

# UNIVERSITÀ DEGLI STUDI DI SALERNO

---

## DIPARTIMENTO DI INFORMATICA



CORSO DI LAUREA MAGISTRALE IN INFORMATICA

*Flaky test: studio sistematico della letteratura e  
individuazione delle root cause per DeFlaker*

Relatori:  
**Ch.ma Prof.ssa  
Filomena FERRUCCI**

**Ch.mo Dott.  
Pasquale SALZA**

**Ch.mo Dott.  
Valerio TERRAGNI**

Candidata:  
**Valeria Pontillo  
Matr.: 0522500463**

ANNO ACCADEMICO 2018/2019

*A me, per non essermi mai arresa  
Alla mia famiglia, che ha sempre creduto in me*

# SOMMARIO

INTRODUZIONE .....	1
CAPITOLO 1 .....	4
1.1 Il testing .....	4
1.1.1 Ispezione delle componenti.....	5
1.1.2 Testing di usabilità .....	5
1.1.3 Testing di unità.....	6
1.1.4 Testing di integrazione .....	7
1.1.5 Testing di sistema.....	7
1.2 Gestione delle configurazioni .....	8
1.2.1 Version Control.....	8
1.2.2 System building.....	9
1.2.3 Change management .....	10
1.2.4 Release management .....	10
1.3 Continuous Integration.....	11
1.4 Regression Testing .....	12
1.4.1 Problemi del testing di regressione automatizzato: i flaky test... ..	13
CAPITOLO 2 .....	19
2.1 Revisione sistematica della letteratura .....	19
2.2 Attività della revisione sistematica .....	20
2.2.1 Pianificazione della revisione .....	20
2.2.2 Svolgimento della revisione.....	20
2.2.3 Report della revisione.....	20
2.3 Revisione sistematica della letteratura per i flaky test .....	21
2.3.1 Domande di ricerca .....	21

2.3.2 Strategia di ricerca.....	23
2.3.3 Selezione degli studi.....	25
2.3.4 Criteri di valutazione della qualità .....	26
2.3.5 Estrazione dei dati .....	29
2.3.6 Sintesi dei dati.....	31
2.3.7 Risultati e discussioni .....	32
2.3.8 Minacce alla validità.....	40
2.3.9 Conclusioni .....	40
CAPITOLO 3 .....	43
3.1 Un nuovo “inizio” .....	43
3.2 Ricerca di dati su Github .....	44
3.2.1 Primo approccio per la ricerca di dati.....	44
3.2.2 Secondo approccio per la ricerca di dati.....	44
3.3 Analisi dei dati già esistenti.....	46
3.3.1 Primo approccio: Gradle.....	48
3.3.2 Secondo approccio: Maven.....	49
3.3.4 Risultati ottenuti .....	67
3.3.5 Limiti della ricerca .....	73
CONCLUSIONI .....	74
INDICE DELLE FIGURE .....	76
APPENDICE A .....	77
BIBLIOGRAFIA .....	81
SITOGRAFIA .....	82
RINGRAZIAMENTI.....	83



# INTRODUZIONE

In un mondo in cui tutto cambia sempre più rapidamente, i prodotti software sono richiesti sempre più con maggiore rapidità e con grandi innovazioni. Gli sviluppatori delle grandi aziende come Huawei, Google, Facebook, Microsoft, ecc. si trovano ogni giorno a sviluppare e testare migliaia e migliaia di righe di codice. Ed è proprio il testing ad essere (ancora oggi) la parte più critica di ogni progetto software.

Con il diffondersi delle metodologie *Agile* ha assunto maggior importanza il testing di regressione; questo, nella sua forma automatizzata, risulta essere utile e poco costoso in quanto permette l'esecuzione di tutti i casi di test all'interno di un prodotto software ogni volta che si inserisce un cambiamento nel codice che si sta implementando. Quindi, con il testing di regressione è possibile controllare se il sistema è ancora funzionante dopo ogni modifica effettuata ed eventualmente si possono rapidamente individuare nuovi *bug*.

Ma cosa succede quando il testing assume un comportamento **non deterministico**? Ci si trova davanti a una nuova “tipologia” di test, il *flaky test*, ossia un caso di test che può passare o fallire senza che sia avvenuto alcun cambiamento all'interno del codice.

I *flaky test* sono una scoperta recente, essi infatti hanno iniziato a fare la loro comparsa negli ultimi quindici anni, ossia da quando sia le componenti hardware che i software hanno iniziato a differenziarsi sempre di più tra loro. Nonostante i *flaky test* siano una “piccola” percentuale rispetto al totale dei casi di test, essi risultano essere un grosso problema per gli sviluppatori delle grandi aziende, problema che ancora oggi non ha trovato una soluzione del tutto valida. La maggiore criticità riscontrata è nel capire la causa scatenante di un *flaky test*. La classificazione delle cause di un *flaky* risulta essere un punto chiave delle ricerche, perché individuare tutte le *root cause* può aiutare a capire le strategie da adottare per eliminare i *flaky test* una volta individuati o addirittura evitare il manifestarsi.

Negli ultimi cinque anni sono stati sviluppati diversi framework che possono individuare o addirittura eliminare un *flaky test*, ma questi framework presentano tutti lo stesso limite: funzionano per un unico linguaggio di programmazione,

spesso **Java**. Appare evidente che ancora oggi devono essere trovate delle soluzioni che possono essere adattate ed estese a più linguaggi di programmazione.

Ed è in questo contesto che si colloca questo lavoro di tesi, ossia in una delle proposte di progetto presentate e premiate nel 2019 durante il *Facebook Testing And Verification Symposium*; tale proposta prevede lo sviluppo di un nuovo tool, che con l’ausilio di cluster possa individuare la *root cause* di un *flaky test* indipendentemente dal linguaggio di programmazione in cui è stato implementato il caso di test.[8] Il primo step necessario allo sviluppo di questa idea è stata la creazione di un nuovo dataset contenente dei *flaky test* con le relative *root cause*. Quindi, adottando una nuova strategia, si è deciso di ampliare le informazioni di un dataset già esistente, ossia quello di **DeFlaker**, infatti in questo dataset erano presenti dei *flaky test* ma senza una classificazione delle *root cause*, concentrandosi soprattutto sulle cause diverse dalla dipendenza dall’ordine di esecuzione dei casi di test.

La strategia adottata prevede l’isolamento del singolo caso di test da tutta la test suite e l’esecuzione ripetuta per vedere eventuali cambiamenti nel risultato del caso di test. Dei *flaky test* individuati, è stata poi fatta una classificazione delle *root cause* che li ha generati, andando ad analizzare l’eccezione lanciata dal caso di test e facendo un’eventuale analisi statica del codice. Su un campione di centoquarant’uno casi di test, si sono riscontrati circa venti *flaky test* e le relative *root cause*, raggruppate in tre categorie: **Network, Asincronia, Multithreading**. Successivamente è stata effettuata una nuova esecuzione di ventiquattro casi di test per verificare qualche cambiamento nel loro comportamento e si sono riscontrati gli stessi risultati ottenuti precedentemente (anche per numeri di successo e fallimenti nel caso di un test *flaky*). Questo quindi, ha fatto pensare a una sorta di determinismo nel comportamento di un caso di test, anche in presenza di flakiness (problematica da dover approfondire in futuro).

Questo lavoro di tesi è strutturato nel seguente modo.

Nel primo capitolo viene presentata una panoramica generale sul testing nell’ambito dell’ingegneria del software e sulla gestione delle configurazioni, in cui si accenna ai sistemi di *versioning* e di *build* per progetti in costante cambiamento.

Segue poi una panoramica sulla *continuous integration* e sul testing di regressione, per arrivare a parlare dei *flaky test* e delle possibili cause per cui si manifesta una flakiness in un caso di test.

Nel secondo capitolo viene presentato una revisione sistematica della letteratura, che ha permesso di raccogliere quante più informazioni possibili sugli studi fatti nell'ambito della flakiness, sia dal punto di vista dei ricercatori che dal punto di vista degli sviluppatori.

Nel terzo capitolo viene presentata la costruzione parziale di un nuovo dataset in cui si vanno a classificare le diverse *root cause* di *flaky test* precedentemente individuati. Viene inoltre fatta un'analisi su un possibile **determinismo** sui *flaky test*.

Le conclusioni riassumono i risultati ottenuti e delineano i possibili sviluppi futuri per il progetto descritto precedentemente.

# CAPITOLO 1

## 1.1 Il testing

Lo sviluppo di un prodotto software è fatto da molteplici fasi: si ha una prima fase di definizione del prodotto, in cui il project manager collabora con il cliente per poter individuare e analizzare tutti i requisiti che il prodotto software dovrà avere. Si passa poi alla fase di progettazione, in cui il team di sviluppo elabora tutte le informazioni date dal cliente e inizia a pensare ai requisiti “tecnici” che il prodotto dovrà avere (architettura del sistema, linguaggi di programmazione da adottare, ecc.). Dopo aver individuato tutte le caratteristiche tecniche, il team di sviluppo si occuperà dell’implementazione del prodotto software. Durante questa fase, il prodotto software dovrà essere costruito rispettando i requisiti del cliente e le scelte progettuali fatte precedentemente. Una volta terminata la fase di implementazione si passa alla fase più “critica” di tutto lo sviluppo software: la fase di testing.

[2] Il testing è il processo attraverso cui si analizza parte del sistema appena sviluppato e si cercano differenze tra il comportamento atteso e il comportamento osservato. La fase di testing risulta essere una delle più costose e lunghe di tutto lo sviluppo del prodotto software, inoltre, essendo un problema indecidibile, non è possibile testare tutte le componenti di un sistema. Per cercare di individuare più *bug* possibili all’interno del prodotto ci sono diverse attività di testing che possono essere eseguite:

- **Ispezione delle componenti;**
- **Testing di usabilità;**
- **Testing di unità;**
- **Testing di integrazione;**
- **Testing di sistema.**

### **1.1.1 Ispezione delle componenti**

Durante la revisione del codice sorgente vengono trovati dei *fault*<sup>1</sup>.

L’ispezione del codice viene condotta da un team di sviluppatori (incluso l’autore del codice che si sta esaminando), un moderatore che facilita il processo di ispezione del codice e almeno un revisore che individua i *fault* nel codice sorgente.

Il metodo di ispezione proposto da Fagan consiste di cinque passi:

- 1. Overview:** gli sviluppatori presentano brevemente lo scopo della componente e l’obiettivo dell’ispezione;
- 2. Preparazione:** il revisore acquisisce familiarità con l’implementazione della componente;
- 3. Meeting per l’ispezione:** viene effettuata la lettura del codice sorgente e il team di revisione evidenzia eventuali problemi trovati all’interno della componente, mentre un moderatore tiene traccia del meeting;
- 4. Rework:** l’autore revisiona la componente;
- 5. Follow-up:** il moderatore controlla la qualità della componente revisionata e decide se la componente ha bisogno di essere nuovamente ispezionata.

La fase di preparazione e la fase di meeting sono le più critiche dell’ispezione, infatti nella fase di preparazione il revisore prenderà confidenza con il codice da analizzare (senza concentrarsi su eventuali *bug*), mentre durante il meeting ci sarà un lettore del codice sorgente e il team di revisori evidenzierà eventuali problemi presenti nel codice. L’efficacia dell’ispezione delle componenti con la metodologia di Fagan dipende dalla preparazione dei revisori, ma viene considerata un’attività che consuma molto tempo, per questo motivo sta cadendo in disuso.

### **1.1.2 Testing di usabilità**

Il testing di usabilità mette alla prova la comprensione dell’utente sul sistema, quindi si focalizza sulle possibili differenze tra il sistema e le aspettative che l’utente ha sul sistema quando lo va a provare. Il testing di usabilità viene fatto tramite un approccio empirico: i partecipanti, che sono un campione rappresentativo

---

<sup>1</sup> Il fault, chiamato anche bug o difetto è la causa di un comportamento errato del sistema.

della popolazione, tramite la manipolazione dell’interfaccia utente identificano eventuali problemi del sistema e testano anche la gradevolezza dell’interfaccia (colori dell’interfaccia, sequenze di interazioni, ecc.). Ci sono tre tipi di testing di usabilità:

- **Scenario test:** durante questo test viene presentato uno scenario del sistema a uno o più partecipanti. Gli sviluppatori identificano quanto velocemente gli utenti riescono a capire lo scenario e la loro reazione alla descrizione del nuovo sistema. Lo scenario test viene realizzato con i mockup<sup>2</sup> o con un prototipo semplificato. I vantaggi sono il costo (molto basso) e la possibilità di ripetere il test, mentre lo svantaggio è che l’utente non può interagire direttamente con il sistema;
- **Test del prototipo:** durante questo test, viene presentato agli utenti una parte del software che implementa aspetti chiave del sistema. I vantaggi di questo test è il fornire una visione realistica del sistema e la possibilità di collezionare dati dettagliati. Lo svantaggio dei prototipi è dato dal tempo, infatti i prototipi richiedono un effort maggiore rispetto agli scenari;
- **Test del prodotto:** questo test è simile al precedente, però al posto di usare un prototipo verrà testata una versione della funzionalità del sistema. Quindi, questo test può essere fatto solo quando è stata sviluppata buona parte del sistema. Inoltre, è richiesto che il sistema possa essere facilmente modificabile in base ai risultati ottenuti durante il test di usabilità.

### 1.1.3 Testing di unità

Il testing di unità si focalizza su singoli blocchi del sistema software che si sta testando, permettendo un’individuazione precoce dei bug del sistema. Il testing di unità viene fatto per tre motivi:

1. Riduce la complessità di tutte le attività di testing successive;
2. Rende più semplice correggere i *fault*, perché le componenti coinvolte sono minori rispetto all’intero sistema;

---

<sup>2</sup> Un mockup è un’illustrazione del sistema senza le funzionalità complete del sistema stesso. Può rappresentare tutto il sistema o solo una parte di esso.

3. Ogni componente può essere testata indipendentemente dalle altre componenti del sistema.

#### 1.1.4 Testing di integrazione

Una volta che il testing di unità ha individuato e rimosso eventuali *bug*, le componenti sono pronte per essere integrate in sottosistemi più grandi. Scopo del testing di integrazione è individuare *fault* che non sono stati trovati durante il testing di unità, infatti il testing di integrazione ha diversi step:

- Vengono testate due o più componenti insieme;
- Se non vengono trovati dei *fault*, allora si aggiungono delle altre componenti al sottosistema già testato, altrimenti si dovranno prima effettuare delle modifiche al sistema per permettere l'integrazione corretta alle due componenti.

Ci sono due strategie di testing di integrazione:

- **Testing di integrazione orizzontale:** in questo caso le componenti sono integrate secondo i *layer*<sup>3</sup> in cui si trovano;
- **Testing di integrazione verticale:** in questo caso le componenti sono integrate in base alle funzioni.

#### 1.1.5 Testing di sistema

Una volta che sono stati svolti i testing di unità e di integrazione, viene svolto il testing di sistema, il cui scopo è assicurare il completo funzionamento del sistema in base ai *requisiti funzionali*<sup>4</sup> e non funzionali<sup>5</sup> descritti nella documentazione del sistema. Anche il testing di sistema prevede molteplici funzionalità:

- **Testing funzionale:** cerca le differenze tra i requisiti funzionali e il sistema implementato;

---

<sup>3</sup> Un layer è un sottosistema nella scomposizione gerarchica di un sistema software. Un layer dipende solo dal layer che si trova al livello inferiore e non ha conoscenza dei layer che si trovano a livello superiore.

<sup>4</sup> Un requisito funzionale rappresenta una funzione del sistema che si dovrà implementare.

<sup>5</sup> Un requisito non funzionale indica un particolare aspetto del sistema visibile all'utente ma non collegato alle funzionalità del sistema.

- **Testing delle performance:** cerca le differenze tra i *design goal*<sup>6</sup> definiti nella documentazione del progetto e il sistema implementato;
- **Pilot testing:** il sistema viene installato e usato da un numero ristretto di utenti, i quali dovranno fornire dei feedback agli sviluppatori;
- **Testing di accettazione:** il sistema verrà provato dal cliente, il quale riporterà al project manager dei feedback (positivi o negativi) sul sistema;
- **Testing di installazione:** una volta che il sistema è stato accettato, esso verrà installato all'interno dell'ambiente target e verrà formalmente rilasciato sul mercato.

## 1.2 Gestione delle configurazioni

[7] I sistemi software subiscono un cambiamento costante, sia durante lo sviluppo del sistema che durante l'uso. Questo cambiamento costante avviene perché vengono continuamente scoperti bug che devono essere rimossi, quindi ogni volta sarà necessario implementare una nuova versione del sistema con questi cambiamenti. Questa fase dello sviluppo del software è denominata *configuration management* ed è indispensabile per evitare di non tenere traccia di eventuali cambiamenti delle componenti. La gestione delle configurazioni è fatta da quattro attività:

1. **Version Control;**
2. **System Building;**
3. **Change Management;**
4. **Release Management;**

### 1.2.1 Version Control

Il version control è il processo tramite il quale si tiene traccia delle diverse versioni delle componenti software e di quali sono usate all'interno del sistema. Al giorno d'oggi è importantissimo utilizzare un servizio di *versioning*, perché garantiscono che eventuali cambiamenti fatti dai diversi sviluppatori al lavoro sullo

---

<sup>6</sup> Un design goal indica una qualità che il sistema dovrebbe ottimizzare. Alcuni esempi di design goal sono l'usabilità, l'affidabilità e la sicurezza.

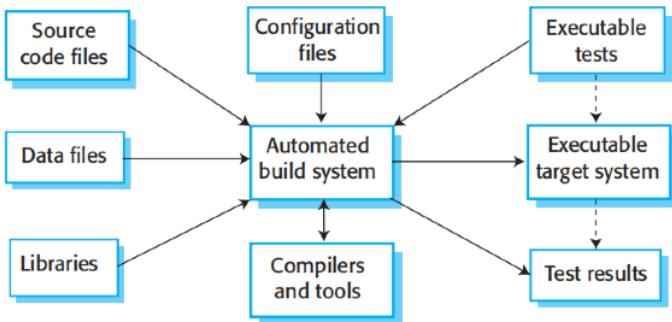
stesso progetto non interferiscono tra loro creando conflitti. Ci sono due tipi di sistemi di version control:

1. **Sistema centralizzato:** c'è un solo repository master in cui sono mantenute tutte le versioni che si stanno sviluppando di una componente del sistema software. Un esempio di sistema centralizzato è *Subversion*.
2. **Sistemi distribuiti:** esistono contemporaneamente più versioni dei repository delle componenti software che si stanno sviluppando. Un esempio di sistema distribuito è *Git*.

Entrambi i sistemi di Version Control forniscono funzioni simili ma sono implementate in modo diverso. L'uso dei sistemi di *versioning* è indispensabile nel caso di progetti open source, in cui molte persone possono lavorare simultaneamente sullo stesso sistema, nello stesso momento e senza alcuna coordinazione.

### 1.2.2 System building

Il system building è il processo di creazione di un sistema completo ed eseguibile tramite la compilazione delle componenti del sistema, delle librerie esterne, dei file di configurazione e altre informazioni utili. Per creare un sistema potrebbe essere utile unire un gran numero di informazioni sul software e sul suo ambiente operativo. Quindi, specialmente su progetti molto grandi potrebbe essere molto utile usare dei tool per la build automatizzata per creare il sistema di build. La build automatizzata risulta essere molto utile perché con essa non si avrà bisogno di informazioni aggiuntive su librerie esterne, file di configurazioni, ecc., ma basterà un comando o un click per avviare l'installazione del software sul proprio computer. I tool per la build di sistema automatizzati contengono diverse feature illustrate in Figura 1:



*Figura 1: Feature della build automatizzata*

### 1.2.3 Change management

Quando si lavora a un sistema software si deve sempre tenere in considerazione la possibilità di dover effettuare delle modifiche. Queste modifiche possono dipendere da bug del sistema che devono essere risolti una volta individuati oppure da cambiamenti nei requisiti richiesti dal cliente. Nell'ambito dell'ingegneria del software, esiste un processo che si occupa di tutto questo: il change management.

Questo processo effettua un'analisi dei costi e dei benefici di ogni modifica richiesta, e una volta che la richiesta verrà approvata, si dovrà tenere traccia di tutte le componenti del sistema che verranno coinvolte nel cambiamento.

### 1.2.4 Release management

La release di un sistema consiste in una versione del software che viene distribuita ai consumatori. Ci sono due tipi di release:

- **Major release:** il sistema software prevede delle nuove funzionalità;
- **Minor release:** in questo caso il sistema software verrà rilasciato senza i *bug* segnalati dai clienti.

Con rilascio del software non si intende solo la diffusione del codice eseguibile, ma anche di altre componenti come i file di configurazione, i programmi di installazione, i manuali, ecc. Per questo motivo l'attività di rilascio risulta essere un'attività costosa, che deve essere documentata in maniera dettagliata in modo da poterla ricreare in futuro.

### 1.3 Continuous Integration

Al giorno d’oggi, con l’avvento delle metodologie di sviluppo *Agile*<sup>7</sup>, alcune delle attività descritte precedentemente vengono effettuate in maniera diversa e automatica. Ad esempio, per progetti software molto grandi e con rilasci frequenti ai clienti, è consigliabile effettuare frequentemente delle build system con test automatizzati per poter individuare problemi all’interno del software. Fare delle build frequenti rientra in un processo più ampio, quello della *continuous integration*.

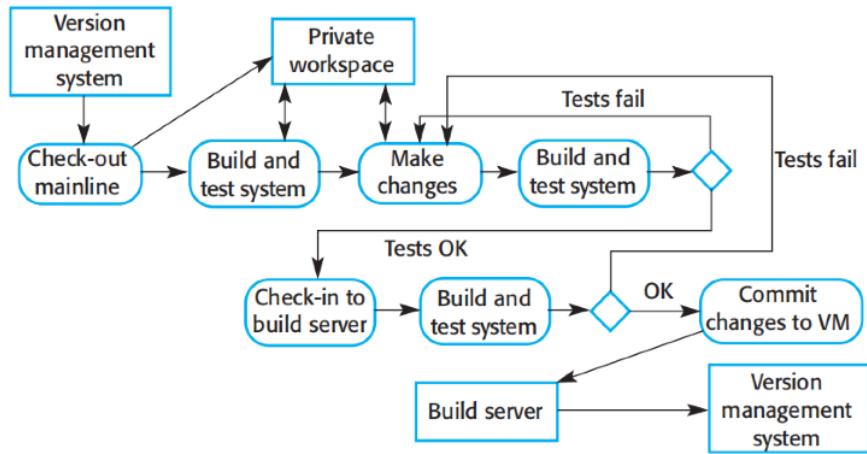
Il processo di *continuous integration*, proposto da **Grady Booch** negli anni ‘90, permette una build del sistema ogni volta che viene introdotta una modifica all’interno del codice sorgente. La *continuous integration* è formata da diversi passi:

- Estrazione del codice sorgente da un sistema di Version Control a uno spazio di lavoro privato dello sviluppatore;
- Build del sistema ed esecuzione automatica dei vari casi di test. Lo sviluppatore dovrà assicurarsi che tutti i test abbiano successo, altrimenti la build risulterà “rotta” e dovrà essere aggiustata dall’ultimo sviluppatore che ha modificato il sistema;
- In caso di successo della build lo sviluppatore potrà effettuare dei cambi all’interno del sistema;
- Una volta terminato il proprio lavoro, lo sviluppatore dovrà effettuare nuovamente l’esecuzione dei test per assicurarsi che tutti i test abbiano successo e che la build sia ancora funzionante. In caso affermativo, potrà caricare il proprio lavoro sul sistema di *versioning* con cui sta lavorando; questa nuova versione diventerà la nuova baseline del sistema software.

Il processo di CI è illustrato nella Figura 2.

---

<sup>7</sup> Lo sviluppo Agile, si distingue dalle altre metodologie di sviluppo software per il suo approccio meno strutturato, focalizzato sul consegnare in tempi brevi un software funzionante e di qualità.



**Figura 2: Processo di Continuos Integration**

Il vantaggio principale della *continuous integration* è la scoperta più rapida degli errori che possono derivare dall’interazione degli sviluppatori software per lo stesso progetto. Ci sono diversi tool che gestiscono la *continuous integration* in maniera automatica come **Jenkins**, **Travis** e **Cruise Control**.

Anche se la *continuous integration* sembra essere sempre utile, non è sempre possibile applicarla. Infatti, essa viene sconsigliata quando:

- Il sistema che si sta sviluppando è molto grande. In questo caso la build automatica con l’esecuzione di tutti i casi di test richiederebbe parecchio tempo;
- La piattaforma di sviluppo è diversa dalla piattaforma target. In questo caso infatti non sarebbe possibile lanciare i casi di test sul workspace privato dello sviluppatore.

## 1.4 Regression Testing

Nell’ingegneria del software, il testing di regressione serve per verificare che l’introduzione di una nuova funzionalità o la modifica di una qualche componente già esistente nel sistema, non comprometta in alcun modo la qualità del sistema stesso. Quindi, l’obiettivo del testing di regressione è ripetere il collaudo del sistema a ogni modifica introdotta, in modo da verificare che la qualità non sia *regredita* (da qui il nome di *regression*).

Il testing di regressione manuale risulta essere molto complesso e ha dei costi di sviluppo elevati. Viceversa, un testing di regressione automatizzato ha un costo più contenuto perché si tratterebbe solo di lanciare le procedure di collaudo già esistenti. Ma perché il testing di regressione (automatizzato) è così importante nell'ambito dello sviluppo software *Agile*?

Con il testing di regressione gli sviluppatori possono ricevere rapidamente molti feedback sul loro lavoro, infatti il testing di regressione può scoprire prima nuovi *bug*, ammortizzando i costi e l'effort delle manutenzioni future. Quindi, se si ha un prodotto software sottoposto a frequenti modifiche, il testing di regressione preserverà la qualità del prodotto finale<sup>[1]</sup>.

#### **1.4.1 Problemi del testing di regressione automatizzato: i flaky test**

[5] [6] Al giorno d'oggi il testing di regressione automatizzato è ampiamente usato dagli sviluppatori di tutto il mondo. Quando un programmatore effettua dei cambi all'interno di un prodotto software, successivamente dovrà effettuare un run di tutti i casi di test per vedere se ha introdotto dei bug. Se il test passa, lo sviluppatore continua con altre modifiche; se il test fallisce lo sviluppatore dovrà concentrarsi sulle ultime modifiche fatte per trovare il *bug* che ha generato la *failure*.

A volte però lo sviluppatore può trovarsi a fare i conti con un test che presenta un comportamento **non deterministico**. Questi test, denominati *flaky test*, sono dei test che possono passare o fallire in maniera apparentemente casuale senza che sia stato effettuato alcun cambiamento all'interno del codice sorgente. I *flaky test* rappresentano un grande problema nell'ambito dello sviluppo software, perché lo sviluppatore che si trova davanti un fallimento del caso di test, non sa se quel fallimento deriva da un problema all'interno del codice (un *bug*) o da un fallimento all'interno del test stesso. Convinto di trovarsi davanti a un *flaky*, potrebbe decidere di non fare attività di *debugging*<sup>8</sup> e potrebbe quindi non trovare i bug presenti nel codice. Viceversa, potrebbe spendere molte ore nell'attività di *debugging* senza

---

<sup>8</sup> Il debugging è l'attività fatta dallo sviluppatore per individuare e correggere uno o più bug all'interno del codice.

trovare alcun errore e senza andare avanti con lo sviluppo del software. Possiamo quindi immaginare che i *flaky test* generino un forte senso di frustrazione nello sviluppatore. Ma cosa fanno gli sviluppatori in queste situazioni?

Alcuni semplicemente ignorano il fallimento, dando per scontato che il problema sia all'interno del test e non all'interno del codice appena introdotto nel sistema. Altri effettuano nuovamente il run del test fino a quando non hanno un successo nella sua esecuzione (marcando quindi il test come *flaky*). Il rerun però non risulta essere una soluzione ottimale, perché porta via molto tempo (e quindi denaro) e questo tempo potrebbe risultare perso nel caso in cui non si riesca a dimostrare il comportamento *flaky* del test.

Un altro problema dei *flaky test*, oltre al comportamento apparentemente casuale, è capire che cosa ha scatenato un comportamento **non deterministico** del test e come risolvere. Alcune cause dei *flaky* possono essere facilmente individuate e corrette, altre invece richiedono parecchio tempo e dei costi elevati. Di seguito sono riportate le tre cause principali dei *flaky test* con le loro strategie di minimizzazione <sup>(2)</sup>:

- **Problemi di asincronia:** rappresentano il 45% dei *flaky test* e si presentano perché spesso gli sviluppatori hanno bisogno di aspettare che qualche altro statement sia completo. In questi casi si genera un errore perché il comando *sleep* non è sempre preciso (portando quindi a un fallimento del test). Questo errore può essere risolto sostituendo la funzione *sleep* con una funzione *waitFor*. Ovviamente, questa soluzione non elimina del tutto il comportamento **non deterministico** del test ma lo riduce di molto. In Figura 3 è riportato un esempio di *flaky test* derivante dall'uso della funzione *sleep*.

```

1  [TestMethod]
2  public void DelayedTaskStaticBasicTest() {
3      int delay = 1000; int i = 0;
4      Scheduler.ScheduleTask(
5          DateTime.UtcNow.AddMilliseconds(delay),
6          new LoggedTask(
7              "TestDelayedTaskFrameworkTask", () => { i = 1; },
8              new Dictionary<string, string> { { "test", "value" } }));
9      Thread.Sleep(500);
10     Assert.IsTrue(i == 0);
11     Thread.Sleep(delay);
12     Assert.IsTrue(i == 1);
13 }

```

*Figura 3: Flaky test causato dall'asincronia*

- **Problemi di concorrenza:** rappresentano il 20% dei *flaky test* e si presentano perché lo sviluppatore fa delle considerazioni errate sull'ordine di esecuzioni delle operazioni sui diversi *threads*<sup>9</sup>. Questi flaky possono essere eliminati aggiungendo dei blocchi di sincronizzazione oppure permettendo al caso di test di accettare una gamma più ampia di comportamenti. In Figura 4 è riportato un *flaky test* causato dalla concorrenza.

---

<sup>9</sup> Un thread è una suddivisione di un processo in due o più istanze o sotto processi che vengono eseguiti in maniera concorrente.

```

1  [TestMethod]
2  public async Task TestDirtyResource() {
3      ...
4      using (var emptyPoolManager = CreatePoolManager(...)) {
5          Resource pm = ResourceUtils.CreatePhysicalMachine(...);
6          await emptyPoolManager.AddOrUpdateResourceAsync(pm, HeartbeatStatus.
7              InUse);
8          ...
9          for (int i = 0; i < 5; i++) {
10              var sessionId = Guid.NewGuid().ToString();
11              var request = new ResourceAllocateRequest(pm);
12              await emptyPoolManager.PreAllocateResourcesAsync(request.Yield(), pm.
13                  Specification, TenantId, sessionId, sessionPriority);
14              var response = (await QueryAllocate(emptyPoolManager, request.Yield()
15                  , TenantId, sessionId)).FirstOrDefault();
16              Assert.IsNotNull(response);
17              Assert.AreEqual(request.Id, response.RequestId);
18              Assert.IsNotNull(response.ResourceId);
19              Assert.AreEqual(pm.ResourceId, response.ResourceId);
20              await emptyPoolManager.ReleaseResourcesAsync(response.ResourceId);
21              await emptyPoolManager.HeartbeatResourceAsync(pm, HeartbeatStatus.
22                  Ready);
23              await emptyPoolManager.ProcessResourcesHeartbeats(CancellationToken.
24                  None);
25          }
26      }
27  }

```

*Figura 4: Flaky test causato dalla concorrenza*

- **Dipendenza dall'ordine di esecuzione:** rappresentano il 12% dei *flaky test* e si presentano quando lo sviluppatore non ha scritto in maniera ottimale il caso di test. Infatti, un buon caso di test dovrebbe essere isolato dagli altri casi di test, ma spesso questi condividono alcuni stati come la memoria, i database e i file. Quindi, in questi casi il successo o il fallimento di un test dipende dall'ordine in cui vengono eseguiti i casi di test. In Figura 5 è riportato un esempio di *flaky test* causato dall'ordine in cui i casi di test vengono eseguiti.

```

1  public void assertIsShutdownAlready() {
2      shutdownListenerManager.new
3          InstanceShutdownStatusJobListener().dataChanged("/
4              test_job/instances/127.0.0.1@-@0", Type.
5                  NODE_REMOVED, "");
6      verify(schedulerFacade, times(0)).shutdownInstance();
7  }

```

*Figura 5: Flaky test causato dalla dipendenza dall'ordine di esecuzione*

Dal momento che gli studi sulla flakiness hanno compiuto dei passi in avanti, sono state riscontrate delle altre cause che portano a comportamenti **non deterministici** del caso di test, ossia:

- **Perdita di risorse:** una perdita di risorse si verifica ogni volta che l'applicazione non gestisce correttamente una o più risorse, come le allocazioni di memoria o le connessioni ai database. Questa gestione scorretta delle risorse porta a guasti intermittenti;
- **Network:** spesso il network risulta essere una risorsa difficile da controllare. I fallimenti legati al network sono raggruppati in due sottocategorie:
  - fallimenti causati da errore nella connessione;
  - fallimenti causati da una cattiva gestione dei socket<sup>10</sup>;
- **Tempo:** basare un caso di test sul tempo introdurrà sicuramente un comportamento *flaky*, in quanto il tempo è gestito in maniera diversa tra una piattaforma e un'altra;
- **Operazioni di I/O:** le operazioni che fanno uso di Input/Output, possono generare comportamenti **non deterministici** nel momento in cui non vengono seguite delle “buone norme” di programmazione (es. effettuare l’operazione di *close* su un file dopo averlo letto);
- **Uso dei numeri random:** in questo caso la *flakiness* si presenta perché non si tiene conto di tutti i possibili valori che il generatore casuale può creare;
- **Operazioni con i numeri in virgola mobile:** spesso la gestione di calcoli con numeri in virgola mobile può portare a casi **non deterministici**, specialmente nei calcolatori ad alte prestazioni;
- **Dipendenza dalla piattaforma:** in questa categoria rientrano tutti casi di test che hanno un comportamento differente in base alla macchina su cui vengono eseguiti.

I *flaky test* sono un problema piuttosto diffuso nelle grandi aziende come Google, Microsoft, Mozilla, ecc., e occupano molto del tempo degli sviluppatori di

---

<sup>10</sup> Un socket è un oggetto software che permette l’invio e la ricezione di dati, tra host remoti (tramite una rete) o tra processi locali.

queste multinazionali. Nel 2016 John Micco, senior manager della Google, ha dichiarato che l'1,5% dei loro test riportano un risultato *flaky* e che questa percentuale porta del lavoro extra e ripetitivo allo sviluppatore. Ovviamente sono state adottate delle strategie per cercare di mitigare i *flaky test*, ma non sono comunque riusciti ad eliminare questa percentuale dai loro progetti.<sup>[3]</sup> Tra le strategie adottate da Google le più rilevanti sono state:<sup>[3]</sup>

- Effettuare nuovamente un'esecuzione manuale del caso di test fallito;
- Un rerun automatico del caso di test fallito;
- Eseguire tre volte il caso di test prima di marcare il suo stato come fail. Questa soluzione riduce i falsi positivi ma porta lo sviluppatore a ignorare i flakiness presenti nel progetto;
- Una quarantena automatica del caso di test per cui si sospetta la flakiness; per quarantena si intende la rimozione del caso di test dal cammino critico<sup>11</sup> e la classificazione di esso come *flaky*. Questa quarantena previene il problema, ma porta anche a “nascondere” i bug reali del codice.

Anche da parte degli sviluppatori di alcuni framework, sono state adottate diverse strategie per cercare di mitigare gli effetti del *flaky test*:

- Android usa il marcitore **@FlakyTest**;
- Jenkins usa il marcitore **@RandomFail**;
- Spring usa il marcitore **@Repeat**;
- Maven Surefire ha la proprietà **rerunFailingTestsCount**.

Nonostante questo, i *flaky test* continuano a rappresentare un problema e una sfida per gli sviluppatori, infatti le grandi multinazionali continuano ad investire per cercare di mitigare e prevenire la presenza dei *flaky test*.

---

<sup>11</sup> Il critical path è un algoritmo usato per la pianificazione delle attività di progetto. Serve per calcolare la data di inizio e la data di fine che alcune attività del progetto devono rispettare per fare in modo che il progetto non subisca ritardi temporali.

# CAPITOLO 2

## 2.1 Revisione sistematica della letteratura

[4] Una revisione sistematica della letteratura è un tipo di studio che ha l'obiettivo di individuare, valutare e interpretare tutte le fonti disponibili per un determinato campo di studio, di ricerca o per un fenomeno di proprio interesse. Gli studi individuali che vengono trovati nella letteratura e che contribuiscono alla revisione sistematica, sono detti *studi primari*; invece, la revisione sistematica è una forma di *studio secondario*.

Ci sono diversi motivi che portano il ricercatore a intraprendere questo tipo di studio. Le più comuni sono:

- Riassumere gli aspetti già esistenti per un particolare trattamento o una tecnologia;
- Identificare alcune mancanze nella ricerca corrente, in modo da evidenziare alcune aree in cui fare ulteriori ricerche;
- Fornire il quadro della situazione generale, in modo da poter posizionare in maniera appropriata una nuova attività di ricerca.

I vantaggi che si possono trarre da una revisione sistematica della letteratura sono:

- Una valenza scientifica dello studio, perché avendo una metodologia ben definita non dovrebbe portare ad avere dei risultati di parte;
- Per gli studi quantitativi si possono combinare i dati in modo da poter effettuare alcune analisi tecniche. In questo modo aumenta la probabilità di individuare dei rapporti causa-effetto che singoli studi non possono individuare.

Lo svantaggio principale di questa metodologia è la fatica maggiore richiesta rispetto a una normale revisione della letteratura.

## **2.2 Attività della revisione sistematica**

Una revisione sistematica della letteratura è formata da diverse attività raggruppate in tre fasi: pianificazione della revisione, svolgimento della revisione e report della revisione.

### **2.2.1 Pianificazione della revisione**

Scopo della pianificazione della revisione sistematica, è capire se c'è una reale necessità di svolgere questo tipo di ricerca nella letteratura. Una volta individuati i motivi dietro la decisione di fare una revisione sistematica, l'attività principale da svolgere è la definizione delle *research question*. Inoltre, durante la fase di pianificazione, si dovrà anche definire il protocollo (ovvero le linee guida) che si andranno a seguire per tutte le attività della revisione sistematica.

### **2.2.2 Svolgimento della revisione**

Una volta che è stato approvato il protocollo definito precedentemente, la revisione sistematica della letteratura può partire. Ovviamente, questa è la fase più lunga di tutta la revisione, in quanto si dovranno svolgere molte attività, ossia:

- Identificare la strategia di ricerca;
- Selezione degli studi primari;
- Valutazione della qualità degli studi primari;
- Estrazione dei dati;
- Sintesi dei dati.

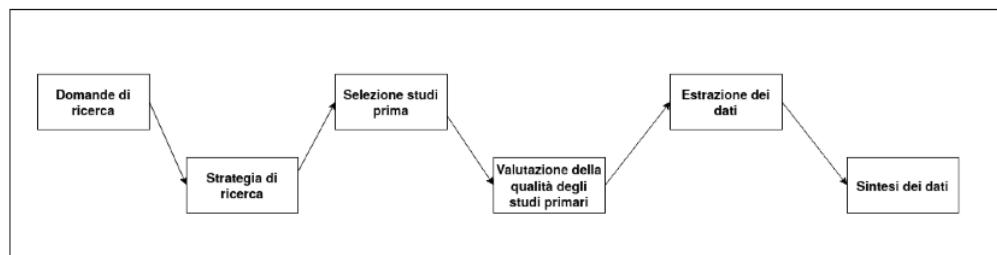
### **2.2.3 Report della revisione**

La fase finale della revisione sistematica della letteratura consiste nella scrittura e pubblicazione dei risultati ottenuti. Una volta che i risultati della revisione sistematica sono stati pubblicati, essi andranno valutati da esperti per assicurarsi che non siano stati influenzati dalle idee dei ricercatori che hanno svolto la revisione sistematica o che siano risultati di parte.

## 2.3 Revisione sistematica della letteratura per i flaky test

In questi anni, quando ancora si parlava di “test intermittenti”, i *flaky test* erano già oggetto di dibattito e di diversi studi scientifici che hanno cercato di indicare delle strategie per mitigarli oppure hanno fornito dei tool per individuarli. Ancora oggi i *flaky test* rappresentano un argomento poco conosciuto e oggetto di numerosi dibattiti, basti pensare che solo nel 2019 sono stati proposti diversi framework per il loro riconoscimento. Quindi, lo scopo di questa revisione sistematica è fare chiarezza su cosa è stato fatto fino a ora, per poi provare a capire cos’altro potrà essere fatto in futuro.

La metodologia usata per condurre questa revisione sistematica della letteratura segue le linee guida fornite da **Kitchenham e Charters**, quindi, dopo aver identificato le domande di ricerca, si è passati a definire una strategia di ricerca e a selezionare gli studi. Successivamente è stata fatta una valutazione della qualità degli studi primari individuati, seguita dall'estrazione dei dati e da una sintesi dei dati ottenuti. In Figura 6 è riportata uno schema riassuntivo del protocollo seguito per questa revisione sistematica.



*Figura 6: Protocollo della revisione sistematica della letteratura*

### 2.3.1 Domande di ricerca

Individuare le domande di ricerca è la parte più importante di qualsiasi revisione sistematica della letteratura per i seguenti motivi:

- Il processo di ricerca dovrà essere fatto in modo da individuare degli studi primari inerenti alle domande di ricerca scelte;
- Il processo di estrazione dei dati dovrà individuare delle informazioni che possano essere utili per dare risposta alle domande di ricerca;

- Il processo di analisi dei dati dovrà sintetizzare i dati in modo che essi rispondano alle domande di ricerca scelte.

Le domande di ricerca devono seguire una struttura ben precisa, che può cambiare in base all'ambito scientifico in cui la revisione sistematica si sta svolgendo. Nell'ambito dell'ingegneria del software, la struttura delle RQ deve seguire i seguenti criteri:

- **Popolazione:** nel campo dell'ingegneria del software la popolazione può essere uno specifico ruolo, come il tester o il manager, oppure una categoria in cui viene inquadrato un ruolo, come novizio o esperto, ecc.;
- **Intervento:** è una metodologia/tool/procedura/ecc. che affronta una specifica questione, come il testing o la stima dei costi software;
- **Comparazione:** indica la metodologia/tool/procedura/ecc. con il quale si va a comparare l'intervento;
- **Risultati:** devono riguardare fattori importanti per gli operatori del settore, come la riduzione dei costi di produzione, miglioramento dell'affidabilità di un sistema, ecc...

Per questo studio sono state individuate cinque domande di ricerca, ossia:

**RQ1:** Quali sono i temi di ricerca affrontati nell'ambito della flakiness?

**RQ2:** Quali risultati sono stati raggiunti?

**RQ3:** Quali sono i limiti di questi studi?

**RQ4:** Sono stati studiati degli approcci di Machine Learning che possono aiutare a individuare un *flaky test*?

**RQ5:** Come affrontano il problema della flakiness gli sviluppatori delle grandi aziende?

Per ogni domanda di ricerca individuata è stata fatta anche una breve spiegazione sul perché bisognava affrontare quel tema, in modo da essere sicuri di avere ben chiaro cosa andare a cercare in seguito nella letteratura.

In particolare, per la **RQ1**, si vuole individuare e analizzare il focus degli studi fatti fino a ora (riconoscimento dei *flaky*, ricerca delle cause dei *flaky*, ecc.); per la **RQ2**, si vogliono individuare le soluzioni e le strategie che sono state sviluppate fino a ora; per quanto riguarda la **RQ3**, si vuole andare ad analizzare se gli

argomenti di ricerca affrontati sono limitati e su quali linguaggi di programmazione si stanno concentrando le diverse ricerche; per la **RQ4**, si vuole capire se sono stati affrontati degli studi sui *flaky test* con l'applicazione di metodologie di Machine Learning ed eventualmente se sono stati fatti dei confronti tra le tecniche con il ML e senza il ML, in modo da poter capire se l'uso del ML porta effettivamente dei miglioramenti. Infine, per quanto riguarda la **RQ5**, dato che la problematica dei *flaky test* affligge le grandi aziende, si vuole andare ad analizzare come gli sviluppatori affrontano la scoperta di un flaky e le tecniche che vengono usate per risolvere il problema. Per maggiore chiarezza, di seguito è riportata una tabella riassuntiva delle domande di ricerca e delle loro motivazioni.

<b>Domande di Ricerca</b>	<b>Motivi</b>
Quali sono i temi di ricerca affrontati nell'ambito della flakiness?	Si vuole individuare e analizzare il focus degli studi fatti fino a ora
Quali risultati sono stati raggiunti?	Si vogliono individuare le strategie e le soluzioni sviluppate fino a ora
Quali sono i limiti di questi studi?	Si vuole capire i limiti degli studi fatti fino a ora (linguaggi di programmazione usati, dimensione dei campioni analizzati, ecc....)
Sono stati studiati degli approcci di Machine Learning che possono aiutare a individuare un <i>flaky test</i> ?	Si vuole capire che tipologie di studi sono state affrontate nel Machine Learning e se sono stati fatti dei confronti con tecniche che non usano il ML
Come affrontano il problema della flakiness gli sviluppatori delle grandi aziende?	Si vuole analizzare come gli sviluppatori di grandi aziende affrontano la scoperta di un <i>flaky</i> e le tecniche usate per risolvere un <i>flaky test</i>

### 2.3.2 Strategia di ricerca

Una volta individuate le domande di ricerca bisogna individuare una strategia di ricerca nella letteratura. Le ricerche di studi primari partono dalle librerie digitali, ma spesso queste non sono sufficienti per una revisione sistematica. Altre fonti che si possono usare per la ricerca degli studi primari sono:

- Giornali;

- Grey literature<sup>12</sup>;
- Snowballing<sup>13</sup>;
- Internet.

Per la ricerca degli studi primari sono stati eseguiti i seguenti passi:

- Identificazione dei termini principali dalle domande di ricerca;
- Considerazioni di termini alternativi e di sinonimi delle parole chiavi individuate precedentemente;
- Combinazione di tutti i termini individuati con l'aiuto degli operatori booleani per formare delle query.

La ricerca è stata fatta su tre librerie digitali: **IEEEExplore**, **Scopus** e **ACM Library**. Si è deciso di ricercare in questi siti perché rappresentano le librerie maggiormente usate nella comunità dell'ingegneria del software. Nella seguente tabella sono riportate le query di ricerca fatte per ognuno di questi siti. Inoltre, sono state considerate solo le pubblicazioni fatte dal **01/01/2000** al **31/01/2020**.

<b>IEEEExplore</b>	((("flaky" OR "flakiness" OR "test pass and fail") AND "test" AND ("software" OR "software engineering" OR "computer science")) AND ( ("root cause" OR "detection" OR "classification") OR (( "framework" OR "strateg*") AND ("detect*" OR "minimiz*")) OR (( "machine learning" OR "ML") AND "approach") OR (( "developer" OR "tester") AND("large compan*") OR "Google" OR "Firefox" OR "Huawei" OR "software compan*"))))
<b>ACM Library</b>	[[All: "flaky"] OR [All: "flakiness"] OR [All: "test pass and fail"]] AND [All: "test"] AND [[All: "software"] OR [All: "software engineering"] OR [All: "computer science"]] AND [[All: "root cause"] OR [All: "detection"] OR [All: "classification"] OR [[[All: "framework"] OR [All: "strateg*"]]] AND [[All: "detect*"] OR [All: "minimiz*"]]] OR [[[All: "machine learning"] OR [All: "ml"]]] AND [All: "approach"]]] OR [[[All: "developer"] OR [All:

<sup>12</sup> La grey literature è l'insieme di testi non pubblicati attraverso i normali canali ufficiali, ma diffusi dagli stessi autori o da enti e organizzazioni pubbliche e private.

<sup>13</sup> La tecnica dello snowball è la ricerca di altri studi primari nella bibliografia degli studi primari già individuati

	"tester"]]] AND [[All: "large compan*"] OR [All: "google"] OR [All: "firefox"] OR [All: "huawei"] OR [All: "software compan*"]]]]
<b>Scopus</b>	(( ("flaky" OR "flakiness" OR "test pass and fail") AND "test" AND ( "software" OR "software engineering" OR "computer science" )) AND (( "root cause" OR "detection" OR "classification" ) OR (( "framework" OR "strateg*" ) AND ( "detect*" OR "minimiz*" ))) OR (( "machine learning" OR "ML" ) AND "approach" ) OR (( "developer" OR "tester" ) AND ( "large compan*" OR "Google" OR "Firefox" OR "Huawei" OR "software compan*" ))))

A termine di questa prima ricerca sono stati selezionati quattrocentoventuno articoli; per ogni articolo si è tenuta traccia delle seguenti informazioni:

- Archivio da cui era stato preso;
- Titolo;
- Tipo;
- Anno di pubblicazione;
- Autori;
- Rivista su cui era stato pubblicato;
- Note aggiuntive (ad es. il numero di pagine in cui era presente l'articolo nella rivista).

La tabella sottostante riporta il numero di articoli trovati per ogni archivio.

IEEE	Articoli trovati: 122
Scopus	Articoli trovati: 146
ACM Library	Articoli trovati: 153

### 2.3.3 Selezione degli studi

Una volta che sono stati individuati i potenziali studi primari, è necessario valutarne la rilevanza ai fini della nostra ricerca. I criteri di inclusione ed esclusione con cui fare la selezione degli studi, si basano sulle domande di ricerca individuate

precedentemente e devono essere usati per interpretare e classificare correttamente gli studi primari.

I criteri di inclusione applicati per questa revisione sistematica sono:

- Lettura dell'abstract per verificare se il paper è inerente alle RQ;
- Lettura degli articoli selezionati al punto precedente per verificare se il contenuto è di interesse per la revisione sistematica;
- Applicazione dello snowballing sugli articoli selezionati al principio precedente;

I criteri di esclusione utilizzati sono:

- Esclusione di articoli non interamente scritti in inglese;
- Esclusione di tutti gli artefatti non classificati come *journal*, *article* o *conference*;
- Esclusione di articoli che trattano il testing in maniera generica;
- Esclusione di articoli che trattano i *flaky test* come effetto collaterale di un altro studio.

Al termine dell'applicazione dei principi di inclusione ed esclusione, il numero totale di artefatti selezionati è stato di ventotto articoli. Inoltre, con l'applicazione dello snowballing sono stati considerati altri quattordici artefatti tra tesi di laurea (di cui poi sono stati individuati articoli riassuntivi), articoli presenti su blog online e slide.

Si è deciso di estendere lo snowball anche a classificazioni diverse da journal, article e conference perchè le informazioni sulla problematica della flakiness negli archivi digitali sono davvero limitate. Purtroppo, però la maggior parte delle informazioni riportate negli artefatti individuati con lo snowballing non avevano informazioni rilevanti per la revisione sistematica ma erano utili per una conoscenza generale della problematica.

### **2.3.4 Criteri di valutazione della qualità**

Una volta selezionati gli studi bisogna andare a valutare la qualità di essi. Questa valutazione è necessaria per i seguenti motivi:

- Fornire criteri di inclusione/esclusione ancora più dettagliati;
- Avere una guida sui risultati ottenuti nello studio;
- Fornire indicazioni su sviluppi futuri.

Le domande per la valutazione della qualità sono state selezionate dalla checklist presentata da **Kitchenham e Charters**. Inoltre, avendo individuato pochi artefatti per questa revisione, si è deciso di non fare delle valutazioni troppo stringenti poiché era possibile leggere tutti gli artefatti fino a ora approvati. Di seguito sono riportate le domande usate.

**Q1:** Gli obiettivi sono chiaramente indicati?

**Q2:** Il campione usato è rappresentativo della popolazione per cui si andranno a generalizzare i risultati?

**Q3:** Le dimensioni del campione sono state giustificate?

**Q4:** Se lo studio coinvolge la valutazione di una tecnologia, questa è stata chiaramente spiegata?

**Q5:** È stata data una risposta a tutte le domande di ricerca presentate?

**Q6:** Sono stati presentati tutti i risultati ottenuti (sia positivi che negativi)?

**Q7:** Il processo di ricerca è stato documentato in maniera dettagliata?

Una volta individuate le domande, si è deciso di non assegnare una valutazione binaria (0 o 1), ma di considerare diversi range come suggerito in altre revisioni sistematiche [1]. A ogni domanda è stato assegnato il seguente punteggio:

- 0 (No)
- 0,1 - 0,3 (Raramente)
- 0,4 - 0,6 (Parzialmente)
- 0,7 - 0,9 (Per lo più)
- 1 (Si)

Quindi, avendo sette domande per valutare la qualità, il punteggio complessivo è stato suddiviso nel seguente modo:

- 0 (No)
- 0,1 - 2,1 (Raramente)
- 2,2 - 4,5 (Parzialmente)

- 4,6 - 6,2 (Per lo più)
- 6,3 - 7,0 (Si)

Una volta definiti i range si è deciso di considerare gli articoli che avessero almeno una qualità parziale. Di seguito è riportata una tabella riassuntiva sulla qualità dei paper, mentre nell'Appendice A sono riportati i riferimenti agli articoli studiati con il relativo codice.

<b>Codice studio</b>	<b>Punteggio valutazione qualità</b>
S1	Parzialmente
S2	Raramente
S3	Per lo più
S4	Per lo più
S5	Per lo più
S6	Per lo più
S7	Per lo più
S8	Per lo più
S9	Per lo più
S10	Parzialmente
S11	Per lo più
S12	Per lo più
S13	Parzialmente
S14	Per lo più
S15	Si
S16	Parzialmente
S17	Per lo più
S18	Parzialmente
S19	Si
S20	Per lo più

S21	Per lo più
S22	Per lo più
S23	Per lo più
S24	Per lo più
S25	Per lo più
S26	Per lo più

È possibile notare che solo uno degli articoli non ha raggiunto una valutazione della qualità accettabile, quindi per l'estrazione dei dati sono stati considerati venticinque articoli.

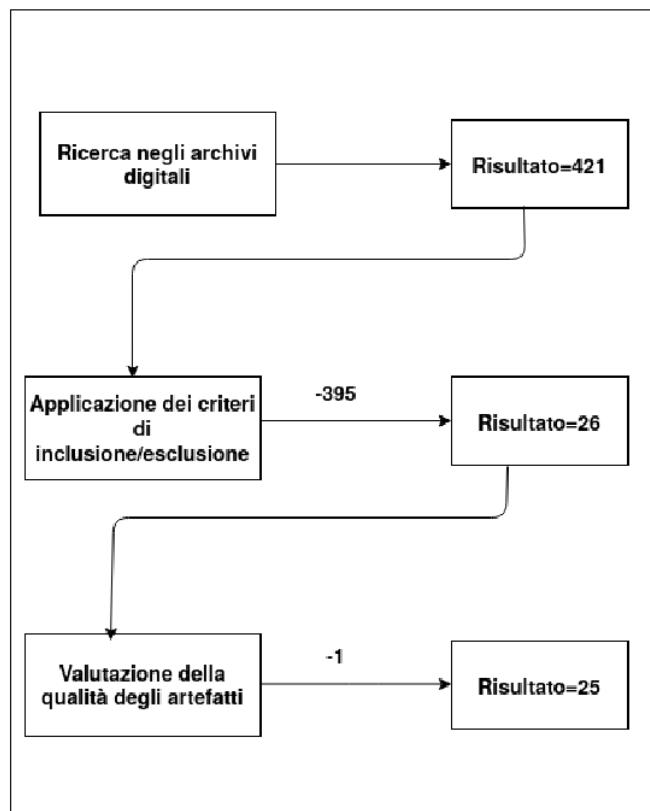
### 2.3.5 Estrazione dei dati

La fase successiva di una revisione sistematica della letteratura consiste nell'estrazione di dati utili dagli studi selezionati, in modo da poter rispondere alle RQ. Per ogni studio è stato fatta un'analisi su quanto fosse utile per la revisione sistematica, cioè si è andato ad analizzare a quante delle domande di ricerca individuate precedentemente potesse dare una risposta. Nella tabella sottostante sono riportati i risultati ottenuti.

ID studio	Research Question			
S1				RQ4
S2	RQ1		RQ3	
S3	RQ1	RQ2	RQ3	
S4	RQ1		RQ3	
S5	RQ1	RQ2	RQ3	
S6	RQ1			
S7	RQ1		RQ3	
S8	RQ1	RQ2	RQ3	
S9	RQ1		RQ3	

S10				RQ4	
S11	RQ1	RQ2	RQ3		
S12	RQ1	RQ2	RQ3		
S13					RQ5
S14	RQ1	RQ2	RQ3		
S15				RQ4	
S16	RQ1		RQ3		
S17	RQ1		RQ3		
S18	RQ1	RQ2	RQ3		
S19	RQ1				
S20					RQ5
S21	RQ1	RQ2	RQ3		
S22	RQ1				
S23	RQ1				RQ5
S24	RQ1	RQ2			
S25	RQ1		RQ3		

In Figura 7 è riportata una breve sintesi del numero di paper che sono stati trattati in ogni fase.



**Figura 7: Sintesi ricerca degli studi primari**

### 2.3.6 Sintesi dei dati

Questa fase ha l'obiettivo di collezionare e riassumere i dati estratti dagli studi primari. La sintesi dei dati deve essere descrittiva e non quantitativa, anche se è possibile inserire alla fine della revisione sistematica una sintesi della ricerca fatta con i diversi risultati quantitativi. Di seguito sono riportati i risultati ottenuti e i dati estratti dagli studi primari, suddivisi in base alle domande di ricerca.

Per sintetizzare i risultati ottenuti da questa revisione sistematica si è usato un metodo narrativo per analizzare i dati e dei grafici o tabelle per mostrare i vari risultati.

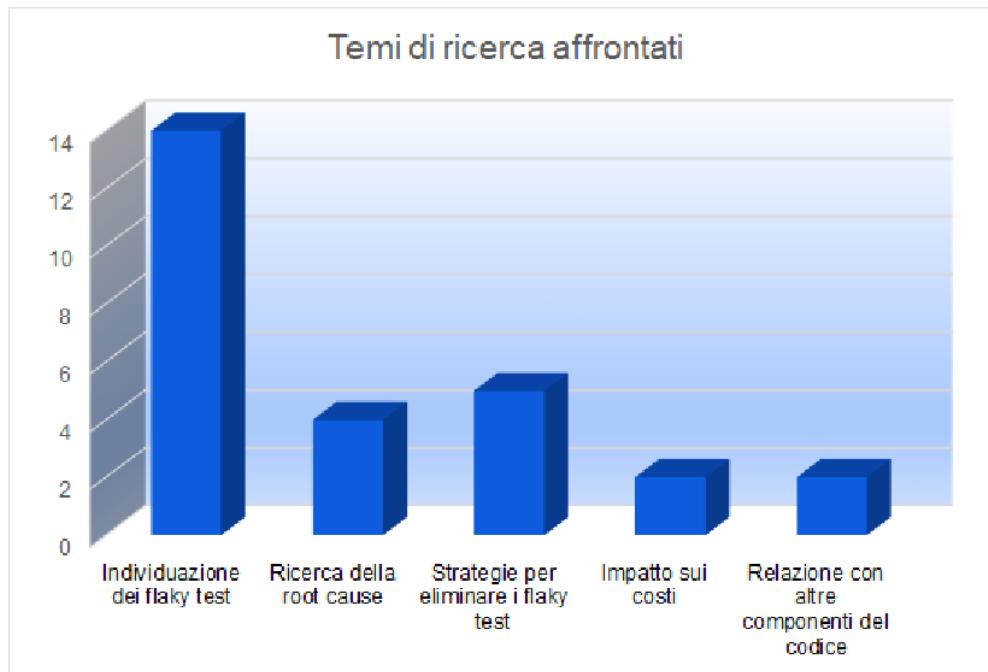
### 2.3.7 Risultati e discussioni

#### 2.3.7.1 RQ1: Quali sono i temi di ricerca affrontati nell'ambito dei flaky test?

Sono stati analizzati i diversi temi di ricerca affrontati nei ventidue articoli che rispondono a questa domanda di ricerca. I temi affrontati sono:

- Individuazione dei *flaky test*;
- Ricerca delle *root cause*;
- Strategie per eliminare i *flaky test*;
- Impatto sui costi;
- Relazione con altre caratteristiche del codice.

In Figura 8 sono riportati per ogni tema il numero di articoli che si sono occupati di quel determinato ambito. È possibile notare che su ventidue artefatti, quattordici (63,63%) si occupano di strategie per individuare *flaky test*, quattro (18,18%) si occupano di strategie per ricercare la *root cause* di un *flaky test*, cinque (22,72%) si occupano di strategie per eliminare i *flaky test* dalle classi di test, due (9,09%) si occupano dell'impatto che hanno i *flaky test* sui costi e infine due (9,09%) si occupano di individuare delle relazioni tra i *flaky test* e altre caratteristiche del codice (ad esempio, la generazione di test automatizzata).



*Figura 8: Diagramma riassuntivo RQ1*

### 2.3.7.2 RQ2: Quali risultati sono stati raggiunti?

Scopo di questa *research question* era capire le soluzioni e le strategie messe in atto nell’ambito della ricerca per la problematica della flakiness. Dallo studio fatto in letteratura si è osservato che i risultati raggiunti sono prevalentemente in due ambiti:

- Individuazione dei *flaky test* e classificazione di essi;
- Risoluzione dei *flaky test*.

Per quanto riguarda l’individuazione dei *flaky test* e la loro classificazione, i risultati più importanti raggiunti fino a ora sono due framework, **DeFlaker** e **iDFlakies**. Il primo framework individua la presenza di un *flaky* facendo un’analisi della coverage raggiunta all’interno di un progetto. Il funzionamento prevede che vengano eseguiti i casi di test per vedere la coverage, successivamente si confronta la coverage attuale con le precedenti e si marca come *flaky* un caso di test che mostra un fallimento che prima non presentava (ovviamente è un *flaky* se il fallimento si presenta per un caso di test per cui non è stata fatta alcuna modifica al codice).

Il secondo framework individua un *flaky test* all'interno di un progetto effettuando diverse esecuzioni delle varie classi di test, ogni volta con ordini diversi (sia delle classi che dei metodi) per verificare se in uno di queste esecuzioni si presenta un fallimento del caso di test.

Per quanto riguarda la classificazione dei *flaky test*, il framework **iDFlakies** effettua una classificazione parziale dei *flaky test*. Essi possono essere classificati come dipendenti dall'ordine e non dipendenti dall'ordine. Per effettuare questa classificazione, si effettuano i seguenti passi:

1. I casi di test vengono eseguiti con un determinato ordine;
2. Se viene riscontrata una *failure*, la suite di test viene nuovamente eseguita con lo stesso ordine;
3. Se la *failure* si presenta nuovamente allora il caso di test viene marcato come dipendente dall'ordine;
4. Se la *failure* non si presenta allora il caso di test viene marcato come non dipendente dall'ordine.

Rilevante nell'identificazione dei *flaky test* è il tool **NONDEX** che si occupa del riconoscimento di *flaky* in codice ADINS (*Assumes a Deterministic Implementation of a method with a Non-deterministic Specifications*). Per implementare questo tool, i ricercatori hanno prima rilevato trentuno metodi con specifiche **non deterministiche** nella libreria Standard Java, poi hanno manualmente costruito modelli **non deterministici** per questi metodi e infine hanno usato una Macchina Virtuale Java per esplorare le varie scelte **non deterministiche**.

Caratteristiche simili sono presenti nel tool **PRADET**, anch'esso utile per individuare *flaky test* dipendenti dall'ordine, che effettua un'analisi dinamica delle diverse esecuzioni del codice.

È importante sottolineare che alcuni studi si sono concentrati su singoli aspetti dei *flaky test*. Ne è un esempio il tool **ElectricTest**, il quale si occupa di individuare le dipendenze tra i diversi casi di testi di progetti **Java**.

Un altro ambito in cui la ricerca ha mosso i primi passi sono le applicazioni Android. Infatti, grazie a questa revisione sistematica sono stati individuati degli studi che hanno come obiettivo il riconoscimento e le possibili risoluzioni dei *flaky test* in Android. Per quanto riguarda le applicazioni Android, è molto importante

sottolineare che è stata individuata un’altra causa dei *flaky test*, legata a una mancata comunicazione tra le diverse componenti che compongono un’applicazione Android.

Per quanto riguarda il secondo ambito, cioè la risoluzione di *flaky test*, anche qui ci sono stati individuati diversi artefatti, ognuno con diverse soluzioni. Per quanto riguarda le applicazioni Android, non sono state individuate delle vere e proprio soluzioni ma degli accorgimenti che devono essere messi in pratica per cercare di diminuire i *flaky test*. Sono stati individuati cinque accorgimenti:

1. Migliorare l’implementazione, riducendo o rimuovendo componenti non deterministiche;
2. Replicare l’implementazione, sostituendo l’implementazione di vecchie versioni delle librerie con nuove versioni della stessa libreria;
3. Modificare le asserzioni, soluzione utile quando il comportamento *flaky* è causato da assunzioni sbagliate dello sviluppatore;
4. Ripetere le procedure di installazione;
5. Rimuovere il caso di test, “soluzione” che può essere adottata quando il *flaky* dipende da cause sconosciute o dalla dipendenza dall’ordine di esecuzione.

Nell’Università dell’Illinois invece è stato sviluppato un framework, **iFixFlakies**, che ha l’obiettivo di aggiustare automaticamente i *flaky test* dipendenti dall’ordine.

Anche per quanto riguarda il linguaggio a blocchi Scratch, si è presentata la problematica della flakiness legata all’uso dei numeri casuali. Per evitare il verificarsi di un *flaky test*, il tool **WHISKER** (usato per il testing di Scratch), può fare uso di un *seed*<sup>14</sup> quando va a inizializzare il generatore di numeri casuali.

Infine, un’altra strategia individuata per la risoluzione dei *flaky test* derivanti dall’uso di test automatizzati è l’uso di classi di test con codice mutato<sup>15</sup>.

---

<sup>14</sup> Un seed è un numero usato per inizializzare un generatore di numeri pseudo randomico. Se si va a usare sempre lo stesso seed non si avranno numeri casuali diversi ma verrà inizializzata sempre la stessa sequenza.

<sup>15</sup> Il mutation testing viene usato per valutare la qualità di test già esistenti. Il mutation test introduce delle piccole modifiche al programma, e verifica se queste modifiche comportano dei cambiamenti all’interno del sistema.

Di seguito è riportata una tabella riassuntiva con tutti i risultati che sono stati raggiunti e che sono stati spiegati precedentemente in ordine cronologico.

Individuazione dei flaky test e classificazione dei flaky test	
2015	<b>Electric Test</b>
2016	<b>NonDex</b>
2018	Individuazione dei flaky in Android
2018	<b>Pradet</b>
2018	<b>DeFlaker</b>
2019	<b>IDFlakies</b>

Risoluzione dei flaky test	
2017	Mutation Testing
2018	Possibile risoluzione dei flaky in Android
2019	iFixFlakies
2019	Whisker

### 2.3.7.3 RQ3: Quali sono i limiti di questi studi?

I limiti degli artefatti individuati per la revisione sistematica della letteratura sono sostanzialmente due:

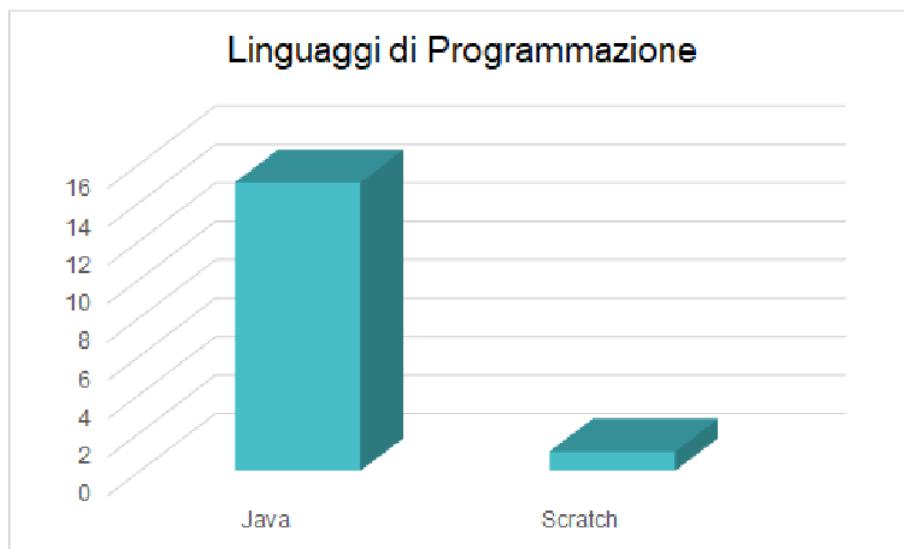
- Linguaggio di programmazione;
- Dimensione del campione su cui sono stati condotti gli esperimenti.

Per quanto riguarda il primo limite, la maggior parte degli studi si sono concentrati sul linguaggio di programmazione **Java**. L'essersi concentrati solo su progetti scritti con questo linguaggio di programmazione, probabilmente deriva dalla grande diffusione che ha **Java**. Ancora oggi infatti, **Java** risulta essere il linguaggio di programmazione più diffuso al mondo <sup>[4]</sup>, quindi, essendosi

concentrati su progetti open source risulta molto facile aver trovato soprattutto progetti scritti con questo linguaggio.

Per quanto riguarda le dimensioni del campione, in alcuni studi sono stati usati dei campioni veramente ristretti, come nello studio per riconoscere *flaky test* in applicazioni Android (hanno usato ventinove progetti). Nella maggior parte dei casi però le dimensioni del campione sono state inserite come minaccia alla validità dello studio, inoltre sono anche stati spiegati i motivi dietro a una scelta così ristretta (ad esempio un'analisi manuale del codice).

Di seguito sono riportati dei grafici in cui si vanno a riassumere i linguaggi di programmazione usati e le dimensioni dei campioni utilizzati (Figura 9 e Figura 10).



*Figura 9: Linguaggi di programmazione riscontrati negli studi primari*



**Figura 10:** Dimensioni dei campioni analizzati negli studi primari

Per il grafico presente in Figura 10, ci riferiamo alla categoria “Altro” per campioni in cui non sono stati analizzati interi progetti ma singole classi di test.

#### **2.3.7.4 RQ4: Sono stati studiati degli approcci di Machine Learning che possono aiutare a individuare i flaky test?**

Per rispondere a questa domanda di ricerca si potrà fare affidamento su un unico artefatto. In questo studio hanno costruito un modello di rete bayesiana<sup>16</sup> per poter individuare la flakiness nei test automatizzati. Con questo approccio i *flaky test* vengono trattati come “malattie” con determinati “sintomi” che possono facilitare l’identificazione delle “cause”.

L’obiettivo di questo studio può essere riassunto nei seguenti punti:

- Individuazione di un *flaky test* partendo da alcune caratteristiche osservabili dall’utente;
- Identificazione di alcune metriche a supporto della collezione di dati;
- Implementazione di un tool per la collezione di dati automatizzata;
- Creazione di un set di training;
- Addestramento e validazione del modello;

---

<sup>16</sup> Una rete bayesiana è un modello grafico probabilistico che rappresenta un insieme di variabili aleatorie con le loro dipendenze condizionali attraverso l’uso di un grafo aciclico diretto (DAG).

- Rilascio del modello che potrà essere usato per individuare *flaky* non ancora conosciuti.

Purtroppo, nell’artefatto analizzato era presente solo una bozza del lavoro che si vorrà svolgere in futuro. Questo quindi dimostra che l’ambito della flakiness collegata al Machine Learning è ancora un terreno poco esplorato; inoltre, non sono stati individuati degli studi in cui sono stati fatti confronti tra la ricerca dei *flaky* con il Machine Learning e senza.

### **2.3.7.5 RQ5: Come affrontano il problema della flakiness gli sviluppatori delle grandi aziende?**

Essendo i *flaky test* un problema che interessa non solo i ricercatori ma anche gli sviluppatori delle grandi aziende, è importante capire a oggi come è stata affrontata questa problematica. Due studi sono stati condotti coinvolgendo gli sviluppatori di **Firefox**. Nel primo studio è stato riportato che gli sviluppatori tendono a ignorare il fallimento di un test se esso si presenta troppo spesso. Questa però non risulta essere una strategia vincente, visto che nella maggior parte dei casi il presentarsi di un *flaky test* indica comunque la presenza di un errore nel sistema. In uno studio successivo a quello descritto precedentemente, sono state raccolte informazioni sulle sfide che gli sviluppatori affrontano quando si presenta un *flaky test*. Queste sfide sono raggruppate nelle seguenti categorie:

- **Contesto in cui avviene la failure:** capire il contesto in cui avviene un fallimento risulta essere una delle cose più difficili da fare, soprattutto per *flaky test* che si presentano raramente;
- **Capire le root cause della flakiness:** spesso questa attività risulta essere molto complessa per gli sviluppatori;
- **Capire le componenti del codice coinvolte nella flakiness:** spesso questa attività risulta essere critica perché non sempre il codice scritto rispetta determinati criteri di qualità;
- **Fixare un flaky test:** essendo un processo manuale risulta essere complesso da eseguire, per questo motivo la soluzione “preferita” dagli sviluppatori è la riesecuzione del caso di test che ha mostrato una *failure*.

Oltre a fare delle analisi empiriche sulla percezione che hanno gli sviluppatori, in uno studio è stato anche presentato un tool che potesse aiutare a capire, attraverso i file di log, quali metodi fossero responsabile della flakiness. Questo tool, chiamato **RootFinder**, è stato integrato a un framework sviluppato dalla Microsoft.

In conclusione, è evidente che la strada che spesso viene seguita è proprio quella del rieseguire un caso di test fino a quando non avviene un successo nell'esecuzione. Purtroppo, questa strada non potrà essere seguita per sempre, perché appare sempre più chiaro che i costi che comporta una soluzione simile sono davvero elevati.

### 2.3.8 Minacce alla validità

Le linee guida fornite da **Kitchenham e Charters** analizzano le minacce alla validità dello studio da tre aspetti diversi: bias nella selezione degli studi, bias nella pubblicazione e scarsa accuratezza nell'estrazione dei dati.

Per quanto riguarda la selezione degli studi, sono state usate delle stringhe di ricerca il più ricche possibile di parole chiavi e di sinonimi. Nonostante questo, è possibile che alcuni studi rilevanti non siano stati selezionati dalle query usate nei tre database elettronici. Per mitigare questa minaccia, si è deciso di applicare la tecnica dello *snowballing*, cioè la ricerca di artefatti rilevanti nei riferimenti bibliografici di articoli già selezionati. Un'altra minaccia alla validità dello studio deriva dall'aver fatto selezionare gli studi da una sola persona. Questa minaccia è stata in parte mitigata perché in caso di dubbi è stato chiesto a una persona esterna allo studio un parere sull'artefatto che si stava analizzando.

Infine, per quanto riguarda l'estrazione dei dati, è stata rilevata come minaccia alla validità l'aver fatto collezionare i dati a una sola persona; questo potrebbe aver portato a collezionare dati errati o che non sono stati interpretati correttamente.

### 2.3.9 Conclusioni

In questo capitolo è stata presentata una revisione sistematica della letteratura per poter capire i passi che sono stati fatti fino a ora nell'ambito della flakiness. La

ricerca è stata fatta dal 2000 al 2020 e sono stati selezionati e analizzati venticinque studi primari. I principali risultati che sono stati ottenuti da questo lavoro sono:

1. RQ1: i temi trattati nei diversi studi sulla flakiness sono sostanzialmente cinque: individuazione dei *flaky test* all'interno di progetti open source, ricerca delle *root cause* di un *flaky test*, strategie per mitigare ed eliminare i *flaky test*, impatto sui costi di progetto e relazioni con altre caratteristiche del codice.
2. RQ2: i risultati raggiunti fino a oggi sono in due ambiti: individuazione dei *flaky test* con eventuale classificazione della causa scatenante e risoluzione dei *flaky test*. Per entrambi gli ambiti sono stati sviluppati e resi disponibili open source diversi framework, soprattutto per l'individuazione e classificazione dei *flaky test*.
3. RQ3: sono stati evidenziati dei limiti in questi studi, legati alle dimensioni dei campioni analizzati (in alcuni casi inferiori a venti progetti) e ai linguaggi di programmazione per cui sono stati sviluppati i framework (prevolentemente **Java**).
4. RQ4: il machine learning risulta essere ancora oggi un territorio poco esplorato nella ricerca dei *flaky test*; è stata individuata una sola proposta di sviluppo legata alle reti bayesiane per poter ricercare i flakiness nei test automatizzati.
5. RQ5: da interviste effettuate con gli sviluppatori di Mozilla è apparso evidente come la problematica flaky sia molto difficile da affrontare per loro. Infatti, sono state evidenziate diverse sfide che ogni sviluppatore deve affrontare quando rileva un *flaky test*, la più difficile riguarda l'individuazione della *root cause* una volta che si è manifestato il comportamento *flaky*. Ancora oggi la soluzione più adottata in ambito aziendale risulta essere il *rerun* del test con comportamento **non deterministico** fino a quando non si ottiene un successo.

Da questa revisione sistematica è apparso evidente che i *flaky test* sono ancora un argomento poco conosciuto e di interesse sempre più crescente sia in ambito aziendale che didattico. Gli studi futuri dovrebbero concentrarsi soprattutto

sull'applicazione di qualche strategia di intelligenza artificiale per la ricerca dei *flaky* da poter eventualmente proporre anche in ambito aziendale.

# CAPITOLO 3

## 3.1 Un nuovo “inizio”

Analizzando gli artefatti utili per la revisione sistematica della letteratura, ci si è resi conto che ci sono una serie di problematiche non ancora affrontate, cioè:

- I diversi framework sono adatti a un solo linguaggio di programmazione (**Java**);
- Quasi sempre viene considerato un unico sistema di build (**Maven**);
- I dataset rilasciati e che possono essere replicati sono pochi.

Appare quindi evidente che ancora oggi si ha una conoscenza “limitata” dei flakiness e che si stanno ancora affrontando diverse sfide.

Per cercare di ampliare le conoscenze del settore informatico (non solo quello sui *flaky test*) ogni anno le grandi multinazionali come Facebook invitano i ricercatori a sottomettere le proprie idee su determinati temi (selezionati da Facebook stessa), in modo che un team di esperti possa valutarle ed eventualmente finanziare i ricercatori per dar loro la possibilità di sviluppare l’idea considerata “rivoluzionaria”.

Nel 2019 il tema di ricerca affrontato durante il “*Testing and Validation Symposium*” della Facebook è stato proprio quello della flakiness. Tra le idee risultate vincenti <sup>[5]</sup>, vi è la proposta da cui è partito questo lavoro di tesi, ossia l’esecuzione ripetuta di un *flaky test* in diversi cluster<sup>17</sup>. Ogni cluster deve esplorare un certo ambito **non deterministico** (ossia le *root cause* dei *flaky* come la concorrenza, il network, etc.) in un container software dedicato e un generatore di carico di risorse guidato da fuzzy<sup>18</sup>; il cluster che manifesta l’insieme più bilanciato (o sbilanciato) di successi e fallimenti nelle esecuzioni sarà quello che spiegherà al meglio il tipo di flakiness del test. Nella proposta presentata, è previsto l’uso di tecniche di intelligenza artificiale e di approcci statistici per poter avere dei risultati più accurati. Il vantaggio di questo nuovo approccio è sicuramente l’indipendenza

---

<sup>17</sup> Il cluster indica un gruppo di elementi tra loro omogenei in un insieme di dati.

<sup>18</sup> La logica fuzzy è un’estensione della logica booleana, in cui si va ad attribuire ad ogni proposizione un grado di verità, indicato con un valore compreso tra 0 e 1, estremi esclusi.

dal linguaggio di programmazione in cui è stato implementato un sistema, dal sistema di build utilizzato e dalle piattaforme su cui è sviluppato.

## 3.2 Ricerca di dati su Github

Appare evidente che la prima necessità per sviluppare questa nuova idea sia la ricerca di nuovi *flaky test* e delle loro *root cause*. La ricerca di nuovi dati è avvenuta tramite il sito di Github e con due approcci diversi:

- Una ricerca automatizzata nei progetti *open* di grandi multinazionali (Google, Microsoft, Mozilla, ecc.);
- Una ricerca manuale a livello globale nel sito di Github.

### 3.2.1 Primo approccio per la ricerca di dati

In un primo momento si è deciso di effettuare la ricerca di *flaky test* sfruttando le API<sup>19</sup> di Github. Tramite Python, si è cercato di usare le API per accedere alle *issue* dei progetti delle grandi organizzazioni e di ricercare all'interno delle *issue* riferimenti alla problematica flakiness per quel progetto.

Purtroppo, questo approccio è stato scartato perché le API permettono un numero limitato di accessi e per questa ricerca il numero di accessi consentiti si esauriva molto velocemente.

### 3.2.2 Secondo approccio per la ricerca di dati

Successivamente, si è deciso di effettuare una ricerca manuale con alcune parole chiavi nelle *issue* di Github. Infatti, su GitHub è possibile ricercare delle stringhe nel titolo o nel corpo di una *issue*, sia aperta che chiusa. Si è deciso di considerare solo le *issue* aperte perché in questo modo il *flaky* era presente nella versione attuale del progetto. Parte della ricerca delle *issue* è riportata in Figura 11.

---

<sup>19</sup> Con il termine API (Application Programming Language) si indicano una serie di procedure usate per poter completare un determinato compito. Spesso, con questo termine, si vanno a indicare le librerie software di un linguaggio di programmazione.

Created	Assigned	Mentioned	Search Bar	Visibility	Organization	Sort
① 5,843 Open ✓ 16,106 Closed			Is:open is:issue flaky in:title,body			
① golang/go runtime: TestGdbPythonCgo is flaky on linux-mips64le-rtrk builder NeedsInvestigation				Visibility	Organization	Sort
#37794 opened 1 hour ago by bcmills ↗ Backlog				1		
① gluonhq/substrate [Github Actions] Flaky linux builds help wanted						
#432 opened 2 hours ago by abhinayagarwal						
① prestosql/presto Flaky test TestWebUi.testWorkerResource bug test				1		
#3069 opened 2 hours ago by findpepi						
① chainer/chainer 'TestMultiprocessIteratorStalledDatasetDetection' is flaky				1		
#8551 opened 3 hours ago by tosunar						
① knative/serving [flaky] pkg/reconciler/autoscaling/kpa.TestControllerCreateError auto:flaky				1		
#7209 opened 3 hours ago by knnative-test-reporter-robot						
① omisego/plasma-contracts fix flaky python test						
#606 opened 8 hours ago by bootalish						
① hrbrmstr/RSwitch RStudio server sessions copy/paste bug				4		
#16 opened 12 hours ago by dawnwilson						
① kubernetes/minikube Flaky test: TestFunctional/parallel/CertSync co/kic-driver kind/failing-test kind:flake						
#6993 opened 13 hours ago by medyagh						
① kubeflow/pipelines Presubmit/Postsubmit/Sample test flakiness and long running time area/testing priority:p1						
#3256 opened 15 hours ago by jingzhang36						
① flutter/flutter Include a search option on the build dashboard team team: Infra				1		
#52370 opened 16 hours ago by Plinks						
① kubernetes/kubernetes TestStressingCascadingDeletion is flaky kind:flake sig/api-machinery				1		
#89021 opened 17 hours ago by tedyu						
① ampproject/amphtml Story responsive font-sizing seems to be flaky P2: Soon Type: Bug WG: stories						
#27185 opened 18 hours ago by gmajoulet						
① wix/yoshi Integration test: flaky stylable test Bug Infrastructure				1		
#2086 opened 19 hours ago by yanivefraim						
① ampproject/amphtml Frequent Percy failures P2: Soon Related to: Flaky Tests WG: Infra				5		
#27181 opened 20 hours ago by samouri						

**Figura 11: Esempio di issue su Github**

Obiettivo di questa ricerca era individuare i progetti in cui si parlava di *flaky test*, scaricarli e successivamente analizzarli. Purtroppo, anche questo approccio è stato scartato per due motivi:

1. Essendo le issue scritte da persone non sempre esperte del settore, non sono sempre riportate informazioni utili, quindi risultava difficile capire la classe e/o il metodo con un comportamento *flaky*;
2. Spesso le issue vengono aperte, ma non vengono chiuse quando un problema è risolto, quindi risultava difficile capire a quale commit del progetto risalire per poter trovare il *flaky test*.

Nonostante questo approccio sia stato scartato è importante sottolineare un vantaggio della ricerca su Github, ossia la possibilità di effettuare la ricerca dei *flaky* anche su progetti implementati in un linguaggio di programmazione diverso da **Java**.

### 3.3 Analisi dei dati già esistenti

Visti gli scarsi risultati ottenuti nella ricerca di nuovi *flaky* test, si è deciso di fare un'analisi dei dataset già presenti in letteratura, in particolare del dataset rilasciato con il framework **DeFlaker** <sup>[6]</sup>. Come già spiegato precedentemente, **DeFlaker** è un framework che indica la presenza o meno di un *flaky test* in base alla coverage. Infatti, **DeFlaker** tiene traccia delle coverage di tutti gli statement che sono stati modificati dall'ultima build avvenuta con successo e in caso di fallimento di un caso di test fa un controllo sulla coverage per vedere se il test tocca degli statement che sono appena stati modificati. In caso sia stata fatta una modifica al codice allora il test potrebbe essere fallito a causa della modifica, in caso negativo allora il caso di test viene marcato come *flaky*.

Il dataset rilasciato dagli ideatori di **DeFlaker** contiene le seguenti informazioni:

- Nome del progetto;
- *Sha*<sup>20</sup> della commit;
- Nome della classe;
- Nome del metodo.

In Figura 12 è riportata una parte del dataset di **DeFlaker**.

---

<sup>20</sup> Con questo termine si vuole indicare una famiglia di cinque funzioni crittografiche di hash sviluppate dal 1993.

**Figura 12:** Esempio dataset di DeFlaker

È possibile notare che nel dataset di **DeFlaker** non viene fornita alcuna indicazione sulla *root cause* dei *flaky test* individuati, quindi si è deciso di eseguire nuovamente questi casi di test e di verificare la *root cause* dei *flaky test*. Di seguito, si discuterà di due approcci che sono stati tentati per individuare le *root cause* dei *flaky test*.

### 3.3.1 Primo approccio: Gradle

Dato che la maggior parte degli approcci fino ad ora adottati prevedeva l'uso di **Maven**, si è tentato un approccio diverso con un altro sistema di build: **Gradle**. Al contrario di **Maven** che definisce il ciclo di vita di un processo, **Gradle** utilizza un grafo aciclico diretto (DAG)<sup>21</sup> per determinare l'ordine in cui i processi possono essere eseguiti. Quindi, in caso di modifiche al progetto, l'esecuzione risulterà essere più rapida con **Gradle** perché si andranno a eseguire solo le componenti modificate (a differenza di **Maven** che a ogni modifica esegue nuovamente tutte le classi del progetto).

Essendo però tutti progetti Maven, come prima cosa è stata effettuata una conversione automatica del sistema di build con il comando *gradle init*. Successivamente si è tentato di eseguire il comando *./gradlew :moduleTest:test --tests "path del caso di test da eseguire"* che permette l'esecuzione di un singolo caso di test presente nel progetto. Quindi, supponendo di voler testare il metodo *testDescribeKeyspaces* presente nel repository **Hector**<sup>22</sup>, i passi da dover eseguire sono i seguenti:

- Clono il progetto con il comando *git clone*;
- Entro nella directory del progetto ed effettuo il comando *git checkout* seguito dallo *sha* indicato nel dataset di **DeFlaker**;
- Eseguo il comando *gradle init* per la conversione automatica da **Maven** a **Gradle**;

---

<sup>21</sup> Un grafo aciclico diretto è un particolare tipo di grafo che non ha cicli diretti, cioè comunque scegliamo un vertice del grafo non possiamo tornare ad esso percorrendo gli archi del grafo.

<sup>22</sup> <https://github.com/hector-client/hector>

- Eseguo il comando `./gradlew :hector-core:test -tests "me.prettyprint.cassandra.service.CassandraClusterTest.testDescribeKeysPacs"`. Per poter usare questo comando bisogna indicare il sotto modulo più vicino al caso di test che si vuole testare (in questo caso `hector-core`) e tutto il path, dal sotto modulo al caso di test.

Purtroppo, l'esecuzione con **Gradle** è risultata molto complessa perché, trattandosi di progetti vecchi, **Gradle** non sempre riusciva a fare delle conversioni ottimali in quanto spesso non trovava le versioni delle dipendenze indicate che dovevano essere scaricate. Per cercare di ovviare a questo problema, prima di effettuare la conversione si è utilizzato il comando `mvn dependency:copy-dependencies`. Questo comando ha l'obiettivo di scaricare le dipendenze del progetto dal repository in una cartella specificata dallo sviluppatore. Purtroppo, nemmeno in questo modo si è riusciti ad avere una esecuzione ottimale dei progetti, quindi si è deciso di rinunciare a questo approccio e di trovare un'altra soluzione.

### 3.3.2 Secondo approccio: Maven

Scartato quindi l'approccio con **Gradle**, si è tornati su **Maven** e si cercata una soluzione che potesse ottimizzare il più possibile l'esecuzione dei vari casi di test. La costruzione di un nuovo dataset è avvenuta in diversi step, riassunti nei successivi tre punti:

1. Pulizia del dataset iniziale;
2. Installazione delle dipendenze con **Maven**;
3. Esecuzione del caso di test che si analizzando e successiva costruzione del nuovo dataset.

#### 3.3.2.1 Pulizia del dataset iniziale

Il primo passo è stata l'analisi e la pulizia del dataset iniziale; il dataset rilasciato da **DeFlaker**, era una raccolta di 5075 *flaky test* individuati in diversi progetti open source.

Per prima cosa, sono state selezionate casualmente centocinquantacinque righe da questo dataset, facendo attenzione a non considerare il progetto **Hadoop**

(scelta necessaria perché il sistema di build impiega circa diciassette ore per completare una singola esecuzione). Una volta selezionate queste righe si è deciso di sostituire il nome dei repository presente in **DeFlaker**, con il link di Github ai repository, ottenendo quindi questo nuovo dataset iniziale (Figura 13).

	column 1	column 2	column 3
2	https://github.com/apache/coyote.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.command.coord.TestCoordNotificationCommand
3	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.command.coord.TestCoordJobXCommand
4	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
5	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
6	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
7	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
8	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
9	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
10	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
11	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
12	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
13	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
14	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
15	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
16	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
17	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
18	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
19	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
20	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
21	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
22	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
23	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
24	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
25	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
26	https://github.com/apache/oozie.git	5ee5554654a1bbe6240cra050722bd459773926	org.apache.coyote.coord.coord.coord.TestCoordKillAllAction
27	https://github.com/dropwizard/dropwizard	c4b3a27349f6124ed3d0038a9a19617437fb	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
28	https://github.com/dropwizard/dropwizard	6m0s5f66a8295e6f87c4d5776a9a946	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
29	https://github.com/dropwizard/dropwizard	25fd0d6bc8915620208fe20a7b59110407123977	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
30	https://github.com/dropwizard/dropwizard	7fec2f40b0b15160c734520561ea577288c10e758102341616a	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
31	https://github.com/dropwizard/dropwizard	827045163c67be575a97723459773926	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
32	https://github.com/dropwizard/dropwizard	84e2054cd6e2d26723a99aa71833cb4217795e	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
33	https://github.com/dropwizard/dropwizard	c0f24ad78ba3a0de0efbcb731b10147194e	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
34	https://github.com/dropwizard/dropwizard	81575391cf12a06d31e981d238986e99952e66	io.dropwizard.server.DeafauServerFactory testLesGraceUShutdown
35			https://github.com/dropwizard/dropwizard

**Figura 13:** Nuovo dataset di DeFlaker

Successivamente, è stato implementato uno script in *bash*<sup>23</sup> che permettesse una formattazione del dataset.

```
#!/bin/bash

common() {
    string1=$pathFile
    string2=${arrayPom[$i]}
    first_diff_char=$(cmp <(< echo "$string1" ) <(< echo "$string2"
) | cut -d " " -f 5 | tr -d ",")
    ris=${string1:0:$((first_diff_char-1))}
    echo "$ris"
}

getclassName() {
    local stringPath=$testName
    IFS='.' read -ra ADDR <<< "$stringPath" # str is read into
an array as tokens separated by IFS
    sizeArray=${#ADDR[@]}
    className=${ADDR[sizeArray-2]}
    echo "$className"
}

getNameMethod() {
    local stringPath=$testName
    IFS='.' read -ra ADDR <<< "$stringPath" # str is read into
an array as tokens separated by IFS
    sizeArray=${#ADDR[@]}
    nameMethod=${ADDR[sizeArray-1]}
    echo "$nameMethod"
}

cloneAndCheckRepo() {
    local link=$linkRepo
    local sha=$shaCommit
    local baseDir=$BASEDIR
    local repoFolder=$REPOFOLDER
```

---

<sup>23</sup> Acronimo di Bourne Again Shell, s' tratta di un interprete di comandi che permette all'utente di comunicare con il sistema operativo attraverso una serie di funzioni predefinite, oppure di eseguire programmi e script.

```

cd $baseDir
cd $repoFolder
echo $PWD
echo "cloning repo... " $linkRepo
git clone $linkRepo
basename=$(basename $link)
nameRepo=${basename%.*}
echo "cd " $nameRepo
cd $nameRepo
echo "git checkout " $sha
git checkout $sha
}
BASEPATH=$1
REPOFOLDER=$2
fileOutput=$BASEPATH"outDeFlaker.csv"
maxLength=-1
path=$BASEPATH"dataset.csv"
while IFS= read -r line
do
IFS=', ' read -r -a array <<< "$line"
linkRepo=${array[0]}
shaCommit=${array[1]}
testName=${array[2]}
cloneAndCheckRepo "$linkRepo" "$shaCommit" "$BASEPATH"
"$REPOFOLDER"
className=$(getClassName)
nameMethod=$(getNameMethod)
pathFile=$(find -name "$className".java)
readarray -d '' arrayPom < <(find . -name pom.xml -print0)
lengthArray=${#arrayPom[@]}
var=$((lengthArray -1))
for i in $( seq 0 $var )
do
common $pathFile ${arrayPom[$i]}
length=${#ris}
if [ $length -gt $maxLength ]
then
maxLength=$length

moduleName=$ris
echo "qui c'è longpath=\"$moduleName"

```

```

        fi
done
echo
$nameRepo,$linkRepo,$shaCommit,$moduleName,$className,$nameMe
thod >> $fileOutput
cd $BASEDIR
longPath=""
length=-1
maxLength=-1
ris=""
done < "$path"

```

Questo script permette di scaricare la repo da Github e di allinearsi alla commit di nostro interesse. Successivamente suddivide in due colonne la classe e il metodo da testare e per ogni classe va a individuare il sotto modulo più vicino a essa e lo va a indicare in un'altra colonna. È importante individuare il sotto modulo più vicino perché servirà nei prossimi step di esecuzione. Una volta terminata l'esecuzione dello script il dataset si presenta con le seguenti informazioni:

- Nome progetto;
- Link al progetto;
- *Sha* della commit;
- Path del sottomodulo;
- Nome della classe;
- Nome del metodo.

Nella seguente tabella è riportato un piccolo esempio di questo nuovo dataset.

Nome Repo	Link Repo	Sha commit	Nome modulo	Nome Classe	Nome metodo
dropwizard	<a href="https://github.com/dropwizard/dropwizard">https://github.com/dropwizard/dropwizard</a>	c4b3ad27349c6a26dc3e0d3c89a119617af3f7bb	./dropwizard-core/	DefaultServerFactoryTest	testGracefulShutdown
dropwizard	<a href="https://github.com/dropwizard/dropwizard">https://github.com/dropwizard/dropwizard</a>	609d55f66b8a2b54cff578577a4572695ab29480	./dropwizard-core/	DefaultServerFactoryTest	testGracefulShutdown
dropwizard	<a href="https://github.com/dropwizard/dropwizard">https://github.com/dropwizard/dropwizard</a>	25fdd060ca365020884e28fa059110e07f23a7f7	./dropwizard-core/	DefaultServerFactoryTest	testGracefulShutdown
dropwizard	<a href="https://github.com/dropwizard/dropwizard">https://github.com/dropwizard/dropwizard</a>	71ca2d4b0b3e9e69c736450061ca917c6bb20085	./dropwizard-core/	DefaultServerFactoryTest	testGracefulShutdown
dropwizard	<a href="https://github.com/dropwizard/dropwizard">https://github.com/dropwizard/dropwizard</a>	8270d51653c676e577288c1dc7598f023234b16a	./dropwizard-core/	DefaultServerFactoryTest	testGracefulShutdown

### 3.3.2.2 Installazione delle dipendenze con Maven

Il secondo step per la costruzione del dataset prevede l'installazione delle dipendenze di **Maven**; si è deciso di eseguire questa operazione in uno step separato per poter tenere traccia del successo o del fallimento della build e poter comparare questo risultato con quello che si otterrà dalla ricerca dei *flaky* (per esempio, se un caso di test fallisce sempre e si osserva che anche la build del sistema è fallita, allora probabilmente c'è una correlazione tra questi due dati). Anche per questa operazione è stato creato uno script bash, riportato di seguito

```

#!/bin/bash

checkout() {
    local sha=$SHACOMMIT
    echo "lo sha è: $sha"
    git checkout $sha
}

cloneRepo() {
    echo "clone repo " $LINKREPO
    local nameRepo=$NAMEREPO
    local linkRepo=$LINKREPO
    local shaCommit=$SHACOMMIT
    local baseDir=$BASEDIR
    local repoFolder=$REPOFOLDER
    cd $baseDir
    cd $repoFolder
    message=$(git clone $linkRepo 2>&1)
    TOSEARCH="fatal: destination path"
    toReturn=0
    cd $nameRepo
    if echo "$message" | grep -q "$TOSEARCH"; then #la repo già
        esiste
        actualSha=$(git rev-parse HEAD)
        if echo "$actualSha" | grep -q "$shaCommit"; then # lo
            sha è lo stesso di quello actual
            echo "project \"$nameRepo\" already exist with sha:
            \"$shaCommit"
            cd $baseDir
            echo $PWD
            toReturn=0
        else
            echo "cd \"$nameRepo"
            echo "nameRepo \"$nameRepo"
            echo "git checkout \" $shaCommit"
            git checkout $shaCommit
            toReturn=1
        fi
    else
        echo "cd \"$nameRepo"

```

```

        echo "nameRepo"$nameRepo
        echo "git checkout " $shaCommit
        git checkout $shaCommit
        toReturn=1
    fi
}

mvnStep() {
    local baseDir=$BASEDIR
    TOSEARCH="BUILD FAILURE"
    OUTPUTBUILD=""
    echo " _____ RUN mvn install -"
    DskipTests_____
    message=$(mvn install -DskipTests -fn -B)
    if echo "$message" | grep -q "$TOSEARCH"; then
        OUTPUTBUILD="BUILD FAILED"
    else
        OUTPUTBUILD="BUILD PASS"
    fi
    echo " _____ END mvn install -"
    DskipTests_____
    cd $baseDir
}

CSVINPUTFIRSTSTEP=$1
REPOFOLDER=$2
BASEDIR=$3
CSVOUTPUTFIRSTSTEP=$4
while IFS= read -r line
do
    IFS=',' read -r -a array <<< "$line"
    NAMEREPO=${array[0]}
    LINKREPO=${array[1]}
    SHACOMMIT=${array[2]}
    MODULENAME=${array[3]}
    cloneRepo "$NAMEREPO" "$LINKREPO" "$SHACOMMIT"
    "$BASEDIR" "$REPOFOLDER"
    echo $toReturn
    if [ $toReturn -eq 1 ]; then
        mvnStep $BASEDIR
    fi
done

```

```

echo
"$NAMEREPO", "$LINKREPO", "$SHACOMMIT", "$OUTPUTBUILD" >>
$CSVOUTPUTFIRSTSTEP
fi

done < "$CSVINPUTFIRSTSTEP"

```

Importante sottolineare in questo script la funzione *cloneRepo()*, infatti questa funzione effettua un controllo sulla commit a cui è allineato il repository in esame. Se il repository si trova già allineato alla commit individuata nello step precedente, allora evita di eseguire il comando *mvn install -DskipTests*, altrimenti lo esegue perché non sono ancora state installate le dipendenze. Grazie a questa semplice funzione è stato possibile ridurre i tempi di computazione, poiché l'installazione delle dipendenze avviene solo quando il repository non si trova già sul branch di nostro interesse. Nel dataset creato con questo script verranno riportate le informazioni sul nome e sul link del repository e informazioni sul fallimento o successo della build. In Figura 14 è riportato un breve esempio del dataset ottenuto da questo script.

	column 1	column 2	column 3	column 4
1	jimfs	https://github.com/google/jimfs.git	2b304c6c9accc8674156be1063efc46787f4d6	BUILD PASS
2	jimfs	https://github.com/google/jimfs.git	80a99d31a4d65ebd583c40255474bf481ca0ff6	BUILD PASS
3	jimfs	https://github.com/google/jimfs.git	3abd95a268ae109c9a13237703060911d2f0fb37	BUILD FAILED
4	jimfs	https://github.com/google/jimfs.git	2b16ca2f23b55c6c2c25bd0ff6d23e8a74ba56ce	BUILD PASS
5	jimfs	https://github.com/google/jimfs.git	2b304c6c9b9accc8674156be1063efc46787f4d6	BUILD PASS
6	jimfs	https://github.com/google/jimfs.git	80a99d31a4d65ebd583c40255474bf481ca0ff6	BUILD PASS
7	jimfs	https://github.com/google/jimfs.git	3abd95a268ae109c9a13237703060911d2f0fb37	BUILD FAILED
8	jimfs	https://github.com/google/jimfs.git	2b16ca2f23b55c6c2c25bd0ff6d23e8a74ba56ce	BUILD PASS
9	oozie	https://github.com/apache/oozie.git	5eec5564654e1bbe6240c3050722b45977392e	BUILD FAILED
10	dropwizard	https://github.com/dropwizard/dropwizard	c0b3ad27349c6a26dc3e0d3c89a119617ef3f7bb	BUILD PASS
11	dropwizard	https://github.com/dropwizard/dropwizard	609d55f66b8a2b54cff578577a4572695ab29480	BUILD PASS
12	dropwizard	https://github.com/dropwizard/dropwizard	25fd060ca365020884e28fa059110e0f723a77	BUILD PASS
13	dropwizard	https://github.com/dropwizard/dropwizard	71ca2d4bb0b9e9c736450061ca917c6bb20085	BUILD PASS
14	dropwizard	https://github.com/dropwizard/dropwizard	8270d51653c676e577288c1d7598f023234b16a	BUILD PASS
15	dropwizard	https://github.com/dropwizard/dropwizard	84a2054cd6ea2d67233a9a77f83cb4b2f17795e	BUILD PASS
16	dropwizard	https://github.com/dropwizard/dropwizard	c0f22a0d78b23e0dedfc8bc763f19b701471a94e	BUILD FAILED
17	dropwizard	https://github.com/dropwizard/dropwizard	8f57539fc12e433fe981d23b86e3b99952a6f	BUILD FAILED
18	dropwizard	https://github.com/dropwizard/dropwizard	6bc4620ffcc881063c6441dad40311ec1d9b9a74	BUILD FAILED
19	dropwizard	https://github.com/dropwizard/dropwizard	a57d107cfra2ca37b2e8eab525719f3c696416308	BUILD FAILED
20	dropwizard	https://github.com/dropwizard/dropwizard	21af19b0c471652656b832f38395e5419b21	BUILD FAILED
21	dropwizard	https://github.com/dropwizard/dropwizard	d4f9ee341abf0ddce40c24642c6a93be1f45f9	BUILD FAILED
22	dropwizard	https://github.com/dropwizard/dropwizard	0767a11daea9393bb9d7cd8329777b6ca55d8c9	BUILD FAILED
23	dropwizard	https://github.com/dropwizard/dropwizard	06e9a425e82a311ec84b7cc7162262c/ebad67f8	BUILD PASS
24	hector	https://github.com/hector-client/hector.git	cf0a4c4334efbcff410cea2a595e20ad0b11a5	BUILD PASS
25	hector	https://github.com/hector-client/hector.git	54cc4ea3f6dbc8846795c815f4ca683a3f1daeb	BUILD PASS
26	hector	https://github.com/hector-client/hector.git	ca77f998b72fcd139228b4d7a52d6ab485b630b	BUILD PASS
27	hector	https://github.com/hector-client/hector.git	6ac711f34034bccb774547f55f9f37f5157160ea	BUILD PASS
28	hector	https://github.com/hector-client/hector.git	c6fd613fa896a6f61cb2332dacf9491316fdb	BUILD PASS
29	hector	https://github.com/hector-client/hector.git	5e6705207d35f28bcce9a2e9c4e6426eb684e9aa1	BUILD PASS
30	hector	https://github.com/hector-client/hector.git	7a2f60b1c04149673f6881db9e5e21bf8cafc	BUILD PASS
31	hector	https://github.com/hector-client/hector.git	66c68a67fb7b47516424b757fee2e2020ba10e3f	BUILD PASS
32	hector	https://github.com/hector-client/hector.git	e9a0ef419ee61f41b87413224d574f29769b7e19	BUILD PASS
33	hector	https://github.com/hector-client/hector.git	2cb17ecf3a35dadb675b0bde75a2c40a4e7483	BUILD PASS
34	hector	https://github.com/hector-client/hector.git	0d805b1676ce85b3b166f125bb26b9d48640f74	BUILD PASS
35	hector	https://github.com/hector-client/hector.git	3b582672f3d4f43b648c413de983873a982edf	BUILD PASS

**Figura 14: Risultati della build**

### 3.3.2.3 Ricerca del flaky test

Nel terzo step della costruzione del dataset avviene la ricerca vera e propria del *flaky test*. Anche in questo caso è stato sviluppato uno script bash con il quale avvengono mille esecuzioni di ogni caso di test e si va a verificare il suo comportamento **non deterministico**. Di seguito, è riportato il codice di questo terzo step.

```
#!/bin/bash

checkRepo () {
    local nameRepo=$NAMEREPO
    local linkRepo=$LINKREPO
    local shaCommit=$SHACOMMIT
    local baseDir=$BASEDIR
    local repoFolder=$REPOFOLDER
    cd $baseDir
    cd $repoFolder
    cd $nameRepo

    actualSha=$(git rev-parse HEAD)
    echo "actual sha:" $actualSha
```

```

        echo "SHA COMMIT:" $shaCommit
        if echo "$actualSha" | grep -q "$shaCommit"; then # lo
sha è lo stesso di quello actual
            echo "project \"$nameRepo\" already exsist with sha:
$shaCommit"
            cd $baseDir
            toReturn=0
        else
            echo "cd \"$nameRepo"
            echo "$nameRepo$nameRepo"
            echo "git checkout \" $shaCommit"
            git checkout $shaCommit
            toReturn=1
        fi
    }

mvnStep() {
    local baseDir=$BASEDIR
    TOSEARCH="BUILD FAILURE"
    OUTPUTBUILD=""
    echo "_____RUN mvn install -
DskipTests_____"
    message=$(mvn install -DskipTests -fn -B)
    if echo "$message" | grep -q "$TOSEARCH"; then
        OUTPUTBUILD="BUILD FAILED"
    else
        OUTPUTBUILD="BUILD PASS"
    fi
    echo "_____END mvn install -
DskipTests_____"
    cd $baseDir
}

searchFlaky() {
echo "-----SEARCH
FLAKY-----"
local concatName=$CLASSNAME#"${METHODNAME}
local nrounds=$NUMBERSROUNDS
local nameRepo=$NAMEREPO
local moduleName=$MODULENAME
local baseDir=$BASEDIR

```

```

local RepoFolder=$REPOFOLDER
local csvOutput=$CSVOUTPUT
local resultTest=""
local shaCommit=$SHACOMMIT
local linkRepo=$LINKREPO
local outputLog=$OUTPUTLOG
local stateLog=$STATELOG
PASSTEST=0
FAILTEST=0
TOSEARCH="BUILD SUCCESS"
i=0
for i in $( seq 1 $nrounds )
do
    cd $baseDir
    cd $RepoFolder
    cd $nameRepo
    cd $moduleName
    timestampInitial=$(date +%s)
    timestampInitialDate=$(date -d @$timestampInitial)
    echo "exec command: mvn -Dtest=\"$concatName\" test" $i
    stateMachineInitial=$(vmstat -t)

dirLog="$baseDir""$outputLog""$CLASSNAME"_"$METHODNAME"_"$SHACOMMIT".txt"
    message=$(mvn -Dtest=$concatName test)
    echo "$message" >> "$dirLog"
    stateMachineFinal=$(vmstat -t)
    echo "$stateMachineInitial", "$stateMachineFinal" >>
"$baseDir""$stateLog""$CLASSNAME"_"$METHODNAME"_"$SHACOMMIT".txt"
    timestampFinal=$(date +%s)
    timestampFinalDate=$(date -d @$timestampFinal)
    if echo "$message" | grep -q "$TOSEARCH"; then
        echo "test pass";
        resultTest="test pass"
        PASSTEST=$((PASSTEST+1))
    else
        resultTest="test fail"
        echo "test fail"
        FAILTEST=$((FAILTEST+1))
    fi

```

```

        echo "$nameRepo","$linkRepo", "$shaCommit","$moduleName",
"$CLASSNAME", "$METHODNAME", "$resultTest",
$timestampInitialDate,$timestampFinalDate >>
"$baseDir""$csvOutput""$CLASSNAME""_""$METHODNAME""_""$SHACOM
MIT"".csv"
done
echo "number of pass: "$PASSTEST
echo "number of failure: "$FAILTEST
if [ $PASSTEST -gt 0 -a $FAILTEST -gt 0 ]; then
    echo "test flaky"
    echo "numbers of runs:"$i
else
    echo "test isn't flaky"
fi
}

CSVINPUTFIRSTSTEP=$1
CSVOUTPUT=$2
BASEDIR=$3
REPOFOLDER=$4
NITER=$5
NUMBERSROUNDS=$6
CSVOUTPUTFINAL=$7
OUTPUTLOG=$8
STATELOG=$9

while IFS= read -r line
do
    IFS=',' read -r -a array <<< "$line"
    NAMEREPO=${array[0]}
    LINKREPO=${array[1]}
    SHACOMMIT=${array[2]}
    MODULENAME=${array[3]}
    CLASSNAME=${array[4]}
    METHODNAME=${array[5]}
    checkRepo "$NAMEREPO" "$LINKREPO" "$SHACOMMIT"
    "$BASEDIR" "$REPOFOLDER"
    echo $toReturn
    if [ $toReturn -eq 1 ]; then
        mvnStep $BASEDIR
    fi

```

```

searchFlaky $NAMEREPO $CLASSNAME $METHODNAME
$NUMBERSROUNDS $BASEDIR $REPOFOLDER $CSVOUTPUT $SHACOMMIT
$LINKREPO $OUTPUTLOG $STATELOG
cd $BASEDIR
echo
$LINKREPO,$SHACOMMIT,$MODULENAME,$CLASSNAME,$METHODNAME,$NUMB
ERSROUNDS,$PASSTEST,$FAILTEST >> "$CSVOUTPUTFINAL"".csv"
done < "$CSVINPUTFIRSTSTEP"

```

È importante sottolineare che anche in questo step si dovrà eseguire nuovamente il download delle dipendenze *con mvn install -DskipTests*, questo perché quando avviene un cambio di branch all'interno di un repository, si perdonano le tracce di eventuali modifiche fatte su quel branch.

Per quanto riguarda la ricerca dei *flaky test*, analizziamo nel dettaglio la funzione *searchFlaky()*: questa funzione ha come obiettivo la ricerca di un *flaky test* durante l'esecuzione di un singolo caso di test. Ci sono diversi step affrontati da *searchFlaky()*:

- Per ogni caso di test viene eseguito il comando *mvn -Dtest=nomeClasse#nomeMetodo test*. Il risultato ottenuto durante l'esecuzione viene salvato in un file csv in cui viene riportato anche il timestamp di inizio esecuzione e il timestamp di fine esecuzione; (Figura 15)

	Sha	ModelName	ClassName	MethodName	Result	FinalDate
1	NameRepos_LinDepo	CassandraClusterTest	testDescribeReplicates	tests pass	mei 11 mar 2020-21 7:19	mei 11 mar 2020-21 47:23
2	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:23
3	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:23
4	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:26
5	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:30
6	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:33
7	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:37
8	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:40
9	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:44
10	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:48
11	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:51
12	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:55
13	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 47:58
14	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:02
15	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:06
16	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:10
17	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:14
18	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:18
19	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:22
20	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:26
21	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:31
22	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:35
23	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:38
24	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:42
25	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:46
26	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:50
27	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:53
28	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 48:57
29	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:01
30	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:04
31	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:08
32	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:11
33	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:15
34	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:19
35	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:23
36	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:26
37	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:30
38	hector	https://github.com/hector-client/hector	get	test	pass	mei 11 mar 2020-21 49:34

**Figura 15:** Dataset ottenuto dalle mille esecuzioni di un caso di test

- Subito prima e subito dopo aver eseguito il test si lancia il comando `vmstat -t`. Questo comando permette di collezionare informazioni sul sistema operativo, sui processi, sulla memoria, ecc.;
- Per tutte le esecuzioni del caso di test viene salvato un file di log in cui sono riportate informazioni sul risultato di quella esecuzione, eventuali errori o eccezioni del codice;
- Una volta terminate le diverse iterazioni dello stesso caso di test, verrà creata una riga nel dataset finale in cui sarà riportato un sunto di tutte le esecuzioni. Il dataset finale conterrà le seguenti informazioni (Figura 16):
  - Nome Repo;
  - Link Repo,
  - *Sha commit*;
  - Path del sottomodulo;
  - Nome Classe;
  - Nome Metodo;
  - Numero totale di esecuzioni;
  - Numero di successi del test;
  - Numero di fallimenti del test.

**Figura 16:** Dataset finale

Per cercare di automatizzare e rendere ripetibile il processo di ricerca dei *flaky* è stato anche aggiunto un file di configurazione in cui sono riportate delle informazioni utili, come i nomi dei vari dataset che verranno creati alla fine di ogni step, i nomi delle cartelle in cui verranno salvati i dati (cartella dei repository, cartella dei file di log, ecc.) e il numero di iterazioni che sono state effettuate per ogni caso di test (Figura 17). Inoltre, grazie a questo file di *config*, non sarà necessario modificare le informazioni all'interno del codice per rendere funzionante il codice anche su altre macchine.



```

configStep
home > vale > Scaricati > tesiMagistraleSesa > configStep
1 CSVINPUTFIRSTSTEP=outIDFlakies.csv
2 CSVOUTPUT=outputStep/
3 CSVOUTPUTFINAL=outputFinal/csvFinal
4 CSVOUTPUTFIRSTSTEP=outFirstStep.csv #risultato della build numero inferiore di righe perché la clone viene fatta 1 sola volta
5 REPOFOLDER=Repos/
6 NROUND=1000
7 BASEDIR=/home/sesa/Scrivania/tesiMagistrale/
8 OUTPUTLOG=LOG/
9 STATELOG=STATELOG/

```

**Figura 17:** File di configurazione

### 3.3.4 Risultati ottenuti

Come detto precedentemente, l'esecuzione di questi script è avvenuta su un dataset di centocinquanta cinque righe; ogni caso di test presente nel dataset è stato eseguito mille volte.

I tre step descritti precedentemente sono stati tutti eseguiti in maniera indipendente tra di loro, in modo da poter controllare i risultati ottenuti in uno script prima di passare all'esecuzione dello script successivo.

#### 3.3.4.1 Primo Step

Durante l'esecuzione del primo script, si è passati da centocinquantaquattro righe a centoquarantuno. Infatti, durante questa fase c'è stata una perdita di informazioni perché lo *sha* presente nel dataset originale non era più valido, quindi non è stato possibile risalire al path del sotto modulo più vicino al caso di test in analisi. In Figura 18 sono riportati i dati delle righe che non sono state considerate.

<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	4571346261c33d6110639879902db81db75c020	controllers.ApiControllerDocTest.testGetAndPostArticleViaJson
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	4571346261c33d6110639879902db81db75c020	controllers.ApiControllerDocTesterTest.testGetAndPostArticleViaJson
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	example.ExampleIntegrationTest.testThatInvalidStaticAssetsAreNotFound
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	example.ExampleIntegrationTest.testThatStaticAssetsWork
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	example.ExampleApiTest.testThatInjectorAccessibleFromNinjaTestIsTheApplicationInjector
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	example.ExampleApiTest.testThatStaticAssetsWork
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	conf.RoutesTest.testReverseRoutingWithArrayAndWrongAmountOfQueryParameters
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	conf.RoutesTest.testReverseRoutingWithMap
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	conf.RoutesTest.testReverseRoutingWithoutMap
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	conf.RoutesTest.testRoutes
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	conf.RoutesTest.testTestRoutesAreHiddenFromProduction
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	controllers.PersonControllerTest.testPostPersonJson
<a href="https://github.com/dvla/oepp-customer-portal.git">https://github.com/dvla/oepp-customer-portal.git</a>	28d0b5f692c1e73e62ff9556d2dea1e4560d6bc7	controllers.PersonControllerTest.testPostPersonXml

**Figura 18:** Righe non considerate

### 3.3.4.2 Secondo Step

Una volta che il dataset è stato “pulito”, si è passati alla fase di building dei progetti, in modo da tenere traccia del successo o fallimento della build ed eventualmente poter fare qualche comparazione con i risultati che si otterranno nello step successivo. Di seguito è riportata una tabella che riassume il numero di successi e di fallimenti della build.

Numero Progetti	Risultato della build
36	BUILD FAILED
38	BUILD PASS

È possibile notare che la build non è stata eseguita cento uarantuno volte ma settantaquattro volte perché nel dataset sono presenti più righe che fanno riferimento allo stesso *sha* per un progetto.

### 3.3.4.3 Terzo step

Una volta eseguita l’installazione delle dipendenze di **Maven**, si è passati alla ricerca dei *flaky*. Non tutti i casi di test riportati nel dataset hanno manifestato un comportamento *flaky*, ma questo non ha stupito in quanto il *flaky* presenta un comportamento **non deterministico**. Ogni caso di test è stato eseguito mille e sono stati individuati dodici casi di *flaky test*. In Figura 19 è riportato una parte del dataset finale.

	column 1	column 2	column 3	column 4	column 5	column 6	column 7
1	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentDeletePushMessageDemand	testUncaughtCorrdTeleBasic	1000 0	1000
2	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentRejectionXCommand	testConcurrentActionTimeout	1000 0	1000
3	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentXCommand	testConcurrentJobGet	1000 0	1000
4	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentXCommand	testConcurrentFailedAction	1000 0	1000
5	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentXCommand	testConcurrentFavorSupport	1000 0	1000
6	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentXCommand	testConcurrentRemovePushMissingDeos	1000 0	1000
7	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentSuccess1	testConcurrentSuccess1	1000 0	1000
8	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentSuccess2	testConcurrentWarning	1000 0	1000
9	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentCommand	testConcurrentCommandQueue	1000 0	1000
10	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarker	1000 0	1000
11	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerConflict	1000 0	1000
12	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerConflictInConcurrentR	1000 0	1000
13	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerConflictInConcurrentR	1000 0	1000
14	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerConflictInConcurrentR	1000 0	1000
15	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerConflictInConcurrentR	1000 0	1000
16	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerWithPauseTime1	1000 0	1000
17	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentTransitionXCommand	testActionMarkerWithPauseTime2	1000 0	1000
18	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentCommand	testConcurrentWithErrorResume	1000 0	1000
19	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentStartCommand	testActionStarWithErrorReported	1000 0	1000
20	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentStartXCommand	testActionStarWithEscapedString	1000 0	1000
21	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentStopCommand	testConcurrentNegative	1000 0	1000
22	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentStopCommand	testConcurrentPositive	1000 0	1000
23	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestConcurrentStopXCommand	testConcurrentWithErrorPositive	1000 0	1000
24	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestFutureCommand	testEngine	1000 0	1000
25	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		TestFutureCommand	testGracefulShutdown	1000 0	568
26	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	568
27	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	432
28	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	432
29	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	511
30	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 573	427
31	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 525	475
32	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 539	461
33	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
34	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
35	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
36	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
37	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
38	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000
39	https://github.com/autche/moveoil	Seu554654a1be664bf0c05072ba450f73926 /core/		DefaultServerFactoryTest	testGracefulShutdown	1000 0	1000

**Figura 19:** Parte del dataset finale

È possibile notare che per il primo repository **Oozie** i casi di test sono sempre falliti. Si è deciso quindi di analizzare il risultato della build ottenuto precedentemente e si è notato che anche qui il risultato era FAILED. Ragionamento simile si è potuto fare anche con un altro repository, **Achilles**, anche in questo caso infatti tutti i casi di test eseguiti su questo repository sono falliti e anche la build iniziale ha ottenuto come risultato FAILED. Si è quindi deciso di tentare un approccio alternativo per scaricare le dipendenze usando il comando *mvn dependency:copy-dependencies*, ma nemmeno in questo modo si è riusciti a scaricare le dipendenze del POM.

Successivamente si è deciso di effettuare una nuova esecuzione di alcuni casi di test (scelti in maniera casuale) per verificare se ci fosse un qualche cambiamento dei risultati; in Figura 20 e 21 sono riportati gli output ottenuti nelle due esecuzioni di 24 repository.

linkRepo	sha	moduleName	methodName	ApplicationControl	#Iteration	#pass	#fail
<a href="https://github.com/mjframework/kafka.git">https://github.com/mjframework/kafka.git</a>	db4d4e027cd22910c26a45c87377	/mjframework/resources/archetype-resources/	testCheckTheKeySpaces	testHomePageWorks	1000	0	1000
<a href="https://github.com/hector-client/hector.git">https://github.com/hector-client/hector.git</a>	7e311021ec417f4da667c3ecc4	/core/	CassandraCluster	testCheckTheKeySpaces	1000	1000	0
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	b91130d77621a3b9fc150ba4	/wrold-maven-plugin/src/test/java/	TestWroldMojito	shouldBeAtWhenRunning	1000	0	1000
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	5b6522828d9e024b4db5a4008	/wrold-maven-plugin/src/test/java/	ExceptionTest	checkInExecution	1000	1000	0
<a href="https://github.com/1154893257/hektor.git">https://github.com/1154893257/hektor.git</a>	40c2c66ae0c01394e9b5b58560c	/src/test/java/	CookieTest	testHeadersSentToCookieJar	1000	1000	0
<a href="https://github.com/1164854257/hektor.git">https://github.com/1164854257/hektor.git</a>	d9b56e52a166e3429a61fb1b13	/src/test/java/	Cookies	testCookieFromStore	1000	1000	0
<a href="https://github.com/apache/amban/amban.git">https://github.com/apache/amban/amban.git</a>	652b2ea1455ae1d4631a38d8a4	/amban/	TestActionQueue	testConcurrentOperations	1000	0	1000
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	a7c03b6d9b3e3beach	/wrold-maven-plugin/src/test/	CassandraCluster	testCheckTheKeySpaces	1000	1000	0
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	443a76e870dc155b5663910f5fa	/wrold-maven-plugin/src/test/	TestWroldMojito	shouldBeAtWhenRunning	1000	0	1000
<a href="https://github.com/116451527909a6e6a100t/hektor.git">https://github.com/116451527909a6e6a100t/hektor.git</a>	03265b125504fe56960d250556	/src/test/java/	DefaultService	testGracefulShutdown	1000	0	1000
<a href="https://github.com/apache/amban/amban.git">https://github.com/apache/amban/amban.git</a>	889bc76451527909a6e6a100t	/amban/	URLConnectionTest	testGetWithCrunkedRequest	1000	1000	0
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	9638cf00d045b4a5b25705a9f3	/src/test/java/	ResponseCache	maxAgeInTheFutureWindow	1000	1000	0
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	4aa97553a5c80ef0252705a9f3	/amban/	TestActionQueue	testConcurrentOperations	1000	0	1000
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	04c08778530c083a16ee363e1c4	/wrold-maven-plugin/src/test/	TestWroldMojito	shouldBeAtWhenRunning	1000	0	1000
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	f8736ea0d7959bade1deaf	/src/test/java/	States�s	testStatesTest	1000	863	137
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	07e293344f93c1b6ef90a21d1f7c	/src/test/java/	TestTimeRegionsS	should_trigger_for_insert	1000	0	1000
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	40c2c66ae0c901394e9b5b58560c	/src/test/java/	TestTimeRegionsS	should_trigger_for_update	1000	0	1000
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_delete_by_id	1000	0	1000
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_delete_child	1000	0	1000
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_insert1	1000	0	n

Figura 20: Prima esecuzione

linkRepo	sha	moduleName	methodName	methodName	methodName	#Iteration	#pass	#fail
<a href="https://github.com/hector-client/hector.git">https://github.com/hector-client/hector.git</a>	7e311021ec417f4da667c3ecc4	/core/	CassandraCluster	test	testCheckTheKeySpaces	1000	1000	0
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	b91130d77621a3b9fc150ba4	/src/test/java/	Exception	test	checkInExecution	1000	1000	0
<a href="https://github.com/1154893257/hektor.git">https://github.com/1154893257/hektor.git</a>	d9b56e52a166e3429a61fb1b13	/src/test/java/	Cookie	test	testHeadersSentToCookieJar	1000	1000	0
<a href="https://github.com/apache/amban/amban.git">https://github.com/apache/amban/amban.git</a>	652b2ea1455ae1d4631a38d8a4	/amban/	TestActionQueue	test	testConcurrentOperations	1000	1000	0
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	443a76e870dc155b5663910f5fa	/src/test/java/	TestWroldMojito	should	beAtWhenRunning	1000	1000	0
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	03265b125504fe56960d250556	/src/test/java/	States	test	testGracefulShutdown	1000	1000	0
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	9638cf00d045b4a5b25705a9f3	/amban/	TestAllEntitys	should	delete	1000	1000	0
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	4aa97553a5c80ef0252705a9f3	/amban/	TestAllEntitys	should	insert1	1000	0	1000
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	04c08778530c083a16ee363e1c4	/wrold-maven-plugin/src/test/	TestWroldMojito	should	beAtWhenRunning	1000	1000	0
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	f8736ea0d7959bade1deaf	/src/test/java/	States	test	testStatesTest	1000	863	137
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	07e293344f93c1b6ef90a21d1f7c	/src/test/java/	TestTimeRegionsS	should_trigger_for_insert	1000	0	1000	
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	40c2c66ae0c901394e9b5b58560c	/src/test/java/	TestTimeRegionsS	should_trigger_for_update	1000	0	1000	
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_delete_by_id	1000	0	1000	
<a href="https://github.com/conduit/conduit-orbit.git">https://github.com/conduit/conduit-orbit.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_update_child_value	1000	0	1000	
<a href="https://github.com/wrold/wrold.git">https://github.com/wrold/wrold.git</a>	3e7c5139edbd18a506253251	/src/test/java/	TestAllEntitys	should_insert	1000	0	n	

Figura 21: Seconda esecuzione

Dopo aver visionato i risultati ottenuti dalle due esecuzioni, si sono potute fare le seguenti considerazioni:

- I casi di test che non hanno manifestato un comportamento *flaky* alla prima esecuzione, non lo hanno mostrato nemmeno nella seconda esecuzione. Questo può dipendere da diversi fattori, come il **non determinismo** del *flaky test*, la *root cause* del flaky (ricordiamo che questo lavoro si è concentrato solo su *flaky test* indipendenti dall'ordine, quindi se la *root cause* è la dipendenza dall'ordine il *flaky* non si manifesterà) oppure dall'output ottenuto nella fase precedente, ossia l'installazione delle dipendenze di **Maven**;
- I casi di test che hanno manifestato un comportamento *flaky* nella prima esecuzione, lo hanno mostrato anche nella seconda esecuzione ottenendo più o meno gli stessi numeri di pass e fail. Si potrebbe quindi pensare che alcuni *flaky test* scatenati da determinate *root cause* possano assumere un comportamento **deterministico**.

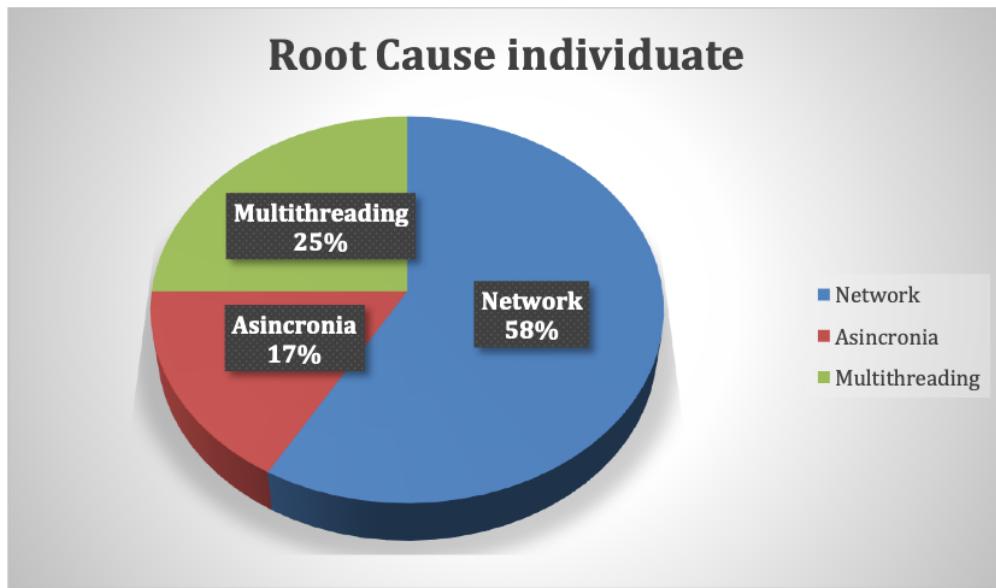
Infine, una volta terminata la verifica del *flaky test* si è passati a una classificazione delle *root cause* per i flaky individuati. Questa analisi è stata effettuata in due step:

1. Analisi del file di log generato durante le mille iterazioni del caso di test per verificare l'eccezione lanciata in caso di errore;
2. Analisi statica<sup>24</sup> del codice del caso di test per avere ulteriori chiarimenti della *root cause* del *flaky*.

Solo per un metodo è stato necessario effettuare l'analisi statica del codice perché l'eccezione che lanciava non forniva informazioni utili sulla *root cause*. Le *root cause* individuate sono tre: **Network, Asincronia e Multithreading**. In Figura 22 è riportato un grafico che mostra la percentuale di riscontro di ogni *root cause*.

---

<sup>24</sup> L'analisi statica è l'analisi del codice sorgente che viene effettuata tramite la lettura del codice senza la sua esecuzione.



*Figura 22: Grafico delle root cause individuate*

### 3.3.5 Limiti della ricerca

Per quanto questi risultati facciano ben sperare per le ricerche future, bisogna anche sottolineare e tenere conto di tutti i limiti individuati fino a ora, ossia:

- Lentezza del sistema di build che ha portato a non considerare alcuni repository che avrebbero richieste ore per terminare questo step;
- Lentezza di esecuzione di alcuni casi di test, impiegando anche più di 24 ore per completare le mille iterazioni;
- Limiti degli strumenti hardware, infatti i computer a disposizione non avevano alte prestazioni (Ubuntu 16.04 4GB di RAM 2. Intel Core i3-2100 CPU 3.10 GHzx4), rallentando ulteriormente l'esecuzione della build e dei casi di test;
- I *flaky test* fino a ora individuati non sono molti, quindi non è possibile fare ulteriori analisi ma è indispensabile ampliare il dataset (anche con repository più “recenti”).

# CONCLUSIONI

Dalla revisione sistematica della letteratura è apparso chiaro che uno dei temi principali riguardo i *flaky test* sia l'individuazione delle *root cause* che li genera. Individuare le *root cause* infatti, potrebbe essere il primo passo per lo sviluppo di strategie “personalizzate” (sulle *root cause*) che permettano, almeno in parte, di mitigare questa problematica.

Per quanto riguarda i framework fino a ora sviluppati, questi presentano dei limiti legati al linguaggio di programmazione su cui si possono usare e sui sistemi di building che devono essere presenti nel sistema. Per questi motivi, spesso in ambito aziendale la soluzione maggiormente applicata risulta essere ancora l'esecuzione continua di un caso di test fino a quando non appare evidente il suo comportamento **non deterministico**. Questa però non risulta essere una soluzione valida, perché comporta una grande quantità di tempo “perso” e costi di sviluppo piuttosto elevati.

Infine, dalla revisione sistematica è emerso che un campo di ricerca poco conosciuto è l'individuazione dei *flaky test* sfruttando tecniche di Machine Learning. Infatti, in questo ambito per ora sono state proposte delle soluzioni che non sono ancora state messe in pratica.

Per quanto riguarda l'individuazione delle *root cause* sui *flaky test* individuati precedentemente da **DeFlaker**, sui centoquarantuno casi di test analizzati, sono stati individuati dodici *flaky test*, le cui *root cause* sono tre: **Asincronia, Network e Multithreading**.

Inoltre, le analisi condotte su una seconda esecuzione effettuata su ventiquattro casi di test hanno portato ad ipotizzare una sorta di determinismo anche per *flaky test* causati da specifiche *root cause*.

Questi risultati rappresentano un buon punto di inizio ma dimostrano anche la necessità di trovare delle strade alternative che richiedano meno tempo di quelle percorse fino a ora (lunghi tempi legati all'installazione delle dipendenze da parte di **Maven**).

Sviluppi futuri di questo lavoro saranno l'estensione dell'approccio usato che permetta l'individuazione di *flaky test*, la cui *root cause* sia la dipendenza dall'ordine di esecuzione, l'estensione del dataset con *flaky test* presenti in altri linguaggi di programmazione, una strumentazione del codice che permetta l'estensione di questo approccio a progetti che usano sistemi di build diversi da **Maven**, un'analisi più approfondita sulla possibilità che alcuni *flaky test* generati da determinate *root cause* abbiano un comportamento **deterministico** e l'implementazione di tecniche di Machine Learning per la ricerca e classificazione dei *flaky test*, con un confronto con quelle già esistenti in modo da poter individuare pregi e difetti di entrambe.

## INDICE DELLE FIGURE

<b>Figura 1:</b> Feature della build automatizzata.....	10
<b>Figura 2:</b> Processo di Continuos Integration.....	12
<b>Figura 3:</b> Flaky test causato dall'asincronia .....	15
<b>Figura 4:</b> Flaky test causato dalla concorrenza .....	16
<b>Figura 5:</b> Flaky test causato dalla dipendenza dall'ordine di esecuzione ...	16
<b>Figura 6:</b> Protocollo della revisione sistematica della letteratura .....	21
<b>Figura 7:</b> Sintesi ricerca degli studi primari.....	31
<b>Figura 8:</b> Diagramma riassuntivo RQ1.....	33
<b>Figura 9:</b> Linguaggi di programmazione riscontrati negli studi primari ....	37
<b>Figura 10:</b> Dimensioni dei campioni analizzati negli studi primari .....	38
<b>Figura 11:</b> Esempio di issue su Github.....	45
<b>Figura 12:</b> Esempio dataset di DeFlaker.....	47
<b>Figura 13:</b> Nuovo dataset di DeFlaker.....	51
<b>Figura 14:</b> Risultati della build.....	59
<b>Figura 15:</b> Dataset ottenuto dalle mille esecuzioni di un caso di test.....	64
<b>Figura 16:</b> Dataset finale.....	66
<b>Figura 17:</b> File di configurazione .....	67
<b>Figura 18:</b> Righe non considerate.....	68
<b>Figura 19:</b> Parte del dataset finale .....	69
<b>Figura 20:</b> Prima esecuzione .....	71
<b>Figura 21:</b> Seconda esecuzione .....	71
<b>Figura 22:</b> Grafico delle root cause individuate.....	73

## APPENDICE A

Di seguito sono riportati i riferimenti bibliografici degli articoli usati per la revisione sistematica della letteratura.

[S1] **Shali Y., Zhenyu C., Zhihong Z., Chen Z., Yuming Z.**, “A Dynamic Test Cluster Sampling Strategy by Leveraging Execution Spectra Information”, *Third International Conference on Software Testing, Verification and Validation, Paris, France, 2010.*

[S2] **Qingzhou L., Farah H., Lamyaa E., Darko M.**, “An Empirical Analysis of Flaky Tests”, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014; 643–653.*

[S3] **Jonathan B., Gail K., Eric M., Mohan D.**, “Efficient Dependency Detection for Safe Java Test Acceleration”, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015; 770–781.*

[S4] **Alex G., August S., Farah H., Darko M.**, “Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency”, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore MD, USA, 2015; 223–233.*

[S5] **Shi A., Gyori, A., Legunsen O., Marinov D.**, “Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications”, *Proceedings of the 2016 International Conference on Software Testing, Verification and Validation, USA, 80–90.*

[S6] **Palomba F., Zaidman A.**, “Does Refactoring of Test Smells Induce Fixing Flaky Tests?”, *International Conference on Software Maintenance and Evolution, Shanghai, China, 2017.*

[S7] **Adriaan L., Laura I., Reid H.**, “Measuring the Cost of Regression Resting in Practice: A Study of Java Projects Using Continuous Integration”,

*Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 2017; 821–830.*

[S8] **August S., Jonathan B., Darko M.**, “Mitigating the Effects of Flaky Tests on Mutation Testing”, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 2017; 112–122.*

[S9] **Thomas R., Waldemar H., Philipp L., Stefan S.**, “An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-based Open-source Software”, *Proceedings of the 14th International Conference on Mining Software Repositories, Buenos Aires, Argentina, 2017; 345–355.*

[S10] **Tariq M. K., Dionny S., Justin P., Peter J. C.**, “Towards a Bayesian Network Model for Predicting Flaky Automated Tests”, *International Conference on Software Quality, Reliability and Security Companion, Lisbon, Portugal, 2018;*

[S11] **Swapna T., Chandani S., Na M.**, “An Empirical Study of Flaky Tests in Android”, *International Conference on Software Maintenance and Evolution, Madrid, Spain, 2018.*

[S12] **Alessio G., Jonathan B., Andreas Z.**, “Practical Test Dependency Detection”, *11th International Conference on Software Testing, Verification and Validation, Västerås, Sweden, 2018.*

[S13] **Md T. R., Peter C. R.**, “The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds”, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista FL, USA, 2018; 857–862.*

[S14] **Jonathan B., Owolabi L., Michael H., Lamyaa E., Tifany Y., Darko M.**, “DeFlaker: Automatically Detecting Flaky Tests”, *Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 2018; 433–444.*

[S15] **Marc R.**, “Using Machine Learning to Classify Test Outcomes”, *International Conference on Artificial Intelligence Testing, San Francisco East Bay, California, USA, 2019*.

[S16] **Kai Presler-M., Eric H., Sarah H., Kathryn S.**, “Wait, Wait. No, Tell Me. Analysing Selenium Configuration Effects on Test Flakiness”, *14th International Workshop on Automation of Software Test, Montreal, QC, Canada, 2019*.

[S17] **Zhiyu F.**, “A Systematic Evaluation of Problematic Tests Generated by EvoSuite”, *41st International Conference on Software Engineering: Companion Proceedings, Montreal, QC, Canada, 2019*.

[S18] **Wing L., Reed O., August S., Darko M., Tao X.**, “iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests”, *12th IEEE Conference on Software Testing, Validation and Verification, Xi'an, China, 2019*.

[S19] **Claire L., Abhayendra S., Mike P., Yves Le T., John M.**, “Assessing Transition-Based Test Selection Algorithms at Google”, *41st International Conference on Software Engineering: Software Engineering in Practice, Montreal, QC, Canada, 2019*.

[S20] **Eck M., Palomba F., Castelluccio M., Bacchelli A.**, “Understanding Flaky Tests: The Developer’s Perspective”, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019; 830–840*.

[S21] **August S., Wing L., Reed O., Tao X., Darko M.**, “iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests”, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019; 545–555*.

[S22] **Armin N., Peter C. R., Weiyi S.**, “Bisecting Commits and Modelling Commit Risk During Testing”, *Proceedings of the 2019 27th ACM Joint Meeting*

*on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019; 279–289.*

[S23] **Wing L., Patrice G., Suman N., Anirudh S., Suresh T.**, “Root Causing Flaky Tests in a Large-scale Industrial Setting”, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 2019; 101–111.*

[S24] **Andreas S., Marvin K., Gordon F.**, “Testing Scratch Programs Automatically”, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019; 165–175.*

[S25] **Palomba F., Zaidman A.**, “The Smell of Fear: On the Relation Between Test Smells and Flaky Tests”, *Empirical Software Engineering, Salerno, Italy, 2019; 2907–2946.*

## BIBLIOGRAFIA

- [1] **Ali A., Gravino C.** (2019), “A systematic literature review of software effort prediction using machine learning methods”, *Journal of Software: Evolution and Process, Vol. 31, Issue 10, 2019 October*
- [2] **Bruegge & Allen H.Dutoit** (2010), “Object-Oriented Software Engineering- Using UML, Patterns and Java”, *Third Edition, pp 451-471*
- [3] **Eloussi L.** (2015), “Determining Flaky Tests From Test Failures”
- [4] **Kitchenham B., Stuart C.** (2007), “Guidelines for performing Systematic Literature Reviews in Software Engineering”, *EBSE Technical Report, 2007 January*
- [5] **Lam W, Godefroid P., Nath S., Santhiar A., Thummalapenta S.** (2019), “Root Causing Flaky Tests in a Large-Scale Industrial Setting”, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 July, pp 101–111*
- [6] **Qingzhou L., Farah H., Lamyaa E., Darko M.** (2014), “An Empirical Analysis of Flaky Tests”, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014 November, pp 643–653*
- [7] **Sommerville I.**, “Software Engineering”, *Tenth Edition, pp 730-755*
- [8] **Terragni V., Salza P., Ferrucci F.** (2020), “A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests”, *Proceedings of The ACM/IEEE 42th International Conference on Software Engineering, 2020 October*

## SITOGRAFIA

- [<sup>1</sup>] **Flaky Test (And How to Avoid Them),**  
<https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60>, (Ultimo accesso: Febbraio 2020)
- [<sup>2</sup>] **What is Regression Testing? Definition, Tools & How to Get Started,**  
<https://www.katalon.com/resources-center/blog/regression-testing/>,  
(Ultimo accesso: Febbraio 2020)
- [<sup>3</sup>] **Flaky Tests at Google and How We Mitigate Them,**  
<https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>, (Ultimo accesso: Febbraio 2020)
- [<sup>4</sup>] **Top Computer Languages,** <http://statisticstimes.com/tech/top-computer-languages.php>, (Ultimo accesso: Marzo 2020)
- [<sup>5</sup>] **Announcing the winners of the 2019 Testing and Verification research awards,** <https://research.fb.com/blog/2019/10/announcing-the-winners-of-the-2019-testing-and-verification-research-awards/>, (Ultimo accesso: Marzo 2020)
- [<sup>6</sup>] **Get rid of your flakes - DeFlaker detects and help diagnose flaky tests,**  
<https://www.deflaker.org/>, (Ultimo accesso: Marzo 2020)

## RINGRAZIAMENTI

Quando ho iniziato questo “secondo” percorso universitario, non avrei mai pensato di concluderlo in questo modo. Purtroppo, questo giorno così importante si è svolto in maniera completamente diversa da come mi aspettavo e speravo. Nonostante tutto, anche se oggi voi non siete qui con me a condividere questo momento di gioia è mio dovere ringraziarvi perché se io sono qui è anche merito vostro.

Desidero innanzitutto ringraziare la Professoressa Ferrucci per avermi aiutato e supportato con i suoi consigli durante l’ultimo anno del mio percorso di studi e durante la stesura di questa tesi. Ringrazio anche i Dottori Pasquale Salza e Valerio Terragni, che nonostante tutti i loro impegni sono stati pronti e disponibili ad aiutarci e a consigliarci.

Un grazie enorme va a Giammaria, per aver reso più produttive e meno noiose le giornate di studio. Grazie per tutte le risate che ci siamo fatti in questi due anni, grazie per tutte le volte che mi hai spronato ad andare avanti e grazie per avermi insegnato ad avere un po’ più di fiducia nelle mie conoscenze e capacità.

Il riconoscimento più grande va alla mia famiglia: grazie Alessio, mio compagno di vita per essere ancora qui, per aver accettato qualsiasi mia scelta, grazie per aver sopportato tutte le serate in cui non ho potuto darti retta, tutti i miei scleri e i miei malumori. Ora potrò finalmente concentrarmi sulla nostra vita insieme. Grazie ad Anna per questa seconda “giovinezza”: sono contentissima di averti avuto di nuovo con me come quando eravamo piccole, di aver condiviso le serate con la tv e il cibo spazzatura, di aver sopportato i tuoi mille “non ce la farò mai”. Ti voglio un bene immenso! Il grazie più grande va a voi, mamma e papà, per avermi spronato ad andare avanti e a non fermarmi al primo traguardo raggiunto. Il non avervi fisicamente qui oggi è il mio più grande dispiacere, ma sono sicura che ci rifaremo presto. Vi voglio un bene dell’anima e non smetterò mai, nemmeno per un’istante, di essere fiera di avere due genitori come voi. Spero di avervi reso un po’ orgogliosa di me. Grazie a Salvatore, per aver accettato “ufficialmente” la cognata nel pacchetto ed essere sempre disponibile per me, spero di poter tornare presto a darti fastidio a casa!

Grazie ad Anna, mia grande amica per essere sempre presente anche ora che il tempo è poco. Grazie per tutte le chiacchierate che ci siamo fatte, grazie per tutte le serate ignoranti passate insieme, grazie per tutti i consigli che mi hai dato e per sopportare i miei messaggi vocali di tre minuti. Grazie a Francesco, per tutte le serate indimenticabili organizzate all'ultimo minuto. Mi dispiace per te, ma per vedere Ale dovrà sempre accettare anche la nostra presenza!

Grazie agli amici “acquisiti”, Mario e Manuela, e all’ultimo arrivato Fabio, per tutte le serate passate insieme e per volermi bene nonostante sia un Toro (:-P).

Grazie alle amiche di una vita, Luciana, Enza e Simona, anche se ormai è quasi impossibile vedersi tutte insieme so che sarete sempre presenti.

Grazie a Diego e a Ciccio per le risate che mi avete fatto fare con le vostre idee “geniali” scaturite dal cibo della mensa e per tutte le ore di studio matto e disperato trascorse insieme.

Infine, grazie a me per non averci creduto nemmeno per un attimo, ma per essere comunque riuscita ad arrivare in fondo.