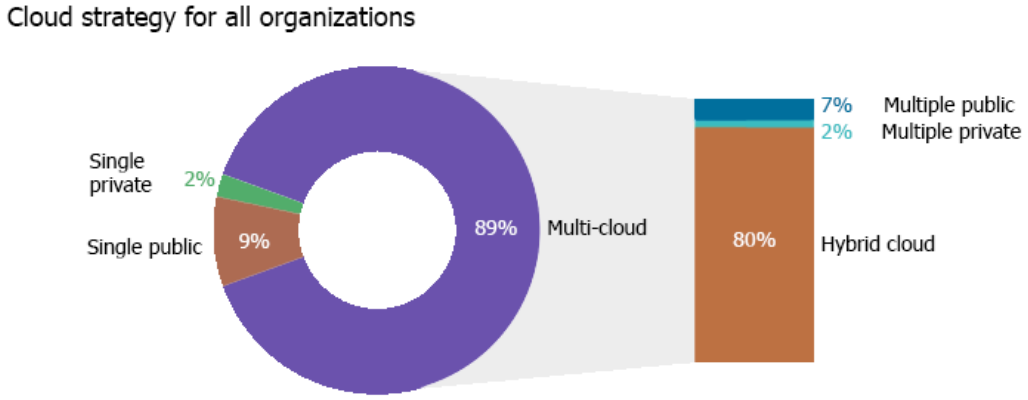

Figura 1.7. Fonte: Flexera [8]

In alcuni casi, l'utilizzo di un approccio multicloud è il risultato di un piano strategico per sfruttare la tecnologia più adatta per lo use case più adatto. In altri casi, che vedono spesso protagonisti singole business units o sviluppatori di applicazioni, vengono prese decisioni indipendenti senza un approccio centralizzato, trovandosi di fronte un drammatico incremento della complessità dell'ambiente cloud prima che l'azienda stessa se ne accorga.

Secondo Forrester, infatti, gli ultimi cinque anni di conferenze sul cloud hanno reso evidente che il multi cloud era la norma, ma allo stesso tempo non lo era: per

molte aziende, multi cloud significava un unico cloud primario con i workload principali e casi d'uso una tantum ospitati su cloud alternativi; infatti, le compagnie che iniziavano ad utilizzare due cloud primari hanno spesso riscontrato ritardi significativi e per questo motivo le aziende pragmatiche hanno scelto strategicamente di standardizzare su un unico cloud per garantire qualità e innovazione più rapida. Tuttavia, tale approccio sta svanendo: oggi le aziende considerano le implementazioni multicloud strategiche, sia per mitigare il rischio di dipendenza da un singolo fornitore (a fini di resilienza o negoziazione) sia per eseguire le principali categorie di carichi di lavoro con la piattaforma cloud più adatta a gestirli [7]

È fondamentale, però, avere una strategia operativa e avere ben chiare le esigenze perché, come visto nel capitolo precedente, la complessità e i costi aumentano necessariamente con l'utilizzo di più di un cloud provider e occorre saperli gestire per massimizzare i benefici ricavati dal multi cloud.

1.4.1 Casi d'uso

Le esigenze che portano un'azienda ad adottare il multicloud sono molteplici, tra cui le più comuni

- **Accesso a nuove Regions**: non tutti i fornitori di servizi cloud supportano le Region in tutte le parti del mondo. Per migliorare le prestazioni del servizio nei confronti dei clienti in una regione specifica o per raggiungere determinati segmenti di mercato in modo esclusivo tramite un cloud provider, è necessario posizionare i workload e i dati più vicini ai clienti stessi, in modo da soddisfare esigenze di questo tipo
- **Obblighi legali sui dati**: bisogna preoccuparsi della giurisdizione che opera nelle regioni in cui i dati vengono immagazzinati. In particolare, si parla di
  - Località dei dati (data residency): fa riferimento al luogo in cui l'organizzazione decide di archiviare i propri dati, di solito per normative interne
  - Sovranità dei dati (data sovereignity): fa riferimento ai diritti di un governo di accedere ai dati situati entro i propri confini; questi possono differire ampiamente da paese a paese e possono estendersi oltre i confini del paese stesso in base a dove risiede l'organizzazione; la differenza rispetto al precedente è che non solo i dati vengono archiviati in una determinata regione, ma sono soggetti alle leggi in cui sono fisicamente conservati.
  - Localizzazione dei dati (data localization): è il più stringente dei tre; implica che i dati creati all'interno di determinati confini non escano al di fuori o che una copia dei dati sia conservata all'interno del paese; è quasi sempre applicato alla creazione e all'archiviazione di dati personali. È necessario, in questo caso, più di un cloud provider (e sistema di archiviazione associato) per rispettare le norme e i regolamenti del paese

- **Evitare il "vendor lock-in"**: il termine "vendor lock-in", in ambito cloud, fa riferimento a quel rapporto di dipendenza che si instaura tra un'azienda e un cloud provider tale per cui l'azienda si ritrova costretta ad utilizzare i servizi dello stesso fornitore per non dover affrontare rilevanti rischi e costi in caso di cambiamento.

- **Disaster recovery**: fa riferimento a guasti o indisponibilità dei servizi di un determinato cloud provider; replicare i dati e le applicazioni su un secondo fornitore consente ai servizi di un'azienda di rimanere operativi

- **Esclusività del servizio**: nonostante ormai sia sempre meno frequente, un'azienda può scegliere di utilizzare il servizio esclusivo offerto da un particolare cloud provider, in quanto unico nel suo genere

È fondamentale tenere presente che i servizi dei diversi cloud provider, sebbene alcuni siano molto simili tra loro in quanto appartenenti alla stessa categoria (ad esempio AWS EC2 e Azure VMs per le VM o AWS EKS e Azure AKS come servizi gestiti per Kubernetes), sono totalmente indipendenti: doversi interfacciare su cloud diversi per un'azienda significa dover possedere le abilità e le conoscenze per essere in grado di gestire la complessità che viene necessariamente introdotta con l'utilizzo di più di un cloud provider.

1.4.2 Modelli di approccio ad architetture multi-cloud

Avere ben chiaro un piano strategico per sfruttare un approccio multi-cloud è fondamentale, dunque, per trarre il massimo vantaggio dall'architettura dell'ambiente e tenerne sotto controllo la complessità e i costi. Esaminare gli approcci comuni al multi-cloud e le motivazioni dietro di essi aiuta a effettuare la scelta corretta.

Per comprendere meglio le motivazioni che spingono le aziende a progettare un'architettura multi-cloud è bene elencare gli scenari più comuni, che descrivono le modalità di gestione dei workload da parte delle aziende

- Arbitrary: distribuzione di qualsiasi workload su qualsiasi cloud provider

- Segmented: distribuzione di workload specifici su determinati cloud provider; approccio chiamato *Composite* da Gartner

- Choice: distribuzione di workload specifici su un cloud provider specifico, vincolato da policy aziendali

- Parallel: distribuzione dello stesso workload su più cloud provider, per abilitare *high availability* o *disaster recovery*; approccio chiamato *Redundant* da Gartner

- Portable: disitrubuzione dei workload su un cloud provider, ma in modo tale da renderli portabili per il trasferimento su un provider diverso

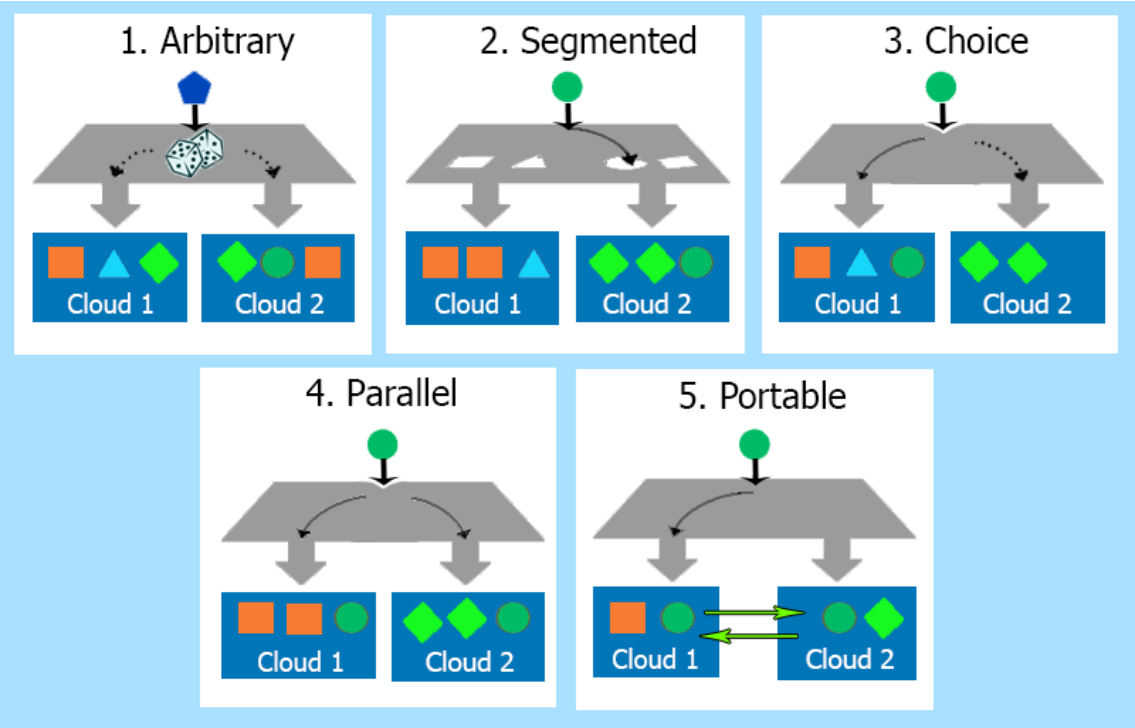

Figura 1.8. Scenari più comuni di utilizzo di un approccio multi-cloud.
Fonte: architectelevator [5]

Arbitrary

Un'enorme percentuale delle aziende che adottano un approccio multi-cloud utilizza il modello Arbitrary come soluzione, risultato di scarsa governance ed eccessiva influenza del fornitore. In questa situazione, dunque, si ha un insieme di workload presso il cloud 1 e un altro insieme presso il cloud 2, senza una precisa ragione o, molto spesso, per motivi di circostanza: succede infatti che, dopo aver iniziato con l'utilizzo di un cloud provider per ospitare i propri workload, un'azienda riceva degli sconti o dei crediti di benvenuto da un secondo cloud provider per cominciare ad utilizzare i suoi servizi e risorse.
L'approccio Arbitrary non è il risultato di una vera e propria strategia, ma spesso il comportamento di un'azienda, priva di tecnici dotati di formazione adeguata, che vuole muovere i primi passi verso il cloud e ha intenzione di farlo in fretta, nel frattempo guadagnando esperienza su diversi provider.

Segmented

Lo scenario Segmented prevede la distribuzione di un workload sul cloud provider più adatto. È spesso il risultato di preferenze di fornitori diverse per workload diversi, ad esempio a causa della migliore tecnologia utilizzata o di licenze più convenienti a parità di performance. Le scelte vengono effettuate su diversi fattori

- Tipo di workload, ad esempio *legacy* o moderno
- Tipo di dati, ad esempio confidenziali o pubblici

- Tipo di prodotto, ad esempio *data analytics* o software collaborativo o computazionale

Nel seguire questo approccio, è utile comprendere le componenti comuni tra le applicazioni in modo da non incorrere in eccessivi oneri in uscita.
Infine, una dettagliata e approfondita conoscenza da parte dei tecnici è fondamentale per non ricadere nello scenario precedente: succede spesso, infatti, che a causa dell'affinità dei servizi di diversi cloud provider, il team vendite del cloud 1 riesce a convincere ad adottare i propri servizi per altri workload o, di rado, a spostarli dal cloud 2 per beneficiare di sconti e promozioni.

Choice

Lo scenario Choice è lo scenario più comunemente adottato da aziende molto grandi che prevedono di supportare una grande quantità di business units e, di conseguenza, le rispettive preferenze IT. Spesso, una tale configurazione implica una relazione commerciale centrale e un framework comune per creare istanze sul cloud provider di maggiore preferenza, ma con la governance aziendale e i vincoli applicati.
Il vantaggio di questo scenario è che i diversi progetti sono liberi di utilizzare servizi cloud proprietari, come i database gestiti (a seconda del loro compromesso preferito tra evitare il lock-in e ridurre al minimo il sovraccarico operativo). Questa configurazione rappresenta, quindi, un buon passo iniziale per un'azienda che si approccia al multi-cloud.

Parallel

Sebbene l'opzione precedente dia la possibilità di scegliere tra i cloud provider, si è comunque vincolati dal tipo di servizio di quest'ultimo. Molte aziende cercano di distribuire applicazioni critiche su più CSP per garantire livelli di disponibilità più elevati di quelli che potrebbero ottenere con un singolo provider, magari anche con più zone di disponibilità. È molto comune anche l'utilizzo di un secondo cloud provider per raggiungere con la propria applicazione nuovi segmenti di mercato sfruttando le Region esclusive del secondo fornitore.
Distribuire la stessa applicazione su più cloud richiede un determinato disaccoppiamento dalle funzionalità proprietarie di quel fornitore. Ciò può essere ottenuto in diversi modi, ad esempio

- Gestendo funzioni specifiche per uno specifico provider (gestione identità, automazione dei deployment, monitoring, etc.) in modo separato dall'applicazione
- Utilizzando il più possibile componenti open source, i quali generalmente possono essere utilizzati su qualsiasi cloud provider. Se da un lato questo approccio va bene per servizi puramente computazionali (ad esempio Kubernetes), dall'altro può ridurre la possibilità di trarre vantaggio da servizi completamente gestiti, come monitoring o archiviazione dati. Dal momento che i servizi gestiti sono uno dei maggiori benefici della migrazione verso il cloud, è importante effettuare questa decisione con cautela

- Utilizzando un framework di astrazione multi-cloud, in modo da sviluppare l'applicazione una volta e distribuirla su qualsiasi cloud
- Mantenendo due branches per quei componenti dell'applicazione che sono specifici per un determinato cloud provider e sviluppando un'interfaccia comune da posizionare davanti; ad esempio, si può avere un'interfaccia comune per l'archiviazione dati a blocchi.

L'aspetto chiave a cui prestare attenzione è la complessità, che può facilmente annullare il guadagno previsto in *uptime*. Ulteriori livelli di astrazione e strumenti aumentano anche la possibilità di un'errata configurazione. Inoltre, se si distribuisce un'applicazione non funzionante su entrambi i cloud, è facile avere dei momenti di inattività, quindi è importante tenere conto dell'errore umano.

Portable

Uno dei vantaggi di adottare un approccio multi-cloud per i propri servizi è la portabilità tra i dversi provider, che implica la possibilità di distribuire i carichi di lavoro ovunque e spostarli in base alle esigenze. Il vantaggio principale è la possibilità di evitare il vendor lock-in.
Il meccanismo per abilitare la portabilità dei workload richiede un alto livello di automazione e di astrazione dai servizi cloud: mentre per le distribuzioni parallele il processo può essere gestito da un'installazione o distribuzione semi-manuale, la portabilità completa richiede di essere in grado di spostare il carico di lavoro in qualsiasi momento, il che richiede alti livelli di automatizzazione (i framework di astrazione multi-cloud, come ad esempio Anthos, semplificano questo tipo di setup).

La tabella di seguito riassume gli approcci, i driver principali e gli aspetti da tenere in considerazione nel valutare l'adozione di un approccio multi-cloud

Style	Key Capability	Key Mechanism	Consideration
Arbitrary	Deploying applications to the cloud.	Cloud skill	Lack of governance. Network traffic cost.
Segmented	Clear guidance on cloud usage.	Governance	Vendors may steer you back to "Arbitrary".
Choice	Support project needs and preferences; reduce lock-in	Common framework for provisioning, billing, governance	Yet-another layer of abstraction. Lack of guidance.
Parallel	Higher availability than single cloud.	Automation, abstraction, load balancing.	Complexity; under-utilization of cloud services;
Portable	Shift workloads as you please.	Full automation, abstraction. Data portability.	Complexity; Lock-in into multi-cloud frameworks.

Figura 1.9. Fonte: architectelevator [5]

1.5 Obiettivi della tesi

Dopo aver presentato, nei capitoli precedenti, in che modo le API rapprsentano un fattore chiave per una strategia di business di successo e in che modo possano aiutare ad accelerare il processo di sviluppo e distribuzione di un servizio, e dopo aver introdotto i modelli di approccio più comuni a livello Enterprise nell'utilizzo di più cloud provider, l'obiettivo della tesi è affrontare il problema della gestione delle API in un ambiente multi cloud, analizzandone le problematiche e in che modo una piattaforma di API Management può essere d'aiuto. Verrà presentato un caso d'uso comparativo, con l'obiettivo di confrontare due soluzioni di API Management, ovvero **Azure API Management** e **RedHat 3scale API Management**, in uno scenario multi-hybrid-cloud d'esempio; verranno svolti dei test di performance sulla componente "gateway" dei due prodotti, valutando, anche dal punto di vista delle funzionalità di security offerte, quale piattaforma risulta essere la più flessibile ed "adatta" in uno scenario multi-hybrid-cloud.

Capitolo 2

Strategie di gestione delle API su più piattaforme cloud

2.1 API Management in ambienti multi-hybrid-cloud e principali problematiche legate

Nel mondo odierno, incentrato sul cloud, una piattaforma di API Management si trova generalmente distribuita su un particolare cloud provider, in quanto viene venduta al cliente come un prodotto, un servizio di quel fornitore. Tuttavia, le applicazioni e i dati esposti dalle API gestite da un'azienda sono probabilmente distribuiti su più ambienti on-premise e cloud. Gestire queste API in modo coerente in tutta l'organizzazione è una parte fondamentale per aumentare la produttività e massimizzare il valore aziendale. Per fare ciò, è necessario avere un approccio unificato di gestione delle API in questo ambiente ibrido e multi-cloud.

Avere una piattaforma centralizzata di API Management in un singolo cloud provider, quindi, non è sempre la scelta migliore.

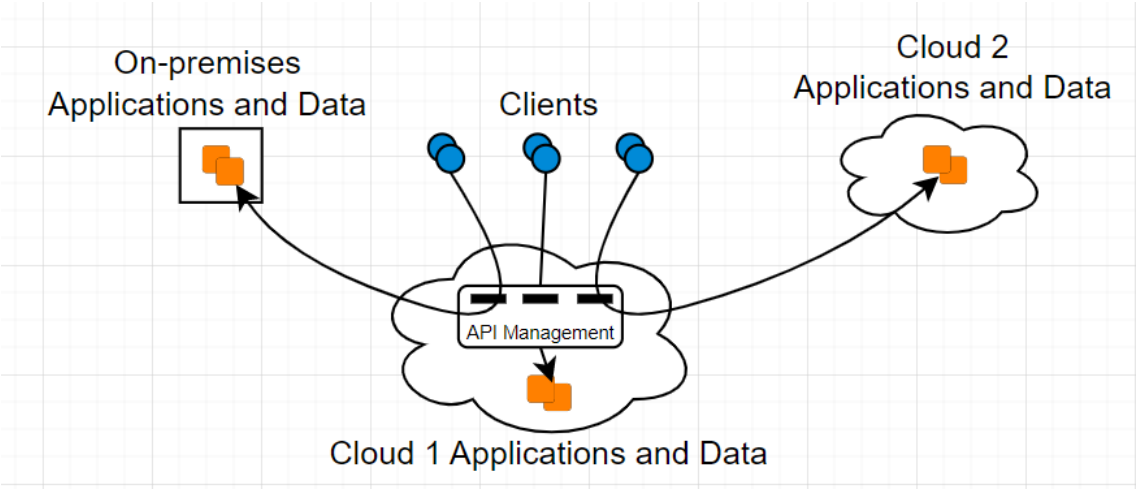

Figura 2.1. Fonte: Azure [15]

Come si può notare in Figura 2.1, tutte le richieste dei client verso le applicazioni backend passano dal cloud provider che fornisce il servizio di API Management. Per applicazioni e dati che si trovano nello stesso cloud (Cloud 1 nella figura), tale soluzione non crea problemi, in quanto quelle richieste devono comunque accedere a quel cloud, mentre per quanto riguarda le richieste ad applicazioni e dati che si trovano su cloud diversi (Cloud 2 in figura) o on-prem, esse, per ottenere un servizio di gestione di API, devono passare necessariamente attraverso il Cloud 1. Ciò solleva tre questioni importanti:

1. Come è possibile fare in modo che il traffico di un client che accede ad applicazioni e dati on-premise all'interno dello stesso environment non lasci tale environment? Possono esistere alcuni requisiti che obbligano tali informazioni a rimanere nella rete aziendale, o semplicemente l'accesso al cloud è proibito dalle policies dell'azienda. Qualunque sia il motivo, scenari come questo non possono prevedere l'utilizzo di un servizio di gestione delle API in esecuzione in qualsiasi cloud pubblico: il servizio deve essere eseguito in locale. Questo è il problema più significativo con la gestione delle API in molte organizzazioni, quindi è anche il più importante da risolvere.

2. Quando un client accede ad applicazioni e dati in un altro public cloud, è costretto a pagare la banda due volte. I cloud provider pubblici tipicamente addebitano il costo in base al traffico in uscita, quindi se una richiesta verso un backend situato nel Cloud 2 deve passare prima per il servizio di API management nel Cloud 1, entrambi i cloud providers terranno traccia di tale traffico, che verrà addebitato. Non si tratta di costi ingenti, ma sono comunque non necessari

3. Se il client e il backend sono nello stesso environment (entrambi nel Cloud 2 o entrambi on-premise), nell'architettura illustrata in figura viene aggiunta latenza inutile, in quanto le richieste passeranno attraverso il servizio di API management nel Cloud 1. Ancora una volta, la latenza addizionale è minima, ma non necessaria; tuttavia potrebbe essere un problema non trascurabile in scenari di business che utilizzano dei workload critici che richiedono latenza minima.

Al giorno d'oggi, sono tante le aziende che risolvono questi problemi servendosi di più piattaforme di API Management; questa strategia però aggiunge complessità e aumenta i costi, oltre a rendere molto complicate governance e discovery delle API in modo unificato.

2.2 La soluzione

Diversi fornitori hanno affrontato le problematiche descritte, spesso mosse direttamente dai clienti che utilizzavano il prodotto, come nel caso di Google ([12]-9:20), il quale ha aggiunto all'offerta di Apigee un nuovo piano di deployment e utilizzo, Apigee Hybrid, approfondito nel Capitolo 3.3.
In generale, la soluzione comune adottata dai fornitori di soluzioni di gestione delle

API è il disaccoppiamento della piattaforma stessa, fornendo una topologia in cui la componente di controllo e quella operativa sono separate: il primo, il **Control Plane**, si occupa di configurare il secondo, il **Data Plane**, che gestirà le chiamate in entrata in base alle istruzioni ricevute.

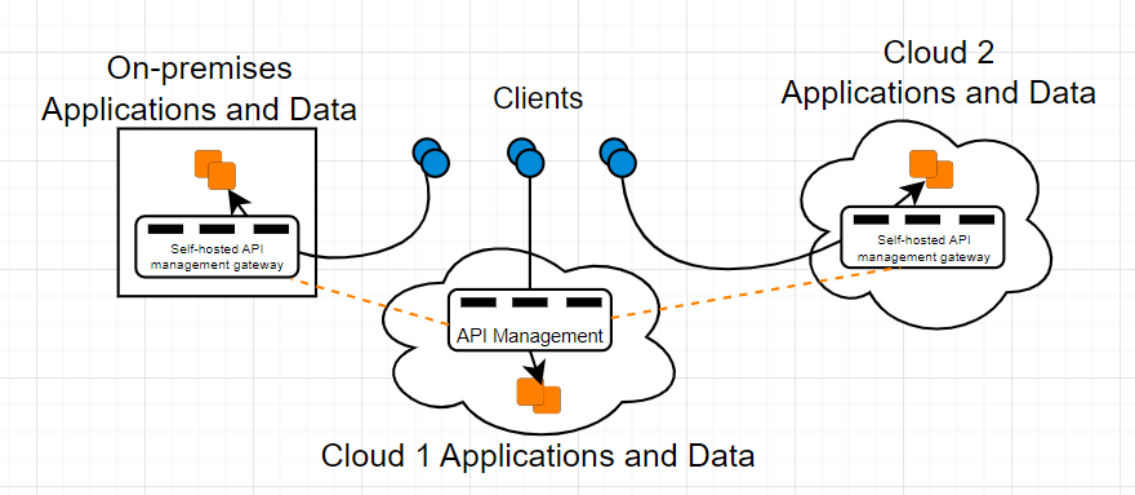

Figura 2.2. Fonte: Azure [15]

Osservando la Figura 2.2 è possibile notare che

- Il Data Plane e i servizi on-premise abilitano lo scenario hybrid cloud: ospitare il Data Plane nel datacenter on-premise consente ai client che si trovano nello stesso ambiente di contattare le applicazioni backend on-premise senza forzare il traffico a lasciare la rete aziendale, rendendo molto più semplice il rispetto delle policies. Le richieste, infatti, verrano intercettate dal gateway on-premise e reindirizzate verso i backend, senza interagire con il cloud provider

- Il Data Plane e i servizi nel Cloud 2 abilitano lo scenario multi cloud: ospitare il Data Plane in un altro cloud provider (Cloud 2) consente ai client di accedere ai servizi che si trovano nel Cloud 2 contattando direttamente il gateway stesso. In questo modo, le richieste esterne verranno intercettate dal gateway ospitato nel secondo cloud provider e reindirizzate verso i servizi ospitati presso quest'ultimo, eliminando sia la latenza aggiuntiva sia il costo extra dovuto all'accesso di due differenti cloud provider

2.2.1 Architettura di alto livello

Sebbene i metodi implementativi variano in base al prodotto preso in considerazione, è possibile rilevare tre componenti fondamentali in una piattaforma di gestione API in grado di risolvere i problemi esposti nel capitolo precedente e garantire dunque un valore aggiunto in ambiente Enterprise in termini di gestione centralizzata delle API verso servizi di più cloud provider.

Il Control Plane

Il Control Plane, chiamato anche *Management Plane*, è il punto di riferimento per la gestione centralizzata e unificata delle API; è composto da una serie di strumenti necessari per configurare le istanze dei Data Plane e di aggiungerle o eliminarle in base all'esigenza, gestire le API e amministrare gli utenti che le utilizzeranno; il Control Plane costituisce il pannello di controllo per la gestione del prodotto nella sua interezza. Solitamente è situtato in un cloud provider e viene offerto come risorsa/servizio di quel fornitore, ma in alcuni casi (come quello di RedHat 3scale, approfondito nel Capitolo 3.2) può essere installato nell'ambiente desiderato, rispettando determinati requisiti.

Il Data Plane

Spesso coincidente con l'API gateway (con alcune eccezioni, ad esempio Apigee di Google, il cui Data Plane è chiamato *runtime plane* ed è costituito da diversi elementi, tra cui anche il gateway), funge da proxy per le API realmente esposte dalle applicazioni, che costituiscono il backend; è il componente responsabile dell'esecuzione delle policies (se esistenti) e del routing delle richieste in entrata verso i servizi backend configurati. Riceve le configurazioni dal Control Plane, non appena esse vengono aggiornate, e viene contattato esternamente dai client tramite un endpoint pubblico. Il Data Plane, posizionandosi tra il cient e l'applicazione, rappresenta un *single-point-of-entry*, intercettando le richieste e consentendo di avere il massimo controllo su ogni chiamata API effettuata dai client. Dal momento che è un componente separato dal resto dell'architettura, può essere installato ovunque il cliente lo ritenga necessario (di solito, nello stesso ambiente che ospita i servizi, quindi on-premise o su un altro cloud provider).

Il developer portal

Gli utilizzatori delle API, ovvero gli Sviluppatori, possono documentarsi sulle API disponibili tramite l'apposito portale. Il Developer Portal fornisce la possibilità agli utenti di registrarsi e documentarsi su quali API sono autorizzati ad utilizzare e sulle loro configurazioni (rate limit, parametri necessari, servizi backend collegati), insieme a un set di tool per testare le chiamate e visualizzare analytics di base.

2.3 Attori coinvolti

Il setup e la gestione di una piattaforma di API Management e il conseguente utilizzo dei servizi tramite le API configurate coinvolgono principalmente due attori: l'**amministratore** e lo **sviluppatore**.

2.3.1 L'amministratore

Nel contesto di API Management l'amministratore è considerato il fornitore delle API. Si occupa della gestione dell'intera infrastruttura ed è responsabile di operazioni come creazione di un'API e configurazione con il servizio backend, installazione e configurazione del Data Plane nell'ambiente desiderato, scrittura delle policy e delle regole del Data Plane, monitoraggio dell'utilizzo delle API configurate, RBAC (*role-based access control*) per gli sviluppatori e creazione e gestione dei piani di utilizzo. Controlla tutte le operazioni dal **portale admin**, che contiene anche il Control Plane della piattaforma; si evince come una buona soluzione di API Management debba essere in grado di fornire all'amministratore strumenti potenti e efficaci per tutte le operazioni di cui è responsabile, semplificando il più possibile la QoL (Quality of Life) di questa figura.

2.3.2 Lo sviluppatore

Nel contesto di API Management lo sviluppatore è considerato il consumatore delle API. Il suo punto di riferimento è il **Developer Portal**, un portale creato appositamente per contenere tutte le informazioni di cui ha bisogno per utilizzare i servizi ai quali gli è stato garantito l'accesso; può registrarsi al portale autonomamente o tramite invito di un amministratore. Una volta registrato, riceve le credenziali necessarie per consumare le API (di solito delle stringhe che fungono da API keys), che possono essere testate e monitorate tramite le funzionalità dedicate.

2.4 Responsabilità in multi-hybrid-cloud e differenze con un ambiente single-cloud

Supponendo di avere un'architettura come quella in Figura 2.2, l'amministratore configura, dal Control Plane, tutti i Data Plane necessari, in base al tipo di servizio che un determinato gateway dovrà servire in un determinato ambiente (Cloud 1, Cloud 2 o on-premise). Lo sviluppatore si documenta dal Developer Portal su quali API sono disponibili e quali servizi sono raggiungibili tramite uno specifico Data Plane, e contatterà il gateway e l'API corretta per il servizio desiderato.

Le differenze in ambiente single cloud riguardano principalmente il Data Plane. Se nell'ambiente distribuito il client sceglieva quale gateway contattare in base alle API che quest'ultimo gestiva e dunque in base al servizio desiderato, in single cloud il gateway da contattare è uno solo. L'amministratore del servizio di API Management configura, dal Control Plane, il Data Plane, che in questo caso è unico e si troverà nello stesso cloud provider; lo sviluppatore, dopo essersi documentato dal Developer Portal sulle API presenti, contatta il Data Plane per tutti i servizi, in quanto è presente una sola istanza del gateway per gestire tutte le chiamate.

È importante notare che la possibilità di estensione da single a multi (o hybrid) cloud, mantenendo l'applicazione delle strategie di soluzione delle problematiche descritte nel Capitolo 2, è garantita solo se l'architettura della piattaforma di API Management è modulare, quindi costituita da Control Plane e Data Plane. In caso

contrario, l'utilizzo del servizio di API Management non consentirebbe di adottare tali strategie, lasciando irrisolti i problemi di latenza, policy e banda aggiuntiva. Di fatto, una soluzione come quella offerta da Amazon API Gateway [21], la cui architettura è costituita da un blocco unico senza possibilità di disaccoppiare Control Plane e Data Plane, non risponde ai problemi affrontati nel Capitolo 2, ricadendo nel caso presentato in Figura 2.1.

Capitolo 3

Introduzione ai principali sistemi di API Management

Il mercato offre una vasta scelta di piattaforme di gestione delle API, dalle proposte dei maggiori hyperscaler quali AWS (Amazon API Gateway), Azure (Azure API Management) e Google (Google Apigee API Platform) a prodotti come RedHat 3scale API Management, Kong gateway e Apache Apisix, ma solo alcuni sono in grado di portare un valore aggiunto alle aziende che scelgono di adottare i servizi di più Cloud provider.
Verranno ora passati in rassegna, individualmente, i principali prodotti di API Management con supporto al multi-cloud. Il criterio di selezione è effettuato sulla modularità dell'architettura (Control Plane e Data Plane), necessaria per l'estensione da single a multi-cloud. Verranno descritti l'architettura sottostante e le caratteristiche principali.

3.1 Azure API Management

Azure API Management è un servizio offerto da Azure come PaaS (Platform-as-a-Service), per supportare l'intero lifecycle di un'API.

3.1.1 Control Plane

Azure API Management è disponibile come risorsa Azure; può quindi essere creata direttamente dall'Azure Portal e di conseguenza il Control Plane, in questo caso chiamato *Management Plane*, è cloud-hosted.
Il Management Plane offre l'esperienza classica del portale Azure, integrandosi perfettamente con tutti i servizi Azure compatibili, come EventHub, Monitor, Application Insights, ServiceBus, KeyVault, Functions e Logic Apps. È possibile interagire con il Control Plane tramite l'Azure portal, ma anche attraverso Azure PowerShell, Azure CLI, SDK client disponibili in diversi linguaggi e anche grazie all'estensione per Visual Studio Code dedicata: ciò consente di gestire programmaticamente tutte le configurazioni e le capabilities del prodotto, a partire dalla sua creazione e distruzione tramite gli ARM templates.

Il Management Plane di Azure API Management è il punto di riferimento centralizzato per qualsiasi operazione da effettuare lato amministratore, come creare e configurare i vari servizi, definire o importare un'API da una vasta gamma di sorgenti quali specifiche OpenAPI o backend WebSocket e GraphQL, gestire configurazioni e policies per i gateway, osservare i log e le metriche delle risorse, gestire gli utenti e il Developer Portal.

Il Developer Portal è totalmente open-source, è generato automaticamente ed è completamente personalizzabile; è il punto di riferimento per gli utilizzatori delle API ed è anche possibile ospitarlo autonomamente, on-prem o su un altro cloud provider.

3.1.2 Data Plane

Quando si utilizza Azure API Management, si devono gestire due tipi di gateway

- Cloud gateway: è l'istanza del gateway nel cloud, disponibile di default per ogni istanza di Azure API Management, nella stessa Region della risorsa principale; viene gestita direttamente dall'Azure Portal
- Self-hosted gateway: è l'istanza del gateway separata dal resto dell'infrastruttura; è opzionale e consente di abilitare scenari di hybrid cloud e multi cloud. "Self-hosted", in quest'ambito, significa che è l'azienda la responsabile dell'installazione e il mantenimento del SHG ovunque necessario. Principali caratteristiche del SHG
  - È distribuito come immagine Docker: può essere deployato come container a sé stante o su Kubernetes per sfruttare vantaggi e funzionalità ad esso collegati, come ad esempio un deployment in alta affidabilità
  - Necessita solamente di una connessione outbound verso il cloud service principale: evita potenziali rischi di security nell'aprire una connessione inbound, ma consentendo comunque una comunicazione in entrambi i sensi (per esempio, il SHG può essere configurato per inviare logs al cloud service principale)
  - Di default, il SHG non possiede nessun local storage: viene configurato dal Management Plane, e tali configurazioni vengono salvate in memoria; se per qualche motivo il gateway viene riavviato (per esempio in seguito a un crash) e perde la connessione con il cloud service principale, perde anche tali informazioni di configurazione. Per evitarlo, si può configurare un local persistent storage che mantiene un backup dei dati necessari al corretto funzionamento e delle configurazioni.

L'installazione del Data Plane in ambienti containerizzati e un'operazione molto semplice e veloce: il portale admin, infatti, nella sezione Gateways, dopo aver creato un *cloud placeholder* per il gateway da installare nell'ambiente desiderato, mette a disposizione dei comandi e dei file di configurazione autogenerati con tutti i minimi parametri necessari al deployment. Ciò garantisce un'esperienza plug-and-play all'amministratore, che non deve far altro che eseguire il comando o applicare

il file nell'ambiente desiderato. Tra i parametri preconfigurati, è presente anche l'endpoint che il gateway utilizzerà per instaurare la connessione con il Control Plane.

Il punto di forza di Azure API Management è senza dubbio la possibilità di configurare il servizio in modo programmatico, grazie a Azure PowerShell e le SDK, e l'integrazione con altri servizi Azure compatibili, come quelli dedicati per il monitoring, reporting e alerting e i workload come LogicApps e Functions, garantendo in modo nativo l'accesso a numerose funzionalità dell'ecosistema Azure.

3.2 RedHat 3scale API Management

Red Hat 3scale API Management è una piattaforma di API Management che semplifica la condivisione, sicurezza, distribuzione, controllo e monetizzazione delle API. Fa parte del set di tecnologie RedHat Integration.
3scale API Management è progettato per supportare e scalare architetture ibride (on-premise, nel cloud, o una combinazione delle due).

3.2.1 Control Plane

Il Control Plane rappresenta l'elemento centrale per la gestione dell'intero prodotto. Si compone di

- API Manager: servizio responsabile di operazioni quali l'autorizzazione e il report del traffico da e verso il Data Plane
- Developer Portal: il portale sviluppatori, personalizzabile in base alle esigenze; è il punto di riferimento per l'utilizzatore delle API, lo sviluppatore, che può registrarsi al portale autonomamente o su invito dell'amministratore. Il portale consente di documentarsi sulle API disponibili per l'utente autenticato, effettuare dei test e avere a portata di mano delle analytics sull'utilizzo dell'API e sui codici di risposta
- Admin Portal: il componente principale dal quale l'amministratore gestisce l'intera infrastruttura, dalla creazione degli elementi che costituiscono una API (Application plan, Product, Backend) alla configurazione dei gateway, dall'analisi delle statistiche per ogni API alla gestione degli utenti

In base al tipo di offerta scelto, il Control Plane può essere managed o self-hosted. 3scale viene offerto in tre opzioni:

- 3scale.net: l'offerta SaaS
- 3scale: offerta self-managed; richiede un cluster OpenShift su cui essere installato ed è disponibile usando un template o tramite l'operator
- RedHat OpenShift API Management: offerta managed; disponibile come addon (servizio gestito) per OpenShift Dedicated, ovvero l'offerta managed di OpenShift

3.2.2 Data Plane

Il gateway è un componente separato dal resto dell'infrastruttura, e può essere installato su un cluster OpenShift esistente contenente 3scale o in altri ambienti grazie all'immagine Docker fornita; ovunque si trovi, l'istanza del gateway deve essere connessa al Control Plane, in modo da consentire l'instaurarsi di una connessione per lo scambio di aggiornamenti di configurazione e eventuali report, se configurati.
Il componente gateway è chiamato **APIcast**; è basato su NGINX e viene usato per integrare servizi API interni ed esterni con la piattaforma 3scale, con la quale deve essere connesso; effettua un load-balancing round-robin. Viene offerto in diverse opzioni:

- Embedded APIcast (solo 3scale self-hosted): sono disponibili due gateway APIcast, pre-configurati e pronti all'uso, denominati *staging* e *production*. Il primo viene usato solo per configurazione e test dell'integrazione delle API; quando il comportamento è quello desiderato, si passa all'ambiente di produzione utilizzando l'stanza *production*.

- Hosted APIcast (solo 3scale SaaS): un'istanza di APIcast disponibile in cloud; è già installata ed è limitata a 50k req/giorno e 20 req/secondo (in caso fosse necessario superare questo limite, è possibile passare all'istanza self-hosted di APIcast)

- Self-managed APIcast: può essere installato ovunque ce ne sia bisogno, in ambienti quali

  - Docker containerized environment: API cast è disponibile come immagine Docker pronta all'uso, la quale include tutte le dipendenze necessarie per essere usata come container Docker
  - RedHat OpenShift: APIcast può essere installato su una versione supportata di OpenShift, direttamente dalla console

  Anche Hosted APIcast e Self-managed APIcast, come Embedded APIcast, sono disponibili in doppia istanza per gli environment di *staging* e *production*.

  La connessione tra Data Plane e Control Plane avviene configurando la variabile d'ambiente `THREESCALE_PORTAL_ENDPOINT` del gateway, che all'avvio stabilirà una connessione con il Control Plane.

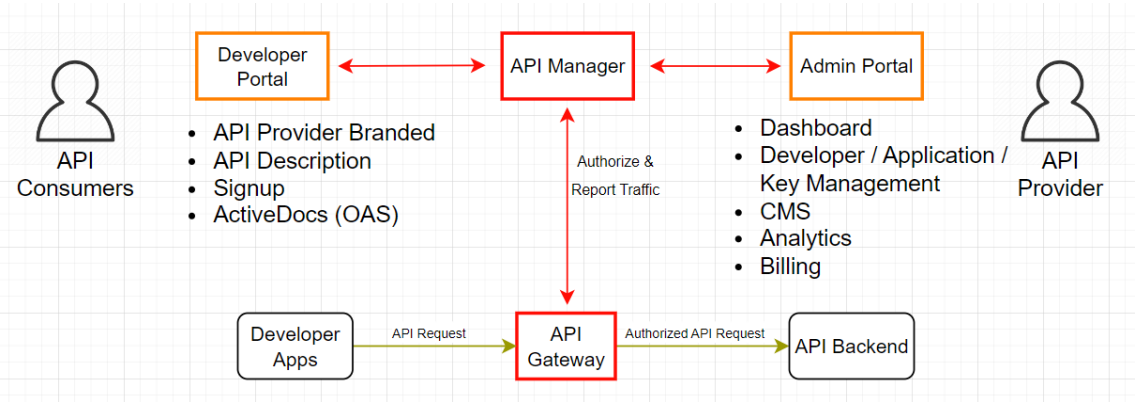

Figura 3.1. Architettura di 3scale. Fonte: RedHat [10]

3.3 Google Cloud Apigee API Platform

Google Cloud Apigee API Platform è la soluzione proposta da Google per lo sviluppo e la gestione di API. L'offerta di Apigee era originariamente costituita da una completamente SaaS e una completamente on-premise; nel tempo sono state raccolte, da parte del cloud provider, le diverse preoccupazioni mosse dai clienti, le quali si concentravano particolarmente su aspetti di latenza, sicurezza, personalizzazione e overhead, già affrontati nel capitolo 2.1. Accogliendo queste richieste, Google ha sviluppato una nuova versione di Apigee, chiamata Apigee hybrid, che consente di disaccoppiare Control Plane, chiamato *Management Plane* e ospitato su Google Cloud, e Data Plane, chiamato *runtime plane*, gestito dal cliente e ospitato in un environment compatibile di sua scelta.
La nuova topologia modulare di Apigee ha così aperto la strada a scenari di cloud ibrido e multi cloud, adattandosi a nuove esigenze Enterprise.

3.3.1 Control Plane

Il Management Plane è ospitato e gestito da Google nel cloud. Include una UI, RBAC e analytics, che consentono di amministrare l'intera infrastruttura, e ha il compito di inviare istruzioni ai runtime planes, ospitati e gestiti dall'utente.

Opzionalmente è possibile attivare anche il Developer portal, che supporta tutte le funzionalità di base (documentazione e test delle API, registrazione dell'utente, analytics). È disponibile in due varianti

- Integrated portal: generato automaticamente e istantaneamente; non personalizzabile
- Drupal-based portal: consente di creare un portale personalizzato grazie a Drupal 9, un tool open-source di gestione dei contenuti (CMS - Content Management System)

3.3.2 Data Plane

La filosofia di Google, alla base dello sviluppo del Data Plane, è quella di "non reinventare la ruota" e utilizzare componenti e funzionalità già esistenti come l'autoscaling, per sfruttarne al massimo i vantaggi. Per questo motivo, il runtime plane di Apigee hybrid può essere esclusivamente utilizzato su Kubernetes, connettendo ad Anthos i cluster compatibili.
Il runtime plane è costituito da diversi componenti

- Istio gateway (routing e load balancing): una versione personalizzata di Istio, una piattaforma open source di service mesh; il motivo di tale scelta è riscontrabile nel fatto che quest'ultimo risolve già diverse problematiche relative a load balancing, routing intelligente e terminazione TLS.

- Message processor: l'API gateway, identico a quello presente nel cloud; può essere ospitato on-premise

- Synchronizer: un agent in grado di replicare i dati tra il Management Plane e il luogo dove è ospitato il runtime plane; necessario per sincronizzare le configurazioni e fare in modo che le modifiche effettuate sul Control Plane vengano applicate nel Data Plane corrispondente

- Cassandra: il meccanismo di persistenza per il gateway locale (API products, access tokens, API keys, cache, etc.); installato solo localmente, non viene mantenuta una copia di questi dati nel cloud. Cassandra rappresenta il livello di persistenza del runtime plane, abilitando meccanismi di *failover* in caso di mancata connessione verso il Management Plane.

- MART (Management API for Runtime data): aiuta a gestire alcune entità necessarie al runtime plane, come API products, API keys, access tokens e cache

- UDCA (Universal Data Collection Agent): un servizio che estrae periodicamente analytics, dati di debug e deployment, successivamente inviati al servizio UAP (Unified Analytics Platform) nel Management Plane e processati, in modo da renderli disponibili nella UI. Questo servizio è progettato per inviare i dati raccolti a più destinazioni, dando la possibilità all'amministratore di configurare altri servizi cloud per lo storage e il processing dei dati (Google Cloud Storage, Simple Storage Service o Data Lake).

Quando l'utente installa il runtime plane in un environment, è responsabile della gestione di alcuni dei componenti di cui è costituito, ovvero Message Processor, Synchronizer, Cassandra e MART; gli altri componenti sono gestiti dal Management Plane in modo automatico.

Nella Figura 3.2 è schematizzata l'architettura del runtime plane di Apigee hybrid: ogni riquadro grigio rappresenta un pod in Kubernetes, e il load balancer Istio è il point-of-entry della singola istanza del runtime plane. Il load balancer Istio divide il carico delle richieste sui message processor, che a sua volta contatta

il servizio backend in modo che l'API venga servita. Il componente Analytics raccoglie dati grezzi da ogni gateway e le invia in modo asincrono nel cloud.
La maggior parte delle applicazioni installate sono stateless; l'unico componente stateful è Cassandra. Quest'ultimo, quando ospitato su un cloud provider (GCP, AWS o Azure) è in grado di effettuare un backup automatico sfruttando la funzionalità offerta dal servizio dedicato in quel CSP.

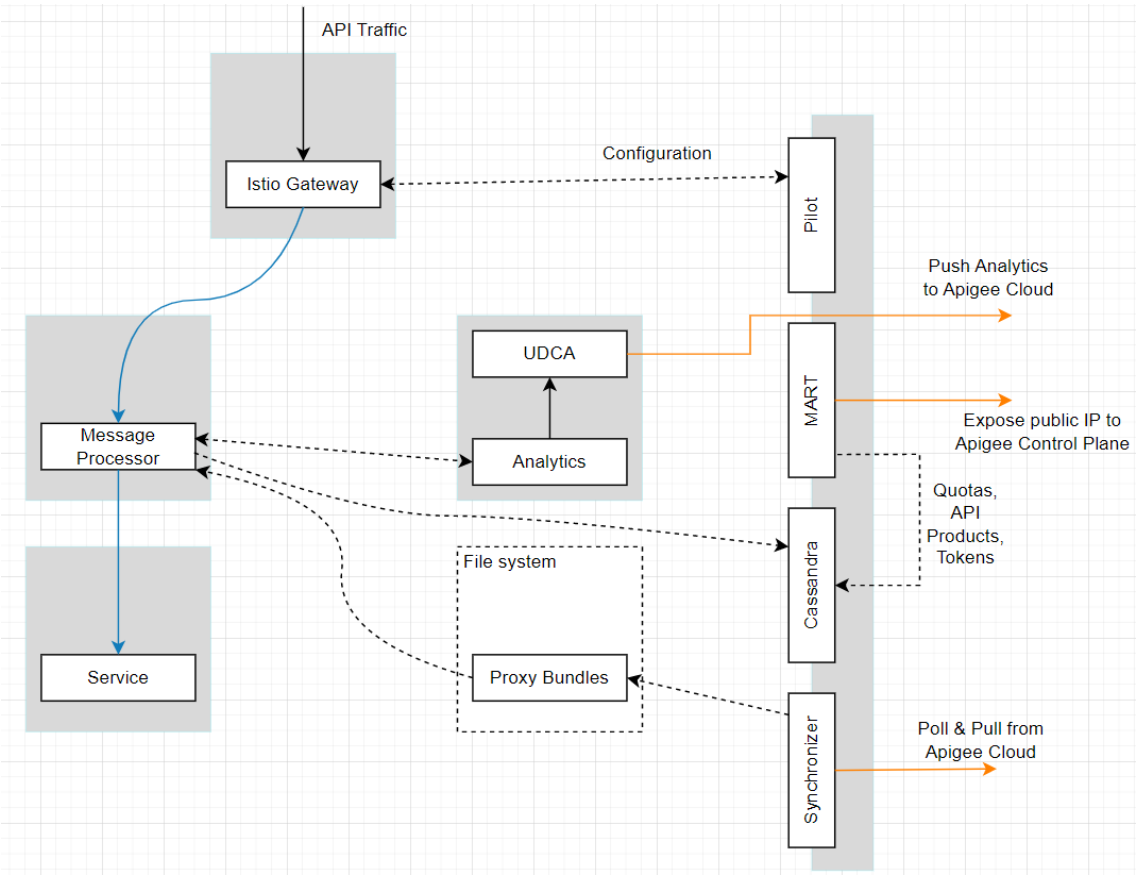

Figura 3.2. Architettura del runtime plane di Apigee hybrid. Fonte: Google Apigee [12]-22:38

3.4 Kong API Gateway

Kong API gateway è un API gateway open-source progettato per architetture decentralizzate, inclusi ambienti hybrid-cloud e multi-cloud, e per essere estensibile. È un'applicazione LUA realizzata su OpenResty, ovvero un server proxy NGINX esteso per eseguire degli script LUA. Il core di Kong gateway implementa il routing, l'astrazione di un database e la gestione di plugin.

Le capabilities integrate di Kong gateway sono abbastanza ridotte, limitandosi a funzioni di autenticazione e autorizzazione, controllo del traffico e monitoring, ma grazie al supporto ai plugin è possibile estendere la configurazione base e aggiungere le funzionalità desiderate seguendo un approccio modulare, permettendo dunque l'attivazione e disattivazione delle stesse in base alle necessità.
Kong mette a disposizione una sezione dedicata ai plugin, chiamata Plugin Hub, dalla quale è possibile selezionare le estensioni desiderate, la cui quantità varia in base al tier selezionato (Free, Plus, Enterprise).
Kong gateway può essere gestito tramite Admin API, configurazione dichiarativa oppure Kong Manager; quest'ultimo utilizza le Admin API per amministrare e controllare il gateway.

Kong ha due modalità di funzionamento

- Con database: usa un database per memorizzare le configurazioni; in questa modalità Kong può essere configurato usando le Admin API o in modalità dichiarativa attraverso dei file di configurazione (.yaml)
- Senza database (DB-less mode): memorizza la configurazione in memoria, sul nodo corrente. In questo modo le Admin API sono read-only, ed è necessario gestire Kong usando una configurazione dichiarativa (.yaml); in questa modalità, se Kong è stato installato in un ambiente Kubernetes, le configurazioni sono memorizzate in *etcd*.

Kong supporta una grande varietà di modalità di installazione: può essere installato su diversi sistemi operativi quali CentOS, Amazon Linux, Ubuntu e RHEL, oppure su ambienti di containerizzazione come Docker, Kubernetes, Openshift grazie all'immagine Docker pronta all'uso.

Originariamentetutte le informazioni di configurazione di Kong (Routes, Services, plugins, etc.) venivano memorizzate nel database integrato. Nelle release future, è stata implementata la modalità ibrida, disponibile con i tier Enterprise e Plus del prodotto: viene introdotta una nuova topologia, che prevede una divisione in Control Plane e Data Plane. Un'istanza di Kong gateway viene utilizzata come Control Plane, quindi effettua solo operazioni di gestione e amministrazione, mentre una o più istanze fungono da Data Plane, utilizzate come proxy per le richieste.

3.4.1 Control Plane

Il Control Plane ha il compito di configurare i nodi che fungono da Data Plane connessi ad esso e, dopo aver interrogato il database, distribuirle periodicamente ad essi così da tenerli sempre aggiornati, in modo consistente.

Il Control Plane può essere utilizzato in modalità self-managed grazie alle offerte Enteprise e Plus di Kong gateway, oppure viene offerto come servizio totalmente gestito grazie a Kong Konnect, la piattaforma cloud PaaS centralizzata per il supporto completo alla gestione del lifecycle delle API.
Per utilizzare un Control Plane self-hosted, occorre generare un certificato o una coppia asimmetrica di chiavi crittografiche, necessari per l'instaurazione di un handshake TLS mutuo (mTLS) per garantire una comunicazione sicura tra Control Plane e Data Plane. Sono supportate due modalità

- *shared mode*: viene generato un certificato/coppia di chiavi *self-signed* tramite la Kong CLI
- *PKI mode*: si fornisce un certificato/coppia di chiavi firmate da una Certification Authority (CA).

Infine occorre installare un'istanza di Kong gateway (le modalità variano in base alla piattaforma scelta) e applicare le configurazioni per la modalità ibrida, in modo da far funzionare l'istanza di Kong appena creata come Control Plane. Tale configurazione avviene fornendo un file di configurazione denominato `kong.conf`, contenente i parametri `role` (da settare al valore `control_plane`) e i path per il certificato/coppia di chiavi configurati precedentemente. Verranno poi aggiunte successivamente tutte le istanze di Data Plane ritenute necessarie.

Il Developer Portal è incluso in Kong Konnect, mentre in Kong Gateway è disponbile solo con il tier Enterprise; viene utilizzato dagli sviluppatori per documentarsi, accedere e consumare i servizi tramite le API, nonchè effettuare dei test sugli endpoint e gestire le proprie credenziali. Il livello di personalizzazione è molto limitato.

3.4.2 Data Plane

L'istanza di Kong che funge da Data Plane ha il ruolo di proxy per le richieste, effettuando il routing delle stesse verso i servizi di backend dopo aver applicato eventuali policies configurate. Il Data Plane mantiene una cache in cui viene memorizzata la versione più aggiornata delle configurazioni, in modo che, nel caso in cui il Control Plane smetta di funzionare, il Data Plane sia in grado di continuare a fare da proxy per le richieste.

La configurazione di un nodo Data Plane avviene seguendo una procedura simile a quella per la configurazione del Control Plane: si può fornire un file denominato `kong.conf` contenente i parametri opportuni, come la variabile `role` settata a `data_plane`, i path per il certificato/coppia di chiavi configurati precedentemente e, inoltre, l'endpoint del Control Plane nella varaibile `cluster_control_plane`.

Health check

La peculiarità di Kong gateway è un meccanismo di *health check*, utilizzato dal gateway per determinare se una API è *healthy* o *unhealthy*. Esistono due modalità

- Active check: vengono periodicamente inviate delle richieste HTTP o HTTPS all'endpoint e la salute dello stesso è determinata in base alla risposta ricevuta. Il vantaggio è che, quando un endpoint torna attivo dopo essere stato spento, questo viene automaticamente individuato dall'health check; lo svantaggio è che viene generato del traffico addizonale.

- Passive check (ciruit breaker): Kong analizza le risposte generate conseguentemente alle richieste da parte dei client; quando un target non risponde, Kong lo etichetterà come *unhealthy*. Il vantaggio è che non viene generato traffico aggiuntivo, ma quando un endpoint torna attivo dopo essere stato spento quest'ultimo deve essere configurato manualmente dall'amministratore tramite le Admin API.

Kong utilizza quattro fattori per determinare se una API è *healthy* o meno, basandosi su altrettanti contatori che vengono incrementati o resettati in base alla risposta e al tipo di errore

- Success: incrementato se non sono stati individuati errori di alcun tipo, vengono resettati tutti gli altri

- TCP failure: incrementato in presenza di errori di connessione; "Success" viene resettato

- Timeout: incrementato allo scadere di un timeout; "Success" viene resettato

- HTTP failure: incrementato in presenza di codice di risposta *unhealthy*; "Success" resettato

Non appena uno degli ultimi tre contatori raggiunge la soglia configurata, l'endpoint verrà contrassegnato come *unhealthy*; se il contatore "Success" raggiunge la soglia configurata, l'endpoint verrà contrassegnato come *healthy*.

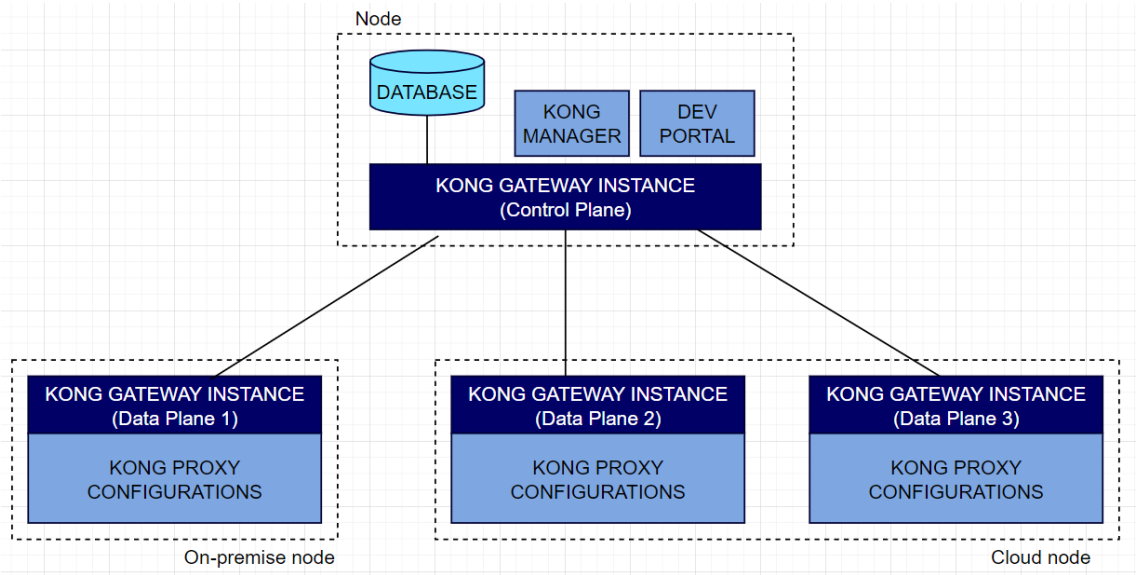

Figura 3.3. Architettura di Kong hybrid. Fonte: Kong [11]

La tabella 3.1 rappresenta un riepilogo delle caratteristiche dei prodotti di API Management analizzati in questo capitolo

Prodotto	**Control Plane**	**Data Plane**	**Developer Portal**	**Note**
Azure API Management	Solo cloud-hosted; integrazione con servizi Azure dedicati (es. EventHub, LogicApps, Monitor, Application Insights, KeyVault)	Disponibile in cloud (default, istanza gestita) e come immagine Docker (self-hosted gateway, istanza self-managed); disponibilità di template preconfigurati per il deployment	Open-source, generato automaticamente e completamente personalizzabile	–
RedHat 3scale API Management	Disponibile in modalità managed o self-managed, in base all'offerta	Disponibile in diverse versioni gestite (in base all'offerta) e come immagine Docker (istanza self-managed)	Generato automaticamente, offre scarsa personalizzabilità	Il Control Plane richiede OpenShift
Google Cloud Apigee API Platform	Solo cloud-hosted	Architettura complessa, costituita da diversi componenti specializzati in determinate funzioni (routing e load-balancing, sincronizzazione con Control Plane, persistenza dei dati, analytics reporting)	Disponibile in due varianti: Integrated (generato automaticamente, non personalizzabile) e Drupal-based (personalizzabile tramite Drupal 9)	Il Data Plane richiede Kubernetes
KONG API Gateway	Disponibile in modalità managed (Kong Konnect) o self-managed (istanza di KONG che effettua solo operazioni di gestione e amministrazione)	Istanza di KONG che svolge funzionalità di routing, load-balancing e policy enforcement	Disponibile con Kong Konnect (offerta managed) o con il tier Enterprise di Kong Gateway (offerta self-managed)	Capabilities di default limitate, estendibili tramite plugin; funzionalità di health check per le API

Tabella 3.1. Riepilogo delle caratteristiche dei prodotti di API Management presentati

Capitolo 4

Analisi e confronto di due solzioni di API management in uno scenario multi-cloud

Verranno ora presi in considerazione per l'analisi e il confronto due prodotti: **Azure API Management** e **RedHat 3scale API Management**. La ragione di tale scelta è legata all'intenzione di confrontare i due prodotti più simili tra di loro a livello architetturale, ovvero costituiti da un Control Plane progettato e sviluppato esclusivamente per operazioni di gestione e amministrazione (a differenza di Kong, dove il Control Plane non è altro che un'istanza del gateway che svolge funzioni di controllo) e da un Data Plane semplice e portabile, distribuito come immagine Docker (al contrario di Google Apigee, il cui Data Plane è un progetto complesso costituito da più componenti separati); in questo modo si è cercato di evitare il più possibile che i risultati dei test e delle analisi venissero influenzati dall'architettura interna dei prodotti.
Al fine di confrontare le prestazioni dei gateway dei due prodotti ed effettuare dunque dei test di performance, è stato realizzato un caso d'uso, nell'ambito dell'esperienza maturata all'interno di **Blue Reply**, illustrato in questo capitolo. Sarà descritta l'infrastruttura e saranno evidenziati costi, requisiti e dettagli tecnici di implementazione. Il caso d'uso è stato valutato in un contesto multi-hybrid-cloud, all'interno di uno scenario "segmented": la scelta di tale scenario è pilotata dalle esigenze dello use case, ovvero l'utilizzare workload specifici in base alle necessità del cliente; scegliere uno scenario differente non comprometterebbe la significatività dei risultati ottenuti, in quanto non sono legati a tale scelta.
Verrà poi effettuata una seconda analisi, svolta su 5 driver di security, al fine di confrontare le soluzioni e valutare quale delle due risulta essere più vantaggiosa e adatta ad un contesto multi-cloud.

4.1 Introduzione al problema

Un big cliente possiede un'applicazione i cui servizi vengono sviluppati da diversi team e forniti tramite dei workload su più cloud provider e on-premise. La scelta di

utilizzare i servizi proprietari dei CSP è stata dettata da esigenze interne all'azienda, al fine di utilizzare il servizio esclusivo più adatto al workload da eseguire e di minimizzare i costi.

- **Una Lambda su AWS**: il team di sviluppo ha necessità di utilizzare FaaS (Function as a Service), il cui codice subisce numerosi cambiamenti nel tempo; si è scelto di utilizzare il servizio di AWS in quanto garantisce un *cold start* notevolmente inferiore rispetto alle Azure Functions.
- **Una Logic App su Azure**: un secondo team, più orientato verso il *block coding* e con una solida esperienza nel mondo Azure, ha necessità di utilizzare un PaaS che permetta di scrivere meno codice possibile e di utilizzare altri servizi Azure: Logic Apps infatti offre dei blocchi built-in per risorse quali ServiceBus, Blob Storage, Office365 Outlook, Event Hub e molti altri.
- **Un json server on-prem**: un team è reponsabile del controllo di dati sensibili che devono rispettare precise policies dell'azienda; da qui la decisione di mantenere il workload on-prem e gestirne la manutenzione all'interno del perimetro aziendale.

Tali servizi/dati vengono esposti tramite delle API.

In un contesto tradizionale, gli utenti, dopo essere stati notificati dell'esistenza o dell'aggiornamento di un servizio, devono chiamare singolarmente l'API corrispondente, generando non poca confusione in fase di documentazione e utilizzo del servizio stesso. Trattandosi di workload eterogenei distribuiti su diversi CSP, aspetti essenziali come l'autenticazione dovrebbero essere gestiti da un servizio esterno o in modo diverso separatamente per ogni servizio, in base al CSP in cui si trova. Una soluzione di API Management consente di centralizzare l'accesso a tutti i servizi, fornendo un sistema di controllo centralizzato dal quale gestire tutti gli aspetti di configurazione e manutenzione. Inoltre, grazie al Developer Portal, un utente può registrarsi e accedere ad un unico portale che raccoglie tutte le API che è autorizzato ad usare, nonchè la loro documentazione e la modalità di utilizzo.

Vediamo in che modo i due prodotti, oggetto di studio di questa tesi, possono essere utilizzati per garantire benefici agli attori coinvolti, e soprattutto quale valore aggiunto sono in grado di portare al tipo di scenario multi-hybrid-cloud preso in esame.

4.2 Soluzione proposta

Si illustra di seguito la soluzione proposta, descrivendone l'infrastruttura e evidenziandone requisiti e costi.

4.2.1 Descrizione

Per l'analisi e il confronto delle due soluzioni, sono stati organizzati 3 cluster per servizi containerizzati, completamente self-managed, in modo da avere il controllo

completo sulle macchine utilizzate e spegnerle quando non utilizzate, riducendo notevolmente i costi. I cluster ospitano le risorse dei prodotti studiati e, nel caso dell'ambiente on-prem, anche il workload del rispettivo servizio.

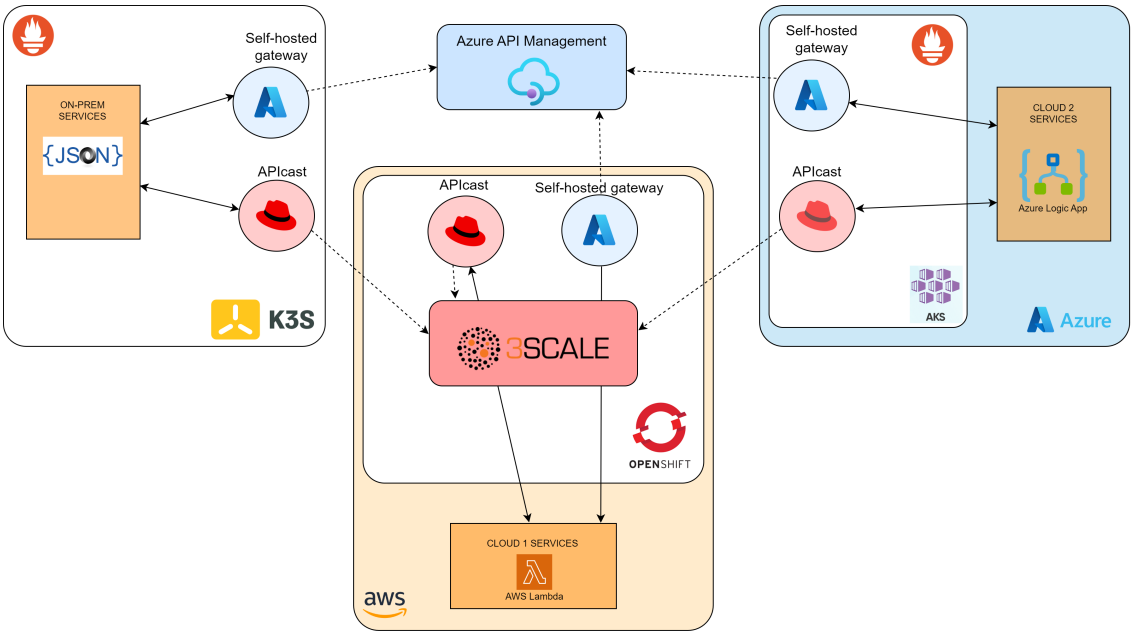

Figura 4.1. Architettura di alto livello dell'infrastruttura realizzata

- OCP su AWS: ospita il Control Plane di 3scale più i gateway di entrambe le soluzioni, responsabili di effettuare il routing verso il workload servito tramite AWS Lambda
- AKS su Azure: ospita i gateway di entrmbe le soluzioni, responsabili di effettuare il routing verso il workload sevito tramite Azure Logic App
- K3S on-prem: ospita i gateway di entrambe le soluzioni, più il workload servito da un *json server*; i gateway sono responsabili del routing verso il servizio interno al cluster

I gateway, dovendo essere raggiungibili dall'esterno, avranno un endpoint esposto per ricevere le richieste dei client. Nella Figura 4.1 è possibile osservare il disegno di archiettura di alto livello della soluzione proposta: le linee nere continue indicano la comunicazione tra gateway e backend in seguito a una richiesta del client, proveniente dall'esterno. Le linee tratteggiate, invece, sono le connessioni in uscita che il Data Plane utilizza per contattare il Control Plane per diverse necessità, la più importante delle quali è la richiesta di un aggiornamento di configurazione. È importante notare come il differente tipo di linea corrisponde alla differente natura della comunicazione: una linea continua corrisponde a una comunicazione scatenata da una richiesta esterna, mentre una linea tratteggiata ad una comunicazione automatica o semi-automatica effettuata dai gateway periodicamente.

4.2.2 Requisiti

Occorre dapprima analizzare i requisiti richiesti dai componenti delle soluzioni, ovvero il **Control Plane** e il **Data Plane**.

- Control Plane
  - Azure: il Control Plane di Azure API Management viene fornito esclusivamente come servizio gestito; è disponibile e utilizzabile come qualsiasi altra risorsa Azure, direttamente dall'Azure Portal. È stato scelto il Tier Developer, ideale per use-cases valutativi e con la possibilità di aggiungere i self-hosted gateway.
  - 3scale: il Control Plane di 3scale è disponibile sia come SaaS sia come risorsa self-managed; nel secondo caso, è necessario avere un cluster OpenShift operativo con licenza attiva. È stata scelta un'installazione (OCP + 3scale) completamente self-managed in quanto, a differenza dell'alternativa gestita, offre pieno controllo sulle macchine virtuali e consente di spegnerle quando non utilizzate, riducendo notevolmente i costi.
    I requisiti hardware minimi per il Control Plane di 3scale sono i seguenti

Risorsa	**Quantità**
CPU	8 vCPU (Compute Optimized)
Memoria	16 GB
Storage	* 3 RWO (ReadWriteOnce) persistent volumes per persistenza Redis e MySQL * 1 RWX (ReadWriteMany) persistent volumes per persistenza Developer Portal e assets di sistema

Tabella 4.1. Requisiti hardware minimi per RedHat 3scale

mentre i requisiti minimi per un cluster OpenShift Container Platform sono i seguenti

è evidente che i requisiti hardware verticali di 3scale superano quelli di OpenShift Container Platform; pertanto, in fase di installazione di quest'ultimo, è stato necessario aumentare le risorse delle macchine utilizzate adattandole ai requisiti imposti da 3scale.

Macchina	**OS**	**vCPU**	**RAM**	**Storage**	**Note**
Bootstrap	RHCOS	4 vCPU	16 GB	100 GB - 300 IOPS	Provvisorie
Control Plane	RHCOS	4 vCPU	16 GB	100 GB - 300 IOPS	Minimo 3 nodi
Compute (workers)	RHCOS, RHEL 8.4, or RHEL 8.5	2 vCPU	8 GB	100 GB - 300 IOPS	Minimo 2 nodi

Tabella 4.2. Requisiti hardware minimi per OpenShift Container Platform

- Data Plane: entrambi i prodotti offrono la possibilità di utilizzare i gateway tramite immagine Docker, dunque un qualsiasi ambiente in grado di gestire dei container (Docker containerized environment, Kubernetes, OpenShift) è considerato valido; non sono dichiarati requisiti minimi per le immagini di nessuno dei due prodotti.

Le caratteristiche delle macchine che costituiranno il cluster su AWS seguiranno dunque i requisiti hardware necessari per l'installazione di OpenShift Container Platform e il Control Plane di 3scale.

Per quanto riguarda il cluster K3S on-prem, dovendo ospitare sia i Data Plane delle due soluzioni sia il workload di backend, sono stati considerati i requisiti minimi per un server K3S (1 CPU / 1 GB RAM) come punto di partenza; è stato tenuto poi in considerazione anche il workload per il servizio interno, aggiungendoci della contingency. Si noti come i requisiti scalano in base alla dimensione dei workload stessi.

È importante sottolineare che, per i cluster su AWS e Azure, sono state fatte delle previsioni molto "larghe" sulle risorse hardware necessarie. Questo perchè, trattandosi di risorse cloud, esiste sempre la possibilità di effettuare uno scaling delle stesse in base alle esigenze in corso d'opera, che naturalmente possono cambiare. L'**elasticità**, infatti, è uno dei principali vantaggi nell' utilizzare un ambiente cloud per le proprie necessità. Ciò permette, inoltre, di ottimizzare i costi: le risorse utilizzate in questa tesi infatti seguono, se possibile, uno schema di fatturazione *pay-as-you-go*, consentendo al cliente di pagare solo ed esclusivamente la quantità di risorse utilizzate, solo quando necessario. Il nodepool di AKS, ad esempio, pur partendo con 1 solo nodo, può essere configurato per effettuare uno *scale-out* fino a un numero massimo di nodi, in modo automatico (in base alla quantità di carico richiesto nel tempo) o manuale.
Durante l'esperienza, infatti, è stato necessario effettuare uno *scale out* dei nodi workers di OpenShift su AWS, da 2 a 3.

4.2.3 Costi

Considerati i requisiti hardware presentati nel capitolo precedente, i servizi necessari per la gestione dei cluster (Kubernetes, OpenShift e K3S) e le soluzioni di API Management scelte per l'analisi e il confronto (Azure API Management e RedHat 3scale API Management), verranno riassunte di seguito le stime dei costi dell'intera infrastruttura. Si è previsto di utilizzare le risorse per circa 4 ore al giorno per 4 giorni a settimana; i costi riportati sono mensili, dunque la stima è su 64 ore al mese.
Il requisito di bassa latenza è stato decisivo nella scelta della Region per il deployment delle risorse cloud (es. France, Frankfurt).

Microsoft Azure Estimate

Azure AKS

Service type	Custom name	Region	Description	Estimated monthly cost
Azure Kubernetes Service (AKS)	Workers	France Central	3 D4s_v3 (4 vCPUs, 16 GB RAM, 32 GB temporary storage) (Pay as you go), Linux; 1 (free) managed control plane	3 x $0,224/h = **$0,672/h**
Managed disks			E10 (standard SSD), 500 IOPS - 128 GB ($10,56/month)	3 x $10,56 = **$31,68/month**
			Total (64 h = 4 h/day * 4 days/week)	**$74,69**

Figura 4.2. Stima Azure AKS

Amazon Estimate

AWS VMs

Service type	Custom name	Region	Description	Estimated monthly cost
AWS EC2	Masters	UE (Paris)	3 m5.2xlarge (8 vCPUs, 32 GB RAM) (On-demand), Linux	3 x $0,448/h = **$1,344/h**
AWS EC2	Workers	UE (Paris)	2 c5.2xlarge (8 vCPUs, 16 GB RAM) (On-demand), Linux	2 x $0,404/h = **$0,808/h**
AWS EBS	Storage		(1 per EC2 instance) general purpose SSD (gp3), 3000 IOPS / 125 MBps, 128 GB	128 GB x $0,0928/GB/month x 5 (= **$59,39/month)** + (3000 IOPS free + 125MBps free) = **$59,39/month**
			Total (64 h = 4 h/day * 4 days/week)	**$197,12**

Figura 4.3. Stima AWS EC2 (macchine virtuali per OpenShift Container Platform)

Si noti che

- Lo storage e la licenza per Azure API Management non sono risorse a consumo e vengono dunque fatturati mensilmente (730 ore)
- Per OpenShift Container Platform self-managed è stata utilizzata la licenza Trial (60 giorni). Per quanto riguarda l'acquisto di una licenza, è necessario contattare direttamente RedHat (la sottoscrizione include il tipo di supporto offerto durante l'utilizzo del prodotto stesso).
- Per RedHat 3scale API Management è stata utilizzata la licenza Free Evaluation (90 giorni). Per quanto riguarda l'acquisto di una licenza (annuale), il prodotto fa parte del set di strumenti RedHat Integration e dalla documentazione non sembra essere acquistabile separatamente. Per ogni ulteriore informazione occorre contattare direttamente RedHat.

		64 hours	730 hours
Ambiente	Cloud1 (AWS VMs with OpenShift) + Cloud2 (Azure AKS)	$271,81	-
1)	Azure API Management - $0,07/h	-	$ 48,04
2)	Red Hat 3scale API Management	Free Evaluation (90 days)	
3)	Red Hat OpenShift Container Platform	Free Trial (60 days)	
4)	Rancher K3S	Free (open source)	
	Total (1 month)	**$ 319,85**	

Figura 4.4. Stima totale costi

		64 hours	730 hours	Subscription
Ambiente	Cloud1 (AWS VMs with OpenShift) + Cloud2 (Azure AKS)	$271,81	-	
1)	Azure API Management - $0,07/h	-	$ 48,04	-
2)	Red Hat 3scale API Management (RedHat Integration)	$5940/year = $495/month		
3)	Red Hat OpenShift Container Platform	Free Trial (60 days)		-
4)	Rancher K3S	Free (open source)		-
	Total (1 month)	**$319,85 + $495 = $814,85**		

Figura 4.5. Stima totale costi (con licenza 3scale)

Tutte le risorse cloud create sono state taggate opportunamente al momento della creazione, in modo da semplificarne il monitoraggio dei costi.

Vale la pena notare che, tra le risorse acquistate, ne esistono alcune che non prevedono un piano di fatturazione a consumo: questo significa che esiste una componente *fissa* dei costi stimati, tipicamente le licenze e lo storage, e una *variabile*, che carraterizza tutte quelle risorse che supportano anche un piano *pay-as-you-go*. È importante per un'azienda tenere sotto controllo la seconda componente in particolare, in quanto rappresenta l'unico margine di flessibilità per la variazione dei costi in corso d'opera: non solo possono essere ridotti eliminando o riducendo la quantità o la dimensione delle risorse non utilizzate o scarsamente utilizzate, grazie anche alle tecniche FinOps (Capitolo 1.3.1), ma possono facilmente lievitare se non si configurano adeguatamente degli alert e dei limiti alla creazione automatica di nuove risorse o allo scaling verticale e orizzontale automatico di quelle esistenti.

4.3 Implementazione

Verranno ora forniti maggiori dettagli sull'implementazione della soluzione proposta, approfondendo l'installazione dell'infrastruttura, descrivendo i 3 cluster che la compongono.

4.3.1 Installazione dell'infrastruttura

AWS

Per l'installazione di OpenShift self-managed è stata seguita la procedura di installazione di un cluster su un'infrastruttura *installer-provisioned* su AWS con personalizzazioni, la quale prevede la generazione di un template .yaml da arricchire e modificare in base alle esigenze. Tenendo conto dei requisiti hardware minimi per l'installazione di OpenShift e 3scale, elencati nel Capitolo 4.2.2, le caratteristiche dei componenti utilizzati per il cluster su AWS, e quindi anche come parametri per il suddetto template di configurazione, sono le seguenti

Macchina	**vCPU**	**RAM**	**AWS Flavor**	**Note**
Bootstrap	4 vCPU	16 GB	EC2 - i3.large	Provvisorie
Control Plane	4 vCPU	16 GB	EC2 - m5.2xlarge	3 nodi
Compute (workers)	2 vCPU	8 GB	EC2 - c5.2xlarge	2 nodi

Tabella 4.3. Caratteristiche hardware macchine virtuali per cluster OpenShift su AWS

Piattaforma	**AWS Flavor**	**Note**
EC2	gp3 - 128 GB	Una per ogni nodo del cluster
3scale	AWS S3	Persistenza Developer portal e assets di sistema - necessario in quanto EBS non supporta l'access mode RWX (ReadWriteMany)

Tabella 4.4. Caratteristiche storage per cluster OpenShift su AWS

L'installazione vera e propria è effettuata tramite l'*installation program* di **OpenShift Container Platform**, che prende in input il file di configurazione .yaml con tutti i dettagli dell'infrastruttura ed esegue automaticamente il provisioning delle risorse indicate: vengono dapprima instanziate due macchine provvisorie chiamate *bootstrap*, responsabili della creazione dei nodi master; questi ultimi saranno responsabili a loro volta della creazione dei nodi worker. Al termine del processo, che ha richiesto circa 40 minuti al completamento, le macchine *bootstrap* vengono eliminate.

L'installazione di **3scale** è stata effettuata tramite l'Operator dedicato, direttamente dalla console di OpenShift, il quale consente poi di creare un'istanza della custom resource **APIManager**. Questa CRD è reponsabile della creazione dei diversi componenti che vanno a costituire il Control Plane di 3scale, come i database e le due istanze di Apicast *embedded* (*staging* e *production*). Di seguito il template .yaml utilizzato

```
34  kind: Secret
35  metadata:
36    creationTimestamp: null
37    name: aws-auth
38  stringData:
39    AWS_ACCESS_KEY_ID: <user-access-key-id>
40    AWS_SECRET_ACCESS_KEY: <user-secret-access-key>
41    AWS_BUCKET: <bucket-name>
42    AWS_REGION: eu-central-1
43  type: Opaque
```

Figura 4.6. Configurazione Secretper AWS S3

```
1   apiVersion: apps.3scale.net/vlalphal
2   kind: APIManager
3   metadata:
4     name: apimanager-3scale-thesis
5   spec:
6     system:
7       appSpec:
8         replicas: 1
9       sidekiqSpec:
10        replicas: 1
11      fileStorage:
12        simpleStorageService:
13          configurationSecretRef:
14            name: aws-auth
15    zync:
16      appSpec:
17        replicas: 1
18      queSpec:
19        replicas: 1
20    backend:
21      cronSpec:
22        replicas: 1
23      listenerSpec:
24        replicas: 1
25      workerSpec:
26        replicas: 1
27    apicast:
28      productionSpec:
29        replicas: 1
30      stagingSpec:
31        replicas: 1
32    wildcardDomain: <wildcard-domain>
33
```

Figura 4.7. Configurazione APIManager

Notare come, a causa dell'incompatibiltà tra EBS e l'access mode richiesto per il Control Plane di 3scale (RWX - ReadWriteMany), è stato necessario configurare uno storage differente per quest'ultimo, ovvero un AWS S3 Bucket, un servizio cloud di *object storage*, che supporta tale modalità. Dopo aver creato il bucket direttamente dalla console AWS, è stato sufficiente creare un Secret con le seguenti informazioni

- `AWS_ACCESS_KEY_ID` e `AWS_SECRET_ACCESS_KEY`: le chiavi di accesso dell'utente IAM AWS, necessarie per l'autenticazione verso AWS in modo programmatico
- `AWS_BUCKET`: nome del bucket S3 da utilizzare
- `AWS_REGION`: Region del bucket S3

e configurarlo nel template stesso dell'APIManager, nell'attributo `spec.system.fileStorage.simpleStorageService.configurationSecretRef` [Figura 4.6].

On-prem

Il cluster on-premise, che abilita la strategia di cloud ibrido, è stato ospitato su un datacenter privato. È stato utilizzato un host VMware vSphere su cui si è effettuato il provisioning della VM, una macchina AMD64 con le seguenti caratteristiche

Macchina	**vCPU**	**RAM**	**Storage**
Single-node K3S	6 vCPU	16 GB	40 GB

Tabella 4.5. Caratteristiche hardware macchina virtuale per K3S

Il datacenter è protetto all'interno della rete aziendale e il traffico da e verso internet è gestito da un proxy + firewall.

L'installazione di K3S single-node è stata effettuata, all'interno della VM, tramite il semplice comando

```
curl -sfL https://get.k3s.io | sh -
```

che abilita il nodo alla doppia funzione di *server* + *agent* K3S. Sono sati mantenuti tutti i parametri di default, incluso quello per il CNI (Flannel).

Azure

Non avendo requisiti minimi da rispettare per il cluster AKS, si è scelto di partire da 1 nodepool costituito da un solo nodo, che costituisce la parte *worker* del cluster Kubernetes (in AKS, infatti, è possibile scegliere solo la quantità e le caratteristiche dei worker nodes, in quanto il Control Plane è completamente gestito dal servizio Azure); le caratteristiche hardware seguono quelle dei worker nodes dell'installazione di OpenShift su AWS

Macchina	**vCPU**	**RAM**	**Storage**	**Azure Flavor**	**Note**
AKS nodepool (1 node)	4 vCPU	16 GB	32 GB (temporary)	`D4s_v3`	3 nodi

Tabella 4.6. Caratteristiche hardware nodo per nodepool AKS

AKS, essendo un servizio Azure, è gestibile direttamente dall'Azure Portal: il deployment avviene inserendo i parametri di configurazione richiesti, quali Region,

flavor delle macchine, numero di nodepool e numero di nodi, modalità e configurazioni di autoscaling.
È importante notare come il servizio Kubernetes di Azure sia completamente gestito ed è interagibile via command-line: occorre infatti installare kubectl, il tool per linea di comando per la gestione di cluster Kubernetes, sulla propria macchina, e configurarlo con la coppia di chiavi asimmetriche per autorizzare l'accesso, seguendo la procedura guidata su Azure.
AKS, inoltre, essendo un servizio *pay-as-you-go*, può essere arrestato all'occorrenza, in modo da non incidere sui costi: tale operazione spegne i nodi workers (gli unici sui quali si ha il controllo), mantenendo le configurazioni del cluster (etcd) inalterate.

Componenti aggiuntive

- Dominio pubblico: dal momento che il cluster OpenShift non permette raggiungibilità via IP, ma necessita del FQDN con cui collegarsi in fase di creazione del cluster stesso, è stato necessario registrare un dominio pubblico.
- Ngrok reverse proxy: si è rivelato necessario l'utilizzo di un reverse proxy basato su Nginx per consentire la raggiungibilità dall'esterno dei servizi interni alla rete aziendale; ciò è stato possibile grazie a una funzionalità di Ngrok, che consente di creare un tunnel sicuro. Verranno forniti maggiori approdondimenti in seguito.

4.3.2 Installazione dei Data Plane

Una volta installati e configurati i cluster con gli ambienti di orchestrazione, sono stati installati i Data Plane di entrambi i prodotti, secondo la topologia di Figura 4.1. Trattandosi di immagini Docker, è stato necessario scrivere dei file .yaml che descrivessero le risorse necessarie per il loro deployment, prestando particolare attenzione ai loro parametri di configurazione.

Azure API Management offre un'utilissima funzione per semplificare il deployment del gateway in base all'ambiente scelto, in questo caso Kubernetes/OpenShift. Occorre in primo luogo creare, per ogni self-hosted gateway necessario, un placeholder nel Control Plane: quest'ultimo consente di avere sotto controllo lo stato del gateway (in base ai segnali periodici di *heartbeat*), le chiavi di autorizzazione dell'istanza che si intende installare e di generare automaticamente un file di configurazione .yaml con tutti i parametri di default già configurati, naturalmente liberamente personalizzabile in base alle esigenze; l'esempio seguente utilizza il suddetto template come base ed è stato arrichito con le configurazioni per StatsD e Prometheus (maggiore approfondimento nei capitoli successivi).

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: onprem-gateway-token
  namespace: azure-shg-thesis
type: Opaque
stringData:
  value: <Gateway-Key>
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: onprem-gateway-env
  namespace: azure-shg-thesis
data:
  config.service.endpoint:
      "azure-apim-tesi-accogli.configuration.azure-api.net"
  config.service.auth: <Gateway-Key>
  telemetry.metrics.local: "statsd"
  telemetry.metrics.local.statsd.endpoint: "10.43.78.134:8125"
  telemetry.metrics.local.statsd.sampling: "1"
  telemetry.metrics.local.statsd.tag-format: "dogStatsD"
  telemetry.metrics.cloud: "true"
```

Figura 4.8. SHG - esempio di Secret e ConfigMap per il Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: onprem-gateway
  namespace: azure-shg-thesis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: onprem-gateway
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 25%
  template:
    metadata:
      labels:
        app: onprem-gateway
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: onprem-gateway
        image: mcr.microsoft.com/azure-api-management/gateway:v2
        ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8081
        readinessProbe:
          httpGet:
            path: /status-0123456789abcdef
            port: http
            scheme: HTTP
          initialDelaySeconds: 0
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
        env:
        - name: config.service.auth
          valueFrom:
            secretKeyRef:
              name: onprem-gateway-token
              key: value
        envFrom:
        - configMapRef:
            name: onprem-gateway-env
```

Figura 4.9. SHG - esempio di Deployment

3scale, al contrario, non offre una possibilità simile. Il supporto all'installazione su Kubernetes è molto povero e la procedura per l'installazione su Kubernetes

tramite Operator ha problemi di API deprecate, ad oggi non ancora risolti. Il Deployment, dunque, è stato scritto da zero dopo numerosi tentativi dovuti a diversi problemi nati in corso d'opera, in particolare con l'image registry di RedHat, che richiede il login manuale con le credenziali del service account, e con una configurazione automatica errata dell'endpoint verso il servizio di backend del Control Plane, sulla quale è stato necessario intervenire eseguendo un override tramite variabile d'ambiente [Figura 4.10-riga 38].

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: apicast-deploy
5    namespace: apicast-thesis
6  spec:
7    selector:
8      matchLabels:
9        app: apicast
10   replicas: 1
11   template:
12     metadata:
13       labels:
14         app: apicast
15     spec:
16       containers:
17       - name: apicast-gateway
18         image: registry.redhat.io/3scale-amp2/apicast-gateway-rhel8:3scale2.11
19         env:
20         - name: THREESCALE_PORTAL_ENDPOINT
21           value: <endpoint-with-access-token>
22         - name: APICAST_RESPONSE_CODES
23           value: "true"
24         - name: APICAST_HTTPS_PORT
25           value: "8443"
26         - name: APICAST_EXTENDED_METRICS
27           value: "true"
28         - name: APICAST_CONFIGURATION_LOADER
29           value: "lazy"
30         - name: APICAST_CONFIGURATION_CACHE
31           value: "0"
32         - name: THREESCALE_DEPLOYMENT_ENV
33           value: "staging"
34         - name: APICAST_MANAGEMENT_API
35           value: "status"
36         - name: APICAST_LOG_LEVEL
37           value: "info"
38         - name: BACKEND_ENDPOINT_OVERRIDE
39           value: "https://backend-3scale.apps.aws-ocp-accogli.aws-tesi-accogli.it"
40         ports:
41         - name: http
42           containerPort: 8080
43         - name: https
44           containerPort: 8443
```

Figura 4.10. Apicast - esempio di Deployment

4.3.3 Configurazione backend per i gateway

La configurazione dei servizi di backend per i gateway avviene, per entrambe le soluzioni, direttamente dal portale admin. Le istanze dei gateway interessati, poi, effettueranno una richiesta verso il Control Plane per recuperare la versione aggiornata della configurazione.

I servizi Azure LogicApp e AWS Lambda sono stati esposti tramite degli endpoint HTTP, automaticamente generati alla creazione: grazie al trigger configurato, una richiesta in entrata scatena l'esecuzione del workload che, una volta terminato, ritornerà la risposta al chiamante. È sufficiente, dunque, sia per Azure API Management che per 3scale, selezionare il gateway corretto e configurare nella sezione **Backend** l'endpoint verso il quale effettuare il routing delle richieste.

Nel cluster on-prem, il workload interno è esposto tramite una risorsa Service di tipo ClusterIP, in modo tale che sia raggiungibile solo all'interno del cluster: sarà il gateway, che avrà un endpoint contattabile esternamente, a effettuare il routing delle richieste verso di esso; occorre quindi configurare nella sezione Backend il ClusterIP del servizio interno. I due esempi seguenti mostrano la configurazione backend di due istanze differenti del self-hosted gateway

```
<set-backend-service base-url="https://ypexmtppwttd4ohfob7s6jigpa0ypexm.lambda-url.eu-central-1.on.aws/ " />
```

Figura 4.11. SHG - routing verso AWS Lambda

```
<set-backend-service base-url="http://10.43.53.41" />
```

Figura 4.12. SHG - routing verso servizio interno on-prem

Come si può notare, il SHG situato su AWS effettuerà il routing verso un endpoint pubblico, in quanto il servizio Lambda è già raggiungibile pubblicamente; viceversa, il SHG posizionato on-premise effettua il routing delle richieste verso un endpoint privato, corrispondente all'indirizzo ClusterIP del servizio interno, non raggiungibile dall'esterno.

4.3.4 Esposizione degli endpoint per i gateway

Il gateway, che in questi prodotti di API Management costituisce il Data Plane, ovvero quel componente fondamentale dell'architettura che si occupa di inoltrare le richieste verso il backend corrispondente dopo aver applicato le opportune policies, deve essere raggiungibile esternamente dai client; il Control Plane, infatti, consente di aggiornare le configurazioni di ogni istanza di Data Plane disponibile e quindi di modificarne il comportamento, ma, come abbiamo visto nel Capitolo 2.2, in ambienti multi-cloud e hybrid-cloud è necessario che le richieste non passino dal Control Plane, ma vengano effettuate direttamente verso gli endpoint opportuni in base a dove si trova il servizio richiesto (on-prem, nello stesso cloud provider in cui si trova il Control Plane o in uno differente).

Per questo motivo, dopo aver installato tutti i gateway e dopo aver configurato i rispettivi backend, occorre esporre un endpoint per ognuno, in modo che i client possano contattarli dall'esterno.

- AKS: dopo aver installato i relativi Deployment, le istanze di Apicast e del self-hosted gateway sono stati esposti grazie alle relative risorse Service di tipo LoadBalancer, avendo cura di mappare correttamente le porte.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: apicast-deploy-lb
  namespace: apicast-thesis
spec:
  externalTrafficPolicy: Local
  selector:
    app: apicast
  ports:
  - name: http
    port: 8080
    protocol: TCP
    targetPort: 8080
    nodePort: 30627
  - name: https
    port: 8443
    protocol: TCP
    targetPort: 8443
    nodePort: 31730
  type: LoadBalancer
```

Figura 4.13. Service per il Deployment di Apicast su AKS

```yaml
apiVersion: v1
kind: Service
metadata:
  name: aks-gateway
  namespace: azure-shg-thesis
spec:
  externalTrafficPolicy: Local
  selector:
    app: aks-gateway
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
    nodePort: 30102
  - name: https
    port: 443
    protocol: TCP
    targetPort: 8081
    nodePort: 31100
  type: LoadBalancer
```

Figura 4.14. Service per il Deployment del self-hosted gateway di Azure su AKS

Gli endpoint esposti possono essere visualizzati tramite il comando `kubectl get svc -A`, che indica indirizzo IP e porta da contattare per il servizio corrispondente [Figura 4.15]

```
bruno@DESKTOP-67ULHOP:~$ kubectl get svc -A
NAMESPACE                NAME                TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
apicast-thesis           apicast-deploy-lb   LoadBalancer   10.0.24.42     20.4.106.53     8080:30627/TCP,8443:31730/TCP
azure-shg-thesis         aks-gateway         LoadBalancer   10.0.16.40     20.23.112.96    80:30102/TCP,443:31100/TCP
```

Figura 4.15.

- OpenShift su AWS: con l'installazione di 3scale tramite Operator, vengono automaticamente generate due istanze di Apicast (una *staging* e una *production*), i relativi servizi e le loro Route. Il self-hosted gateway di Azure API Management, i cui Deployment e Service sono stati installati manualmente, è stato esposto tramite la creazione di una Route, configurata dalla console di OpenShift: la procedura guidata consente infatti di creare la risorsa Route scegliendo il Service da raggiungere e altri parametri quali Target Port, Hostname e Path.
  Gli endpoint generati sono visualizzabili direttamente dalla console, nella sezione Routes [Figura 4.16]

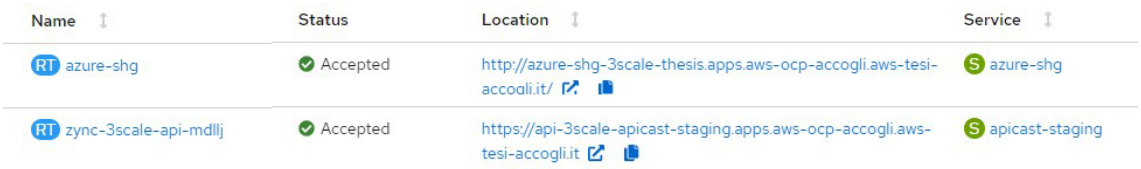

Figura 4.16. Routes per i gateway su OpenShift

- On-premise: è stata creata una risorsa Service di tipo NodePort per entrambe le istanze dei gateway, in modo da mappare le porte della macchina locale sui due servizi corrispondenti.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: apicast-svc
  namespace: apicast-thesis
spec:
  selector:
    app: apicast
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
      nodePort: 30100
    - name: https
      port: 8443
      protocol: TCP
      targetPort: 8443
      nodePort: 31102
    - name: web
      port: 9421
  type: NodePort
```

Figura 4.17. Service per il Deployment di Apicast onprem

```yaml
apiVersion: v1
kind: Service
metadata:
  name: onprem-gateway
  namespace: azure-shg-thesis
spec:
  selector:
    app: onprem-gateway
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
    nodePort: 30102
  - name: https
    port: 443
    protocol: TCP
    targetPort: 8081
    nodePort: 31100
  type: NodePort
```

Figura 4.18. Service per il Deployment del self-hosted gateway di Azure onprem

A causa di componenti interni come proxy, NAT e firewall, per l'esposizione degli endpoint dei gateway deployati on-prem è stato necessario utilizzare un servizio esterno, **ngrok**, un *reverse proxy* globalmente distribuito, che abilita un tunnel sicuro per la comunicazione tra l'edge Ngrok e la rete privata, creando una connessione cifrata. Dopo aver creato un account sul sito ufficiale e recuperato il token di autenticazione, occorre installare l'agent nella macchina in cui è presente il serivizio da esporre e configurare il token di autenticazione tramite il comando

```
ngrok config add-authtoken <auth-token>
```

infine, il comando

```
ngrok http <port> --log=stdout > ngrok.log &
```

espone la porta `<port>` della macchina locale sulla quale è mappato il Service verso il quale abilitare il tunnel sicuro, ed esegue un *redirect* dell'output verso

un file di log locale; il processo viene eseguito in background.
In alternativa, è possibile indicare tutti i parametri di configurazione in un file .yaml [Figura 4.19], ed ottenere lo stesso risultato con un unico comando

```
ngrok start --all --log=stdout > ngrok.log &
```

```
1  version: "2"
2  authtoken: <auth-token>
3  tunnels:
4    first:
5      addr: 30100
6      proto: http
7    second:
8      addr: 30102
9      proto: http
```

Figura 4.19. Configurazione ngrok

Per ogni tunnel sicuro creato, ngrok genera l'endpoint da contattare (Figura 4.20: in questo caso è visibile direttamente nel file di log); le richieste verranno poi inoltrate al servizio locale.

```
name=shg addr=http://localhost:30102 url=https://ad6e-91-218-224-5.eu.ngrok.io
name=apicast addr=http://localhost:30100 url=https://af7f-91-218-224-5.eu.ngrok.io
```

Figura 4.20. Endpoint generati da Ngrok

4.4 Principali strumenti utilizzati

In questo capitolo verranno presentati brevemente i principali strumenti utilizzati durante l'esperienza.

4.4.1 Docker

Docker è una tecnologia che sfrutta la virtualizzazione a livello di sistema operativo per distribuire software in unità standardizzate chiamate *container*. Un container, dunque, contiene il software e tutto il necessario per la sua corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime.
Docker focalizza la sua attenzione sulle applicazioni, semplificando la loro distribuzione ed esecuzione creando dei contenitori leggeri, portabili e autonomi che possono essere eseguiti in qualsiasi ambiente (a patto che venga utilizzata la stessa architettura di CPU), risolvendo dunque il problema della portabilità del software.
L'architettura di Docker è costituita da

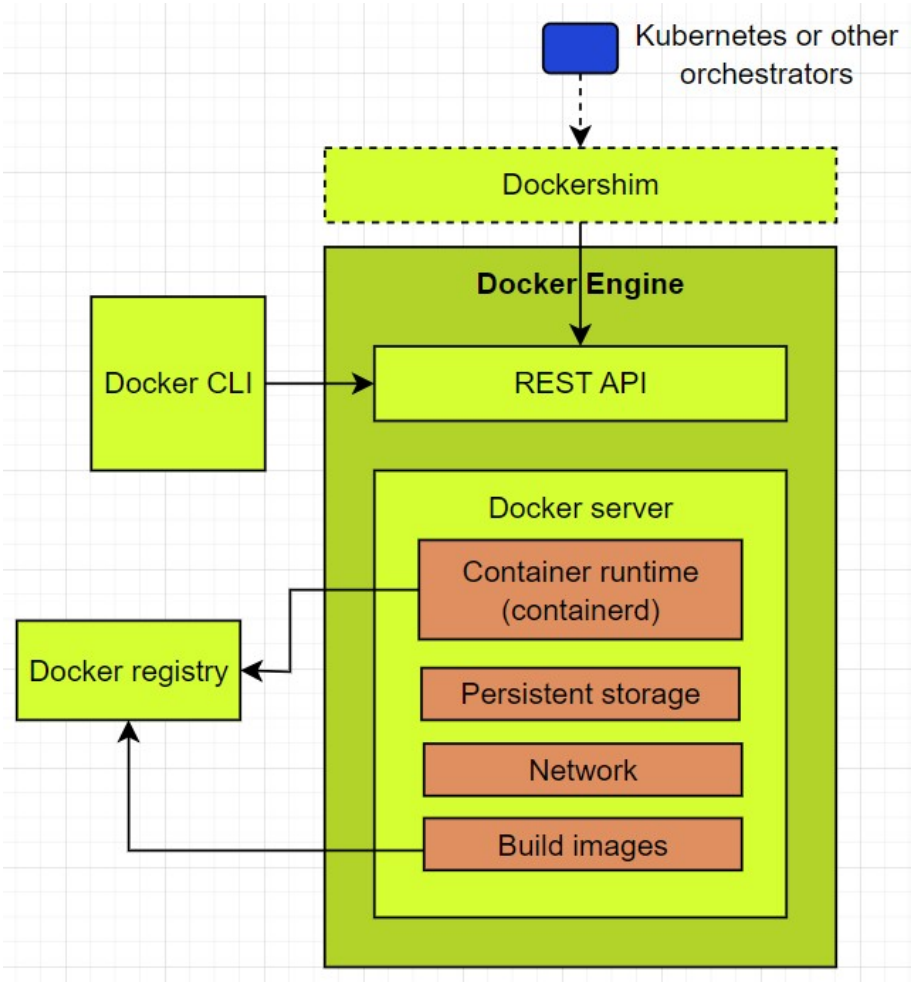

Figura 4.21. Arrchitettura di Docker. Fonte: slides del corso Cloud Computing del Politecnico di Torino

- Docker client: è il client utilizzato dall'utente per dialogare con il docker deamon; consente di specificare i comandi per la gestione delle immagini e dei container (build, pull, run, etc.). Un esempio di client è la Docker CLI.

- Docker Engine: è il sistema centrale di Docker; segue un'architettura client-server ed è installato nella macchina host. è composto da

  - Server: costituisce il docker deamon, chiamato "dockerd". è reponsabile della creazione e gestione di immagini, container, rete, volumi persistenti, etc.
  - REST APIs: utilizzate per indicare a dockerd le azioni da eseguire

- Docker registry: è il luogo in cui vengono immagazzinate le immagini; può essere pubblico o privato: DockerHub è il registry pubblico di default utilizzato da Docker, ma è possibile crearne e gestirne uno privato personale.

La creazione di un'immagine avviene partendo dalla compilazione di un file chiamato Dockerfile, un file di testo che raccoglie tutte le informazioni delle componenti che la costituiscono. L'immagine viene creata tramite il comando "docker build", al quale viene passato come argomento il percorso del Dockerfile. L'immagine viene poi lanciata grazie al comando "docker run"; un container non è altro che un'istanza di un'immagine in esecuzione. Le immagini possono poi essere salvate in un Docker registry, per gestire meglio versioning e tagging delle stesse e semplificare e velocizzare l'adozione dell'approccio DevOps durante il processo di gestione del lifecycle del software.

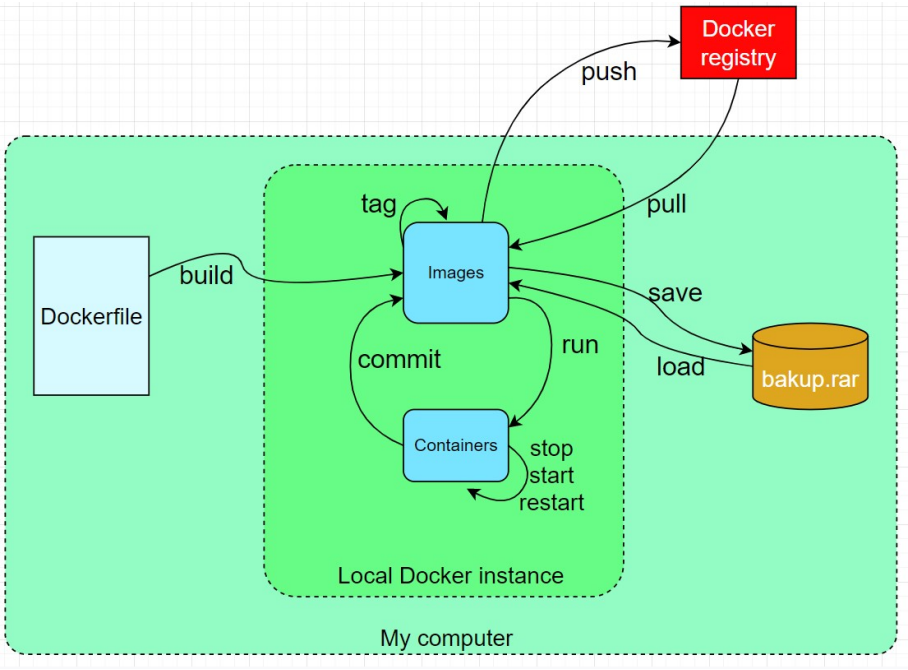

Figura 4.22. Docker lifecycle. Fonte: slides del corso Cloud Computing del Politecnico di Torino

4.4.2 Kubernetes

Kubernetes è un sistema open-source di orchestrazione e gestione di container per automatizzare la distrubuzione e la gestione del software. Inizialmente progettato e sviluppato da Google, adesso è mantenuto da Cloud Native Computing Foundation.
Funziona con diversi sistemi di containerizzazione (container runtimes). Un *container runtime* è un software in grado di eseguire i container che costituiscono un Pod in Kubernetes: in ogni nodo, l'agent *kubelet* utilizza la CRI (Container Runtime Interface) come astrazione, in modo che si possa utilizzare qualsiasi container runtime compatibile. Originariamente Kubernetes era compabile solo con il Docker runtime, attraverso un adapter chiamato Dockershim, il quale è stato rimosso nella release 1.24 di Maggio 2022. Contemporaneamente, però, il progetto ha sviluppato la CRI (Container Runtime Interface), estendendo la compabilità ad altri container runtimes.

Kubernetes espone le proprie capabilities tramite delle API REST, consumabili da qualsiasi client HTTP, come "curl". Il client di default di Kubernetes è chiamato "kubectl", che rende molto più semplice la creazione, gestione e scoperta delle risorse nel cluster.
Kubernetes segue un'architettura master/slave. I componenti software di cui è costituito possono essere classificati in due categorie: quelli che si occupano di controllare l'esecuzione dei container applicativi sono raggruppate nel Control Plane, mentre il Data Plane raggruppa i componenti software coinvolti nelle funzionalità che gestiscono il carico di lavoro del cluster.
Il controllo del sistema avviene specificando un desired state (stato desiderato): ogni partecipante si attiva per contribuire a mutare il sistema verso il desired state definito nel master.

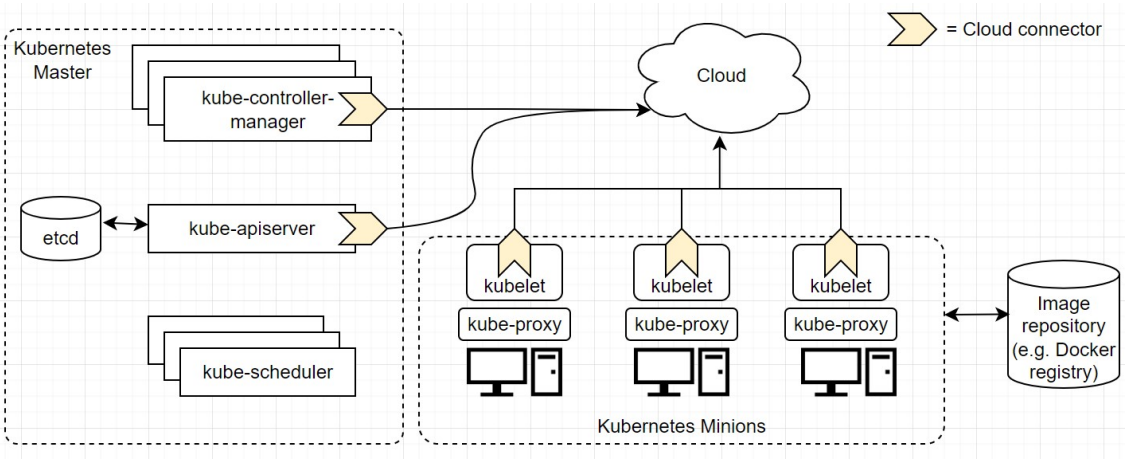

Figura 4.23. Architettura di Kubernetes. Fonte: slides del corso Cloud Computing del Politecnico di Torino

Control Plane

Il master è l'unità di controllo principale del cluster e gestisce le comunicazioni nel sistema. Il Control Plane consiste di vari componenti, ognuno dei quali può essere eseguito su un singolo nodo master o più nodi master, in modalità alta affidabilità

- etcd: un database chiave-valore che si occupa di mantenere lo stato del sistema. I componenti del Control Plane sono stateless e fanno riferimento, tramite kube-apiserver, allo stato mantenuto in etcd. Per questo motivo anche il demone etcd viene ridondato nelle installazioni ad alta disponibilità, anche se ad essa favorisce la consistenza, secondo il teorema CAP

- kube-apiserver: implementa e rende disponibili le API di Kubernetes, esponendo una interfaccia REST verso lo stato del cluster, validando le richieste e aggiornando lo stato degli oggetti in etcd. Rappresenta il canale di coordinamento e controllo.
  kube-apiserver richiede l'autenticazione per l'accesso alle API: in kubectl questa è effettuata tramite un identity file, chiamato "kubeconfig", che contiene tutti i parametri per l'autenticazione.

- kube-controller-manager: gestisce un insieme di controller, ovvero dei loop di riconciliazione che controllano costamtemente che lo stato attuale del sistema coincida con quello desiderato. Gli interventi necessari al raggiungimento del desired state vengono fatti da controller per specifiche funzionalità attivati a loro volta dal controller manager.

- kube-scheduler: componente estensibile che che decide, in base a un algoritmo di allocazione e alla disponibilità di risorse, come assegnare il carico di lavoro specificato dal desired state sui nodi che compongono il cluster. Per effettuare ciò, kube-scheduler tiene traccia delle risorse su ogni nodo nel tempo, in modo da assicurarsi che un determinato workload non venga schedulato in eccesso di risorse disponibili.

Worker node

Un nodo worker di occupa di eseguire il carico di lavoro secondo le modalità definite dal Control Plane. Per poter eseguire i carichi di lavoro, il nodo deve disporre di un container runtime.

- kubelet: è responsabile di controllo e gestione del carico di lavoro sul singolo nodo. Si assicura che tutti i container nel nodo siano healthy: mantiene una comunicazione col master e interviene costantemente sul nodo al fine di raggiungere e mantenere il desired state.

- kube-proxy: è l'implementazione di un proxy di rete e un bilanciatore di carico, tramite i quali implementa l'astrazione per i Service; è repsonsabile del routing del traffico verso l'apposito container, basandosi sull'indirizzo IP e la porta della richiesta.

- container runtime: astrae aspetti quali le risorse computazionali disponibili, lo storage e il networking. i container vengono inseriti nel container runtime, controllato da kubelet.

In Kubernetes lo stato del sistema è descritto mediante l'utilizzo del concetto di risorsa. Le risorse descrivono il desired state, cioè lo stato desiderato del sistema. Una volta creata o modificata la descrizione di una risorsa sul master, le varie componenti di Kubernetes apportano le necessarie modifiche per variare dallo stato attuale del sistema verso lo stato desiderato.
'utilizzo di un modello a risorse permette di descrivere con un linguaggio dichiarativo lo stato che il sistema deve assumere, senza la necessità di conoscere la tecnologia sottostante, aspetto particolarmente importante nei contesti cloud in cui Kubernetes viene offerto come servizio avendo la flessibilità di scelta sulla tecnologia sottostante.
Alcuni esempi di risorse

- Pod: è l'unità di elaborazione elementare e la più semplice risorsa distribuibile in Kubernetes. Ogni Pod consiste di uno o più container per i quali viene garantita la locazione sul medesimo nodo; i container in un Pod sono interraggiungibili. Ad ogni Pod in Kubernetes viene assegnato un indirizzo IP univoco all'interno del cluster.
  Il Pod astrae rete e storage al fine di poter essere spostato e replicato facilmente sui nodi del cluster, permettendo una forte scalabilità orizzontale, in particolare alle applicazioni orientate ai microservizi.

- Service: è un'astrazione che definisce un metodo per esporre un'applicazione in esecuzione su un insieme di Pod e una politica con la quale farvi accesso. Questo livello di astrazione disaccoppia l'esposizione dell'applicazione dalla sua esecuzione: i Pod possono essere aggiornati, distrutti o rischedulati, cambiando l'indirizzo IP e il modo in cui possono essere raggiunti dai client; i Service forniscono un punto di accesso stabile indipendentemente dalle modifiche che possono subire i Pod retrostanti. L'esposizione dell'applicazione tramite Service può avvenire internamente al cluster (ClusterIP) o esternamente (es. NodePort).

- Secrets e ConfigMaps: sono risorse utilizzate per memorizzare informazioni di configurazione confidenziali e non, in coppie chiave-valore. Tali oggetti possono essere poi montati nei container come file o passati come variabili d'ambiente.

Kubernetes è progettato per essere estensibile: è possibile infatti dichiarare nuovi tipi di risorse, definendole tramite una CRD (Custom Resource Definition). Gli Operatori sono applicazioni specifiche progettate per monitorare e reagire alla manipolazione di una CRD, seguendo un pattern molto simile a quello di Kubernetes (il control loop). Nell'architettura, gli operatori sono i client delle API di Kubernetes, che agiscono da controllers per una CR (Custom Resource).

4.4.3 K3S

K3S è una distribuzione leggera di Kubernetes rivolta all'IoT e all'edge computing, sviluppata da Rancher Labs. Ha requisiti di risorse molto bassi ed è completamente certificata dalla CNCF (Cloud Native Computing Foundation). Viene distribuita tramite un unico file di installazione dal peso di soli 100 MB, che include tutte le dipendenze, tra le quali

- sqlite3 al posto di etcd, come meccanismo di storage di default
- containerd
- Flannel
- CoreDNS
- CNI
- Ingress controller (Traefik)
- Embedded service load balancer
- Embedded network policy controller

In K3S il Control Plane è chiamato "server" e un nodo worker prende il nome di "agent". K3S è production-ready e può essere installato in modalità di alta affidabilità, utilizzando due o più nodi server e un database esterno (in sostituzione al database sqlite integrato). Considerato l'ambito per cui è stato sviluppato, non è raro trovare installazioni di K3S su singolo nodo, ovvero una singola macchina che funge sia da server sia da agent. Tale installazione è anche quella di default.

4.4.4 OpenShift

OpenShift è una piattaforma Kubernetes enterprise sviluppata da RedHat, che offre un'esperienza cloud-like in qualunque ambiente venga installata. Il prodotto di punta è **OpenShift Container Platform**, una PaaS costruita attorno a container Linux orchestrati e gestiti da Kubernetes sulla base di RedHat Enterprise Linux, a partire da una lunga esperienza di utilizzo e gestione di container in ambienti di produzione ad alte prestazioni.
Si può scegliere di distribuire Red Hat OpenShift come servizio gestito sul provider cloud preferito per un'esperienza ottimale su Azure, AWS, IBM Cloud o Google Cloud oppure, per un maggiore controllo, scegliere un deployment autogestito di Red Hat OpenShift su qualsiasi cloud privato o pubblico, su bare metal o all'edge. Sono disponibili i seguenti metodi di distribuzione:

- Self-managed
- Cloud native: ROSA, ARO e RHOIC risptettivamente su AWS, Azure, e IBM Cloud

- OpenShift Online, l'offerta SaaS
- OpenShift Dedicated come servizio gestito

OpenShift è esso stesso un sistema distribuito fatto da diversi componenti, ciascuno dei quali è installabile in alta affidabilità; sfrutta diversi altri progetti e tecnologie open source (come *Source 2 Image*, la tecnologia che permette di passare dal codice sorgente di un'applicazione all'applicazione containerizzata, *Jenkins*, un popolarissimo server di CI/CD, e *Prometheus*, sistema di monitoring open-source molto utilizzato). Risorse come ImageStreams (sequenza di riferimenti ad immagini associabili a dei Deployments) e Template (meccanismo di packaging per i componenti delle applicazioni) sono unici in OpenShift e semplificano la distribuzione e la gestione delle applicazioni.
È inoltre pensato per assecondare la collaborazione tra utenti minimizzando le interferenze indesiderate: la piattaforma è multitenancy e dotata di un sistema articolato di gestione degli utenti, dei ruoli e dei permessi.
L'installazione out-of-the-box comprende inoltre un image repository e un OperatorHub, una GUI dove gli utenti possono esplorare e installare una libreria di Operatori Kubernetes impacchettati per una gestione del lifecycle più semplice.

OpenShift è completamente compatibile con le API Kubernetes e qualsiasi applicazione che può essere eseguita su Kubernetes può essere eseguita anche su OpenShift.
In Kubernetes, il principale programma per la gestione dei cluster è `kubectl`. Il command-line tool di OpenShift invece si chiama `oc`. Quest'ultimo è molto simile a `kubectl`, ma dispone di funzionalità extra che semplificano le attività, in particolare l'accesso ai cluster. Un'altra differenza è che OpenShift offre una console di gestione web integrata, mentre Kubernetes offre un'interfaccia simile ma solo come componente aggiuntivo opzionale.
La console OpenShift ha delle viste dedicate per sviluppatore e amministratore:

- Developer: la vista developer è orientata al lavoro con risorse applicative all'interno di un namespace
- Administrator: la vista administrator-oriented consente il monitoraggio delle risorse e della salute dei container, la gestione degli utenti, gestione degli Operatori, etc.

Nel caso di OpenShift anche i nodi, cioè gli ambienti per il runtime, devono essere forniti con Red Hat Enterprise Linux CoreOS, mentre i nodi Kubernetes usano qualsiasi sistema operativo basato su Linux. Openshift infatti controlla scrupolosamente i sistemi operativi utilizzati: i componenti "master" devono avere Red Hat CoreOS (questo livello di controllo consente al cluster di supportare aggiornamenti e patch dei nodi master al minimo sforzo), mentre i nodi "worker" possono usare altre varianti di Linux o anche Windows.

4.4.5 Prometheus

Prometheus è un sistema open-source di monitoring e alerting, originariamente sviluppato da SoundCloud. Nel tempo è diventato molto popolare ed è stato adottato

da diverse compagnie e organizzazioni, costruendo una community di utenti e sviluppatori molto attiva. Oggi è un progetto open-source standalone ed è entrato a far parte della CNCF nel 2016.
Prometheus colleziona e memorizza le metriche come dati di serie temporali: ogni record viene memorizzato con le informazioni temporali alle quali è stato registrato, insieme ad altri dati di tipo chiave-valore opzionali chiamate "labels". Consente quindi, per una metrica, di registrarne il cambiamento nel tempo. Tramite un linguaggio di querying flessibile chiamato PromQL, è possibile filtrare e aggregare le metriche basandosi sulle labels.

I componenti che costituiscono Prometheus sono i seguenti

- Prometheus server: il componente principale responsabile di raccogliere e immagazzinare i dati delle metriche
- Librerie client per integrare con Prometheus i servizi da monitorare
- Un push gateway per permettere ai workload effimeri e ai batch jobs di esportare le loro metriche a Prometheus: dal momento che tali servizi potrebbero smettere di esistere prima che Prometheus raccolga le metriche, sono loro stessi a inviare le metriche al pushgateway, il quale a sua volta espone le metriche a Prometheus.
- Special-purpose exporters per i servizi (HAProxy, StatsD, Graphite, etc): nel caso in cui non è fattibile integrare un dato sistema direttamente con Prometheus
- Un Alertmanager per gestire gli alterts
- Tools di supporto

I target vengono configurati in modo statico o tramite service discovery. In Kubernetes, quando Prometheus è installato come Operatore, le pseudo-CRD ServiceMonitor e PodMonitor servono a specificare in modo dichiarativo gli endpoint dai quali Prometheus raccoglierà le metriche; una volta creati, il Prometheus Operator aggiorna le configurazioni di Prometheus in modo da includere quelle specificate dal ServiceMonitor e PodMonitor: Prometheus comincerà così a raccogliere le metriche anche dai nuovi endpoint configurati.

Il funzionamento di Prometheus si basa su una fase di "scraping" dei jobs che sono stati integrati, direttamente o grazie a un pushgateway che fa da intermediario per i servizi di breve durata. Successivamente memorizza localmente i dati raccolti e applica delle regole per aggregare e registrare nuovi record o per generare degli alerts.
Prometheus è progettato per essere affidabile, in modo da rappresentare il sistema di riferimento per il monitoring durante un outage per esaminarne velocemente le cause. Tutto ciò va a scapito dell'accuratezza: non è infatti consigliabile utilizzarlo in contesti in cui è necessario effettuare una fatturazione per ogni richiesta, in quanto le informazioni raccolte potrebbero non essere abbastanza dettagliate e complete.

4.5 Analisi di performance dei Data Plane

4.5.1 Metriche considerate e risultati ottenuti

È stata effettuata una valutazione sulle prestazioni dei gateway di entrambe le soluzioni, al fine di scegliere la soluzione che offre il gateway più veloce e reattivo nella risposta; a tal fine, sono state prese in considerazione 3 metriche, 2 esposte da entrambi i gateway e una ricavata da dei test ad hoc

- Tempo di risposta del backend: dopo aver inoltrato la richiesta, quanto tempo passa fino al momento in cui il gateway riceve una risposta; denominata `request_backend_duration_seconds` per il self-hosted gateway di Azure e `upstream_response_time_seconds` per Apicast di RedHat 3scale
- Tempo impiegato dal gateway per rispondere al client: dopo aver ricevuto la richiesta dal client, quanto tempo passa fino al momento in cui il gateway ritorna una risposta al client (è la somma del precedente più il tempo di processamento interno da parte del gateway): denominata `request_duration_seconds` per il self-hosted gateway di Azure e `total_response_time_seconds` per Apicast di RedHat 3scale
- Latenza totale: dopo aver inviato la richiesta, quanto tempo passa fino al momento in cui il client riceve una risposta dal gateway; denominata `total_client_time`

Per garantire uniformità e congruenza nei test, sono state scelte due macchine: una è stata utilizzata come effettivo client per le richieste e l'altra per il monitoraggio delle metriche in tempo reale. La scelta di utilizzare due macchine differenti è stata maturata in previsione dell'esecuzione del test verso i gateway on-prem: dovendo utilizzare una VPN per il monitoraggio delle metriche on-prem, dedicare una macchina separata per questo compito si è rivelato necessario per garantire le stesse condizioni per tutti i test. Se si fosse utilizzato un solo client, connettere quest'ultimo alla VPN aziendale per il monitoraggio in tempo reale delle metriche avrebbe forzato anche le richieste per i test a passare per la rete aziendale, condizione differente rispetto a quella per i test effettuati per i gateway nei cloud provider.

I test sono stati effettuati grazie a un programma Javascript scritto appositamente, in modo da generare regolarmente del traffico verso ogni singolo gateway. Il programma fa uso di *axios* come client HTTP per le richieste; la variabile `delayInMilliseconds` consente di modificare il delay tra una richiesta e l'altra (e dunque la frequenza delle richieste), mentre la variabile `gateway_endpoint` contiene l'endpoint del gateway da contattare.

```
'use strict';
const axios = require('axios').default;
var delayInMilliseconds = 2000
const sleep = n => new Promise(resolve => setTimeout(resolve, n))

async function makeRequests () {
    while (1) {
      axios.get(gateway_endpoint)
        .then(function (response){
          //console.log(response)
        }).catch(function (error) {
            // handle error
            console.log(error);
        })
        await sleep(delayInMilliseconds)
    }
  }

makeRequests()
```

Figura 4.24. Codice del programma utilizzato per i test

Seguono i risultati ottenuti dai test effettuati

Azure API Management

self-hosted gateway

metrica	AKS	AWS OCP	ONPREM
request_backend_duration_seconds	14,3 ms	92,1 ms	1,1 ms
request_duration_seconds	16,5 ms	93,6 ms	2,4 ms
total_client_time	60,8 ms	142,3 ms	68,4 ms

RedHat 3scale API Management

APIcast

metrica	AKS	AWS OCP	ONPREM
upstream_response_time_seconds	98,2 ms	99,2 ms	15,4 ms
total_response_time_seconds	150,5 ms	137,3 ms	84,0 ms
total_client_time	270,7 ms	168,4 ms	144,0 ms

4.5.2 Costi in termini di risorse

Vengono riportati di seguito i consumi, in termini di CPU e memoria, dei gateway durante i test effettuati. I dati raccolti sono identici per tutti e tre gli ambienti presi in esame (on-prem, Azure AKS e OpenShift su AWS)

Prodotto	CPU	Memoria
Apicast	1m	34 Mi
Azure SHG	6m	85 Mi

Tabella 4.7. Situazione di *idle*

Prodotto	CPU	Memoria
Apicast	167m	61 Mi
Azure SHG	65m	88 Mi

Tabella 4.8. 20 req/s

Legenda unità di misura

In Kubernetes (e dunque anche in OpenShift), la risorsa CPU è misurata in *CPU units*. Una CPU è equivalente a

- 1 vCPU su AWS
- 1 Core su GCP
- 1 vCore su Azure
- 1 Hyperthread su un processore bare-metal Intel con Hyperthreading

Il suffisso *m* significa *milli*, e serve a indicare valori frazionari: per questo motivo, ad esempio, un Container che richiede 0.5 CPU ha intenzione di utilizzarne la metà di un Container che richiede 1 CPU (100m CPU = 100 milliCPU = 0.1 CPU). La CPU viene sempre richiesta come quantità assoluta, mai relativa: 0.1 è la stessa quantità di CPU su una macchina single-core, dual-core o 48-core.

L'unità di misura utilizzata per indicare la quantità di memoria utilizzata, in questo caso, è il *Mebibyte*, che corrisponde a 2^{10} bytes. Non è da confondere con il *Megabyte*, che equivale a 10^3 bytes, spesso utilizzato come sinonimo, creando ambiguità. Nota: Mi = MiB = Mebibyte.

4.6 Analisi dei security drivers

I due prodotti sono stati valutati anche in merito alle funzionalità offerte in ambito *sicurezza*. In particolare sono stati presi in considerazione i seguenti driver di sicurezza e i relativi criteri di valutazione, effettuando un'estrazione di specifiche dalla documentazione delle soluzioni stesse

- Autenticazione e autorizzazione: è importante trovare un framework che supporta i vari modelli di autenticazione/autorizzazione usati dai vari cloud provider, ma in modo da permettere la creazione e la definizione di account,

ruoli e policies in modo centralizzato; è fondamentale avere autenticazione e autorizzazione separati da qualsiasi cloud provider, gestiti dunque da un componente esterno agli stessi

- Reporting/monitoring: operando nel cloud si può fare affidamento agli strumenti di monitoring offerti dal cloud provider; tuttavia, in multi-cloud, è necessario avere un tool che supporta più CSP e attiva la visibilità su tutto l'ambiente. Una visione d'insieme del sistema su tutto l'environment multi-cloud è essenziale per identificare, investigare e reagire ai cyber attacchi

- Controllo connessioni/traffico: avere la possibilità di limitare il tipo e la quantità di connessioni gestibili nell'unità di tempo è fondamentale per identificare eventuali attacchi informatici e distinguerli dal traffico "utile", ovvero quello mirato all'utilizzo del servizio; la maggiore superficie d'attacco derivata dall'utilizzo di più cloud provider rende questo aspetto ancora più importante e necessario per una corretta gestione del traffico stesso.

- Policy enforcement: poter modificare il comportamento delle soluzioni tramite il policy enforcement è sicuramente un importante valore aggiunto per utilizzare al meglio le soluzioni stesse, ma è indispensabile in ambienti multi-cloud quando tale personalizzazione è strettamente legata al tipo di servizio offerto da quel cloud provider.

Verranno di seguito analizzate individualmente le due soluzioni.

4.6.1 Azure API Management

- **Autenticazione**: il processo di autenticazione di un utente viene offerto dal Developer Portal. Un utente, chiamato in questo contesto "sviluppatore", può registrarsi nel portale ed usare le proprie credenziali per accedere. Nuovi utenti (sviluppatori) possono essere aggiunti anche dall'amministratore. Azure API Management consiglia caldamente di utilizzare strumenti di autenticazione più sicuri della coppia username-password, come OpenID Connect, certificati e AzureAD.
  È possibile inoltre delegare registrazione e autenticazione dell'utente a un endpoint esterno, da utilizzare in alternativa alla procedura built-in del Developer Portal: questa funzionalità è molto utile in quelle situazioni in cui si preferisce gestire i dati degli utenti tramite un proprio sito e dunque eseguire delle azioni custom durante il processo di validazione.
  Per attivare la delegazione di registrazione/autenticazione dell'utente, occore crare un endpoint speciale nel proprio sito, il quale verrà utilizzato come entry-point per qualsiasi richiesta di registrazione/autenticazione proveniente dal Developer Portal. Il workflow finale sarà

  1. L'utente clicca sul link *sing-in* o *sign-up* del Developer Portal
  2. Il browser reindirizza all'endpoint esterno di delegazione
  3. L'endpoint di delegazione a sua volta reindirizza l'utente al processo di registrazione/autenticazione

4. Al termine della registrazione/autenticazione, l'utente viene reindirizzato al Developer Portal nel punto in cui era prima del primo redirect

- **Autorizzazione**: in questo ambito, l'autorizzazione è il modo in cui le API vengono utilizzate dagli utenti (sviluppatori), una volta autenticati. In Azure API Management esiste il concetto di Product: ad esso possono essere associate delle API e degli utenti, e specifica il modo in cui gli utenti associati possono utilizzare le API associate. In alternativa, si possono raggruppare più utenti in un Gruppo, utilizzato per gestire la visibilità dei Products a più Sviluppatori. Le api vengono associate a un Product dall'amministratore, il quale decide la caratteristica del Product, che può essere

  - *Open*: le API associate al Product possono essere consumate liberamente; gli sviluppatori possono consumare le API senza una subscription, ma possono comunque essere utilizzati diversi meccanismi per mettere in sicurezza l'accesso dei client all'API, come **OAuth2.0** o **certificati**). Quando arriva una richiesta ad un'API senza una subscription key, Azure API Management controlla se tale API è associata a un "open" Product e, in caso affermativo, gestisce tale richiesta nel contesto di tale Product (applicando eventuali policies e regole di accesso), altrimenti controlla se quella singola API richiede una subscription key e, in caso affermativo, gestisce la richiesta nel contesto di quella API e Operation; in caso contrario l'accesso è vietato (HTTP response status code 401)
  - *Protected*: gli sviluppatori devono prima registrarsi al Product per ottenere l'accesso alle API associate, ottenendo una **subscription key** che può accedere qualsiasi API in quel Product. Una novità consente agli utenti (sviluppatori) di ottenere una subscription key per singole API o tutte le API in un'istanza di Azure API Management, evitando dunque, da parte dell'amministrarore, la necessità di creazione di un Product per poi associarci l'API.

  Un amministratore può, di fatto, configurare una API come "non autenticata", ovvero affinchè non venga richiesta alcuna subscription key, e quindi liberamente utilizzabile da tutti (si applicano però le sue policy, come il *rate limiting*).

  Come per il processo di registrazione/autenticazione, è possibile effettuare la delegazione ad un endpoint esterno anche per il processo di sottoscrizione ad un Product. Il workflow finale è simile

  1. Lo sviluppatore seleziona un Product nel Developer Portal e clicca su *Subscribe*
  2. Il browser reindirizza all'endpoint esterno di delegazione
  3. L'endpoint di delegazione esegue le azioni richieste per la sottoscrizione ad un Product, progettate dall'utente. Ad esempio, tali step possono includere
     - Redirect ad una pagina per la richiesta di informazioni di pagamento
     - Richiesta di ulteriori informazioni

  - Memorizzare l'informazione senza richiedere all'utente ulteriori azioni

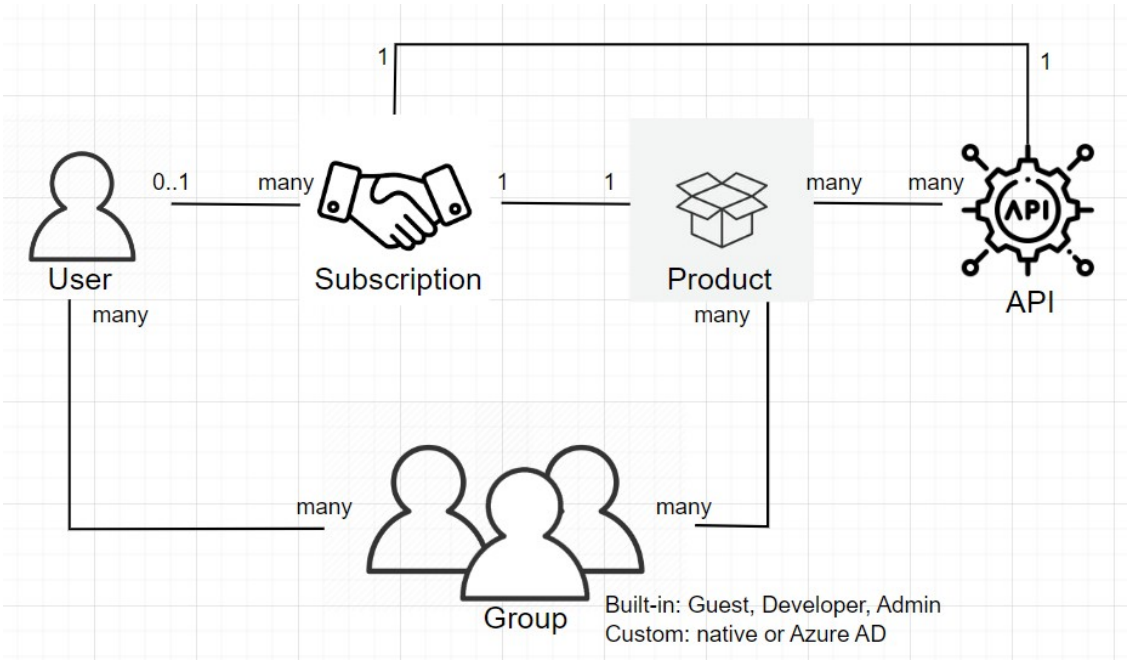

Figura 4.25. Relazione tra Sviluppatori, Subscriptions e Prodotti. Fonte: Azure [14]-p.82

- **Reporting**: il gateway espone periodicamente segnali di "heartbeat" e metriche verso il Control Plane; queste ultime possono essere esporate localmente a Prometheus, appoggiandosi a StatsD, un protocollo per la collezione e l'aggregazione delle metriche stesse.
  Azure API Management, come anche per altre funzionalità, è integrato con altri servizi Azure, in questo caso specifico con

  - Azure Monitor: utilizzato per logging, reporting e alerting di operazioni di gestione, eventi di sistema e API requests
  - Azure Application Insights: utilizzato per visualizzare le metriche live e per troubleshooting; è possibile emettere metriche custom
  - Azure EventHub: utilizzato per il logging di eventi specifici e personalizzati tramite la policy `<log-to-eventhub>`

  Azure mette inoltre a disposizione una serie di REST API che consentono di ottenere degli usage reports in base allo scope (by API, by geography, by Operations, by Product, by Request, by subscription, by Time, by User). In ogni caso, **Microsoft Defender for Cloud** consente di monitorare la security baseline di Azure API Management in qualsiasi momento, ed eventuali raccomandazioni per mettere in sicurezza le soluzioni basate su Azure API Management.

- **Controllo connessioni/traffico**: Il self-hosted gateway di Azure necessita di una connessione outbound TCP/IP sulla porta 443 verso il Control Plane per

  - Inviare report periodici
  - Controllare regolarmente (ogni 10 secondi) aggiornamenti di configurazione dal Control Plane ed eventualmente applicarli quando disponibili
  - Inviare le metriche ad Azure Monitor ed eventi ad Application Insights, se configurato

  Supporta, tramite policies, funzionalità di rate limiting per ogni istanza di gateway e per ogni API: se la quantità di richieste per unità di tempo supera tale valore, le richieste in eccesso non verranno servite; è possibile personalizzare il codice di errore della risposta e il relativo messaggio.

- **Policy enforcement**: Azure API Management consente di specificare policies in diversi scopes
  - Globale: tutte le API
  - Product: solo le API associate al Product selezionato
  - API: tutte le Operations in una API
  - Operation: singola Operation di una API

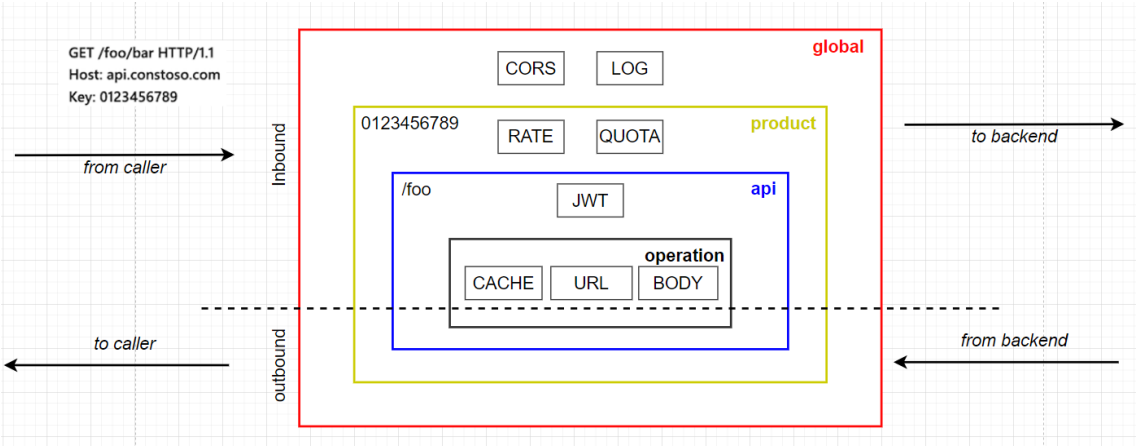

Figura 4.26. Policy scopes. Fonte: Azure [14]-p.25

  e, nel dettaglio di uno scope API, si distingue tra policies di
  - Inbound: ricevuta la richiesta dal client, prima di inoltrarla al backend
  - Backend: prima di inoltrare la richiesta al backend
  - Outbound: ricevuta la risposta dal backend, prima di inoltrarla al client
  - On-error: in caso di errore durante il processing di una richiesta

  modificabili tramite un editor XML. Dall'editor si possono scegliere degli *snippets* preconfigurati di policies di autenticazione, caching, access restriction, trasformazione, CORS, validazione.

```
<policies>
    <inbound>
        <!-- statements to be applied to the request go here -->
    </inbound>
    <backend>
        <!-- statements to be applied before the request is
             forwarded to the backend service go here -->
    </backend>
    <outbound>
        <!-- statements to be applied to the response go here -->
    </outbound>
    <on-error>
        <!-- statements to be applied if there is an error
             condition go here -->
    </on-error>
</policies>
```

Figura 4.27. Editor di API policies integrato in Azure APIM

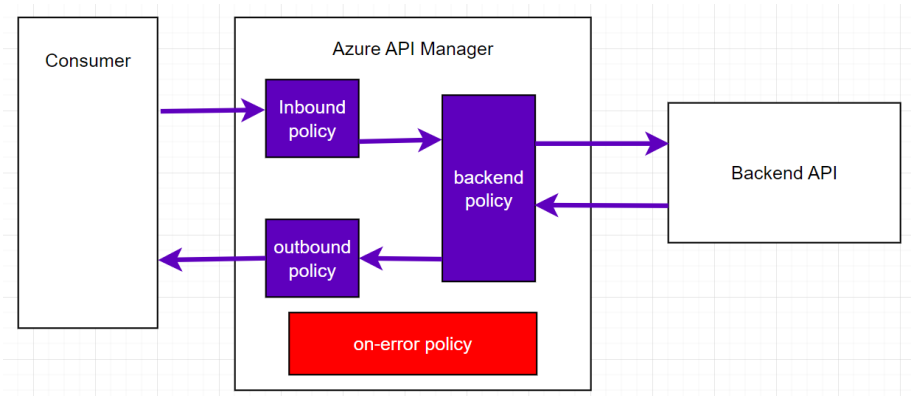

Figura 4.28. Call chain delle policies nei policy scopes. Fonte: Azure [17]

La Figura 4.29 riporta un esempio completo di policy in una API utilizzata nello use case.
La policy policy `<validate-content>` consente di effettuare una validazione del contenuto ricevuto nel body della richiesta (in questo caso di tipo JSON), tramite un *JSON schema* che definisce il formato e il tipo dei campi sui quali effettuare il controllo; lo schema è stato definito nell'apposita sezione del pannello di controllo di Azure API Management, tramite editor dedicato. In caso di errore di validazione, nella sezione `<on-error>` è presente, tra le altre, la policy `<log-to-eventhub>`, che consente di inviare un report di log direttamente ad Event Hub, specificando l'id del *logger*, una risorsa di API Management dedicata all'invio di logs ai servizi supportati (fino ad ora Application Insights, Event Hub e Azure Monitor), e il contenuto del messaggio da inviare.

```
<policies>
    <inbound>
        <base />
        <set-header name="BrokerProperties" exists-action="override">
            <value>@{
                    var json = new JObject();
                    json.Add("MessageId", context.RequestId);
                    return json.ToString(Newtonsoft.Json.Formatting.None);
            }</value>
        </set-header>
        <set-backend-service base-url="http://10.43.53.41" />
        <validate-content unspecified-content-type-action="prevent" max-size="102400"
                          size-exceeded-action="prevent" errors-variable-name="errorMessage">
            <content type="application/json" validate-as="json" action="prevent" schema-id="onprem-json-schema" />
        </validate-content>
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
    </outbound>
    <on-error>
        <base />
        <set-variable name="errorMessage" value="@{
            return new JObject(
                new JProperty("EventTime", DateTime.UtcNow.ToString()),
                new JProperty("ErrorMessage", context.LastError.Message),
                new JProperty("ErrorReason", context.LastError.Reason),
                new JProperty("ErrorSource", context.LastError.Source),
                new JProperty("ErrorScope", context.LastError.Scope),
                new JProperty("ErrorSection", context.LastError.Section),
                new JProperty("ErrorPath", context.LastError.Path),
                new JProperty("ErrorPolicyId", context.LastError.PolicyId)

            ).ToString();
        }" />
        <choose>
            <when condition="@((bool)((string)context.LastError.Source == "validate-content"))">
                <log-to-eventhub logger-id="logger-id">
                    @{
                        return new JObject(
                            new JProperty("EventTime", DateTime.UtcNow.ToString()),
                            new JProperty("ServiceName", context.Deployment.ServiceName),
                            new JProperty("RequestId", context.RequestId),
                            new JProperty("RequestIp", context.Request.IpAddress),
                            new JProperty("OperationName", context.Operation.Name)
                        ).ToString();
                    }
                </log-to-eventhub>
                <return-response>
                    <set-status code="412" reason="Precondition Failed" />
                    <set-header name="Content-Type" exists-action="override">
                        <value>application/json</value>
                    </set-header>
                    <set-body>@((string)context.Variables["errorMessage"])</set-body>
                </return-response>
            </when>
        </choose>
        <return-response>
            <set-status code="500" reason="Error" />
            <set-header name="Content-Type" exists-action="override">
                <value>application/json</value>
            </set-header>
            <set-body>@((string)context.Variables["errorMessage"])</set-body>
        </return-response>
    </on-error>
</policies>
```

Figura 4.29. Esempio di API policy

4.6.2 RedHat 3scale API Management

- **Autenticazione**: in 3scale API Management, il Developer Portal è il punto di riferimento per il processo di autenticazione dell'utente (lo "sviluppatore"). Quest'ultimo può registrarsi nel portale ed usare le proprie credenziali per accedere o può essere aggiunto dall'amministratore. RedHat 3scale, di default, configura l'accesso al Developer Portal tramite username-password; altri metodi di autenticazione dell'utente supportati sono tramite **GitHub**, **Auth0** e **RedHat Single Sign-On**.

- **Autorizzazione**: l'autorizzazione è il modo in cui le API vengono utilizzate dagli utenti (sviluppatori), una volta autenticati. In RedHat 3scale Api Management, l'API è chiamata Product, e il modo in cui gli utenti possono utilizzarla è chiamato Application Plan; una Application è di fatto un binding tra Account, Product e Application plan. Per ogni Application si può scegliere in che modo effettuare l'autenticazione. Il tipo di credenziali richieste cambia in base alla scelta

  - Standard API key: il metodo più semplice; consiste in una singola stringa di caratteri che può essere rigenerata in qualsiasi momento
  - `App_ID/App_key` pair: ogni application ha un identificativo (`App_ID`) e fino a 5 Application keys (`App_key`); questo consente a un developer di evitare il downtime dopo la generazione di una nova chiave, come avviene nel caso precedente
  - OpenID Connect: è un protocollo di autenticazione basato su OAuth2.0; 3scale può essere integrato con un identity provider conforme alle specifiche di OpenID Connect, come RedHat SSO.

  Le credenziali possono essere inserite come headers della richiesta (HTTP headers), direttamente nei parametri della stessa (query parameters) o utilizzando HTTP Basic Authentication.
  In 3scale è possibile simulare una API "non autenticata", tramite una policy: si generano delle credenziali di default e le si rendono pubbliche, affinchè tutti le possano utilizzare per autenticare le richieste associate; Apicast, infatti, per autenticare la richiesta, ne effettua prima sempre una verso il Control Plane per l'autenticazione, utilizzando le credenziali associate ad essa.

- **Reporting**: Apicast espone periodicamente delle metriche e dei segnali di "heartbeat" verso il Control Plane. Supporta inoltre l'esposizione delle metriche a Prometheus localmente e nativamente: ciò consente dunque di specificare regole di alerting in modo più granulare, in base al comportamento registrato dalla singola istanza del gateway.
  3scale API Management mette a disposizione delle REST API per consultare gli usage reports in base allo scope (Service, Product, Account, Billing)

- **Controllo connessioni/traffico**: Apicast, di default, ha una connessione outbound TCP/IP per autenticare in modo sicuro ogni API, effettuando una richiesta verso il Control Plane con le credenziali fornite, e richiedere l'aggiornamento delle configurazioni. Di default, gli aggiornamenti di configurazione

vengono richiesti per ogni richiesta in entrata; è possibile, tuttavia, grazie a due variabili d'ambiente, modificare questo comportamento

  - `APICAST_CONFIGURATION_LOADER`: permette di specificare se gli aggiornamenti vadano richiesti solo all'avvio del gateway (`boot`) o per ogni richiesta in entrata (`lazy`); nel primo caso il gateway andrebbe riavviato ogni volta che viene cambiata la configurazione dello stesso dal Control Plane
  - `APICAST_CONFIGURATION_CACHE`: permette di specificare la durata, in secondi, della cache di configurazione, di fatto andando a specificare ogni quanto tempo richiedere degli aggiornamenti di configurazione al Control Plane

Di conseguenza, una configurazione `lazy`, per funzionare, deve essere accompagnata da un valore `0` per la variabile `APICAST_CONFIGURATION_CACHE`. Apicast supporta funzionalità di rate limiting per singola API, in modo da limitare la quantità di richieste in entrata da servire; dal Control Plane è possibile personalizzare codice e messaggio di errore per le richieste in eccesso, che di conseguenza non verranno servite.

- **Policy enforcement**: 3scale permette di configurare un "policy chain" di polcies di default, senza aver modo di specificare la direzione (inbound/outbound) o lo scope per singola policy; tuttavia, 3scale offre un'elevata flessibilità, in quanto consente di aggiungere delle policy personalizzate, scritte in LUA.
  Si possono specificare policy per diversi ambiti: autenticazione, caching policies, access restriction, transformation policies, CORS policies, validation policies.

  In Figura 4.30 viene riportato un esempio di policy chain

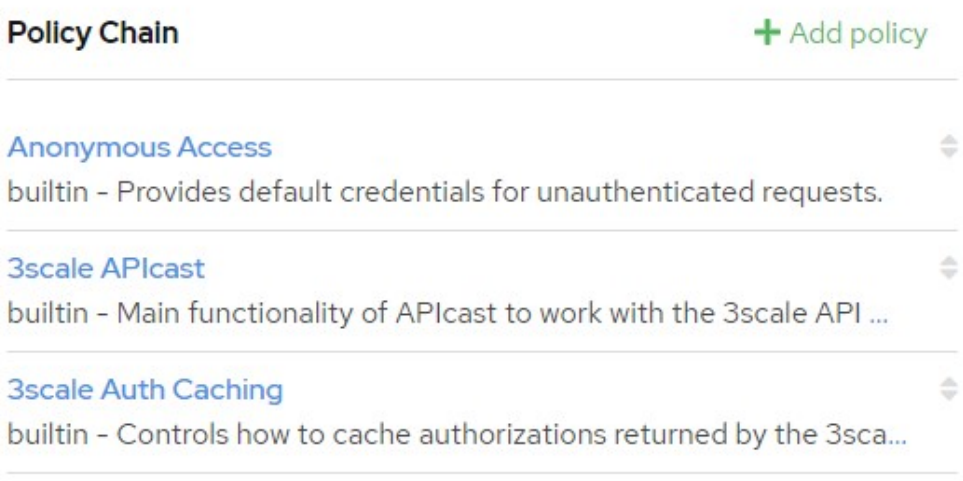

Figura 4.30. Esempio di API policy chain

Le policy vengono eseguite in ordine, dall'alto verso il basso; in particolare, in questo esempio:

- Anonymous Access: consente di fornire le credenziali di default per richieste non autenticate. Può essere utile, per esempio, per applicazioni legacy che non possono essere adattate all'invio di parametri di autenticazione. In questo caso la policy è stata utilizzata per rendere l'API "non autenticata": quando arriva una richiesta priva di parametri di autenticzione, Apicast autorizzerà la chiamata con le credenziali di default configurate nella policy; nel dettaglio, gli unici parametri configurabili per questa policy sono
  - Tipo di credenziali: standard API key o coppia `App_ID/App_key`
  - Valore delle credenziali, in base al tipo selezionato

  Infine, tramite l'Application Plan, si possono configurare i rate limits sulla Application utilizzata per le credenziali di Default.
- 3scale APIcast: policy di default sempre presente; costituisce le funzionalità principali di Apicast nell'interfacciamento con l'API manager di 3scale, ovvero il matching delle mapping rules, autorizzazione, reporting. Non è configurabile.
- 3scale Auth Caching: controlla il modo in cui avviene il caching delle autorizzazioni ritornate dal Control Plane. L'unico parametro configurabile è la modalità di caching
  - *Strict*: esegue il caching delle sole chiamate autorizzate; se una chiamata fallisce o non è autorizzata, la policy invalida la entry della cache. Se il contol plane diventa irraggiungibile, tutte le chiamate in cache vengono respinte, indipendentemente dal loro stato in cache.
  - *Resilient*: fa il caching di chiamate autorizzate e non autorizzate; chiamate fallite non invalidano la entry della cache. Se il Control Plane diventa irraggiungibile, le chiamate che effettuano un *cache hit* continuano ad essere autorizzate o negate in base al loro stato in cache.
  - *Allow*: fa il caching di chiamate autorizzate e non autorizzate; le chiamate in cache continueranno ad essere autorizzate o negate in base al loro stato in cache. Tuttavia, le nuove chiamate vengono memorizzate in cache come autorizzate.
  - *None*: disabilita il caching; questa modalità è utile se si vuole disattivare il cachine mantenendo la policy nel policy chain, senza doverla rimuovere

4.7 Espandibilità

Le soluzioni offrono, infine, diverse possibilità di espansione. In generale, supportano i **webhook**, consentendo di specificare un URL da contattare in seguito a uno o più eventi (selezionabili da una lista di opzioni predeterminate) scatenati dal Developer Portal e/o Admin Portal (ad esempio creazione/modifica/cancellazione di un nuovo utente, subscription, Product/Application, chiave di autorizzazione).

Azure API Management è, inoltre, nativamente integrato con **EventGrid**, un servizio di routing di eventi che utilizza un modello publisher-subscriber: in questo modo si possono integrare altri servizi di Azure come **Functions** e **Logic Apps**, di fatto aumentando notevolmente il grado di estensione raggiungibile.
Azure API Management supporta piena estensibilità del Developer Portal: se le capabilities di default non sono sufficienti, è possibile richiedere una feature su GitHub, contribuire al codice o persino fare un fork del repository, estendere il codice e ospitare autonomamente il portale, in modo molto semplice grazie ai file statici generati automaticamente, pronti per essere ospitati su cloud o on-prem.

Inoltre, in merito agli aspetti di security analizzati

- **Autenticazione**: Il Developer Portal di Azure API Management consente di delegare registrazione/autenticazione dell'utente e sottoscrizione ad un Product a endpoint esterni (nel caso in cui si possiede un proprio sito)

- **Reporting**: Azure API Management, grazie al supporto nativo con Azure-Monitor, EventHub e Application Insights può estendere le proprie funzionalità di reporting e alerting sfruttando questi servizi; di particolare rilevanza è sicuramente la policy `<log-to-eventhub>` che, grazie alla possibilità di essere aggiunta nello schema XML delle policy di un'API tramite l'editor dedicato, consente di inviare un report molto dettagliato ad EventHub. Un esempio di utilizzo di tale policy può essere visto nella Figura 4.29, riga 41: in caso di errore di validazione del contenuto della richiesta, prima di ritornare una risposta con codice di errore `412 - Precondition Failed`, viene inviato un report su Event Hub, specificando alcune informazioni utili sull'errore appena generatosi.
  Infine, entrambe le soluzioni supportano l'esposizione delle metriche a Prometheus, garantendo dunque massima estensibilità sul tipo di comportamento da adottare per tutte le metriche esposte (AlertManager, AlertManagerConfig(s) e PrometheusRule(s)).
  Vengono riportati in Figura 4.31 e Figura 4.32 due esempi di configurazione (file .yaml) utilizzati nello use case, per abilitare l'esposizione locale delle metriche dei gateway a Prometheus.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sputnik-metrics
  namespace: metrics
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sputnik-metrics
  template:
    metadata:
      labels:
        app: sputnik-metrics
    spec:
      containers:
      - name: sputnik-metrics-statsd
        image: prom/statsd-exporter
        ports:
        - name: tcp
          containerPort: 9102
        - name: udp
          containerPort: 8125
          protocol: UDP
        args:
          - --statsd.mapping-config=/tmp/statsd.y
          - --statsd.listen-udp=:8125
          - --web.listen-address=:9102
        volumeMounts:
          - mountPath: /tmp
            name: sputnik-metrics-config-files
      - name: sputnik-metrics-prometheus
        image: prom/prometheus
        ports:
        - name: tcp
          containerPort: 9090
        args:
          - --config.file=/tmp/prometheus.yaml
        volumeMounts:
          - mountPath: /tmp
            name: sputnik-metrics-config-files
      volumes:
        - name: sputnik-metrics-config-files
          configMap:
            name: sputnik-metrics-config
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sputnik-metrics-statsd
  namespace: metrics
spec:
  type: NodePort
  ports:
  - name: udp
    port: 8125
    targetPort: 8125
    protocol: UDP
  selector:
    app: sputnik-metrics
---
apiVersion: v1
kind: Service
metadata:
  name: sputnik-metrics-prometheus
  namespace: metrics
spec:
  type: NodePort
  ports:
  - name: http
    port: 9090
    targetPort: 9090
  selector:
    app: sputnik-metrics
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sputnik-metrics-config
  namespace: metrics
data:
  statsd.yaml: ""
  prometheus.yaml: |
    global:
      scrape_interval:     3s
      evaluation_interval: 3s
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']
      - job_name: 'test_metrics'
        static_configs:
          - targets: ['localhost:9102']
```

Figura 4.31. Prometheus/StatsD per self-hosted gateway

Il self-hosted gateway di Azure API Management può essere abilitato all'esposizione delle metriche grazie al supporto a StatsD, un protocollo unificante per la raccolta e l'aggregazione delle metriche. L'esempio precedente crea un Deployment con due container, uno per StatsD e uno per Prometheus, e un Service di tipo NodePort per ognuno. Per completare la configurazione delle metriche, sarà sufficiente, una volta che il Deployment sarà completato, prendere

nota di `CLUSTER-IP` e `PORT` del Service di StatsD e aggiornare la configurazione del self-hosted gateway aggiungendo la chiave `telemetry.metrics.local` nella ConfigMap dello stesso nel file .yaml corrispondente e applicando le modifiche (un esempio completo della configurazione del self-hosted gateway è mostrato in Figura 4.8). In questo modo, il gateway invierà le metriche al servizio StatsD, contattandolo al `<CLUSTER-IP>:<PORT>` configurati; Prometheus raccoglierà poi le metriche da StatsD. La dashboard di Prometheus è accessibile tramite `<CLUSTER-IP>:<PORT>` del servizio NodePort configurato.

Il secondo esempio riguarda il servizio di Apicast in un ambiente in cui è installato Prometheus tramite Operator. Il container di Apicast serve le metriche sulla porta 9421, per cui il Service corrispondente è stato configurato in modo tale da esporre tale porta [Figura 4.17].
Per completare la configurazione, occorre indicare al Prometheus Operator il modo in cui andare a monitorare il servizio di Apicast, utilizzando la CRD ServiceMonitor, indicando le label opportune per selezionare il servizio e l'endpoint corretti.

```
1  apiVersion: monitoring.coreos.com/v1
2  kind: ServiceMonitor
3  metadata:
4    name: apicast-servicemonitor
5    namespace: apicast-thesis
6  spec:
7    selector:
8      matchLabels:
9        app: apicast-svc
10   endpoints:
11   - port: 'web'
```

Figura 4.32. ServiceMonitor per Apicast

- **Policy enforcement**: sicuramente l'ambito più rilevante per quanto riguarda le estensioni, in quanto influisce in modo determinante sul comportamento dei gateway. 3scale consente di aggiungere delle policy custom scritte in LUA, mentre Azure API Management non offre una possibilità simile.

4.8 Confronto delle soluzioni

Lavorare con soluzioni che fanno uso di più cloud provider contemporaneamente garantisce più flessibilità rispetto all'utilizzo di una sola piattaforma, consentendo alle aziende di gestire meglio i costi e evitare il vendor lock in, e migliora la resilienza. Tuttavia, l'elevata complessità di un'architettura multi-cloud aumenta la superficie d'attacco e dunque l'esposizione ad attacchi informatici, aggiungendo nuove sfide per le aziende; la sicurezza in tale ambito, perciò, richiede un approccio olistico che affronti diverse vulnerabilità e stabilisca controlli coerenti in ambienti multipli ed eterogenei.

Dall'analisi e dai confronti effettuati risulta evidente come entrambi i prodotti siano provvisti di soluzioni molto robuste dal punto di vista della sicurezza, garantendo dunque ottimi risultati sia in ambienti single-cloud che multi-cloud.

Se da un lato, però, Azure API Management offre la completa integrazione con servizi dedicati a operazioni specifiche (monitoring, alerting, configurazione) grazie all'Azure Portal, al contrario di 3scale, il cui admin portal è sprovvisto di alcune funzioni basilari (come il monitoraggio dello stato dei gateway), dall'altro l'offerta di Azure non consente la stessa personalizzazione di RedHat 3scale in termini di policies per i data plane, aspetto di notevole rilevanza soprattutto in ambienti multi-cloud. In tali contesti, infatti, la centralizzazione delle funzioni principali in un'unica sezione di controllo e il supporto ad una larga personalizzabilità sono aspetti estremamente importanti per la semplificazione delle principali attività dell'amministratore, fornendogli allo stesso tempo il pieno controllo su di esse e la loro pesonalizzabilità.

Azure API Management offre un Data Plane che, dai test effettuati in questa tesi, risulta superiore, a livello di **performance**, rispetto ad Apicast, offerto da 3scale. Dai test effettuati emerge che il self-hosted gateway ha una minore latenza, conseguenza di un tempo di processing interno inferiore rispetto a quello del concorrente.
Il Data Plane di Azure API Management risulta superiore, in base ai test effettuati, anche in termini di **consumo di risorse**, in quanto utilizza meno della metà di CPU rispetto ad Apicast, in situazioni di carico sostenuto.
Tuttavia, il gateway di 3scale, nei test effettuati, si comporta molto meglio in stato di idle, arrivando a consumi di CPU decisamente ridotti: ciò lo rende particolarmente adatto a contesti IoT, soprattutto in situazioni in cui la frequenza delle richieste è ridotta e dunque il tempo di idle nettamente superiore rispetto a quello di carico.

La Tabella 4.9 evidenzia una valutazione finale complessiva dei principali aspetti dei prodotti, a seguito dell'analisi svolta. La valutazione tiene conto dei criteri di valutazione indicati nel 4.6

- Insufficiente: la funzionalità/caratteristica è incompleta e/o inefficace, rendendola inutilizzabile in un ambiente multi-cloud
- Sufficiente: la funzionalità/caratteristica è limitata ma presenta le caratteristiche base per l'utilizzo in un ambiente multi-cloud
- Buono: la funzionalità/caratteristica possiede tutte le caratteristiche fondamentali per un utilizzo efficace in un ambiente multi-cloud, ma può essere migliorata sotto alcuni aspetti
- Ottimo: la funzionalità/caratteristica è completa e stabile sotto tutte le caratteristiche fondamentali per un utilizzo efficace in un ambiente multi-cloud e presenta eventuali feature extra

Le soluzioni offerte da Microsoft Azure e Red Hat si dimostrano molto valide in termini di capabilities, grazie anche al loro consistente background tecnologico che permette di offrire, oltre a performance stabili e feature esclusive, anche una consistente integrazione e piena compatibilità con servizi già esistenti, come OpenShift per RedHat e servizi di monitoring e reporting quali EventHub, Application Insights e Monitor per Azure. È risultato evidente, in ultima analisi, come Azure API Management fa la differenza in termini di usabilità generale e semplificazione delle attività dell'amministratore e degli utenti, supportate dall'integrazione di tutta una serie di servizi aggiuntivi che completano e potenziano l'offerta iniziale, senza dubbio un valore aggiunto non trascurabile in un ambiente cloud distribuito.

Nonostante i due prodotti semplifichino già non poche operazioni di provisioning e gestione delle API su diversi Cloud Provider e l'observability delle istanze di Data Plane su più ambienti, la scelta di rilasciare il gateway anche sotto forma di immagine Docker, se da un lato garantisce piena libertà allo sviluppatore sulla scelta dell'ambiente sui cui effettuarne il deployment (es. Docker containerized environment o Kubernetes), l'operazione di gestione, configurazione e aggiornamento di tali immagini risulta essere ancora molto "manuale".
Ad esempio, nel caso in cui uno sviluppatore volesse aggiungere all'architettura in esame un'istanza del gateway su Google Cloud, dopo aver fatto il provisioning dell'ambiente su cui fare il provisioning dell'immagine (es. Kubernetes), deve

- Procurarsi il .yaml per la distrubuzione del gateway (nel caso di Azure API Management) o scriverselo da sè (nel caso di 3scale)
- Configurarne le variabili d'ambiente, modificando il precedente file di configurazione
- Effettuare il deployment del gateway sulla piattaforma
- Per qualsiasi futura modifica all'immagine stessa, accedere all'ambiente sul quale si trova quell'istanza del gateway per apportare modifiche al .yaml

Ciò rende tutte quelle operazioni di installazione, gestione e mantenimento delle istanze dei gateway più onerose del dovuto, costringendo l'amministratore a replicare più volte gli stessi passaggi. Un tentativo di miglioramento di tale esperienza, venendo in supporto all'amministratore soprattutto nella fase di installazione di una nuova istanza del Data Plane, è offerto da Azure: grazie all'integrazione tra Azure API management e Azure Arc su Kubernetes, è possibile distribuire il gateway come estensione in un **cluster Kubernetes abilitato per Azure Arc**. Quest'uttimo consente di collegare e configurare cluster Kubernetes in esecuzione su differenti cloud provider (ad esempio GCP o AWS) o on-prem (ad esempio VMWare vSphere) ad **Azure Arc**, lo strumento che consente di estendere la piattaforma Azure per semplificare la creazione di applicazioni e servizi su più datacenter, nell'edge e in ambienti multi-cloud. In questo modo l'amministratore può effettuare il provisioning di una nuova istanza del gateway con le relative configurazioni direttamente dall'Azure Portal.
Tale soluzione, tuttavia, è disponibile solo in preview ed è quindi sconsigliato utilizzarla in ambiente di produzione. Approfondimenti e analisi esulano dagli obiettivi di questo lavoro di tesi.

Data Plane			
Caratteristica	**Note**	**Azure APIM**	**RedHat 3scale**
Prestazioni	Il self-hosted gateway di Azure garantisce una latenza minore	Ottimo	Buono
Consumo risorse	A parità di traffico e a pieno carico, il self-hosted gateway di Azure consuma molte meno risorse; Apicast adatto per applicazioni IoT	Ottimo	Buono
Security drivers			
Funzionalità	**Note**	**Azure APIM**	**RedHat 3scale**
Autenticazione	Entrambi offrono supporto ad un framework esterno (OpenID Connect), consentendo la delegazione	Ottimo	Ottimo
Autorizzazione	Azure APIM non consente di specificare in che modo indicare le credenziali nella richiesta; le API "non autenticate" sono in realtà anonime (autorizzate con credenziali note di default)	Buono	Buono
Reporting/ monitoring	Azure APIM si integra con servizi di reporting/monitoring Azure dedicati e centralizzati; l'admin portal di 3scale offre solo una sezione per analytics molto povera e minimale	Ottimo	Insufficiente
Controllo connessioni/traffico	Entrambi supportano impostazioni di *rate limiting*; Apicast consente una maggiore personalizzazione nella comunicazione con il Control Plane	Sufficiente	Ottimo
Policy enforcement	Azure APIM offre un buon livello di granularità in un editor intuitivo e smeplice da usare; 3scale consente di raggiungere i medesimi (e anche superiori) livelli di personalizzazione ma con un macchinoso e oneroso sistema di integrazione con script LUA custom	Buono	Buono
Espandibilità e costi			
Caratteristica	**Note**	**Azure APIM**	**RedHat 3scale**
Espandibilità	Azure APIM vanta l'integrazione con servizi proprietari per diverse esigenze; in 3scale è possibile estendere le policy di Apicast tramite dei custom script in LUA	Ottimo	Buono
Costi	3scale, oltre ad avere un costo molto più elevato rispetto al concorrente, necessita anche di una licenza OpenShift	Ottimo	Insufficiente

Tabella 4.9. Riepilogo dei principali aspetti confrontati

Capitolo 5

Conclusioni

5.1 Considerazioni finali

Nei capitoli precedenti sono stati definiti dei criteri di valutazione per l'applicabilità di soluzioni di API Management in contesti multi-cloud, coincidenti con i componenti comuni alle soluzioni stesse e senz'altro indispensabili per il deployment su più cloud provider. Un'architettura modulare, costituita da diversi elementi quali Control Plane, API gateway e Developer Portal, si rivela necessaria per l'utilizzo di un prodotto di API Management in un ambiente cloud distribuito.

Sono stati poi scelti due prodotti di API Management simili a livello di componenti architetturali, e sono stati confrontati su due fronti

- Test di performance del Data Plane: i gateway di entrambi i prodotti sono stati confrontati dal punto di vista delle perfomance sulla base di test effettuati nell'ambito di uno use-case realizzato appositamente, ovvero un'architettura multi-hybrid-cloud in uno scenario "segmented"; i test sono stati condotti in riferimento a 3 metriche comuni a entrambi i prodotti (tempo di risposta del backend, tempo impiegato dal gateway per rispondere al client e latenza totale)
- Analisi di 5 driver di security: i due prodotti di API Management sono stati confrontati sulla base di cinque funzionalità di sicurezza fondamentali e valutati, secondo i rispettivi criteri, in un contesto multi-cloud

Dai test di performance effettuati risulta che Il self-hosted gateway di Azure API Management risulta più performante rispetto ad Apicast, il gateway offerto da RedHat 3scale, dal momento che il primo presenta una minore latenza rispetto al secondo, conseguenza di un tempo di processamento interno inferiore rispetto a quest'ultimo.
Inoltre, dai test effettuati, in situazioni di carico sostenuto (20 req/s) il gateway di Azure risulta più efficiente dal momento che consuma meno CPU rispetto al concorrente (per la precisione meno della metà), ma utilizza circa il 30% in più di memoria. Al contrario, in stato di *idle*, Apicast presenta consumi estremamente ridotti, sia in termini di CPU che in termini di memoria: ciò lo rende particolarmente

adatto in contesti IoT, soprattutto in situazioni in cui la frequenza delle richieste è ridotta e dunque il tempo di *idle* nettamente superiore rispetto a quello di carico.

Infinte, dal confronto effettuato sulle funzionalità di sicurezza:

- Autenticazione: entrambe le soluzioni offrono supporto ad un framework esterno (OpenID Connect), consentendo la delegazione del processo di autenticazione. Inoltre, Azure API Management supporta l'autenticazione degli utenti anche tramite Azure Active Directory e RedHat 3scale anche tramite RedHat SSO.

- Autorizzazione: RedHat 3scale consente di specificare il tipo di credenziali da indicare nella richiesta, al contrario di Azure API Management; quest'ultimo supporta la creazione di API non autenticate, mentre 3scale consente solo di simularle (tramite credenziali di default)

- Reporting: i gateway di entrambe le soluzioni supportano l'esportazione delle metriche a Prometheus; tuttavia, Azure API Management vanta tutta una serie di servizi dedicati di monitoring, alerting e analytics integrati perfettamente nel prodotto tramite l'Azure Portal, mentre l'admin portal di 3scale offre solo una sezione per analytics molto povera e minimale

- Controllo connessioni / traffico: i Data Plane di entrambi i prodotti supportano funzionalità di *rate limiting* e mantengono una connessione costante con il Control Plane; tuttavia, Apicast offre la possibilità di configurazione di questa connessione da parte dell'amministratore, a differenza del self-hosted gateway di Azure, nel quale è predefinita

- Policy enforcement: entrambi i prodotti offrono un buon livello di granularità per la configurazione delle policies sui data plane; Azure offre un editor XML dedicato ed intuitivo, mentre 3scale consente solo una configurazione limitata di policies predefinite. Tuttavia, la soluzione di RedHat si differenzia da quella offerta da Azure in quanto offre la possibilità di realizzare delle policies custom, scritte in LUA.

Dall'analisi e dai confronti effettuati risulta evidente come entrambi i prodotti siano provvisti di soluzioni molto robuste dal punto di vista della sicurezza, garantendo dunque ottimi risultati sia in ambienti single-cloud che multi-cloud; si dimostrano molto validi in termini di capabilities, grazie anche al loro consistente background tecnologico che permette di offrire, oltre a performance stabili e feature esclusive, anche una consistente integrazione e piena compatibilità con servizi già esistenti, come OpenShift per RedHat e servizi di monitoring e reporting quali EventHub, Application Insights e Monitor per Azure.

Il caso d'uso analizzato, nonostante i test e le analisi effettuate siano agnostiche dallo scenario preso in considerazione, risponde a una precisa esigenza aziendale, la quale porta alla scelta univoca del tipo di ambiente distribuito da utilizzare. Tale scelta, descritta in fase di presentazione del caso d'uso, a sua volta obbliga la decisione su quali capabilities prendere in considerazione riguardo alle soluzioni analizzate: queste ultime, infatti, possiedono numerose altre funzionalità e caratteristiche per

altrettante necessità, come il supporto al DevOps o alla monetizzazione, rendendo dunque l'analisi effettuata in questo lavoro di tesi non esaustiva e non completa da questo punto di vista.

5.2 Possibili sviluppi futuri

L'infrastruttura realizzata permette di utilizzare le principali funzionalità delle soluzioni provate, sfruttandole al meglio per ottenere ottimi risultati nel caso d'uso proposto. Esiste, tuttavia, un margine di miglioramento dal punto di vista della **resilienza**. Inoltre, in termini di confronto tra le soluzioni, può essere interessante prendere in considerazione scenari multi-cloud differenti da quello analizzato nello use-case, valutando i due prodotti dal punto di vista del controllo dello stato di salute di un'API e dei tool di DevOps integrati offerti da entrambe le soluzioni.

Verrà avanzata di seguito una proposta per un possibile sviluppo futuro dell'infrastruttura, con l'obiettivo di migliorare la risposta delle due soluzioni in ambito Disaster Recovery rendendole più solide in situazioni di outage di control plane e/o data plane, e presentati due spunti di approfondimento per lo studio e il confronto delle due soluzioni in due ulteriori ambiti differenti.

5.2.1 Disaster recovery

In caso il Data Plane dovesse effettuare in riavvio in seguito a un crash o a un aggiornamento di versione e/o configurazione, sia Azure che RedHat prevedono che esista sempre una connessione tra il Control Plane e il gateway, assicurandosi che quest'ultimo abbia sempre la versione più recente dei parametri di configurazione. Talvolta può accadere, però, che tale connessione venga meno, e in questa situazione un eventuale crash del gateway potrebbe creare problemi per tutti i client che effettuano richieste verso di esso, causando una totale assenza del servizio. I gateway, tuttavia, sono progettati per caricare la configurazione anche da un file locale, come alternativa al download dello stesso dal Control Plane.

Azure API Management

Quando la connettività verso il Control Plane cade, il self-hosted gateway non sarà più in grado di ricevere aggiornamenti di configurazione o inviare report sullo status o sulla telemetria. Tuttavia, il self-hosted gateway è in grado di continuare a funzionare anche in temporanea assenza di connettività verso Azure.
È possibile configurare un local storage volume per il container del gateway: quest'ultimo effettuerà regolarmente una copia di backup della versione più aggiornata della configurazione nel volume indicato, in modo da recuperarla in caso di riavvio e mancanza di connessione verso il Control Plane.
Il mount path del volume deve essere necessariamente `/apim/config`.
Un esempio di configurazione in Kubernetes potrebbe essere il seguente (si riportano solo i campi rilevanti)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: private-gateway
spec:
  template:
    spec:
      containers:
      - name: private-gateway
        image: mcr.microsoft.com/azure-api-management/gateway:latest
        ...
      # crea un volume collegato al PVC in basso e lo monta nel container
        volumeMounts:
          - mountPath: /apim/config
            name: private-gateway-volume
      volumes:
        - name: private-gateway-volume
          persistentVolumeClaim:
              claimName: private-gateway-pvc

---
# Questo PVC si lega a un PV (Persistent Volume), già esistente o creato
# dinamicamente, di dimensione sufficiente e con StorageClass di default
# se configurata per il cluster.
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: private-gateway-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 50Mi # Liberamente configurabile in base all'esigenza
```

Se la configurazione di backup è spenta e la connettività verso Azure si interrompe:

- I self-hosted gateway in funzione continueranno a funzionare utilizzando la configurazione presente in memoria
- I self-hosted gateway spenti non saranno in grado di funzionare

Se la configurazione di backup è attiva e la connettività verso Azure si interrompe:

- I self-hosted gateway in funzione continueranno a funzionare utilizzando la configurazione presente in memoria
- I self-hosted gateway spenti potranno funzionare grazie alla copia di backup della configurazione

quando la connettività verrà ristabilita, ogni gateway che è stato interessato dall'interruzione si riconnetterà automaticamente al Control Plane ed effettuerà una sincronizzazione di eventuali aggiornamenti che si sono verificati durante l'assenza di connettività.

RedHat 3scale API Management

Come illustrato nel capitolo 4.6.2, Apicast consente di configurare la modalità di aggiornamento della configurazione grazie a due variabili d'ambiente. Tale aggiornamento avviene tramite una richiesta verso Control Plane, il cui endpoint è specificato nella variabile d'ambiente `THREESCALE_PORTAL_ENDPOINT` del container. Apicast offre un'altra modalità per il caricamento della configurazione: settando la variabile d'ambiente `THREESCALE_CONFIG_FILE` è possibile indicare il path locale di un file di configurazione, in modo tale che quest'ultimo venga caricato dal gateway al bootstrap. Nel caso in cui le due variabili siano settate contemporaneamente, `THREESCALE_CONFIG_FILE` avrà la precedenza.
Il gateway, dunque, allo startup, caricherà il file di configurazione specificato da `THREESCALE_CONFIG_FILE` solo se settata; in caso contrario o in caso di problemi durante questo processo, Apicast effettuerà una richiesta all'endpoint specificato da
`THREESCALE_CONFIG_FILE` per scaricare gli ultimi aggiornamenti di configurazione.
Nel caso in cui Apicast venga utilizzato come container, la documentazione consiglia di

1. Configurare il file nell'immagine come volume di sola lettura
2. Specificare il path che indica dove è stato montato il volume

Un esempio di comando `docker run` potrebbe essere il seguente

```
docker run
       -- publish 8080:8080
       -- volume ./config.json:/config.json
       quay.io/3scale/apicast:master
```

il quale esegue un mount del file di configurazione `config.json` presente sulla macchina locale all'interno del file system del container; quest'ultimo caricherà le configurazioni dal file indicato.
In questo modo, in caso di riavvio, il gateway provvederà a caricare la configurazione dal file montato in un volume esterno, garantendo la piena operatività anche in caso di assenza di connettività verso il Control Plane.

Purtroppo, nella documentazione non viene approfondito il comportamento di Apicast sull'utilizzo di tale file una volta caricata la configurazione: ad esempio, in stato di buona salute di Apicast e Control Plane, sarebbe opportuno da parte del gateway effettuare regolarmente un backup delle configurazioni scaricate periodicamente dal Control Plane, eventualmente sostituendo il file montato, in modo da avere una versione sempre aggiornata delle configurazioni ed evitare dunque di utilizzarne una precedente in caso di riavvio.

5.2.2 API Health check

La funzionalità di health check di un'API consente ad un sistema di API Management di rilevare lo stato di salute di un endpoint ed eseguire delle azioni in

caso di risultato negativo, come alerting, trigger di pipelines, reporting. Una delle più interessanti è sicuramente la possibilità di switching ad un eventuale endpoint configurato come backup.

Studiare i due prodotti in uno scenario multi-cloud **Parallel** sarebbe interessante per valutare le capabilities delle due soluzioni in termini di health check delle API in modo da configurare lo switch tra i due (o più) CSP.
In uno use-case d'esempio, un cliente decide di replicare i workload posseduti su un cloud provider, chiamato *Cloud 1*, su un secondo cloud provider, chiamato *Cloud 2*, per avere un backup dei servizi stessi. Tali servizi sono esposti tramite API, in modo identico su entrambi i cloud provider. Si potrebbero valutare le due soluzioni di API Management in tale contesto, per capire il livello di supporto all'health check degli endpoint raggiungibili tramite i gateway e valutarne la qualità; secondo tale meccanismo, in caso di health check negativo per un'API sul *Cloud 1*, le richieste vengono reindirizzate verso l'endpoint replicato sul *Cloud 2*, valutando metriche quali ad esempio

- Rapidità di rilevazione dello stato di salute dell'API
- Latenza attuale e percepita (da parte del client finale) dello switch da un endpoint alla sua replica nel secondo CSP

e valutando una soluzione tanto migliore quanto minore è l'impatto sull'esperienza del client finale (in caso di caduta dell'endpoint verso un workload sul *Cloud 1*, garantire la continuità del servizio effettuando lo switch verso il *Cloud 2* il prima possibile), naturalmente tenendo conto della presenza di eventuali falsi positivi (dovuti ad esempio ad assenza di connettività o congestione della rete), la quale può essere controproducente per la soluzione più veloce nella rilevazione del guasto o nell'esecuzione dello switch.

5.2.3 API lifecycle DevOps

La disciplina DevOps è l'insieme di processi e tecnologie utilizzate per offrire continuamente un valore ai clienti. DevOps permette a ruoli in precedenza isolati, tra cui sviluppo, operazioni IT, controllo della qualità e sicurezza, di coordinarsi e collaborare per fornire prodotti migliori e più affidabili, rispondendo meglio alle esigenze dei clienti e raggiungere più rapidamente gli obiettivi aziendali. Le pratiche più diffuse ci sono sicuramente quelle di CI (Continuous Integration, ovvero, per gli sviluppatori, quel processo che prevede il merging continuo di piccole e frequenti modifiche in un branch principale) e CD (Continuous Delivery o Continuoud Deployment, ovvero quel processo che consiste nella distribuzione del software frequente e automatica).

L'approfondimento delle capabilities dei due prodotti in termini di DevOps ha sicuramente enorme valore in uno scenario multi-cloud **Portable**, in cui automazione e astrazione dal cloud provider sono fondamentali. La regola per un'elevata portabilità delle API in un contesto simile è proprio la massima automatizzazione di tutti i processi riguardanti il loro lifecycle, a partire dalla progettazione sino ad arrivare alla distribuzione e pubblicazione del prodotto. Potrebbe essere sensato,

dunque, valutare entrambi i prodotti in un caso d'uso dedicato, in cui un cliente decide di progettare i propri workload massimizzando la loro portabilità, in vista della migrazione dei servizi tra differenti CSP per evitare il vendor lock-in e dunque massimizzare il proprio potere di negoziazione.
Sia RedHat 3scale API Management che Azure API Management offrono dei tool integrati per l'automazione di processi di CI/CD, 3scale toolbox da una parte e Azure API Management DevOps Resource Toolkit dall'altra: studiare tali tool in termini qualitativi, valutando il numero di capabilities di DevOps offerte in termini di API lifecycle e la loro efficacia, nonchè la complessità di utilizzo (command line o interfaccia grafica) può essere estremamente rilevante per la scelta del tool da utilizzare, soprattutto in quei contesti in cui la metodologia DevOps ricopre un'elevata importanza nei processi di distribuzione e testing del software.

Bibliografia

[1] VentureBeat, "Google Maps Platform arrives with pay-as-you-go billing, free support, and Cloud requirement starting June 11", Available: https://venturebeat.com/business/google-maps-platform-arrives-with-pay-as-you-go-billing-free-support-and-cloud-requirement-starting-june-11/, online

[2] Programmableweb.com, "APIs show Faster Growth Rate in 2019 than Previous Years", Available: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17, online

[3] Netflix, "Completing the Netflix Cloud Migration", Available: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration, online

[4] Forrester, "Predictions 2022: Cloud Computing Reloaded", Available: https://www.forrester.com/blogs/predictions-2022-cloud-computing-reloaded/, online

[5] Gregor Hohpe, "Multi Cloud Architecture: Decisions and Options", Available: https://architectelevator.com/cloud/hybrid-multi-cloud/, online

[6] Gartner, "Magic Quadrant for Full Life Cycle API Management", Available: https://www.gartner.com/doc/reprints?id=1-26OL5RKA&ct=210630&st=sb, online

[7] Forrester, "Executive Guide 2022: Cloud", Available: https://reprints2.forrester.com/#/assets/2/108/RES177084/report?ocid=eml_pg362109_gdc_comm_az, online

[8] Flexera, "Cloud Computing Trends: Flexera 2022 State of the Cloud Report", Available: https://www.flexera.com/blog/cloud/cloud-computing-trends-2022-state-of-the-cloud-report/, online

[9] FinOps Foundation, "What is FinOps", Available: https://www.finops.org/introduction/what-is-finops/, online

[10] RedHat, Available: https://developers.redhat.com/sites/default/files/3scale_api_graphic.png, online

[11] Kong, Available: https://docs.konghq.com/assets/images/docs/ee/deployment/deployment-hybrid-2.png, online

[12] Google, Available: https://youtu.be/-UW8umABp9k, online

[13] Microsoft Azure, Azure API Management documentation, https://learn.microsoft.com/en-us/azure/api-management/, online

[14] Azure, "Azure API Management Grand Tour", Available: https://azure.github.io/apim-lab/assets/slides/Grand%20tour%20of%20Azure%20API%20Management.pdf, online

[15] Azure, "API Management in a hybrid and multi-cloud world", Available: https://aka.ms/apimarc, online
[16] Azure, "Build a CI/CD pipeline for API Management", Available: https://azure.microsoft.com/it-it/blog/build-a-ci-cd-pipeline-for-api-management/, online
[17] Azure, Available: https://devblogs.microsoft.com/wp-content/uploads/sites/31/2019/04/azapimgr.png, online
[18] RedHat, RedHat 3scale API Management documentation, https://access.redhat.com/documentation/en-us/red_hat_3scale_api_management/2.11, online
[19] Google, Google Apigee Hybrid documentation, https://cloud.google.com/apigee/docs/hybrid/v1.8/what-is-hybrid, online
[20] Kong, Kong gateway documentation, https://docs.konghq.com/, online
[21] AWS, Amazon API gateway documentation, https://aws.amazon.com/it/api-gateway/, online
[22] Docker, Docker documentation, https://docs.docker.com/, online
[23] Kubernetes, Kubernetes documentation, https://kubernetes.io/docs/home/, online
[24] Rancher, K3S documentation, https://docs.k3s.io/, online
[25] RedHat, OpenShift documentation, https://docs.openshift.com/container-platform/4.10/welcome/index.html, online
[26] Prometheus, Prometheus documentation, https://prometheus.io/docs/introduction/overview/, online