ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sistema di autoscaling per runner self-hosted GitHub Actions su Proxmox con fallback su runner GitHub-hosted

Tesi di laurea in:
Ingegneria e Scienze Informatiche

Relatore
Prof. Danilo Pianini

Candidato
Alan Barzanti

Correlatori
Ing. Martina Baiardi
Prof. Giovanni Ciatto

II Sessione di Laurea
Anno Accademico 2025-2026

Sommario

Nello sviluppo software, l'esigenza di superare i rigidi processi sequenziali tradizionali ha portato all'affermazione della metodologia Agile e, successivamente, della filosofia DevOps, volta a integrare sviluppo e operazioni IT. Un pilastro di questo approccio è la Continuous Integration, che prevede la frequente integrazione del codice attraverso verifiche automatizzate racchiuse in una pipeline. Eseguire queste pipeline in cloud o on-premise comporta una serie di trade-off: un'infrastruttura on-premise offre maggiore controllo, prestazioni più stabili e un potenziale risparmio sui costi, ma può comportare spreco di risorse fisiche e maggiore latenza. Il presente progetto di tesi si colloca in questo contesto, ponendosi l'obiettivo di utilizzare per le pipeline di CI un'infrastruttura on-premise basata su Proxmox, ottimizzando la gestione delle risorse e minimizzando la latenza. La piattaforma utilizzata, GitHub Actions, esegue i workflow su risorse computazionali dette runner, fornite dalla piattaforma stessa (GitHub-hosted) o gestite dall'utente (self-hosted). Il sistema progettato comprende due componenti: un meccanismo di autoscaling dei runner self-hosted, lato infrastruttura, e un fallback sui runner GitHub-hosted, lato workflow. Per la gestione dell'infrastruttura, dopo aver valutato una soluzione custom, è stato scelto Actions Runner Controller, installato su un cluster Kubernetes statico. Lato workflow, è stato aggiunto un job probe, dal cui output dipende l'esecuzione del job di CI, per rilevare la presenza di un runner self-hosted immediatamente disponibile. Il sistema è stato testato in un contesto reale, applicandolo al repository Protelis-demo. In questa fase è stato verificato che il sistema soddisfacesse i requisiti funzionali definiti, ed è stato osservato come limite di questa validazione la mancanza di una misurazione quantitativa di parametri quali l'utilizzo delle risorse fisiche e la latenza di esecuzione.

Indice

Sommario **iii**

1 Introduzione **1**
1.1 DevOps . . . 2
1.2 Continuous Integration . . . 2
1.3 Pipeline come servizio . . . 3
1.4 Pipeline on-premise . . . 3
1.5 Motivazioni e obiettivi della tesi . . . 4
1.6 Struttura della tesi . . . 4

2 Background **7**
2.1 Tecnologie di CI . . . 7
2.1.1 GitHub Actions . . . 8
2.2 Virtualizzazione . . . 9
2.2.1 Proxmox . . . 9
2.3 Infrastructure as Code . . . 10
2.4 Kubernetes . . . 11
2.5 Actions Runner Controller . . . 12

3 Analisi **13**
3.1 Autoscaling . . . 13
3.1.1 Strategie di autoscaling . . . 13
3.2 Fallback su runner GitHub-hosted . . . 14
3.3 Requisiti . . . 14
3.3.1 Requisiti funzionali . . . 15
3.3.2 Requisiti non funzionali . . . 15

4 Design **17**
4.1 Runner self-hosted effimeri . . . 17
4.2 Autenticazione tramite GitHub App . . . 18
4.3 Soluzione custom . . . 18

4.3.1 Problema del registration token 18
4.3.2 Master come server API REST 19
4.3.3 Limiti della soluzione custom 20
4.4 Soluzione con Actions Runner Controller 21
4.4.1 Vantaggi di Actions Runner Controller 21
4.4.2 Limiti di Actions Runner Controller 22
4.5 Soluzione scelta lato infrastruttura 23
4.6 Fallback lato workflow 23
4.6.1 Limiti della soluzione basata sul job probe 24

5 Implementazione **27**
5.1 Cluster Kubernetes 27
5.1.1 Provisioning con Terraform 28
5.1.2 Configurazione con Ansible 29
5.2 Configurazione di ARC 31
5.3 Implementazione del fallback lato workflow 32
5.3.1 Action find-runner 32
5.3.2 Utilizzo nei workflow 34

6 Validazione **37**
6.1 Preparazione dell'ambiente di test 37
6.1.1 Modifiche al workflow 37
6.1.2 Configurazione delle credenziali 38
6.1.3 Creazione del runner scale set 39
6.2 Risultati 39
6.2.1 Test di trasparenza in assenza di credenziali 40
6.2.2 Test di fallback per assenza di runner self-hosted 40
6.2.3 Test di instradamento sull'infrastruttura self-hosted 41
6.2.4 Test di scaling 43
6.3 Discussione 44

7 Conclusione e sviluppi futuri **47**

49

Bibliografia **49**

Elenco delle figure

4.1 Architettura della soluzione custom: ciclo di rimpiazzo 20
4.2 Meccanismo di fallback con job probe 24

5.1 Cluster Kubernetes su infrastruttura Proxmox 28

6.1 Runner self-hosted registrati. 39
6.2 Esecuzione del job probe nel test senza credenziali. 40
6.3 Esecuzione del job probe nel secondo test 41
6.4 Esecuzione del job probe nel terzo test. 42
6.5 Ambiente di esecuzione del job build nel terzo test. 43
6.6 Runner self-hosted registrati durante l'esecuzione di un job. 43
6.7 Runner self-hosted registrati al termine dell'esecuzione. 44

Elenco dei listati

5.1 Installazione del control plane K3s 27
5.2 Installazione di K3s e aggiunta di un nodo al cluster 27
5.3 Installazione di Helm 28
5.4 Configurazione Terraform per il provisioning di una macchina virtuale 28
5.5 Playbook Ansible per l'aggiunta di un worker K3s 30
5.6 Creazione del secret Kubernetes per le credenziali di GitHub 31
5.7 Definizione degli input e dell'output dell'action find-runner 32
5.8 Verifica della configurazione delle credenziali 32
5.9 Generazione dell'installation access token 33
5.10 Verifica dei runner self-hosted disponibili 33
5.11 Selezione della label del runner 34
5.12 Utilizzo dell'action find-runner in un workflow 34
6.1 Workflow all'interno di Protelis-demo, job probe 37
6.2 Workflow all'interno di Protelis-demo, job di ci 38

Capitolo 1

Introduzione

Lo sviluppo software ha storicamente adottato processi organizzati in fasi sequenziali e rigide, riunite in un unico ciclo; queste fasi, che devono essere completate prima di poter proseguire alla successiva, includono la raccolta dei requisiti, l'analisi, la progettazione, l'implementazione, il collaudo e infine il rilascio. Questo approccio, se da un lato favorisce una pianificazione anticipata e dettagliata, dall'altro rende difficile adattarsi a requisiti che cambiano nel tempo, e comporta che eventuali errori di analisi o progettazione vengano scoperti solo in fasi avanzate, quando correggerli risulta più costoso [1]. È in questo contesto che nasce una metodologia di sviluppo software detta *Agile*, un approccio che mira a rendere questo processo più flessibile e che si fonda su quattro principi: individui e interazioni, versioni software funzionanti, collaborazione con il cliente e adattamento ai cambiamenti. Lo sviluppo viene quindi suddiviso in iterazioni brevi, ciascuna delle quali produce una versione funzionante del software, permettendo di raccogliere feedback dal cliente e di adattarsi ai cambiamenti [2]. Questa metodologia si è diffusa notevolmente, tanto da diventare uno standard in molte organizzazioni; tuttavia, nonostante lo sviluppo sia effettivamente diventato più rapido, non è stato rilevato un corrispondente aumento nella frequenza di rilascio del software. È infatti emerso un nuovo collo di bottiglia, dovuto alla mancanza di coordinazione tra il team di sviluppo e il team delle operazioni IT, ovvero quello responsabile dell'effettiva messa in produzione del software [3].

1.1 DevOps

Il termine *DevOps* indica un approccio che mira ad integrare due mondi: lo sviluppo software (*Development*) e le operazioni IT (*Operations*), attraverso un cambiamento organizzativo volto all'automazione e alla collaborazione; l'obiettivo principale è favorire un processo di sviluppo e rilascio continuo, rapido e affidabile, riducendo i problemi di comunicazione e accelerando la risoluzione degli errori [4]. La diffusione di questo approccio è cresciuta significativamente, tanto da essere considerato tra gli approcci maggiormente adottati nell'ambito IT; la letteratura evidenzia inoltre diversi benefici associati alla sua adozione, tra cui una maggiore collaborazione tra i team, una maggiore automazione e un miglioramento del *time-to-market* [5]. Tra le pratiche adottate nell'ambito DevOps e volte ad automatizzare le diverse fasi del ciclo di vita del software, riveste un ruolo centrale la *Continuous Integration*.

1.2 Continuous Integration

La Continuous Integration (CI) è una delle pratiche adottate nell'ambito DevOps ed è volta a facilitare e accelerare l'integrazione delle modifiche apportate da più sviluppatori attraverso frequenti *merge* del codice e verifiche automatizzate, con l'obiettivo di individuare il prima possibile eventuali errori. Ogni modifica viene quindi sottoposta a una serie di verifiche automatizzate, tipicamente comprendenti la compilazione del codice (*build*) e l'esecuzione dei test. Tali attività sono generalmente organizzate all'interno di una *pipeline*, che ne automatizza l'esecuzione in risposta a determinati eventi, come un *commit* o un *push*, e ne restituisce l'esito[1]. Eseguire una pipeline richiede quindi risorse computazionali dedicate, che possono essere messe a disposizione mediante due modelli: come servizio, gestito da un fornitore esterno, oppure *on-premise*, gestite dall'organizzazione stessa [6]. La CI si è affermata come una pratica rilevante nello sviluppo software; una revisione sistematica della letteratura ha analizzato 101 studi empirici, evidenziando prevalentemente effetti positivi della CI sul processo di sviluppo, tra cui un miglioramento della collaborazione e delle attività di controlla sulla qualità [7].

[1] `https://martinfowler.com/articles/continuousIntegration.html`

1.3 Pipeline come servizio

Il *cloud computing* è un modello di accesso *on-demand* a risorse computazionali, come server o storage, fornite tramite rete da un gestore [8]. Anche nel caso delle pipeline di CI esiste la possibilità di appoggiarsi su risorse in cloud, eseguendole come servizio secondo un modello di prezzo tipicamente a consumo. Adottare un approccio di questo tipo semplifica la configurazione e la gestione dell'infrastruttura rispetto a una soluzione on-premise, oltre ad evitare l'acquisto e la manutenzione di hardware dedicato[6]; inoltre, utilizzare un servizio in cloud offre maggiore scalabilità [9]. Tuttavia, un potenziale rischio è il *vendor lock-in*, ovvero la dipendenza da uno specifico fornitore causata dal costo necessario per effettuare una migrazione verso un servizio diverso [10]. Una soluzione in cloud risulta quindi particolarmente interessante per team che necessitano di scalabilità ma intendono ridurre l'onere associato alla gestione dell'infrastruttura [9].

1.4 Pipeline on-premise

Un modello alternativo prevede di eseguire le pipeline di CI su un'infrastruttura on-premise, ovvero gestita e mantenuta direttamente dall'organizzazione. Adottare un approccio di questo tipo ha il vantaggio di garantire prestazioni più stabili e prevedibili rispetto a una soluzione in cloud, oltre a risultare più conveniente, se utilizzata in modo intensivo [9]. Un'infrastruttura on-premise è anche più flessibile, e può essere modificata per adattarsi alle proprie esigenze specifiche; inoltre, permette di mantenere un maggiore controllo sull'ambiente di esecuzione, risultando più sicura nella gestione dei dati[6]. D'altra parte, questo approccio richiede all'organizzazione di occuparsi direttamente della configurazione e della manutenzione delle macchine utilizzate, con un conseguente aggravio in termini di costi e di impegno gestionale [6]. Una soluzione on-premise risulta quindi particolarmente adatta a team che necessitano di prestazioni stabili, e che sono disposti a sostenere l'onere della gestione infrastrutturale [9].

1.5 Motivazioni e obiettivi della tesi

Un'organizzazione che sceglie di adottare un'infrastruttura on-premise per ottenere prestazioni più stabili, un maggiore controllo sull'ambiente di esecuzione o un eventuale risparmio in termini di costi, si trova a fronteggiare due problemi pratici, che un servizio cloud risolve implicitamente. Il primo riguarda l'utilizzo delle risorse: mantenere attivo un numero fisso di macchine adibite all'esecuzione di pipeline porta ad uno spreco di risorse nei periodi in cui il carico di lavoro è basso o assente. Il secondo riguarda invece la latenza: utilizzando un'infrastruttura on-premise può accadere che in momenti di particolare carico non ci siano macchine disponibili per l'esecuzione di nuove pipeline, che sono costrette a rimanere in attesa per un tempo indefinito. Il presente progetto di tesi mira a risolvere questi problemi; gli obiettivi possono quindi essere riassunti in due punti:

1. ottimizzare l'utilizzo delle risorse on-premise attraverso una strategia di *autoscaling*, un meccanismo che consente di regolare dinamicamente le risorse computazionali disponibili in risposta alle variazioni del carico di lavoro [11];

2. minimizzare la latenza nell'esecuzione delle pipeline, problema che permane seppur mitigato dall'autoscaling, attraverso una strategia di fallback verso un servizio in cloud.

1.6 Struttura della tesi

Dopo aver introdotto il contesto, le motivazioni e gli obiettivi di questo lavoro di tesi, il resto del documento è organizzato come segue.

Il capitolo 2, Background, introduce i concetti e gli strumenti implementativi alla base del lavoro svolto, dalle piattaforme di CI/CD e virtualizzazione utilizzate agli strumenti di automazione dell'infrastruttura e orchestrazione.

Il capitolo 3, Analisi, definisce gli obiettivi della tesi in termini di requisiti funzionali e non funzionali.

Il capitolo 4, Design, presenta le soluzioni progettuali considerate, motivando le scelte effettuate attraverso un'analisi approfondita vantaggi e limitazioni.

Il capitolo 5, Implementazione, descrive nel dettaglio come sono state realizzate le soluzioni progettuali presentate nel capitolo precedente.

Il capitolo 6, Validazione, descrive la modalità usata per testare e validare il sistema.

Il capitolo 7, Conclusione e sviluppi futuri, riassume brevemente il lavoro svolto, giudicandolo in ottica di obiettivi raggiunti e limiti, ed elenca delle possibili direzioni di sviluppo futuro.

Capitolo 2

Background

In questo capitolo vengono introdotti concetti relativi a tecnologie di CI, virtualizzazione e Infrastracture as Code (IaC), con un approfondimento verticale sugli strumenti che verranno utilizzati nel progetto di tesi. Vengono successivamente accennati i concetti principali di Kubernetes, oltre a due tecnologie ad esso legate. Infine, una breve introduzione ad Actions Runner Controller (ARC) per spiegarne scopo e struttura.

2.1 Tecnologie di CI

Le tecnologie di CI sono strumenti che permettono di implementare, all'interno dei propri sistemi, i principi di CI sopra descritti, automatizzando processi come la compilazione del codice, l'esecuzione dei test e le verifiche di qualità, fornendo inoltre un riscontro sull'esito delle pipeline [12]. Alcuni di questi strumenti sono: GitLab e Azure DevOps, utilizzabili sia in cloud che on-premise; AWS CodePipeline, pensato per essere totalmente in cloud; GitHub Actions, anch'esso nativamente cloud ma configurabile con runner self-hosted gestiti dall'utente; e Jenkins, pensato esclusivamente per un utilizzo on-premise.

2.1.1 GitHub Actions

GitHub Actions[1], lanciato nel 2019, è rapidamente divenuto lo strumento di CI dominante per quanto riguarda i repository GitHub [13]. Il file di configurazione che contiene l'insieme delle istruzioni da eseguire, e che rappresenta la pipeline, è chiamato *workflow*: è in formato YAML e viene eseguito in risposta a determinati eventi. Un workflow è composto da uno o più *job*, unità di esecuzione indipendenti lanciate di default in parallelo; è tuttavia possibile introdurre dipendenze tra job, in modo che uno venga eseguito solo al completamento di un altro. Ogni job viene eseguito all'interno di un ambiente chiamato *runner*, ovvero una macchina tipicamente virtuale. Esistono due tipologie di runner: i *runner GitHub-hosted*, forniti direttamente da GitHub, e i *runner self-hosted*[2], gestiti dall'utente e ospitati all'interno della propria infrastruttura on-premise o in cloud. Gestire in autonomia i runner permette di avere più controllo sull'ambiente di esecuzione dei workflow, in questo modo è possibile, per esempio, creare configurazioni hardware o software custom che soddisfino al meglio i propri bisogni. Un runner self-hosted è costituito da un'applicazione installata sulla macchina che si intende utilizzare come runner. Il runner deve essere registrato a GitHub con un determinato *scope*, ovvero il livello a cui viene reso disponibile: un singolo repository, un'intera organizzazione, oppure, nel caso di GitHub Enterprise, l'intero account enterprise. La registrazione avviene fornendo all'applicazione un *registration token*, un token di breve durata ottenuto tramite una richiesta autenticata ad un determinato endpoint delle API di GitHub. Una volta registrato, il runner rimane in attesa di job da eseguire, richiedendoli attivamente a GitHub tramite polling. Per autenticarsi alle API di GitHub ci sono tre modi[3]: il primo consiste nell'utilizzo di un *personal access token*, un token associato a un account utente, a cui vengono assegnati permessi tramite degli scope. Il secondo consiste nello sfruttare il *GITHUB_TOKEN*, un token generato automaticamente da GitHub e reso disponibile all'interno di un workflow durante la sua esecuzione. Il terzo metodo consiste nell'utilizzare una *GitHub App*, un'entità installabile su un account utente o un organizzazione, alla quale vengono assegnati permessi granulari sulle risorse a cui può accedere; una volta installata,

[1] `https://docs.github.com/en/actions/get-started/understand-github-actions`
[2] `https://docs.github.com/en/actions/concepts/runners/self-hosted-runners`
[3] `https://docs.github.com/en/rest/authentication/authenticating-to-the-rest-api`

la GitHub App può generare dei token di breve durata, detti *installation access token*, utilizzando le sue credenziali, quali un *client id* ed una *private key*. A questo punto, includendo nella richiesta un installation access token, è possibile utilizzare gli endpoint protetti da autenticazione.

2.2 Virtualizzazione

La virtualizzazione è una tecnologia che permette di astrarre le risorse hardware o software sottostanti, come CPU, memoria, storage e rete, per creare uno o più ambienti virtuali isolati tra loro, nei quali possono essere eseguiti sistemi operativi e applicazioni [14]. Questa astrazione può avvenire a diversi livelli, dando origine a due approcci principali: la virtualizzazione hardware, in cui l'astrazione avviene tramite un hypervisor, e la virtualizzazione a livello di sistema operativo, in cui l'isolamento avviene condividendo il kernel dell'host [15]. L'hypervisor è il componente che si occupa di emulare l'hardware e garantire l'isolamento tra i vari ambienti virtuali; si divide in due categorie: il tipo 1, detto *bare metal*, viene eseguito direttamente sopra l'hardware in modalità privilegiata, agendo come un sistema operativo; il tipo 2, detto *hosted*, invece, viene eseguito su un sistema operativo esistente, non è quindi in contatto diretto con l'hardware [16]. Esistono diverse piattaforme per la gestione di ambienti virtualizzati, tra le principali si possono citare VMware vSphere[4], piattaforma per la gestione delle infrastrutture virtualizzate basate su ESXi, oVirt[5], soluzione open-source per la gestione distribuita della virtualizzazione, e OpenNebula[6], piattaforma open-source per la gestione di cloud e data center virtualizzati.

2.2.1 Proxmox

Proxmox Virtual Environment[7] è un ulteriore alternativa; si tratta di uno strumento open-source che permette di creare Macchine Virtuali (VM) e LXC container attraverso un'unica interfaccia web. È possibile installarlo su un singolo nodo

[4] `https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/8-0.html`
[5] `https://www.ovirt.org/community/about.html`
[6] `https://docs.opennebula.io/7.4/getting_started/overview/`
[7] `https://pve.proxmox.com/pve-docs`

oppure organizzare più nodi in un cluster, ottenendo una maggiore capacità di calcolo complessiva. Oltre all'interfaccia web, viene messa a disposizione un'API REST che consente di automatizzare la creazione, la configurazione e la rimozione di macchine virtuali e container senza necessità di intervento manuale. Proxmox permette inoltre di convertire una VM in un *template*, ovvero un'immagine di sola lettura, non avviabile direttamente, utilizzabile per creare rapidamente nuove VM; questo processo è detto clonazione. Utilizzando un template con *cloud-init*, uno strumento per configurare automaticamente una VM al primo avvio (hostname, chiavi SSH, indirizzo di rete), è possibile ottenere istanze pronte all'uso senza dover ripetere manualmente la configurazione per ognuna di esse.

2.3 Infrastructure as Code

IaC è un paradigma che si applica alla gestione dell'infrastruttura, definendone lo stato desiderato attraverso codice anziché tramite interventi manuali, permettendo quindi di versionare tale definizione e di ricreare l'infrastruttura in modo automatico e ripetibile [17]. Gli strumenti che implementano questo paradigma possono essere utilizzati per diverse attività, tra cui il *provisioning*, responsabile della creazione delle risorse, come le VM, e il *configuration management*, responsabile della configurazione del software all'interno delle risorse già esistenti. Un esempio di strumento per il provisioning IaC è Terraform[8], sviluppato da HashiCorp, che permette di definire l'infrastruttura desiderata tramite file di configurazione scritti in HCL (*HashiCorp Configuration Language*). Il concetto centrale di Terraform è la *resource*, ovvero un qualsiasi oggetto infrastrutturale da creare e gestire. L'interazione con la piattaforma su cui tali risorse vengono effettivamente create è demandata a un *provider*, un plugin specifico per ciascuna piattaforma, che utilizza le API messe a disposizione da quest'ultima. Invece, un esempio di strumento per il configuration managment è Ansible[9], sviluppato da Red Hat, utilizzato per automatizzare la configurazione di sistemi già esistenti. È *agent-less*, cioè si collega da remoto ai nodi che gestisce, tramite protocollo SSH, senza la necessità che su questi sia installato software particolare. Il file che contiene le operazioni da

[8] `https://developer.hashicorp.com/terraform/intro`
[9] `https://docs.ansible.com/ansible/latest/getting_started/index.html`

eseguire sui nodi è detto *playbook*; si tratta di un file in formato YAML composto da una serie di *task*, ciascuno dei quali rappresenta un'operazione da eseguire.

2.4 Kubernetes

Kubernetes[10] è un progetto open source per l'orchestrazione di container, che permette di automatizzare il deployment, lo scaling e la gestione di applicazioni containerizzate. Un cluster Kubernetes è composto da un *control plane*, responsabile delle decisioni relative allo stato del cluster, e da uno o più nodi *worker*, macchine fisiche o virtuali che si occupano dell'esecuzione vera e propria. L'unità di base eseguita su un nodo è il *pod*, che incapsula uno o più container in esecuzione, condividendone rete e storage. Tra i componenti di Kubernetes, quello che merita di essere approfondito per lo scopo di questa tesi è il Cluster Autoscaler[11], il componente dedicato allo scaling dei nodi di un cluster. I suoi compiti principali sono aggiungere nuovi nodi quando alcuni pod non possono essere pianificati per mancanza di risorse disponibili, e rimuovere nodi sottoutilizzati quando i pod in esecuzione su di essi possono essere ricollocati altrove. Per poter effettuare queste operazioni, è richiesta un'integrazione con l'infrastruttura sottostante, in grado di creare ed eliminare le istanze, tipicamente VM, che fungono da nodi del cluster.

Tra le varie distribuzioni Kubernetes, K3s[12], sviluppata da Rancher Labs, è pensata per essere leggera e semplice da configurare. A differenza di un'installazione standard, K3s incapsula tutti i componenti del control plane in un singolo binario, facilitando e velocizzando il processo di installazione. Nonostante questo, rimane pienamente conforme alle specifiche Kubernetes, garantendo la compatibilità con gli strumenti e le applicazioni dell'ecosistema standard.

Uno dei principali gestore di pacchetti per Kubernetes, volto a semplificare l'installazione e la gestione di applicazioni all'interno di un cluster, è Helm[13]. Le applicazioni distribuite tramite Helm sono impacchettate in un *chart*, una collezione di file che descrive le risorse Kubernetes necessarie alla sua esecuzione. Tra i file

[10] `https://kubernetes.io/docs/concepts/overview/`
[11] `https://kubernetes.io/docs/concepts/cluster-administration/node-autoscaling/`
[12] `https://docs.k3s.io/`
[13] `https://helm.sh/docs/`

che compongono un chart, il file `values.yaml` contiene i valori di configurazione utilizzati nei template, che possono essere sovrascritti al momento dell'installazione per personalizzare il comportamento di un'applicazione.

2.5 Actions Runner Controller

ARC[14] è un operatore Kubernetes mantenuto ufficialmente da GitHub che scala dinamicamente runner self-hosted come pod all'interno di un cluster. ARC è distribuito tramite due Helm chart distinti: `gha-runner-scale-set-controller`, che installa il controller responsabile della gestione dei runner scale set, e `gha-runner-scale-set`, che definisce un singolo runner scale set, ovvero un gruppo di runner associato a un repository, un'organizzazione o un enterprise.

[14] `https://docs.github.com/en/actions/concepts/runners/actions-runner-controller`

Capitolo 3

Analisi

3.1 Autoscaling

Come menzionato nel capitolo 1 di Introduzione, il primo obiettivo prefissato per questa tesi è l'ottimizzazione delle risorse on-premise, messe a disposizione da un cluster Proxmox, attraverso una strategia di autoscaling. In particolare l'obiettivo è mantenere sempre disponibile un numero prestabilito di runner, aggiungendo e rimuovendo istanze in base al carico di lavoro rilevato.

3.1.1 Strategie di autoscaling

In letteratura, le strategie di autoscaling possono essere classificate in due criteri[11]:

1. orizzontale, verticale o ibrido, cioè scalare aggiungendo o rimuovendo nuove istanze oppure aumentando o diminuendo la capacità di un numero fisso di istanze già esistenti. La tipologia ibrida è una combinazione dei due approcci.

2. Reattivo, proattivo o ibrido, cioè in base al criterio con cui viene effettuata la scelta di scalare: in un sistema reattivo questa viene fatta in risposta ad eventi già accaduti, ad esempio il superamento di una soglia nell'utilizzo di una data risorsa. In un sistema proattivo, invece, la scelta viene presa cercando di prevedere il carico futuro, agendo in anticipo. Anche in questo caso per ibrido si intende una combinazione dei due criteri.

La strategia che viene adottata in questa tesi è quindi classificabile come orizzontale e reattiva. Una limitazione importante per questo tipo di autoscaling è detta *scaling delay*: si tratta di un peggioramento nelle prestazioni che si può verificare nell'intervallo di tempo tra la rilevazione dell'aumento del carico di lavoro e l'effettiva disponibilità delle nuove istanze [11].

3.2 Fallback su runner GitHub-hosted

Il secondo obiettivo menzionato nel capitolo 1 di Introduzione, riguarda la latenza di esecuzione dei job: utilizzando un numero fisso di runner self-hosted può accadere che, in momenti di carico particolarmente elevato, un job sia costretto a rimanere in coda per un tempo indefinito. Dato che anche in questo caso il problema deriva dall'utilizzo di un numero fisso di runner, adottare una strategia di autoscaling può sembrare una soluzione; tuttavia, come osservato nella sezione precedente, una strategia orizzontale e reattiva, pur mitigando il problema, ha il limite dello scaling delay. Nel contesto della tesi questo si traduce in un aumento della latenza di un job, non causato dall'attesa che un runner si liberi, ma dal tempo necessario per crearne uno nuovo; questo tempo, detto di provisioning, può variare sensibilmente a seconda di quale tipo di infrastruttura si è scelto per ospitare i runner. Per minimizzare la latenza di un job, risulta quindi utile adottare in aggiunta all'autoscaling, una strategia di fallback sui runner GitHub-hosted; ovvero un meccanismo per rilevare l'assenza di runner self-hosted immediatamente disponibili e dirottare di conseguenza il job verso un runner GitHub-hosted.

3.3 Requisiti

A partire dagli obiettivi definiti nell'introduzione e analizzati approfonditamente in questo capitolo, è possibile derivare un insieme di requisiti funzionali e non funzionali che il sistema deve soddisfare.

3.3.1 Requisiti funzionali

1. Il sistema deve mantenere sempre attivo un pool minimo di runner self-hosted, creandone di nuovi quando questi vengono utilizzati e rimuovendo quelli non più necessari.

2. Il sistema deve implementare un meccanismo per rilevare l'assenza di runner self-hosted immediatamente disponibili ed effettuare un fallback sui runner GitHub-hosted.

3.3.2 Requisiti non funzionali

1. Il sistema deve minimizzare la latenza introdotta dallo scaling delay, riducendo il tempo che un job trascorre in attesa prima di essere eseguito.

2. Il sistema deve utilizzare le risorse fisiche dell'infrastruttura Proxmox in modo efficiente.

Capitolo 4

Design

Il sistema si articola nel complesso di due componenti indipendenti tra loro: uno lato infrastruttura e uno lato workflow. Il primo componente, lato infrastruttura, è responsabile dell'autoscaling dei runner self-hosted; il secondo invece, lato workflow, è quello che si occuperà del meccanismo di fallback descritto in precedenza. Le seguenti sezioni partono descrivendo le scelte effettuate per quanto riguarda il lato infrastruttura, comprendendo anche l'analisi delle due soluzioni considerate, per passare poi al lato workflow.

4.1 Runner self-hosted effimeri

La prima scelta di desing effettuata riguarda l'utilizzo di runner persistenti o effimeri[1]. Il normale lifecycle di un runner è registrarsi a GitHub, rimanere in attesa di un job, eseguirlo quando arriva, e tornare in uno stato di attesa. La classica tipologia di runner è quindi detta persistente. GitHub mette però a disposizione un'altra modalità: i runner effimeri. In questo caso il runner al termine dell'esecuzione di un job non rimane in attesa del prossimo ma si deregistra automaticamente, diventando nella pratica un runner usa e getta. Il vantaggio principale è quello di mettere a disposizione del job sempre un ambiente pulito e nuovo, evitando residui di job precedenti. Questo aspetto è quindi particolarmente rilevante in termini di sicurezza e isolamento. Inoltre, è GitHub stessa a promuoverne l'utilizzo in pre-

[1] `https://docs.github.com/en/actions/reference/runners/self-hosted-runners`

senza di una strategia di autoscaling. Per queste ragioni, in questa tesi è stato deciso di utilizzare i runner effimeri.

4.2 Autenticazione tramite GitHub App

Per registrare un nuovo runner è necessario utlizzare un metodo di autenticazione. È stato scelta una GitHub App rispetto ad un Personal Access Token per vari motivi: un PAT ha una durata di vita molto più lunga rispetto agli installation token utlizzati da una GitHub App, aumentando il rischio in caso di compromissione della credenziale; un PAT è legato a un account utente specifico, al contrario di una GitHub App che è un'entità autonoma; infine, i permessi di un PAT sono meno granulari di quelli che è possibile dare ad una GitHub App. Ovviamente l'utilizzo di un GITHUB_TOKEN non è stato preso in considerazione in quanto il suo scope è solo quello del repository nel quale il workflow è contenuto.

4.3 Soluzione custom

Inizialmente, lato infrastruttura, è stata vagliata l'idea di sfruttare direttamente le API di Proxmox e realizzare un sistema totalmente custom. Questa prima soluzione prevedeva il mapping di un runner in una VM, quindi utilizzare una VM *master* che tramite l'utilizzo delle API Proxmox si occupasse di creare ed eliminare le macchine virtuali runner. Queste ultime sarebbero state create a partire da un template precedentemente configurato.

4.3.1 Problema del registration token

Come menzionato in precedenza, un runner necessita di un registration token per potersi registrare correttamente a GitHub. Una nuova VM, generata a partire dal template, avrebbe quindi avuto bisogno di accedere in qualche modo ad uno di questi token. La prima soluzione a cui si può pensare è quella di inserire il token direttamente nel template, così che la VM lo abbia immediatamente disponibile. Tuttavia, oltre a non essere una buona pratica, questa strada non risolve il problema in quanto il registration token non è permanente ma ha una durata

di vita limitata. Una soluzione simile prevede di lasciare che sia direttamente la VM appena generata a fare richiesta alle API di GitHub per ottenere un token. Tuttavia, questo significherebbe inserire nel template le credenziali della GitHub App utilizzata per l'autenticazione, generando un importante vulnerabilità di sicurezza visto che tali credenziali non sono temporanee. Altra strada prevedeva l'uso di *cloud-init*, ma è stata anche questa scartata poiché richiede uno snippet già presente sul nodo fisico.

4.3.2 Master come server API REST

Per risolvere questo problema è stato pensato di rendere la VM master un server API REST e, al contrario di quanto pensato inizialmente, lasciare il compito di gestire il proprio lifecycle ai runner piuttosto che al master. Con una soluzione di questo tipo, l'API avrebbe dovuto mettere a disposizione tre endpoint come: GET `/config`, POST `/create-runner` e DELETE `/destroy-runner`. Una VM runner appena clonata avrebbe fatto richiesta ad un endpoint `/config` per ricevere un registration token e potersi registrare a GitHub. Non appena le fosse stato assegnato un job avrebbe notificato il master tramite un endpoint `/create-runner`, così che venisse generata una nuova VM in sostituzione di quella appena occupata, mantenendo il pool di runner attivi. Terminata l'esecuzione del job, e deregistratasi come runner da GitHub, la VM avrebbe infine notificato il master tramite un endpoint `/destroy-runner`, richiedendo la propria eliminazione.

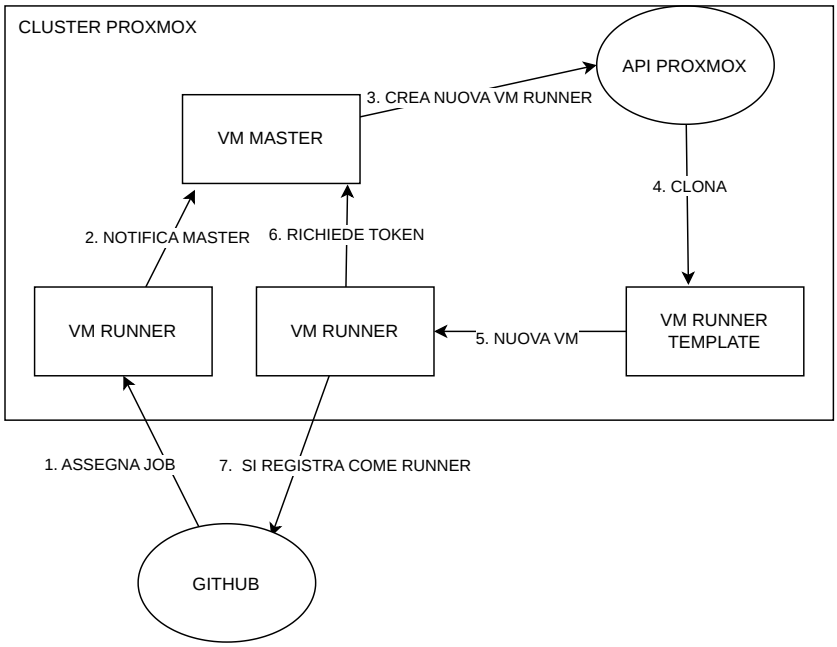


Figura 4.1: Architettura della soluzione custom: ciclo di rimpiazzo

4.3.3 Limiti della soluzione custom

Il primo limite da prendere in considerazione è relativa alla gestione del registration token; in questa soluzione infatti chiunque abbia accesso alla rete privata del cluster Proxmox potrebbe potenzialmente ottenerne uno. Un ipotetico attaccante potrebbe quindi registrare una sua macchina come runner legittimo, eseguendo qualsiasi job gli venga assegnato. Questo potrebbe esporre il codice sorgente, eventuali segreti configurati nel repository o nell'organizzazione e il GITHUB_TOKEN associato al job[2]. Nonostante la rete privata sia stata considerata una difesa accettabile nel contesto di questa tesi, è comunque un limite architetturale da segnalare. Il secondo limite riguarda in generale il costo di sviluppo di un sistema del genere, infatti, oltre all'implementazione della logica già descritta, ci sono altri aspetti da affrontare: la gestione di errori e casi limite, come il fallimento di una richiesta verso l'API del master o verso l'API di Proxmox, l'assenza di uno strumento per monitorare lo stato del sistema e i log di eventuali errori, che andrebbe imple-

[2] `https://docs.github.com/en/actions/reference/security/secure-use`

mentato separatamente. Il terzo limite riguarda invece la manutenibilità: essendo interamente sviluppato ad hoc, l'API REST del master e il template delle VM runner, deveno essere mantenute autonomamente.

4.4 Soluzione con Actions Runner Controller

La seconda soluzione analizzata prevede l'adozione di ARC, uno strumento che, a differenza della soluzione custom, non scala interi runner ospitati in VM, ma pod Kubernetes. Una soluzione di questo tipo per essere dispiegata richiede un cluster Kubernetes, da costruire quindi sopra al cluster Proxmox messo a disposizione per questa tesi. Per la sua realizzazione è stata scelta la distribuzione K3s al posto di Kubernetes standard, per il suo minore overhead in termini di risorse richieste, oltre che per la maggiore facilità di installazione. Come nodi (statici) del cluster, è stato deciso di utilizzare delle VM invece di container LXC. Sebbene questi ultimi risultino generalmente più efficienti nell'uso delle risorse fisiche rispetto a una VM, condividendo il kernel del nodo fisico[3], la loro adozione come nodi del cluster Kubernetes avrebbe richiesto configurazioni particolari, sia a livello del container sia del nodo fisico Proxmox, necessarie per garantire il corretto funzionamento di Kubernetes.

4.4.1 Vantaggi di Actions Runner Controller

Questa soluzione permette di superare i limiti individuati nella soluzione custom: non dovendo reimplementare nulla da zero, il costo di sviluppo è minore, ed è possibile usufruire dei sistemi di monitoraggio offerti da Kubernetes. La manutenzione ricade direttamente su GitHub e il problema del registration token non si pone, poiché la registrazione avviene internamente senza esporre nulla sulla rete privata. Un ulteriore punto a favore è il tempo di provisioning dei runner; se nella prima soluzione questo è condizionato dalla clonazione e dall'avvio di una VM, in questa soluzione si traduce nel tempo necessario per creare e avviare un pod Kubernetes.

[3] `https://pve.proxmox.com/wiki/Linux_Container`

4.4.2 Limiti di Actions Runner Controller

Il primo limite di ARC riguarda il livello su cui opera: infatti la sua capacità di scaling è relativa ai pod, ma non ai nodi. Solitamente questo compito è affidato al *cluster autoscaler*, il componente ufficiale del progetto Kubernetes dedicato allo scaling automatico dei nodi di un cluster[4]. Tuttavia, mentre per i principali cloud provider esistono integrazioni mature e ufficialmente supportate, per un ambiente on-premise come Proxmox non è stato individuato uno strumento altrettanto maturo: le soluzioni disponibili richiedono l'adozione di ulteriori livelli di infrastruttura, come Cluster API, oppure si affidano a progetti di terze parti ancora sperimentali. Questo significa che la capacità totale di risorse è limitata a quella allocata al cluster Kubernetes, senza la possibilità di scalare qualora il carico di lavoro la eccedesse. D'altra parte, allocare al cluster Kubernetes una quantità di risorse eccessivamente ampia comporterebbe una sottrazione di risorse fisiche al resto del cluster Proxmox, vanificando parte dei benefici dell'autoscaling stesso. Il secondo limite riguarda l'esecuzione di job che fanno uso di container, ad esempio usando comandi Docker nei loro step, oppure richiedendo di essere eseguiti all'interno di uno specifico container. Essendo il runner stesso un pod, esso non dispone di un ambiente per eseguire tali container. Per ovviare a questo problema, ARC mette a disposizione principalmente due modalità[5]. La prima prevede di eseguire, all'interno di ogni pod, un ulteriore container aggiuntivo a quello del runner vero e proprio, configurato per il *Docker-in-Docker*. Questa opzione è la più flessibile e permette l'esecuzione di qualsiasi tipo di job, anche quelli che utilizzano internamente comandi Docker. Gli svantaggi sono la necessità di eseguire container privilegiati e un maggiore consumo di risorse, dato dal container aggiuntivo che viene sempre eseguito, a prescindere dal fatto che serva realmente. Nella seconda modalità, quando un job richiede l'utilizzo di un container, ARC sfrutta le API di Kubernetes per creare un pod dedicato a tale esecuzione. Rispetto alla prima modalità, questa evita la necessità di container privilegiati ed è più efficiente, creando il pod dedicato solo se effettivamente necessario; richiede tuttavia di assegnare al runner permessi elevati sul cluster per poter creare e gestire pod. Un ulterio-

[4] `https://kubernetes.io/docs/concepts/cluster-administration/node-autoscaling`

[5] `https://docs.github.com/en/actions/tutorials/use-actions-runner-controller/deploy-runner-scale-sets`

re svantaggio è la minore flessibilità: questa modalità non supporta l'esecuzione di comandi Docker direttamente all'interno del job. Il terzo limite, più generale, riguarda l'adozione di Kubernetes in sé, che introduce un livello di complessità aggiuntivo rispetto all'infrastruttura Proxmox sottostante, dato dalle competenze necessarie per dispiegarlo.

4.5 Soluzione scelta lato infrastruttura

Analizzando le due soluzioni, e confrontando i relativi benefici e limiti, è stato scelto di utilizzare, lato infrastruttura, la soluzione che adotta ARC. In particolare, è stato ritenuto che i vantaggi in termini di costo di sviluppo e manutenibilità giustificassero i limiti individuati. Nella scelta finale, ha avuto un ruolo determinante la possibilità di mitigare il limite principale di questa soluzione, ovvero il primo descritto nella sezione precedente riguardante l'utilizzo delle risorse del nodo fisico, a causa della mancanza di un cluster autoscaler. Questo è possibile grazie al comportamento dinamico che ha Proxmox nell'allocazione delle risorse; per la CPU, l'overcommit viene gestito in autonomia. Per la RAM, invece, entra in gioco il *memory ballooning*[6], un meccanismo che dà la possibilità al nodo fisico, solo quando è sottopressione, di reclamare parte della memoria inutilizzata dalla VM. Inoltre, è stato pensato di integrare questa soluzione con un approccio IaC per velocizzare un eventuale espansione del cluster Kubernetes aggiungendo un nodo, pur richiedendo un intervento manuale. In particolare è stato deciso di utilizzare Terraform, per il provisioning di un nuovo nodo, e Ansible per l'aggiunta automatica del nodo al cluster. Per quanto riguarda invece la gestione dei job che fanno uso di container, è stata preferita la prima modalità, per la maggiore flessibilità che offre.

4.6 Fallback lato workflow

Per progettare un meccanismo di fallback è necessario agire precedentemente all'instradamento del job verso il runner, quindi direttamente nel workflow. Questo

[6] `https://pve.proxmox.com/wiki/Dynamic_Memory_Management`

è dato da un vincolo di GitHub Actions: il campo `runs-on` di un job, cioè quello che specifica dove eseguirlo, viene valutato staticamente nel momento in cui il job viene messo in coda, e non esiste nativamente un modo per reindirizzarlo nel caso in cui il runner target non sia disponibile. Da qui nasce la necessità di conoscere in anticipo la destinazione verso cui si vorrà instradare il job. La soluzione adottata prevede la creazione di un job *probe* separato, ovvero un job che rilevi se un runner self-hosted è disponibile e mandi in output una stringa che identifichi quale tipologia di runner utilizzare. Questo job deve poter essere eseguito a prescindere dallo stato dell'infrastruttura on-premise, quindi su un runner GitHub-hosted. Il job di CI imposta una dipendenza verso il job probe, così da aspettare il termine dell'esecuzione di quest'ultimo, e utilizza nel proprio campo `runs-on` il suo output.

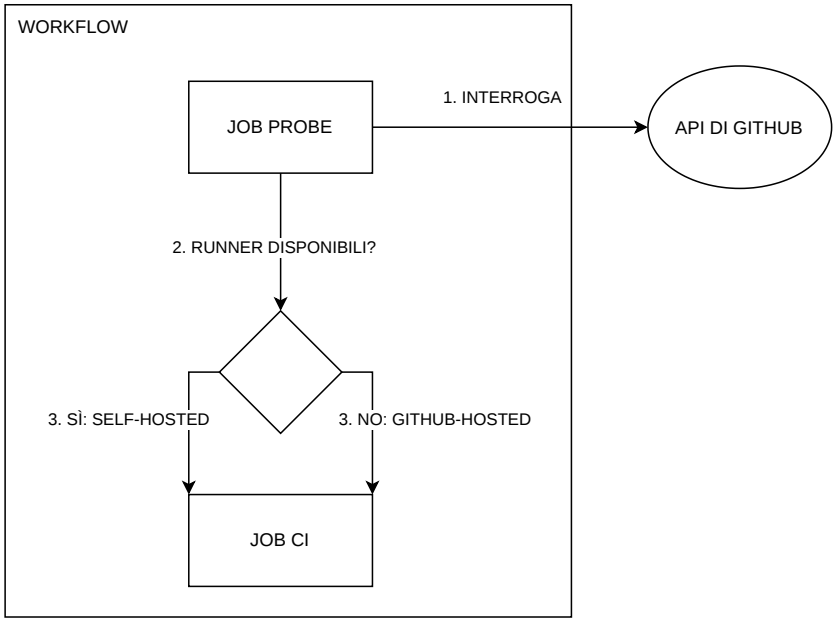


Figura 4.2: Meccanismo di fallback con job probe

4.6.1 Limiti della soluzione basata sul job probe

Il primo limite della soluzione adottata riguarda il fatto che il job probe, pur rilevando un runner self-hosted disponibile, non può riservarlo. Questo introduce una

possibile *race condition*: infatti, può succedere che dopo aver rilevato la disponibilità di un runner, nel tempo che intercorre tra il termine della sua esecuzione e l'effettiva assegnazione del job di CI al runner, quest'ultimo venga occupato da un altro job. Va tuttavia notato che, anche nel caso in cui si verifichi questa race condition, l'esito peggiore per il job è di rimanere in coda in attesa del provisioning di un nuovo runner self-hosted, ovvero lo stesso scenario che si presenterebbe in assenza del meccanismo di fallback. Il secondo limite è intrinseco alla necessità di calcolare in anticipo la destinazione del job, e riguarda l'esecuzione stessa del job probe, che introduce una latenza aggiuntiva al job di CI. Tale latenza risulta tuttavia contenuta, dato che il job probe non contiene operazioni computazionalmente onerose e il tempo di avvio di un job su un runner GitHub-hosted è generalmente ridotto.

Capitolo 5

Implementazione

In questo capitolo si descrive come sono state implementate le due soluzioni, lato infrastruttura e lato workflow, spiegate nel capitolo precedente; cominciando con il lato infrastrutturale.

5.1 Cluster Kubernetes

La prima operazione che è stata effettuata è la creazione di un cluster Kubernetes, su cui poi installare ARC. Dato che il cluster Proxmox messo a disposizione per questa tesi è composto da tre nodi fisici, è stato pensato di comporre il cluster Kubernetes di tre nodi, ognuno su un diverso nodo fisico, così da distribuire il carico riducendo l'impatto sui singoli nodi. Dopo aver creato la prima VM, abilitando il memory ballooning, è stato installato Kubernetes utilizzando la distribuzione K3s. Questo nodo assume quindi il ruolo di control plane del cluster.

Listing 5.1: Installazione del control plane K3s

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

Il processo di creazione delle VM è stato ripetuto anche per gli altri due nodi; tuttavia, in questo caso il comando di installazione di K3s diverge rispetto al precedente, poiché questi nodi devono unirsi a un cluster già esistente.

Listing 5.2: Installazione di K3s e aggiunta di un nodo al cluster

```
curl -sfL https://get.k3s.io | K3S_URL=https://${IP_ADDRESS}:6443 K3S_TOKEN=${
    TOKEN} sh -
```

Si noti che il comando sopra deve essere riempito con l'indirizzo IP del control plane e con il token, ovvero la credenziale necessaria per unirsi al cluster, ricavabile dal control plane. Infine, sulla macchina control plane, è stato installato Helm, dato che ARC viene distribuito tramite Helm chart.

Listing 5.3: Installazione di Helm

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

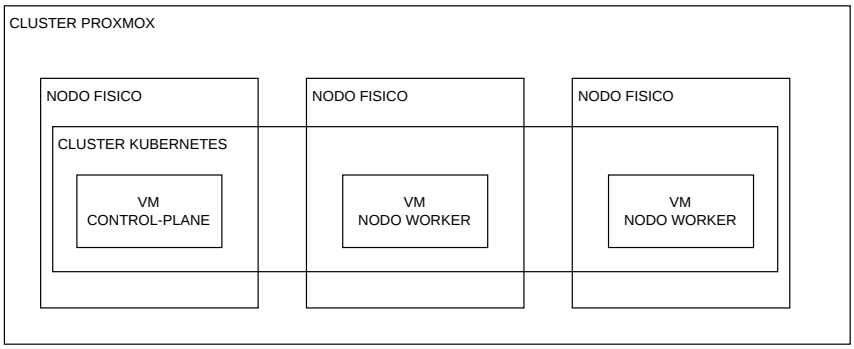


Figura 5.1: Cluster Kubernetes su infrastruttura Proxmox

5.1.1 Provisioning con Terraform

Il provisioning di una nuova VM, da aggiungere come nodo del cluster K3s, è automatizzato tramite Terraform, utilizzando il provider `bpg/proxmox` per interfacciarsi con le API di Proxmox. La nuova VM viene clonata a partire da un template preesistente, configurato con cloud-init. Tra i parametri configurabili figurano il nodo fisico Proxmox su cui creare la VM, il numero di core e la quantità di memoria assegnati, e la chiave pubblica SSH da associare all'utente creato tramite cloud-init, necessaria per la successiva fase di configurazione con Ansible.

Listing 5.4: Configurazione Terraform per il provisioning di una macchina virtuale

```
resource "proxmox_virtual_environment_vm" "k3s-worker" {
  name      = var.vm_hostname
  node_name = var.target_node

  agent {
    enabled = false
  }
```

```
8
9   clone {
10    vm_id = var.template_id
11    full  = true
12    node_name = var.template_node
13  }
14
15  disk {
16    interface = "scsi0"
17    size      = var.vm_disk_size
18  }
19
20  cpu {
21    cores = var.vm_cpu_cores
22  }
23
24  memory {
25    dedicated = var.vm_memory
26  }
27
28  network_device {
29    bridge      = "vmbr0"
30    mac_address = var.vm_mac_address
31  }
32
33  initialization {
34    ip_config {
35      ipv4 {
36        address = "dhcp"
37      }
38    }
39
40    user_account {
41      username = "ubuntu"
42      keys     = [var.ssh_public_key]
43    }
44  }
45 }
```

5.1.2 Configurazione con Ansible

Una volta creata, la nuova VM viene unita al cluster K3s esistente tramite un playbook Ansible. Il playbook installa K3s in modalità *agent*[1], ovvero come nodo worker, utilizzando lo stesso script di installazione ufficiale già impiegato per il

[1] https://docs.k3s.io/cli/agent

control plane, a cui vengono però passati l'indirizzo del control plane e il token di join come variabili d'ambiente. Prima di procedere con l'installazione, il playbook verifica se K3s sia già presente sull'host: questo rende l'esecuzione idempotente, permettendo di eseguirlo più volte senza effetti indesiderati.

Listing 5.5: Playbook Ansible per l'aggiunta di un worker K3s

```
- name: Join VM to existing k3s cluster as worker (agent)
  hosts: new_nodes
  become: true

  tasks:
    - name: Check required variables are provided
      assert:
        that:
          - k3s_url is defined
          - k3s_token is defined
        fail_msg: "Pass -e k3s_url=... -e k3s_token=... when running this playbook
            "

    - name: Check if k3s-agent is already installed
      stat:
        path: /usr/local/bin/k3s
      register: k3s_installed

    - name: Download and install k3s agent, joining it to the master
      shell: |
        curl -sfL https://get.k3s.io | \
        K3S_URL="{{ k3s_url }}" \
        K3S_TOKEN="{{ k3s_token }}" \
        sh -
      when: not k3s_installed.stat.exists
      register: k3s_install_result

    - name: Make sure k3s-agent service is running and enabled
      systemd:
        name: k3s-agent
        state: started
        enabled: true

    - name: Show install result (if it ran)
      debug:
        msg: "{{ k3s_install_result.stdout_lines | default('k3s was already
            installed, nothing to do') }}"
```

5.2 Configurazione di ARC

ARC è stato installato tramite i due Helm chart menzionati in precedenza: `gha-runner-scale-set-controller`, per il controller, e `gha-runner-scale-set`, che definisce un runner scale set. Per il primo chart, sono stati ritenuti adeguati i valori di default impostati dal team di GitHub, e non è stata quindi apportata alcuna modifica al file `values.yaml`. Per il secondo chart, invece, sono stati configurati alcuni campi secondo le scelte già discusse nel capitolo precedente; le configurazioni più rilevanti sono le seguenti.

Il campo `containerMode`, relativo alla modalità con cui ARC gestisce l'esecuzione di container all'interno dei job, è stato impostato a *dind*, in seguito all'analisi effettuata nella parte di Design.

Il campo `githubConfigSecret`, relativo alle credenziali utilizzate da ARC per autenticarsi alle API di GitHub, è stato impostato con il nome di un Kubernetes *secret* precedentemente creato, contenente le credenziali della GitHub App descritta nel capitolo precedente; questo evita di dover scrivere tali credenziali in chiaro all'interno del file `values.yaml`.

Listing 5.6: Creazione del secret Kubernetes per le credenziali di GitHub

```
sudo kubectl create secret generic github-app-authentication \
--namespace=arc-runners \
--from-literal=github_app_id="" --from-literal=github_app_installation_id="" \
--from-literal=github_app_private_key="$(cat path/to/PrivateKey.pem)"
```

Si noti che, nel comando riportato, `github-app-authentication` è un nome arbitrario assegnato al segreto, mentre `arc-runners` è il Kubernetes *namespace* in cui è stato installato il secondo chart.

Sono stati inoltre configurati i campi `minRunners` e `maxRunners`, che rappresentano rispettivamente il pool minimo di runner da mantenere attivi e il tetto massimo per lo scaling. Il campo `maxRunners`, dato che il cluster è adibito solo ad ARC, è stato lasciato senza un valore, così da scalare fino all'esaurimento delle risorse.

5.3 Implementazione del fallback lato workflow

Il job probe è stato implementato come *composite action*, pubblicata in un repository separato, così da avere un componente riutilizzabile ed evitare di doverlo scrivere per intero all'interno di ogni workflow. Le due seguenti sottosezioni documentano l'action `find-runner`, ovvero il job probe, e il suo utilizzo in un workflow.

5.3.1 Action find-runner

Per prima cosa sono stati definiti input e output dell'action; in particolare, oltre al nome dell'organizzazione, è necessario fornire due credenziali, approfondite in seguito. L'output è una label che identifica i runner self-hosted oppure quelli GitHub-hosted, anche questo aspetto è approfondito più avanti.

Listing 5.7: Definizione degli input e dell'output dell'action find-runner

```yaml
inputs:
  org:
    description: 'The organization to check for self-hosted runners'
    required: true
  app-client-id:
    description: 'The GitHub App client ID'
    required: true
  app-private-key:
    description: 'The GitHub App private key'
    required: true

outputs:
  runner_label:
    description: 'Label to use in runs-on'
    value: ${{ steps.runner-label.outputs.runner_label }}
```

Il primo step verifica se le credenziali sono state passate correttamente in input e, in caso contrario, permette di saltare i due step successivi, che altrimenti provocherebbero il fallimento del workflow, arrivando direttamente all'ultimo. Questo step ha lo scopo di facilitare l'integrazione dell'action in un workflow già esistente, permettendo di farlo anche prima che le credenziali siano state configurate a livello di organizzazione, senza comprometterne l'esecuzione.

Listing 5.8: Verifica della configurazione delle credenziali

```yaml
- name: Check if self-hosted config is present
```

```
  id: check-config
  shell: bash
  run: |
    if [ -n "${{ inputs.app-client-id }}" ]; then
      echo "configured=true" >> "$GITHUB_OUTPUT"
    else
      echo "configured=false" >> "$GITHUB_OUTPUT"
    fi
```

Nel secondo step si utilizzano le credenziali fornite per generare un installation access token, tramite l'action `create-github-app-token`.

Listing 5.9: Generazione dell'installation access token

```
- name: Generate an installation token
  id: generate-token
  if: steps.check-config.outputs.configured == 'true'
  uses: actions/create-github-app-token@v3
  with:
    client-id: ${{ inputs.app-client-id }}
    private-key: ${{ inputs.app-private-key }}
```

Per verificare la presenza di un runner self-hosted disponibile, viene effettuata una chiamata alle API di GitHub, autenticata tramite l'installation access token generato nello step precedente, che restituisce l'elenco di tutti i runner registrati per l'organizzazione fornita in input. Tale elenco viene poi filtrato per contare i soli runner online e non occupati, e il numero risultante viene salvato per essere utilizzato nello step successivo.

Listing 5.10: Verifica dei runner self-hosted disponibili

```
- name: Call to GitHub API
  id: api-call
  if: steps.check-config.outputs.configured == 'true'
  shell: bash
  env:
    GH_TOKEN: ${{ steps.generate-token.outputs.token }}
  run: |
    ACTIVE_RUNNERS=$( gh api \
      -H "Accept: application/vnd.github+json" \
      -H "X-GitHub-Api-Version: 2026-03-10" \
      /orgs/${{ inputs.org }}/actions/runners --jq '[.runners[] | select(.status
          == "online" and .busy == false)] | length')
    echo "active_runners=$ACTIVE_RUNNERS" >> "$GITHUB_OUTPUT"
- name: Write runner-label
```

Infine, sulla base del numero di runner disponibili rilevato nello step precedente, viene impostato l'output dell'action: se il numero di runner disponibili è maggiore di zero, viene restituita la label del runner scale set gestito da ARC; in tutti gli altri casi, incluso quello in cui le credenziali non risultino configurate, viene restituita la label di un runner GitHub-hosted standard.

Listing 5.11: Selezione della label del runner

```
- name: Write runner-label
  id: runner-label
  shell: bash
  run: |
    if [ "${{ steps.check-config.outputs.configured }}" == "true" ] && (( ${{
        steps.api-call.outputs.active_runners || 0 }} > 0 ))
    then
      echo "runner_label=self-hosted-runners" >> "$GITHUB_OUTPUT"
    else
      echo "runner_label=ubuntu-latest" >> "$GITHUB_OUTPUT"
    fi
```

5.3.2 Utilizzo nei workflow

Per utilizzare l'action `find-runner` all'interno di un workflow, è stato definito un job dedicato:

Listing 5.12: Utilizzo dell'action find-runner in un workflow

```
jobs:
  find-runner:
    runs-on: ubuntu-latest
    outputs:
      runner_label: ${{ steps.find-runner.outputs.runner_label }}
    steps:
      - uses: Thesis-repositories/find-runner-action@v1
        id: find-runner
        with:
          org: OrganizationName
          app-client-id: ${{ vars.API_AUTHENTICATION_APP_CLIENT_ID }}
          app-private-key: ${{ secrets.API_AUTHENTICATION_APP_PRIVATE_KEY }}
```

Si noti che le credenziali della GitHub App sono configurate a livello di organizzazione: l'`app-client-id`, non essendo un dato sensibile, è stato memorizzato come variabile, mentre l'`app-private-key` come segreto. Il job di CI dichiara quindi una dipendenza verso il job probe tramite il campo `needs`, così da essere eseguito

solo al completamento di quest'ultimo, e imposta il proprio campo `runs-on` sulla base dell'output restituito dal job probe.

Capitolo 6

Validazione

L'obiettivo di questo capitolo è verificare se il sistema sviluppato soddisfa i requisiti funzionali definiti nel capitolo di Analisi, ovvero scalare i runner self-hosted mantenendone un pool minimo sempre attivo, ed effettuare un fallback sui runner GitHub-hosted qualora un runner self-hosted non sia immediatamente disponibile. La validazione è stata effettuata sperimentando il sistema in un contesto reale, sfruttando il repository Protelis-Demo all'interno dell'organizzazione GitHub Protelis. È stato utilizzato il cluster K3s con ARC già esistente, configurando per l'occasione un apposito runner scale set, mentre per l'autenticazione è stata creata e installata sull'organizzazione una nuova GitHub App.

6.1 Preparazione dell'ambiente di test

6.1.1 Modifiche al workflow

Per prima cosa è stata aperta una Pull request verso Protelis-Demo, modificando la pipeline di CI esistente in due modi:

- Aggiungendo il job probe, che utilizza al suo interno l'action `find-runner`.

Listing 6.1: Workflow all'interno di Protelis-demo, job probe

```
1 find-runner:
2   runs-on: ubuntu-latest
3   outputs:
4     runner_label: ${{ steps.find-runner.outputs.runner_label }}
```

```
  steps:
    - uses: Thesis-repositories/find-runner-action@v1
      id: find-runner
      with:
        org: Protelis
        app-client-id: ${{ vars.PROTELIS_SELF_HOSTED_RUNNERS_APP_ID }}
        app-private-key: ${{ secrets.
            PROTELIS_SELF_HOSTED_RUNNERS_PRIVATE_KEY }}
```

- Modificando il campo `runs-on` del job di CI, così che utilizzasse l'output del job probe.

Listing 6.2: Workflow all'interno di Protelis-demo, job di ci

```
build:
  needs:
    - find-runner
  strategy:
    matrix:
      os: [ windows, macos, ubuntu ]
    fail-fast: false
  runs-on: ${{ matrix.os == 'ubuntu' && needs.find-runner.outputs.
      runner_label || format('{0}-latest', matrix.os) }}
```

Si noti che nel job `build`, viene utilizzata una matrice contenente i tre sistemi operativi supportati da GitHub Actions (`windows`, `macos`, `ubuntu`), con lo scopo di eseguire l'elaborazione su tutti. Siccome l'infrastruttura on-premise sviluppata supporta solo il sistema operativo `ubuntu`, è stato necessario inserire nel campo `runs-on` una condizione per cui venisse utilizzato l'output del job probe solo per il ramo `ubuntu` della matrice, lasciando invariati gli altri due.

6.1.2 Configurazione delle credenziali

Successivamente, è stata creata una nuova GitHub App con i permessi appropriati e installata sull'organizzazione `Protelis`. Sono state quindi recuperate le credenziali necessarie: App client ID, App installation ID e App private key. Con questi dati è stato possibile effettuare due operazioni:

- Creare la variabile `PROTELIS_SELF_HOSTED_RUNNERS_APP_ID` e il segreto `PROTELIS_SELF_HOSTED_RUNNERS_PRIVATE_KEY` su Protelis; necessari per il corretto funzionamento del workflow.

- Creare il segreto Kubernetes all'interno del cluster, necessario per autenticare il nuovo runner scale set con Protelis.

6.1.3 Creazione del runner scale set

Infine, configurando il file `values.yaml` del chart `gha-runner-scale-set` con gli opportuni valori, è stato installato un nuovo runner scale set.

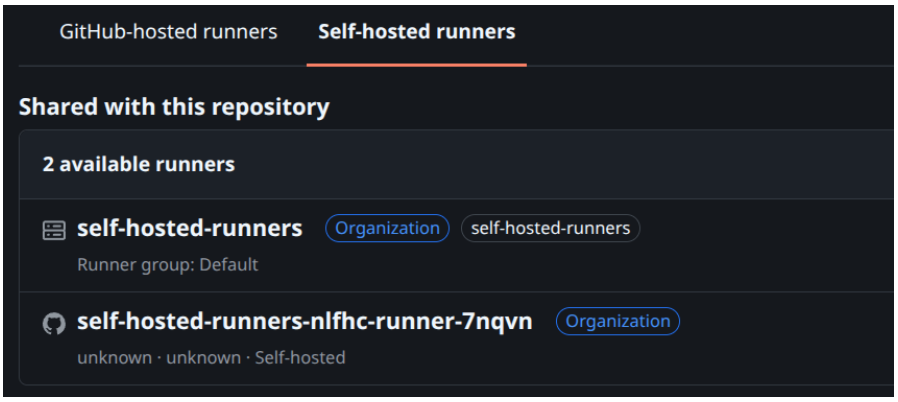


Figura 6.1: Runner self-hosted registrati.

A dimostrazione che tutto è stato eseguito correttamente, si può osservare che nella pagina GitHub dedicata è comparso il runner scale set appena creato, oltre ad un runner vero e proprio. Si noti che il runner è esattamente uno poiché, nel campo `minRunners` del file `values.yaml`, è stato impostato questo stesso valore; di conseguenza, il sistema manterrà un pool minimo di un solo runner, creandone uno nuovo non appena questo viene occupato da un job.

6.2 Risultati

Con l'obiettivo di verificare il funzionamento del sistema in tutte le sue sfaccettature, sono stati realizzati tre test:

6.2.1 Test di trasparenza in assenza di credenziali

Questo test è stato realizzato in assenza delle credenziali sull'organizzazione, per verificare se il workflow avrebbe comunque avuto successo, effettuando il fallback, oppure sarebbe fallito.

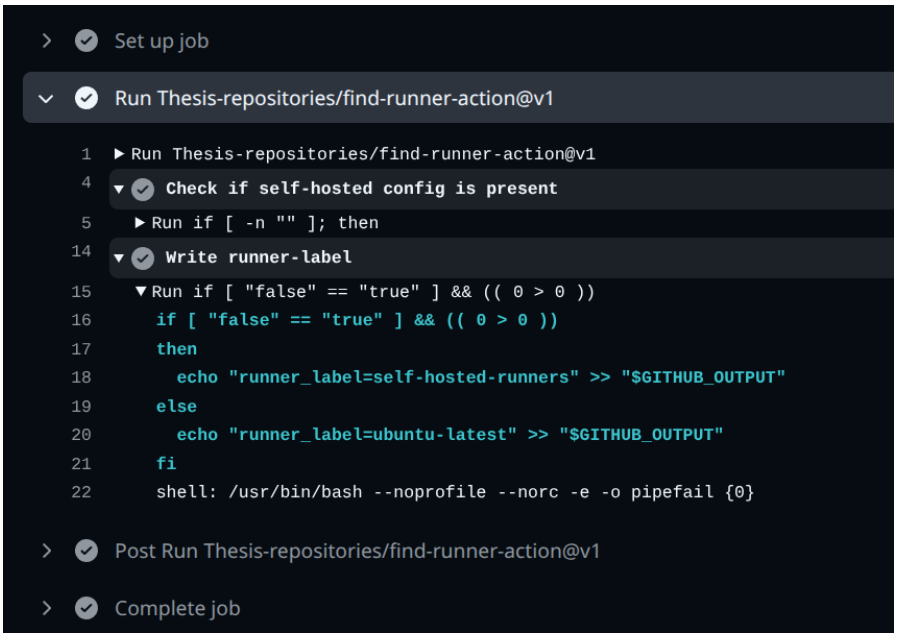


Figura 6.2: Esecuzione del job probe nel test senza credenziali.

Come si può notare dall'esecuzione del job probe, i due step intermedi dell'action `find-runner`, che in assenza di credenziali avrebbero causato il fallimento del job, sono stati saltati. Il nodo decisionale all'interno dell'ultimo step porta al ramo `ubuntu-latest`, ovvero l'identificativo per i runner GitHub-hosted di tipo ubuntu. Dopo aver atteso la naturale conclusione del job `build`, è stato quindi confermato il corretto funzionamento del sistema in questo scenario.

6.2.2 Test di fallback per assenza di runner self-hosted

Questo test verifica se il sistema effettua correttamente il fallback sui runner GitHub-hosted, non in caso di assenza delle credenziali come il primo, ma nello scenario in cui non ce ne fosse uno self-hosted immediatamente disponibile.

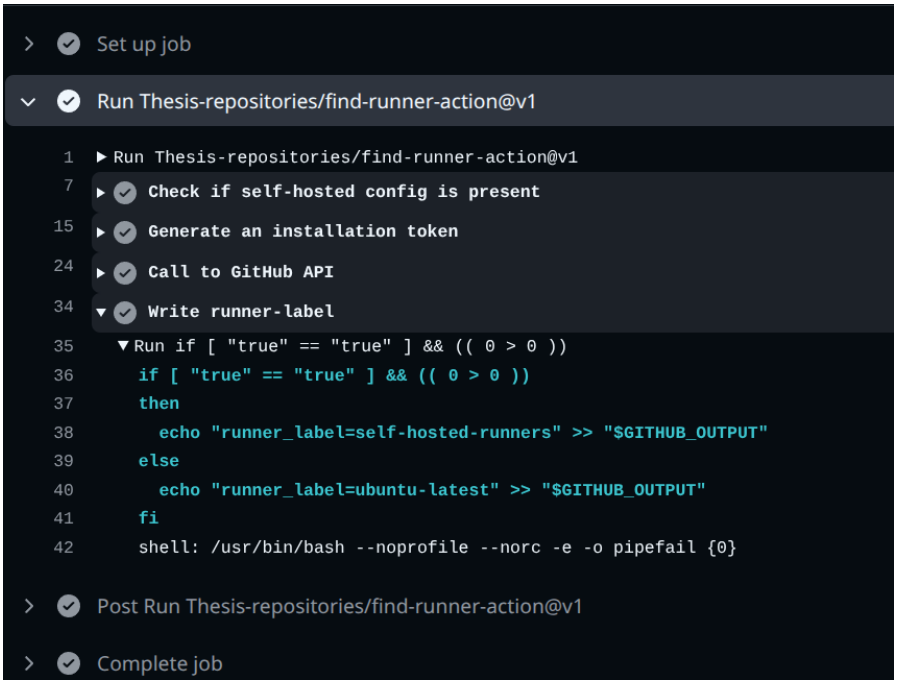


Figura 6.3: Esecuzione del job probe nel secondo test

Come si può osservare, in questo caso tutti gli step sono stati eseguiti, e la chiamata alle API di GitHub è stata effettuata correttamente; tuttavia, avendo rilevato zero runner self-hosted disponibili, nell'ultimo step la condizione porta comunque al ramo `ubuntu-latest`; il job `build` è stato quindi interamente eseguito sull'infrastruttura GitHub, confermando la riuscita del test.

6.2.3 Test di instradamento sull'infrastruttura self-hosted

Questo test verifica che, con la configurazione delle credenziali completata, il job venga effettivamente instradato verso l'infrastruttura on-premise.



Figura 6.4: Esecuzione del job probe nel terzo test.

Anche in questo caso sono stati eseguiti tutti gli step dell'action, compresi i due responsabili della comunicazione con le API di GitHub; la condizione dell'ultimo step porta al ramo `self-hosted-runners`, l'identificativo del runner scale set creato. Questo output conferma, oltre al corretto funzionamento dell'action `find-runner`, che l'autenticazione lato GitHub è avvenuta correttamente.

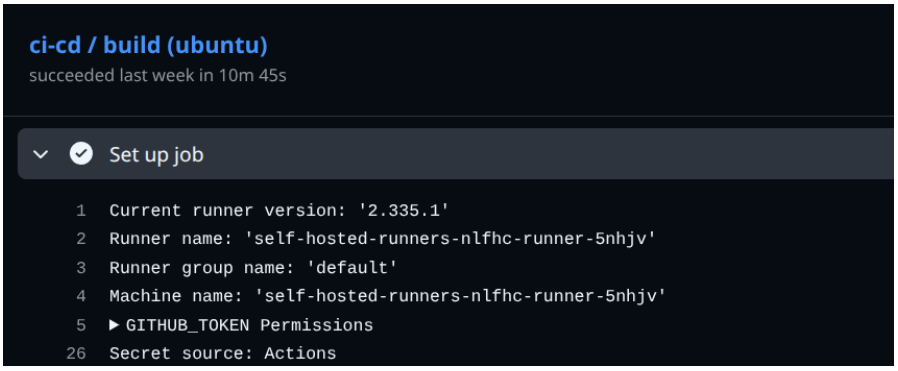


Figura 6.5: Ambiente di esecuzione del job build nel terzo test.

Infine, per verificare che il job `build`, ramo `ubuntu`, sia stato eseguito sull'infrastruttura on-premise, è possibile osservare l'output della fase di set up del job. Come si può notare, tra le informazioni che compaiono è presente quella relativa al runner utilizzato, corrispondente ad uno di quelli del runner scale set all'interno del cluster.

6.2.4 Test di scaling

Questo test verifica che, quando un runner self-hosted viene occupato dall'esecuzione di un job, il sistema crei automaticamente un nuovo runner in sostituzione, mantenendo così il pool minimo configurato.

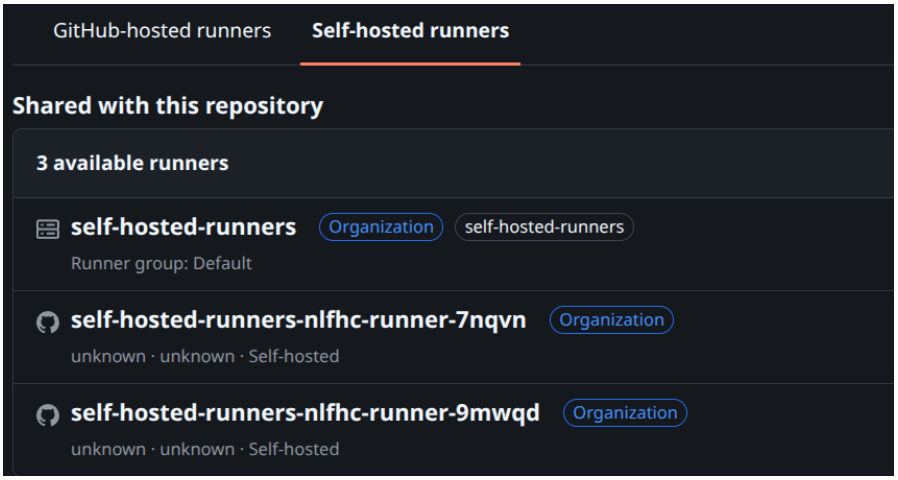


Figura 6.6: Runner self-hosted registrati durante l'esecuzione di un job.

La figura mostra che, subito dopo l'avvio di un workflow, compare nella lista un nuovo runner; questo conferma che il sistema, con l'obiettivo di mantenere sempre attivo un pool di esattamente un runner, crea una nuova istanza in sostituzione di quella appena occupata.

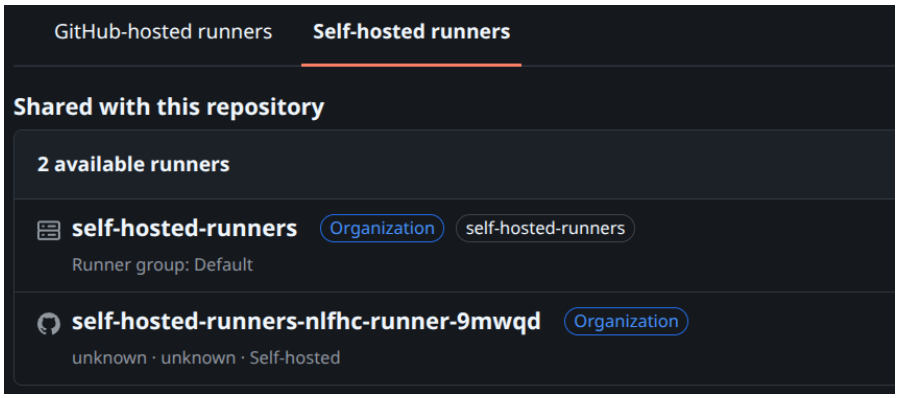


Figura 6.7: Runner self-hosted registrati al termine dell'esecuzione.

Al termine dell'esecuzione il runner utilizzato dal job è stato rimosso, e il numero di runner è tornato al valore minimo configurato, confermando il corretto funzionamento del meccanismo di scaling.

6.3 Discussione

I test condotti coprono complessivamente entrambi i requisiti funzionali definiti in Analisi; il primo e il secondo test dimostrano il corretto funzionamento del fallback nei due scenari possibili: in assenza di credenziali e in assenza di runner self-hosted disponibili. Il terzo e il quarto, invece, dimostrano che l'instradamento verso l'infrastruttura on-premise avviene e che il meccanismo di scaling funziona come previsto. È quindi possibile affermare che il sistema soddisfa i requisiti funzionali.

Tuttavia, è opportuno segnalare alcuni limiti della validazione condotta. In primo luogo, non è stato verificato il comportamento del sistema in presenza di più job concorrenti; di conseguenza, la race condition, discussa nel capitolo di Design, non è stata oggetto di osservazione diretta. In secondo luogo, questa validazione è stata prettamente funzionale, non è stata quindi effettuata alcuna misurazione

quantitativa, ad esempio sulla latenza dei job o sull'utilizzo delle risorse fisiche; per questo motivo non è possibile affermare che il sistema soddisfi anche i requisiti non funzionali definiti in Analisi, nonostante siano stati largamente trattati e tenuti in considerazione nello svolgimento del progetto.

Capitolo 7

Conclusione e sviluppi futuri

L'obiettivo di questo progetto di tesi è stato quello di spostare il workload di CI da un'infrastruttura cloud ad una on-premise, per avere prestazioni più stabili, un maggiore controllo sull'ambiente di esecuzione e un eventuale risparmio sui costi. I problemi nell'effettuare tale passaggio sono stati due, entrambi dovuti al mantenimento di un numero fisso di runner: lo spreco di risorse nei periodi di carico basso o assente, e la latenza aggiuntiva nei momenti di carico particolarmente elevato. Per risolverli, è stato deciso di adottare una strategia di autoscaling orizzontale e reattiva, affiancata da un meccanismo di fallback verso i runner GitHub-hosted. Per lo scaling, dopo aver analizzato e confrontato una soluzione custom con una basata su ARC, è stata scelta quest'ultima; il meccanismo di fallback è stato invece implementato direttamente a livello di workflow, tramite un job probe. Il sistema realizzato è stato validato tramite una sperimentazione pratica in un contesto reale, nella quale sono stati realizzati una serie di test per coprire i requisiti funzionali precedentemente definiti, dimostrando che il sistema li soddisfa. Il lavoro svolto presenta alcuni limiti, già discussi nei capitoli precedenti, che è opportuno sintetizzare. Lato infrastrutturale, l'adozione della soluzione basata su ARC ne presenta due: la capacità di scaling, relativa ai pod ma non ai nodi stessi del cluster, e la scelta della modalità Docker-in-Docker per la gestione dei job che fanno uso di container, che comporta un compromesso a livello di sicurezza e un maggiore consumo di risorse. L'implementazione del meccanismo di fallback tramite job probe, invece, presenta una possibile race condition che può aumentare la latenza di un

job. Infine, la validazione condotta è solo di tipo qualitativo: manca una valutazione quantitativa di metriche come latenza o utilizzo di risorse. Il lavoro svolto in questa tesi lascia aperte diverse direzioni di sviluppo. In primo luogo, riprendendo il limite relativo alla mancanza di una valutazione quantitativa, sarebbe opportuno approfondire diversi aspetti del sistema. Si potrebbe, ad esempio, effettuare un confronto diretto in termini di prestazioni e costi tra l'esecuzione di pipeline esclusivamente in cloud e l'utilizzo di questo sistema; oppure tra le due modalità di ARC nel gestire i container, Docker-in-Docker e Kubernetes. Un confronto quantitativo di questo tipo richiederebbe, tra l'altro, di completare l'implementazione della soluzione custom discussa nel capitolo di Design, così da poterla misurare concretamente, anche in relazione alla soluzione scelta. Un'ulteriore direzione riguarda il limite di ARC relativo allo scaling dei soli pod: lo sviluppo, o l'adozione, di uno strumento maturo di cluster autoscaling per infrastrutture Proxmox permetterebbe di automatizzare anche l'aggiunta e la rimozione di nodi fisici del cluster, superando la necessità di intervenire manualmente tramite gli script Terraform e Ansible sviluppati in questa tesi.

Bibliografia

[1] Winston W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, pages 1–9, Los Angeles, CA, USA, 1970.

[2] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9):833–859, 2008.

[3] Aymeric Hemon, Barbara Lyonnet, Frantz Rowe, and Brian Fitzgerald. From agile to devops: Smart skills and collaborations. *Information systems frontiers*, 22(4):927–945, 2020.

[4] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE software*, 33(3):94–100, 2016.

[5] João Faustino, Daniel Adriano, Ricardo Amaro, Rubén Pereira, and Miguel Mira Silva. Devops benefits: A systematic literature review. *Software: Practice and Experience*, 52(9):1905–1926, 2022.

[6] Vasanth K. Makam. Continuous integration on cloud versus on premise: A review of integration tools. *Advances in Computing*, 10(1):10–14, 2020.

[7] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. The effects of continuous integration on software development: a systematic literature review. *Empirical software engineering : an international journal*, 27(3), 2022.

[8] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical Report NIST SP 800-145, National Institute of Standards and Technology, 2011.

[9] Volodymyr Obrizan. Ciperf: a benchmark for continuous integration services performance and cost analysis. *Herald of Advanced Information Technology*, 7:275–283, 09 2024.

[10] Petr John, Kristýna Zaklová, Juraj Lazúr, Jiří Hynek, and Tomáš Hruška. A self-hosted approach to automatic ci/cd using open-source tools on low-power devices. In *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pages 100–107, 2024.

[11] Byeonghui Jeong and Young-Sik Jeong. Autoscaling techniques in cloud-native computing: A comprehensive survey. *Computer Science Review*, 58:100791, 2025.

[12] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[13] Pooya Rostami Mazrae, Alexandre Decan, Tom Mens, and Mairieli Wessel. An empirical study of the evolution of github actions workflows. *Journal of Systems and Software*, 236:112824, 2026.

[14] Murugiah Souppaya, Karen Scarfone, and Paul Hoffman. Guide to security for full virtualization technologies, 2011-01-28 2011.

[15] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[16] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. A survey of mobile device virtualization: Taxonomy and state of the art. *ACM Comput. Surv.*, 49(1), April 2016.

[17] Giovanni Quattrocchi and Damian A. Tamburri. Infrastructure as code. *IEEE Software*, 40(1):37–40, 2023.

Ringraziamenti

Innanzitutto, ringrazio il prof. Pianini, l'ing. Baiardi e il prof. Ciatto per avermi guidato durante questo percorso di tesi; riflettere sulle vostre osservazioni e sui vostri suggerimenti alle mie domande, idee e soluzioni penso mi abbia permesso di maturare nel mio approccio a questo mondo, oltre ad avermi aiutato a guardare le cose da prospettive diverse rispetto a quelle a cui ero abituato.

Ringrazio la mia famiglia per avermi supportato in questi tre anni e per avermi concesso l'opportunità di concentrarmi serenamente sullo studio; tutti, dai miei genitori ai miei nonni, zii e cugini, avete contribuito a rendere la famiglia un rifugio sicuro, dove potermi sentire apprezzato.

Ringrazio quelli che probabilmente hanno avuto l'influenza maggiore sul mio percorso: i compagni con cui ho condiviso tante lezioni e pause pranzo, Fede, Mouchi e Lollo. I confronti, le discussioni, le chiacchierate e le risate: tutto questo mi rimarrà dentro anche se le nostre strade si sono divise.

Voglio ringraziare anche tutto il mio gruppo di amici: Aldo, Ila, Giordi, Stive, Cecca, Bato e Panu, per regalarmi sempre momenti di leggerezza e allegria.

Infine, ultima in ordine ma ovviamente non per importanza, voglio ringraziare Michelle, la persona con cui da ormai quasi tre anni sto condividendo gran parte del mio tempo e della mia vita; ti ringrazio per essermi vicina, per cercare sempre di supportarmi e, semplicemente, per volermi bene.