



Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Implementazione di un ambiente di esecuzione basato su Kubernetes nel linguaggio FLY

Relatore

Prof. Vittorio Scarano

Correlatore

Dott. Giuseppe D'Ambrosio

Dott. Carmine Spagnuolo


Candidato

Luigi Barbato

Anno Accademico 2020-2021

Abstract

Negli ultimi anni, il paradigma del Cloud Computing ha ottenuto grande successo grazie ai numerosi vantaggi che esso offre. I motivi del suo successo sono ampiamente giustificati dal grande numero di vantaggi che ha portato. Tra questi, il più importante consiste nella possibilità di accedere a diversi tipi di risorse hardware e software mediante un paradigma basato sui servizi basati su processi containerizzati. La portabilità e la riproducibilità di un processo containerizzato offre l'opportunità di spostare e scalare le applicazioni tra cloud e data center. I container garantiscono efficacemente che tali applicazioni vengano eseguite allo stesso modo ovunque, consentendoci di sfruttare rapidamente e facilmente tutti questi ambienti. Inoltre, man mano che le applicazioni vengono scalate, abbiamo bisogno di alcuni strumenti per automatizzare la manutenzione di tali applicazioni come ad esempio Kubernetes. Kubernetes è una piattaforma portatile, estensibile e open-source per la gestione e l'automazione di applicativi Cloud-Native, ovvero applicativi basati su processi containerizzati e sviluppati appositamente per essere utilizzati in ambienti Cloud. Esso consente di orchestrare applicazioni containerizzate all'interno di cluster di nodi su cui vengono eseguiti, grazie agli strumenti ed alle API fornite, operazioni per la gestione del carico di lavoro, del traffico di rete e dell'archiviazione. Nonostante i vantaggi, Kubernetes rimane un sistema estremamente complesso ed articolato da utilizzare, necessitando di conoscenze specifiche per essere usato in modo efficace. Fly è un Domain-Specific Language per il calcolo scientifico su multi-cloud il cui obiettivo è quello di semplificare lo sviluppo di applicazioni su Cloud introducendo un livello di astrazione che permetta all'utente di utilizzare le funzionalità e le potenzialità di un ambiente cloud in modo semplice ed efficiente. Questo lavoro di tesi presenta l'integrazione di un ambiente di esecuzione basato su Kubernetes all'interno del linguaggio di programmazione Fly. Grazie ai costrutti specifici introdotti all'interno del linguaggio FLY è possibile automatizzare completamente il processo di esecuzione di un'applicazione su Kubernetes.

Questa tesi è stata sviluppata in  **ISISLab**

Indice

1	Introduzione	1
1.1	Cloud Computing	1
1.1.1	Modelli di servizio	1
1.1.2	Modelli di distribuzione	2
1.2	Multi-cloud	3
1.3	Containerizzazione	4
1.3.1	Vantaggi	4
1.4	Docker	5
1.5	Kubernetes	6
1.5.1	Architettura	6
1.5.2	Componenti del Control Plane	7
1.5.3	Componenti del Worker Node	8
1.5.4	Oggetti Kubernetes	8
2	Stato dell'Arte	11
3	Fly Language	13
3.1	Obiettivi	14
3.2	Architettura	16
3.3	Definizione del linguaggio	18
3.3.1	Struttura di un progetto Fly	23
3.4	Generazione del codice	23
3.4.1	Ambiente di esecuzione	26
3.4.2	<i>Channel</i>	29
3.4.3	Script di deploy e undeploy delle funzioni	29
3.4.4	Sincronizzazione	31
4	Ambiente di esecuzione Kubernetes	32
4.1	Kubernetes in Fly	32

4.2	Integrazione in Fly	33
4.3	Implementazione	34
4.4	Generazione codice	35
5	Conclusioni	37
5.1	Obiettivi raggiunti	38
5.2	Sviluppi futuri	38

Capitolo 1

Introduzione

1.1 Cloud Computing

Il Cloud Computing è una particolare ed innovativa forma di erogazione delle risorse, permette infatti di erogare risorse di computazione quando ne sia necessario. Si basa sul modello di costo pay-as-you-go. Questo modello consente alle aziende di pagare in base ai loro consumi. Grazie al Cloud Computing, inoltre, le aziende non hanno più la necessità di possedere un'infrastruttura fisica di calcolo, ma possono noleggiarne una dai vari Cloud Provider, che si occuperanno inoltre di gestirla e mantenerla. Il Cloud Computing ha avuto una rapida e veloce espansione negli ultimi anni principalmente grazie a questi vantaggi.

1.1.1 Modelli di servizio

Il Cloud Computing è un mondo in continua crescita ed evoluzione, nuovi prodotti e servizi cloud arrivano quasi ogni giorno, spinti dalle costanti innovazioni tecnologiche. Nonostante la maturità del mercato, molte organizzazioni non sono ancora a conoscenza dei servizi e dei modelli d'implementazione che i diversi Cloud Provider forniscono. I tre modelli principali a oggi maggiormente diffusi sono così definiti:

Software as a Service (SaaS)

Software as a Service (SaaS) è il modello di servizio cloud che fornisce l'accesso a un prodotto software completo, eseguito e gestito dal fornitore del servizio. In questo particolare modello, il fornitore ha la piena responsabilità sull'intero ciclo di vita del software e sulla manutenzione dell'infrastruttura

sottostante. Questo consente al cliente di concentrarsi esclusivamente su come utilizzare al meglio le specifiche e le funzionalità del software offerto.

Platform as a Service (Saas)

Platform as a Service (PaaS) è il modello usato più comunemente per lo sviluppo di applicazioni. Il fornitore del servizio fornisce l'accesso alle proprie risorse infrastrutturali come: basi di dati, sistemi operativi e server, senza la complessità di gestione che ne deriva. Questo permette al cliente di dedicare le proprie risorse all'ottimizzazione dell'applicativo invece che all'installazione e alla configurazione dell'infrastruttura.

Infrastructure as a Service (Saas)

Infrastructure as a Service (IaaS) è il modello che permette al cliente d'implementare la propria infrastruttura cloud. Il Cloud Provider fornisce l'accesso on-demand delle proprie risorse (sia fisiche che virtuali) volte alla computazione, archiviazione e alla rete. Il cliente ottiene così l'enorme vantaggio di poter distribuire e gestire i propri applicativi avendo la sicurezza di avere risorse sempre disponibili, affidabili e scalabili.

1.1.2 Modelli di distribuzione

Ogni modello d'implementazione del cloud ha una propria configurazione unica con una gamma di requisiti diversi e vantaggi associati.

Public Cloud

Il Public Cloud è un modello di distribuzione in cui i servizi cloud sono di proprietà di fornitori esterni. Scegliere questa metodologia garantisce una maggiore agilità operativa e una scalabilità pressoché illimitata, perché sfrutta le funzionalità e le risorse di grandi fornitori come Google, Microsoft e Amazon. Trattandosi inoltre di ambienti gestiti, il cliente si solleva dalle responsabilità di manutenzione e controllo delle risorse utilizzate.

Private Cloud

Nel modello di distribuzione Private Cloud si sceglie di sviluppare, mantenere e gestire la propria infrastruttura cloud, fornendo l'accesso solo alla propria rete interna. Con il Private Cloud le organizzazioni mantengono un controllo completo dell'infrastruttura. Questo si traduce in un controllo completo delle

proprie risorse e ad una maggior libertà di personalizzazione dei servizi, che sono consumati all'interno del cloud.

Hybrid Cloud

L'Hybrid Cloud è un modello di distribuzione ibrido in quanto Si sceglie di unire i vantaggi del cloud pubblico e di quello privato. Ad esempio, si possono utilizzare le risorse del Public Cloud per attività di computazione, e tenere al sicuro i dati sensibili e le applicazioni critiche nel Private Cloud. Oppure, di fronte a picchi improvvisi e temporanei della domanda di risorse, le organizzazioni possono scegliere di spostare i carichi di lavoro dal Private Cloud al Public Cloud. Utilizzato in questo modo, l'Hybrid Cloud consente di ottenere il meglio da entrambe le infrastrutture.

1.2 Multi-cloud

il successo del cloud computing ha portato alla nascita di molteplici Cloud Provider. Questo ha spinto gli sviluppatori a sfruttare diversi cloud provider nella stessa architettura portando alla nascita del multi-cloud Il multi-cloud è una forma architetturale in cui si utilizzano risorse e servizi di Cloud Provider differenti convergendoli in una singola architettura eterogenea. Le motivazioni che muovono un'organizzazione ad adottare questo tipo di strategia sono varie:

- **Maggiore flessibilità** - Ogni Cloud Provider propone la propria gamma di servizi che differiscono tra loro in caratteristiche diventando migliori da utilizzare per particolari situazioni rispetto che altre. La possibilità e quindi la libertà di poter scegliere e selezionare distintamente i singoli servizi concedono all'utente il vantaggio di poter coprire totalmente le proprie esigenze.
- **Localizzazione** - Ogni Cloud Provider detiene le proprie infrastrutture dislocate in diverse aree geografiche, l'utente può decidere di utilizzarle in base alle proprie esigenze logistiche e legali.
- **Prestazioni** - L'utilizzo di risorse provenienti da fornitori distinti, permette una distribuzione più ampia dei servizi dell'applicativo.

1.3 Containerizzazione

La virtualizzazione è una tecnologia che permette di mettere a disposizione risorse hardware come CPU e memoria sottoforma di risorse virtuali. Attraverso la virtualizzazione è possibile utilizzare delle risorse virtuali allo stesso modo di come si farebbe con delle risorse fisiche permettendo di migliorare la scalabilità e i carichi di lavoro, usando al contempo un minor numero di macchine, una quantità di energia elettrica ridotta, generando così un risparmio sui costi di infrastruttura e gestione. Uno dei principali vantaggi della virtualizzazione è la razionalizzazione e l'ottimizzazione delle risorse hardware in quanto più macchine virtuali possono girare contemporaneamente su un sistema fisico condividendo le risorse della piattaforma. La metodologia più semplice di virtualizzazione prevede la creazione di una macchina virtuale che va a simulare un'intero sistema fisico con risorse come CPU, memoria e componenti di reti. Una macchina virtuale dovrà quindi essere utilizzata come un normale computer desktop, eseguendo un proprio sistema operativo e rimanendo completamente isolata dall'hardware fisico sottostante. Con il termine containerizzazione si intende una para-virtualizzazione dell'ambiente applicativo che consente di eseguire software, librerie, dipendenze e tutte le componenti necessarie, in un processo isolato che prende il nome di contenitore. Esso può essere considerato un vero e proprio eseguibile che non dipende da alcuna sorgente esterna. Questo lo rende estremamente portatile e affidabile in quanto può essere eseguito e trasferito in ogni tipo di ambiente e d'infrastruttura. L'idea alla base del paradigma non è in realtà nuova in quanto sfrutta una funzionalità del kernel Linux presente già dalla versione 2.6.24 che permette l'isolamento e la gestione delle risorse di uno o più processi.

1.3.1 Vantaggi

Il grande successo della virtualizzazione basata su container deriva dai grandi vantaggi che essa garantisce. Tra essi troviamo:

- **Portabilità** - I container sono altamente portabili in quanto contengono al loro interno tutte le dipendenze e le componenti necessarie all'esecuzione dell'applicazione, evitando così problemi di compatibilità
- **Efficienza** - I container hanno la capacità di condividere il kernel della macchina ospitante evitando così di dover disporre di risorse hardware e software dedicate. Inoltre è sempre possibile aumentare e diminuire

le risorse in modo istantaneo, in base alle proprie esigenze, in modo da non incorrere in sprechi o in carenze.

- **Ottimizzazione dello spazio** - L'immagine dell'applicazione che viene eseguita all'interno del contenitore, incapsula solo ed unicamente le componenti e le informazioni necessarie alla sua esecuzione.

1.4 Docker

Docker è una tecnologia open-source per la costruzione, spostamento, distribuzione e rilascio di applicazioni basate su container, distribuita sotto Licenza Apache Common 2.0, disponibile per tutte le maggiori piattaforme in grado di eseguire container in ambienti Linux oppure Windows. La struttura stessa dei container garantisce il loro isolamento, permettendo l'esecuzione multipla su un singolo host, sfruttando direttamente il kernel della macchina senza la necessità di un server hypervisor. In questo modo, i container sono il centro dello sviluppo dell'applicazione, la quale viene costruita come insieme di container per esprimere le dipendenze, permettendone una facile distribuzione e testing in ambienti diversi, fino alla messa in campo in produzione in un data center locale, presso un cloud provider o una soluzione ibrida tra le precedenti. Docker utilizza e crea differenti tipi di oggetti, come immagini e container. Un container costituisce un'istanza di un'immagine in esecuzione, con il quale è possibile interagire direttamente dalla macchina host, e può essere visto come un'evoluzione di un processo, isolato dagli altri container e dal sistema host. Un'immagine, invece, costituisce un template di sola lettura, utilizzato per la creazione di un container Docker. Solitamente, un'immagine viene costruita sulla base di un'altra immagine alla quale vengono aggiunte le configurazioni necessarie e peculiari dell'applicazione. La definizione di un'immagine avviene all'interno di un file, chiamato Dockerfile, nel quale vengono indicate le istruzioni, ognuna delle quali aggiunge un layer all'interno dell'immagine finale, in modo tale che, di volta in volta, vengano ricompilati solo gli strati che effettivamente sono stati modificati. Sebbene Docker sia un potente strumento di sviluppo e di gestione degli ambienti containerizzati, presenta grosse limitazioni che non consentono un'efficace gestione, scalabilità e manutenzione degli applicativi. A partire da questa necessità, sono quindi nate le più conosciute soluzioni di container orchestration, tra cui Kubernetes.

1.5 Kubernetes

Kubernetes è una piattaforma portatile, estensibile e open-source per la gestione e l'orchestrazione di applicativi Cloud-Native. La piattaforma è scritta in linguaggio Go ed è stato inizialmente sviluppato da Google per migliorare la gestione dei propri applicativi. Il progetto è attualmente parte della Cloud Native Computing Foundation¹, organizzazione che promuove e mantiene progetti open-source volti all'approccio Cloud Native. Kubernetes utilizza un insieme di oggetti fruibili tramite API per descrivere lo stato desiderato del cluster, indicando, ad esempio, quali applicazioni eseguire, quali immagini utilizzare, il numero di repliche da istanziare, quali risorse di rete e di spazio su disco rendere disponibili. L'interazione viene resa possibile grazie a kube-apiserver, il server HTTP che implementa l'API Kubernetes. L'utente ha dunque due possibilità per manipolare le configurazioni del cluster, la prima è quella di eseguire esplicitamente richieste di tipo RESTful al server API, la seconda è quella di utilizzare kubectl, un'interfaccia da linea di comando costituita da una serie di comandi e sotto-comandi che astraggono le chiamate API.

1.5.1 Architettura

Un cluster Kubernetes è un insieme di macchine, chiamate nodi, che eseguono carichi applicativi. Il cluster deve avere almeno un Worker Node ed un Master Node. Il Worker Node è un tipo di nodo che ospita i Pod, particolari componenti di Kubernetes che contengono uno o più container. Il Master Node è il nodo su cui viene eseguito il Control Plane, una componente fondamentale di Kubernetes che svolge le attività operative del cluster.

¹Cloud Native Computing Foundation - <https://www.cncf.io/>

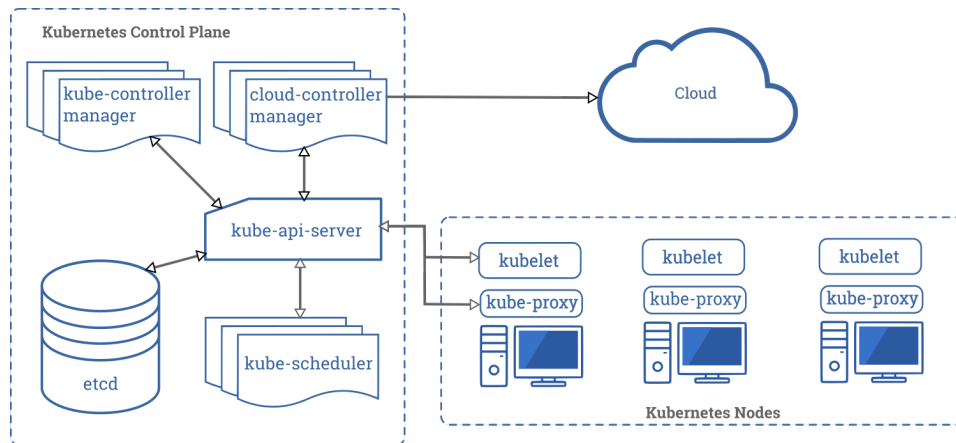


Figura 1.1: Architettura di Kubernetes

1.5.2 Componenti del Control Plane

Il Kubernetes Control Plane è il centro nevralgico delle operazioni del cluster. È costituito da una serie di componenti, in esecuzione all'interno del cluster, che hanno il compito di garantire l'integrità esecutiva del cluster come ad esempio lo scheduling dei Worker Node e la gestione dei possibili errori dei Pod.

- **kube-apiserver** è il server API per il cluster Kubernetes. È il punto di contatto centrale a cui accedono tutti gli utenti, l'automazione e i componenti nel cluster Kubernetes. Il server API implementa un'API RESTful su HTTP, esegue tutte le operazioni API ed è responsabile dell'archiviazione degli oggetti API in un backend di archiviazione persistente.
- **etcd** è la componente di archiviazione principale di Kubernetes, etcd memorizza e replica tutti gli stati dei cluster Kubernetes. Il meccanismo di memorizzazione che adotta è di tipo chiave-valore ovvero ad ogni valore memorizzato viene associato una chiave che rappresenta il suo identificatore univoco. In questo modo il Control Plane è in grado di storicizzare gli stati del Cluster confrontando lo stato attuale con quello desiderato in modo di intervenire tempestivamente in caso di necessità.
- **kube-scheduler** è la componente che consente al Control Plane lo scheduling dei Pod sui nodi. il kube-scheduler controlla i pod appena

creati che non hanno un nodo assegnato, e dopo averlo identificato glielo assegna.

1.5.3 Componenti del Worker Node

Il Worker Node è il nodo del cluster su cui viene eseguito effettivamente l'applicazione. Su ogni nodo di tipo Worker Node vengono eseguiti tre componenti di Kubernetes.

- **kubelet** La componente kubelet è un agente che è sempre in esecuzione su ogni Worker Node e ha come compito quello di controllare il ciclo di vita dei container all'interno dei Pod. La kubelet riceve un set di specifiche definite in fase di deployment dall'utente e si assicura che i container rispettino tali specifiche.
- **kube-proxy** La componente kube-proxy è un proxy eseguito su ogni Nodo, ossia un particolare componente che consente di identificare i nodi all'interno della rete del cluster e permette che essi possano comunicare sia all'interno che all'esterno del cluster.
- **Container Runtime** il Container Runtime è il software responsabile dell'esecuzione dei container. Kubernetes supporta tutte le implementazioni di Kubernetes CRI (Container Runtime Interface) come ad esempio Docker e containerd.

1.5.4 Oggetti Kubernetes

Kubernetes offre una varietà di oggetti per definire le specifiche del proprio sistema. Questi oggetti costituiscono entità persistenti per rappresentare lo stato del cluster, quali applicazioni sono in esecuzione e su quali nodi, le risorse da allocare e le politiche da adottare. Per istanziare un oggetto con le direttive e le informazioni desiderate, si utilizza spesso un file di tipo YAML oppure JSON. Ciascun oggetto è caratterizzato da due campi che ne descrivono la configurazione:

Specifiche Le specifiche descrivono lo stato desiderato dell'oggetto, cioè le caratteristiche che deve assumere.

Stato Lo stato descrive lo stato attuale dell'oggetto e viene automaticamente aggiornato da Kubernetes.

- **Pod** Un Pod costituisce la più semplice, piccola e basilare unità di esecuzione in Kubernetes. La sua natura funzionale è quella di incapsulare uno o più container e di dividerne le risorse come l'archiviazione dei dati e le risorse di rete. I Pod generalmente non vengono creati direttamente in Kubernetes perchè questi sono considerati oggetti effimeri. Solitamente i Pod vengono creati e gestiti da controller di più alto livello come Deployment, StatefulSet o DaemonSet.
- **Deployment** L'oggetto Deployment si occupa di creare e gestire il ciclo di vita di uno o più Pod e dei ReplicaSet. Un Deployment fornisce un approccio dichiarativo per la creazione e la modifica di Pod e ReplicaSet, con esso è possibile dunque descrivere lo stato che il Pod deve assumere ed il Deployment Controller si occuperà di aggiornare lo stato attuale con quello desiderato.
- **ReplicaSet** Un ReplicaSet ha il compito specifico di mantenere attive una o più copie di un Pod. Questo meccanismo garantisce la piena disponibilità del Pod a cui il ReplicaSet è agganciato in quanto nel momento in cui un Pod, per un qualsiasi motivo, cessa il suo funzionamento, verrà sostituito da una sua esatta copia.
- **Service** I Pod in Kubernetes sono entità effimere, essi vengono creati e distrutti in base allo stato desiderato del cluster. Per natura quindi non hanno un'identificazione statica all'interno della rete del cluster ma ad ogni Pod viene assegnato un indirizzo IP dinamico che varia nel tempo. Un Service in Kubernetes costituisce un'astrazione che definisce un insieme logico di Pod e una politica con cui accedervi. Kubernetes offre diversi tipi di Service:
 - ClusterIP** Espone il Service con un indirizzo IP interno al cluster, rendendo il cluster raggiungibile solo dall'interno del cluster stesso.
 - NodePort** Espone il Service su ciascun indirizzo IP corrispondente ad un nodo del cluster, su una porta scelta automaticamente, uguale per tutti i nodi.
 - LoadBalancer** Permette di esporre le applicazioni all'esterno del cluster.
- **Job** Un oggetto di tipo Job permette di eseguire e gestire task complessi all'interno del cluster. Un Job infatti permette di eseguire un insieme di Pod in maniera sequenziale o parallela considerandoli come un'unica attività di lavoro da portare a termine. Il Job terrà in

esecuzione l'attività fino a quando un numero specifico di Pod non verrà terminato correttamente. Quando viene raggiunto un numero specifico di completamenti riusciti, l'attività risulterà completata ed il Job cesserà la sua esecuzione. L'eliminazione del Job distruggerà dunque tutti i Pod precedentemente assegnati.

Capitolo 2

Stato dell'Arte

La crescente popolarità dell'approccio multi-cloud ha portato alla nascita di varie tecnologie che ne permettano l'utilizzo, ognuna delle quali utilizza un approccio specifico in base agli obiettivi che si pone. Inizialmente lo scopo degli strumenti era quello di fornire un modo per unificare la gestione e il controllo delle varie risorse fornite dai diversi provider, raggruppandole sotto un unico ambiente. È il caso di OpenStack [48], un software open-source rilasciato sotto licenza Apache che opera come un sistema operativo per Cloud. OpenStack permette di creare e gestire ambienti Cloud, sia pubblici che privati, attraverso risorse virtuali relative al modello IaaS, di conseguenza manca il supporto agli altri modelli, soprattutto al Serverless Computing. Un approccio simile è seguito da MODAClouds [5], un framework che segue un approccio model-driven per la progettazione e l'esecuzione di applicazioni multi-cloud. Il punto di forza è il suo approccio model-driven che consente di progettare le applicazioni ad alto livello, in completa astrazione rispetto al Cloud di riferimento, e poterle eseguire su qualsiasi piattaforma. Il codice in questione viene infatti tradotto in base all'ambiente di esecuzione permettendo una facile integrazione di più servizi Cloud e rendendo particolarmente semplice la migrazione delle applicazioni da provider a provider.

Con l'avvento del Serverless Computing è nata la necessità di estendere gli strumenti esistenti per supportare questo modello. È il caso di TOSCA [62], acronimo per Topology and Orchestration Specification for Cloud Applications, un linguaggio di standard OASIS per la modellazione per lo sviluppo e la gestione di applicazioni su Cloud. In Wurster et al. [67] viene esteso per integrare la possibilità di sfruttare le componenti Serverless all'interno delle applicazioni che si vuole creare. TOSCA permette di descrivere la struttura e il comportamento dei servizi forniti dal Cloud concentrandosi su

portabilità e interoperabilità delle applicazioni descritte, risultando tuttavia mancante di alcune funzionalità critiche per le applicazioni Cloud in quanto non permette di specificare i requisiti di risorse. Un approccio molto simile è quello di CAMEL [17], un Multi-Domain-Specific Language per la gestione del ciclo di vita delle applicazioni. Acronimo di Cloud Application Modelling and Execution Language, CAMEL consente di modellare componenti per applicazioni o servizi sfruttando più ambienti Cloud. Anche in questo caso il linguaggio viene esteso in Kritikos et al. [37] che introduce un'estensione che supporti le componenti Serverless. CAMEL è un multi-DSL, l'unione di diversi DSL per coprire tutti gli aspetti del ciclo di vita di un'applicazione, ed è basato sulla tecnologia Xtext di Eclipse. Basato su CAMEL MELODIC [30] è un middleware che permette di automatizzare ed ottimizzare lo sviluppo di applicazioni multi-cloud. Tutte le componenti di MELODIC non sono legate ad un particolare Cloud provider ma possono essere utilizzate per qualsiasi ambiente su Cloud. L'integrazione all'interno di MELODIC del supporto alle componenti Serverless è descritto in Kritikos e Skrzypek [36].

Un altro tipo di soluzioni è quello dedicato esclusivamente alle componenti Serverless, di cui l'esponente maggiore è certamente Serverless [57], un framework web open-source scritto in Node.js. Serverless è il primo framework sviluppato per la costruzione di applicazioni basate sui servizi di Serverless Computing offerte dai principali Cloud provider, sono infatti supportati AWS Lambda di AWS, Azure Function di Microsoft Azure, IBM Bluemix e Google Cloud Platform. Serverless fornisce una CLI, Command Line Interface, per sviluppare le varie applicazioni fornendo esempi, strutture pre-costruite per agevolare il lavoro dello sviluppatore. Supportando tutti i maggiori Cloud provider, la Serverless Framework CLI fornisce un'esperienza di sviluppo singola per diversi provider. Punto a favore è la possibilità di testare le proprie applicazioni in locale, senza necessità di interagire con il provider, evitando quindi di incorrere in costi di esecuzione. Al contrario il limite di Serverless è proprio la specificità di servizio, dedicato alla costruzione di applicazioni puramente Serverless.

Un approccio diverso è proposto in Vasconcelos et al. [65] in cui viene descritta un'architettura basata su container che permette di progettare e costruire un sistema distribuito tra diversi Cloud provider sul modello FaaS. Questa può essere vista come un'integrazione dei classici sistemi di High Performance Computing, formati da cluster di computer, declinati sul modello FaaS. Quello che si ottiene sono meccanismi di auto-scaling sia per macchine virtuali che per container in un ambiente distribuito multi-cloud, in aggiunta ad un sistema di bilanciamento del carico che tiene conto dei diversi cluster FaaS coinvolti.

Capitolo 3

Fly Language

Per sviluppare un'architettura multi-cloud non basta raggruppare i vari servizi dei diversi Cloud provider in un unico sistema ma bisogna avere un approccio specializzato e ben ragionato. Bisogna creare infatti un ambiente omogeneo, tale da sfruttare efficacemente le cooperazioni fra i diversi Cloud provider, cercando di non rendere troppo difficile la sua gestione. La difficoltà di costruire un ambiente multi-cloud risiede nella mancanza di standard nei vari servizi a disposizione. Ognuno dei diversi cloud provider mette a disposizione API e strumenti specifici per l'accesso e l'utilizzo dei propri servizi, costringendo lo sviluppatore a conoscerli tutti per la costruzione di un'architettura multi-cloud. È chiaro quindi che, anche se il multi-cloud comporta enormi vantaggi nel suo utilizzo, esso risulta particolarmente complesso da utilizzare e da sfruttare facendo nascere il bisogno di strumenti che facilitino lo sviluppo di applicazioni basate sul multi-cloud

La chiave di volta nell'utilizzo del multi-cloud è sicuramente l'astrazione, ovvero l'utilizzo di strumenti che sollevano l'utente dalla necessità di conoscere tutti i dettagli implementativi per la gestione dell'ambiente, permettendo così di ottenere i vantaggi del multi-cloud evitando di introdurre complessità aggiuntiva [59]. A questo scopo è nato **Fly**, un **Domain-Specific Language** per il Calcolo Scientifico sul multi-cloud, al fine di sviluppare con facilità applicazioni che utilizzino la complessità computazionale elargita da diversi Cloud provider, grazie al paradigma FaaS, garantendo alte prestazioni ed alta scalabilità. Il punto di forza di Fly risiede nella gestione dell'interazione con il Cloud la quale viene completamente astratta all'utente finale che non necessita di conoscere le specifiche API del provider che vuole utilizzare [19]. La novità in Fly risiede nel concetto di **funzione Fly**, ovvero un blocco di codice indipendente eseguibile concorrentemente, in linea con il modello

FaaS, che usufruisce di funzioni Serverless.

Le caratteristiche principali di Fly sono:

- Fly è *efficace* perché permette di sfruttare la potenza computazionale di diversi Cloud provider in poche righe di codice, utilizzando le soluzioni più efficienti in base alle necessità;
- Fly è *chiaro* perché è un linguaggio user-friendly, che rimuove diverse preoccupazioni alle quali il programmatore dovrebbe tener conto come il dover gestire e configurare diversi ambienti di esecuzione;
- Fly è *efficiente* perché permette di scegliere i servizi migliori sia dal punto di vista delle funzionalità che del costo, inoltre riduce i tempi di sviluppo grazie all'astrazione.

In questo capitolo saranno descritti i dettagli del linguaggio Fly, descrivendo gli obiettivi per cui è stato sviluppato, la sua architettura e il suo funzionamento. Viene inoltre introdotto il concetto di Domain-Specific Language e Xtext, il framework utilizzato per lo sviluppo di Fly.

3.1 Obiettivi

Fly ha come obiettivo congiungere il Cloud Computing con l'High Performance Computing, fornendo un applicativo potente, semplice ed efficace per lo sviluppo di applicativi in grado di sfruttare il sistema multi-cloud al massimo delle sue potenzialità.

I principali obiettivi di Fly sono:

- *espressività* - possibilità di scrivere applicativi in maniera precisa, intuitiva e comprensibile;
- *alta usabilità* - scrivere ed eseguire un applicativo Fly sono operazioni che devono essere semplici e completamente astratte dai processi necessari all'interazione con il Cloud Provider, eliminando così la necessità di conoscere le API del Cloud provider;
- *scalabilità* - I processi eseguiti in locale come quelli eseguiti in ambiente Cloud devono essere altamente scalabili.

Fly è stato realizzato per permettere agli esperti di un particolare campo che non hanno padronanza dei concetti articolati che riguardano i sistemi

paralleli e distribuiti, di implementare i propri algoritmi sfruttando il parallelismo mediante architetture Serverless. Infatti, la sintassi di Fly è ispirata a linguaggi come Java, JavaScript, Python e R. Linguaggi utilizzati spesso nel Calcolo Scientifico. Vengono forniti numerosi costrutti specifici per il dominio di interesse che vanno a formare un ricco linguaggio utilizzabile facilmente per interagire con i servizi su Cloud.

Fly supporta in modo implicito il paradigma di calcolo distribuito e parallelo insieme con la gestione della memoria fornendo inoltre un sistema di comunicazione di processi tramite appositi canali di comunicazione. Un programma Fly è infatti eseguibile sia su un'architettura multiprocessore sia su ambiente Cloud che supporti il modello FaaS, il tutto senza la necessità di conoscere in maniera precisa le risorse di computazione che occorrono per l'esecuzione [19].

Quando si parla di un **Domain-Specific Language (DSL)** si intende un linguaggio di programmazione sviluppato per l'utilizzo nel contesto di un particolare dominio, che elargisce una notazione su misura fondata sui criteri e sulle caratteristiche principali per tale dominio. Mentre i linguaggi General Purpose possono essere utilizzati in molteplici situazioni e problemi, i DSL invece ne rappresentano il concetto opposto. Utilizzando un DSL si introduce un livello di astrazione tale da rendere possibile la realizzazione di soluzioni efficaci senza aver bisogno di competenze tecniche specifiche anche ad esperti del dominio. Lo sviluppo di applicazioni software può così essere affidato a figure con conoscenze del contesto più specifiche e non più soltanto ai soli informatici, in modo da aumentare efficienza ed efficacia del procedimento. Un DSL però non è genericamente utile a tutti, è invece ideato per la risoluzione di problemi riguardanti contesti molto specifici. Fra i DSL possiamo prendere come esempio HTML, che è stato pensato per sviluppare siti web ma di certo non è compatibile, ad esempio, con la costruzione di applicazioni di simulazione, dominio a cui invece è dedicato OpenABL. Altri esempi di DSL potrebbero essere invece SQL per la stesura di query su database relazionali, che permette di scrivere interrogazioni semplici per l'ottenimento di dati senza la necessità delle competenze di un amministratore di database. Il motivo per cui nasce un DSL è quello di essere di facile comprensione ed impiego, merito della forte specificità di utilizzo, divenendo così espressivo, piccolo e poco ridondante e permettendo a sviluppatori ed esperti del dominio di utilizzarlo come punto d'incontro.

Per la costruzione di un DSL è necessario realizzare un compilatore capace di leggere il programma scritto in quello specifico linguaggio, esaminarlo, processarlo e interpretarlo per poter infine generare il codice eseguibile. Andranno quindi attraversate una serie di fasi esposte a seguire:

- *analisi lessicale* - il programma iniziale viene suddiviso in singole unità chiamate *token*, ognuna delle quali corrisponde ad un singolo elemento del linguaggio;
- *analisi sintattica* - i *token* identificati vengono analizzati per assicurarsi che formino uno statement valido per il linguaggio. È in questa fase che viene prodotto l'Abstract Syntax Tree (ABS), ovvero la rappresentazione della struttura sintattica del programma;
- *analisi semantica* - comprende la fase di type checking che controlla che gli assegnamenti di valore siano compatibili con il relativo tipo di dati insieme con la fase di tracciamento degli identificatori, del loro tipo e delle espressioni che si assicura anche che i primi siano stati dichiarati prima dell'uso;
- *generazione del codice* - parte finale che utilizza i risultati delle fasi precedenti per generare il codice macchina o, eventualmente, il programma scritto in un altro linguaggio.

Xtext

Xtext [68] è un framework open-source per lo sviluppo di linguaggi di programmazione e DSL. Diversamente dai comuni strumenti disponibili, Xtext genera non solo un parser ma anche un class model per l'Abstract Syntax Tree, fornendo anche un IDE, ovvero un ambiente di sviluppo, basato su Eclipse completo di tutte le funzioni e personalizzabile. Lo sviluppo di un linguaggio attraverso Xtext si basa sulla scrittura di una grammatica che lo definisce in tutte le sue parti, essa permette di ottenere un'infrastruttura completa inclusa di parser, linker, typechecker e compilatore insieme con il supporto all'editing per Eclipse. L'utilizzo di un framework come Xtext offre enormi vantaggi agli sviluppatori di DSL ed è per questo motivo che è stato scelto per la creazione del compilatore source-to-source di Fly.

3.2 Architettura

L'architettura di Fly è stata realizzata in modo tale che le risorse dell'applicativo sviluppato dall'utente vengano stabilite in maniera automatica in base ai requisiti computazionali. Essa supporta due modelli di esecuzione: in locale ed in Cloud. Quando si esegue l'applicativo Fly in locale, è la macchina ospitante a farsi carico dell'intero carico computazionale richiesto mentre se lo stesso applicativo viene eseguito in ambiente Cloud allora l'esecuzione

del codice avviene attraverso un'architettura di calcolo parallelo grazie a costrutti specifici atti a facilitare l'esperienza dell'utente. Fly risulta un linguaggio staticamente e fortemente tipizzato che sfrutta varie tipologie di inferenza per determinare il tipo delle variabili e delle costanti dichiarate. Un programma scritto in Fly viene tradotto in automatico dal compilatore in codice Java, nello specifico ogni tipo di dato presente all'interno di Fly ha un suo corrispettivo in quest'ultimo linguaggio, così come le varie funzionalità specifiche del dominio vengono realizzate sfruttando l'ampia gamma di metodi offerti dal linguaggio General Purpose.

L'elemento fondamentale di Fly è rappresentato dal concetto di **funzione Fly**, ovvero una porzione di codice indipendente che eseguita in maniera concorrente, similmente a quanto avviene con le funzioni Serverless del modello FaaS. Le funzioni Fly possono essere eseguite in sequenziale o in parallelo, sia su infrastruttura locale multi-processore che su Cloud mediante l'utilizzo di appositi costrutti che ne permette la definizione, l'esecuzione, la sincronizzazione e la gestione della comunicazione. Quest'ultima viene resa disponibile anche tra processi differenti in esecuzione su diversi ambienti attraverso l'uso di appositi canali di comunicazione virtuali i quali rappresentano il metodo principale per lo scambio di dati tra diverse funzioni Fly.

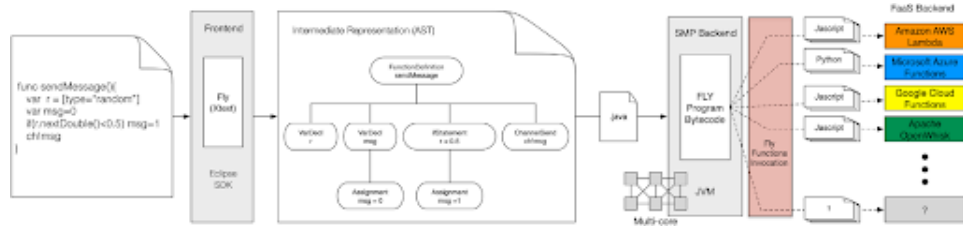


Figura 3.1: Flusso di compilazione di Fly.

La Figura 3.1 mostra il **flusso di una compilazione** in Fly. Partendo da sinistra il programma Fly viene fornito in input al compilatore che genera un *Abstract Syntax Tree (AST)*, ovvero un albero rappresentante la struttura sintattica del codice in cui ogni costrutto corrisponde ad un nodo. Viene definito astratto in quanto non vengono rappresentati tutti i dettagli del codice sorgente ma solo la sua struttura. In seguito la rappresentazione in AST intermedia viene trasformata in un programma Java da cui vengono estratte le singole funzioni Fly, ognuna delle quali verrà tradotta in diversi codici eseguibili, uno per ogni ambiente dichiarato dall'utente. Infine, osservando il lato destro della Figura 3.1 è possibile osservare l'output finale che consiste

nel codice compilato delle funzioni Fly pronto per essere eseguito sui vari ambienti dichiarati.

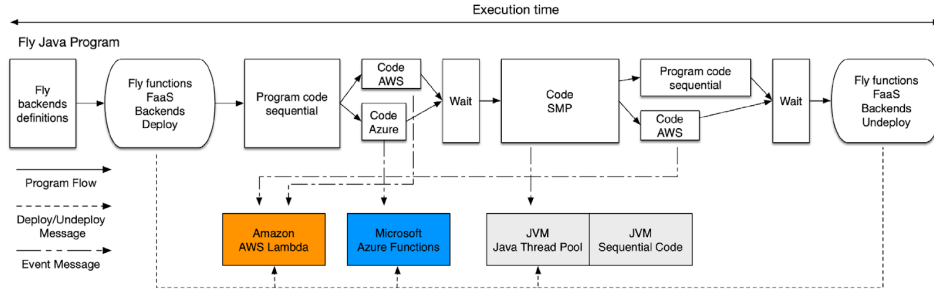


Figura 3.2: Flusso di esecuzione di Fly.

Nella Figura 3.2 viene mostrato nel dettaglio il flusso di esecuzione che inizia al lancio di un programma Fly. Seguendo il tempo di esecuzione, la prima fase prevede l’inizializzazione degli ambienti di back-end dichiarati all’interno del codice, questa prevede il login sul Cloud provider e l’istanziamento dei servizi necessari. Il codice generato a partire dalla funzione Fly viene poi caricato sul rispettivo ambiente Cloud, esso risulterà già compilato nel momento in cui il programma principale viene lanciato, in questo modo si evitano i tempi di attesa che sarebbero causati da una compilazione a tempo di esecuzione. Effettuata la fase di inizializzazione il programma principale viene mandato in esecuzione seguendo le istruzioni contenute nel codice Fly e al suo termine vengono effettuate una serie di procedure di undeploy sull’ambiente di back-end, avente lo scopo di eliminare tutte le istanze dei servizi Cloud creati in precedenza.

3.3 Definizione del linguaggio

Fly fornisce diversi tipi di dati che possono essere raggruppati in due insiemi, *basici* e di *dominio*. I primi sono ereditati da Java e comprendono *booleani*, *interi*, *reali* e *stringhe*, essi possono essere usati anche per la dichiarazione di array monodimensionali, bidimensionali e tridimensionali. Di numero maggiore sono invece i tipi di dominio utilizzabili in Fly che permettono all’utente di interagire e comunicare con l’ambiente di esecuzione.

Il Listato 3.1 mostra un semplice esempio di programma Fly per il calcolo di una stima di Pi Greco tramite il metodo Monte Carlo su ambiente Amazon Web Services, in particolare sfruttando il servizio AWS Lambda [8]. È utile fornire una prima descrizione del codice i cui elementi verranno poi appro-

fonditi in seguito. Alla Riga 1 troviamo la dichiarazione dell'ambiente di esecuzione, in questo caso viene definito su AWS. Su tale ambiente è dichiarato un `channel`, Riga 3, che permette al programma principale di comunicare con la funzione `Fly hit` definita alla Riga 5, la quale genera un punto casuale e calcola se appartiene o meno ad un cerchio che abbia origine al punto 1.0, realizzando l'algoritmo per il metodo Monte Carlo. Il valore ottenuto viene poi inviata sul canale `ch`. La funzione `estimation`, Linea 16, legge l'output della funzione `hit` e scrive a schermo la stima di Pi ottenuta. Alla linea 28 è possibile vedere come sono lanciate le funzioni `Fly`, in particolare viene utilizzata la parola chiave `fly` per eseguire 10000 funzioni `hit` sull'ambiente AWS. Infine l'utilizzo della parola chiave `thenall` fa in modo che quando tutte le funzioni `hit` hanno terminato venga eseguita la funzione `estimation`.

```

1  var local = [type="smp", nthread=4]
2
3  var cloud = [type="aws", user="user_name", access_id_key = "
   access_id_key", secret_access_key="secret_access_key", region="eu-
   west-2", language="nodejs12.x", thread=2, memory=256, time_=300]
4
5  var ch = [type="channel"] on cloud
6
7  func hit(){
8      var r = [type="random"]
9      var x = r.nextDouble()
10     var y = r.nextDouble()
11     var msg = 0
12
13     if((x * x)+(y * y) < 1.0){msg = 1}
14     ch!msg on cloud
15 }
16
17
18 func estimation(){
19     var sum = 0
20     var crt = 0
21     for i in [0:2] {
22         sum += ch? as Integer
23         crt += 1
24     }
25     println "pi estimation: " + (sum*4.0) \ crt
26 }
27
28 fly hit in [0:2] on cloud thenall estimation

```

Listing 3.1: Stima di PI Greco usando il metodo Monte Carlo su ambiente AWS.

Object

Il principale tipo di dato di dominio è il tipo **object**, un insieme eterogeneo di elementi di tipo basico e/o di dominio. Un Fly **object** può essere visto come l'unione di un **array**, un classico vettore, e di una **mappa**, ovvero una struttura dati in grado di memorizzare elementi nella forma di coppie chiave-valore. Il valore di un elemento può essere ottenuto in due diversi modi, utilizzando la sua posizione, come si fa per gli array, o la sua chiave, come nel caso di una mappa. Quando un nuovo valore è assegnato ad una data chiave o posizione viene creato un nuovo elemento, altrimenti il nuovo valore va a sostituire il precedente.

Ogni tipo di dato di dominio presente in Fly è un'istanza del tipo **object**, questo fa sì che essi siano costruiti tutti con una sintassi simile utilizzando il campo **type** per specificarne il tipo.

Ambiente di esecuzione

Il codice di un programma Fly inizia sempre con la dichiarazione di uno o più **ambienti di esecuzione**, necessari affinché il generatore possa configurare le risorse utili al funzionamento dell'applicazione, in particolare è sempre necessario un ambiente di esecuzione locale corrispondente alla macchina su cui viene lanciato il programma. Ad esempio nel caso di un ambiente locale multiprocessore viene utilizzata una **Java Thread Pool** mentre per gli ambienti su Cloud è necessario creare le istanze dei servizi necessari. Il grande vantaggio offerto da Fly è l'astrazione, dal punto di vista dello sviluppatore infatti tutti i tipi di ambienti possono essere utilizzati allo stesso modo, sarà il compilatore ad occuparsi dei dettagli implementativi specifici relativi al loro utilizzo.

La dichiarazione di un ambiente di esecuzione necessita di una serie di parametri che variano in base alla tipologia di ambiente, specificata dal campo **type**. Oltre ad eventuali informazioni aggiuntive richieste dai diversi Cloud provider per accedere ai loro servizi, la dichiarazione relativa ad un'esecuzione su Cloud richiede che l'utente specifichi il linguaggio di programmazione in cui verranno tradotte le funzioni Fly per essere lanciate sul servizio Serverless del provider. In particolare le possibilità sono JavaScript [32] o Python [52]. Attualmente gli ambienti supportati sono tre:

- **smp** - ambiente locale che sfrutta il parallelismo basato su un sistema multiprocessore simmetrico sfruttando i thread Java;

- **aws** - ambiente su Cloud legato al provider Amazon Web Services (AWS) [4] che sfrutta il servizio AWS Lambda [8] per eseguire le funzioni Fly;
- **azure** - ambiente su Cloud legato al provider Microsoft Azure [43] che sfrutta il servizio Azure Function [15] per eseguire le funzioni Fly. In questo caso il supporto a Python non è previsto in quanto al momento dell'implementazione non ancora reso disponibile da Azure.

Channel

La comunicazione e la sincronizzazione sia tra le varie funzioni Fly, sia tra queste e il programma principale avviene mediante la dichiarazione di un `type="channel"` che necessita di specificare l'ambiente di esecuzione sul quale funzionare mediante la parola chiave `on`. Questo metodo di comunicazione segue il modello delle code di messaggi bloccanti, ovvero quando si tenta di ricevere un messaggio l'esecuzione si ferma fino a che un nuovo messaggio non viene ricevuto. L'utilizzo del carattere "!" permette l'invio di messaggi, vedi Riga 13 del Listato 3.1, mentre il carattere "?" è necessario per la loro ricezione, vedi Riga 22 del Listato 3.1. Fly implementa un meccanismo di serializzazione in quando la comunicazione utilizza le infrastrutture di rete per scambiare messaggi con l'ambiente su Cloud.

Funzione Fly

Le **funzioni Fly** sono già state in parte descritte, esse sono frammenti di codice scritto per realizzare una funzionalità specifica indipendente dal programma principale e da altre funzioni ed eseguibile in maniera concorrente. Queste funzioni sono ben diverse da quelle disponibili in altri linguaggi di programmazione e prendono ispirazione da quelle utilizzate nei linguaggi funzionali.

La dichiarazione di una funzione Fly è possibile utilizzando la parola chiave `func` a seguito della quale si definisce il nome della funzione e si inseriscono eventuali parametri di input tra parentesi che vengono passati per copia e sono considerati immutabili. Le funzioni Fly possono restituire un valore mediante l'uso della parola chiave `return` ed hanno scoping privato, ovvero solo i parametri della funzione e le variabili locali sono visibili all'interno del corpo della funzione. Per superare tale limitazione possono essere utilizzati sia gli oggetti `channel` che le costanti a patto che la dichiarazione e l'accesso avvengano nello stesso ambiente di esecuzione. Questo significa che se una funzione è in esecuzione sull'ambiente X essa può utilizzare canali e oggetti

disponibili su tale ambiente X a prescindere da chi li abbia dichiarati. Le funzioni Fly possono essere eseguite in modo concorrente utilizzando la parola chiave `fly`, non essendo ammessa la ricorsione essa non può essere utilizzata all'interno del corpo di una funzione. Il suo utilizzo causa la generazione di un evento sull'ambiente che si sta utilizzando, sia esso SMP o su Cloud, in modo che le funzioni vengano eseguite, in linea con il modello di programmazione event-driven che caratterizza il Serverless Computing. Il parallelismo esplicito da cui è caratterizzato Fly è racchiuso nel modo in cui vengono lanciate le funzioni, a seguito della parola chiave `fly` viene dichiarata il nome della funzione da eseguire e, se necessario, il loro numero insieme alla variabile che rappresenta l'ambiente di esecuzione da utilizzare specificato in seguito alla parola chiave `on`.

Funzione di Callback

Le **funzioni di Callback** sono funzioni Fly che vengono eseguite al termine dell'esecuzione di una precedente funzione. Esse possono essere dichiarate dopo la specifica dell'ambiente di esecuzione su cui deve essere eseguita una funzione Fly. Due tipi di funzioni di Callback sono supportate, la prima riguarda quelle specificate in seguito alla parola chiave `then` che indica che la sua esecuzione deve avvenire dopo ogni singola funzione Fly, mentre utilizzando la parola chiave `thenall` si richiede che la funzione venga eseguita solamente quando tutte le funzioni Fly hanno terminato.

Esecuzioni asincrone

Fly permette **esecuzioni asincrone** attraverso l'uso della parola chiave `async` e tipo di dato di dominio chiamato `async-object` che permette all'utente di controllare ed interagire esse. Invocando una funzione utilizzando `async` viene restituito immediatamente il controllo al programma principale in modo che l'esecuzione possa continuare, a questo punto l'utente può controllare lo stato delle funzioni asincrone usando il metodo `status()` dell'`async-object`, mentre il metodo `wait()` mette in pausa l'esecuzione fino a che tutte le funzioni non hanno terminato.

Codice nativo

La possibilità di includere **codice nativo** all'interno di applicazioni Fly è data dalla parola chiave `native`, tale funzionalità consente, ad esempio, di inserire in una funzione Fly codice scritto direttamente in Python ed esso non verrà tradotto o modificato dal compilatore Fly ma copiato così com'è.

Tramite la parola chiave **require** è inoltre consentito includere ed installare librerie esterne addizionali nell'ambiente di esecuzione.

3.3.1 Struttura di un progetto Fly

Fly utilizza il gestore di pacchetti *Maven* [41] per la costruzione e la generazione dei progetti. Maven è uno strumento per la gestione di progetti software e di build automation basato sul concetto di **Project Object Model (POM)**, un file XML che contiene le dipendenze necessarie al funzionamento dell'applicazione.

La creazione di un **progetto Fly** genera due cartelle principali insieme con un file POM, il tutto racchiuso in un progetto Java Maven che include tutte le dipendenze, il programma principale e il codice delle funzioni, il tutto costruito sulla base del programma Fly da parte del compilatore. Attraverso il comando `mvn package` di Maven tale progetto viene utilizzato per la generazione del file *JAR* eseguibile [19].

Un progetto Fly è così strutturato:

- **cartella *src*** - cartella in cui è posizionato il file con estensione *.fly* contenente il codice del programma scritto in linguaggio Fly;
- **cartella *src-gen*** - cartella contenente i file generati dal compilatore che consistono in un file Java e due file di script. Il file Java ha il compito di orchestrare l'intera esecuzione del programma, ovvero l'intero ciclo di vita comprensivo del lancio dei due file di script. Questi ultimi sono file con estensione *.sh* e contengono distintamente i comandi per il deploy delle funzioni Fly e le procedure per l'undeploy. Il file di deploy si occupa di creare il file necessario alla creazione della funzione Serverless sul servizio FaaS del provider definito, comprensivo di codice, librerie e file di configurazione, mentre il file di undeploy ha il compito di ripulire l'ambiente Cloud dalle risorse create;
- **file XML *pom.xml*** - unità fondamentale di Maven per la gestione del progetto, si tratta di un file XML che contiene informazioni e i dettagli di configurazione necessari per effettuare la build, nel caso specifico al suo interno sono presenti le librerie necessarie al programma [42].

3.4 Generazione del codice

Il punto di forza di Fly è quello di fornire un livello di astrazione tale per cui l'utente non deve conoscere le API di ogni Cloud provider per scrivere un

programma che sfrutti al meglio i suoi servizi. Tali API restano comunque essenziali per interagire con essi, infatti è il compilatore di Fly che si occupa di prendere in input il programma per tradurlo in codice effettivamente eseguibile. È quello che avviene nella fase di generazione del codice in cui ogni componente scritta in linguaggio Fly viene trasformata nel linguaggio di destinazione, il quale varia a seconda del contesto, come visto nei paragrafi precedenti. Possiamo quindi affermare che il fulcro del compilatore di Fly è sicuramente la parte di generazione di codice che produce un programma diverso in base all'ambiente di esecuzione dichiarato dall'utente. Nel caso di un ambiente multiprocessore il parallelismo viene implementato attraverso l'uso di una **Java Thread Pool** che permette l'utilizzo di thread ai quali viene assegnata una specifica esecuzione, in particolare essa permette di riutilizzare thread precedentemente creati, eliminando il tempo necessario alla loro creazione. Il programma Fly viene quindi eseguito su una *Java Virtual Machine (JVM)* come un classico programma Java, sulla quale viene eseguito anche l'ambiente SMP, assumendo quindi che siano disponibili almeno due core fisici, uno per l'esecuzione del programma e uno per l'ambiente. L'esecuzione su Cloud prevede invece l'utilizzo delle API fornite dal provider di riferimento che consentono di interagire con i servizi disponibili in base ai quali vengono tradotti i vari costrutti e funzionalità di Fly.

Il codice generato per le funzioni Fly varia in base all'ambiente di esecuzione scelto, in particolare il linguaggio di destinazione può essere Java, JavaScript o Python in base a quanto specificato dall'utente. In caso di esecuzione su ambiente SMP locale le funzioni sono tradotte in Java, in quanto eseguite su JVM, diversamente dal Cloud i cui servizi FaaS permettono solitamente l'esecuzione di codice in JavaScript o Python. Oltre al linguaggio utilizzato, l'ambiente di destinazione determina anche il modo in cui viene generato lo script di deploy che costruisce il pacchetto di esecuzione contenente codice sorgente e librerie.

Durante la fase di esecuzione l'interazione con il Cloud avviene mediante l'utilizzo degli strumenti di **Command Line Interface** forniti dai vari provider come la *AWS CLI* [6] o la *Azure CLI* [12], mentre le funzioni vengono invocate con delle chiamate *HTTP POST* asincrone, le quali assicurano la minor latenza permettendo al programma di continuare la sua esecuzione in attesa di risposta da parte del servizio.

Amazon Web Services

Amazon Web Services mette a disposizione diverse SDK che permettono di interagire con i suoi servizi praticamente con tutti i principali linguaggi di

programmazione [9]. Il programma principale, essendo scritto in Java fa uso delle *AWS SDK for Java* [55] che consentono di accedere a tutti i servizi forniti da AWS mediante delle API dedicate. Gli script di deploy e undeploy invece utilizzano la AWS CLI [6] per creare le istanze di servizi necessarie al funzionamento del programma e per caricare le funzioni Fly e i relativi pacchetti su AWS Lambda. Queste ultime sfruttano la libreria *boto3* per Python [16] e la libreria *aws-sdk* per JavaScript [56] per interagire con i servizi di AWS.

Microsoft Azure

Le SDK di Microsoft Azure soffrono di alcune problematiche di utilizzo derivanti sia dalla difficile integrazione al di fuori dell'IDE VisualStudio Code, sia dalla mancanza di funzionalità per gran parte dei servizi. Per questo motivo il supporto ad Azure passa per l'utilizzo di un servizio REST API che prescinde dal linguaggio di programmazione in quanto basato su operazioni HTML. Le *Representational State Transfer (REST) API* consistono in endpoint specifici per ogni servizio che supportano un insieme di operazioni HTTP i quali forniscono l'accesso alle principali funzionalità di un servizio come la creazione, l'aggiornamento, la cancellazione o l'ottenimento di informazioni. L'utilizzo delle REST API è regolato dall'utilizzo di un token di autorizzazione ottenibile tramite il servizio Azure Active Directory che ne permette l'acquisizione fornendo i propri dati di accesso, in particolare ID, password e TenantID.

Le sezioni seguenti analizzano come il codice scritto in Fly viene tradotto dal compilatore nei vari linguaggi di destinazione per generare i file eseguibili. Tutti i Listati mostrati fanno riferimento al programma per il calcolo di Pi Greco, proposto nella sua interezza nel Listato 3.1.

3.4.1 Ambiente di esecuzione

Ambiente locale - SMP

La dichiarazione di un **ambiente locale** necessita di due parametri, il primo definisce il tipo di ambiente che sarà di `type="smp"`, il secondo specifica il numero di thread che dovranno essere utilizzati.

```
1 var local = [type = "smp", nthread = 4]
```

Listing 3.2: Dichiarazione di un ambiente di esecuzione locale.

Il codice Java generato risulta in un oggetto `ExecutorService` che semplifica l'esecuzione asincrona, permettendo di eseguire i task in modo concorrente. Esso fornisce automaticamente un insieme di thread con una serie di metodi che permettono di assegnare loro dei task.

```
1 static ExecutorService __thread_pool_local = Executors.  
    newFixedThreadPool(4);
```

Listing 3.3: Codice generato per l'ambiente di esecuzione locale.

Ambiente su Cloud - Amazon Web Services

L'utilizzo di un ambiente di esecuzione che utilizzi il **Cloud AWS** è possibile mediante la dichiarazione di una variabile di `type="aws"` la quale necessita di una serie di parametri corrispondenti alle caratteristiche necessarie al lancio delle funzioni Serverless su AWS Lambda [18]:

- **dati di accesso dell'account AWS:** in ordine sono necessari `user_name`, `access_id_key`, `secret_access_key`;
- **regione:** id della regione su cui si vogliono lanciare i servizi;
- **linguaggio:** linguaggio di programmazione in cui verranno tradotte ed eseguite le funzioni Lambda. Può essere `nodejs` per JavaScript o `python` per Python;

- **thread:** numero di istanze concorrenti da lanciare per l'esecuzione delle funzioni;
- **memoria:** quantità di memoria disponibile per l'esecuzione di una funzione;
- **time:** tempo limite di esecuzione per ogni funzione.

```
1 var cloud = [type="aws", user="user_name", access_id_key = "
    access_id_key", secret_access_key="secret_access_key", region="eu-
    west-2", language="nodejs12.x", thread=2, memory=256, time_=300]
```

Listing 3.4: Dichiarazione di un ambiente Cloud su AWS.

Nel Listato 3.5 vediamo come vengono istanziati i servizi necessari all'esecuzione di un programma Fly utilizzando le informazioni fornite nella dichiarazione dell'ambiente. Tra i servizi troviamo AWS SQS [2] per le code di messaggi, AWS IAM [7] per l'autenticazione, AWS S3 [3] per la memorizzazione di dati e AWS Lambda [8] per l'esecuzione di funzioni Serverless.

```
1 static BasicAWSCredentials cloud = new BasicAWSCredentials("
    access_id_key", "secret_access_key");
2
3 static AmazonSQS __sqs_cloud = AmazonSQSClient.builder()
4     .withCredentials(new AWSSStaticCredentialsProvider(cloud))
5     .withRegion("eu-west-2")
6     .build();
7
8 static AmazonIdentityManagement __iam_cloud =
9     AmazonIdentityManagementClientBuilder.standard()
10    .withCredentials(new AWSSStaticCredentialsProvider(cloud))
11    .withRegion("eu-west-2")
12    .build();
13
14 static AWSLambda __lambda_cloud = AWSLambdaClientBuilder.standard()
15    .withCredentials(new AWSSStaticCredentialsProvider(cloud))
16    .withRegion("eu-west-2")
17    .build();
18
19 static AmazonS3 __s3_cloud = AmazonS3Client.builder()
20    .withCredentials(new AWSSStaticCredentialsProvider(cloud))
21    .withRegion("eu-west-2")
22    .build();
```

Listing 3.5: Codice generato per l'ambiente Cloud su AWS.

Ambiente su Cloud - Microsoft Azure

Utilizzare i servizi a disposizione sul **Cloud di Microsoft Azure** mediante linguaggi di programmazione risulta complesso e macchinoso a causa delle

SDK fornite che, attualmente, presentano molteplici mancanze oltre ad una documentazione particolarmente scarna e spesso non aggiornata. Per tale motivo si è scelto di utilizzare il sistema di REST API attraverso delle chiamate *HTTP*. L'integrazione dei servizi di Microsoft Azure all'interno del codice Java generato dal compilatore Fly è resa possibile grazie alla libreria *AzureClient* sviluppata presso l'ISISLab [29], la quale permette di superare della frammentazione delle SDK fornendo un'unica interfaccia che permette un più agevole utilizzo dei servizi. La gestione dei servizi di Azure con JavaScript soffre dei medesimi problemi in termini di SDK ed anche in questo caso è necessario utilizzare delle chiamate *HTTP* per utilizzarli, in particolare vengono utilizzate le librerie *axios* [10] e *qs* [53]. In entrambi i casi rimane indispensabile il servizio Azure AD [11] per ottenere il *token* di autorizzazione che permette di effettuare tali chiamate.

L'utilizzo di un ambiente di esecuzione che utilizzi il **Cloud Azure** è possibile mediante la dichiarazione di una variabile di `type="azure"` la quale necessita di una serie di parametri corrispondenti alle caratteristiche necessarie al lancio delle funzioni Serverless sul servizio Azure Function [14]:il problema

- **dati di accesso dell'account Azure:** in ordine sono necessari `client_id`, `tenant_id`, `secret_key`, `subscription_id`;
- **regione:** id della regione su cui si vogliono lanciare i servizi;
- **linguaggio:** linguaggio di programmazione in cui verranno tradotte ed eseguite le funzioni Lambda. Può essere `nodejs` per JavaScript o `python` per Python;
- **thread:** numero di istanze concorrenti da lanciare per l'esecuzione delle funzioni;
- **time:** tempo limite di esecuzione per ogni funzione.

```
1 var cloud = [type="azure", clientID="client_id", tenantID="tenant_id",
  secret_key="secret_key", subscriptionID="subscription_id", region
  ="France Central", language="nodejs12.x", threads="2", seconds="
  300"]
```

Listing 3.6: Dichiarazione di un ambiente Cloud su Azure.

Il codice generato, visibile nel Listato 3.7, comprende la creazione di una variabile istanza della libreria *AzureClient* necessaria per avviare tutte le procedure relative ai servizi di Azure. In particolare il metodo `init()` si occupa di istanziare tutti i servizi di base indispensabili per l'esecuzione di qualsiasi

programma Fly, sfruttando le informazioni fornite nella dichiarazione dell'ambiente per effettuare l'accesso al Cloud. Il metodo `createFunctionApp(...)` invece è adibito alla creazione dell'istanza del servizio Serverless di Azure che permette l'esecuzione delle funzioni.

```
1 cloud = new AzureClient("client_id",
2   "tenant_id",
3   "secret_key",
4   "subscription_id",
5   __id_execution+"",
6   "France Central");
7
8 cloud.init();
9
10 cloud.createFunctionApp("flyappcloud", "nodejs12.x");
```

Listing 3.7: Codice generato per l'ambiente Cloud su Azure.

3.4.2 Channel

La dichiarazione di un `type="channel"`, ovvero il mezzo di comunicazione e sincronizzazione utilizzato da Fly, necessita come unica specifica l'ambiente di esecuzione su cui dovrà funzionare, come mostrato nel Listato 3.8. Questo viene specificato mediante la parola chiave `on` e in base ad esso il compilatore genererà un codice differente, adattandolo in modo che sfrutti i servizi di gestione delle code forniti dall'ambiente specificato. In particolare per AWS si utilizza il servizio AWS Simple Queue Service (SQS) [2] mentre per Azure si sfruttano i metodi presenti all'interno della libreria `AzureClient`.

La logica di funzionamento dei prevede che ad uno dei thread dell'ambiente locale venga associato il task di lettura della coda rimanendo costantemente in ascolto e non appena vengono intercettati dei messaggi essi vengono immediatamente scritti sul canale sul quale è possibile leggerli.

```
1 var ch = [type="channel"] on cloud
```

Listing 3.8: Dichiarazione di un Channel su Cloud.

3.4.3 Script di deploy e undeploy delle funzioni

Il funzionamento delle funzioni Serverless su un ambiente Cloud richiede una serie di procedure sia per il deploy che per l'undeploy che vengono eseguite mediante gli strumenti di Command Line Interface forniti dai provider, di conseguenza i comandi cambiano in base al Cloud ma al linguaggio utilizzato.

Nonostante ciò queste procedure non sono particolarmente diverse tra loro in quanto tutti necessitano, oltre che del codice della funzione, di una serie di file in formato JSON sia per la specifica del ruolo e delle autorizzazione con le quali la funzione viene eseguita sia per indicare le dipendenze del codice. Fly costruisce l'intero pacchetto per l'esecuzione delle funzioni Serverless prima che queste vengano caricate su Cloud, in particolare questo contiene solo i vari documenti JSON ma anche le librerie necessarie per l'esecuzione che vengono installate mediante il gestore di pacchetti *npm* [47] per JavaScript e *pip* [61] per Python.

Il codice contenuto all'interno degli script può essere suddiviso in sezioni, ognuna delle quali si occupa di una fase necessaria per il funzionamento del programma, descritte di seguito.

- **Verifica dei parametri e dei requisiti** La prima parte dello script si occupa di verificare che siano presenti i parametri necessari all'interazione con l'ambiente di esecuzione per poi salvarli in apposite variabili. Vengono poi effettuati una serie di controlli per assicurarsi che tutte i pacchetti necessari siano installati sulla macchina, come ad esempio le Command Line Interface.
- **Inizializzazione dei servizi** Attraverso la CLI viene inizializzato l'ambiente del provider di riferimento utilizzando i dati di accesso forniti nella dichiarazione dell'ambiente. In particolare per AWS è necessario creare un'istanza del servizio IAM [7] mentre per Azure si effettua semplicemente il login.
- **Creazione del virtual env - Python** In caso il linguaggio scelto per la funzione sia Python è necessario creare un *virtual env*, ossia di un ambiente virtuale isolato che l'interprete di Python utilizza per l'installazione di librerie e script.
- **Creazione del progetto locale** Mediante l'uso dei comandi *bash* lo script crea il progetto Fly occupandosi sia della creazione delle cartelle che dei file. È in questa fase che vengono creati i file JSON necessari per configurare il servizio FaaS, contenenti parametri come il tipo di trigger da utilizzare, nel nostro caso HTTP, i permessi e le autorizzazioni.
- **Codice della funzione Fly Serverless** La funzione Fly viene tradotta nel codice scelto, JavaScript o Python, scritto all'interno di un file che viene inserito all'interno del progetto.

- **Deploy della funzione** Le procedure di deploy della funzione cambiano in base all'ambiente scelto. AWS richiede che le librerie necessarie all'esecuzione siano già presenti all'interno del pacchetto che verrà caricato su Cloud, per questo motivo lo script utilizza i gestori di pacchetti *npm* o *pip* per la loro installazione per poi creare un file *.zip* contenente l'intero progetto che viene usato per il deploy. A causa dei limiti di dimensione che tale pacchetto deve avere viene fatto un controllo aggiuntivo e in caso di superamento si utilizza il servizio di archiviazione S3 [3]. Il deploy per Azure risulta invece meno macchinoso in quanto le librerie necessarie vengono specificate all'interno di un file di requisiti che permette di mantenere la dimensione del progetto sempre molto contenuta.
- **Undeploy della funzione** Una volta eseguite le varie funzioni Fly lo script di undeploy si occupa di eliminare tutte le istanze dei servizi creati in modo da lasciare l'ambiente Cloud pulito. Questa fase viene effettuata tramite CLI per AWS mentre per Azure si utilizza la libreria *AzureClient*, di conseguenza lo script di undeploy contiene solo i comandi per effettuare il logout.

3.4.4 Sincronizzazione

L'esecuzione delle funzioni Fly su Cloud necessita di un sistema di sincronizzazione in grado di accertarsi che tutte le funzioni Fly abbiano terminato la loro esecuzione. Questo meccanismo viene implementato attraverso una coda di terminazione sulla quale ogni funzione invia un messaggio una volta completata. Mediante l'uso di un contatore il sistema può notificare che tutte le funzioni sono terminate nel momento in cui sono ricevuti tanti messaggi quante sono le funzioni lanciate. La coda di terminazione viene implementata con la stessa logica vista per i **channel** in maniera totalmente astratta all'utente, essa viene creata, gestita ed eliminata dal programma stesso.

Capitolo 4

Ambiente di esecuzione Kubernetes

4.1 Kubernetes in Fly

Fly permette all'utente di sviluppare ed eseguire i propri applicativi senza preoccuparsi della gestione delle risorse, dell'ambiente su cui viene eseguito e tutto ciò che riguarda il ciclo di vita dell'applicativo stesso. Ciò è possibile grazie al paradigma FaaS (Function as a Service), il quale permette di sfruttare i servizi offerti dai vari cloud provider per eseguire applicativi in ambienti serverless. Sebbene ciò abbia portato innumerevoli vantaggi nell'esecuzione e nella gestione del costruito Fly, si riscontrano comunque delle problematiche e delle limitazioni:

- gli ambienti serverless offerti dai diversi cloud provider limitano il tempo di esecuzione degli applicativi;
- le risorse disponibili per l'esecuzione del codice sono predefinite;
- l'ambiente in cui viene eseguito l'applicativo Fly è gestito dal Cloud Provider stesso. Un livello di astrazione così alto comporta l'impossibilità di gestire possibili errori e avere il pieno controllo dell'ambiente di esecuzione.

L'introduzione di un ambiente di esecuzione basato su Kubernetes all'interno di Fly permette di risolvere alcune di queste problematiche permettendo l'esecuzione delle funzioni Fly all'interno di un cluster Kubernetes. Ciò permette di avere un flusso di esecuzione simile a quello eseguito su ambiente serverless garantendo però un maggior controllo e meno limitazioni. La

tecnologia sfruttata dai servizi di serverless computing, infatti, è basata sul lancio di container, ognuno dei quali si occupa dell'esecuzione di una singola funzione.

4.2 Integrazione in Fly

L'integrazione all'interno del linguaggio Fly, è stata progettata in modo da seguire le medesime sintassi e semantiche utilizzate per gli altri ambienti di esecuzione. Nello specifico, è stata introdotta l'entità **K8s** la quale richiede tre campi necessari come si può leggere a Linea 4 del Listato 4.1

- **type** indica il tipo di ambiente di esecuzione che stiamo utilizzando. Nel nostro caso equivale a **K8s**;
- **clusterName** indica il nome del cluster di riferimento per l'esecuzione dell'applicativo Fly;
- **registryName** è un campo particolare in cui l'utente definisce il nome del registro nel quale verranno caricate le immagini docker necessarie all'esecuzione dell'applicativo Fly.

Per mantenere coerenza con Fly, si è preferito conservare la possibilità di scegliere come eseguire il proprio applicativo, ovvero se in locale o su cloud. L'utente può infatti decidere fra computare la propria applicazione su un cluster in locale, oppure su un cloud provider, come ad esempio AWS o Azure.

- **type** indica il tipo di ambiente di esecuzione che stiamo utilizzando. Nel nostro caso equivale a **sm** per un'esecuzione in locale oppure **aws** per un'esecuzione su AWS oppure **Azure** per un'esecuzione su Azure;
- **threads** indica il numero di repliche dell'applicativo da eseguire in parallelo sul cluster.
- **language** indica il linguaggio nel quale l'applicativo Fly deve essere tradotto ed eseguito.

Il Listato 4.1 mostra un esempio del programma per il calcolo di pi greco su un ambiente **k8s**

```
1  # Computazione del pi greco
2  var cloud = [type="azure", language="nodejs", threads=100]
3
```

```

4      var cluster = [type="k8s",clusterName="Fly",registryName="
FlyRegistry.azurecr.io"] on cloud
5
6      var ch = [type="channel"] on cloud
7
8      func pi(){
9          var r = [type="random"]
10         var x = r.nextDouble()
11         var y = r.nextDouble()
12         var msg = 0
13         if((x * x)+(y * y) < 1.0){msg = 1}
14         ch!msg on cluster
15     }
16     func estimation(){
17         var sum = 0
18         var crt = 0
19         for i in [0:100] {
20             sum += ch? as Integer
21             crt += 1
22         }
23         println "pi estimation: " + (sum*4.0) / crt
24     }
25
26     fly pi in [0:100] on cluster thenall estimation

```

Listing 4.1: Stima di PI Greco usando il metodo Monte Carlo su ambiente k8s su Azure

4.3 Implementazione

L'implementazione dell'ambiente di esecuzione Kubernetes è stata realizzata in modo tale che tutti i processi interni che consentono il corretto funzionamento della computazione, venissero realizzati sfruttando gran parte dei tools e degli oggetti forniti da Kubernetes stesso, elencati e descritti di seguito:

- **Pod** Un Pod costituisce la più semplice, piccola e basilare unità di esecuzione in Kubernetes. Essa verrà utilizzata per incapsulare ed eseguire il container della risorsa necessaria al corretto funzionamento dell'applicativo Fly.
- **ReplicaSet** Un ReplicaSet ha il compito specifico di mantenere attive una o più copie di un Pod. Esso verrà utilizzato per avere più copie dello stesso applicativo Fly in modo da poter innescare meccanismi di calcolo parallelo.
- **Job** Un oggetto di tipo Job permette di eseguire e gestire task complessi all'interno del cluster. Sfrutteremo questa sua peculiarità per poter soddisfare la richiesta di computazione parallela.

- **NodePort** I Pod in Kubernetes sono entità effimere, essi vengono creati e distrutti in base allo stato desiderato del cluster, per natura quindi non hanno un'identificazione statica all'interno della rete del cluster ma ad ogni Pod viene assegnato un indirizzo IP dinamico che varia nel tempo. Utilizzeremo un NodePort per poter interagire con essi.
- **LoadBalancer** Un LoadBalancer permette di esporre i Pod all'esterno del cluster. Utilizzeremo un LoadBalancer nel caso in cui il cluster sia fornito da un Cloud Provider.
- **Deployment** L'oggetto Deployment si occupa di creare e gestire il ciclo di vita degli Oggetti all'interno del cluster. Utilizzeremo questa componente per definire lo stato ed i comportamenti del cluster.
- **Kubectl** Kubectl è un potente tool da linea di comando fornito da Kubernetes che permette di eseguire comandi e chiamate al server API kube. Lo utilizzeremo per comunicare con il cluster Kubernetes.

4.4 Generazione codice

Una volta dichiarata l'entità K8s, l'utente dunque può decidere sia il linguaggio da utilizzare, che se eseguire l'applicativo in un cluster locale o un cluster in cloud ed anche quanti processi dello stesso applicativo eseguire in maniera parallela. Verranno quindi generati diversi file YAML, che conterranno le informazioni necessarie ad una corretta configurazione dello stato desiderato del cluster ed a seconda del linguaggio scelto, verranno richiamati due differenti generatori: un generatore per il codice JavaScript ed un generatore per il codice Python.

JavaScript

Il metodo principale, responsabile non solo della generazione del codice Javascript ma anche di tutto il processo di esecuzione è: `k8sDeploy()`. Nello specifico, genererà le seguenti componenti:

- **redis.yml** il file di deployment per l'immagine ed il servizio Redis;
- **Dockerfile** il file Docker per generare l'immagine dell'ambiente Fly;
- **node.yml** il file di deployment contenente tutte le informazioni necessarie all'esecuzione dell'ambiente all'interno del cluster;

Ed infine verrà generato il file `main.js` che conterrà il codice Fly scritto dall'utente e tradotto in linguaggio JavaScript con una configurazione automatizzata in modo da poter interagire con la coda Redis.

Python

Analogamente al generatore JavaScript, il generatore Python attraverso il metodo `k8sDeploy()` genererà tutte le risorse necessarie all'esecuzione dell'applicativo Fly:

- **redis.yml** il file di deployment per l'immagine ed il servizio Redis;
- **Dockerfile** il file Docker per generare l'immagine dell'ambiente Fly;
- **python.yml** il file di deployment contenente tutte le informazioni necessarie all'esecuzione dell'ambiente all'interno del cluster;

Ed infine verrà generato il file `main.py` che conterrà il codice Fly scritto dall'utente e tradotto in linguaggio Python con una configurazione automatizzata in modo da poter interagire con la coda Redis.

Capitolo 5

Conclusioni

Il Cloud Computing ha avuto una rapida e veloce espansione negli ultimi anni grazie alla sua innovativa forma di erogazione delle risorse: fornire e richiedere risorse solo quando strettamente necessarie. Grazie a tale intuizione, i diversi Cloud Provider hanno iniziato a fornire servizi di tipo Serverless ossia servizi in cui le applicazioni vengono avviate solo quando necessario. Quando un evento attiva l'esecuzione del codice, il Cloud Provider assegna dinamicamente le risorse all'applicazione fino al termine dell'esecuzione. Fly è un linguaggio Domain-Specific che si basa proprio su tale paradigma, l'utente così non deve preoccuparsi della gestione delle risorse, dell'ambiente su cui viene eseguito e tutto ciò che riguarda il ciclo di vita dell'applicativo stesso. Sebbene ciò abbia portato innumerevoli vantaggi nell'esecuzione e nella gestione del costrutto Fly, si sono riscontrate alcune problematiche e limitazioni:

- gli ambienti serverless offerti dai diversi cloud provider limitano il tempo di esecuzione degli applicativi;
- le risorse disponibili per l'esecuzione del codice sono predefinite;
- l'ambiente in cui viene eseguito l'applicativo Fly è gestito dal Cloud Provider stesso. Un livello di astrazione così alto comporta l'impossibilità di gestire possibili errori e avere il pieno controllo dell'ambiente di esecuzione.

Questo lavoro di tesi ha voluto risolvere i problemi che limitavano le risorse e le libertà computazionali di Fly, introducendo un ambiente di esecuzione in Kubernetes, ponendosi i seguenti obiettivi:

- **Tempi di esecuzione illimitati**

- **Allocazione dinamica delle risorse**
- **Controllo dell'ambiente di esecuzione**

5.1 Obiettivi raggiunti

L'obiettivo di introdurre la possibilità di eseguire un applicativo FLY su un cluster kubernetes è stato raggiunto. In particolare, i costrutti introdotti consentono di mantenere la semplicità di utilizzo e l'astrazione caratteristiche di FLY permettendo all'utente di eseguire le funzioni FLY sia su un cluster on-premise sia su Cloud.

5.2 Sviluppi futuri

Riguardo i possibili sviluppi futuri che possono interessare l'ambiente Kubernetes in Fly sono:

- **Gestione dello storage** - La gestione dello storage in Kubernetes è estremamente potente e potrebbe portare grandi vantaggi
- **Bilanciamento del carico** - Kubernetes ha un proprio Load Balancer che se sfruttate le potenzialità aiuterebbe a gestire ancora meglio le risorse

Bibliografia

- [1] *Amazon RDS - Amazon Web Services.*
URL: <https://aws.amazon.com/it/rds/>.
- [2] *Amazon Simple Queue Service - Amazon Web Services.*
URL: <https://aws.amazon.com/it/sqs/>.
- [3] *Amazon Simple Storage Service - Amazon Web Services.*
URL: <https://aws.amazon.com/it/s3/>.
- [4] *Amazon Web Services (AWS).* URL: <https://aws.amazon.com/>.
- [5] D. Ardagna et al. «MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds». In: *2012 4th International Workshop on Modeling in Software Engineering (MISE)* (2012).
- [6] *AWS Command Line Interface - Amazon Web Services.*
URL: <https://aws.amazon.com/it/cli/>.
- [7] *AWS Identity and Access Management - Amazon Web Services.*
URL: <https://aws.amazon.com/it/iam/>.
- [8] *AWS Lambda - Amazon Web Services.*
URL: <https://aws.amazon.com/it/lambda/>.
- [9] *AWS SDK - Amazon Web Services.*
URL: <https://aws.amazon.com/it/tools/>.
- [10] *axios - npm.* URL: <https://www.npmjs.com/package/axios>.
- [11] *Azure Active Directory — Microsoft Azure.* URL: <https://azure.microsoft.com/it-it/services/active-directory/>.
- [12] *Azure Command Line Interface — Microsoft Azure.*
URL: <https://docs.microsoft.com/it-it/cli/azure/?view=azure-cli-latest>.
- [13] *Azure Cosmos DB — Microsoft Azure.* URL: <https://azure.microsoft.com/it-it/services/cosmos-db/>.

- [14] *Azure Function* — Microsoft Azure. URL: <https://docs.microsoft.com/it-it/azure/azure-functions/>.
- [15] *Azure Functions* — Microsoft Azure. URL: <https://azure.microsoft.com/it-it/services/functions/>.
- [16] *Boto3 SDK AWS per Python - Amazon Web Services*. URL: <https://aws.amazon.com/it/sdk-for-python/>.
- [17] *CAMEL*. URL: <http://camel-dsl.org/>.
- [18] *Configuration functions in the AWS Lambda Console*. URL: <https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html>.
- [19] Gennaro Cordasco et al. «Toward a domain-specific language for scientific workflow-based applications on multicloud system». In: *Concurrency and Computation: Practice and Experience* (2020).
- [20] *Database di Azure* — Microsoft Azure. URL: <https://azure.microsoft.com/it-it/product-categories/databases/>.
- [21] *Database in AWS - Amazon Web Services*. URL: <https://aws.amazon.com/it/products/databases/>.
- [22] *Database MySQL* — Microsoft Azure. URL: <https://azure.microsoft.com/it-it/services/mysql/>.
- [23] *dataframe-js*. URL: <https://gmousse.gitbooks.io/dataframe-js/>.
- [24] *Docker*. URL: <https://www.docker.com/>.
- [25] *Docker Compose*. URL: <https://docs.docker.com/compose/>.
- [26] *Dynalite*. URL: <https://github.com/mhart/kinesalite>.
- [27] Ana Juan Ferrer, David García Pérez e Román Sosa González. «Multi-cloud Platform-as-a-service Model, Functionalities and Approaches». In: *Procedia Computer Science* 97 (2016). 2nd International Conference on Cloud Forward: From Distributed to Complete Computing, pp. 63–72.
- [28] Noel Yuhanna with Gene Leganza e Jeremy Vale. *The Forrester Wave: Database-As-A-Service, Q2 2019*. 2019.
- [29] G. Grieco. «Progettazione e implementazione del supporto a Microsoft Azure per il compilatore del linguaggio FLY». 2019.
- [30] Geir Horn e Pawel Skrzypek. «MELODIC: Utility Based Cross Cloud Deployment Optimisation». In: *2018 32nd International Conference (WAINA)* (2018).

- [31] K. Hwang, G.C. Fox e J.J. Dongarra. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann. Morgan Kaufmann, 2012.
- [32] *JavaScript*.
URL: <https://developer.mozilla.org/it/docs/Web/JavaScript>.
- [33] *JDBC — MySQL Connector*.
URL: <https://www.mysql.com/it/products/connector/>.
- [34] Eric Jonas et al. «Cloud Programming Simplified: A Berkeley View on Serverless Computing». In: *ArXiv* (feb. 2019).
- [35] *Kinesalite*. URL: <https://github.com/mhart/dynalite>.
- [36] K. Kritikos e P. Skrzypek. «Towards an Optimized, Cloud-Agnostic Deployment of Hybrid Applications». In: *Lecture Notes in Business Information Processing* (2019).
- [37] K. Kritikos et al.
«Towards the Modelling of Hybrid Cloud Applications». In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019).
- [38] Philipp Leitner et al. «A mixed-method empirical study of Function-as-a-Service software development in industrial practice». In: *Journal of Systems and Software* (2019).
- [39] *LocalStack*. URL: <https://www.localstack.cloud>.
- [40] D.C. Marinescu. *Cloud Computing: Theory and Practice*. Elsevier Science, 2017. ISBN: 9780128128114.
URL: <https://books.google.it/books?id=09smDwAAQBAJ>.
- [41] *Maven*. URL: <https://maven.apache.org/>.
- [42] *Maven - POM Reference*.
URL: <https://maven.apache.org/pom.html>.
- [43] *Microsoft Azure*. URL: <https://azure.microsoft.com/>.
- [44] *Moto: Mock AWS Services*.
URL: <http://docs.getmoto.org/en/latest/>.
- [45] *MySQL*. URL: <https://www.mysql.com/it/>.
- [46] *mysql - npm*. URL: <https://www.npmjs.com/package/mysql>.
- [47] *npmjs*. URL: <https://www.npmjs.com/>.
- [48] *OpenStack*. URL: <https://www.openstack.org>.

- [49] Dana Petcu. «Multi-Cloud: expectations and current approaches». In: *MultiCloud 2013 - Proceedings of the International Workshop on Multi-Cloud Applications and Federated Clouds* (apr. 2013), pp. 1–6.
- [50] B. Peterson, G. Baumgartner e Q. Wang. «A Hybrid Cloud Framework for Scientific Computing». In: *2015 IEEE 8th International Conference on Cloud Computing* (2015), pp. 373–380.
- [51] *PyMySQL*. URL: <https://pymysql.readthedocs.io/en/latest/>.
- [52] *Python*. URL: <https://www.python.org/>.
- [53] *qs - npm*. URL: <https://www.npmjs.com/package/qs>.
- [54] *Regular-Expression*. URL: <https://www.regular-expressions.info/>.
- [55] *SDK AWS per Java - Amazon Web Services*. URL: <https://aws.amazon.com/it/sdk-for-java/>.
- [56] *SDK AWS per JavaScript - Amazon Web Services*. URL: <https://aws.amazon.com/it/sdk-for-node-js/>.
- [57] *Serverless Framework*. URL: <https://www.serverless.com>.
- [58] Maddie Stigler. *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Apress, gen. 2018.
- [59] Ruslan Synytsky. *How to overcome the challenges of Gaining multi-cloud interoperability*. URL: <https://www.forbes.com/sites/forbestechcouncil/2018/10/25/how-to-overcome-the-challenges-of-gaining-multi-cloud-interoperability>.
- [60] *Tablesaw — Java dataframe and visualization library*. URL: <https://jtablesaw.github.io/tablesaw/>.
- [61] *The Python Package Installer*. URL: <https://pip.pypa.io/en/stable/>.
- [62] *TOSCA*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.
- [63] *Use bridge network - Docker*. URL: <https://docs.docker.com/network/bridge/>.
- [64] *util - npm*. URL: <https://www.npmjs.com/package/util>.

- [65] Adbys Vasconcelos et al. «DistributedFaaS: Execution of Containerized Serverless Applications in Multi-Cloud Infrastructures». In: (2019).
- [66] *Visual Studio - Ambiente di sviluppo integrato*.
URL: <https://visualstudio.microsoft.com/>.
- [67] M. Wurster et al. «Modeling and Automated Deployment of Serverless Applications Using TOSCA». In: *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)* (2018).
- [68] *Xtext*. URL: <https://www.eclipse.org/Xtext/>.