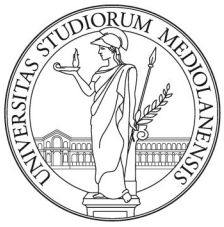

UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Sicurezza Dei Sistemi e Delle Reti Informatiche

DESIGN E SVILUPPO DI UN SISTEMA DISTRIBUITO AVANZATO PER VERIFICHE DI SECURITY ASSURANCE

Relatore: Prof. Marco ANISETTI

Correlatori: Dott. Nicola BENA, Dott. Filippo BERTO

Tesi di:
Alex DELLA BRUNA
Matricola: 986715

Anno Accademico 2023-2024

Prefazione

Le nuove tecnologie edge, cloud e ibride stanno rivoluzionando il modo in cui le organizzazioni gestiscono le proprie risorse informatiche. La flessibilità e l'elasticità offerte dai servizi cloud permettono di ridurre i costi e di aumentare l'efficienza, ma al contempo introducono nuove sfide in termini di sicurezza e compliance. In questo contesto, la security assurance, ovvero la verifica continua di proprietà non funzionali, e la certificazione dei servizi cloud, ovvero il rilascio di certificati di conformità in seguito al superamento di opportune verifiche, diventano sempre più importanti per garantire la sicurezza e la privacy dei dati degli utenti. Si rendono quindi necessarie nuove metodologie e strumenti per valutare e certificare la sicurezza dei servizi cloud, on-premise e ibridi, al fine di garantire la conformità alle normative vigenti e la protezione dei dati sensibili. In tal senso, Moon Cloud offre una piattaforma di assurance per la valutazione e certificazione della sicurezza dei servizi e delle infrastrutture ICT. L'obiettivo di questa tesi è quello di modernizzare l'infrastruttura di Moon Cloud esistente per supportare la security assurance e la certificazione dei servizi in qualsiasi modalità di deployment, dal cloud, all'on-premise e ibridi. La nuova infrastruttura di Moon Cloud è stata progettata per garantire la sicurezza e la privacy dei dati degli utenti, offrendo una piattaforma sicura e un ambiente di lavoro sicuro e affidabile per la certificazione dei servizi e dei sistemi. La tesi descrive le criticità del sistema esistente e gli obiettivi, vincoli e azioni che hanno guidato la migrazione verso la nuova infrastruttura. Vengono illustrate le scelte progettuali e le soluzioni implementative adottate per la realizzazione e validazione. Viene presentato un caso di studio reale che ne dimostra un flusso di funzionamento completo. Vengono infine presentati degli esperimenti condotti per valutare le prestazioni della nuova infrastruttura e confrontarle con quelle dell'architettura precedente.

Organizzazione della Tesi

Il lavoro di tesi è strutturato come segue:

- **Capitolo 1**: definisce il contesto in cui si colloca il lavoro di tesi, descrive le tematiche di assurance e certificazione e le infrastrutture moderne di cloud computing.
- **Capitolo 2**: presenta lo stato attuale della security assurance e certificazione e fornisce una breve panoramica sull'infrastruttura di Moon Cloud.
- **Capitolo 3**: descrive la roadmap di migrazione progettata in termini di criticità del sistema esistente, obiettivi, vincoli e azioni che hanno guidato verso una nuova infrastruttura.
- **Capitolo 4**: illustra le scelte progettuali e le soluzioni implementative adottate per implementare la roadmap.
- **Capitolo 5**: presenta un caso di utilizzo della nuova versione di Moon Cloud, nello specifico, una verifica di assurance per applicazioni basate su Machine Learning, ne descrive l'integrazione con la nuova infrastruttura di Moon Cloud e ne presenta i risultati.
- **Capitolo 6**: descrive gli esperimenti condotti per valutare le prestazioni della nuova infrastruttura di Moon Cloud e un confronto con l'architettura precedente.
- **Capitolo 7**: riassume i risultati ed i vantaggi ottenuti della nuova infrastruttura di Moon Cloud e presenta i possibili sviluppi futuri.

Ringraziamenti

Ringrazio sinceramente il mio relatore, prof. Marco Anisetti per le molteplici opportunità offerte in questi anni, per la sua guida e il suo supporto costante. Ringrazio i miei correlatori, dr. Nicola Bena e dr. Filippo Berto, per il loro prezioso contributo, per avermi sempre sostenuto e fornito innumerevoli consigli. Ringrazio tutti i membri del SESAR Lab per l'ambiente di lavoro stimolante e collaborativo e per avermi sempre supportato, in particolare ringrazio il dr. Antongiacomo Polimeno e il dr. Ruslan Bondaruc per il loro aiuto e la loro disponibilità. Ringrazio infine la mia famiglia per il loro costante supporto e per avermi sempre incoraggiato durante tutto il percorso.

Alex Della Bruna

Indice

Elenco delle Figure

Elenco delle Tabelle

Capitolo 1. Introduzione

Negli ultimi anni le architetture software sono cambiate radicalmente, passando da sistemi monolitici a sistemi distribuiti, scalabili, e flessibili. Questo cambiamento è stato reso possibile grazie all'avvento del cloud computing, che ha permesso di sviluppare, distribuire e gestire applicazioni in modo semplice ed efficiente. Si è passati da architetture monolitiche a architetture a microservizi, che permettono di sviluppare applicazioni modulari, flessibili, e dinamiche. Questo cambiamento ha portato nuove sfide e opportunità, in particolare sul fronte della sicurezza delle applicazioni, che sono sempre più esposte a minacce e attacchi informatici. In questo contesto il lavoro di tesi si è sviluppato nella definizione di un'architettura moderna e scalabile per l'assurance e certificazione delle applicazioni.

1.1 Cloud Computing, Microservizi, e Architetture Scalabili

Il cloud computing ha rivoluzionato il modo in cui le applicazioni vengono sviluppate, distribuite e gestite. Le applicazioni moderne vengono sviluppate sotto forma di microservizi, in particolare, si definisce un microservizio come un'unità di sviluppo indipendente che svolge una specifica funzionalità, riuscendo così a specializzare lo sviluppo delle applicazioni e ottimizzare le pipeline di sviluppo. L'esecuzione avviene tramite architetture scalabili che permettono di adattare le risorse computazionali in base alle esigenze del sistema, garantendo un'alta disponibilità e scalabilità. Si tratta, in particolare, di architetture a microservizi, dove ogni servizio viene solitamente implementato sotto forma di container che comunica con gli altri servizi tramite API RESTful e gRPC. Questo permette di avere un sistema modulare, flessibile e scalabile, in cui ogni servizio può essere sviluppato, testato e rilasciato indipendentemente dagli altri. Le funzionalità di containerizzazione sono state rese possibili grazie a Docker, un progetto open-source che permette di creare, distribuire e gestire container in modo semplice ed efficiente. Un container è un'istanza di un'immagine, che contiene il codice sorgente, le librerie e le dipendenze necessarie per eseguire un'applicazione in modo isolato, creando così un ambiente di esecuzione indipendente dal sistema operativo sottostante, garantendo la portabilità e la compatibilità dell'applicazione su qualsiasi sistema.
Sebbene Docker svolga un ottimo lavoro sul fronte della containerizzazione degli applicativi, deficita di funzionalità avanzate di orchestrazione, perciò bisogna ricorrere ad applicativi esterni. L'applicativo che ha maggiormente contribuito negli ultimi anni sul fronte dell'orchestrazione lato container è Kubernetes, un orchestratore open-source che permette di gestire, scalare e monitorare i container in modo semplice ed efficiente. Vengono in particolare definiti nuovi concetti

quali *pod*, *deployment*, *service*, *ingress*, *namespace*, etc. che permettono di definire e gestire l'intera infrastruttura in modo dichiarativo e scalabile, ogni componente Kubernetes implementa una specifica funzionalità e garantisce una gestione indipendente delle funzionalità architetturali. In particolare, l'entità più importante di Kubernetes è il pod, che rappresenta un gruppo di container che vengono gestiti come unità atomica. Si hanno poi i deployment, che permettono di definire e gestire il ciclo di vita di un'applicazione, garantendo la scalabilità, la disponibilità e la replica dei container. I service permettono di esporre un'applicazione, garantendo la comunicazione tra i vari servizi in modo trasparente e scalabile. Infine gli ingress permettono di esporre un'applicazione all'esterno, garantendo la comunicazione con il mondo esterno. Ognuna di queste entità è infine racchiusa in un namespace, che permette di definire e gestire l'intera infrastruttura in modo isolato. Unendo le funzionalità di Docker e Kubernetes si ottiene un ambiente di sviluppo, distribuzione e gestione delle applicazioni moderno, flessibile e scalabile, che permette di sviluppare applicazioni moderne.

1.2 Sicurezza, Assurance, e Certificazione

La sicurezza delle applicazioni, al contempo, è diventata un aspetto fondamentale, in quanto i sistemi informatici sono sempre più esposti a minacce e attacchi informatici. Vi è inoltre la necessità di garantire la compliance dei sistemi e delle applicazioni, questo viene fatto dall'assurance, definita come garanzia di comportamento corretto, sia in termini funzionali che normativi. Sotto quest'ottica il mondo della security assurance si è dovuto aggiornare per supportare i nuovi modelli di sviluppo, distribuzione e gestione delle applicazioni. Si sono dovute affrontare nuove sfide in quanto si è passati dalla certificazione di sistemi statici alla certificazione di sistemi fortemente dinamici, mutevoli e scalabili. E' quindi stato necessario sviluppare nuove metodologie e strumenti per garantire la sicurezza delle applicazioni in un ambiente cloud-native, in particolare sul lato della sicurezza dei container e delle applicazioni distribuite. Per fare ciò, il lavoro di tesi si è concentrato sullo sviluppo di una nuova architettura ad alte prestazioni che prevede la possibilità di deployment in diverse modalità, al fine di supportare la certificazione di tutti gli asset interni ed esterni. Partendo dall'implementazione di riferimento già esistente di Moon Cloud, non adatta per supportare scenari fortemente distribuiti e dinamici, si è definita e implementata la struttura della nuova architettura, basata su Kubernetes, che tramite l'uso di microservizi permette di garantire la certificazione delle applicazioni in modo flessibile e scalabile. Nella fase di migrazione si è tenuto conto di numerosi requisiti quali la scalabilità, la disponibilità, la sicurezza, la flessibilità e la compatibilità con i componenti già esistenti. Si illustra quindi il processo di migrazione dell'architettura di Moon Cloud, partendo dalle criticità del sistema esistente alla

definizione e implementazione degli obiettivi. Viene infine illustrato un caso di studio che mostra il flusso di esecuzione del nuovo sistema e degli esperimenti di comparazione prestazionale rispetto all'architettura precedente.

Capitolo 2. Stato dell'Arte

Nel mondo moderno è sempre più importante garantire la sicurezza e la qualità dei sistemi software ad ogni livello, a questo scopo vengono utilizzate le tecniche di assurance e certificazione. Le tecniche di assurance mirano a garantire che un sistema soddisfi certi requisiti non funzionali di sicurezza e qualità, mentre la certificazione è il processo di verifica che attesta la conformità del sistema ai requisiti definiti. In questo contesto si colloca Moon Cloud, un sistema cloud-native decentralizzato per verifiche di assurance, che permette di garantire la sicurezza e la qualità dei sistemi software in modo flessibile e scalabile. In questo capitolo verranno presentati i concetti di assurance e certificazione e l'architettura del sistema Moon Cloud.

2.1 Security Assurance e Certificazione

L'assurance è un processo che permette di garantire che un sistema soddisfi, nell'istante di verifica e in modo continuativo, certi requisiti non funzionali di sicurezza e qualità. La certificazione è la tecnica di assurance maggiormente adottata, quando il sistema supera con successo il processo di verifica, viene rilasciato un certificato che garantisce la conformità del sistema ai requisiti di sicurezza e qualità definiti. Lo stato dell'arte dei sistemi di assurance e certificazione è in continua evoluzione, si stanno sviluppando nuovi approcci che coinvolgono non solo la classica fase di certificazione al termine dello sviluppo, ma metodologie innovative che prevedono l'inserimento di controlli di assurance lungo l'intera pipeline di sviluppo, sono quindi nati nuovi approcci che mirano al raggiungimento del *DevSecOps* [4] o ancora più ambiziosi che mirano al raggiungimento del *DevCertOps* [3].

Formalmente uno schema di certificazione definisce il processo di verifica per accertare che il sistema si comporti in modo conforme a quanto ci si aspetta e che dimostri di possedere certe proprietà non funzionali. La Figura 1 mostra il processo di certificazione, così articolato, come prima cosa la *Certification Authority (CA)* definisce il *Certification Model* per accertare le proprietà su uno specifico target detto *ToC (Target of Certification)* in relazione al *Evidence Collection Model*, successivamente l'*Accredited Lab*, delegato dalla *CA*, istanzia l'*Evidence Collection Model* per raccogliere evidence e verificarne il rispetto delle proprietà dichiarate. Se il *ToC* supera con successo il processo di accreditamento, l'*Accredited Lab* rilascia un certificato che attesta la conformità del *ToC* alle proprietà dichiarate. Nel dettaglio si definiscono:

- *ToC (Target of Certification)*: entità che vuole ottenere la certificazione e che deve dimostrare di possedere certe proprietà non funzionali

- Proprietà non funzionali: proprietà che riguardano il comportamento in termini di sicurezza, qualità, affidabilità, ecc., piuttosto che il comportamento operativo del sistema
- *Evidence Collection Model*: insieme di regole e procedure che definiscono come raccogliere le evidenze per verificare le proprietà non funzionali

Un possibile esempio di certificazione è lo standard *ISO 27001*, che definisce come gestire la sicurezza delle informazioni in un'organizzazione, in particolare definisce i requisiti per stabilire, implementare, mantenere e migliorare un sistema di gestione della sicurezza delle informazioni (*ISMS*) all'interno del contesto dei rischi globali dell'organizzazione. In questo caso il *ToC* è l'organizzazione che vuole ottenere la certificazione, le proprietà non funzionali sono i requisiti definiti dalla *ISO 27001* e l'*Evidence Collection Model* è il sistema di audit e di raccolta delle evidenze per dimostrare la conformità ai requisiti.

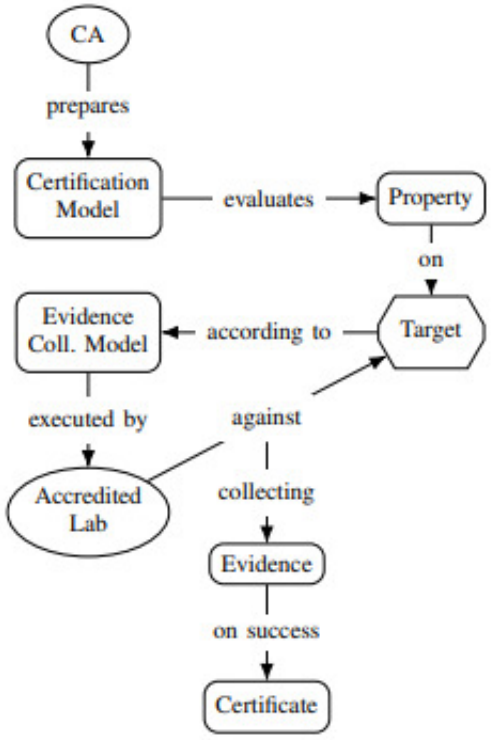

Figura 1: Modello di certificazione [3]

2.2 Moon Cloud: un Sistema per l'Assurance

Moon Cloud [2] [24] è un sistema cloud-native decentralizzato per verifiche di assurance, si basa su un'architettura modulare e scalabile, con possibilità di deployment full-managed, hybrid e on premise. Offre una completa visione dello stato di certificazione e compliance del sistema in tempo reale grazie ad un serie di controlli(probe) che vengono eseguiti in modo continuo.

2.2.1 Entità

Le entità principali coinvolte nel processo di assurance utilizzate da Moon Cloud sono:

- **Target**: entità che si vuole certificare, può essere un sistema, un'applicazione, un servizio, ecc.
- **Control**: controllo che deve essere eseguito sul target per verificarne la conformità
- **AER (Abstract Evaluation Rule)**: regola di valutazione che definisce i controlli da eseguire sul target e i criteri di valutazione
- **UER (User Evaluation Rule)**: istanziazione di una regola di valutazione sulla base dell'input dell'utente, permette di adattare le regole di valutazione agli standard e alle policy dell'utente
- **Credential**: credenziali necessarie per l'esecuzione di alcuni controlli
- **Test**: singola istanziazione di un controllo verso un target
- **Test Execution**: singolo risultato dell'esecuzione di un test
- **Evaluation Execution**: singolo risultato dell'esecuzione di una regola di valutazione, formato dall'unione di più test execution

2.2.2 Probe

Le probe sono le entità centrali del sistema Moon Cloud, gestiscono l'esecuzione dei controlli sui target e la raccolta delle evidenze. Ogni probe è un container Docker che esegue un controllo specifico, ne analizza i risultati e li invia al sistema centrale per la valutazione. Nel dettaglio si presenta come una macchina a stati finiti con due catene di esecuzione, in particolare una che gestisce il funzionamento normale (forward) e una nel caso si verifichino errori (rollback) che ne provvede alla gestione. Ogni controllo infine ritorna un risultato costituito da tre campi:

Risultato	Descrizione
INTEGER_RESULT_TRUE (0)	Risultato positivo
INTEGER_RESULT_FALSE (1)	Risultato negativo
INTEGER_ RESULT_ TARGET_ CONNECTION_ ERROR (2)	Errore di connessione al target
INTEGER_ RESULT_ TARGET_ EXECUTION_ ERROR (3)	Errore di esecuzione sul target
INTEGER_ RESULT_ INPUT_ ERROR (4)	Errore di input
INTEGER_ RESULT_ MOON_ CLOUD_ ERROR (5)	Errore interno
INTEGER_ RESULT_ UNKNOWN (6)	Errore sconosciuto

Tabella 1: Risultati della valutazione della probe

- **integer_result**: risultato numerico della valutazione che rappresenta lo stato di uscita della probe, la figura 1 mostra i possibili risultati della valutazione della probe.
- **pretty_result**: risultato testuale della valutazione che rappresenta lo stato di uscita della probe
- **base_extra_data**: dati aggiuntivi che descrivono nel dettaglio lo stato rilevato

2.2.3 Architettura

Il sistema si compone come segue:

- **Dashboard**: interfaccia utente che permette l'interazione grafica dell'utente finale con l' API Moon Cloud
- **API**: set di funzionalità per la gestione completa di tutte le entità dell'ecosistema Moon Cloud (controlli, target, AER, UER, credenziali, ecc.)
- **Evidence DB**: database utilizzato come single point of trust per le evidenze raccolte
- **Director**: scheduler interno delle probe, consente la ripetizione a certi intervalli di tempo delle regole di valutazione utente(UER) preconfigurate
- **Director DB**: database interno utilizzato per il funzionamento del Director
- **Master**: servizio che permette l'invio delle richieste di esecuzione delle regole di valutazione e dei relativi controlli
- **Master DB**: database interno utilizzato per il funzionamento del Master, tiene traccia delle code SQS su cui inviare i task

- **AWS SQS** [6]: servizio di storage online temporaneo che consente l'approccio pub/sub
- **Puppet**: esecutore effettivo delle regole di valutazione, analizza le regole e ne avvia i relativi controlli tramite chiamate al Docker engine
- **Credential Sniffer**: servizio di lettura delle credenziali precedentemente inviate tramite coda SQS
- **Docker engine** [10]: motore di containerizzazione presente sull'esecutore delle probe, necessario per il relativo avvio
- **Credential Manager**: gestore delle credenziali interno, permette l'injection delle credenziali dove richieste dalle probe
- **Probe**: controllo effettivo che verrà lanciato contro il target della valutazione
- **Evidence Writer**: servizio di valutazione, aggregazione e salvataggio dei risultati ricevuti dalle probe
- **Evaluation Result Manager**: modulo di valutazione collegato strettamente all'Evidence Writer

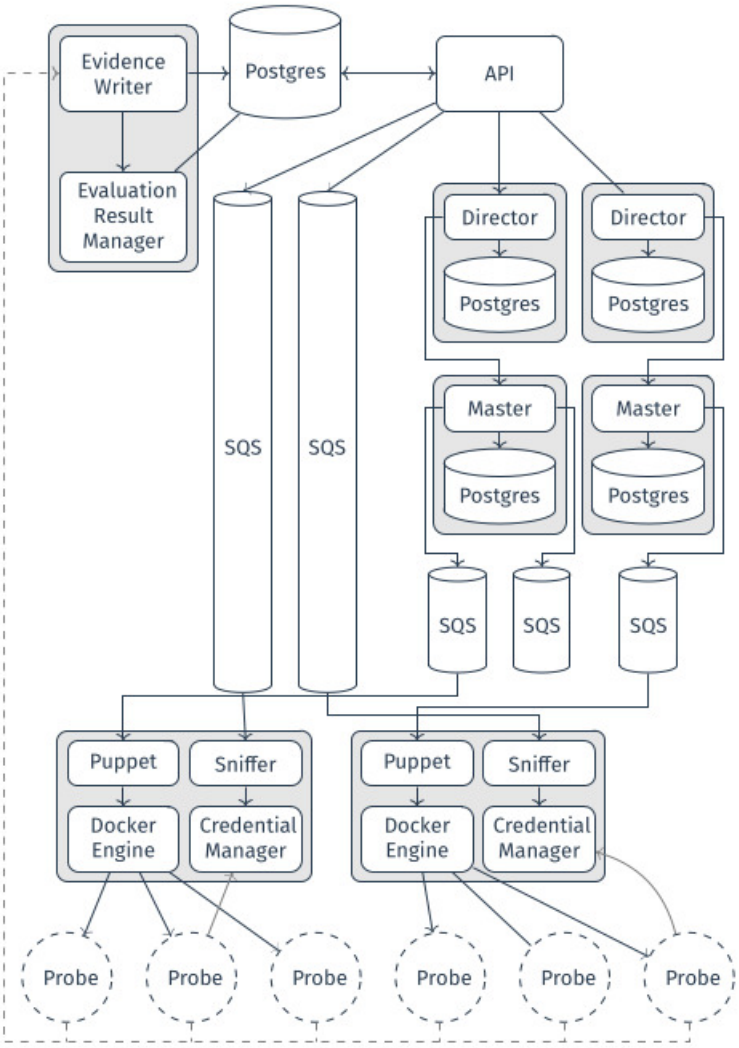

Figura 2: Modello architettura precedente

2.2.4 Flusso d'Esecuzione

Una volta che l'utente effettua l'accesso alla Dashboard può impostare la configurazione di tutte le entità coinvolte nel funzionamento:credenziali, zone, target e regole di valutazione. L'utente può aggiungere le credenziali necessarie per l'esecuzione di controlli legate a target che ne necessitano, queste saranno ricevute dall'API ed inviate tramite la coda SQS dell'utente corrente. Il Credential Sniffer procederà alla lettura delle credenziali e successivamente le salverà sul Creden-

tial Manager. L'utente terminate tutte le attività di configurazione può avviare la regola di valutazione, quindi l'API prende in carico la richiesta registrandone la creazione e invia il task al Director relativo all'utente corrente. Il Director se il job è one-shot provvederà semplicemente all'invio al Master, altrimenti se è un job di tipo scheduled salverà il job nel suo DB interno e provvederà ad inviarlo al Master all'intervallo di tempo configurato. Il Master ricevuto il task di esecuzione provvederà all'invio di esso sulla coda SQS relativa all'utente e il Puppet provvederà a scaricarla ed inviare l'esecuzione al Docker engine installato sull'esecutore. Successivamente le probe entreranno in esecuzione, scaricheranno le relative credenziali dal Credential Manager, effettueranno i relativi controlli ed invieranno il risultato all'Evidence Writer. Infine quest ultimo provvederà alla valutazione, aggregazione e sincronizzazione dei risultati per il salvataggio finale.

2.2.5 Problematiche e Obiettivi

Il sistema Moon Cloud, pur essendo basato su un'architettura modulare e scalabile, presenta alcune problematiche che ne limitano l'efficienza e la scalabilità. In particolare si evidenziano problemi legati all'eccessivo numero di componenti custom e al deployment in un sistema legacy. Partendo da ciò si è deciso di ristrutturare l'architettura del sistema Moon Cloud, con l'obiettivo di ridurre il numero di componenti custom e di adottare un approccio cloud-native per il deployment del sistema, segue quindi il capitolo dedicato alla progettazione e migrazione del nuovo sistema Moon Cloud.

Capitolo 3. Migrazione

Il sistema esistente presenta molteplici criticità, che ne limitano l'efficienza e la scalabilità (3.1). Pertanto l'obiettivo della migrazione è stato quello di rendere il sistema più moderno, efficiente, flessibile, affidabile e sicuro, per fare ciò si sono quindi analizzate le criticità del sistema esistente, definiti gli obiettivi (3.2) da raggiungere e i vincoli (3.3) da rispettare, per poi definire le azioni da intraprendere per raggiungere gli obiettivi e rispettare i vincoli.

3.1 Criticità Sistema Esistente

Il sistema esistente presenta molteplici criticità: la gestione delle credenziali troppo semplice, la dipendenza da un sistema pub/sub esterno e l'eccessiva presenza di componenti custom sono strettamente collegate alla scelta di un deployment in un sistema legacy. Si sono inoltre riscontrate delle criticità sulla non puntualità dei risultati delle scansioni. Nel dettaglio:

C1 **Gestione delle credenziali basica**: le credenziali sono salvate in un database criptato custom, senza ampia possibilità di definire policy di accesso, rotazione, revoca, modalità di accesso tramite sistemi moderni come Kubernetes, Oauth2.0, OpenID Connect, ecc.

C2 **Dipendenza da un sistema pub/sub esterno(SQS) [6]**: il sistema è dipendente da un sistema esterno per la comunicazione tra i vari componenti, rendendo elevata la latenza nella comunicazione e non garantendo la scalabilità e l'affidabilità del sistema.

C3 **Eccessiva presenza di componenti custom**: il sistema è composto da molti componenti custom, rendendo difficile la manutenzione e la scalabilità del sistema.

C4 **Deployment in un sistema legacy**: il sistema viene eseguito tramite Docker Swarm[11], non garantendo la scalabilità, l'affidabilità e la replicazione dei vari servizi, inoltre non essendo ampiamente supportato dalla community, non garantiva la possibilità di futuri sviluppi.

C5 **Risultati non puntuali**: i risultati delle scansioni non sono collegati in maniera puntuale con le scansioni effettuate, rendendo difficile la valutazione dei risultati e la loro visualizzazione all'utente finale.

Queste criticità fan sì che il sistema non sia facilmente manutenibile, scalabile, affidabile e sicuro, ne segue quindi la definizione degli obiettivi da raggiungere e dei vincoli da rispettare.

3.2 Obiettivi

Poste le criticità sopra descritte, lo scopo della migrazione è stato quello di rendere il sistema più moderno, efficiente, flessibile, affidabile e sicuro, per fare ciò si sono definiti gli obiettivi da raggiungere:

O1 **Costruzione di un sistema modulare facilmente installabile nelle modalità managed, hybrid e on-premise (C3,C4)**: il sistema deve essere garantire la possibilità di installazione in ambienti diversi e la possibilità di futuri sviluppi. Le varie modalità permettono l'esecuzione anche in presenza di forti vincoli normativi ed elevate richieste di privacy. In particolare la modalità managed mira a garantire la massima semplicità di installazione e manutenzione, è adatta a clienti con richieste di privacy non elevate e che necessitano di verifiche di conformità solo su target pubblicamente raggiungibili. La modalità hybrid permette di avere un cluster Moon Cloud installato on-premise dal cliente, garantendo così la possibilità di effettuare verifiche anche su target interni, è adatto a clienti con richieste di privacy medie che necessitano di verifiche di conformità anche su target interni, ma senza dovere gestire la complessità di installazione di un cluster Moon Cloud. Infine la modalità on-premise permette di interconnetere un cluster on-premise di proprietà del client con l'infrastruttura backend di Moon Cloud, questo garantisce il mantenenimento totale del controllo sull'infrastruttura e sui dati, è adatto a clienti con richieste di privacy elevate che necessitano di verifiche di conformità anche su target interni e che hanno la necessità di mantenere il controllo totale sull'infrastruttura e sui dati.

O2 **Garanzia di affidabilità, scalabilità e replica (C1, C2, C4)**: il sistema deve garantire la possibilità di scalare, replicare e garantire l'affidabilità dei vari servizi

O3 **Migrazione verso un sistema cloud-native (C2, C4)**: rimozione dei componenti custom non necessari e migrazione verso un sistema cloud-native, al fine di garantire le proprietà di scalabilità, affidabilità e replicazione

O4 **Risultati fine-grained (C5)**: i risultati delle scansioni devono essere collegati in maniera puntuale con le scansioni effettuate, rendendo la valutazione e visualizzazione dei risultati più dettagliata e collegata con le scansioni effettuate anche in termini temporali

3.3 Vincoli

Sono stati individuati i seguenti vincoli:

V1 **Interconnessione fra microservizi**: ogni componente dipende strettamente dal corretto funzionamento dei componenti che lo precedono, è necessario garantire la compatibilità dei vari formati d'interscambio al fine di garantire la corretta comunicazione tra i vari componenti

V2 **Sicurezza delle informazioni**: essendo il sistema operante su dati spesso sensibili, quali credenziali e risultati delle scansioni, è necessario garantire un elevato livello di sicurezza delle informazioni salvate durante tutta la catena di microservizi

V3 **Elevate prestazioni**: è necessario prevedere il supporto ad una grande mole di dati, per cui è necessario garantire le proprietà di affidabilità e scalabilità

V4 **Manutenibilità**: è necessario costruire un sistema che sia facilmente manutenibile, avendo un'architettura che possa essere adatta anche in ottica di futuri sviluppi

3.4 Azioni

La migrazione ha previsto la divisone del sistema in due sottosistemi:

- **Evaluation Center**: insieme dei servizi che gestiscono l'interazione con l'utente, la comunicazione con la customer zone e la valutazione dei risultati, è composto da:
  - **Dashboard**: interazione con l'utente finale
  - **API**: backend ingress dell'infrastruttura, gestisce la comunicazione tra la Dashboard e la customer zone, gestisce tutte le entità coinvolte(user,project,target,zone,ecc.) e avvia l'esecuzione dei task tramite invio all'esecutore tramite subject NATS.
  - **NATS** [25]: bus di comunicazione subject-based che intermedia la comunicazione tra API ed esecutore e tra probe ed Evidence Writer.
  - **Evidence Writer**: riceve in input i risultati delle probe, aggrega, sincronizza e valuta l'esecuzione secondo le regole di valutazione prefissate e salva il risultato per una successiva lettura da parte dell'API.
- **Customer Zone**: insieme dei servizi che gestiscono l'esecuzione dei controlli, può essere managed o on-premise:

  - **K8s Manager**: esecutore effettivo delle probe, gestisce la comunicazione col control plane del cluster managed o on-premise e avvia l'esecuzione di job/cron job
  - **HashiCorp Vault** [15]: sistema di gestione delle credenziali, salva e gestisce il lifecycle delle credenziali che necessitano di essere usate nelle probe, applica le relative policy per garantire l'utilizzo solo all'utente autorizzato e inietta le credenziali nelle probe di destinazione
  - **Probe**: controlli effettivi, verificano la compliance del target con una particolare proprietà e inviano il risultato per la valutazione all'Evidence Writer tramite NATS

Per il raggiungimento degli obiettivi e garantire i vincoli precedenti si sono attuate le seguenti misure:

- sistemazione dell'interfaccia grafica(Dashboard) per semplificare la visualizzazione dei risultati all'utente finale (C5, O4)
- aggiunta del nuovo componente Evidence Writer per effettuare una valutazione più approfondita dei risultati ritornati dalle probe (C5, O4, V4)
- migrazione a nuova piattaforma Kubernetes [21] per garantire scalabilità, affidabilità e replicazione dei vari servizi (C*, O1, O2, O3, V3, V4)
- aggiunta del nuovo componente K8s Manager per garantire la gestione delle credenziali e l'esecuzione delle probe nelle varie modalità di Deployment Managed, Hybrid e On-premise (C4, O1, O2, V4)
- migrazione del credential store a HashiCorp Vault [15] per garantire la gestione delle credenziali in maniera sicura e scalabile (C1, O2)
- modifiche ad ogni componente in modo incrementale verificando la compatibilità dei vari formati d'interscambio (V1)
- replicazione delle istanze su più nodi, utilizzo di load balancer e replicazione dello storage con più backup ad intervalli di tempo prefissati (O2, V3)
- autenticazione di ogni servizio prima delle chiamate operative, utilizzo di policy per garantire le modalità di accesso e di utilizzo dei vari servizi ad ogni utente/servizio, in particolare possibilità di lettura dei dati sensibili solo agli utenti legittimi (V2)

Criticità	Azioni
Gestione delle credenziali basica (C1)	Migrazione del credential store a HashiCorp Vault
	Migrazione a nuova piattaforma Kubernetes
Dipendenza da un sistema pub/sub esterno (SQS) (C2)	Migrazione a nuova piattaforma Kubernetes
Eccessiva presenza di componenti custom (C3)	Migrazione a nuova piattaforma Kubernetes
Deployment in un sistema legacy (C4)	Scrittura del nuovo componente K8s Manager
	Migrazione a nuova piattaforma Kubernetes
Risultati non puntuali (C5)	Sistemazione dell'interfaccia grafica (Dashboard)
	Scrittura del nuovo componente Evidence Writer
	Migrazione a nuova piattaforma Kubernetes

Tabella 2: Azioni di migrazione per criticità

Obiettivi	Criticità	Azioni
Costruzione di un sistema modulare installabile nelle modalità managed, hybrid e on-premise (O1)	Eccessiva presenza di componenti custom (C3)	Scrittura del nuovo componente K8s Manager
	Deployment in un sistema legacy (C4)	Migrazione a nuova piattaforma Kubernetes
Garanzia di affidabilità, scalabilità e replicazione (O2)	Gestione delle credenziali basica (C1)	Scrittura del nuovo componente K8s Manager
	Dipendenza da un sistema pub/sub esterno (SQS) (C2)	Migrazione del credential store a HashiCorp Vault
	Deployment in un sistema legacy (C4)	Replicazione delle istanze su più nodi, utilizzo di load balancer e replicazione dello storage con più backup a intervalli di tempo prefissati
		Migrazione a nuova piattaforma Kubernetes
Migrazione verso un sistema cloud-native (O3)	Dipendenza da un sistema pub/sub esterno (SQS) (C2)	Migrazione a nuova piattaforma Kubernetes
	Deployment in un sistema legacy (C4)	
Risultati fine-grained (O4)	Risultati non puntuali (C5)	Sistemazione dell'interfaccia grafica (Dashboard)
		Scrittura del nuovo componente Evidence Writer

Tabella 3: Azioni di migrazione per obiettivi e criticità

Vincoli	Azioni
Interconnessione fra microservizi (V1)	Modifiche ad ogni componente in modo incrementale
Sicurezza delle informazioni (V2)	Autenticazione di ogni servizio prima delle chiamate operative
Elevate prestazioni (V3)	Migrazione a nuova piattaforma Kubernetes
	Replicazione delle istanze su più nodi, utilizzo di load balancer e replicazione dello storage con più backup ad intervalli di tempo prefissati
Manutenibilità (V4)	Migrazione a nuova piattaforma Kubernetes
	Scrittura del nuovo componente Evidence Writer
	Scrittura del nuovo componente K8s Manager

Tabella 4: Azioni di migrazione per vincoli

Capitolo 4. Implementazione

La fase di implementazione ha previsto la definizione e l'implementazione di una nuova architettura per Moon Cloud, basata su Kubernetes e microservizi. In questo capitolo verranno presentati i dettagli dell'implementazione, partendo dalla definizione dei requisiti e degli obiettivi, per poi passare alla progettazione e all'implementazione dei componenti.

4.1 Progettazione della Migrazione

La prima fase della migrazione ha previsto la progettazione dell'architettura e dei componenti necessari per la migrazione, in particolare si è partiti dall'analisi delle tipologie di architetture multi cluster disponibili e delle relative problematiche, per poi passare alla definizione delle strategie di deployment ed alla configurazione di un cluster AWS EKS.

4.1.1 Architettura Multi Cluster

La definizione dell'architettura multi cluster ha richiesto un'attenta analisi delle possibili soluzioni disponibili al momento, in particolare si sono analizzati i progetti relativi alla Kubernetes Federation [20], Rancher [28] e Liqo [23], approcci single cluster con l'utilizzo di worker remoti Konnectivity [19] e approcci ibridi tramite l'utilizzo di cluster indipendenti o kube-scheduler replicato/detached.

In particolare sono stati presi in considerazione le seguenti tecnologie:

- **Single cluster con worker remoti**: tecnologia che sfrutta un singolo nodo master per gestire più nodi worker tramite la rete, permette di gestire più nodi distribuiti come un unico cluster, permettendo di distribuire le risorse e di gestire le applicazioni in modo trasparente. Tuttavia, questa tecnologia è ancora in fase di sviluppo e non è ancora pronta per l'uso in produzione. Le tecnologie analizzate sono le seguenti:
  - **Konnectivity**: progetto che fornisce un livello di astrazione del routing tramite mapping 1:1 su stream TCP, si basa su un'architettura client/server e permette di gestire più cluster Kubernetes come un unico cluster.
- **Multicluster Federato**: tecnologia che permette di unire più cluster Kubernetes in uno unico e di gestire le applicazioni in modo trasparente, permette di scalare orizzontalmente i cluster e di garantire la disponibilità del-

le applicazioni in caso di guasti, oltre ad una minore latenza di rete. Le tecnologie analizzate sono le seguenti:

  - **Rancher**: progetto che offre funzionalità avanzate per la gestione dei cluster multipli, inclusa la federazione dei cluster. Permette di unire più cluster Kubernetes in uno unico e di gestire le applicazioni in modo trasparente. Fornisce inoltre strumenti per il monitoraggio, la scalabilità e la sicurezza dei cluster.
  - **KubeFed**: progetto che fornisce una soluzione per la federazione dei cluster Kubernetes. Offre funzionalità avanzate per il bilanciamento del carico, la scalabilità e la disponibilità delle applicazioni distribuite su cluster federati.
  - **Liqo**: progetto che permette la federazione di cluster multipli e offre funzionalità avanzate per la gestione delle risorse, la scalabilità e la sicurezza dei cluster federati.

- **Approccio ibrido**: tecnologia che prevede l'utilizzo di componenti custom (kube-scheduler) o di cluster indipendenti, il collegamento tra i cluster avviene tramite un sistema di comunicazione custom NAT-traversal. Le tecnologie analizzate sono le seguenti:

  - **Kube-scheduler replicato detached**: utilizzo di un kube-scheduler custom per la gestione di più cluster, permette di gestire le applicazioni in modo trasparente e di garantire la disponibilità delle applicazioni in caso di guasti.
  - **Cluster indipendenti**: utilizzo di cluster indipendenti per la gestione delle applicazioni, il collegamento tra i cluster avviene tramite un sistema di comunicazione custom NAT-traversal, permette di scalare orizzontalmente i cluster e di garantire la disponibilità delle applicazioni in caso di guasti.

Dopo un'attenta analisi delle caratteristiche tecniche e delle specifiche necessarie per il testing è emersa l'inadeguatezza dei sistemi single cluster con worker remoti poichè totalmente deficitari di fault tolerance e critici sotto un profilo di latenza di rete, l'inadeguatezza dei sistemi federati per il forte uso di potenza di calcolo nelle fasi di sincronizzazione e la complessità generale nella gestione. Si è quindi optato per un approccio multi cluster indipendenti con una connessione infra-cluster gestita dall'ecosistema Moon Cloud, questo permette una forte flessibilità nelle modalità di deployment e una grande scalabilità del deployment senza forti modifiche on premise.

La tabella 5 mostra un confronto tra le diverse architetture analizzate, in particolare si può notare che l'approccio a cluster singolo con worker remoti tramite Konnectivity risulta il più semplice da implementare ma anche il meno scalabile e con meno fault tolerant, l'approccio multi cluster federato risulta il più complesso da implementare ed anche più costoso in termini di risorse, mentre l'approccio ibrido risulta il più flessibile e scalabile, con un consumo di risorse proporzionato allo use case.

Sono state inoltre prese in considerazione le problematiche relative alla rete, in particolare si sono analizzate la latenza, la banda necessaria, il tipo di proxy e gli spazi di indirizzamento necessari.

Confronto approcci	**Cluster singolo + worker remoti (Konnectivity)**	**Multicluster federato (Rancher / KubeFed / Liqo)**	**Approccio ibrido (kube-scheduler replicato detached/cluster indipendenti)**
Numero di cluster	1	1(Host) + N(Member)	1
IP pubblico	Solo per master	Solo per cluster host	Solo per master (potrebbe essere necessario l'IP pubblico del cliente in caso di fault→No NAT traversal, se non implementato a parte)
Latenza	Potenzialmente elevata→Etcd critico	Bassa	Bassa in base all' implementazio-ne→Possibile replica on site dell'Etcd
Fault tolerance	Praticamente inesistente	Alta	Media→No monitoraggio
Scalabilità	Da verificare	Alta	Da verificare
Bandwidth	Basso	Potenzialmente alto nella sincronizzazione	Basso
CPU usage	Medio	Medio/Elevato	Medio
Ram usage	Basso	Medio→supponendo MicroK8s(circa 1GB)	Medio
Proxy type	gRPC(mapping 1:1 su stream TCP)	??	??

Namespace	Solo condivisi→possibilità di policy solo su namespace	Possibile condivisi o separati→possibilità di policy per ogni cluster	Solo condivisi→possibilità di policy solo su namespace
IP duplicati	Possibili→viene garantito il routing trasparente dei pacchetti fino all'host finale, poi invio su rete locale(e valutazione dell'IP)		
Crittografia	Sì	Sì	Da valutare→se comunicazione solo inter-cluster non gestita(senza entità intermedie di crittografia) non essere garantita
Costo	Basso	Medio/Elevato	Medio

Note aggiuntive	Implementazione semplice Poco supportato Poca documentazione	Rancher→molto utilizzato e con grande documentazione KubeFed→ attualmente deprecato(da pochi mesi anche la v2) Liqo→supporto completo al multi cluster introdotto da circa un anno, possibile instabilità Implementazione potenzialmente complessa	Non esiste uno standard de facto→tecnologia da implementare tramite forzature, sistemi custom o cluster indipendenti Master replicato solo parzialmente/master indipendenti: •No difficoltà derivate dal multicluster •No monitoraggio/observability •Possibile instabilità e complessità nella gestione

Tabella 5: Comparazione architetture cluster

La tabella 6 mostra alcuni test effettuati sui vari tipi di infrastruttura e i requisiti necessari, i parametri presi in considerazione sono il numero di istanze e macchine necessarie per il testing, gli spazi di indirizzamento, la banda necessaria e le risorse hardware necessarie (CPU e RAM).

Macchine necessarie per il testing	**Cluster singolo + worker remoti (Konnectivity)**	**Multicluster federato (Rancher / KubeFed / Liqo)**	**Approccio ibrido (kube-scheduler replicato detached / cluster indipendenti)**	
Istanze / Macchine	1 cluster con Z nodi worker LAB+N nodi remoti	1 cluster con Z nodi worker LAB+N cluster remoti con 1 master e 1 worker	1 cluster con Z nodi worker LAB+N worker remoti+N kube scheduler detached / scheduler custom	Se cluster indipendenti uguale a multicluster
Spazi di indirizzamento	Z nodi in LAB nello steso spazio di indirizzamento N nodi di remoti con spazi di indirizzamento misti: 3→su stesso spazio nodi LAB N-3→su spazi diversi			
Bandwith	Verificare se esistono limiti di traffico ingoing / outgoing su firewall o simili			
CPU	Core cluster LAB+X1 core*N nodi	Core cluster LAB+X2 core*N cluster	Core cluster LAB+X1 core*N nodi	
RAM	RAM cluster LAB+Y1 GB*N nodi	RAM cluster LAB+Y2 GB*N cluster	RAM cluster LAB+Y1 GB*N nodi	

Tabella 6: Requisiti comparazione architetture cluster

Note:

- Supponendo Z nodi worker LAB=2→CPU cluster LAB=16 core, RAM cluster LAB=16 GB
- Supponendo un ambiente di test con uno scenario di N clienti, con N ragionevolmente elevato, ipotizzabile uguale a 20/30
- Supponendo CPU/RAM con dei valori bassi per il testing e più elevati in ottica di produzione si propongono queste stime:

CPU	RAM
X1→2 core (single cluster per testing)	Y1→2 GB (single cluster per testing)
X2→4 core (multi cluster per testing)	Y2→4 GB (multi cluster per testing)
X3→4 core (single cluster per produzione)	Y3→4 GB (single cluster per produzione)
X4 →8 core (multi cluster per produzione)	Y4→8 GB(multi cluster per produzione)

Tabella 7: Requisiti CPU/RAM architetture cluster

Supponendo N=20 come scenario abbastanza affidabile per il testing si può stimare un consumo di risorse che nel caso di un cluster singolo con worker remoti risulta essere il più basso, nel caso di un multi cluster federato risulta essere il più alto, mentre l'approccio ibrido si colloca in una posizione intermedia a seconda della modalità di deployment prescelta.

<table>
<tr><th>Macchine necessarie per il testing</th><th>Cluster singolo + worker remoti (Konnectivity)</th><th>Multicluster federato (Rancher / KubeFed / Liqo)</th><th colspan="2">Approccio ibrido (kube-scheduler replicato detached / cluster indipendenti)</th></tr>
<tr><td>Istanze / Macchine</td><td>1 cluster con 1 master e 2 worker (LAB)+20 nodi remoti</td><td>1 cluster con 1 master e 2 worker (LAB)+20 cluster remoti con 1 master e 1 worker</td><td>1 cluster con 1 master e 2 worker (LAB)+20 nodi remoti+20 kube scheduler detached / scheduler custom</td><td rowspan="5">Se cluster indipendenti uguale a multicluster</td></tr>
<tr><td>Spazi di indirizzamento</td><td colspan="3">2 worker LAB nello stesso spazio di indirizzamento
20 worker(per multicluster anche 20 master) remoti con spazi di indirizzamento misti:
3→su stesso spazio nodi LAB
17→su spazi diversi</td></tr>
<tr><td>Bandwith</td><td colspan="3">Verificare se esistono limiti di traffico ingoing / outgoing su firewall o simili</td></tr>
<tr><td>CPU</td><td>16 core cluster LAB+40 core per i nodi</td><td>16 core cluster LAB+80 core per i cluster</td><td>16 core cluster LAB+40 core per i nodi</td></tr>
<tr><td>RAM</td><td>16 GB cluster LAB+40 GB per I nodi</td><td>16 GB cluster LAB +80 GB per i cluster</td><td>16 GB cluster LAB+40 core per i nodi</td></tr>
</table>

Tabella 8: Requisiti comparazione architetture cluster con N=20

Supponendo N=30 come scenario in ottica di scaling, si può notare una crescita proporzionata delle risorse necessarie per il testing, in particolare si può notare che l'approccio a cluster singolo con worker remoti risulta il più scalabile, l'approccio multi cluster federato risulta il meno scalabile, mentre l'approccio ibrido risulta il più scalabile e flessibile.

Macchine necessarie per il testing	**Cluster singolo + worker remoti (Konnectivity)**	**Multicluster federato (Rancher / KubeFed / Liqo)**	**Approccio ibrido (kube-scheduler replicato detached / cluster indipendenti)**	
Istanze / Macchine	1 cluster con 1 master e 2 worker (LAB)+30 nodi remoti	1 cluster con 1 master e 2 worker (LAB)+30 cluster remoti con 1 master e 1 worker	1 cluster con 1 master e 2 worker (LAB)+30 nodi remoti+30 kube scheduler detached / scheduler custom	Se cluster indipendenti uguale a multicluster
Spazi di indirizzamento	2 worker LAB nello stesso spazio di indirizzamento 30 worker(per multicluster anche 30 master) remoti con spazi di indirizzamento misti: 3→su stesso spazio nodi LAB 27→su spazi diversi			
Bandwith	Verificare se esistono limiti di traffico ingoing / outgoing su firewall o simili			
CPU	16 core cluster LAB+60 core per i nodi	16 core cluster LAB+120 core per i cluster	16 core cluster LAB+60 core per i nodi	
RAM	16 GB cluster LAB+60 GB per i nodi	16 GB cluster LAB+120 GB per i cluster	16 GB cluster LAB+60 core per i nodi	

Tabella 9: Requisiti comparazione architetture cluster con N=30

Il nuovo approccio multi cluster prevede che quest ultimi possano dialogare tra loro grazie a NATS che fornisce una comunicazione NAT-traversal senza ulteriori configurazioni lato client.

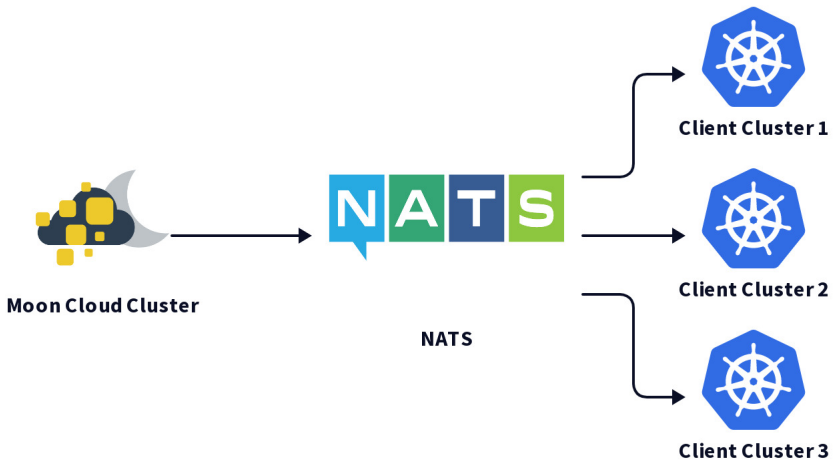

Figura 3: Architettura multi cluster

4.1.2 Strategie di Deployment

La nuova architettura offre un'ampia flessibilità in fase di deployment, in particolare è possibile scegliere tra tre diverse modalità di deployment che prevedono un diverso grado di controllo e gestione dei dati. Nel dettaglio le modalità di deployment sono le seguenti:

Full Managed: modalità di Deployment che prevede il completo affidamento delle valutazioni all'ecosistema Moon Cloud, che quindi andrà a gestire sia la parte di Evaluation Center che la Customer Zone.
Vantaggi:

- Semplice da installare: nessuna necessità di installazioni on-premise, in quanto tutto il sistema viene gestito da Moon Cloud che si occuperà di installare e configurare i componenti necessari senza alcun intervento da parte del cliente on-premise
- Basso effort dell'utente finale: l'utente finale non dovrà preoccuparsi di installare e configurare i componenti necessari, dovrà solo preoccuparsi delle configurazioni delle valutazioni
- Gestione dei dati centralizzata e replicata: Moon Cloud si occuperà di gestire i dati in modo centralizzato e replicato, garantendo la disponibilità e la consistenza dei dati

Svantaggi:

- Possibilità di eseguire verifiche solo su endpoint esposti in rete: non è possibile eseguire verifiche su endpoint interni in quanto non è possibile installare agent di monitoraggio che si possono collegare ad endpoint interni
- Minore granularità di valutazione dell'asset management: non è possibile avere una gestione completa dell'asset management in quanto non è possibile eseguire verifiche su endpoint interni

Customer Managed On Premise: modalità di Deployment che prevede l'installazione di un cluster Moon Cloud nell'ambiente on premise del cliente, per cui Moon Cloud gestirà la parte di Evaluation Center e il cliente la Customer Zone.
Vantaggi:

- Possibilità di eseguire verifiche su endpoint interni e esposti in rete: è possibile eseguire verifiche su endpoint interni e esposti in rete in quanto è possibile installare agent di monitoraggio che si possono collegare ad endpoint interni

- Ampia granularità di valutazione dell'asset management: è possibile avere una gestione completa dell'asset management in quanto è possibile eseguire verifiche su endpoint interni
- Gestione dei dati distribuita e replicata: Moon Cloud si occuperà di gestire i dati in modo distribuito e replicato, garantendo la disponibilità e la consistenza dei dati

Svantaggi:

- Installazione più complessa: il cliente dovrà installare e configurare i componenti necessari per far funzionare il cluster Moon Cloud nell'ambiente on-premise, questo richiede un maggiore sforzo da parte del cliente rispetto alla modalità Full Managed in quanto potrebbero essere necessarie modifiche alla rete e all'infrastruttura esistente

Customer On Premise: modalità di Deployment simile alla Customer Managed On Premise, che prevede l'utilizzo di un cluster già in essere di proprietà del cliente, la divisone è la stessa della precedente.
Vantaggi:

- Possibilità di eseguire verifiche su endpoint intra cluster, interni e esposti in rete: è possibile eseguire verifiche su endpoint intra cluster, interni e esposti in rete in quanto è possibile installare agent di monitoraggio che si possono collegare ad endpoint interni
- Ampia granularità di valutazione dell'asset management: è possibile avere una gestione completa dell'asset management in quanto è possibile eseguire verifiche su endpoint interni, con l'ulteriore aggiunta di verifiche intra cluster
- Gestione dei dati distribuita e replicata: Moon Cloud si occuperà di gestire i dati in modo distribuito e replicato, garantendo la disponibilità e la consistenza dei dati

Svantaggi:

- Installazione più complessa: il cliente dovrà installare e configurare i componenti necessari per far funzionare il cluster Moon Cloud nell'ambiente on-premise, questo richiede un maggiore sforzo da parte del cliente rispetto alla modalità Full Managed in quanto potrebbero essere necessarie modifiche alla rete e all'infrastruttura esistente

4.1.3 Simulazione di un Cluster Client

Come testbed per la simulazione di un cliente, si è scelto di configurare un cluster EKS [5] in ambiente AWS, questo si compone di tutti i servizi della Customer Zone e permette una completa simulazione delle valutazioni anche in ambienti remoti.

4.2 Migrazione a Kubernetes

La seconda fase della migrazione ha previsto la migrazione dei componenti dell'ecosistema Moon Cloud su Kubernetes, in particolare si è partiti dalla conversione delle recipe, per poi passare alla configurazione dei Deployment e dei Service ed infine alla configurazione degli Ingress con collegamento ad un sistema di gestione dei certificati automatizzato (Cert Manager).

4.2.1 Conversione delle Recipe

Tutte le recipe dei componenti sono state convertite in YAML Kubernetes o Helm Chart [17] ove disponibili. Ogni componente prevede solitamente le seguenti entità Kubernetes:

- *Deployment/Statefull Set*: specifica delle configurazioni e dei pod con cui verrà eseguito il componente
- *Service*: specifica delle modalità di esposizione del deployment/statefull set e load balancer del relativo traffico su tutte le istanze sottostanti
- *PVC*: specifica della richiesta di storage per il componente a cui seguirà l'assegnazione di un PV di egual dimensione
- *Ingress*: mapping del nome DNS sul componente e annessa esposizione

Ogni Deployment successivamente andrà a generare i relativi replica set e pod per l'avvio del componente secondo specifica.

4.2.2 Deployment dei Componenti

Ogni componente dell'ecosistema Moon Cloud è stato riscritto sotto forma di template YAML sia per la fase di Deployment che per l'esposizione agli altri servizi, ovvero la configurazione dei Service. La figura 4 mostra la configurazione YAML del Deployment per il componente Dashboard, mentre la figura 5 mostra la configurazione YAML del Service relativo. Si possono notare le specifiche di configurazione del pod, delle risorse richieste, delle variabili d'ambiente e delle configurazioni di rete. In particolare si può notare che il Deployment specifica una policy di aggiornamento dell'immagine settata ad Always, ciò permette un aggiornamento continuo del componente in caso di nuove versioni e un imagePullSecret per l'accesso al registry privato. Il Service specifica inoltre una modalità di esportazione del servizio tramite ClusterIP e l'esposizione della porta 80 del pod agli altri servizi.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    mooncloud: dashboard
  name: dashboard
spec:
  replicas: 1
  selector:
    matchLabels:
      mooncloud: dashboard
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        mooncloud: dashboard
    spec:
      containers:
        - image: registry.v2.moon-cloud.eu/mooncloudnewdashboard/newdashboard2/
          ↪ experimental-fix:0.0.15
          imagePullPolicy: Always
          name: dashboard
          ports:
            - containerPort: 80
          resources: {}
      imagePullSecrets:
        - name: mooncloud-registry-secret
      restartPolicy: Always
status: {}
```

Figura 4: Dashboard deployment YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    mooncloud: dashboard
  name: dashboard
spec:
  type: ClusterIP
  ports:
    - name: "80"
      port: 80
      targetPort: 80
  selector:
    mooncloud: dashboard
status:
  loadBalancer: {}
```

Figura 5: Dashboard service YAML

4.2.3 Deployment dei Servizi Esterni

Per i componenti esterni con configurazioni modulari già presenti si è scelta la configurazione tramite Helm Chart, seguono alcuni esempi:

```yaml
nats:
  config:
    jetstream:
      enabled: true

    nats:
      tls:
        enabled: true
        secretName:
        ↪ nats-server-tls-secret
        merge:
          timeout: "2s"

    ...

  service:
    merge:
      spec:
        type: NodePort
        ports:
          - name: nats
            targetPort: nats
            protocol: TCP
            appProtocol: tls
            port: 4222
            nodePort: 30222
          - name: nats-prometheus
            targetPort: 7777
            port: 7777
  reloader:
    enabled: true

  promExporter:
    enabled: true
```

(a) NATS values YAML

```yaml
vault:
  injector:
    enabled: true
  server:
    enabled: true
    extraContainers:
      - name: vault-auto-unseal-1
        image: "ghcr.io/lrstanley/vault
        ↪ -unseal:latest"
        env:
          - name: ENVIRONMENT
            value: "mooncloud"
          - name: ALLOW_SINGLE_NODE
            value: "true"
          ...
          - name: TOKENS
              valueFrom:
                  secretKeyRef:
                    name: vault-share-1
                    key: share
      - name: vault-auto-unseal-2
        image: "ghcr.io/lrstanley/vault
        ↪ -unseal:latest"
        env:
          - name: ENVIRONMENT
            value: "mooncloud"
          - name: ALLOW_SINGLE_NODE
            value: "true"
          ...
          - name: TOKENS
            valueFrom:
                secretKeyRef:
                  name: vault-share-2
                  key: share
```

(b) Vault values YAML

4.2.4 Politiche di Accesso e Routing

Per i componenti che necessitavano di essere esposti tramite nome DNS sono state configurate le relative entry Ingress. Ciò avviene tramite la configurazione di un reverse proxy Nginx che provvede inoltre alla richiesta dei certificati SSL necessari.

```yaml
proxies:
  - port: 8000
    host: api.v2.moon-cloud.eu
    name: api
  - port: 80
    host: dashboard.moon-cloud.eu
    name: dashboard
  - externalName: 172.20.28.181
    port: 8080
    host: doc.v2.moon-cloud.eu
    name: proxy-doc-v2-moon-cloud
  - port: 80
    host: moon-cloud.eu
    name: presentation-frontend
  - port: 80
    host: www.moon-cloud.eu
    name: presentation-frontend
  - externalName: 172.20.28.149
    port: 5009
    host: registry.v2.moon-cloud.eu
    name: registry-v2-moon-cloud-eu
  - externalName: 172.20.28.149
    port: 8081
    host: repository.v2.moon-cloud.eu
    name: repository-v2-moon-cloud-eu
```

Figura 7: Ingress values YAML

4.2.5 Gestione dei Certificati

Si è scelto Cert Manager [8] come gestore interno dei certificati SSL, esso provvede alla richiesta e rinnovo automatizzato dei certificati SSL per gli endpoint esposti. Esso è stato configurato come Cluster Issuer, per cui ad ogni nuova esposizione di un Ingress provvederà a rimandare tutte le richieste ad un server ACME, nel nostro caso Let's encrypt [22].

4.3 Componenti

L'ultima fase della migrazione ha previsto la definizione e l'implementazione di nuovi componenti e la riscrittura di alcuni dei precedenti. In questo capitolo verranno presentati i dettagli dei nuovi componenti, partendo dalla definizione dei requisiti e degli obiettivi, per poi passare alla progettazione e all'implementazione.

4.3.1 Dashboard

Il componente Dashboard fornisce un'interfaccia grafica che permette l'interazione dell'utente finale con l'ecosistema Moon Cloud. Le modifiche effettuate sono le seguenti:

Nuovo formato risultati conversione dei modelli Angular [1] della Dashboard per lettura corretta dei nuovi risultati prodotti dall'Evidence Writer. La figura 8 mostra la nuova interfaccia per la visualizzazione dei risultati delle UER. Si può notare la suddisione dei risultati per target e per controllo, con la possibilità di visualizzare i dettagli di ogni singolo controllo ordinati temporalmente.

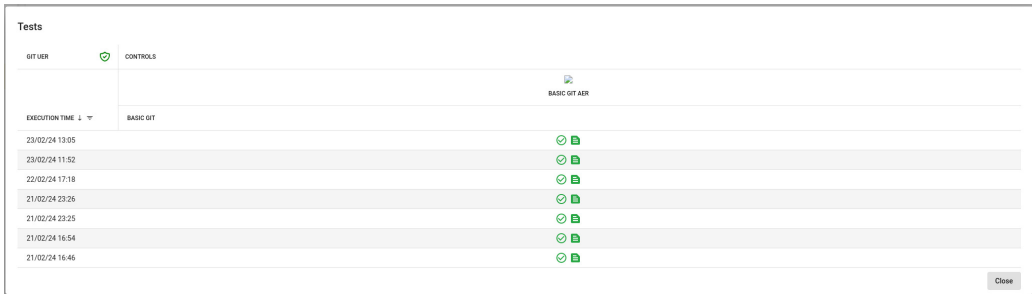

Figura 8: Dashboard UER

Zone delle credenziali aggiunta della visualizzazione delle zone relative alla nuova modalità di identificazione dell'esecutore responsabile della gestione delle credenziali. La figura 9 mostra la nuova interfaccia per la visualizzazione delle credenziali. Si può notare la suddivisione delle credenziali per zone e per esecutore, con la possibilità di visualizzare i dettagli di ogni singola credenziale.

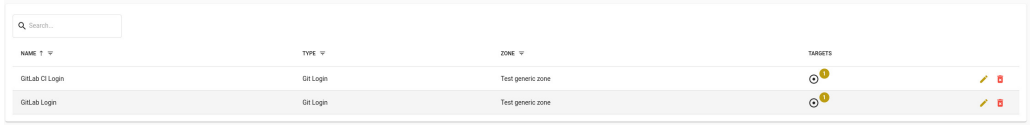

Figura 9: Dashboard Credenziali con Zone

Tipi generici di target aggiunta della possibilità di creazione e filtraggio per il nuovo tipo di target generico. Sono state aggiunte la possibilità di avere zone generiche con differenti tipi di target e target generici che supportano tutte le tipologie di controlli indipendentemente dal tipo del target.

Interfaccia di aggiunta UER sistemazione del form di creazione delle UER per permettere una corretta creazione di UER con più controlli, anche in caso di chiavi di configurazione duplicate. Sono inoltre stati aggiunti dei template custom per la visualizzazione più agevole dei campi di configurazione. La figura 10 mostra la nuova interfaccia per la creazione delle UER. Si può notare la possibilità di aggiungere più UER, la possibilità di avere UER con più controlli e la visualizzazione delle singole configurazioni.

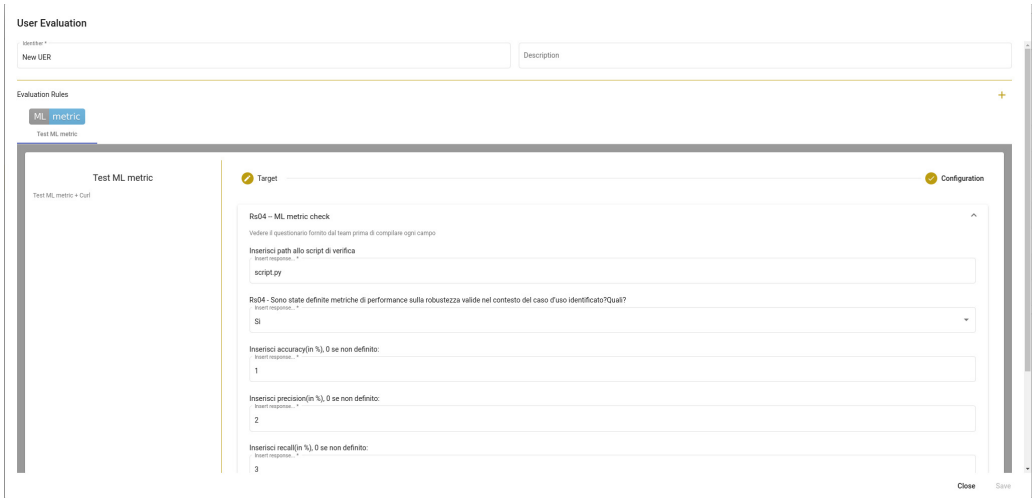

Figura 10: Dashboard Creazione UER

4.3.2 API

Il componente API fornisce un'interfaccia per la gestione completa di tutte le entità dell'ecosistema Moon Cloud. Le modifiche effettuate sono le seguenti:

Rimozione vecchi riferimenti rimozione dei riferimenti relativi all'utilizzo di code SQS e Director per conversione al nuovo sistema tramite NATS [25] e K8s Manager e rimozione dei riferimenti non più utili relativi a Influx [18] e Redis [29] in favore dell'uso diretto dei risultati scritti dall'Evidence Writer su Postgres [27].

NATS e K8s Manager aggiunta della connessione tramite NATS [25], conversione del formato di interscambio al nuovo formato di invio task e credenziali al K8s Manager. La figura 11 mostra la nuova interfaccia di connessione tra API e K8s Manager attraverso NATS, si può notare la gestione delle richieste di task e credenziali e la creazione e utilizzo di una coda di default in caso non ne sia specificata una più specifica.

```python
class K8sManager():
    nc:any=None

    ...

    @staticmethod
    async def nats_connect():
        if K8sManager.nc is None:
            logger.debug("Connecting to
            ↪  tls://"+settings.NATS_HOST+":"+settings.NATS_PORT)
        K8sManager.nc=await nats.connect("tls://"+settings.NATS_USER+":"+settin
        ↪  gs.NATS_PASSWORD+"@"+settings.NATS_HOST+":"+settings.NATS_PORT)

    @staticmethod
    def get_related_subject(zone_id,project_id,user_id):
        K8sManager.create_nats_default_subject()
        return NatsSubject.objects.annotate(
                priority=Case(
                    When(zone_id=zone_id, project_id=project_id,
                    ↪  user_id=user_id, then=Value(1)),
                    When(zone_id=None, project_id=project_id, user_id=user_id,
                    ↪  then=Value(2)),
                    When(zone_id=None, project_id=None, user_id=user_id,
                    ↪  then=Value(3)),
                    default=Value(4),   # this will implicity be the default
                    ↪  subject
                    output_field=models.IntegerField()
                )
            ).order_by('priority').first()

    @staticmethod
    async def genericHandleNatsReq(subject:str,body:any,log_msg:str,user_id:int
    ↪  ,project_id:int,zone_id:int):
        response=None
        try:
            await K8sManager.nats_connect()
            nats_sub=K8sManager.get_related_subject(zone_id=zone_id,project_id=
            ↪  project_id,user_id=user_id)
            sub_prefix="agent."+nats_sub.subject_name+"."
            logger.debug("Sending message on subject "+sub_prefix+subject)
            response = await K8sManager.nc.request(sub_prefix+subject,
            ↪  json.dumps(body).encode(), timeout=0.5)
            logger.debug("Received response: {message}".format(
            ↪  message=response.data.decode()))
        except TimeoutError:
            logger.debug("Request timed out")
        K8sManager.check_response_or_raise(response=response,log_msg=log_msg)
```

Figura 11: API K8s Manager

Ripristino SMTP correzione degli errori di configurazione del sistema SMTP per il ripristino del corretto invio delle mail nelle fasi di login/registrazione.

4.3.3 NATS

Il componente NATS è stato aggiunto all'ecosistema Moon Cloud, in sostituzione del servizio SQS offerto da AWS, per fornire un sistema di messaggistica ad alte prestazioni, bassa latenza e affidabilità, che permette la comunicazione tra i vari microservizi dell'ecosistema Moon Cloud. Le modifiche effettuate sono le seguenti:

Autenticazione e configurazione dei permessi aggiunta tramite file server.conf degli utenti relativi ai servizi e configurazione delle autorizzazioni di ognuno per il dialogo su code NATS. In particolare:

- all'API viene consentita la sola pubblicazione sulla coda "agent.⟩", per cui può pubblicare su tutte le code dei K8s Manager
- l'Evidence Writer invece può sottoscriversi solo a "results.⟩" per ricevere i risultati dalle Probe, è necessaria un'ulteriore sottoscrizione a "_INBOX.⟩" per gestire le risposte interne di JetStream [26]
- I due K8s Manager possono sottoscriversi solo alla relativa coda con tutti gli endpoint sottostanti("agent.k8s-manager-mooncloud.⟩"/"agent.k8s-manager-aws.⟩")
- Le probe possono pubblicare solo sulla coda col relativo riferimento al K8s Manager che l'ha avviata per tutte le possibili UER a cui è collegata("results.agent.k8s-manager-mooncloud.uer.*"/ "results.agent.k8s-manager-aws.uer.*")

```
users:
  - user: "api"
    password: REDACTED
    permissions:
      publish: "agent.>"

  - user: "ew"
    password: REDACTED
    permissions:
      subscribe:
        - "results.>"
        - "_INBOX.>"

  - user: "k8s-manager-mooncloud"
    password: REDACTED
    permissions:
      subscribe: "agent.k8s-manager-mooncloud.>"

  - user: "probe-mooncloud"
    password: REDACTED
    permissions:
      publish: "results.agent.k8s-manager-mooncloud.uer.*"

  - user: "k8s-manager-aws"
    password: REDACTED
    permissions:
      subscribe: "agent.k8s-manager-aws.>"

  - user: "probe-aws"
    password: REDACTED
    permissions:
      publish: "results.agent.k8s-manager-aws.uer.*"
```

Figura 12: Configurazione NATS

Definizione di un'algoritmo per la distribuzione delle UER agli Evidence Writer L'Evidence attualmente è in grado di gestire solo una UER dall'inizio della sua elaborazione alla fine, ovvero non possono intervenire eventuali repliche durante una valutazione, per ovviare a questo problema si è definito il seguente pattern:

- L'Evidence Writer viene deployato come statefull set in modo tale da garantire un indice ordinale dell'istanza anche dopo eventuali riavvii/crash
- Questo indice viene esportato nella variabile d'ambiente EW_INDEX che verrà poi letta dall'Evidence Writer in fase di avvio
- Nella configurazione di NATS viene settato il numero di mapping della coda uguale al numero di repliche

In seguito a questa prima fase di configurazione si può procedere con l'algoritmo effettivo:

- La probe legge lo UER ID dalla configurazione dalla sua configurazione corrente
- La probe invia la nuova test execution sulla coda "uer.⟨UER ID⟩"
- Il server NATS mappa la coda "uer.⟨UER ID⟩" sulla coda "evidence_writer.⟨EW ID⟩.uer.⟨UER ID⟩"
- L'Evidence Writer legge il risultato dalla coda "evidence_writer.⟨EW ID⟩.uer.⟨UER ID⟩"

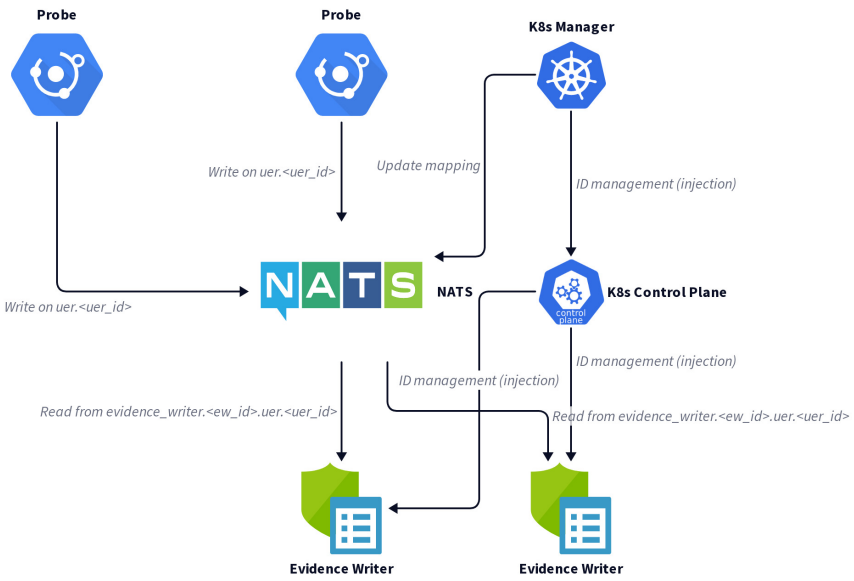

Figura 13: Architettura NATS

Il mapping è ottenuto tramite NATS DSTP(deterministic subject token partioning), la formula utilizzata è la seguente:

$$"uer.*" -> "evidence\_writer.partition(<n>,1).uer.wildcard(1)"$$

L'Evidence Writer provvederà ad aggiornare il parametro n ogni qual volta scala

La tabella 10 mostra un esempio di funzionamento del sistema di mapping, si può notare che sebbene il load balancing non sia perfetto, il sistema è in grado di distribuire in modo equo le UER tra le varie repliche dell'Evidence Writer, tuttavia è possibile adottare un sistema di UUID per ottenere un load balancing più preciso. Di seguito sono riportati i vantaggi e gli svantaggi di questo sistema.

Input subject	Output subject (n=3)
uer.1	evidence_writer.1.uer.1
uer.2	evidence_writer.1.uer.2
uer.3	evidence_writer.2.uer.3
uer.4	evidence_writer.1.uer.4
uer.5	evidence_writer.2.uer.5
uer.6	evidence_writer.2.uer.6
uer.7	evidence_writer.0.uer.7

Tabella 10: Mapping NATS

Vantaggi:

- Separazione dei microservizi(deaccoppiamento dall'API): l'API non deve conoscere quali UER sono associate a quali Evidence Writer e viceversa
- Nessuna necessità di conoscere l'IP/ID dell'Evidence Writer a priori: non è necessario conoscere l'indirizzo IP/l'identificatore univoco dell'Evidence Writer per inviare i risultati
- L'Evidence Writer non deve conoscere quali UER sono associate a lui: l'Evidence Writer non deve conoscere quali UER sono associate a lui, ma solo la coda a cui sottoscriversi, che è univocamente determinata dal suo ID
- Facilmente scalabile: è possibile aggiungere/rimuovere repliche senza dover riconfigurare l'API
- Molti livelli di dettaglio possibili: è possibile aggiungere ulteriori dettagli al mapping, ad esempio dividendo le UER in base all'utente, al test, ecc.
- Possibilità di dividere il workload di un singolo utente su più Evidence Writer: è possibile ridurre il carico di lavoro di un singolo Evidence Writer dividendo le UER di un singolo utente su più repliche
- ID persistente e sempre legato al numero di repliche: l'ID dell'Evidence Writer viene determinato e garantito da Kubernetes, quindi è sempre univoco e legato al suo numero di replica
- Logica di generazione degli ID trasparente all'Evidence Writer: l'Evidence Writer non deve conoscere la logica di generazione degli ID, ma solo il suo ID, che gli verrà fornito tramite una variabile d'ambiente

Svantaggi:

- Complessità di gestione del sistema di mapping: è necessario sincronizzare il numero di repliche col numero di mapping, in modo da garantire che ogni replica abbia un mapping univoco
- Load balancing non perfetto (se possibile adottare UUID): il sistema di mapping non garantisce un load balancing perfetto, in quanto gli ID delle UER sono numeri interi e non UUID, quindi, data la bassa entropia generata in fase di hash, è possibile che alcune repliche abbiano più UER da gestire rispetto ad altre

Rispetto ad una soluzione basata su un modello a *lock condiviso*, ovvero un sistema in cui ogni transazione deve attendere il completamento di tutte le transazioni precedenti, il sistema di mapping risulta molto più efficiente in quanto presenta i seguenti vantaggi:

- Nessun lock necessario, l'istanza può procedere sempre senza attendere nessuno
- Suppponendo una generica UER⟨x⟩_⟨n⟩avremmo un vantaggio di:

$$\left(\sum_{i=1}^{<totalUER>-1}\ \sum_{j=1}^{<totalcontrols>} i\right)/ < instances >$$

  Per cui supponendo:

  - 100 utenti(⟨k⟩)
  - 15 UER per utente(⟨x⟩)
  - 4 controlli per UER(⟨total controls⟩)
  - 3 istanze di Evidence Writer(⟨instances⟩)
  - avremo $(\langle k \rangle) * (\langle x \rangle) = 100 * 15 = 1500$ UER
  - per cui salveremo $(\sum_{i=1}^{1500-1} \sum_{j=1}^{4} i)/3 = 4497000/3 = 1499000$ attese sulle operazioni di lock
  - supponendo un lock time uguale a 5 ms salveremo $1499000 * 0.005 = 7495s = 2.08h$

Nota: questo è il caso peggiore possibile in cui ogni transazione deve attendere $\langle n \rangle - 1 + \langle n\_transazioni\_in\_coda \rangle$ prima di essere eseguito. Un caso più realistico potrebbe essere assumere la metà delle attese(alcune operazioni potrebbero es-

sere completate prima dell'arrivo di quelle nuove) e la metà del lock time(per la stessa ragione), per cui otterremo un risparmio di 0.52h ovvero 31m.

4.3.4 K8s Manager

Il componente K8s Manager si occupa della gestione delle configurazioni necessarie per l'esecuzione delle probe e del loro deployment. Le modifiche effettuate sono le seguenti:

Injection delle variabili Probe aggiunta delle configurazioni sulla UER corrente, modalità di esecuzione e endpoint necessari tramite injection delle relative variabili d'ambiente nel template YAML. La figura 14 mostra l'injection delle variabili d'ambiente che verranno poi utilizzate dalle probe, in particolare si può notare il settaggio della modalità di produzione, dell'input e della configurazione dell'UER corrente coi relativi riferimenti.

```
Spec: v1.PodSpec{
        ServiceAccountName: service_account_name,
        Containers: []v1.Container{
                {
                Name:  request.Name,
                Image: request.Image,
                // Command: strings.Split(request.Cmd, " "),
                Args: request.Args,
                Env: []v1.EnvVar{
                                {
                                        Name:  "input",
                                        Value: string(input),
                                },
                                {
                                        Name:  "MOONCLOUD_PROBE_IS_PRODUCTION",
                                        Value: "True",
                                },
                                {
                                        Name:
                                        ↪  "MOONCLOUD_PROBE_READ_FROM_STDIN",
                                        Value: "False",
                                },
                                {
                                        Name:  "MOONCLOUD_PROBE_UER_ID",
                                        Value: fmt.Sprint(request.requestAPITas⌋
                                        ↪  kCommon.UserEvaluationRuleID),
                                },
                                {
                                        Name:  "MOONCLOUD_PROBE_AER_ID",
                                        Value: fmt.Sprint(request.requestAPITas⌋
                                        ↪  kCommon.AbstractEvaluationRuleID),
                                },
                                {
                                        Name:  "MOONCLOUD_PROBE_TEST_ID",
                                        Value: fmt.Sprint(request.requestAPITas⌋
                                        ↪  kCommon.TestID),
                                },
                                {
                                        Name:  "MOONCLOUD_PROBE_CONTROL_ID",
                                        Value: fmt.Sprint(request.requestAPITas⌋
                                        ↪  kCommon.ControlID),
                                },
                                        ...
                        },
                },
        },
}
```

Figura 14: K8s Manager Injection Variabili d'ambiente

Lettura tramite NATS aggiunta delle chiamate di connessione e sottoscrizione alle relative code NATS [25] per la lettura dei task e delle credenziali. La figura 15 mostra la sottoscrizione alle code NATS per la lettura delle richieste di task e credenziali, in particolare si può notare l'autenticazione al server NATS e la sottoscrizione tramite subject specifici relativi alla configurazione assegnata al singolo esecutore.

```
func startNatsSubscribe() {
        // Connect to a server
        natsURL := "tls://" + EnvConfig["NATS_HOST"] + ":" +
        ↪  EnvConfig["NATS_PORT"]
        apiLogger.Info("Connecting to " + natsURL)
        natsConn, err := nats.Connect(natsURL,
        ↪  nats.UserInfo(EnvConfig["NATS_USER"], EnvConfig["NATS_PASSWORD"]))
        if err != nil {
                apiLogger.Error(err, "NATS connection error")
                return
        }
        sub_prefix := "agent." + EnvConfig["NATS_SUBJECT"] + "."
        subjects := map[string]func(*nats.Msg){
                // TASKS AND ENTRIES
                sub_prefix + "tasks.add":    apiAddTasks,
                sub_prefix + "tasks.pause":  apiPauseTasks,
                sub_prefix + "tasks.resume": apiResumeTasks,
                //sub_prefix + "tasks.stop":
                ↪  apiStopTasks,                // NOT IMPLEMENTED
                //sub_prefix + "tasks.update":
                ↪  apiUpdateTasks,                // NOT IMPLEMENTED
                sub_prefix + "tasks.delete":   apiDeleteTasks,
                sub_prefix + "tasks.list":     apiListTasks,
                sub_prefix + "entries.add":    apiAddEntries,
                ...
        }
        // Subscribe to subjects
        for sub := range subjects {
                apiLogger.Info("Subscribing to " + sub + "...")
                natsConn.Subscribe(sub, subjects[sub])
        }
}
```

Figura 15: K8s Manager NATS

Gestione credenziali aggiunta della gestione del lifecycle delle credenziali tramite chiamate all'API del HashiCorp Vault [15]. La figura 16 mostra la gestione delle credenziali tramite chiamate all'API del Vault, in particolare si può notare la creazione delle credenziali e la suddivisione logica dei path assegnati agli utenti.

```
1  func controllerAddCredential(request *requestCredentialAdd) (*responseInfo,
   ↪  error) {
2          err := checkPath(fmt.Sprint(request.UserID))
3
4          if err != nil {
5                  vaultLogger.Error(err, "Error during path check")
6                  return nil, err
7          }
8
9          vaultLogger.Info("Adding credential " + fmt.Sprint(request.ID) + "
           ↪  (user " + fmt.Sprint(request.UserID) + ")")
10
11         _, err = vaultClient.Secrets.KvV2Write(context.Background(),
           ↪  "cred_"+fmt.Sprint(request.ID), schema.KvV2WriteRequest{
12                 Data: request.Payload,
13         }, vault.WithMountPath("mooncloud_secret/user-"+fmt.Sprint(request.User
           ↪  ID)))
14
15         if err != nil {
16                 vaultLogger.Error(err, "Error during credential creation")
17                 return nil, err
18         }
19
20         vaultLogger.Info("Credential " + fmt.Sprint(request.ID) + " (user " +
           ↪  fmt.Sprint(request.UserID) + ") added successfully")
21         return nil, nil
22 }
23
24 ...
25
```

Figura 16: K8s Manager Gestione Credenziali

Annotazioni Sidecar Vault aggiunta delle annotazioni delle credenziali necessarie per gestire l'injection delle credenziali nella probe che ne necessita tramite sidecar. La figura 17 mostra l'annotazione delle credenziali necessarie per l'injection tramite sidecar, in particolare si può notare la definizione dell'utilizzo del sidecar, la richiesta di injection, il ruolo assegnato al sidecar corrispondente al user ID dell'utente e la definizione del path e del template di injection.

```
if request.requestAPITaskCommon.CredentialID != nil {
                credentials_annotations = map[string]string{
                        "vault.hashicorp.com/agent-pre-populate-only":
                        ↪  "true",
                        "vault.hashicorp.com/agent-inject":
                        ↪  "true",
                        "vault.hashicorp.com/role":
                        ↪  "user-" + fmt.Sprint(request.UserID),
                        "vault.hashicorp.com/agent-inject-secret-secret.json":
                        ↪  "mooncloud_secret/user-" +
                        ↪  fmt.Sprint(request.UserID) + "/cred_" + fmt.Sprint(
                        ↪  *(request.requestAPITaskCommon.CredentialID)),
                        "vault.hashicorp.com/agent-inject-template-secret.json"
                        ↪  : "{{- with secret \"mooncloud_secret/user-" +
                        ↪  fmt.Sprint(request.UserID) + "/cred_" + fmt.Sprint(
                        ↪  *(request.requestAPITaskCommon.CredentialID)) + "\"
                        ↪  -}}\n" +
                                "{{ .Data.data | toUnescapedJSON }}\n" +
                                "{{- end }}\n",
                }
                service_account_name = "user-" + fmt.Sprint(request.UserID)
        }
```

Figura 17: K8s Manager Annotazioni Sidecar Credenziali

Rifinitura codice(generics) ridefinizione dei metodi tramite riscrittura di parti di codice tramite generics. La figura 18 mostra la ridefinizione dei metodi tramite l'utilizzo di generics, in particolare si può notare la definizione di un metodo generic e il relativo utilizzo per la gestione delle richieste di task e credenziali.

```
func genericHandleEntriesUser[T any, R any](m *nats.Msg, callback func(T) (R,
↪  error)) {
        // structure holding the request
        var request T

        // check the correctness of the request
        if err := bindJSON(m.Data, &request); err != nil {
                respondWithBadRequest(err, m)
                return
        }

        // call the controller function
        tmp_response, err := callback(request)
        if err != nil {
                respondWithError(err, m)
                return
        }

        // encode the response
        response, err := json.Marshal(tmp_response)
        if err != nil {
                respondWithError(err, m)
                return
        }

        // send the response
        m.Respond([]byte(response))
}

// apiAddTasks adds new tasks.
func apiAddTasks(m *nats.Msg) {
        genericHandleEntriesUser[[]requestTaskAdd, *responseInfo](m,
        ↪  controllerAddTasks)
}

...

```

Figura 18: K8s Manager Generics

4.3.5 HashiCorp Vault

Il componente HashiCorp Vault è stato aggiunto all'ecosistema Moon Cloud per la gestione delle credenziali e delle policy di accesso. Le modifiche effettuate sono le seguenti:

Secret sharing si è scelto come modalità di unlock un sistema basato su secret sharing che permette l'unlock tramite più chiavi distinte, impostando un minimo k. Quest'ultime possono essere memorizzate su host diversi per garantire che la chiave completa non sia mai ricostruibile da un host singolo. Nel nostro caso si è scelto un approccio a 2 chiavi: una da consegnare al cliente e una mantenuta dall'ecosistema Moon Cloud.

Auto Unseal all'avvio del Vault questo risulta sealed fino all'unlock manuale, per ovviare a questo problema si è cercato di automatizzare la procedura tramite due container sidecar che provvedono ad effettuare l'unlock ogni qual volta il vault si avvia/riavvia o in seguito a crash.

Sidecar injection il Vault fornisce un sistema completo per l'injection delle credenziali tramite mutating webhook in fase di creazione di un nuovo pod. Queste vengono specificate tramite annotation che referenziano la credenziale da iniettare e successivamente il pod creato le troverà in uno specifico path.

Policy il Vault prevede la possibilità di definire policy custom [16], scritte in HCL [14], per la gestione di autenticazione e autorizzazione, nel nostro caso si sono definite le policy relative al K8s Manager ed alle probe come segue:

- K8s Manager:
  - può creare, aggiornare e eliminare, ma non leggere, i secret sotto il path "mooncloud_secret"
  - può leggere tutti i mount dei secret e relativi sottopath
  - può creare, aggiornare e leggere ruoli Kubernetes, ovvero i metodi di autenticazione dei componenti
- Probe:
  - può leggere solo la credenziale associata al suo service account corrispondente all'ID dell'utente

```
path "mooncloud_secret/*" {
   capabilities = ["create","update","delete"]
}

path "sys/mounts" {
   capabilities = ["read"]
}

path "sys/mounts/*" {
   capabilities = ["create","update"]
}

path "auth/kubernetes/role/*"{
        capabilities=["create","update","read"]
}
```

Figura 19: Vault Policy K8s Manager

```
path "mooncloud_secret/{{identity.entity.aliases.auth_kubernetes_2132b1bd.metad
↪  ata.service_account_name}}/*"
↪  {
   capabilities = ["read"]
}
```

Figura 20: Vault Policy Probe

Autenticazione tramite Kubernetes l'autenticazione al vault può essere effettuata in vari modi, la modalità da noi scelta è stata tramite Kubernetes, ogni componente si autentica tramite service account token che verrà successivamente validato dal Vault tramite l'API di Kubernetes.

4.3.6 Probe

Il componente Probe si occupa dell'esecuzione dei test di valutazione e dell'invio dei risultati al componente Evidence Writer. Le modifiche effettuate sono le seguenti:

Connessione con NATS sono state aggiunte le chiamate NATS per l'invio dei risultati e i relativi test, l'invo avviene su un subject NATS specifico, relativo alla UER corrente, che verrà poi gestito da NATS tramite un sistema avanzato di subject routing (Jetstream [26]) che permette la creazione di code, gruppi di code e caching.

```
1  import json
2  from typing import Any, Coroutine
3  import nats
4  import mooncloud_driver.result as result
5  import mooncloud_driver.config as config
6  import dataclasses
7
8
9  class NatsController():
10
11     def __init__(self,current_config:config.Config):
12         self.nc=None
13         self.js=None
14         self.current_config=current_config
15         self.url="tls://"+self.current_config.nats_user+":"+self.current_config
           ↪  .nats_password+"@"+self.current_config.nats_host+":"+self.current_c
           ↪  onfig.nats_port
16
17     async def connect(self):
18         self.nc = await nats.connect(self.url)
19         self.js=self.nc.jetstream()
20
21     async def publishResult(self, subject:str, result_to_write:result.EWResult)
       ↪  -> Coroutine[Any,Any,nats.js.api.PubAck]:
22         return await self.js.publish(subject,
           ↪  json.dumps(dataclasses.asdict(result_to_write)).__str__().encode())
23
```

Figura 21: Connessione Probe a NATS

```python
from typing import Any, Coroutine
import pytest
import mooncloud_driver.nats_controller as nats_controller
import mooncloud_driver.config as config
import mooncloud_driver.result as result
import mooncloud_driver.writer as writer

t_config:config.Config=config.Config(
    logger_name='Probe',
    user_evaluation_rule_id="0",
    abstract_evaluation_rule_id="0",
    test_id="0",
    control_id="0",
    ...
)

async def
↪  internal_nats_debug_add_stream(nats_cont:nats_controller.NatsController):
    await nats_cont.js.add_stream(name="sample-stream", subjects=["foo"])

@pytest.mark.asyncio
async def test_nats_connect():
    nats_cont=nats_controller.NatsController(current_config=t_config)
    await nats_cont.connect()

@pytest.mark.asyncio
async def test_nats_publish():
    result_to_write=result.Result(base_extra_data={"test":"1"},integer_result=1
    ↪  ,pretty_result="test
    ↪  result")
    nats_cont=nats_controller.NatsController(current_config=t_config)
    await nats_cont.connect()
    await internal_nats_debug_add_stream(nats_cont=nats_cont)
    ack=await
    ↪  nats_cont.publishResult(subject="foo",result_to_write=writer.result_to_
    ↪  ew_result(result_to_write=result_to_write,current_config=t_config))
    assert ack is not None and ack !=""
```

Figura 22: Unit Test connessione Probe a NATS

Uso di credenziali aggiunta della lettura delle credenziali dal path in cui precentemente l'HashiCorp Vault le ha iniettate. La probe può quindi essere scritta in modo trasparente rispetto all'ambiente in cui verrà eseguita, senza doversi preoccupare della gestione interna delle credenziali. Inoltre essendo l'injection gestita direttamente dal sidecar del Vault queste ultime rimangono in memoria solo per il tempo necessario all'esecuzione della probe, garantendo un ulteriore livello di sicurezza.

```
    def __post_init__(self,probe,skip_init):
        """
        :raise jsonschema.exceptions.ValidationError if the input is not valid
        :raise jsonschema.exceptions.SchemaError if the schema is not valid
        :return:
        """
        if not skip_init:
            if self.read_from_stdin:
                reader = InputAndSchemaFromStdinReader(config=self)
            else:
                #reader = InputAndSchemaReaderFromFile(config=self)
                reader = InputAndSchemaFromEnvReader(config=self)
            jsonschema.validate(instance=reader.input_data,
            ↪  schema=reader.schema)
            object.__setattr__(self, 'input', reader.input_data)
            if self.is_production and probe.getCredentialType() is not None:
                with open('/vault/secrets/secret.json') as secret_file:
                    secret_data = json.load(secret_file)
                    object.__setattr__(self, 'credential', secret_data)
```

Figura 23: Uso di credenziali nelle probe

4.3.7 Evidence Writer

Il componente Evidence Writer fornisce un servizio di valutazione, aggregazione e salvataggio dei risultati ricevuti dalle probe. Le modifiche effettuate sono le seguenti:

Algoritmo di allineamento l'algoritmo di allineamento dei risultati prevede la possibilità di ottenere risultati in modo asincrono ed allinearli successivamente in modo tale da garantire l'integrità e validità del risultato finale. L'algoritmo finale prevede i seguenti step, quando una nuova test execution arriva:

- se non è la più frequente(scheduling minore) allora tutte le test execution più frequenti conterranno già una fake copy della test execution corrent, per cui verrà creata una nuova evaluation execution che includerà la nuova test execution
- se è una delle più frequenti(scheduling maggiore) possiamo trovarci in due casi:
  - ci sono alcuni test con frequenza minore, allora se:
    - almeno una è arrivata ci sarà almeno una evaluation execution con uno spazio vuoto
    - nessuna test execution è arrivata, allora dovrà essere creata una nuova evaluation execution
  - tutti i test hanno la stessa frequenza, allora se:
    - almeno uno è arrivato ci sarà almeno una evaluation execution con uno spazio vuoto
    - nessuna test execution è arrivata, allora dovrà essere creata una nuova evaluation execution

```
1  func align(data *request) (alignResult, error) {
2          var completedEvaluationExecutions, touchedEvaluationExecutions []int64
3
4          err := db.Transaction(func(tx *gorm.DB) error {
5
6                  ...
7
8                  evaluationExecutionsNotContainingMe, err :=
                   ↪  findPreviousEvaluationExecutionForMe(tx, uer.Id,
                   ↪  data.TestID)
9                  if err != nil && err != gorm.ErrRecordNotFound {
10                         return err
11                 }
12                 // if there are past EvaluationExecution not containing the
                   ↪  current Test(Execution),
13                 // then it means that past executions of this function left
                   ↪  space to include this TestExecution.
14                 // So, we iterate over them and add the current TestExecution.
15                 if len(evaluationExecutionsNotContainingMe) > 0 {
16                         for i := range evaluationExecutionsNotContainingMe {
17                                 // We use Association to add the current
                                   ↪  TestExecution to the EvaluationExecution.
18                                 if err = tx.Model(&evaluationExecutionsNotConta⌋
                                   ↪  iningMe[i]).Association(
19                                         "TestExecutions").Append(&currentTestEx⌋
                                           ↪  ecution); err != nil
                                           ↪  {
20                                         return err
21                                 }
22                                 ...
23                         }
24                 } else {
25                         ...
26                         if complete {
27                                 completedEvaluationExecutions =
                                   ↪  append(completedEvaluationExecutions,
                                   ↪  newEvaluationExecution.Id)
28                         }
29                         // add to the touched list
30                         touchedEvaluationExecutions =
                           ↪  append(touchedEvaluationExecutions,
                           ↪  newEvaluationExecution.Id)
31                 }
32                 return nil
33         })
34         result := alignResult{TouchedEvaluationExecutionID:
           ↪  touchedEvaluationExecutions,
35                 CompletedEvaluationExecutionID: completedEvaluationExecutions}
36         return result, err
37 }
```

Figura 24: Evidence Writer algoritmo di allineamento

Collegamento a NATS aggiunta sottoscrizione a coda NATS, la lettura viene effettuata dalla coda preconfigurata tramite variabile d'ambiente apposita, il for-

mato di interscambio viene condiviso tra l'Evidence Writer e le probe

```
func startSub() error {
        var err error
        // before proceed we must create the stream
        jetStream.AddStream(&nats.StreamConfig{
                Name:      "uer",
                Subjects:
                ↪  []string{viper.GetString(KeyNatsSubscriptionSubject)},
                Retention: nats.WorkQueuePolicy,
        })

        jetStream.AddConsumer("uer", &nats.ConsumerConfig{
                Durable: "ew",
        })

        // so subscribe to it
        natsSubscription, err =
        ↪  jetStream.Subscribe(viper.GetString(KeyNatsSubscriptionSubject),
        ↪  func(msg *nats.Msg) {

                natsLogger.Info("NATS_PreAlign", "Progress", "Started")
                messageReceived()
                // decoding the request from json.
                var request request
                if err := json.Unmarshal(msg.Data, &request); err != nil {
                        natsLogger.Error(err, "MessageProcessor", "Progress",
                        ↪  "FinishedWithError", "Details",
                        ↪  "UnmarshallingError")
                        return
                }
                controllerChan <- request
        }, nats.Durable("ew"))
        return err
}

func publishResultTo(resultToPublish *writeAndEvaluationResult, request
↪  *request) error {
        // now get ready to publish the result.
        natsLogger.Info("NATS_PostAlign", "Action", "SendingPublishResult",
        ↪  "Progress", "Starting",
                "TestID", request.TestID, "UserEvaluationRuleID",
                ↪  request.UserEvaluationRuleID, "UserID", request.UserID,
                "TouchedEvaluationExecution",
                ↪  resultToPublish.TouchedEvaluationExecutionID,
                ↪  "CompletedEvaluationExecution",
                ↪  resultToPublish.CompletedEvaluationExecutionID)

        // encode the result before sending over
        encoded, err := json.Marshal(&resultToPublish)
        ...
        return nil
}
...
```

Figura 25: Evidence Writer NATS

4.4 Architettura Finale

Viene di seguito rappresentata l'architettura finale dell'ecosistema Moon Cloud.

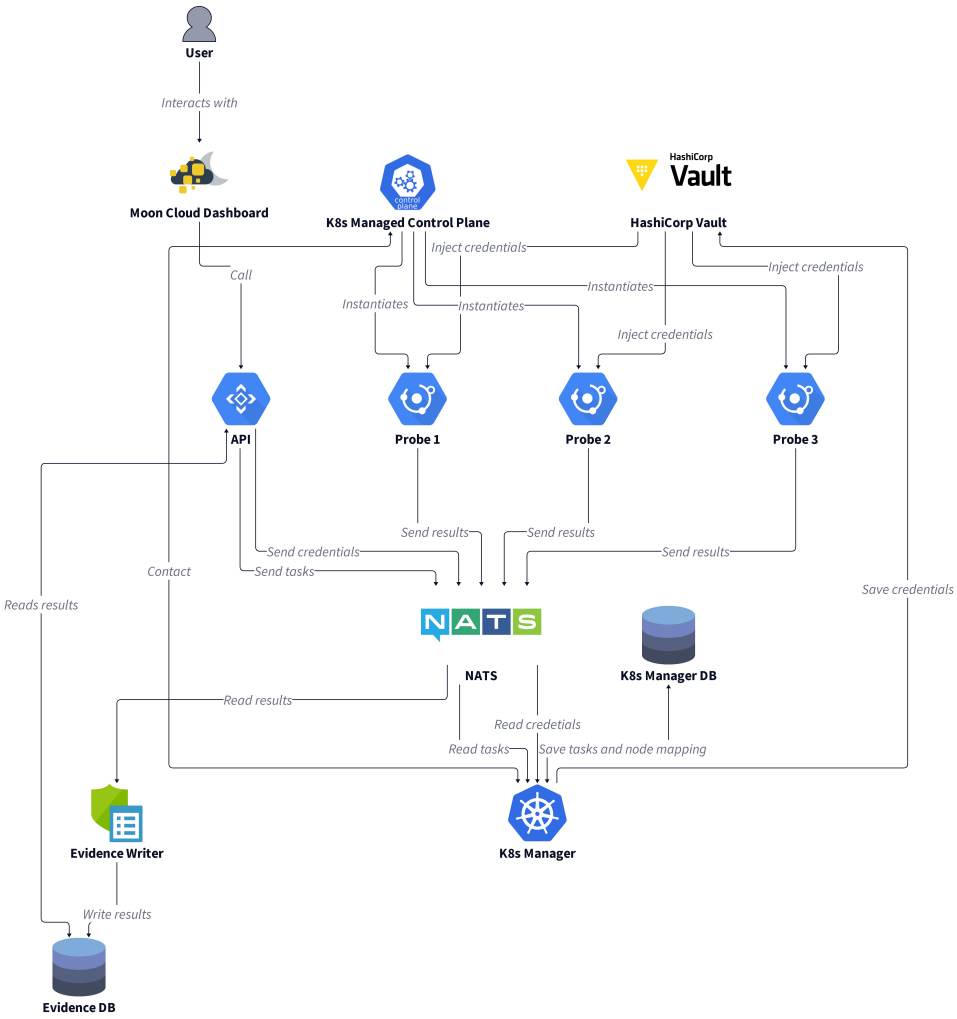

Figura 26: Schema architettura finale

4.5 Continuos Integration/Continuos Delivery

Tutti i componenti supportano un sistema di Continuos Integration/Continuos Delivery tramite Gitlab CI [13] e Dockerfile [12]. E' stato inoltre realizzato il file Docker Compose per il testing locale. Il funziomanto dei precedenti file è il seguente:

- L'immagine Docker viene costruita seguendo le istruzioni presenti nel Dockerfile, in particolare in questa fase vengono installate le dipendenze necessarie al funzionamento del componente, vengono copiati i file necessari e viene impostato il comando di avvio del componente.
- Il file gitlab-ci.yml, presente in ciascun repository, definisce la pipeline di CI/CD. La maggior parte delle CI/CD contiene le seguenti fasi: build per la costruzione dell'immagine Docker, test per l'esecuzione dei test e deploy per il deployment dell'immagine Docker
- Il file docker-compose.yml permette di testare il componente in locale, in particolare è permette di testare l'interoperabilità tra i vari componenti.

```
1  default:
2      image: docker:latest
3  stages:
4    - build
5
6  build_evidence_writer:
7      stage: build
8      variables:
9          CONTAINER_RELEASE_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_BRANCH
10     script:
11     - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
12     - docker build -t $CONTAINER_RELEASE_IMAGE -f test.dockerfile .
13     - docker push $CONTAINER_RELEASE_IMAGE
14
```

Figura 27: Evidence Writer Gitlab CI

```
FROM golang:1.17.0-alpine3.14 as compiler

RUN apk update && apk add git
WORKDIR /usr/src/app

COPY . .

RUN go build -o mooncloud-evidence_writer ./cmd/evidence-writer

FROM alpine:3.14.1 AS executor

COPY --from=compiler /usr/src/app/mooncloud-evidence_writer
↪  /usr/local/bin/mooncloud-evidence_writer
COPY --from=compiler /usr/src/app/test.evidence-writer.toml
↪  /etc/evidence-writer/evidence-writer.toml
COPY --from=compiler /usr/src/app/config.json .

ENTRYPOINT ["/usr/local/bin/mooncloud-evidence_writer"]
```

Figura 28: Evidence Writer Dockerfile

```
version: '3.5'

services:
  nats:
    image: nats:alpine
    ports:
    - '4222:4222'
    - '8222:8222'
    command: '--jetstream'

  ...

  evidence-writer:
    image: repository.v2.moon-cloud.eu:4567/experiments/evidence-writer:${EW_CI
    ↪  _COMMIT_BRANCH:-align-nats-js}
    build:
      dockerfile: test.dockerfile
      context: .
    ports:
    - '${EVIDENCE_WRITER_EXPOSED_PORT-2122}:2122'
    ...
```

Figura 29: Evidence Writer Docker Compose

4.6 Correzioni Architettura Precedente

Come primo punto della migrazione si è scelto di iniziare dalla correzione di alcune criticità del sistema esistente, in particolare:

- Coda di default nel Master: aggiunta possibilità di avere una coda SQS di default che identifica la zona pubblica in caso non venga specificata una più specifica

- Template Docker Compose [9]: correzione di alcune configurazioni errate nei template Docker Compose

Capitolo 5. Caso di studio: Probe ML

Come primo test della nuova architettura si è scelto come caso di studio lo sviluppo di una probe ML per il controllo di conformità delle metriche dichiarate, in particolare accuracy, precision, recall e specificity.

5.1 Obiettivi

La probe prevede la verifica delle metriche dichiarate dall'utente in fase di configurazione della regola di valutazione, in particolare verifica che accuracy, precision, recall e specificity dichiarate siano maggiori o uguali ottenute interrogando il target. La valutazione di compliance viene effettuata comparando i risultati ottenuti dallo script di verifica presente sul target e le metriche dichiarate dall'utente in fase di compilazione della regola di valutazione. Il target della probe è un server Amazon Linux 2 ospitato sul cloud pubblico AWS che contiene una pipeline ML composta dalle classiche fasi di training e valutazione, la probe ha accesso alla rete interna (VPC) [7] del target, l'accesso viene eseguito tramite connessione SSH basata su autenticazione a chiave asimmetrica RSA, ECDSA e ED25519. Il deployment è stato effettuato tramite la console di AWS, la pipeline è invece stata creata ad hoc per il test, essa è composta da un modello di classificazione binaria basato su un dataset di esempio che permette l'esportazione nei formati ONNX e PKL.

5.2 Progettazione e Implementazione

5.2.1 Modello di Esecuzione

Il modello di esecuzione della probe di compone da una macchina a stati finiti, per ogni stato è definito un metodo forward, che esegue le operazioni necessarie e prosegue verso lo stato successivo, e un metodo rollback che in caso di errori durante l'esecuzione ne annulla le modifiche. Le due catene di metodi forward e rollback prevedono un'esecuzione sequenziale, in caso di errore in uno stato la catena di rollback viene eseguita fino al primo stato, in caso di successo la catena di forward viene eseguita fino all'ultimo stato.

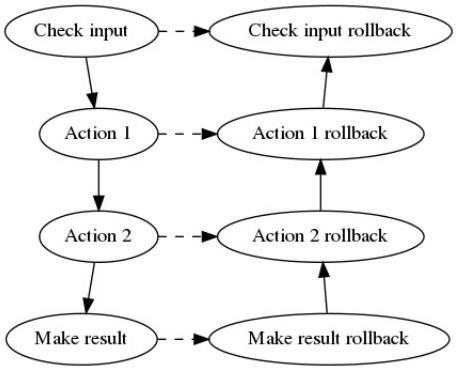

Figura 30: Modello macchina a stati finiti

5.2.2 Gestione delle Credenziali

Le credenziali necessarie per la connessione SSH sono iniettate nella probe dal sidecar del Vault all'avvio. Successivamente la probe legge le credenziali da un path predefinito e prosegue col suo controllo. La probe deve dichiarare la volontà di utilizzare le credenziali iniettate definendo il metodo `requires_credential`.

5.2.3 Connessione

La libreria SSH è stata scritta basandosi sulle librerie *Fabric* e *Paramiko*, essa offre la possibilità di connettersi al target e di eseguire comandi remoti. L'autenticazione può avvenire tramite password, chiave privata o tramite chiave privata e passphrase.

```
...
def connect_ssh(self) -> Connection:
    if self._client is None:
        assert self._host is not None and self._host != "", "[input]
        ↪  host input field not valid"
        assert self._port is not None and self._port != "", "[input]
        ↪  port input field not valid"
        assert self._username is not None and self._username != "",
        ↪  "[input] username input field not valid"
        assert (self._password is not None and self._password != "")
        ↪  or (self._private_key is not None and self._private_key !=
        ↪  ""), "[input] password or private key input field not
        ↪  valid"
        private_key: paramiko.PKey = None
        if self._private_key is not None and self._private_key != "":
            private_key = paramiko.RSAKey.from_private_key(io.StringIO(
            ↪  (self._private_key),self._private_key_passphrase)
        self._client = Connection(host=self._host, port=self._port,
                                  user=self._username,
                                    connect_kwargs={
                                      "password": self._password,
                                      "pkey": private_key
                                  })
    return self._client
...
```

Figura 31: Probe ML Connessione SSH

5.2.4 Ottenimento delle Metriche

Una volta connesso al target la probe lancia il comando di avvio dello script di verifica, quest ultimo propone al modello ML una serie di dati prelevati selettivamente da un dataset, precedentemente fornito, e ottiene le metriche correnti relative allo stao del modello ML.

5.2.5 Valutazione dei Risultati

I risultati vengono quindi comparati con quelli dichiarati dall'utente in fase di compilazione della regola di valutazione. Se questi superano/eguagliano le soglie dichiate la probe tornerà un risultato positivo, altrimenti il risultato sarà negativo.

```
{
    "integer_result": 0,
    "pretty_result": "Declared metrics match the model status",
    "extra_data": {
      "detected_accuracy": 0.9525909592061742,
      "detected_recall": 0.9586206896551724,
      "detected_precision": 0.9434389140271493
    }
}
```

(a) Probe ML Output Success

```
{
    "integer_result": 1,
    "pretty_result": "Declared metrics doesn't match the model status",
    "extra_data": {
        "detected_accuracy": 0.9525909592061742,
        "detected_recall": 0.9586206896551724,
        "detected_precision": 0.9434389140271493
    }
}
```

(b) Probe ML Output Failure

5.2.6 Probe

La figura 33 mostra il codice della probe ML, essa è composta da un metodo `get_data` che si occupa di ottenere i dati dal target, un metodo `evaluate_data` che si occupa di confrontare i dati ottenuti con quelli dichiarati dall'utente.

```
class Probe(abstract_probe.AbstractProbe):

    def get_data(self, inputs=None):
        c: SshClient = SshClient(
            host=self.config.input['config']['target'],
            port=self.config.input['config']['port'] if
            ↪  self.config.input['config'].get('port') is not None
            ↪  else 22,
            username=self.config.credential['username'],
            password=None,
            private_key=self.config.credential['private_key'],
            private_key_passphrase=None)
        c.connect_ssh()
        # check if the script format is valid, it also blocks some
        ↪  types of arbitrary command execution
        script_path=self.config.input['script']['path']
        assert
        ↪  re.fullmatch(re.compile(r'(\/.*?\.[\w:]+)'),script_path),
        ↪  "Error script path not found"
        out_raw=c.send_command(script_path,SshClient.onNotZeroExitCode
        ↪  Action.STOP)
        out_obj=json.loads(out_raw['stdout'])
        return out_obj

    def evaluate_data(self, inputs=None):
        res=True
        metrics=["accuracy","recall","precision","specificity"]
        declared_data={}
        for metric in metrics:
            declared_value=int(self.config.input['metrics'][metric])/1
            ↪  00
            # skip not defined fields (0=not defined)
            if declared_value!=0:
                declared_data[metric]={"value":declared_value,"verifie
                ↪  d":inputs[metric]>=declared_value}
                self.result.base_extra_data['detected_'+metric]=inputs
                ↪  [metric]
                if not declared_data[metric]['verified']:
                    res=False
        if res:
            self.result.integer_result=result.INTEGER_RESULT_TRUE
            self.result.pretty_result="Declared metrics match the
            ↪  model status"
        else:
            self.result.integer_result=result.INTEGER_RESULT_FALSE
            self.result.pretty_result="Declared metrics doesn't match
            ↪  the model status"

    ...
```

Figura 33: Probe ML

5.2.7 File Accessori

Vengono inoltre definiti dei file accessori per il test, il build e la configurazione della probe:

- **test.json**: permette il test in locale dell'input utente (figura 34)
- **schema.json**: fornisce la validazione dell'input utente (figura 35)
- **requirements.txt**: definisce l'installazione delle dipendenze
- **readme.md**: fornisce la documentazione della probe
- **Dockerfile**: definisce la procedura di creazione dell'immagine Docker
- **.gitignore**: definisce i file da escludere dal caricamento sul repository
- **.dockerignore**: definisce i file da escludere dall'immagine Docker
- **.gitlab-ci.yml**: definisce la pipeline di CI/CD

```
{
  "config": {
    "target": "www.google.com",
    "port": 22
  },
  "script":{
    "path":"/home/ec2-user/check-metrics.sh"
  },
  "metrics":{
    "basic_response": "Yes",
    "accuracy": 50,
    "precision": 50,
    "recall": 50,
    "specificity": 50
  }
}
```

Figura 34: Probe ML test json

```
{
    "type": "object",
    "properties": {
        "config": {
            "type": "object",
            "properties":{
                "target":{
                    "type":"string"
                },
                "port":{
                    "type":"number"
                }
            }
        },
        "script":{
            "type":"object",
            "properties":{
                "path":{
                    "type":"string"
                }
            }
        },
        "metrics":{
            "type":"object",
            "properties":{
                "basic_response":{
                    "type":"string"
                },
                "accuracy":{
                    "type":"number"
                },
                "precision":{
                    "type":"number"
                },
                "recall":{
                    "type":"number"
                },
                "specificity":{
                    "type":"number"
                }
            }
        }
    }
}
```

Figura 35: Probe ML schema json

5.3 Esecuzione

Il flusso di esecuzione della probe parte dall'inserimento dell'input utente, per cui l'utente tramite la dashboard inserisce la configurazione necessaria per la valutazione, la dashboard quindi provvederà ad inviarli all'API, la quale a sua volta contatterà l'esecutore assegnato alla specifica zona. L'esecutore provvederà quindi ad eseguire la probe contattando l'API Kubernetes del cluster on-premise, a questo punto verrà eseguito il codice della probe che, dato un input conforme allo schema, si connette al target, esegue lo script di verifica e confronta i risultati ottenuti con quelli dichiarati dall'utente, e ne ritorna il risultato della valutazione. Infine il risultato verrà inviato all'Evidence Writer che provvederà alla validazione e scrittura dell'evidence per la successiva lettura da parte dell'API. La figura 36a mostra un esempio di input conforme, mentre la figura 36b mostra il risultato dell'esecuzione della probe in caso positivo.

```
{
    "config": {
      "target": "3.70.98.116",
      "port": 22
    },
    "script":{
      "path":"/home/ec2-user/check
      ↪  -metrics.sh"
    },
    "metrics":{
      "basic_response": "Yes",
      "accuracy": 50,
      "precision": 50,
      "recall": 50,
      "specificity": 0
    }
}
```

(a) Probe ML input success json

```
{
    "integer_result": 0,
    "pretty_result": "Declared
    ↪  metrics match the model
    ↪  status",
    "extra_data": {
      "detected_accuracy":
      ↪  0.9525909592061742,
      "detected_recall":
      ↪  0.9586206896551724,
      "detected_precision":
      ↪  0.9434389140271493
    }
}
```

(b) Probe ML ouput success json

Figura 36: Probe ML input e output success

Si noti che nell'input vengono dichiarati il target, la porta SSH da utilizzare, lo script che verrà invocato sul target e le metriche da analizzare, in questo caso il risultato è positivo, in quanto le metriche ricavate dal target sono maggiori o uguali di quelle dichiarate dall'utente. La probe supporta inoltre l'analisi solo di determinate metriche, in caso di dichiarazione di metriche con un valore uguale a zero, la probe non le considererà.

Un'altro possibile risultato della probe dato un input superiore alle metriche rilevate sul target è mostrato nella figura 37a che fornisce un esempio di input

conforme, mentre la figura 37b mostra il risultato dell'esecuzione della probe in caso negativo.

```
{
    "config": {
      "target": "3.70.98.116",
      "port": 22
    },
    "script":{
      "path":"/home/ec2-user/check-metrics.sh"
    },
    "metrics":{
      "basic_response": "Yes",
      "accuracy": 50,
      "precision": 50,
      "recall": 100,
      "specificity": 0
    }
}
```

(a) Probe ML input failure json

```
{
    "integer_result": 1,
    "pretty_result": "Declared metrics doesn't match the model status",
    "extra_data": {
        "detected_accuracy": 0.9525909592061742,
        "detected_recall": 0.9586206896551724,
        "detected_precision": 0.9434389140271493
    }
}
```

(b) Probe ML ouput failure json

Figura 37: Probe ML input e output failure

Si noti in questo caso che il risultato è negativo, in quanto le metriche ricavate dal target sono inferiori a quelle dichiarate dall'utente.

Tutto ciò viene riportato nella dashboard come mostrato nella figura 38.

Figura 38: Risulati Dashboard Probe ML

E' inoltre possibile visualizzare lo storico delle esecuzione delle probe, come mostrato nella figura 39.

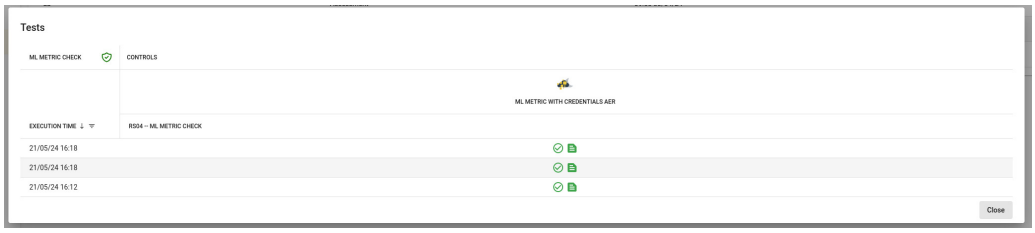

Figura 39: Storico Risulati Dashboard Probe ML

Capitolo 6. Esperimenti e Discussione

In questo capitolo vengono presentati gli esperimenti effettuati per valutare le prestazioni della nuova architettura di Moon Cloud. In particolare, si sono confrontati i tempi di esecuzione delle probe eseguite con la vecchia architettura e con la nuova architettura.

6.1 Setup Sperimentale

La nuova architettura di Moon Cloud è stata testata utilizzando un cluster Kubernetes composto da 3 nodi, 1 master e 2 worker, dotati ognuno di 16 core CPU e 32 GB RAM, per un totale di 48 core CPU e 96 GB RAM. La precedente architettura è stata testata utilizzando un cluster Docker Swarm composto da 1 nodo, dotato di 12 core CPU e 16 GB RAM. Come target delle probe è stato utilizzata una macchina virtuale EC2, ospitata sul cloud pubblico AWS, dotata di 2 vCPU e 2 GB RAM. Per ogni esperimento sono state eseguite 10 esecuzioni indipendenti e sono stati calcolati i tempi di esecuzione medi.

6.2 Overhead

Per valutare l'overhead introdotto dalla nuova architettura rispetto alla soluzione precedente, sono state eseguite le stesse probe con entrambe le architetture. In particolare, si è scelto di confrontare l'overhead introdotto dal framework stesso e l'overhead introdotto in caso di probe di diversa complessità. L'overhead legato al framework risulta una misura particolarmente importante in quanto rappresenta il costo aggiuntivo introdotto dalla nuova architettura rispetto alla soluzione precedente, mentre l'overhead legato alle probe di diversa complessità permette di valutare l'impatto della nuova architettura in caso di probe più o meno complesse.

6.3 Overhead del Framework

Come prima comparazione si è scelto di eseguire una probe vuota per comparare l'effettivo overhead del framework, i risultati di tale esperimento sono mostrati in Figura 40 nella colonna "Empty Probe". Si può notare che l'overhead introdotto dal nuovo framework è di circa 0,06s superiore rispetto alla soluzione precedente, per cui si può affermare che l'overhead introdotto dal nuovo framework rispetto al precedente è trascurabile.

6.4 Overhead Rispetto alla Soluzione Precedente

Sono poi stati effettuati ulteriori esperimenti con probe di diversa complessità per valutare l'overhead della nuova architettura rispetto alla soluzione precedente. I risultati di tali esperimenti sono mostrati in Figura 40 nelle sezioni "Simple Probe", "Medium Probe" e "Complex Probe". Nel dettaglio, le probe utilizzate sono le seguenti:

- **Simple Probe**: verifica di disponibilità di un servizio tramite HTTP(s). La probe effettua una richiesta HTTP(s) ad un servizio e verifica che il servizio risponda con un codice di stato 200. In caso contrario, la probe segnala un errore.
- **Medium Probe**: verifica vulnerabilità alla CVE-2014-0160(Heartbleed). La probe effettua una richiesta di heartbeat malevola che induce il server a rispondere con contenuti della sua memoria, potenzialmente rivelando dati sensibili, quindi segnala un errore se il server è vulnerabile.
- **Complex Probe**: vulnerability assessment tramite OpenVAS. La probe effettua una scansione di vulnerabilità su un server utilizzando OpenVAS e segnala un errore se vengono trovate vulnerabilità.

Dai risultati ottenuti si può notare che l'overhead introdotto dalla nuova architettura rispetto alla soluzione precedente è trascurabile per le probe di tipo "Simple" e "Medium", mentre per le probe di tipo "Complex" l'overhead introdotto è di circa 1,2s. Questo risultato è dovuto al fatto che la nuova architettura è stata progettata per essere più flessibile e scalabile rispetto alla soluzione precedente, per cui è possibile che l'overhead introdotto sia maggiore per le probe più complesse. Tuttavia, si può notare che l'overhead introdotto dalla nuova architettura rispetto alla soluzione precedente è comunque accettabile, considerando i benefici che essa apporta in termini di flessibilità e scalabilità.

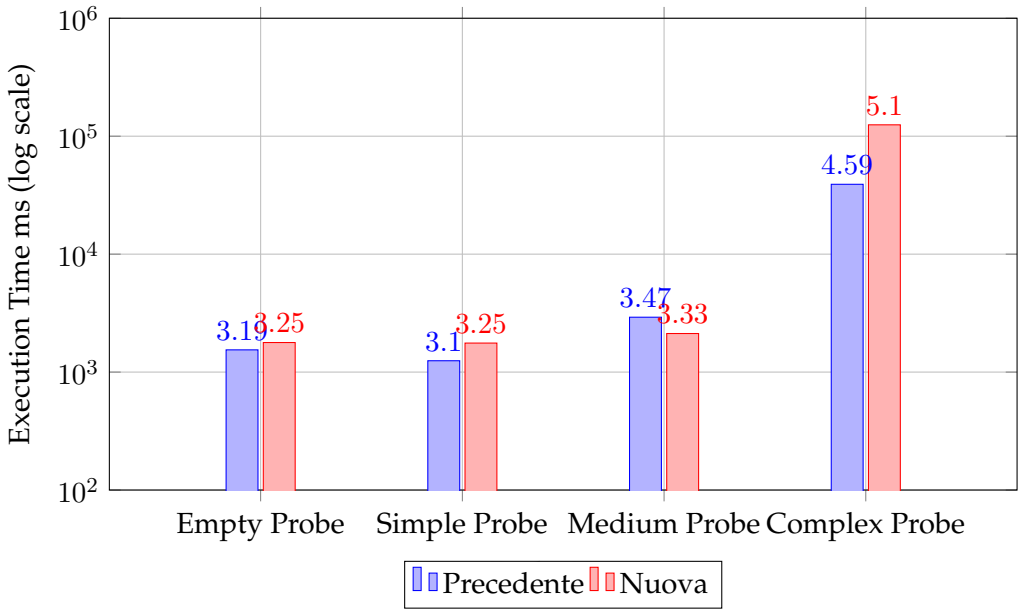

Figura 40: Comparazione tempi di esecuzione per probe di diversa tipologia

6.5 Discussione

L'architettura nuova risolve le criticità della soluzione precedente (Capitolo 3):

- **Gestione delle credenziali basica (C1)**: la nuova architettura permette di gestire le credenziali in modo più sicuro rispetto alla soluzione precedente, in quanto le credenziali sono memorizzate nel Vault e iniettate in modo sicuro all'interno delle probe tramite sidecar. A livello sperimentale si può osservare il passaggio da una sola chiave di cifratura del precedente database custom al passaggio ad un sistema di secret sharing con n=2 chiavi.
- **Dipendenza da un sistema pub/sub esterno (C2)**: la nuova architettura non dipende da un sistema pub/sub esterno, in quanto il sistema di messaggistica è integrato all'interno dell'infrastruttura. Questo permette di semplificare la gestione delle comunicazioni intra e infra-cluster e di eliminare la dipendenza da componenti esterni per il funzionamento del sistema. A livello sperimentale si può osservare il passaggio da molteplici sistemi pub/sub esterni ad un unico sistema pub/sub integrato.

- **Eccessiva presenza di componenti custom (C3)**: la nuova architettura è basata su componenti standard e open source, che permettono di ridurre il numero di componenti custom necessari per il funzionamento del sistema. Questo permette di semplificare la gestione del sistema e di ridurre i costi di manutenzione. A livello sperimentale si può notare la rimozione dei componenti custom Director, Master, Puppet, Credential Sniffer, Credential Manager e Evaluation Result Manager e la relativa sostituzione con componenti standard e open source come NATS e Vault, portando il numero totale di componenti custom da 10 a 5.
- **Deployment in un sistema legacy (C4)**: la nuova architettura, basata su Kubernetes, garantisce scalabilità, affidabilità e replicazione. A livello sperimentale si può notare il maggior numero di file di configurazione necessari per il deployment della nuova architettura rispetto alla soluzione precedente, passando da 1 file *docker-compose* per ogni componente (o aggregazione di componenti) ad un minimo di 2 file YAML per ogni componente (*Deployment* e *Service*) e un massimo di 6 YAML per ogni componente (*Deployment*, *Service*, *ConfigMap*, *PVC*, *Secret*, *Ingress*).
- **Risultati non puntuali (C5)**: la nuova architettura permette di ottenere risultati più puntuali rispetto alla soluzione precedente, in quanto ora i risultati delle probe sono collegati temporalmente con l'esecuzione delle stesse e aggregate secondo un rigoroso algoritmo di allineamento. A livello sperimentale si può notare la maggiore velocità di aggregazione dei risultati a lato API, in quanto ora i risultati delle probe sono già allineati e pronti per essere ottenuti, portando il numero di database coinvolti da 2 (Influx e Postgres) a 1 (Postgres).

Criticità	Metrica	Valore Precedente	Valore attuale
Gestione delle credenziali basica (C1)	Chiavi coinvolte	1	2
Dipendenza da un sistema pub/sub esterno (C2)	Sistemi pub/sub	N	1
Eccessiva presenza di componenti custom (C3)	Componenti custom	10	5
Deployment in un sistema legacy (C4)	File di configurazione	1	2-6
Risultati non puntuali (C5)	Database coinvolti	2	1

Tabella 11: Riepilogo delle modifiche rispetto alla soluzione precedente

Capitolo 7. Conclusioni

Come mostrato nei Capitoli 4 e 6, è ora possibile garantire l'assurance fino ai livelli più interni dei sistemi ICT. In tal senso, l'architettura di Moon Cloud è stata progettata per risolvere le criticità della soluzione precedente. Viene di seguito riassunto il percorso seguito e i possibili sviluppi futuri.

7.1 Risultati Ottenuti

L'evoluzione delle applicazione e dei sistemi verso approcci distribuiti e dinamici richiede la presenza di sistemi di assurance e certificazione sempre più sofisticati per garantirne la sicurezza. L'architettura di Moon Cloud precedente presentava diverse criticità che ne limitavano l'efficienza e la scalabilità. E' stata quindi definita una roadmap in termini di criticità, obiettivi e vincoli. In particolare, la gestione delle credenziali era basata su un database custom (C1), il sistema dipendeva da un sistema pub/sub esterno (C2), era basata su molteplici componenti custom (C3) e richiedeva un deployment manuale in un sistema legacy (C4); tra le varie criticità inoltre, i risultati delle probe non erano puntuali e richiedevano un'aggregazione manuale a lato API (C5). Si è quindi deciso di progettare una nuova architettura che risolvesse queste criticità, partendo dalla definizione di vincoli, criticità e obiettivi, passando per la progettazione dell'architettura e l'implementazione dei componenti, fino alla valutazione delle prestazioni tramite esperimenti. La definizione dei vincoli ha evidenziato la necessità di mantenere la compatibilità con i componenti precedenti (V1), di garantire la sicurezza delle informazioni (V2), di avere un sistema ad elevate prestazioni (V3) e facilmente manutenibile (V4). Sono quindi stati definiti gli obiettivi di migrazione, che prevedevano la costruzione di un sistema mdulare installabile nelle diverse modalità di deployment (O1), la garanzia di affidabilità, scalabilità e replica (O2), la migrazione verso un sistema cloud-native (O3) e la garanzia di risultati fine-grained (O4). Si è quindi progettata la nuova architettura di Moon Cloud al fine di risolvere le criticità e raggiungere gli obiettivi prefissati garantendo i vincoli. La fase di migrazione ha previsto tre macroaree: la progettazione della migrazione, la migrazione del sistema a Kubernetes e la migrazione dei componenti. La progettazione della migrazione ha evidenziato come migliore modalità di deployment una soluzione basata su cluster indipendenti connessi tramite sistemi custom di NAT-traversal, la migrazione a Kubernetes ha previsto la definizione delle nuove recipe per il deployment dei componenti e infine la migrazione dei componenti ha previsto la modifica e aggiunta/rimozione di alcuni componenti per comporre la nuova architettura. Il risultato ottenuto è un sistema più flessibile, scalabile e sicuro rispetto alla soluzione precedente, che permette di gestire le credenziali in modo più sicuro, di ridurre il numero

di componenti custom, di semplificare il deployment e di ottenere risultati più puntuali. In particolare, la nuova architettura di Moon Cloud risolve le criticità sopra descritte introducendo un sistema di gestione delle credenziali basato su Vault, un sistema di messaggistica integrato all'interno del framework, rimozione di molti componenti precedenti e passaggio a componenti standard e open source, un deployment automatizzato su Kubernetes e risultati puntuali e aggregati automaticamente tramite un algoritmo di allineamento. Sono infine stati effettuati degli esperimenti di comparazione tra le due versioni dell'architettura che evidenziano come l'overhead introdotto dalla nuova architettura rispetto alla soluzione precedente sia maggiore, ma comunque accettabile, in quanto i benefici apportati dalla nuova architettura in termini di flessibilità e scalabilità sono superiori ai tempi di esecuzione aggiuntivi. In termini quantitativi, invece si sono evidenziati numerosi vantaggi rispetto alle metriche legate alle criticità, in particolare si è passati da una chiave di cifratura a un sistema di secret sharing con n=2 chiavi, da molteplici sistemi pub/sub esterni ad un unico sistema pub/sub integrato, da 10 componenti custom a 5 componenti custom, da 1 docker compose per ogni componente a 2-6 YAML per ogni componente e da 2 database coinvolti nell'ottenimento dei risultati a 1 database.

7.2 Sviluppi Futuri

La tesi si presta a sviluppi futuri. In primo luogo, è possibile introdurre nuove funzionalità per migliorare le prestazioni e la scalabilità del sistema, ad esempio aggiungendo un sistema di caching per i risultati delle probe a lato cliente, un sistema di monitoraggio per il controllo delle prestazioni e un sistema di logging per il tracciamento delle operazioni. E' inoltre possibile passare ad un modello avanzato di gestione delle credenziali basato su credenziali effimere, ovvero con una durata in termini temporali dell'ordine dei secondi, che permetterebbero di non avere più credenziali long-standing e quindi di aumentare ulteriormente la sicurezza dell'architettura. Infine è possibile introdurre un sistema di orchestrazione delle probe basato su un algoritmo di scheduling avanzato, che permetta di ottimizzare l'esecuzione delle probe in base a diversi criteri come la priorità, la complessità e la disponibilità delle risorse.

Bibliografia

[1] *Angular Documentation*. 2024. URL: https://angular.io/docs (visitato il 24/03/2024).

[2] Marco Anisetti et al. «Moon Cloud: A Cloud Platform for ICT Security Governance». In: *Proc. of IEEE GLOBECOM 2018*. Abu Dhabi, UAE, dic. 2018.

[3] Claudio A. Ardagna e Nicola Bena. «Non-Functional Certification of Modern Distributed Systems: A Research Manifesto». In: *Proc. of IEEE SSE 2023*. Chicago, IL, USA, lug. 2023.

[4] Claudio A. Ardagna, Nicola Bena e Ramon Martín de Pozuelo. «Bridging the Gap Between Certification and Software Development». In: *Proc. of ARES 2022*. Vienna, Austria, ago. 2022.

[5] *AWS EKS Documentation*. 2024. URL: https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html (visitato il 24/03/2024).

[6] *AWS SQS Documentation*. 2024. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html (visitato il 24/03/2024).

[7] *AWS VPC Documentation*. 2024. URL: https://docs.aws.amazon.com/vpc/ (visitato il 24/03/2024).

[8] *Cert Manager Documentation*. 2024. URL: https://cert-manager.io/docs/ (visitato il 24/03/2024).

[9] *Docker Compose Documentation*. 2024. URL: https://docs.docker.com/compose/ (visitato il 24/03/2024).

[10] *Docker Documentation*. 2024. URL: https://docs.docker.com/ (visitato il 24/03/2024).

[11] *Docker Swarm Documentation*. 2024. URL: https://docs.docker.com/engine/swarm/ (visitato il 24/03/2024).

[12] *Dockerfile Documentation*. 2024. URL: https://docs.docker.com/reference/dockerfile/ (visitato il 24/03/2024).

[13] *Gitlab CI Documentation*. 2024. URL: https://docs.gitlab.com/ee/ci/ (visitato il 24/03/2024).

[14] *HashiCorp HCL Documentation*. 2024. URL: https://github.com/hashicorp/hcl (visitato il 24/03/2024).

[15] *HashiCorp Vault Documentation*. 2024. URL: https://developer.hashicorp.com/vault/docs (visitato il 24/03/2024).

[16] *HashiCorp Vault Policies Documentation*. 2024. URL: https://developer.hashicorp.com/vault/docs/concepts/policies (visitato il 24/03/2024).

[17] *Helm Documentation*. 2024. URL: https://helm.sh/docs/ (visitato il 24/03/2024).
[18] *InfluxDB Documentation*. 2024. URL: https://docs.influxdata.com/ (visitato il 24/03/2024).
[19] *Konnectivity Documentation*. 2024. URL: https://kubernetes.io/docs/tasks/extend-kubernetes/setup-konnectivity/ (visitato il 24/03/2024).
[20] *KubeFed Documentation*. 2024. URL: https://github.com/kubernetes-retired/kubefed/blob/master/docs/userguide.md (visitato il 24/03/2024).
[21] *Kubernetes Documentation*. 2024. URL: https://kubernetes.io/docs/home/ (visitato il 24/03/2024).
[22] *Let's Encrypt Documentation*. 2024. URL: https://letsencrypt.org/docs/ (visitato il 24/03/2024).
[23] *Liqo Documentation*. 2024. URL: https://docs.liqo.io/en/v0.10.1/ (visitato il 24/03/2024).
[24] *Moon Cloud Website*. 2024. URL: https://www.moon-cloud.eu (visitato il 24/03/2024).
[25] *NATS Documentation*. 2024. URL: https://docs.nats.io/ (visitato il 24/03/2024).
[26] *NATS Jetstream Documentation*. 2024. URL: https://docs.nats.io/nats-concepts/jetstream (visitato il 24/03/2024).
[27] *PostgreSQL Documentation*. 2024. URL: https://www.postgresql.org/docs/ (visitato il 24/03/2024).
[28] *Rancher Documentation*. 2024. URL: https://rancher.com/docs/ (visitato il 24/03/2024).
[29] *Redis Documentation*. 2024. URL: https://redis.io/docs/ (visitato il 24/03/2024).