UNIVERSITÀ DEGLI STUDI DI NAPOLI

FEDERICO II

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

TESI DI LAUREA

IN

DEVOPS : SVILUPPO DI UNA TOOLCHAIN IN AMBITO C++

RELATORE:

CH.MO.

TRAMONTANA PORFIRIO

CANDIDATO:

DI COSTANZO FERNANDO

M63000925

ANNO ACCADEMICO 2020/2021

Indice

A tutti coloro che mi hanno sostenuto lungo questo cammino.

Introduzione

Nell'ingegneria del software, la *Continous integration (CI)* è una pratica di sviluppo software in cui i membri di un team integrano il loro lavoro frequentemente, di solito ogni persona integra almeno una volta al giorno - portando ad integrazioni multiple al giorno.
Attraverso la CI ogni integrazione è verificata da una compilazione automatica (incluse altre attività come testing e code coverage) per rilevare gli errori di integrazione il più velocemente possibile : molti team trovano che questo approccio porta a problemi di integrazione significativamente ridotti e permette ad una squadra di sviluppare software coeso più rapidamente.[11]
La CI è stata originariamente concepita per essere complementare rispetto ad altre pratiche, in particolare legate al *Test Driven Development* (sviluppo guidato dai test, TDD). In particolare, si suppone generalmente che siano stati predisposti test automatici che gli sviluppatori possono eseguire immediatamente prima di rilasciare i loro contributi verso l'ambiente condiviso, in modo da garantire che le modifiche non introducano errori nel software esistente. Per questo motivo, la CI viene spesso applicata in ambienti in cui sono presenti sistemi di build automatico e/o esecuzione automatica di test, come *Jenkins* o *Azure DevOps*.

L'obiettivo finale di questo progetto è l'implementazione di una toolchain DevOps, attraverso la quale è possibile ottenere la CI, per una *software house* che effettua consulenza per una delle maggiori società musicali europee.

Attraverso l'analisi dello stato dell'arte del mercato DevOps, in combinazione con le esigenze e le risorse della società, è stata sviluppata una toolchain basata su strumenti come Azure, CMake, Catch2 ed altri con conseguente automazione delle fasi DevOps dalla creazione dell'applicazione alla sua consegna e fornendo notevoli profitti a lungo

termine in termini di impegno e tempo investito in tale processo.

Il buon esito del progetto ha comportato anche un aumento della conoscenza della società alle capacità di DevOps, rendendola interessata ad estendere, in futuro, il meccanismo di automazione anche ad altri progetti.

DevOps

Tale capitolo ha lo scopo di fornire una breve definizione e introduzione a DevOps, partendo da quelli che sono stati i precursori di tale approccio e di come gli obbiettivi/requisiti negli anni per la produzione di software siano cambiati molto.
Vi sono varie definizioni di DevOps , una molto sintetica può essere :

> "DevOps è un insieme di pratiche intese a ridurre il tempo tra il commit di un cambiamento in un sistema e l'immissione di quest'ultimo nella normale produzione, garantendo la qualità del software."[4]

Tale definizione mette in rilievo l'importanza della qualità del cambiamento in un sistema. Essa significa sia idoneità all'uso da parte di vari attori come utenti finali, sviluppatori ed amministratori sia includere il rispetto degli attributi della *Dependability* come *Availability* , *Security*, *Reliability* ed altri vari requisiti non funzionali (ilities).

1.1 Descrizione dell'attributo qualità

Nella definizione non viene specificato come la qualità sia assicurata, ma si richiede che il codice prodotto sia di alta qualità. Ci sono vari metodi per garantire la qualità , tra questi abbiamo :

- Avere a disposizione una varietà di *test-case* automatizzati che devono essere superati prima di immettere il codice modificato in produzione
- Testare le modifiche fatte in produzione con un numero limitato di utenti prima di effettuare il rilascio globale
- Monitorare il codice appena distribuito per un certo periodo di tempo al fine di capire che impatto esso ha avuto.

La definizione prima citata afferisce anche alla qualità delle delivery del codice in produzione.
E' necessario che anche il meccanismo di delivery del codice sia di alta qualità implicando quindi che la *reliability* e la *avaiability* abbiano un elevato valore (in termini probabilistici) poichè se tale meccanismo fallisce regolarmente , il tempo richiesto aumenta. E' possibile identificare due periodi di tempo fondamentali.

- T_1 rappresenta il momento in cui lo sviluppatore esegue il commit del codice appena prodotto che indica la fine del processo di sviluppo e l'inizio di quello di delivery
- T_2 rappresenta invece l'effettivo delivery del codice in produzione

Questa caratterizzazione è infatti necessaria poichè vi è un periodo antecendente alla distribuzione in produzione del codice dove quest'ultimo viene testato attraverso dei *live test* e viene costantemente monitorato al fine di trovare potenziali problemi che possono introdurre poi errori in produzione. Solo se vengono superati tali test ed il monitoraggio da esiti positivi il codice viene considerato come una parte del sistema in produzione. Qualsiasi pratica quindi che voglia ottimizzare i tempi che intercorrono tra T_1 e T_2 è una pratica DevOps che implica l'uso di metodi, tools e forme di coordinamento Agili.

1.2 SDLC

In tale sezione si vuole focalizzare l'attenzione sullo storico del processo di sviluppo del software.
SDLC (Software Developement Life Cycle) è un processo seguito per un progetto software, all'interno di un'organizzazione, con l'obbiettivo di produrre un software di alta qualità che soddisfi o superi le aspettative dei clienti, raggiungendo il completamento entro tempi prestabiliti e non superando un upper-buond in termini di stime dei costi. Consiste quindi in un piano dettagliato che descrive come sviluppare, mantenere, sostituire/alterare o migliorare un software specifico. Il ciclo di vita definisce una metodologia per migliorare la qualità del software e il processo di sviluppo complessivo. Non esiste un unico modello SDLC. Sono divisi in gruppi principali, ognuno con le sue caratteristiche e debolezze.[8]

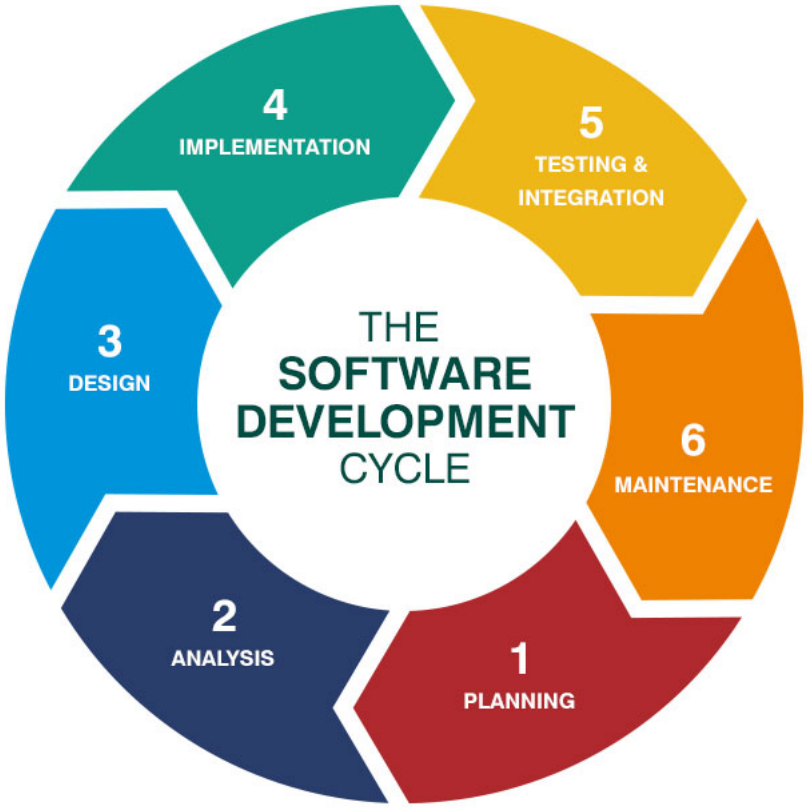

Figura 1.1: SDLC lifecycle

Tutti i modelli hanno solitamente dalle 5 alle 6 fasi principali :

- **Planning** : L'analisi dei requisiti è la fase più importante e fondamentale dell'SDLC. Viene eseguito dai membri senior del team con input dal cliente, dal

reparto vendite, da indagini di mercato ed esperti di dominio nel settore. Queste informazioni vengono quindi utilizzate per pianificare l'approccio progettuale di base e per condurre uno studio di fattibilità del prodotto nelle aree economiche, operative e tecniche. Anche la pianificazione dei requisiti di garanzia della qualità e l'identificazione dei rischi associati al progetto vengono effettuati nella fase di pianificazione. Il risultato dello studio di fattibilità è definire i vari approcci tecnici che possono essere seguiti per realizzare il progetto con successo con rischi minimi.

- **Analysis** : Una volta eseguita l'analisi dei requisiti, il passaggio successivo consiste nel definire e documentare chiaramente i requisiti del prodotto e farli approvare dal cliente o dagli analisti di mercato. Questo viene fatto attraverso un documento SRS (Software Requirement Specification) che consiste in tutti i requisiti del prodotto da progettare e sviluppare durante il ciclo di vita del progetto.

- **Design** : SRS è il riferimento per gli architetti di prodotto per ottenere la migliore architettura per il prodotto da sviluppare. Sulla base dei requisiti specificati in SRS, di solito più di un approccio progettuale per l'architettura del prodotto viene proposto e documentato in un DDS – Design Document Specification.

  Questo DDS viene rivisto da tutte le parti interessate e in base a vari parametri come valutazione del rischio, robustezza del prodotto, modularità del design, vincoli di budget e di tempo, viene selezionato il miglior approccio progettuale per il prodotto.

  Un approccio progettuale definisce chiaramente tutti i moduli architetturali del prodotto insieme alla sua comunicazione e rappresentazione del flusso di dati con i moduli esterni e di terze parti (se presenti). Il design interno di tutti i moduli dell'architettura proposta dovrebbe essere chiaramente definito con i minimi dettagli in DDS.

- **Implementation** : In questa fase di SDLC inizia lo sviluppo vero e proprio e il prodotto viene costruito. In questa fase viene generato il codice di programmazione come da DDS. Se la progettazione viene eseguita in modo dettagliato e organizzato, la generazione del codice può essere eseguita senza troppi problemi.

  Gli sviluppatori devono seguire le linee guida di codifica definite dalla loro organizzazione e strumenti di programmazione come compilatori, interpreti, de-

bugger, ecc. vengono utilizzati per generare il codice. Per la codifica vengono utilizzati diversi linguaggi di programmazione di alto livello come C, C++, Pascal, Java e PHP. Il linguaggio di programmazione viene scelto in relazione al tipo di software che si sta sviluppando.

- **Testing & Integration** : Questa fase è solitamente un sottoinsieme di tutte le fasi poiché nei moderni modelli SDLC, le attività di test sono principalmente coinvolte in tutte le fasi di SDLC. Tuttavia, questa fase si riferisce alla fase di sola prova del prodotto in cui i difetti del prodotto vengono segnalati, monitorati, riparati e ritestati, fino a quando il prodotto raggiunge gli standard di qualità definiti nell'SRS.

- **Maintenance** : Una volta che il prodotto è stato testato e pronto per essere distribuito, viene rilasciato formalmente nel mercato appropriato. A volte la distribuzione del prodotto avviene in più fasi secondo la strategia aziendale di tale organizzazione. Il prodotto può essere prima rilasciato in un segmento limitato e testato nell'ambiente aziendale reale (UAT-User Acceptance Testing).

  Quindi, in base al feedback, il prodotto può essere rilasciato così com'è o con miglioramenti suggeriti nel segmento di mercato di destinazione. Dopo che il prodotto è stato rilasciato sul mercato, viene eseguita la sua manutenzione per la base di clienti esistente.

1.2.1 Waterfall

E' un modello SDLC in cui il processo di sviluppo assomiglia al flusso di una cascata, muovendosi passo dopo passo attraverso le fasi di analisi, progettazione, realizzazione, test, implementazione e supporto. Questo modello SDLC include l'esecuzione graduale di ogni fase completamente. Questo processo è rigorosamente documentato e predefinito con le funzionalità previste per ogni fase di questo modello del ciclo di vita dello sviluppo del software.

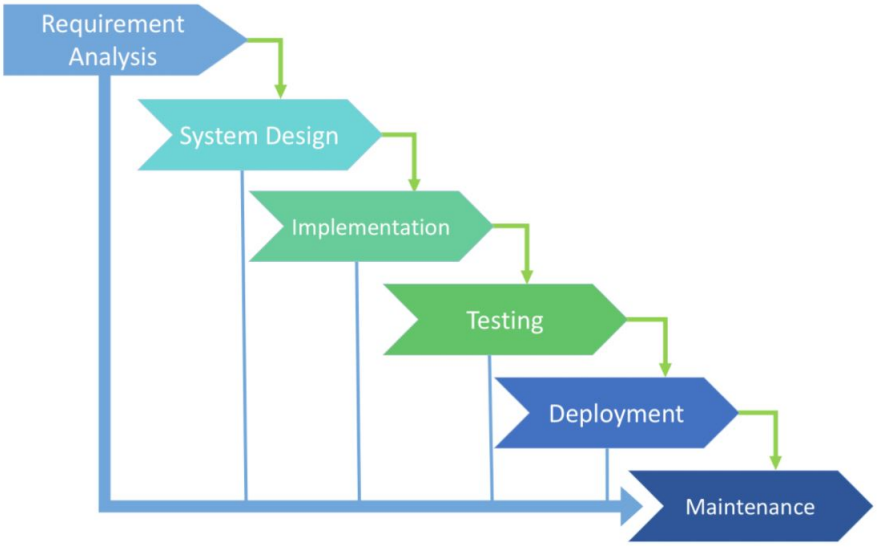

Figura 1.2: SDLC Waterfall

E' un modello indicato quando :

- la qualità è più importante del costo o della schedulazione temporale.
- I requisiti sono ben conosciuti, chiari e fissi.
- È necessaria una nuova versione di un prodotto esistente.
- È necessario eseguire il porting di un prodotto esistente in una nuova piattaforma.

Di seguito i vantaggi e gli svantaggi apportati da questo modello

- **Vantaggi**
  - Facile da capire e da utilizzare;
  - Utile per progetti piccoli,di routine e ripetiti dove i requisiti sono chiaramente definiti;
  - Ogni fase ha risultati specifici;
  - La verifica fase per fase garantisce l'individuazione precoce degli errori/incomprensioni.
- **Svantaggi**
  - È molto difficile tornare indietro ad una qualsiasi fase dopo che questa è stata completata;
  - Inappropriato per progetti a lungo termine;
  - Costoso e richiede più tempo, in quanto è difficile stimare le risorse e i costi se non si è svolta la prima fase di analisi;
  - Non è flessibile e quindi gli aggiustamenti di scope sono molto difficili e costosi.

1.2.2 Iterativo ed Incrementale

Il modello iterativo SDLC non necessita dell'elenco completo dei requisiti prima dell'inizio del progetto. Il processo di sviluppo può iniziare con i requisiti per la parte funzionale, che può essere ampliata in seguito. Il processo è ripetitivo, consentendo di realizzare nuove versioni del prodotto per ogni ciclo. Ogni iterazione (che dura da due a sei settimane) include lo sviluppo di un componente separato del sistema e, successivamente, questo componente viene aggiunto al funzionale sviluppato in precedenza. Parlando con la terminologia matematica, il modello iterativo è una realizzazione del metodo dell'approssimazione sequenziale; ciò significa un avvicinamento graduale alla forma del prodotto finale pianificato.

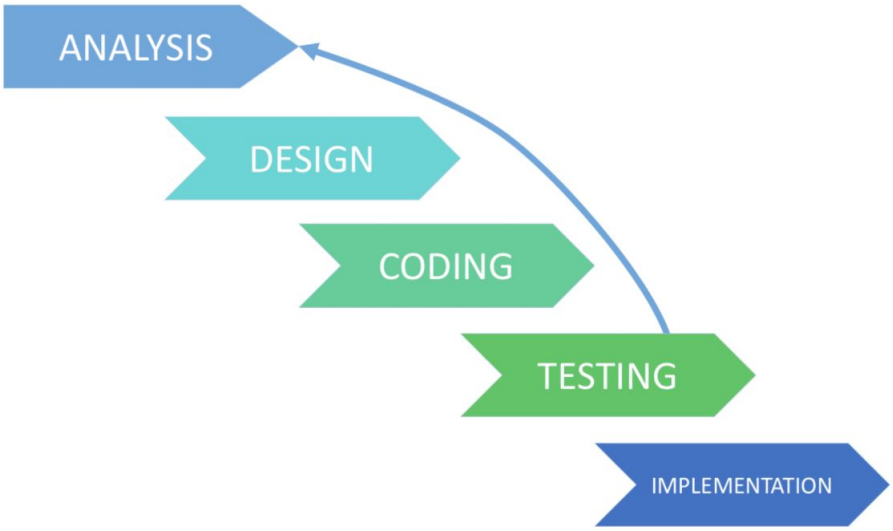

Figura 1.3: SDLC Iterative And Incremental

E' un modello indicato quando :

- I requisiti per il prodotto finale sono rigorosamente predefiniti;
- Applicato ai progetti su larga scala;
- Il compito principale è predefinito, ma i dettagli possono avanzare con il tempo;

Di seguito i vantaggi e gli svantaggi apportati da questo modello

- **Vantaggi**
  - Alcune funzioni possono essere sviluppate rapidamente all'inizio del ciclo di vita dello sviluppo;
  - L'iterazione più breve è: le fasi di test e debug sono più semplici;
  - È più facile controllare i rischi poiché le attività ad alto rischio vengono completate per prime;
  - Flessibilità e disponibilità ai cambiamenti dei requisiti.
- **Svantaggi**
  - Il modello iterativo richiede più risorse rispetto al modello a cascata;
  - Pessima scelta per i piccoli progetti;
  - Il processo è difficile da gestire;
  - L'analisi dei rischi richiede il coinvolgimento di specialisti altamente qualificati.

1.2.3 Spirale

Il modello a spirale è simile al modello incrementale, con più enfasi sull'analisi dei rischi. Il modello a spirale ha quattro fasi: Pianificazione, Analisi del rischio, Ingegneria e Valutazione. Un progetto software passa ripetutamente attraverso queste fasi in iterazioni (chiamate spirali in questo modello). Nella spirale di base, a partire dalla fase di pianificazione, si raccolgono i requisiti e si valuta il rischio. Ogni spirale successiva costruisce sulla spirale di base. I requisiti sono raccolti durante la fase di pianificazione. Nella fase di analisi del rischio, viene intrapreso un processo per identificare il rischio e le soluzioni alternative. Un prototipo è prodotto alla fine della fase di analisi del rischio. Il software viene prodotto nella fase di ingegnerizzazione, con test alla fine della fase. La fase di valutazione permette al cliente di valutare il risultato del progetto fino a quel momento prima che il progetto continui nella spirale successiva. Nel modello a spirale, la componente angolare rappresenta il progresso e il raggio della spirale rappresenta il costo.[1]

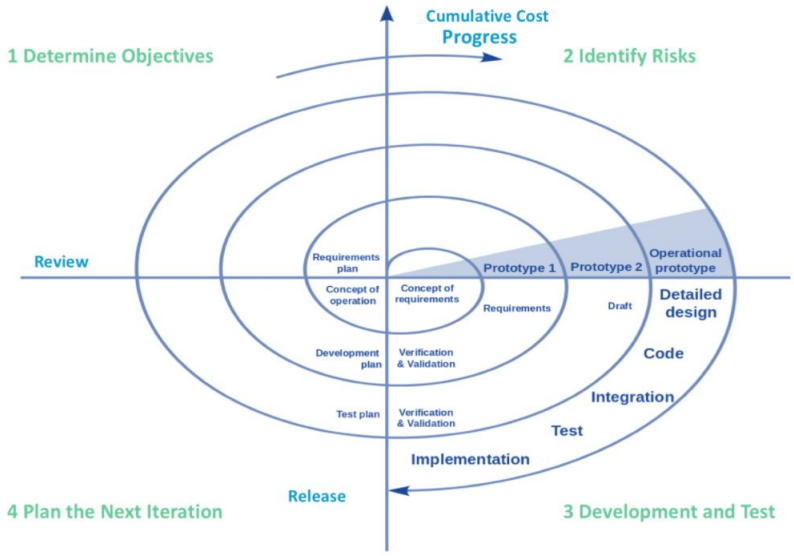

Figura 1.4: SDLC Spiral

E' un modello indicato quando :

- per progetti di medio-alto rischio;
- Quando la valutazione del rischio e del costo sono importanti;

- Quando ci si aspetta significativi cambiamenti al progetto;
- Quando gli utenti nono sono proprio sicuri di quello che desiderano.

Di seguito i vantaggi e gli svantaggi apportati da questo modello

- **Vantaggi**
  - Elevata quantità di analisi del rischio;
  - Buono per progetti di grandi dimensioni e mission-critical;
  - Il software viene prodotto all'inizio del suo ciclo di vita.
- **Svantaggi**
  - Può essere un modello costoso da utilizzare;
  - L'analisi dei rischi richiede competenze molto specifiche. ;
  - Il successo del progetto dipende fortemente dal rischio in fase di analisi.

1.2.4 V-Model

Proprio come il modello a cascata, il ciclo di vita a V è un percorso sequenziale di esecuzione dei processi. Ogni fase deve essere completata prima che la fase successiva inizi. I test sono enfatizzati in questo modello più del modello a cascata. Le procedure di test sono sviluppate all'inizio del ciclo di vita prima di qualsiasi codifica, durante ciascuna delle fasi che precedono l'implementazione. I requisiti iniziano il modello del ciclo di vita proprio come il modello a cascata. Prima che lo sviluppo inizi, viene creato un piano di test del sistema. Il piano di test si concentra sulla soddisfazione della funzionalità specificata nella raccolta dei requisiti. La fase di progettazione di alto livello si concentra sull'architettura e la progettazione del sistema. Un piano di test di integrazione viene creato in questa fase per testare i pezzi della capacità dei sistemi software di lavorare insieme. Tuttavia, la fase di progettazione di basso livello è quella in cui vengono progettati i componenti software effettivi, e anche in questa fase vengono creati i test unitari.[1]

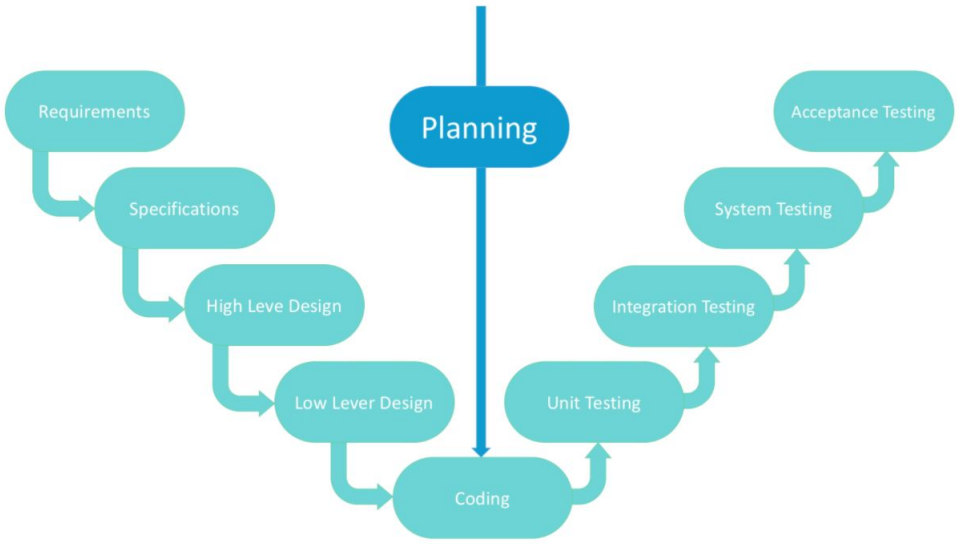

Figura 1.5: SDLC V-Model

Di seguito i vantaggi e gli svantaggi apportati da questo modello

- **Vantaggi**
  - Semplice e facile da usare;
  - Ogni fase ha risultati specifici;
  - Maggiori possibilità di successo rispetto al modello a cascata grazie allo sviluppo precoce dei piani di test durante il ciclo di vita.

- **Svantaggi**
  - Molto rigido come il modello a cascata;
  - Il software viene sviluppato durante la fase di implementazione, quindi non vengono prodotti prototipi iniziali del software;
  - Non fornisce un percorso chiaro per i problemi trovati durante le fasi di test.

1.2.5 Agile

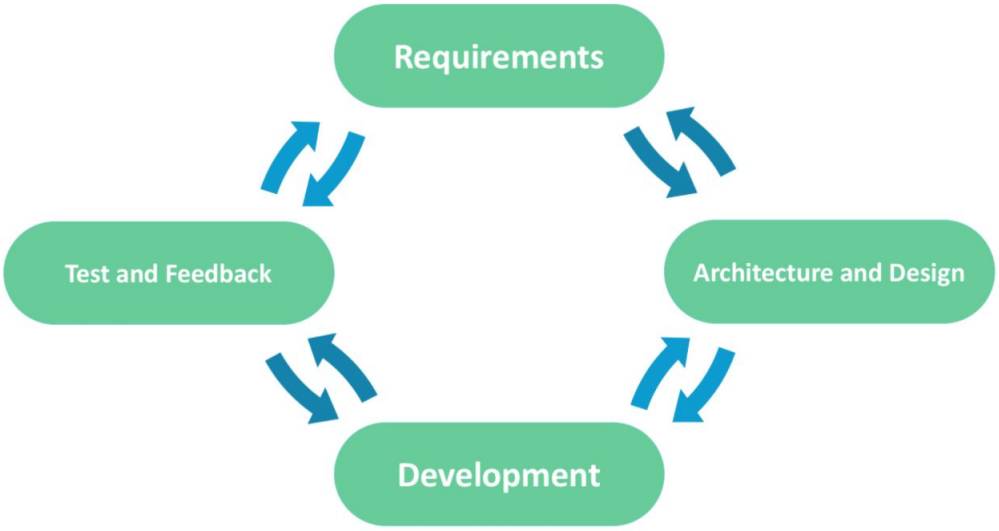

Figura 1.6: SDLC Agile Model

> "Agilità è la capacità di creare e rispondere ai cambiamenti nell'ottica di migliorare il profitto [o produttività] in un ambiente aziendale turbolento. Agilità è l'abilità di trovare un equilibrio tra flessibilità e stabilità" [2]

Nel modello waterfall si ritiene che i sistemi software possano essere realizzati in maniera lineare, cioè per fasi. In tale processo lineare, inizialmente vengono raccolti e analizzati tutti i requisiti ed il completamento della progettazione costituisce il passo successivo. Infine, il risultato di tale progettazione diventa direttamente il software che sarà portato in produzione, passando per una lunga fase di integrazione e test. Nonostante che questo ed altri tradizionali approcci, sembrano validi, non sono in grado di affrontare in modo efficace i problemi legati ai costi e al tempo. Da circa una decina di anni è, tuttavia, emersa una nuova filosofia di sviluppo che è alla base di una vasta raccolta di nuovi processi nota come Agile Software Development. Tali nuovi processi si concentrano maggiormente: sulle interazioni tra le persone e su un rapido sviluppo di codice piuttosto che sulla documentazione e sulla pianificazione. Già dai primi anni 90, iniziano a nascere nuove metodologie innovative quali: eXtreme Programming, Scrum, Feature Driven Development, DSDM, Crystal o Lean Software Development,

che verranno poi considerate come metodologie agili, in quanto consentono di poter cambiare continuamente le specifiche durante lo sviluppo mediamente un attivo coinvolgimento con il committente. Nel 2001, 17 software developer si sono incontrati in un resort, per discutere i metodi di sviluppo, dando origine al Manifesto Agile1 (contenente 12 punti fondamentali), che tuttora rimane il riferimento principale, ufficiale e la fonte di ispirazione per qualsiasi iniziativa che intenda qualificarsi come agile. Il significato letterale della parola Agile è "caratterizzato da rapidità, leggerezza e facilità di movimento". Questo significa quindi che il processo di sviluppo agile consiste in una distribuzione rapida di software caratterizzata da una maggiore facilità e flessibilità di sviluppo. Utilizzando uno dei tanti punti del manifesto, si può dire che: "La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua.", ciò significa creare rapidamente dei prototipi funzionanti da poter condividere con il cliente. In generale per Metodologie agili, si intendono tutte quelle metodologie non tradizionali, cioè che non prevedono una strutturazione lineare dello sviluppo software ma che pongono maggiore attenzione sul cliente, il quale diventa l'elemento principale su cui è incentrato lo sviluppo; nella figura 1.6 sono riepilogate le fasi principali di questo approccio. Agilità è la capacità sia di rispondere che di creare cambiamenti ("abbracciare il cambiamento" rappresenta uno slogan dell'Agile Project Management). Cambiamenti nelle esigenze, nei requisiti e nei bisogni del cliente finale fanno dell'Agile una metodologia dinamica e non prescrittiva o plan-driven, dove deve guidare un piano realizzato a priori.

Il contesto nel quale si deve essere agili è un ambiente di business turbolento, dove la turbolenza può avere diverse origini, tra le quali: - Turbolenza legata ai requisiti della soluzione, e questa turbolenza si traduce in continui cambiamenti di ambito, in ritardi, in maggiori costi che il progetto deve sostenere; - Turbolenza legata alla tecnologia che si è scelto di adottare per realizzare l'oggetto di un progetto; - Turbolenza legata alla numerosità del team di progetto: - Turbolenza legata proprio al mercato di riferimento, alle scelte strategiche dell'azienda o al contesto territoriale e politico nel quale l'azienda si muove. L'agilità è un'abilità capace di far convivere simultaneamente la flessibilità (il caos) con la stabilità (ordine). La flessibilità nel sapere rispondere ai mutamenti del mercato, del contesto e dei requisiti del cliente può essere raggiunta acquisendo quei valori, principi e pratiche dell'Agile e costruendo un ambiente sano nel quale il team possa lavorare proficuamente aumentando così anche la sua produttività (Corbucci, Seconda Edizione).[2]

Affinché una metodologia possa definirsi Agile, questa deve possedere quattro requisiti fondamentali. Deve essere:

- **Iterativa** : lo sviluppo è un processo iterativo, che costruisce versioni parziali di prodotto, esplodendole attraverso successivi e brevissimi periodi di tempo nei quali il prodotto stesso è oggetto di revisioni ed adattamento sia al contesto, in continua evoluzione, sia alle nuove richieste del cliente. Ogni iterazione deve essere timeboxed, ossia le iterazioni devono avere una durata fissa e breve (dalle due alle quattro settimane) e concordata anticipatamente tra tutto il team di progetto.
- **Incrementale** : il prodotto rilasciato con approccio Agile ha la caratteristica di essere un prodotto fatto di incrementi che il team rilascerà al termine di ogni iterazione.
- **Team Cross-Functional** : se il team si impegna a rilasciare micro incrementi di prodotto potenzialmente il team deve possedere tutti gli skill necessari e sufficienti per poterli rilasciare.
- **Team Empowered** : saper creare e rispondere ai cambiamenti. Il saper rispondere ai cambiamenti è la tipica gestione delle change request (richiesta di cambiamento rispetto a quanto previsto dal piano) argomento ampiamente dibattuto in un approccio tradizionale.[2]

Nell'Agile, il lavoro viene scomposto in una lista di piccoli e concreti deliverable e l'elenco viene ordinato in base alla priorità dettata dalle esigenze di business di quel momento. Subito dopo viene stimato lo sforzo rispetto a ciascuno di essi. Sebbene vengano stimati con maggior dettaglio gli elementi più in alto, generalmente le stime di effort e di costo non possono che essere top-down, considerato che il Backlog, questa coda prioritizzata delle funzionalità, è qualcosa in divenire.
Vantaggi :

- Il ciclo di vita dello sviluppo del software è più rapido
- Il programma è prevedibile analizzando ciascuno sprint

- Focus sul cliente
- Flessibile nell'accettare i cambiamenti
- Promuove comunicazioni efficienti
- Ideale per progetti con finanziamento non fisso

Svantaggi :

- Agile richiede un alto grado di coinvolgimento del cliente, che non tutti i clienti sono disposti a dare
- Agile prevede che ogni membro del team sia completamente dedicato al progetto
- Comporta un aumento dei costi al fine di gestire cambiamenti di priorità e sprint aggiuntivi

1.3 DevOps : Development And Operation

DevOps è in realtà l'acronimo di **Development** (attività di sviluppo di un prodotto o servizio) and **Operation** (attività di messa in opera e mantenimento in vita degli stessi) ed è un termine coniato per descrivere l'approccio che viene seguito utilizzando strumenti, metriche e processi per aiutare il team a offrire esperienze di alta qualità, utilizzando l'*automazione* per ridurre al minimo gli errori e massimizzare l'efficienza nella consegna dei prodotti.
Si concentra sul consentire agli sviluppatori di utilizzare e gestire i processi e l'infrastruttura delle operazioni software senza la necessità di un team operativo separato.
DevOps è un nuovo movimento che cerca di realizzare le esigenze di business per rapidità di consegna dei prodotti software, mantenendo stabili gli ambienti già in vita.
Usa due approcci: il primo è la promozione di una stretta collaborazione tra programmatori e sistemisti; il secondo applica le pratiche dello sviluppo Agili di software (collaborazione, automazione, semplicità e così via) ai processi di operation, come l'approvvigionamento, la gestione dei cambiamenti e il monitoraggio della produzione. Comprende quindi cultura, processi e strumenti tutti al supporto di migliorare la comunicazione tramite feedback e ottenere tempi di consegna e risultati più prevedibili.
Per comprendere DevOps, tuttavia, è importante essere consapevoli del contesto da cui provengono le persone in Ops o Dev : Il team di sviluppo non include solo i programmatori (coloro che scrivono il codice vero e proprio) ma tutti i dipendenti che lavorano alla creazione del prodotto: tester, personale addetto al controllo qualità e così via. Il team operativo può includere anche amministratori di database e di sistema, tecnici di rete, esperti di sicurezza e tutti quei dipendenti che supervisionano la spedizione del codice in produzione, monitorano e mantengono il prodotto.
La forma di questi team dipende dalla strategia interna dell'azienda: in alcuni casi, il dipartimento di Quality Assurance potrebbe non essere ritenuto necessario, in altri la rete potrebbe non essere sotto il controllo dell'azienda. Pur variando nelle forme, rimane la distinzione principale.
Lo sviluppo e le operazioni hanno in definitiva lo stesso obiettivo: l'aumento della produttività all'interno dell'azienda al fine di migliorare i profitti. Tuttavia, la sola esistenza di questi team porta a un conflitto che può essere espresso come segue: **"Lo sviluppo tende al cambiamento, le operazioni lo teme"** come mostrato in figura 1.7.
Questo perché, mentre il team di sviluppo punta a introdurre nuove funzionalità (ad

esempio, un'interfaccia utente più intuitiva o una nuova architettura del database che migliora l'efficienza), correzioni di bug e ogni tipo di modifica, che alla fine deve essere messa in produzione, il team delle operazioni vuole evitare l'introduzione di nuovo codice, che potrebbe minare la stabilità, la disponibilità o la robustezza del prodotto/-servizio.

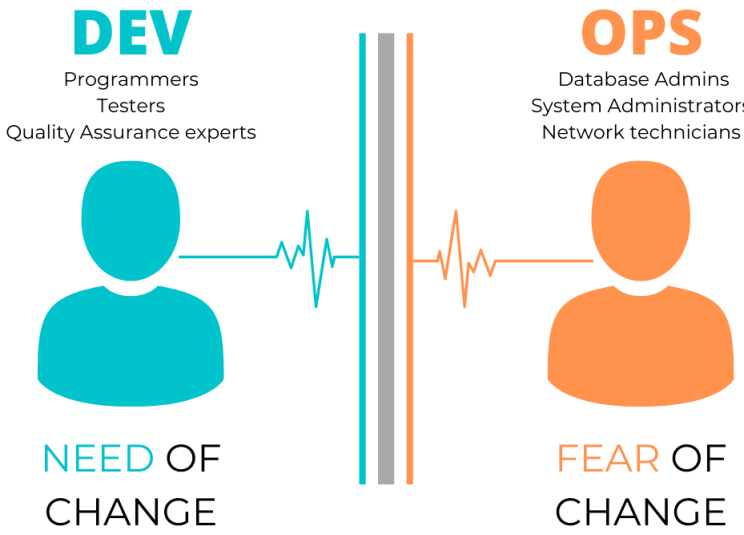

Figura 1.7: Conflitto tra Dev ed Ops

Il risultato principale di questa divisione è che ogni squadra lavora cercando di migliorare il proprio profitto, non interessandosi a comunicare con l'altra fino a quando non emerge un nuovo problema.

La presenza di questi conflitti intrinseci non può essere completamente risolta a causa della natura delle dei due team. Tuttavia, ciò che si può fare è ridurre al minimo il loro impatto: promuovere la collaborazione, la fiducia e la creazione di un ambiente non ostile, ma anche massimizzare l'efficacia delle metodologie utilizzate durante tutto il ciclo di sviluppo del software.

Tutti questi elementi stanno alla base del DevOps, che in definitiva può essere definito come un'ideologia basata su Cultura, Processi e Strumenti (l'ultimo dei quali sarà discusso nel capitolo 2).

Prima di DevOps, il team di sviluppo e il team operazionale lavoravano in modo completamente distaccato. Test e implementazione erano attività isolate che venivano eseguite solo dopo la progettazione e quindi consumavano più tempo rispetto ai cicli attuali di build.
Lo scenario progettuale, senza l'impiego di DevOps, è il seguente:

- i team di sviluppo e il team operazionale avevano tempistiche distinte e non sincronizzate causando ulteriori ritardi. Nello specifico Developer e Operations hanno i loro processi, strumenti e basi di conoscenza indipendenti. L'interfaccia tra loro nell'era pre-DevOps era di solito un sistema di ticket: i team di sviluppo richiedevano l'implementazione di nuove versioni del software e il personale operativo gestiva manualmente quei ticket. In questo accordo però, i team di sviluppo cercavano continuamente di spingere le nuove versioni in produzione, mentre il personale operativo tentava di bloccare queste modifiche per mantenere la stabilità del software. Teoricamente, questo modus operandi offre una maggiore stabilità per il software in produzione però in pratica comportava lunghi ritardi tra gli aggiornamenti del codice e la distribuzione, oltre a processi di risoluzione dei problemi inefficaci;

- i membri del team impiegavano molto tempo a testare, distribuire e progettare invece di costruire il progetto;

- Nessuna pratica di programmazione estrema (XP), ad esempio, comprendeva la distribuzione, di conseguenza, il passaggio alla produzione tendeva ad essere un processo stressante nelle organizzazioni, infatti venivano svolte attività manuali soggette a errori e, anche, correzioni dell'ultimo minuto. [5]

DevOps influenza il ciclo di vita complessivo dell'applicazione che può essere pensato come una pipeline composta da tali fasi [14]:

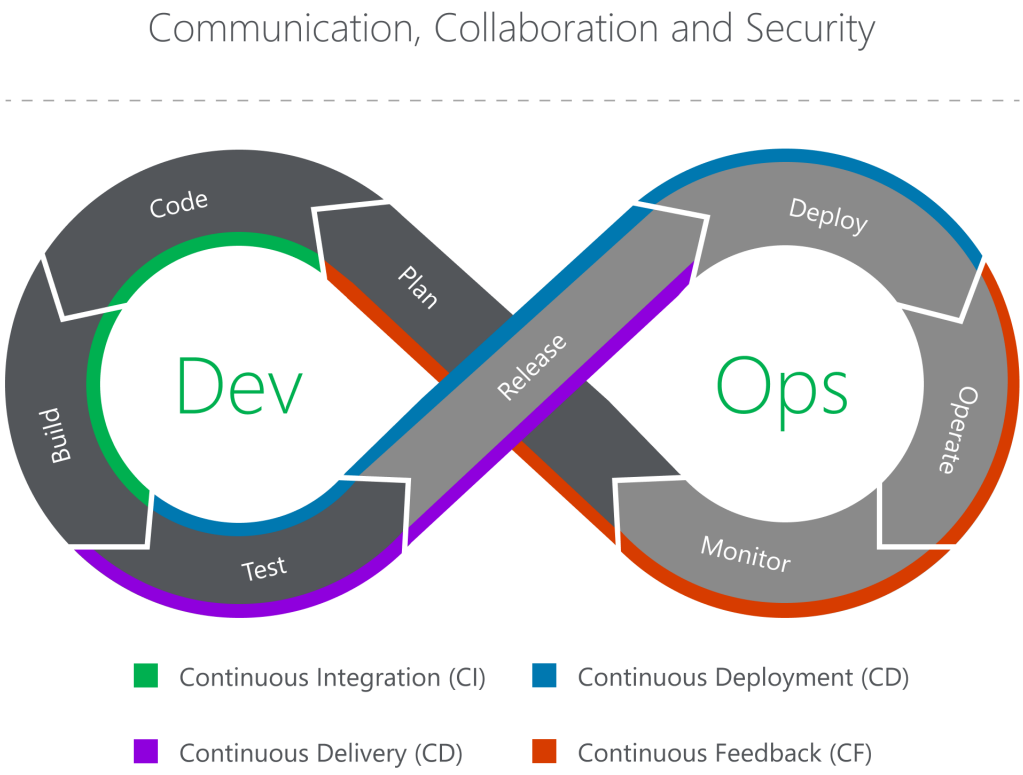

Figura 1.8: DevOps lifecycle

- **Plan** : La fase del piano copre tutto ciò che accade prima che gli sviluppatori inizino a scrivere il codice, ed è qui che un Product Manager o un Project Manager si guadagna da vivere. I requisiti e il feedback vengono raccolti dalle parti interessate e dai clienti e utilizzati per costruire una roadmap del prodotto per guidare lo sviluppo futuro. La roadmap del prodotto può essere registrata e monitorata utilizzando un sistema di gestione dei ticket come Jira, Azure DevOps o Asana che fornisce una varietà di strumenti che aiutano a tenere traccia dell'avanzamento del progetto, dei problemi e delle tappe fondamentali;

- **Code** : la fase in cui inizia lo sviluppo vero e proprio del software;

- **Build** : La fase di build è il punto in cui DevOps emerge. Una volta che uno sviluppatore ha terminato un'attività, esegue il commit del proprio codice in un repository di codice condiviso. Esistono molti modi per farlo, ma in genere lo sviluppatore invia una richiesta pull, una richiesta per unire il nuovo codice con la codebase condivisa. Un altro sviluppatore esamina quindi le modifiche ap-

portate e, una volta soddisfatto che non ci sono problemi, approva la richiesta di pull. Questa revisione manuale dovrebbe essere rapida e leggera, ma è efficace per identificare i problemi in anticipo. Contemporaneamente, la richiesta pull attiva un processo automatizzato che costruisce la codebase ed esegue una serie di test end-to-end, di integrazione e di unità per identificare eventuali regressioni. Se la build ha esito negativo o uno qualsiasi dei test ha esito negativo, la richiesta pull non riesce e lo sviluppatore viene avvisato per risolvere il problema. Controllando continuamente le modifiche al codice in un repository condiviso ed eseguendo build e test, possiamo ridurre al minimo i problemi di integrazione che si verificano quando si lavora su una base di codice condivisa ed evidenziare i bug interrotti all'inizio del ciclo di vita dello sviluppo;

- **Test** : tale fase è eseguita solo in ambienti sicuri e se la compilazione riesce: se automatizzata, non solo farebbe risparmiare tempo, ma garantirebbe anche robustezza in quanto si evita la responsabilità dell'errore umano.
  In questa fase è inoltre possibile impostare parametri di qualità che, se non rispettati, possono comportare la chiusura automatica dell'intero processo;

- **Release** : La fase di rilascio è una *milestone* in una pipeline DevOps: è il punto in cui si decide che una build è pronta per la distribuzione nell'ambiente di produzione. A questo punto, ogni modifica del codice ha superato una serie di test manuali e automatizzati e il team operativo può essere certo che sono improbabili problemi di rottura e regressioni. A seconda della maturità DevOps di un'organizzazione, possono scegliere di distribuire automaticamente qualsiasi build che arrivi a questa fase della pipeline. Gli sviluppatori possono utilizzare i flag delle funzionalità per disattivare le nuove funzionalità in modo che non possano essere viste dai clienti finché non sono pronti per l'azione. Questo modello è considerato il nirvana di DevOps ed è il modo in cui le organizzazioni riescono a distribuire più versioni dei loro prodotti ogni giorno;

- **Deploy** : il punto in cui la build è considerata pronta per essere introdotta all'ambiente di produzione. Poiché questo passaggio è cruciale, un'azienda può decidere se è saggio introdurre automaticamente la build che ha soddisfatto tutti gli standard richiesti dai passaggi precedenti o se sono preferite altre strategie che richiedono un intervento manuale;

- **Operate** : rappresenta la fase in cui il software viene spostato da un ambiente controllato ad altri: possono esistere versioni diverse, ognuna successiva con più funzionalità rispetto alla precedente, ma solo le precedenti potrebbero essere disponibili per gli utenti finali. Ciò consente un maggiore controllo sulla pianificazione aziendale;

- **Monitor** : è la fase durante la quale i dati e le analisi sul comportamento dei clienti, le metriche delle prestazioni e i potenziali problemi vengono raccolti e inviati direttamente al team di sviluppo, per concludere questo ciclo.

1.3.1 Ruolo dell'automazione : CI/CD

Ci sono molte teorie dietro a quale grado di automazione dovrebbe essere introdotto, ma per lo più rientrano in due categorie: come molti sostengono il motto *"Automatizzare tutto"*, altri ritengono che dovrebbe toccare solo pochi elementi chiave dell'intero processo.
Indubbiamente, il raggiungimento di un alto livello di automazione dovrebbe essere l'obbiettivo principale.
Questo porterebbe una convergenza verso la prima categoria di pensiero ma un sistema completamente automatizzato non è vantaggioso dal punto di vista aziendale in uno scenario di sviluppo/manutenzione software, poiché agilità e dinamicità sono aspetti importanti: il costo della modifica di parti di un processo automatizzato è abbastanza rilevante, quindi essere costretti a modificarlo comporterebbe una perdita economica, piuttosto che una crescita. Inoltre se il sistema automatizzato manifesta un errore al suo interno, il sistema stesso non sarebbe in grado di identificarlo e solo un intervento umano potrebbe risolverlo.
Considerando queste motivazioni, un *trade-off* sarebbe la soluzione ottimale.

Con l'aumento della necessità di automazione, molti innovatori hanno iniziato a iniettare automazione in tutte le fasi della pipeline DevOps, a partire dalle fasi iniziali. Tuttavia, nel corso del tempo tale modalità si è ampliata, arrivando anche alle fasi più recenti, quelle gestite dal Team Operations. Nasce così l'idea di avere un flusso "Continuo": nel mondo delle *keyword* DevOps, la più comune da sentire è l'abbreviazione **CI/CD**, che significa *Continuous Integration/Continuous Delivery*. Queste due pratiche sono pilastri chiave della pipeline DevOps, consentendo di sfruttare appieno la strategia di automazione.

Continuous Integration

La Continuous Integration (definita in italiano anche "integrazione continua") è una tecnica di sviluppo agile di software. Con questo metodo di integrazione gli sviluppatori integrano porzioni di codice finiti nell'applicazione anche più volte al giorno, piuttosto che integrarle tutte soltanto alla fine del progetto (Digital Guide IONOS, 2019). Uno dei principali vantaggi dell'adozione della CI è che consente di risparmiare tem-

po, durante il ciclo di sviluppo, identificando e risolvendo tempestivamente i conflitti. È anche un ottimo modo per ridurre la quantità di tempo speso per la correzione di bug e regressione mettendo più enfasi sull'avere una buona suite di test. Infine, aiuta a condividere una migliore comprensione della code base e delle funzionalità che si stanno sviluppando per i clienti. L'obiettivo di questo metodo moderno è quello di suddividere il lavoro in porzioni più piccole, per rendere il processo stesso di sviluppo più efficiente e poter reagire con maggiore flessibilità alle modifiche. Gli sviluppatori caricano il proprio codice ultimato una o più volte al giorno sulla *mainline*, ovvero il codice sorgente a cui hanno accesso tutti i programmatori. Dal momento che si tratta di porzioni di codice relativamente ridotte, anche l'integrazione richiede meno tempo. Nel caso in cui venga individuato un errore è possibile identificarlo e risolverlo in modo relativamente veloce. Bisogna ricordare, tuttavia, che lo sviluppatore non lavora da solo sul programma. Mentre lui ha apportato le sue modifiche, molto probabilmente i suoi pari avranno svolto altri compiti, quindi ciascun sviluppatore del team possiede una versione differente sul proprio computer. Anche la versione della mainline sarà sicuramente cambiata nel frattempo, fatto che costringe il programmatore a integrare tutte le modifiche innanzitutto nella sua Working Copy. Se a questo punto emerge un errore, dovrà risolverlo. Solo allora il programmatore potrà aggiungere le sue modifiche alla mainline e testare nuovamente il programma. Se non emergono altri errori, che possono presentarsi nel caso in cui non abbia aggiornato correttamente la sua Working Copy, la procedura è conclusa e lo sviluppatore può dedicarsi al compito successivo [10].

Tale metodo prevede il rispetto di alcuni principi, tra cui :

- **Una sola sorgente** : Tutti i membri del team devono utilizzare la stessa sorgente (lo stesso repository) quando lavorano sul codice. Questo non vale soltanto per il codice sorgente di per sé, perché per garantire il corretto funzionamento di un'applicazione sono necessari anche altri elementi come i data base;

- **Build automatizzati** : Per creare un programma funzionante dal codice sorgente è necessario compilarlo, attualizzare i data base e spostare i file nelle posizioni corrette. Questo processo può essere automatizzato. Idealmente dovrebbe essere possibile eseguire un processo di build con un solo comando;

- **Sistemi di testing automatico** : L'integrazione di meccanismi di testing nel processo di build consente al team di automatizzare, e dunque velocizzare ulterior-

mente, la Continuous Integration. Proprio come il processo di build, anche i test devono poter essere eseguiti nel modo più efficiente possibile (la cosa migliore è implementare una serie di meccanismi di controllo diversi), per ogni modifica apportata al repository principale.
Questo consente di rilevare eventuali errori in anticipo, riducendo al minimo le interruzioni del team. Il testing è l'elemento centrale del metodo agile Test Driven Development (TDD);

- **Integrazione giornaliera** : L'integrazione continua può funzionare solo se tutti i membri del team condividono il sistema. È sufficiente che un membro del team non integri costantemente il suo codice nella mainline perché tutti gli altri partano da presupposti sbagliati. La comunicazione gioca un ruolo fondamentale, infatti se tutti gli sviluppatori restano aggiornati sul lavoro degli altri, molte difficoltà possono essere prevenute;

- **Riparazione Immediata** : Nella Continuous Integration è di fondamentale importanza che la mainline non contenga mai una versione difettosa. Questo significa per gli sviluppatori che la riparazione non deve mai essere rimandata. Secondo Martin Fowler il fatto che i build non funzionino e che sia necessario rivedere il codice non è un problema, ma il sistema dell'integrazione continua prevede una riparazione immediata. Tutti gli sviluppatori devono infatti poter contare sul fatto che il codice nella mainline sia funzionante, altrimenti l'intero team effettua programmazioni in sistemi instabili innescando una valanga di errori.

Nonostante le sue caratteristiche positive, nel lavoro quotidiano la Continuous Integration non offre solo dei vantaggi. Infatti, è vero che consente di evitare un'ampia e prolungata fase di integrazione alla fine del processo e permette di risolvere i problemi tempestivamente, ma per un team ben coordinato l'adattamento alla Continuous Integration può risultare difficile. In questi casi il metodo può persino provocare l'effetto contrario e risultare più dispendioso in termini di tempo.

Vantaggi	Svantaggi
Individuazione precoce degli errori	Adattamento a processi nuovi
Feedback continuo	Richiede server e ambienti aggiuntivi
Nessun sovraffaticamento a causa di un'unica grande fase di integrazione alla fine	Richiede processi di testing adeguati
Annotazione precisa delle modifiche	Possono crearsi code se più sviluppatori desiderano integrare i propri codici contemporaneamente
Disponibilità costante della versione attuale e funzionante	
Promuove il lavoro progressivo	

Figura 1.9: CI vantaggi e svantaggi

Continuous Delivery

La Continuous Delivery punta a fare un ulteriore passo avanti: estendendo l'automazione alla fase di Release, in presenza di ambienti di testing/staging predisposti prima dell'ambiente di produzione reale, questa pratica consente di consegnare loro i build in tempi più brevi, considerando anche che questo processo è sempre noioso e richiede tempo. Poiché molti clienti non vorrebbero avere versioni diverse del prodotto distribuite frequentemente e direttamente all'utente finale, Continuous Delivery si adatta perfettamente a tale scenario, proprio perché la versione più recente della build può essere automaticamente disponibile per il team operativo per eseguire test aggiuntivi e analisi.

Oltre a questi due aspetti fondamentali di DevOps, che sono comunemente accettati, le persone hanno iniziato a definire altri modelli continui, cercando di espandere il processo di automazione a tutte le restanti fasi DevOps. Chiudere il cerchio non è un compito facile, in quanto il lavoro svolto dal team Operations coinvolge sempre un fattore umano: lo si vede facilmente osservando il feedback degli utenti, una componente ricca e difficile da automatizzare. Tuttavia, la direzione del mercato si sta muovendo in quella direzione, con la creazione di nuovi pattern (Figure 1.8 1.10) come:

- **Continuous Deployment** : La distribuzione continua è una strategia in cui qualsiasi *commit* del codice che supera la fase di test viene automaticamente rilasciato nell'ambiente di produzione, apportando modifiche visibili agli utenti finali. Ciò comporta l'eliminazione dell'intervento della componente umana ma potreb-

be comunque essere implementata se il processo è stato padroneggiato e non ci sono ragioni commerciali per mantenerlo in uno stato non automatizzato;

- **Continuous Feedback** : Prevede che i dati e le analisi raccolte negli ultimi due passaggi della pipeline DevOps vengano elaborati automaticamente in modo significativo per il team di sviluppo e inviati ad esso. È il più ambizioso finora, poiché ola raccolta dei feedback non può essere completamente automatizzata a causa della natura imprevedibile degli utenti.

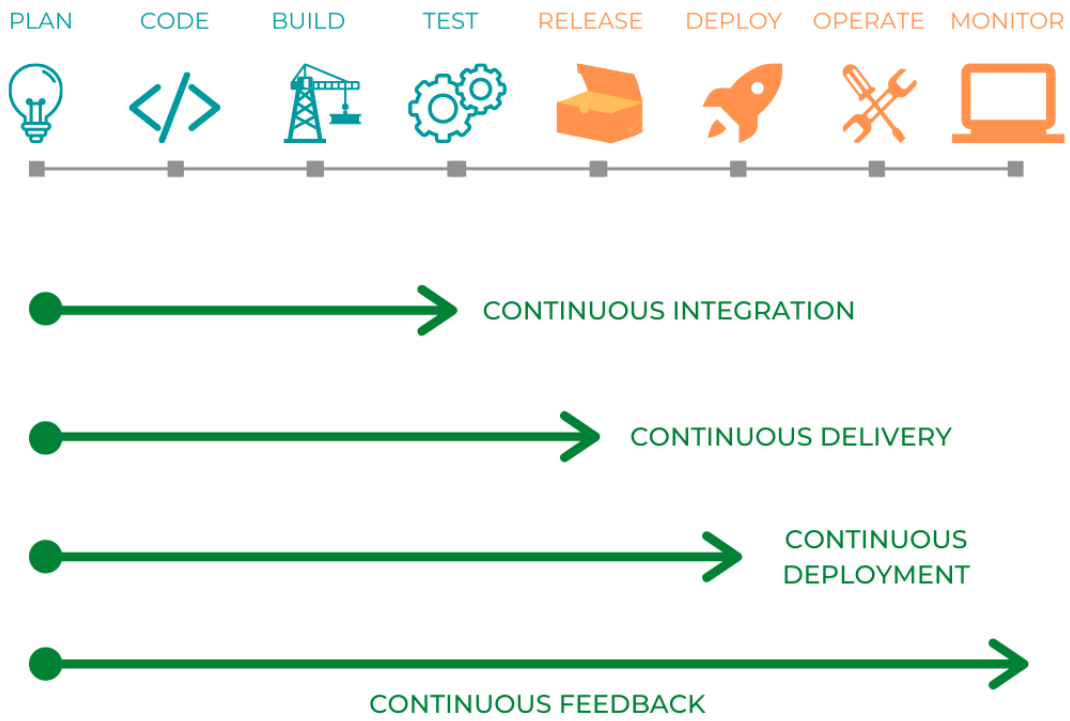

Figura 1.10: DevOps continuous patterns

1.3.2 Differenza tra DevOps ed Agile

Si tende a pensare che Agile e DevOps siano due metodologie distinte, quando in realtà DevOps è un miglioramento recente della metodologia Agile. La principale differenza che DevOps apporta è tra Developer, solitamente orientati esclusivamente alla creazione di cambiamenti e all'aggiunta o alla modifica di funzionalità, e Operation, solitamente orientati allo stabilimento e miglioramento dei servizi, in tutte le fasi del ciclo di vita del prodotto, con l'obbiettivo che questa collaborazione porti ad un enorme valore sia per quanto riguarda la qualità del prodotto finale, sia per la velocità con cui questo viene realizzato. DevOps estende i principi della metodologia Agile che si fondano su valori, metodi e pratiche, concentrandosi sull'intero servizio,ed aggiunge anche gli strumenti. Secondo il Manifesto Agile infatti, gli strumenti assumono una minima importanza, ma questo ha portato nel tempo a trascurarli notevolmente con conseguenze spesso non in linea con l'obiettivo del cliente. Un altro modo di vedere le differenze tra i due metodi è che Agile, risolve i problemi della tecnologia, mentre DevOps cerca di risolvere un problema di business attraverso un insieme di pratiche, cultura e valori che portano ad avere un software più solido e che viene prodotto più velocemente, con meno problemi durante il ciclo di vita. Infatti DevOps accumuna l'obiettivo di Developers e Operators, cioè puntare all'ottimizzazione dell'intero sistema invece che alle ottimizzazioni locali, ciò implica che al raggiungimento degli obbietti di business, sarà risolto il problema del cliente, sia esso un problema di Developer o di Operation.

Stato dell'arte e definizione di una toolchain

In tale capitolo si presterà attenzione al terzo aspetto principale della filosofia DevOps: i **tools** ovvero gli strumenti che aiutano a tradurre l'ideologia in un meccanismo concreto.
Basti pensare che nel corso degli anni, a partire dalla nascita del mercato dell'Integrazione Continua e della Delivery Continua, molti competitors hanno iniziato non solo ad adottare strumenti di terze parti, ma anche a portare avanti le proprie capacità di creazione di tali strumenti.
Pertanto, è iniziata una corsa verso lo sviluppo delle tecnologie DevOps più recenti ed efficienti, scuotendo il mercato e iniettando ancora più dinamicità in un settore già avvincente.
Partendo quindi dal meccanismo principale di DevOps, la pipeline CI/CD, gli innovatori hanno voluto definire strumenti che potessero aiutare a raggiungere una maggiore efficienza in tutte le fasi viste nel capitolo precedente, puntando quindi a gradi più elevati di automazione, ma affrontando anche altri aspetti più umani : come la necessità di una migliore organizzazione del progetto, mezzi di comunicazione più veloci e così via.
Prima di concentrarsi su quali elementi della pipeline DevOps sono diventati i soggetti centrali del mercato e anche su cosa il mercato stesso ha da offrire è opportuno definire il concetto di una delle parole chiave chiave di questo campo, ovvero **toolchain DevOps**.
Come suggerisce lo stesso termine *"toolchain"*, si tratta di una *"combinazione degli strumenti più efficaci per lo sviluppo, la distribuzione e la manutenzione di software secondo principi agili"*. Questa definizione va in parallelo con l'idea della pipeline DevOps: per ogni fase, ci sono uno o più strumenti che possono aiutare a raggiungere i principi DevOps

e che possono essere concatenati per creare un flusso.
Ogni strumento, però, può influenzare anche più di una fase: tutto dipende dall'impatto della corrispondente categoria di appartenenza dello strumento.
Anche se la definizione prima citata può sembrare piuttosto semplice, impostare una vera e propria toolchain non è un compito facile.
Vista l'ampia gamma di strumenti appartenenti a diversi settori, uno dei problemi principali risiede nella compatibilità tra di essi, poiché non tutti possono essere concatenati: gli sviluppatori di strumenti con tale ideologia in mente cercano di rendere il loro prodotto adatto al mercato, ma è non sempre il caso. Un altro aspetto importante è che alcuni prodotti potrebbero essere disponibili a pagamento, oppure presentare diverse versioni low-budget con funzionalità limitate pur mantenendo le principali a prezzo elevato, quindi il costo per adottare tali strumenti e per impostare la toolchain potrebbe in alcuni casi essere piuttosto pertinente. Questi fattori determinano due strategie principali che un'azienda di software può adottare per impostare una toolchain [15]:

- **In-The-Box** : una toolchain di tale categoria è una soluzione creata da un'azienda di terze parti, che fornisce un insieme predefinito di standard e strumenti. Questa strategia permette di garantire la compatibilità degli tools e di risparmiare tempo e fatica, ma a scapito di costi e flessibilità;

- **Custom** : in questo contesto l'azienda seleziona (o costruisce) autonomamente il proprio set di strumenti, consentendo la massima flessibilità e libertà. D'altra parte, questo metodo può richiedere enormi sforzi e investimenti di tempo (soprattutto per le piccole aziende) o addirittura diventare proibitivo in termini di budget se il progetto diventa troppo ambizioso.

Non c'è modo di dire a priori quale sia la strategia più efficiente: ogni azienda dovrebbe decidere in base alle proprie esigenze e al budget a disposizione.
Nel caso di una toolchain Custom DevOps, un ultimo aspetto da considerare è proprio la quantità di strumenti tra cui scegliere: ecco perché è fondamentale avere un'immagine chiara del panorama degli strumenti e delle categorie in cui rientrano.

2.1 Definizione di un toolchain

Qualsiasi sia la categoria della toolchain scelta, essa dovrebbe essere composta da tools che permettano un'ampia gestione ed automatizzazione del progetto dalle sua fase iniziali fino alle operazione di delivery verso gli utenti finali.
Di seguito sono quindi elencati una serie di tools appartenenti a diverse categorie che fanno da padroni in una toolchain.
Tali tools possono essere posti alla base della definizione di una generica toolchain nell'ambito dello sviluppo di un progetto in C++ e saranno descritti con maggior dettaglio nella sezione successiva.

2.1.1 Source Code Management

Poiché nello sviluppo del software il monitoraggio delle modifiche al codice è diventato di alto rilievo, i sistemi *Software di controllo della versione* (VCS) hanno preso man mano una posizione predominante a partire dalla nascita dei processi Waterfall. I principali vantaggi offerti ai team DevOps includono il potenziamento della collaborazione in team, la gestione di più versioni dello stesso progetto, il controllo degli asset e molti altri.
A tal proposito *Azure Repos* è un set di strumenti di controllo della versione che è possibile utilizzare per la gestione di un repository condiviso ed offre sostanzialmente due tipo di VCS :

- **Git** : Git è il server di versionamento del codice sorgente più in diffuso ed è nato da Linus Torvalds, il creatore di linux. E' il sistema più utilizzato in assoluto ed è la base di tutti i più importanti servizi cloud di hosting del codice sorgente. Infatti Git deve la sua popolarità (anche) a GitHub, hosting di progetti software che ha sempre rivolto particolare attenzione alle iniziative open source. GitHub è utilizzato dalle più grandi aziende del pianeta come Google, Microsoft, Apple, Nasa, Facebook e Twitter ed è stato acquisito nel 2018 da Microsoft.

- **Team Foundation Version Control (TFVC)** : TFS (meglio conosciuto come Azure DevOps Service) è un server per la gestione del ciclo di vita del del software a partire dalla raccolta dei requisiti, per passare allo sviluppo del codice, i test, fino alla messa in produzione della funzionalità, per poi continuare con la manutenzione.

La prima versione di TFS rilasciata nel 2006, ha introdotto un nuovo strumento di gestione del versionamento del codice sorgente sviluppato appositamente per questo prodotto: Team Foundation Version Control (TFVC) che rappresenta il prodotto che di fatto ha sostituito quello che era lo strumento Microsoft per gestire il codice sorgente fino a quel momento: Source Safe.

Ciò che porta la maggior parte degli utenti ad usare Git è che il versionamento del codice avviene in locale; il server remoto è utilizzato solo per sincronizzare ed integrare i repository locale gestendo eventuali conflitti. Proprio per questa peculiarità dell'esistenza di un repository locale, la terminologia di Git è molto differente da quella di TFVC (ed egli altri prodotti simili). Infatti TFVC ha un vocabolario in cui le operazioni principali sono due:

- **check out** : preleva l'ultima versione del codice sorgente sul server ed aggiorna il codice locale gestendo eventuali modifiche e notificando possibili conflitti;
- **check in** : preleva le modifiche apportate al codice locale ed invia le modifiche al server che, in assenza di conflitti, sono storicizzate generando quello che è chiamato *change set*

Git presenta invece un vocabolario è più ampio.
Le modifiche sono versionate in locale attraverso un'operazione chiamata *commit* che genera un identificativo univoco del *change set* della modifica. Attraverso poi tale identificativo (il cui formato è un hash) sarà possibile riferirsi e reperire sia in locale che sul server (previa sincronizzazione) il set di modifiche.
Ogni volta che si vuole sincronizzare un repository locale con quello remoto si effettua un'operazione di *push* ed In maniera duale, l'operazione *pull* trasferisce i changeset remoti (non presenti in locale) aggiornando la cartella di lavoro con la situazione corrente. Durante l'operazione di *push* Git è molto restrittivo perché blocca in modo preventivo l'operazione nel caso in cui il proprio codice non è allineato con tutto ciò che c'è sul sul server.

2.1.2 Collaboration

Malgrado il suo semplice concetto, l'implementazione all'interno di una toolchain DevOps di trumenti che facilitino una comunicazione veloce, chiara e personalizzabile è d'obbligo.

Tra quelli di maggior rilievo si colloca sicuramente **Microsof Teams** che rappresenta un sistema di messaggistica per i più svariati tipi di organizzazione: uno spazio di lavoro per collaborazione e comunicazione in tempo reale, riunioni, condivisione di file e app.

2.1.3 Build Automation

Questa categoria può essere considerata una delle più estranee, ma la scelta dello strumento giusto per costruire un progetto è strettamente legato a uno degli obiettivi di DevOps, ovvero il risparmio di tempo. Una volta che un team di sviluppo ha inviato il codice a un sistema di gestione del controllo del codice sorgente, il primo passaggio dell'integrazione continua consiste nella compilazione e nella generazione del codice. Tra i maggiori di rilievo si collocano :

- **CMake** : CMake è uno strumento per la definizione e la gestione di build di codice, principalmente per C++. Esso è uno strumento multipiattaforma : l'idea è di avere una singola definizione di come è costruito il progetto (che si traduce in definizioni di build specifiche per qualsiasi piattaforma supportata) abbinandola a diversi sistemi di costruzione specifici della piattaforma in questione.
  CMake rappresenta quindi un passaggio intermedio che genera input di build per diverse piattaforme specifiche : Su Linux, CMake genera Makefile; su Windows, può generare progetti di Visual Studio e così via. Il comportamento di generazione è definito nei file CMakeLists.txt , uno in ogni directory del codice sorgente e definisce cosa deve fare il buildsystem in quella specifica directory [9].

- **Gradle** : Usato principalmente per progetti Java, Gradle è un sistema di automazione della build completamente open source che espande le idee di Ant e Maven. Esso utilizza un DSL (domain-specific language) basato sul linguaggio di programmazione Groovy, invece di un file XML come faceva Apache, fornendo un modo flessibile, semplice e breve per creare e mantenere lo script di build [12].
  La sua architettura può anche essere definita come basata sulla semplice scrittura di plugins, ma essendo uno strumento recente la loro quantità non è paragonabile a quella offerta da Maven. Con una documentazione ben definita ed esplicita e la sua adozione è iniziata ad essere effettuata da un numero sempre maggiore

di aziende. Ciò è particolarmente vero nel caso di Google, che ha reso Gradle lo strumento di compilazione standard per i progetti Android.

2.1.4 Continuos Integration

Sicuramente la categoria più cruciale, poiché gli strumenti CI uniscono l'intera toolchain DevOps offrendo metodi per orchestrare più fasi in un'istanza unificata. Questi strumenti hanno senza dubbio il maggiore impatto sul mercato, in quanto la concorrenza è aspra e coinvolge anche molte grandi aziende come AWS e Microsoft. Un altro fattore importante è il modo in cui questi strumenti possono far avanzare il flusso continuo: molti possono offrire supporto per ottenere solo l'integrazione continua, mentre altri possono estendere le proprie capacità per supportare fasi come la distribuzione o il monitoraggio. Ecco perché, nell'analisi di questa categoria, è obbligatorio prendere in considerazione ancora di più i fattori precedentemente citati come i costi, la manutenibilità e la compatibilità non solo con le categorie di strumenti storici, ma anche con quelli nuovi ed emergenti.
Tra i maggiori di rilievo troviamo :

- **Azure DevOps** : Azure DevOps rappresenta una combinazione di strumenti e servizi che consentono ai team di pianificare il lavoro, collaborare allo sviluppo del codice, compilare e distribuire applicazioni.È possibile lavorare nel cloud (usando Azure DevOps Services) o in locale (usando Azure DevOps Server).
  Offre quindi la possibilità di usare uno o più dei servizi autonomi in base alle esigenze come **Azure Repos** che fornisce repository Git o controllo della versione di Team Foundation (TFVC) per il controllo del codice sorgente o anche **Azure Boards** che offre una suite di strumenti Agile per supportare la pianificazione e il monitoraggio di lavoro, difetti del codice e problemi usando i metodi Kanban e Scrum piuttosto che **Azure Pipelines** che offre servizi di compilazione e rilascio per supportare l'integrazione e il recapito continui delle applicazioni.
  Quest'ulitmo verrà descritto con maggior dettaglio nella sezione 2.2.5;

- **Jenkins** : Jenkins è uno strumento di CI con una consolidata posizione nel mercato, non solo per essere completamente gratuito e open-source, ma anche per la sua popolarità e la vasta comunità dietro di esso, che ha contribuito a migliorare il capacità di tale prodotto.

Presenta un architettura basata sui plugins sviluppati non solo da utenti indipendenti, ma anche da aziende che offrono ufficialmente l'integrazione con i loro strumenti proprietari.
Altre caratteristiche principali sono: facilità di installazione e aggiornamento su vari sistemi operativi (Windows, macOs, Linux e altri sistemi operativi simili a Unix), supporto di build distribuite con architettura Master-Slave, supporto di esecuzione di comandi shell e Windows, capacità di essere eseguiti all'interno container o come servizio standalone (utilizzando Java Runtime Environment), sistema di comunicazione email integrato, presenza di variabili ambientali e un'interfaccia utente semplice e personalizzabile.

2.2 Realizzazione della toolchain

Chiarito quindi l'obiettivo da raggiungere, in tale sezione verranno quindi descritte le scelte che hanno portato creazione di una toolchain DevOps.
La prima scelta è tra i due principali tipi di toolchain descritti precedentemente : *Custom* o *In-the-box*, con la decisione di adottare la prima per le seguenti ragioni :

- la strategia Custom consente un'introduzione più lenta e graduale degli strumenti all'interno della catena;
- Avere a disposizione un'elevata flessibilità: non solo la toolchain può soddisfare esattamente le esigenze dell'azienda, ma può anche essere ulteriormente migliorata nel corso del tempo;
- il costo: poiché l'azienda intende eseguire un'analisi degli strumenti, preferirebbe evitare di investire denaro in una soluzione che potrebbe non adattarsi alle sue esigenze o ai suoi metodi di lavoro.

Il processo di creazione di una toolchain Custom, tuttavia, richiede un'attenzione particolare alla selezione degli strumenti, poiché la compatibilità risulta essere un punto critico.
Dopo una lunga e dettagliata analisi e considerando quanto detto nella sezione precedente, sono stati infine scelti i tools descritti in seguito.

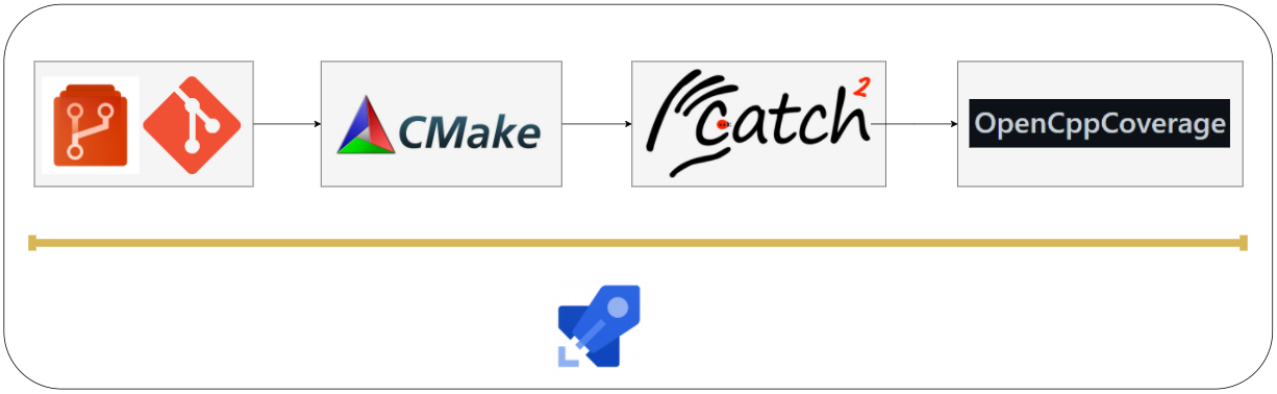

Figura 2.1: Toolchain per MusicTribeJam

2.2.1 Azure Repos : Git

Il primo strumento è ovviamente un Version Control System.
Ogni sviluppatore, dopo aver lavorato in locale sul progetto, invia le proprie modifi-

che al repository remoto, in modo che il server *Continuous Integration* possa agire sulla versione più recente del codice.
Git è stato scelto principalmente per la sua popolarità e la sua integrazione nativa con Azure Repos che ne fornisce un'interfaccia semplice ed intuitiva utilizzabile dall'apposito portale.

2.2.2 CMake

Figura 2.2: Cmake

Il secondo tool che è stato scelto per la toolchain è quello relativo alla categoria delle Build Automation poichè il primo passaggio dell'integrazione continua consiste proprio nella compilazione e nella generazione del codice. CMake (abbreviazione di Cross Platform Make) è un sistema estensibile e open-source che gestisce il processo di compilazione in un sistema operativo in modo indipendente dal compilatore.
A differenza di molti sistemi cross-platform, esso è progettato per essere utilizzato insieme all'ambiente di compilazione nativo. Semplici file di configurazione collocati in ciascuna directory di origine (chiamati file *CMakeLists.txt*) vengono utilizzati per generare file di build standard (ad esempio, makefile su Unix e progetti/aree di lavoro in Windows MSVC).
CMake può generare un ambiente di compilazione nativo che compilerà il codice sorgente, creerà librerie, genererà wrapper e costruirà eseguibili in combinazioni arbitrarie. Esso supporta build *in-place* e *out-of-place* e può quindi supportare più build da un singolo albero di origine.
Un'altra caratteristica interessante di CMake è che genera un file di cache progettato per essere utilizzato con un editor grafico : quando CMake viene eseguito, individua file, librerie ed eseguibili e direttive di compilazione facoltative; queste informazioni vengono raccolte nel file *CMakeCache.txt*, che può essere modificato dall'utente prima

della generazione dei file di build nativi.
CMake è progettato per supportare gerarchie di directory complesse e applicazioni dipendenti da diverse librerie : supporta progetti costituiti da più toolkit in cui ognuno potrebbe contenere diverse directory e l'applicazione dipende dai toolkit e dal codice aggiuntivo.
Poiché CMake è open source e ha un design semplice ed estensibile, può essere esteso secondo necessità per supportare nuove funzionalità : usare CMake è semplice.
Il processo di compilazione è controllato creando uno o più file *CMakeLists.txt* in ciascuna directory (incluse le subdirectory) che compongono un progetto. Ogni *CMakeLists.txt* è costituito da uno o più comandi che hanno la forma *COMMAND(args...)* dove *COMMAND* è il nome del comando e *args* è un elenco di argomenti separato da spazi.

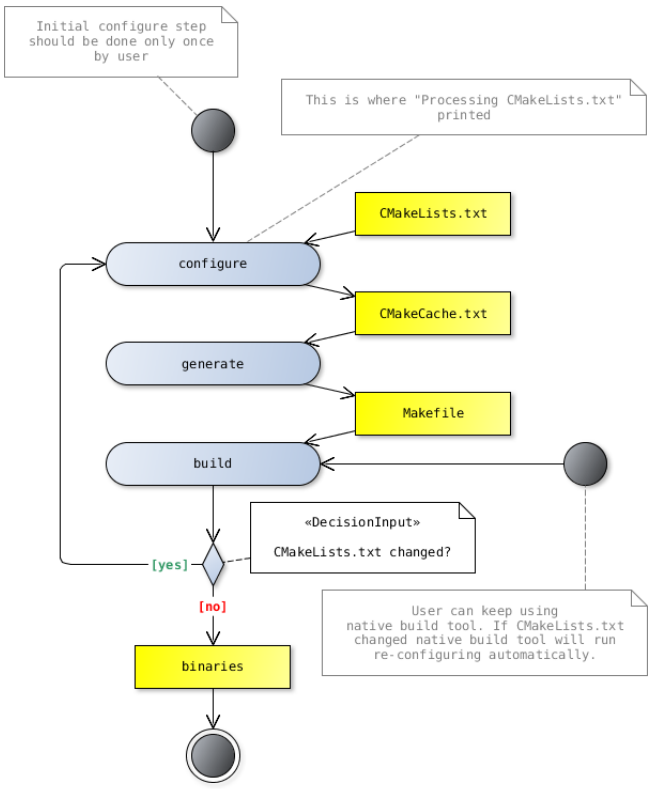

Figura 2.3: Cmake Workflow

Per generare un buildsystem con CMake, è necessario selezionare quanto segue:

- **Source Tree** : La directory di primo livello contenente i file di origine forniti dal progetto. Il progetto specifica quindi il suo sistema di compilazione tramite un

file di primo livello denominato *CMakeLists.txt*.
Questi file specificano le destinazioni di compilazione e le loro dipendenze;

- **Build Tree** : La directory di livello superiore in cui devono essere archiviati i file di buildsystem e gli artefatti di output della build (ad es. eseguibili e librerie). CMake scriverà un file *CMakeCache.txt* per identificare la directory come albero di build e memorizzare informazioni persistenti come le opzioni di configurazione del buildsystem.

- **Generator** : Questo sceglie il tipo di buildsystem da generare (e.g. VisualStudio , XCode , etc...).
  Quando si utilizza uno dei generatori di strumenti di compilazione da riga di comando, CMake prevede che l'ambiente necessario per la toolchain del compilatore sia già configurato nella shell. Quando si utilizza uno dei generatori di strumenti di compilazione IDE, non è necessario alcun ambiente particolare.

2.2.3 Catch2

Figura 2.4: Catch2

Successivamente è oneroso inserire nella toolchain un framework a supporto della scrittura di unit test per incrementare e tenere traccia della qualità del codice sviluppato. Catch2 è principalmente un framework di test delle unità per C ++, ma fornisce anche funzionalità di micro-benchmarking di base e semplici macro BDD.
Il vantaggio principale di Catch2 risiede nel suo è semplice e naturale utilizzo. I test si

autoregistrano e non devono essere nominati con identificatori validi C++, le asserzioni hanno l'aspetto del normale codice C++ e le sezioni forniscono un modo semplice per condividere il codice di *set-up* e *tear-down* nei test.
Ma perchè c'è la necessità di avere un ulteriore framework per i test C++ ? [7]
Considerando che per C++ ci sono un certo numero di framework consolidati, inclusi (ma non solo), Google Test, Boost.Test, CppUnit, Cute e molti... in cosa *Catch2* si differenzia da quest'ultimi ?
Andiamo quindi ad elencare i suoi punti di forza :

- E' molto facile e veloce da utilizzare anche nelle sue fasi iniziali. Basta infatti avere a disposizione appena due file da aggiungere al progetto;
- La sua integrazione con *CMake* è semplice ed intuitiva;
- Nessuna dipendenza esterna finché è possibile compilare con C++14 e si hanno a disposizione le librerie standard C++.
- E' possibile scrivere casi di test come vere e proprie funzioni;
- E' possibile dividere i test cases in sections, ogna delle quali può essere eseguita in isolamento;
- Utilizza sia le sezioni Given-When-Then (in stile BDD) che i casi di test unitari tradizionali;
- Vi è una sola macro di asserzione di base per tutti i confronti. Per il confronto vengono utilizzati operatori C/C++ standard, ma l'intera espressione viene scomposta ed i valori lhs (Left-Hand-Side) e rhs (Right-Hand-Side) vengono registrati;
- I test vengono denominati utilizzando stringhe in formato libero negli identificatori legali.

Altri aspetti importanti possono essere :

- L'output avviene tramite oggetti *reporter* modulari. Sono inclusi reporter testuali e XML di base. È possibile aggiungere facilmente reporter personalizzati;
- L'output XML di JUnit è supportato per l'integrazione con strumenti di terze parti, come i server CI;

- Viene fornita una funzione main() predefinita, ma è possibile fornire la propria per il controllo completo (ad esempio l'integrazione nella propria GUI del test runner);
- Presenta un generatore di dati (implementa di fatto un supporto per il TDD);
- Microbenchmarking support.

Molti sviluppatori stanno volgendo la loro attenzione verso *Catch2*.
Secondo il sondaggio sull'ecosistema C++ Jetbrains del 2021, circa l'11% dei programmatori C++ utilizza Catch2 per i test di unità, rendendolo il secondo framework di test di unità più popolare.

2.2.4 OpenCppCoverage

L'ultimo tool sezionato per la toolchain è relativo alla copertura del codice implementato. La copertura del codice è la percentuale di codice coperta dai test automatizzati.
La misurazione di tale copertura determina essenzialmente quali istruzioni di codice sono state eseguite attraverso un'esecuzione di test e quali no.
Essa fa parte di un ciclo di feedback nel processo di sviluppo : man mano che i test vengono sviluppati, la copertura del codice evidenzia aspetti del codice che potrebbero non essere adeguatamente testati e che richiedono ulteriori test. Questo ciclo continuerà fino a quando la copertura non raggiunge un obiettivo specificato [**AboutCod82**].
In sintesi, la copertura del codice è essenziale per i seguenti motivi :

- Per sapere quanto bene i test implementati testano effettivamente il codice;
- Per sapere se si hanno a disposizione abbastanza test o serve implementarne altri;
- Per mantenere la qualità del test durante il ciclo di vita di un progetto.

OpenCppCoverage è uno strumento di copertura del codice open source per C++.
IL suo utilizzo principale è per la copertura del test di unità, ma puoi anche usarlo per conoscere le righe eseguite in un programma a scopo di debug.
Tra le sue maggiori features troviamo :

- **Non intrusive** : il suo utilizzo risulta essere molto intuitivo : basta eseguire il programma con OpenCppCoverage, senza doverlo ricompilare.
- **Coverage aggregation**: è possibile esegure diverse coperture del codice ed unirle in un unico report.
- **Reporting** : offre la possibilità di ottenere un report in vari formati tra cui *HTML* e *Cobertura*.

2.2.5 Azure Pipelines

L'ultimo passo quindi è quello della configurazione effettiva della pipeline attraverso il portale DevOps.
La pipeline offre servizi di compilazione e rilascio per supportare l'integrazione (CI) e la delivery (CD) continua delle applicazioni : mentre i sistemi CI producono artefatti distribuibili, i processi di CD utilizzano tali elementi per rilasciare nuove versioni e correzioni di essi.
Il punto di partenza per la configurazione di una pipelina CI/CD per le applicazioni è avere il codice sorgente in un sistema di controllo della versione. Successivamente viene definita la vera e propria pipeline.
Le pipeline vengono definite in un file YAML solitamente denominato come "azure-pipelines.yaml".[3]

Figura 2.5: Azure pipelines

Di base è quindi possibile seguire la seguente procedura per l'uso di una pipeline :

1. Configurazione di Azure Pipelines per l'uso del repository Git.
2. Definizione e configurazione del file azure-pipelines.yml file per definirne il comportamento.

3. Eseguire il push del codice nel repository. Questa azione attiva il trigger predefinito di compilazione e distribuzione monitorandone i risultati.

2.2.5.1 Concetti chiave

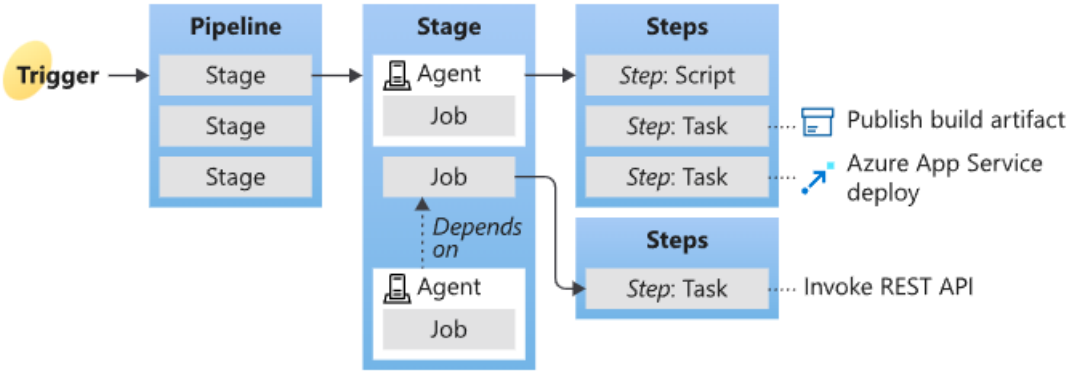

Figura 2.6: Azure pipelines concetti chiave

Di seguito vengono riportati quelli che sono i concetti chiave che riguardano il processo di creazione e gestione delle pipeline:

- **Trigger** : Un trigger è un elemento configurato per indicare alla pipeline quando essere eseguita. È possibile configurare una pipeline per l'esecuzione su un push in un repository, in orari pianificati o al completamento di un'altra compilazione. Tutte queste azioni sono note come trigger.

- **Pipeline** : Una pipeline definisce il processo di integrazione e distribuzione continuo per il progetto in questione.
  È costituita da una o più *fasi* e può essere pensato come un flusso di lavoro che definisce come vengono eseguiti i passaggi di test, compilazione e distribuzione.

- **Fase** : Una fase è un limite logico nella pipeline. Può essere usato per contrassegnare la separazione dei problemi, ad esempio compilazione, controllo di qualità e produzione. Ogni fase contiene uno o più *processi*. Quando si definiscono più fasi in una pipeline, per impostazione predefinita vengono eseguite una dopo l'altra in sequenza.

- **Processo** : Ogni processo viene eseguito in un *agente*. Un processo rappresenta un limite di esecuzione di un set di *passaggi* (che rappresentano il blocco atomico di una pipeline). Tutti i passaggi vengono eseguiti insieme nello stesso agente.

- **Agente** : Quando si esegue la compilazione o la distribuzione, il sistema avvia uno o più processi. Un agente è un entità che rappresenta l'infrastruttura sulla quale verranno eseguiti i processi della pipeline. Gli angenti possono essere ospitati da Microsoft oppure self-hosted.

Caso di studio

Il caso di studio in osservazione sarà l'applicazione della metodologia DevOps in contensto aziendale reale : **Microbees srl**.
L'idea principale che ha posto le basi di tale progetto è stata quella di implementare un meccanismo di Continuos Integration che permettesse quindi un flusso di build/-testing continuo per uno dei prodotti più ambiziosi del cliente richiedente, rappresentato da un grande player europeo in ambito musicale.
Il progetto consiste nella creazione di una DAW (Digital Audio Workstation) attraverso la quale gli utenti potranno creare/modificare tracce audio/Midi interfacciandosi con apparati esterni quali :

- Controller Audio e/o Midi
- Periferiche audio generiche (cuffie, casse ec...)
- Mixer e altro...

Tale sistema è rivolto al mercato degli applicativi desktop per vari sistemi operativi quali Linux, Windows e MacOs con futura estensione anche verso i sistemi operativi mobile come Andorid ed IOs : per il suo sviluppo è stato usato il C++ (17) associato ad un framework dedicato denominato JUCE che, oltre a fornire un motore audio e gestire la costruzione di UI molto complesse e reattive, permette la compilazione dello stesso codice su più piattaforme.
La scelta del C++ rispetto ad un altro linguaggio di programmazione (e.g. Java, C#, ec...) è motivata da vari fattori :

- La presenza di librerie e framework a supporto per l'audio programming più efficienti e dettagliate;
- Si tratta di un linguaggio di programmazione molto efficiente in termini di consumo di risorse e velocità (e.g. Java pecca di prestazioni rispetto al C++ a causa della virtual machine);

- Il C++ risulta avere prestazioni maggiori in ambito Multithreading.
- C++ è ottimo per la programmazione a livello di sistema perché consente al programmatore di effettuare chiamate dirette alle librerie di sistema native.

Pertanto, dopo aver definito un quadro d'insieme dell'applicazione, potrebbe sorgere una domanda: perché Microbees è interessata ad introdurre una soluzione DevOps?
Un primo aspetto aspetto importante è che la società non ha la proprietà del dipartimento Operations, che invece appartiene alla società cliente : ciò è dovuto alla natura commerciale della collaborazione tra le due società, in quanto Microbees sviluppa il codice e fornisce consulenza il cliente, ma alla fine sarà il cliente stesso a decidere quando distribuire l'app in base alla sua pianificazione aziendale.
Quindi, uno dei motivi principali per implementare una toolchain di Continuous Integration sarebbe quello di fornire build, che soddisfino gli standard di qualità, in modo automatizzato al cliente ottenendo il chiaro vantaggio di ridurre il gap tra le due società a cui appartengono i dipartimenti Sviluppo e Operations. Questo fattore è ancora più cruciale nella pratica a causa del contesto, in quanto vi è la necessità di una collaborazione non tra due team della stessa azienda, ma tra team appartenenti a società diverse.
Ulteriore scopo di tale progetto è vedere se il set di strumenti scelto si rivelerà utile alle esigenze dell'azienda e se sarà possibile migliorarlo e/o estenderlo applicandolo anche ad altri progetti futuri.
Pertanto, l'azienda si impegna a migliorare l'adozione di DevOps a partire dalla creazione di una toolchain.

3.1 Analisi dei requisiti

In questa sezione si andrà a descrivere con maggiore precisione l'analisi dei requisiti che è stata effettuata a valle dei vari incontri con il cliente con lo scopo di definire quali sono le milestones ed i vari artefatti che dovranno essere man mano prodotti e convalidati secondo una metologia agile. A tale scopo è stata usata la board di azure DevOps per tener traccia dei vari sprints da effettuare.
In primo luogo sono state decise quali dovrebbero essere le *epics* del progetto, ovvero

le macro user stories da consegnare in modalità iterativa lungo i vari sprints.

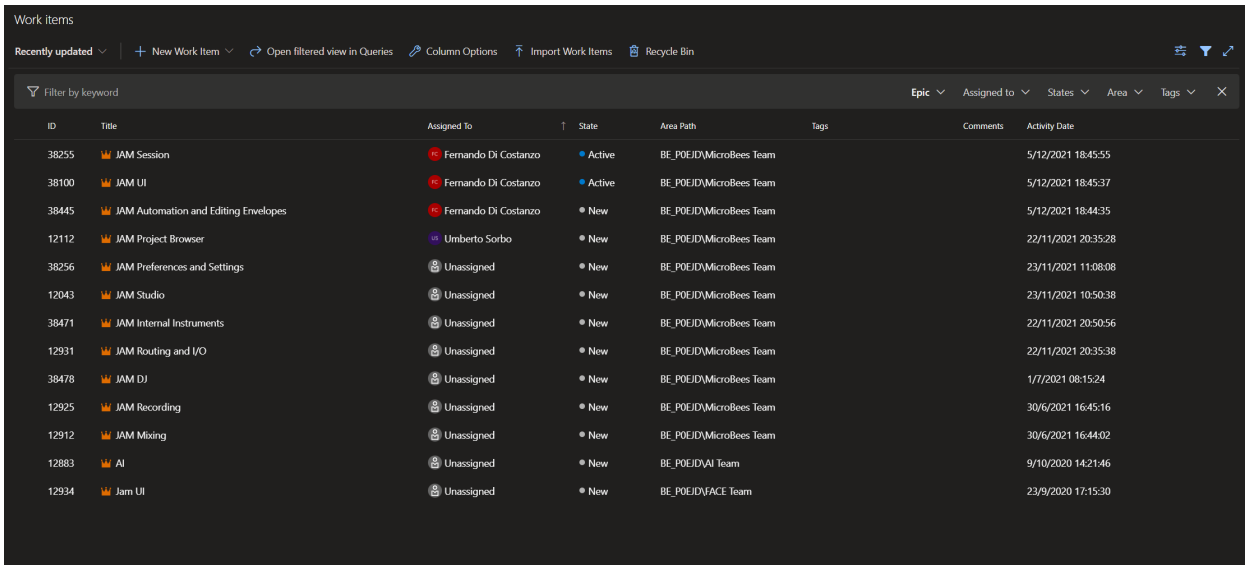

Figura 3.1: Azure boards epics

Da qui sono state assegnate ai vari membri del team che terranno conto di aggiornare il proprio stato in modo coerente con i proprio avanzamenti.
E' possibile per ogni work item (nel caso in questione le epics) andare a configurare un set di specifiche come il membro del team al quale l'item è stato assegnato , lo stato , la descrizione ed altri caratteristiche come mostrato in figura :

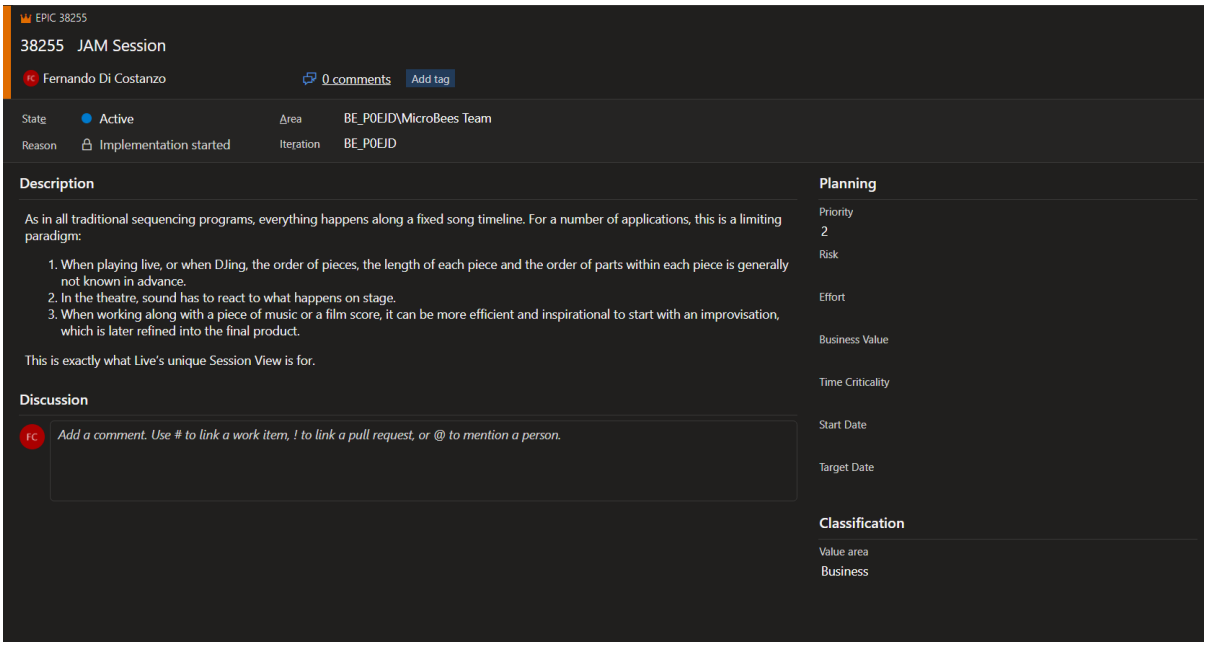

Figura 3.2: Azure boards epics configuration

Successivamente quindi sono state definite le features per ogni epic con le relative user storie ed i relativi task/bug.
A tale scopo andiamo a mostrare come è stato popolato il backlog al termine della pianificazione :

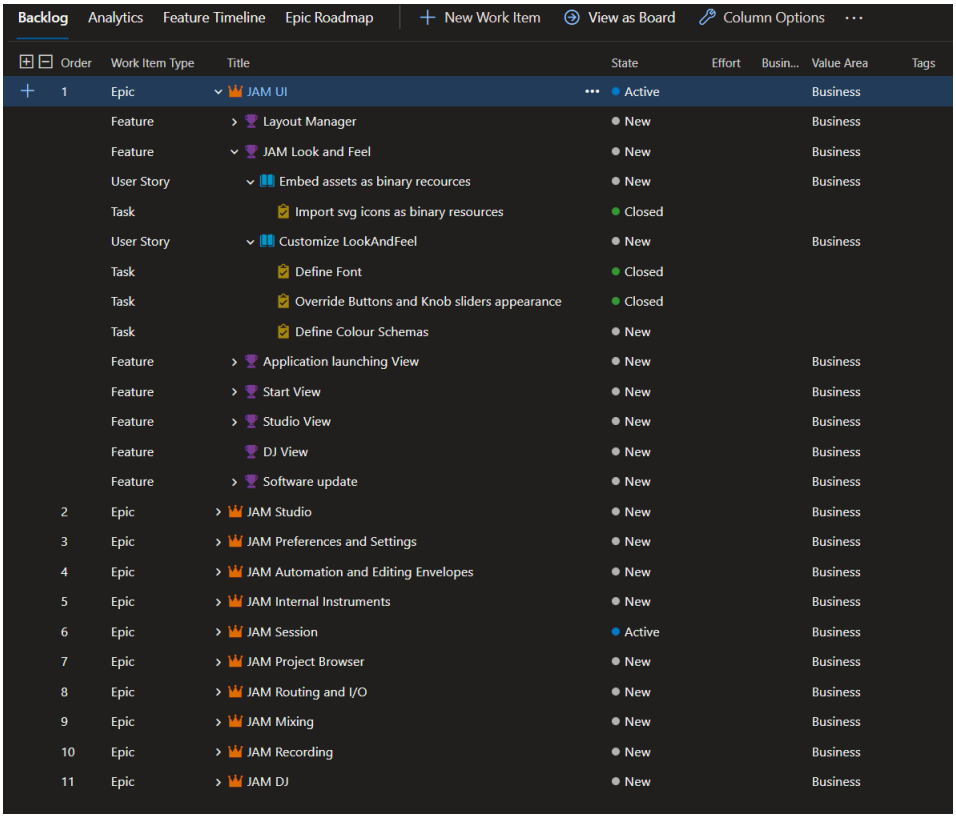

Figura 3.3: Azure backlog

Come è possibile notare dall'immagine 3.3 il risultato è la creazione di una struttura ad albero dove i primi nodi foglia sono le epics e successiamente features,user storie e task/bug.
Questa struttura rende le varie operazioni di creazione, modifica e cancellazione piuttosto che di monitoraggio e specificazione molto flessibili e robustre.
Mentre con il backlog si ha una visione generale del progetto che vede la risoluzione di tutte le user stories nel corso dei vari spint , nella sezione **Sprints** abbiamo invece una visualizzazione più compatta che permette di osservare quali solo le user stories ed i relativi tasks assegnati ai vari membri del gruppo relativi ad un determinato spint.
Nell'immagine seguente viene quindi mostrato l'aspetto di tale schermata ed i task relativi allo sprint denominato *Rel1*

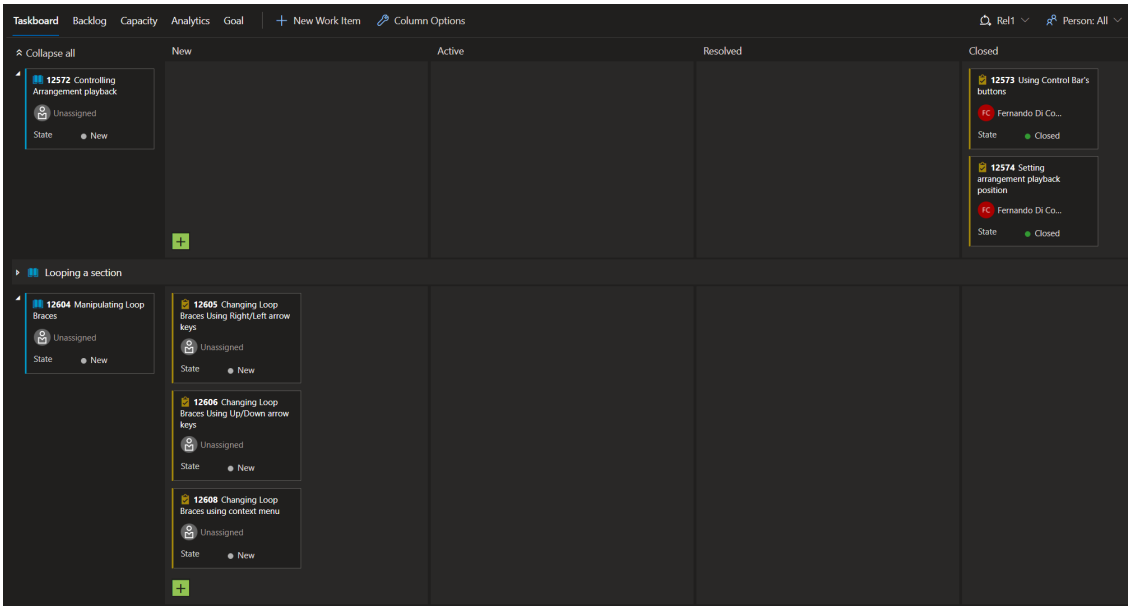

Figura 3.4: Azure Sprint

Si noti come la struttura permetta facilmente di filtrare in base a specifici criteri i task ed intuitivamente cambiarne lo stato (trascinando il task in questione nei corrispettivi riquadri) coerentemente con gli avanzamenti.
Lo stato *Resolved* è dedicato a quei work items categorizzati come bugs pertando un task non potrà mai trovarsi in tale stato.

3.2 Creazione della Toolchain mediante pipeline

In tale sezione andremo quindi a descrivere le varie fasi che hanno portato alla creazione della pipeline CI per il build,test e code coverage automatico.

```
1  trigger:
2  - master
3
4  strategy:
5     matrix:
6      windows_2019:
7          imageName: 'windows-lastest'
8
9  pool:
10   vmImage: $(imageName)
11
```

```
steps:
  - checkout: self
    submodules: recursive

  - task: CMake@1
    displayName: 'CMake ..'
    inputs:
      workingDirectory: 'cmake-build'
      cmakeArgs: '.. -D BUILD_UNIT_TESTS=ON'

  - task: CMake@1
    displayName: 'CMake --build . --config Debug'
    inputs:
      workingDirectory: 'cmake-build'
      cmakeArgs: '--build . --config Debug'

  - task: BatchScript@1
    displayName: 'Executing Catch2 Unit Tests...'
    inputs:
      filename: 'cmake-build\tests_catch\Catch2_UnitTestRunner_artefacts\
    Debug\Catch2_UnitTestRunner.exe'
      arguments: '-r junit > $(System.DefaultWorkingDirectory)\report.xml'
      modifyEnvironment: true
    enabled: true
    continueOnError: true


  - task: PublishTestResults@2
    inputs:
      testResultsFormat: 'JUnit'
      testResultsFiles: '$(System.DefaultWorkingDirectory)\report.xml'
      failTaskOnFailedTests: false
    displayName: 'Publishing test results...'


  - task: BatchScript@1
    displayName: 'Getting code coverage...'
    inputs:
      filename: '3rdparty\OpenCppCoverage\OpenCppCoverage.exe'
      arguments: '--sources=modules\dawtribe_components
```

```
    --export_type=cobertura:$(System.DefaultWorkingDirectory)\codeCovOut\
  cobertura.xml
    cmake-build\tests_catch\Catch2_UnitTestRunner_artefacts\Debug\
  Catch2_UnitTestRunner.exe '
    modifyEnvironment: true
  enabled: true
  continueOnError: true

- task: PublishCodeCoverageResults@1
  inputs:
    codeCoverageTool: 'Cobertura'
    summaryFileLocation: $(System.DefaultWorkingDirectory)\codeCovOut\
  cobertura.xml
```

Con riferimento allo snippet di codice sopra riportato :

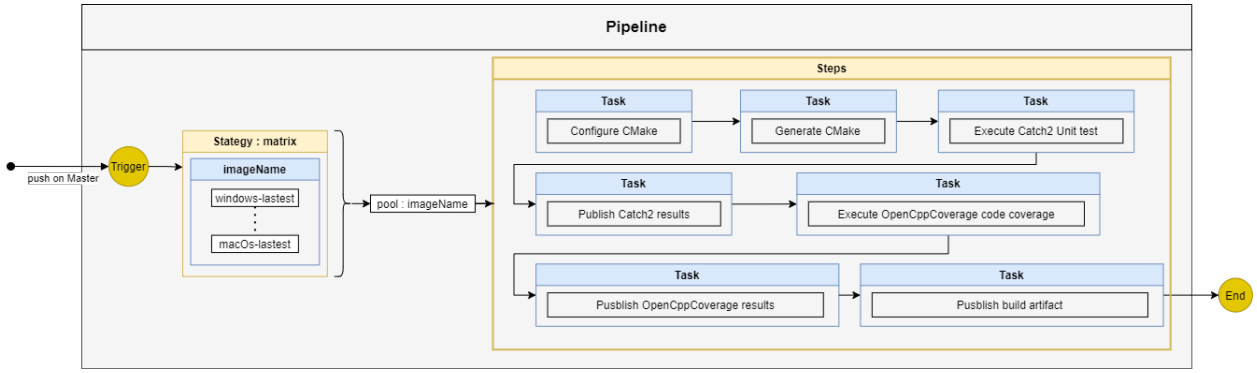

Figura 3.5: Schema a blocchi pipeline

3.2.1 Trigger

Un trigger è un elemento configurato per indicare alla pipeline quando essere eseguita. In questo caso quindi il trigger è configurato sul *master* indicando che la pipeline viene eseguita ogni qual volta viene effettuato un push su tale ramo.
Tale scelta è stata intrapresa considerando che il progetto è nelle fasi iniziali e quindi la CI impostata su ogni push verso il repository può aiutare a risolvere possibili bug tempestivamente.

3.2.2 Strategy

Indica la strategia di esecuzione della pipeline che può essere **matrix** o **parallel**.
L'uso di una matrice genera copie di un processo, ognuna con input diverso , utili per il test su configurazioni o versioni di piattaforme diverse diverse.
Nel caso in questione è stata specificata una sola copia del processo corrispondente alla sua esecuzione tramite un agente che ospita l'immagine *windows-2019*. Negli sviluppi futuri saranno considerate anche le immagini *macOs-lastest* ed *ubuntu-lastest*.

3.2.3 Pool

Invece di gestire ogni agente individualmente essi vengono organizzati in pool che hanno come ambito l'intera organizzazione; in modo da poter condividere le macchine agente tra i vari progetti.
L'esecuzione di una pipeline avviene su un agente del pool specificato che soddisfa le richieste della pipeline in questione. Di base Azure Pipelines opsita un Pool con varie immagini di Windows, Linux e macOS che presentano già CMake installato per impostazione predefinita e quindi non è necessario includere una richiesta per l'installazione di quest'ultimo nel file di configurazione della pipeline.

3.2.4 Steps

Uno *step* è una sequenza lineare di operazioni che compongono un *job*.
Ogni step viene eseguito nel proprio processo su un agente e ha accesso all'area di lavoro della pipeline su un disco rigido locale. Questo comportamento significa che le variabili di ambiente non vengono conservate tra i passaggi, ma le modifiche al file system lo sono.
Gli step sono composti da uno o più *task* che rappresentano gli elementi costitutivi basici di una pipeline. E' possibile configurare un task *from scratch* oppure usando il *wizard assistent* di Azure che fornisce una lista di task configurabili attraverso un'interfaccia più intuitiva.

Ogni step porta con se alcune informazioni indipendenti dalla sua natura come per esempio :

- **displayName** : nome descrittivo visualizzato nell'interfaccia utente;
- **name** : identificatore per questo step;
- **continueOnError** : 'true' se gli step futuri dovrebbero essere eseguiti anche se questo non riesce (il valore predefinito è "false");
- **enabled** : se eseguire questo step; il valore predefinito è "vero"
- **env** : elenco di variabili di ambiente da aggiungere;

3.2.5 Tasks

I task, come detto in precedenza, rappresentano gli elementi costitutivi basici di una pipeline.
L'obbiettivo di tale pipeline è quello di effettuare una CI che preveda build/testing/-code coverage continuo ad ogni push sul repository. Per raggiungere tale scopo sono stati implementati quindi i seguenti task :

3.2.5.1 CMake configuration

```
- task: CMake@1
    displayName: 'CMake ..'
    inputs:
      workingDirectory: 'cmake-build'
      cmakeArgs: '.. -D BUILD_UNIT_TESTS=ON'
```

Permette di effettuare una build del progetto con il sistema di compilazione multipiattaforma CMake. Esso presenta due argomenti opzionali :

- **Working Directory** : rappresenta la directory di lavoro quando viene eseguito CMake.
  Nel caso in questione è stata selezionata la directory "cmake-build" mentre il valore predefinito è "build";

- **Arguments** : rappresentano gli argomenti che vengono passati a CMake.
  Nel caso in questione come argomenti sono stati passati gli argomenti : "*..*" che indica la *source tree* del progetto in cui è presente il file *CMakeLists.txt* dal quale CMake attingerà le informazione per la creazione del *build tree* (nella directory specificata "cmake-build") che conterrà quindi il file *CMakeCache.txt* attraverso il quale successivamente si potrà effettuare il vero e proprio build del progetto; *-D BUILD_UNIT_TESTS=ON* che abilita la costruzione del progetto di test nel *build tree* creato.

3.2.5.2 CMake build

```
- task: CMake@1
   displayName: 'CMake --build . --config Debug'
   inputs:
     workingDirectory: 'cmake-build'
     cmakeArgs: '--build . --config Debug'
```

Viene quindi configurato un altro task CMake che a partire dal *build tree* generato dal precedente effettua la vera e propria build (attingendo dal *CMakeCache.txt*) in configurazione di **Debug** producendo oltretutto gli eseguibili dei vari target tra cui quello relativo al progetto di test con Catch2.

3.2.5.3 Catch2 Unit test

```
- task: BatchScript@1
   displayName: 'Executing Catch2 Unit Tests...'
   inputs:
     filename: 'cmake-build\tests_catch\Catch2_UnitTestRunner_artefacts\
  Debug\Catch2_UnitTestRunner.exe'
     arguments: '-r junit > $(System.DefaultWorkingDirectory)\report.xml'
     modifyEnvironment: true
   enabled: true
   continueOnError: true
```

A tal punto viene invocato un task di tipo *BatchScript* che permette eseguire uno script Windows *.bat* o *cmd*. Per tale task sono stati configurati i seguenti argomenti :

- **filename** : rappresenta il percorso (completo o relativo) dello script cmd o bat da eseguire.
  Come detto in precedenza, attraverso l'operazione di build dei target in configurazione di *Debug*, sono stati generati gli eseguibili a loro associati tra cui anche quello relativo al progetto di test.
  Possiamo quindi lanciare tale eseguibile (che ricordiamo essere un'applicazione hostata dal framework di Catch) con l'argomento

  -r junit > $(System.DefaultWorkingDirectory)\report.xml

  di modo che l'output di tale esecuzione è un file con estenzione *.xml* che contiene i suoi risultati in un formato *JUnit* che è supportato per l'integrazione con strumenti di terze parti, come i server CI di Azure.

3.2.5.4 Catch2 result publisher

```
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'JUnit'
    testResultsFiles: '$(System.DefaultWorkingDirectory)\report.xml'
    failTaskOnFailedTests: false
  displayName: 'Publishing test results...'
```

A valle dell'esecuzione dei test e della generazione di un output in formato *JUnit* è possibile lanciare un task di tipo *PublishTestResults* che pubblica i risultati dei test in Azure Pipelines per fornire un'esperienza di analisi e reportistica completa.
I risultati pubblicati vengono visualizzati nella scheda *Test* nel riepilogo della pipeline e consentono di misurare la qualità della pipeline, esaminare la tracciabilità e risolvere i problemi.

3.2.5.5 OpenCppCoverage code coverage

```
- task: BatchScript@1
  displayName: 'Getting code coverage...'
  inputs:
    filename: '3rdparty\OpenCppCoverage\OpenCppCoverage.exe'
    arguments: '--sources=modules\dawtribe_components
    --export_type=cobertura:$(System.DefaultWorkingDirectory)\codeCovOut\
  cobertura.xml
    cmake-build\tests_catch\Catch2_UnitTestRunner_artefacts\Debug\
  Catch2_UnitTestRunner.exe '
    modifyEnvironment: true
  enabled: true
  continueOnError: true
```

Il penultimo task di tipo *BatchScript* prevede l'esecuzione del tool *OpenCppCoverage* che, a partire dall'eseguibile del progetto di test con Catch2, effettua la code coverage valutando quindi quante righe di codice sono state coperte durante l'esecuzione delle test suite.
Per eseguire il tool è necessario lanciare il suo eseguibile con i seguenti argomenti :

1. **sources** : *'modules\dawtribe_components'* rappresenta la directory contente i file sulla quale la code coverage deve essere effettuata;
2. **export_type** : *'cobertura:$(System.DefaultWorkingDirectory).xml'* rappresenta il tipo di formato per la rappresentazione dei risultati; l'output di tale esecuzione è quindi un file con estensione *.xml* che contiene i suoi risultati in un formato *Cobertura* che è supportato per l'integrazione con strumenti di terze parti, come i server CI di Azure;
3. **target** : *cmake-build\tests_catch\Catch2_UnitTestRunner_artefacts\Debug2_UnitTestRunner.exe* rappresenta l'eseguibile Catch2 tramite a partire dal quale OpenCppCoverage effettuerà la copertura del codice.

3.2.5.6 OpenCppCoverage result publisher

```
- task: PublishCodeCoverageResults@1
    inputs:
      codeCoverageTool: 'Cobertura'
      summaryFileLocation: $(System.DefaultWorkingDirectory)\codeCovOut\
   cobertura.xml
```

A valle dell'esecuzione della code coverage e della generazione di un output in formato *Cobertura* è possibile lanciare un task di tipo *PublishCodeCoverageResults* che pubblica i suoi risultati visualizzabili nella scheda *Code Coverage* nel riepilogo della pipeline consentendo di misurare costantemente la qualità della pipeline.

Validazione e risultati

4.1 Analisi metrica

Con la creazione di una pipeline CI, il divario tra i dipartimenti di sviluppo e operation sarà sicuramente ridotto.
Tuttavia, visti i vantaggi solo da un punto di vista teorico, è ora il momento di analizzare nel dettaglio quali benefici si sono ottenuti anche da un punto di vista pratico.
Storicamente, le metriche tradizionali hanno sempre sottolineato l'importanza della misurazione come strumento per monitorare i progressi e identificare lo stato attuale di un progetto o di un particolare processo. Riferendosi a un argomento ampio e complesso come il DevOps, tuttavia, non esiste una metrica unica e universale che esista come unico indicatore di successo: è importante analizzare l'intero processo e trovare quali metriche si adattano meglio alla sua analisi. A tal fine è obbligatorio definire prima schematicamente l'intero flusso di integrazione che si sta effettuando, come visibile nella figura 4.1.

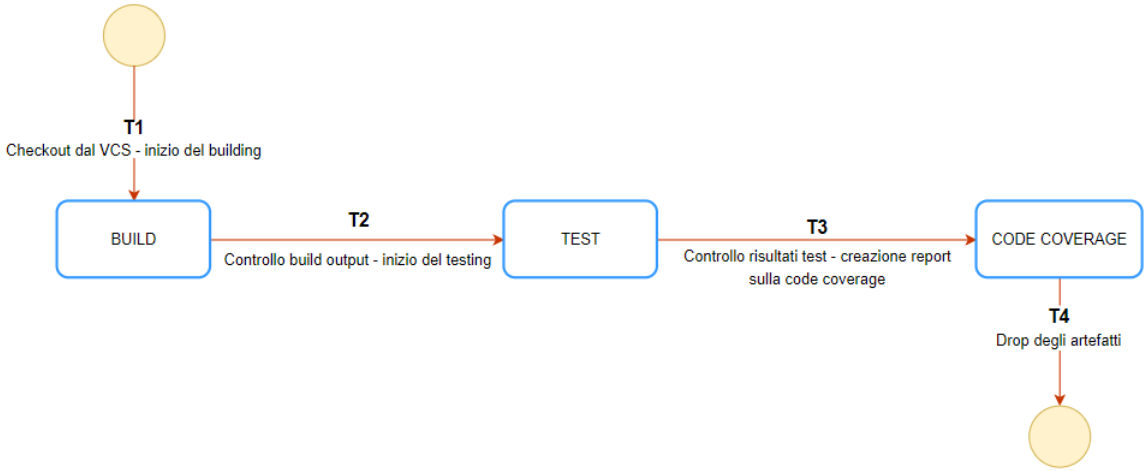

Figura 4.1: Pipeline flow

È possibile evidenziare che tale flusso si riferisce principalmente al lato Development, che cerca di migliorare le proprie metodologie al fine di fornire rapidamente prodotti di qualità superiore al lato Operations.
Nel caso di studio attuale, senza prendere in considerazione la pipeline implementata e concentrandosi esclusivamente sul flusso tradizionale, è possibile rilevare due tipi di attività che vengono eseguite: quelle **automatizzate** (descritte all'interno dei quadrati azzurri), come Build, Test e calcolo della copertura del codice che vengono eseguiti dal compilatore e **manuali** (descritti tramite le frecce arancioni), che consistono principalmente in revisioni di output e input manuali che avviano l'esecuzione delle seguenti attività automatizzate.

Considerando quest'ultime, che vengono svolte da dipendenti dell'azienda, è possibile pensare a due metriche tradizionali, che possono aiutare ad analizzare l'impatto dell'esecuzione manuale di queste attività:

- **Calendar Time (T)** : questa metrica esprime la durata del progetto o di una delle sue attività sotto forma di ore, giorni, settimane, mesi e così via. Può essere definito in forma relativa (ad esempio "Mese1" dall'inizio del progetto) o in forma assoluta (ad esempio "12 settembre").

- **Effort (E)** : esprime il tempo impiegato dai dipendenti a lavorare su un progetto per completare un'attività. Dipende dal *Calendar Time* e dalle persone impiegate nell'azienda. Viene misurato in *persona/ora* dallo standard *IEEE 1045*.
  Questa metrica si traduce direttamente in **Costo**.

Per ogni attività manuale l'azienda ha effettuato delle stime [1] in merito al tempo necessario per essere eseguita e allo sforzo che doveva essere investito in esse considerando che tutti i dipendenti coinvolti hanno pari peso in termini di costo e produttività :

- **T1** e **T4** : poiché non ci sono revisioni coinvolte, il tempo e gli sforzi coinvolti non sono rilevanti.

- **T2** : sono stati stimati i seguenti valori :

  - **Calendar Time** : T = 1h

[1] Le stime sono state effettuate monitorando i tempi e gli sforzi necessari per le revisioni in un regime pre-toolchain

  - **Effort** : E = 1 persona per 30m = 0.5 person/hour

- **T3** : sono stati stimati i seguenti valori :
  - **Calendar Time** : T = 2h
  - **Effort** : E = 1 persona per 50m = 0.83 person/hour

Definiamo quindi il tempo totale speso per i task manuali per ogni build come **Cumulative Calendar Time (CT)** :

$$CT = \sum_i T_i = T_2 + T_3 = 1h + 2h = 3h$$

e lo sforzo totale speso per i task manuali per ogni build come **Cumulative Effort (CE)**:

$$CE = \sum_i E_i = E_2 + E_3 = 0.5 + 0.83 = 1.33person/hour$$

Prendendo adesso in considerazione la pipeline di CI precedentemente introdotta, mentre le attività automatizzate sono ancora presenti, quelle manuali non vengono più considerate: l'utilizzo di Azure garantisce che il processo risulti affidabile, quindi non c'è più bisogno di revisioni costanti manuali per quanto riguarda l'output di build o test ma sarà il sistema a notificare i responsabili in caso di anomalie.
Pertanto, implementando la pipeline di CI, l'azienda ottiene un profitto giornaliero in entrambe le metriche.
Tuttavia, per fare un confronto adeguato, bisogna considerare il tempo e gli sforzi che sono stati investiti per rendere la toolchain pienamente attiva e per implementare il meccanismo di CI.
Pertanto, nuovi valori devono essere considerati:

- **TimeForToolchain (TT)** = 5 ore per 15 giorni lavorativi = 75 hours
- **EffortForToolchain (ET)** = 1 persona per 75 ore = 75 person-hour [2]

[2]Tali valori, una volta acquisito il know-how nella creazione di una toolchain, potrebbero variare per progetti futuri.

All'inizio del progetto, quando la pipeline non era ancora realizzata, questi valori rappresentavano perdite iniziali per la società.
Tuttavia, quando il meccanismo è stato completamente implementato, tempo e sforzo hanno iniziato ad essere guadagnati quotidianamente.

Assumendo una cumulazione giornaliera in termini di *revisioni manuali* e proiettando i rispettivi confronti nel corso del tempo è possibile ottenere, per ognuno, un punto di intersezione.
Tale punto rappresenta di fatto il valore temporale, a partire dall'immissione della toolchain nel progetto, dopo il quale tempi e sforzi rappresentano un effettivo guadagno :

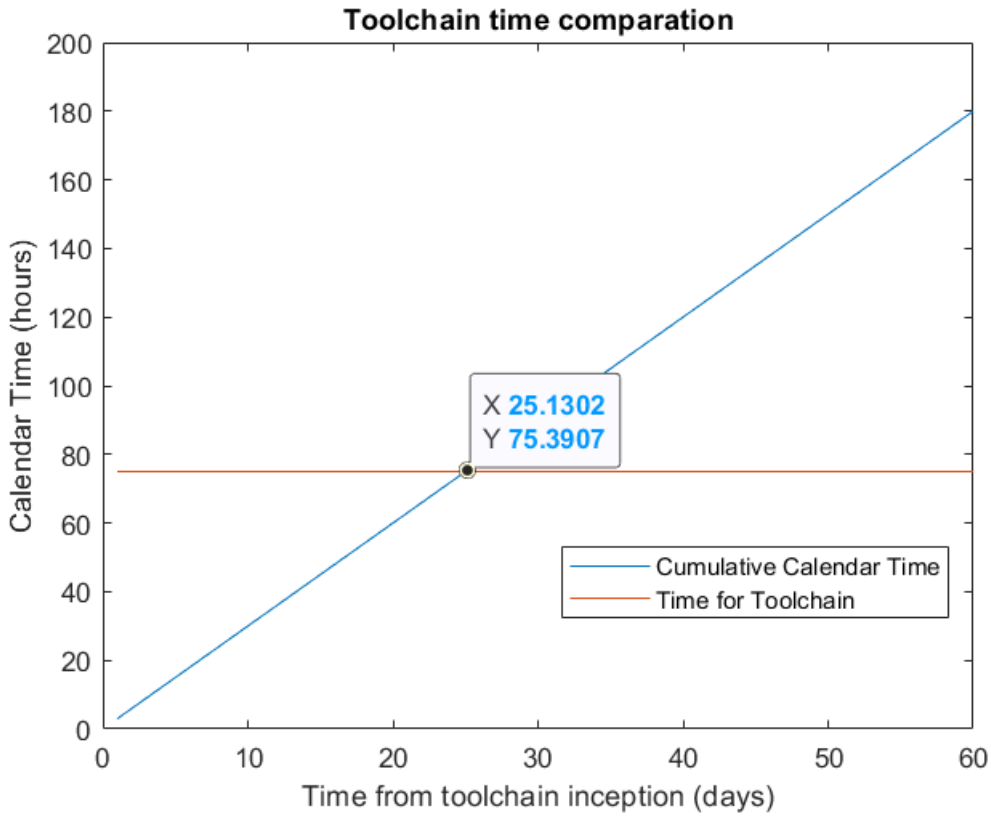

Figura 4.2: Toolchain time comparation

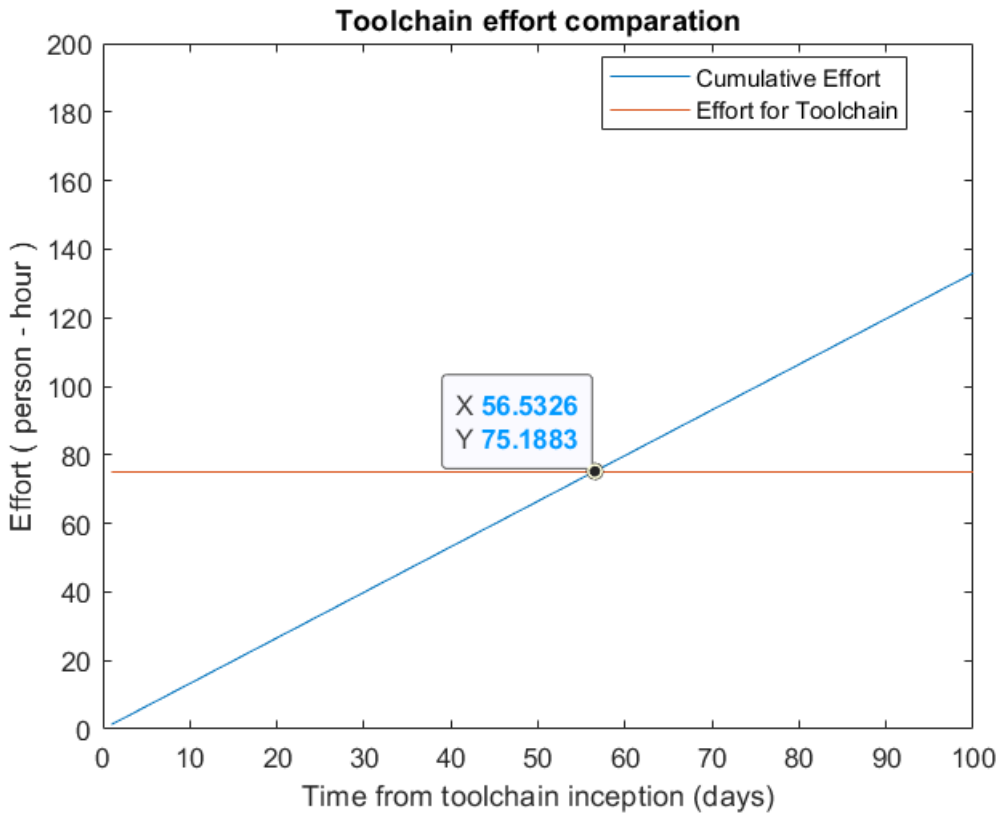

Figura 4.3: Toolchain effort comparation

Attraverso i grafici in figura 4.2 e 4.3 è possibile osservare che l'azienda ottiene un profitto dal punto di vista dell'**Effort** a partire dal 57st giorno dall'implementazione della pipeline di CI, mentre guadagna un profitto dal punto di vista del **Calendar Time** già a partire dal 26st giorno.

4.2 Analisi risultati globali

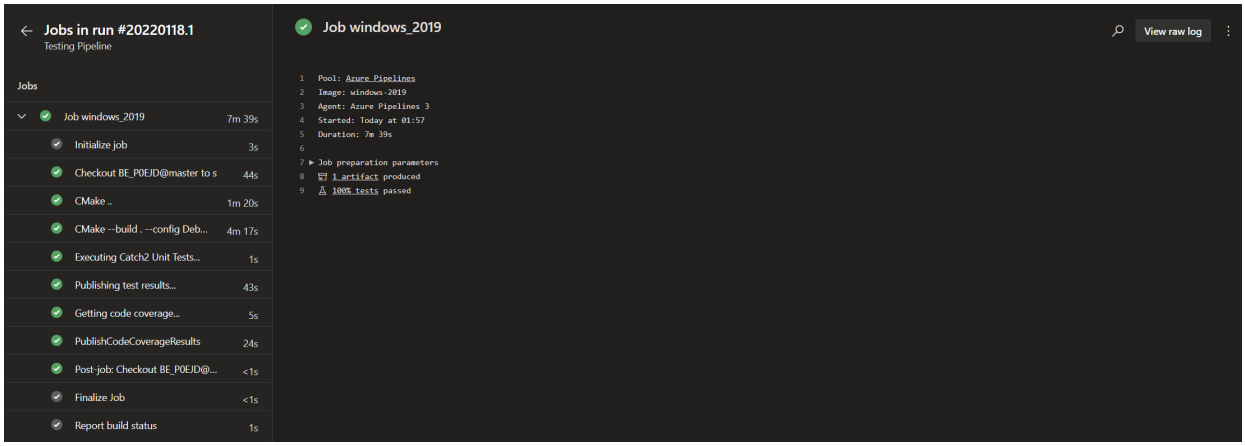

Figura 4.4: Pipeline log

Durante l'esecuzione della pipeline, Azure permette agli utenti di visualizzare i *jobs* in esecuzione arricchiti con vari dati che determinano il log complessivo della pipeline. Quest'ultimi offrono un potente strumento per determinare la causa degli errori della pipeline fornendo un punto di partenza per la loro risoluzione. Oltre alla loro visualizzazione nel riepilogo della compilazione della pipeline, è possibile scaricare i log completi che includono informazioni di diagnostica aggiuntive.
E' inoltre possibile configurare log più dettagliati per facilitare la risoluzione dei problemi.
L'origine delle informazioni per l'analisi della pipeline è il suo set di esecuzioni. Queste analisi vengono accumulate in un periodo di tempo e formano la base delle informazioni dettagliate offerte. Pipelines report mostrano metriche, tendenze e consentono di identificare le informazioni dettagliate per migliorare l'efficienza della pipeline.

È possibile inoltre visualizzare un riepilogo del *pass rate* e della durata della pipeline piuttosto che il *pass rate* dei test in essa eseguiti attraverso la sezione *Analitycs* nella scheda di Analisi della pipeline come mostrato nell'immagine 4.2.

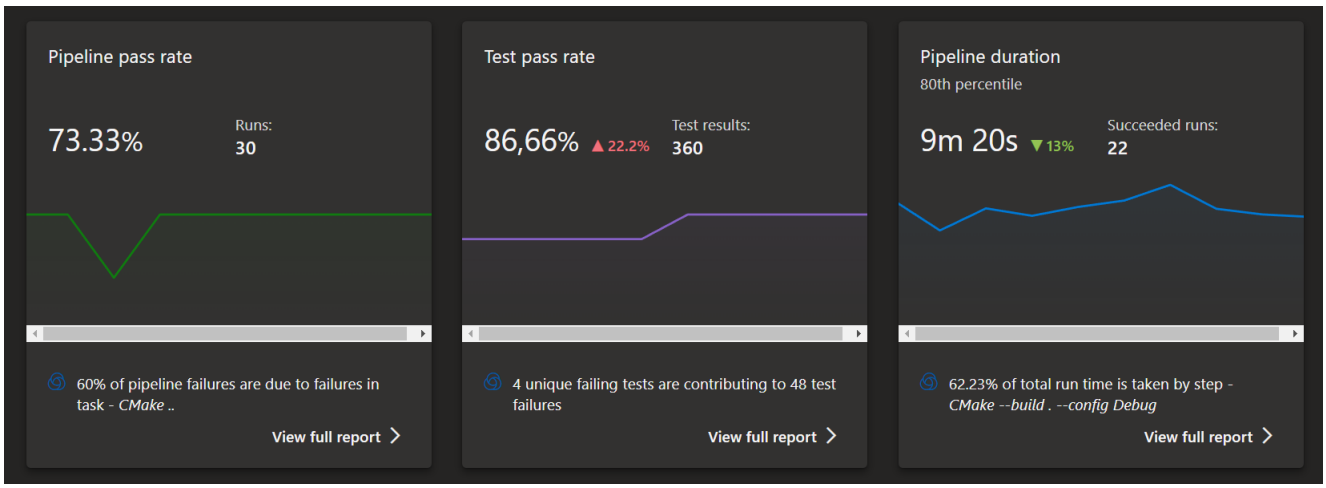

Figura 4.5: Pipeline analytics

Pipeline pass rate

Figura 4.6: Pipeline failure report

Il report della *pipeline pass rate* fornisce una visione dettagliata del pass rate della pipeline e del suo andamento nel tempo. È inoltre possibile visualizzare quale specifico errore dell'attività contribuisce a un numero elevato di errori di esecuzione della pipeline e utilizzare tali informazioni per correggere le principali attività non riuscite. Il rapporto contiene le seguenti sezioni:

- **Summary** : Fornisce le metriche chiave del pass rate della pipeline nel periodo

specificato.

- **Failure trend** : Mostra il numero di errori al giorno. Questi dati sono divisi per fasi se sono applicabili più fasi per la pipeline;
- **Top failing tasks** : Elenca le principali attività non riuscite, la loro tendenza e fornisce puntatori alle loro esecuzioni non riuscite. Analizza gli errori nella build per correggere l'attività non riuscita e migliorare la velocità di passaggio della pipeline.

Pipeline duration report

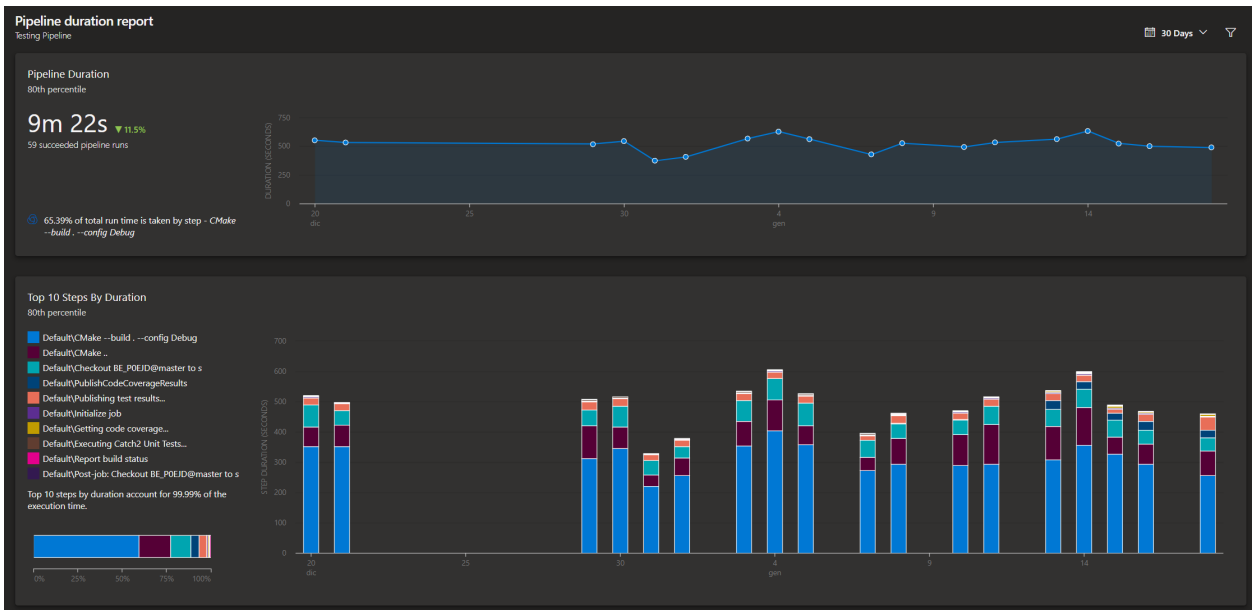

Figura 4.7: Pipeline duration report

Il *Pipeline duration report* mostra quanto tempo impiega in genere la pipeline per completare correttamente. È possibile rivedere l'andamento della durata e analizzare le attività principali in base alla durata per ottimizzare la durata della pipeline.

Pipeline failure test report

Questo report fornisce una visualizzazione dettagliata dei principali test non riusciti nella pipeline, insieme ai dettagli dell'errore.

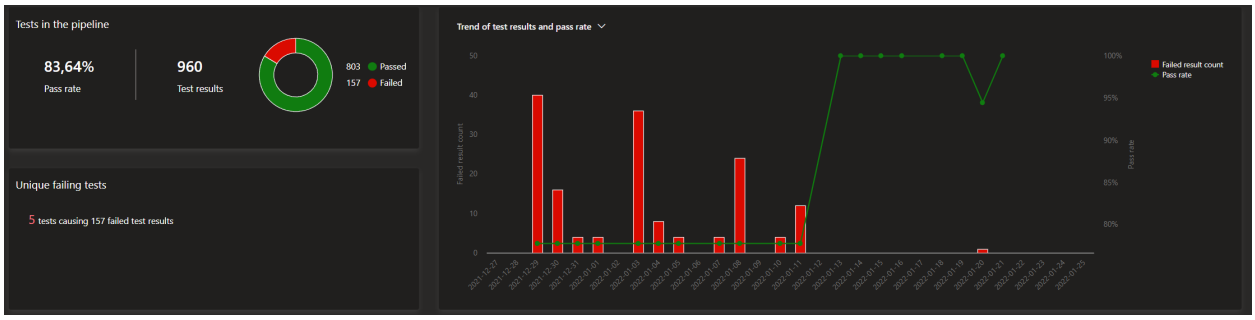

Figura 4.8: Pipeline failure test report [summary]

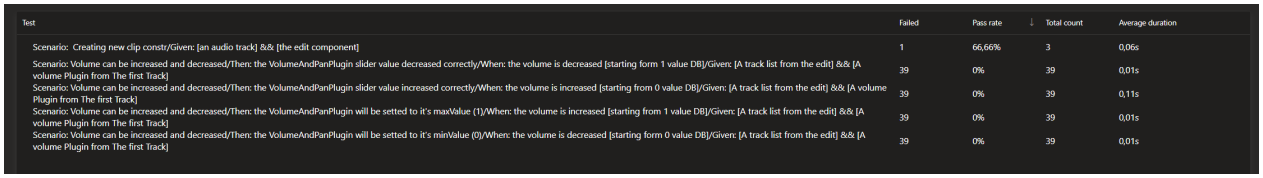

Figura 4.9: Pipeline failure test report [results]

La vista dettagliata contiene due sezioni:

- **Summary** : fornisce metriche quantitative chiave per i test eseguiti in build o release nel periodo specificato. La visualizzazione predefinita mostra i dati per 14 giorni.
  - **Pass rate and results** : mostra la percentuale di superamento, insieme alla distribuzione dei test tra i vari risultati.
  - **Failing tests** : Fornisce un conteggio distinto dei test non riusciti durante il periodo specificato. Nell'immagine 4.2 sopra mostrata, 157 failures sono originate da 5 tests.
  - **Chart view** : Mostra un grafico che descrive i trend del totale dei fallimenti dei test ed il pass rate medio per ogni giorno del periodo specificato.

- **Results** : Mostra l'elenco dei principali test non riusciti in base al numero totale di errori.

  Tale visualizzazione risulta essere molto utile in quanto aiuta ad identificare i test più problematici e consente quindi di approfondire un riepilogo dettagliato dei risultati.

4.3 Analisi risultati di dettaglio

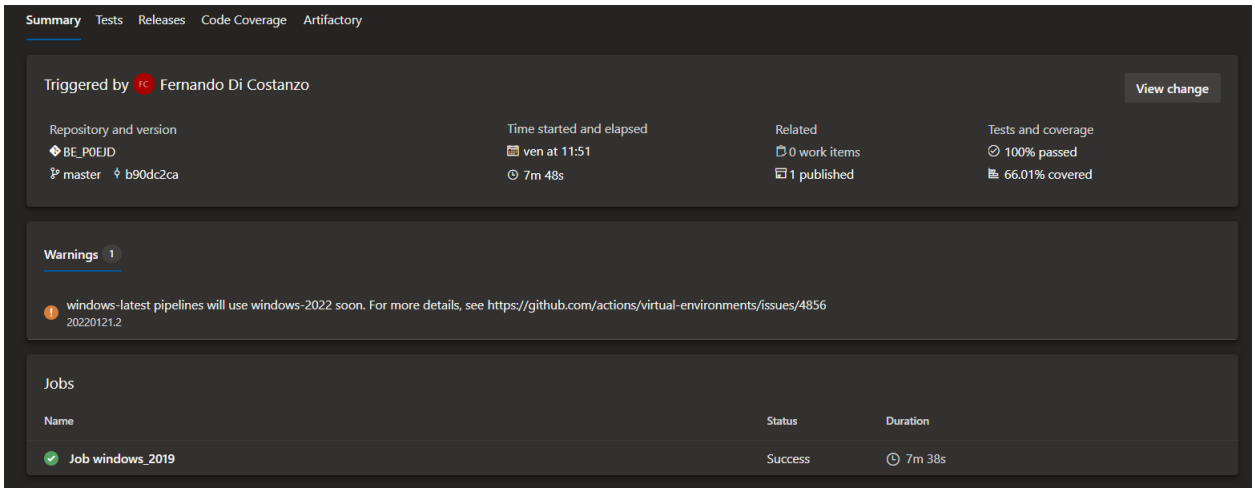

Figura 4.10: Pipeline results [summary]

Nella sezione precedente l'origine dei vari risultati e le statistiche ad essi correlate sono stati generati considerando varie esecuzioni della pipeline accumulate in un dato periodo di tempo.
In generale, per ogni esecuzione della pipeline, vengono generati artefatti e risultati ad essa relativa la cui analisi permette di avere un quadro più specifico dell'esecuzione in questione.
Facendo riferimento allo snippet di codice nella sezione 3.2 è possibile osservare due task che effettuano operazioni di *publish* dei risultati quali esecuzione dei test e code coverage.
Tali pubblicazioni (specifiche per ogni esecuzione della pipeline) vengono quindi riportate nel report della sua esecuzione attraverso il quale è possibile analizzarli.
Con riferimento all'immagine 4.11 è possibile osservare alcuni dati caratteristici dell'esecuzione quali :

- Il repository di appartenenza con il relativo branch ed il collegamento al commit che ha generato il trigger per l'esecuzione della pipeline;

- la data di inizio dell'esecuzione e la relativa durata;
- I *work item* e gli artefatti prodotti
- Le pubblicazioni in merito all'esecuzione dei test e della code coverage.
  E' subito possibile notare la percentuale di successo nell'esecuzione dei test (100%) e quella relativa alla copertura del codice (66.01%).

Focalizzando quindi l'attenzione su *Tests* e *Code Coverage* è possibile avere una visione dettagliata dei risultati in merito all'esecuzione dei task relativi agli argomenti citati. Per quanto riguarda la sezione *Tests* con riferimento all'immagine 4.2 viene come prima cosa mostrata una sezione di *Summary* dove sono elencati alcuni valori quali il numero di test totali eseguiti (con il loro esito in percentuale) piuttosto che la durata complessiva della loro esecuzione.

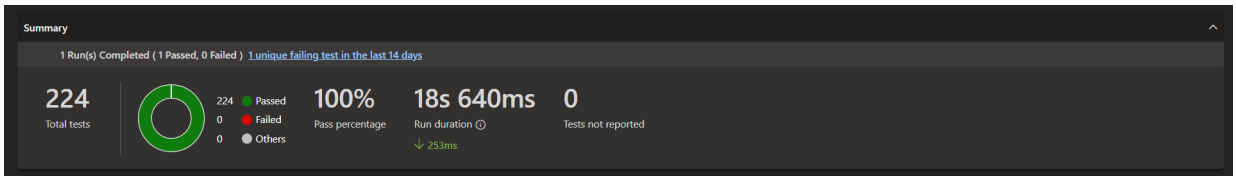

Figura 4.11: Pipeline results [summary]

E' inoltre mostrato l'intero corredo di test eseguiti con il relativo tempo di esecuzione.

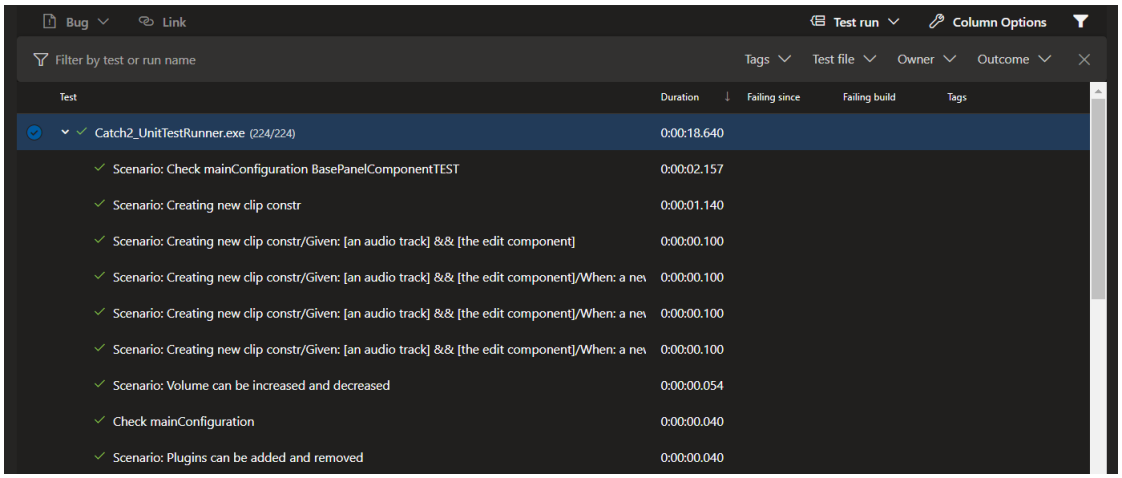

Figura 4.12: Pipeline results [tests]

Tale schermata risulta essere utile in presenza di test falliti poichè presenta, per ogni test, una sezione di *Result Details* che mostra quindi alcune informazione di dettaglio quali *Error message* e *Stack trace* utili ai membri del progetto per la correzzone del fallimento.

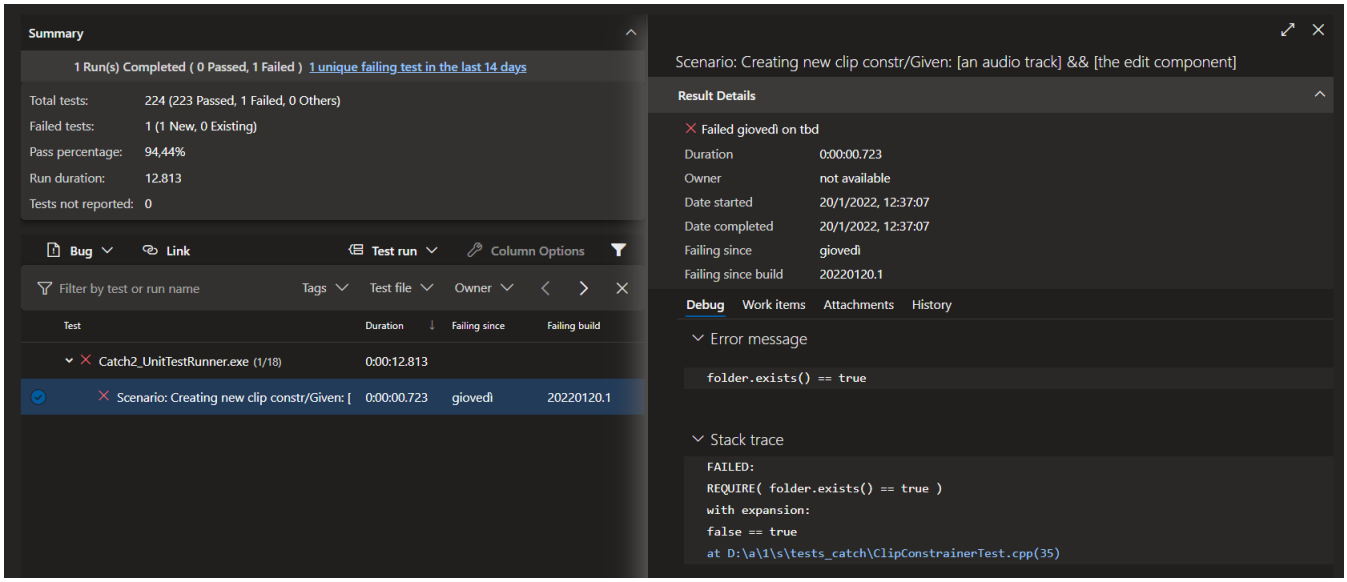

Figura 4.13: Pipeline results [test failed]

Per la sezione *Code Coverage* è invece mostrato il report in figura 4.14 in un formato simil *Cobertura* ottenuto tramite l'esecuzione del task relativo al tool *OpenCppCoverage* descritto nella sezione 2.2.

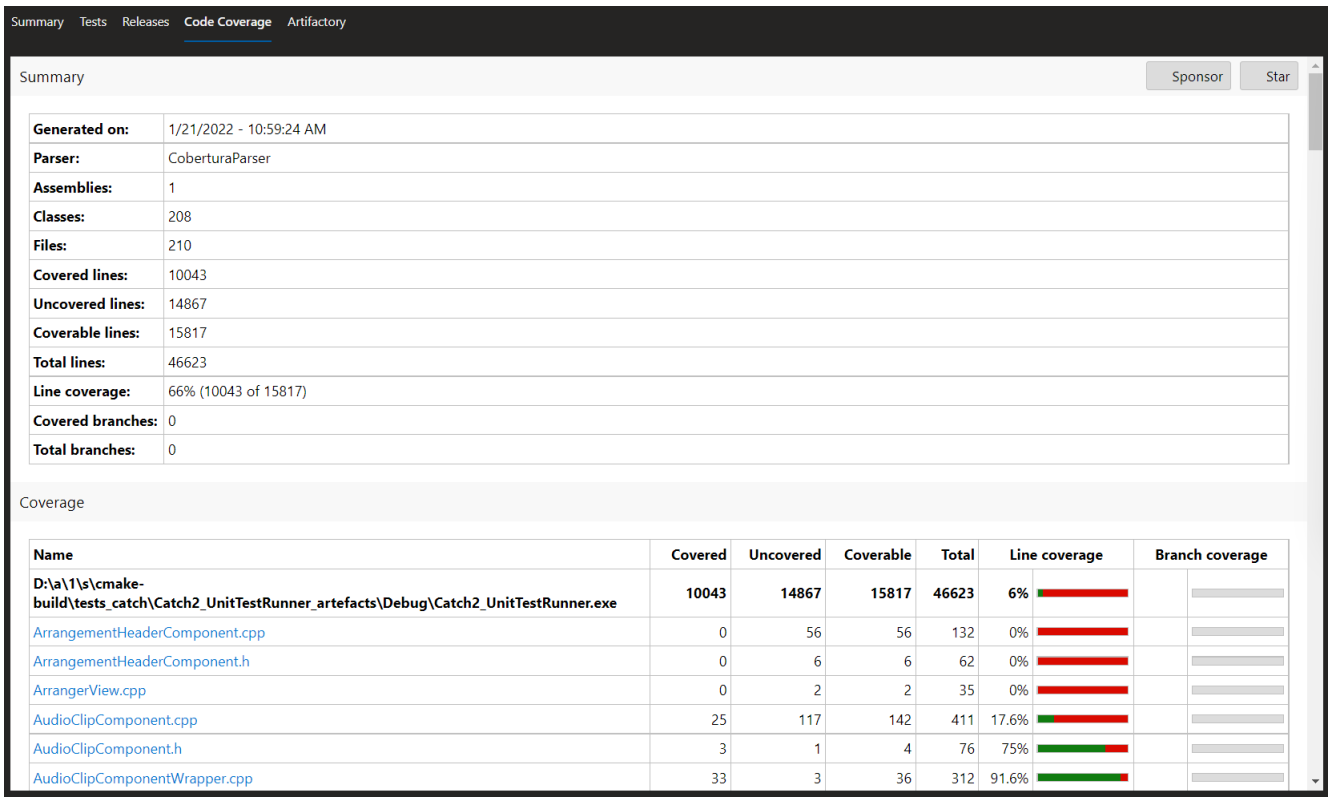

Figura 4.14: Pipeline results [code coverage]

Dal report in questione è possibile osservare alcune informazioni quali il numero di classi e di file presenti piuttosto che il numero di linee totali con la distizione tra *Coverable* e *UnCoverable* oltre che la percentuale di copertura (66% nel caso in questione). E' inoltre possibile osservare per ogni file una sezione di dettaglio (mostrata in figura 4.15) che presenta i dettagli della copertura oltre ad una visualizzazione grafica delle righe di codice coperte (in verde) e non coperte (in rosso).

Class:	AudioClipComponent.cpp
Assembly:	Catch2_UnitTestRunner.exe
File(s):	D:\a\1\s\modules\dawtribe_components\Components\ClipComponent\AudioClipComponent.cpp
Covered lines:	25
Uncovered lines:	117
Coverable lines:	142
Total lines:	411
Line coverage:	17.6% (25 of 142)
Covered branches:	0
Total branches:	0

D:\a\1\s\modules\dawtribe_components\Components\ClipComponent\AudioClipComponent.cpp

#	Line	Line coverage
	1	`/*`
	2	`  ==============================================================================`
	3	
	4	`  AudioClipComponent.cpp`
	5	`  Created: 24 Mar 2020 8:50:36am`
	6	`  Author:  User`
	7	
	8	`  ==============================================================================`
	9	`*/`
	10	`namespace microbees`
	11	`{`
	12	`  namespace dawcomponents`
	13	`  {`
	14	`    using namespace juce;`
	15	
1	16	`    DECLARE_SELECTABLE_OBJECT_AND_CLASS(tracktion_engine::Clip, AudioClipComponentSelectableClass)`
	17	
	18	`    // ========== New section =============//`
0	19	`    void AudioClipComponentSelectableClass::addClipboardEntriesFor(AddClipboardEntryParams& p) {`
0	20	`      c = new Clipboard::Clips();`
0	21	`      for (auto* x : p.items) {`
0	22	`        auto* cl = dynamic_cast<Clip*>(x);`
0	23	`        if (cl)`
0	24	`          c->addClip(0, cl->state);`
0	25	`      }`
	26	
0	27	`      p.clipboard.setContent(make_unique<Clipboard::Clips>(*c));`
0	28	`    }`

Figura 4.15: Pipeline results [code coverage specifications]

Conclusioni

Questo progetto nasce con un'idea in mente: ridurre il ponte che normalmente interessa i team di Sviluppo e Operations in un contesto aziendale concreto.
Cultura, Processi e Strumenti sono già presenti nel flusso di lavoro di Microbees, ma la mentalità dell'azienda è focalizzata sul miglioramento continuo e sulla ricerca di meccanismi più efficienti e affidabili, CI e CD in primis. Tra i vari strumenti illustrati, quello più interessante e flessibile è stato trovato in Azure DevOps, un server per il controllo della versione, reportistica, gestione dei requisiti piuttosto che di Integrazione Continua che nel corso degli anni è stato migliorato sempre di più.
Le capacità di tale strumento sono enormi, alcune delle quali si sono rivelate estremamente utili nell'integrare strumenti di terze parti come GitHub e CMake nell'aiutare a orchestrare un meccanismo di Continuous Integration in modo facilmente accessibile e rapido, grazie anche alla presenza di un Domain Specific Language come quello Declarative Pipeline e la possibilità di definire l'intera pipeline tramite codice all'interno di un file .yaml, versionato e completamente inserito nel progetto stesso. Implementare con successo una toolchain DevOps per sfruttare la Continuous Integration ha indubbiamente mostrato i suoi vantaggi: l'introduzione dell'automazione ha trasformato il processo di integrazione in un processo robusto permettendo l'immediata reportistica e correzione di eventuali bug sin dalle prime fasi dello sviluppo contribuendo all'innalzamento della qualità del codice prodotto lungo i vari sprints.
È importante notare che, grazie a questo progetto, la software house ha anche raccolto importanti informazioni su diversi aspetti del mondo DevOps, come il panorama degli strumenti disponibili sul mercato, i passaggi necessari per costruire una toolchain Custom, la comprensione di quasi tutte le funzionalità che Azure DevOps ha da offrire e il potenziale di questo strumento. Questa conoscenza consentirà all'azienda di migliorare i propri elementi DevOps culturali e di processo, consentendole di diventare un concorrente ancora maggiore nel settore della consulenza: quando un nuovo cliente

si avvicinerà a questa azienda di software, sarà assicurato dalla profonda conoscenza DevOps che Microbees ha da offrire.

Appendice A : *Approfondimento Catch2*

Integrazione con CMake

Idealmente l'uso di Catch2 dovrebbe passare attraverso la sua integrazione con CMake tenendo a mente che esso è definito come un *single header framework* composto quindi da un singolo file (*catch.hpp*) tramite il quale, una volta incluso nel progetto, è possibile usufruire di tutte le funzionalità offerte dal framework.
Oltre al file prima citato viene fornito anche uno script CMake (*catch.cmake*) che, in combinazione con *CTest* (strumento di test distribuito come parte di CMake che può essere utilizzato per automatizzare l'aggiornamento, la configurazione, la creazione e l'esecuzione di test) aiuta gli utenti a registrare automaticamente i loro TEST_CASE.

Il modo più semplice per usare Catch2 è usare il proprio *main()* e lasciare quindi che il framework gestisca gli argomenti dalla riga di comando piuttosto che l'esecuzione della sessione (ciò è ottenuto collegandosi alla libreria *Catch2::Catch2WithMain* tramite il target CMake).
Viceversa è possibile implementare un main custom (definendo come target CMake *Catch2::Catch2*) ed utilizzarlo richiamando opportunamente il test runner di Catch2.

Modalità di utilizzo

In tale sezione si andranno a descrivere quali sono le modalità di utilizzo, i vari elementi e le sintassi che il framework richiede per il suo corretto utilizzo. Verrà come prima cosa mostrato come può essere implementato, per il caso di studio illustrato nel capitolo 3, un *main()* custom che avvia quindi la sessione di Catch2 e la scrittura dei

vari TEST_CASE.

Partendo quindi dalla scrittura del *main()* viene mostrato il seguente snippet di codice:

```
#define CATCH_CONFIG_RUNNER
#define NOGDI
#include <catch2/catch.hpp>
#include <JuceHeader.h>

int main(int argc, char* argv[])
{
    juce::initialiseJuce_GUI();

    int result = Catch::Session().run(argc, argv);

    juce::shutdownJuce_GUI();

    return result;
}
```

Inizialmente sono quindi definite le macro :

- **CATCH_CONFIG_RUNNER** : Catch è progettato per funzionare il più facilmente e velocemente possibile. Per la maggior parte degli utenti l'unica configurazione necessaria è dire a Catch quale file di origine dovrebbe ospitare tutto il codice di implementazione. Ciò avviene attraverso la definizione della macro CATCH_CONFIG_MAIN che indica a Catch l'uso del suo main standard. Tuttavia ci sono ancora alcune occasioni in cui è necessario un controllo più preciso. Per queste occasioni Catch espone una serie di macro per configurare la modalità di compilazione.

  Sebbene Catch sia solo intestazione, internamente mantiene comunque una distinzione tra intestazioni di interfaccia e intestazioni che contengono l'implementazione. Solo un file sorgente nel progetto di test dovrebbe compilare le intestazioni di implementazione e questo è controllato attraverso l'uso di una di queste macro: CATCH_CONFIG_MAIN e CATCH_CONFIG_RUNNER.

  Uno di questi identificatori dovrebbe essere quindi definito prima di includere Catch esattamente **in un file** di implementazione nel tuo progetto.

  Scegliendo quindi l'uso di CATCH_CONFIG_RUNNER è possibile scrivere un proprio main (come mostrato nello snippet sovrastante)

- **NOGDI** : L'interfaccia del dispositivo grafico (GDI) di Microsoft Windows consente alle applicazioni di utilizzare la grafica e il testo formattato sia sul display video che sulla stampante. Le applicazioni basate su Windows non accedono direttamente all'hardware grafico. Al contrario, GDI interagisce con i driver di dispositivo per conto delle applicazioni.
  Ciò può provocare conflitti nei namespace... ad esempio l'interfaccia GDI contiene la classe *Rectangle* che è altresì contenuta nel framwork JUCE (*juce::Rectangle*). L'inclusione di tale macro permette quindi di non usare l'interfaccia GDI ed evitare tutti i conflitti tra namespace.

Successivamente sono incluse le librerie di catch2 e JUCE.
Il vero e proprio *main()* consiste in principio nella chiamata al metodo statico di *juce::initialiseJuce_GUI()* che prevede l'inizializzazione di vari elementi essenziali per la parte grafica di JUCE (e.g. timer, threads di rendering ecc...) e termina con lo shutdown di quest'ultimi tramite l'apposito metodo statico *juce::shutdownJuce_GUI()*.
Tra le due chiamate viene quindi avviata la sessione di Catch2 con il successivo ritorno dell'exit code (valori $\neq$ da 0 indicano un fallimento nell'esecuzione della sessione).

Per quanto riguarda le **asserzioni** : la maggior parte dei framework di test hanno un'ampia raccolta di macro di asserzioni per acquisire tutte le possibili forme condizionali (_EQUALS, _NOTEQUALS, _GREATER_THAN ecc.). In Catch la gestione è diversa poiché esso scompone le espressioni condizionali in un formato C-style naturale riducendole a una o due che verranno sempre utilizzate :

- **REQUIRE()** : verifica un'espressione e interrompe il test case se fallisce;
- **CHECK()** : equivalente alla REQUIRE ma non interrompe il test case se fallisce (utile in presenza di una serie di asserzioni ortogonali e si vogliono osservare tutti i risultati piuttosto che fermarsi al primo nel caso essa fallisca).

Un ulteriore ed importante aspetto da considerare è il comportamento di Catch in relazione ai **confronti tra valori in virgola mobile** poichè durante quest'ultimi è necessario prestare molta attenzione per tollerare errori di round-off e rappresentazioni inesatte (migliorando di fatto l'attributo di **qualità** dei test). A tal proposito Catch fornisce un modo per eseguire confronti tolleranti di valori in virgola mobile tramite l'uso di una

classe wrapper chiamata **Approx**.
Essa può essere utilizzata su entrambi i lati di un'espressione di confronto ed effettua l'overload gli operatori di confronto per tenere conto di una certa tolleranza.
Approx è costruita con impostazioni predefinite che dovrebbero coprire i casi più semplici. Per i casi più complessi, Approx prevede 3 punti di personalizzazione:

- **epsilon** : serve per impostare il coefficiente (in percentuale) di cui un risultato può differire dal valore approsimento tramite Approx di prima che venga rifiutato;
- **margin** : serve per impostare il coefficiente (assoluto) di cui un risultato può differire dal valore approsimento tramite Approx di prima che venga rifiutato;
- **scale** : viene utilizzata per modificare l'ampiezza di Approx per il controllo relativo.

Un semplice esempio può essere il seguente :

```
1 Aprox target1 = Approx(100).epsilon(0.01);
2 100.0 == target1; // Obviously true
3 200.0 == target1; // Obviously still false
4 100.5 == target1; // True, because we set target to allow up to 1%
     difference
5
6 Approx target2 = Approx(100).margin(5);
7 100.0 == target2; // Obviously true
8 200.0 == target2; // Obviously still false
9 104.0 == target2; // True, because we set target to allow absolute
     difference of at most 5
```

Passiamo quindi alla scrittura dei test secondo il formato proposto dal framework.
Come la maggior parte dei framework di test, Catch2 supporta un meccanismo di fissaggio basato sulle classi, in cui gli unit test sono metodi di quest'ultima e le funzioni di *setup()* e *teardown()* possono essere eseguiti nel costruttore/distruttore del tipo della classe.
Tuttavia, il loro utilizzo in Catch2 è raro, perché i test idiomatici utilizzano invece **SECTION** per condividere il codice di *setup()* e *teardown()* tra il codice di test.

I casi di test (**TEST_CASE**) e le sezioni (**SECTION**) sono molto facili da usare nella pratica:

- TEST_CASE(test_name, [tags])
- SECTION(section_name, [section description])

I nomi dei test e delle sezioni (univoci all'interno dell'eseguibile Catch) sono in forma libera rappresentate da stringhe mentre il secondo argomento *tag* (facoltativo) è una stringa contenente uno o più tag.
Quest'ultimi consentono di associare un numero arbitrario di stringhe aggiuntive a un test case che possono essere quindi selezionati (per l'esecuzione o solo per l'elenco) tramite tag o anche tramite un'espressione che combina più tag. Al loro livello più elementare forniscono un modo semplice per raggruppare insieme diversi test correlati.
Per ogni **SECTION** un **TEST_CASE** viene rieseguito dall'inizio.
Ciò assicura che ogni sezione è eseguita dopo il *setup()* e prima del *teardown()*, inoltre le sezioni possono avere una profondità di nidificazione arbitraria e quindi il numero di esecuzioni dipendenze dal livello di nidificazione e consiste nell'esecuzione di tutti i possibili percorsi creati. Ciò è utile quando si hanno più test che condividono parte della configurazione.
Mostriamo quindi un esempio esplicativo :

```
1  TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
2
3      std::vector<int> v( 5 );
4
5      REQUIRE( v.size() == 5 );
6      REQUIRE( v.capacity() >= 5 );
7
8      SECTION( "resizing bigger changes size and capacity" ) {
9          v.resize( 10 );
10
11         REQUIRE( v.size() == 10 );
12         REQUIRE( v.capacity() >= 10 );
13     }
14     SECTION( "resizing smaller changes size but not capacity" ) {
15         v.resize( 0 );
16
17         REQUIRE( v.size() == 0 );
```

```
18          REQUIRE( v.capacity() >= 5 );
19      }
20      SECTION( "reserving bigger changes capacity but not size" ) {
21          v.reserve( 10 );
22
23          REQUIRE( v.size() == 5 );
24          REQUIRE( v.capacity() >= 10 );
25      }
26      SECTION( "reserving smaller does not change size or capacity" ) {
27          v.reserve( 0 );
28
29          REQUIRE( v.size() == 5 );
30          REQUIRE( v.capacity() >= 5 );
31      }
32      SECTION( "reserving bigger changes capacity but not size" ) {
33          v.reserve( 10 );
34
35          REQUIRE( v.size() == 5 );
36          REQUIRE( v.capacity() >= 10 );
37          SECTION( "reserving down unused capacity does not change capacity"
    ) {
38              v.reserve( 7 );
39              REQUIRE( v.size() == 5 );
40              REQUIRE( v.capacity() >= 10 );
41          }
42      }
43  }
```

Seguendo l'esempio ogni sezione viene eseguita con un vettore v appena costruito, che sappiamo ha dimensione 5 e capacità almeno 5, perché le due asserzioni vengono verificate prima dell'esecuzione di ogni sezione. Ogni esecuzione di un test case eseguirà una e solo una sezione foglia.

Sempre dall'esempio è possibile notare che nell'ultima sezione ne è presenta una medesima innestata; in questo caso verrà eseguita prima solamente la sezione padre e successivamente la sezione padre più quella nidificata e così via nel caso di ulteriore nidificazione.

BDD

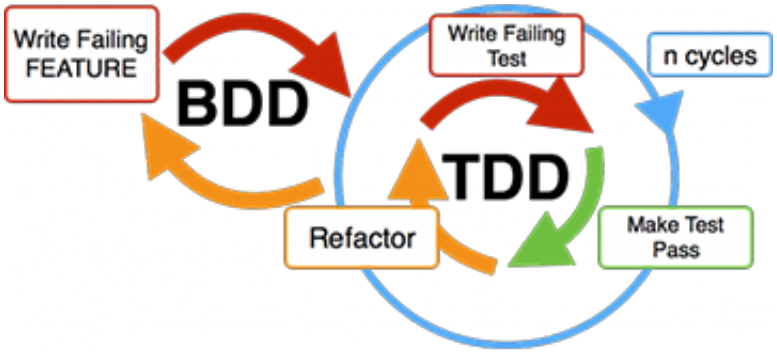

Figura 4.16: BDD - Behavior Driven Development

Una tecnica sempre più diffusa è il behavior-driven development (sviluppo guidato dal comportamento), abbreviato in BDD.
Questa forma di sviluppo software agile ha avuto origine dal test-driven development (TDD) ed è considerata la sua estensione logica. Contrariamente al TDD, in BDD il rispettivo software è considerato principalmente dal punto di vista dell'utente.
Tale approccio promuove quindi una progettazione più olistica del software e facilita la collaborazione tra sviluppatori, responsabili della qualità e clienti.[6]
In tale contesto gli sviluppatori creano, parallelamente al software, dei test unitari o test di sistema appropriati. Nel processo di progettazione di un software, tuttavia, ha senso coinvolgere non solo i programmatori ma anche i membri del team o le parti interessate privi di conoscenza tecnica del codice ed il behavior-driven development (BDD) permette proprio questo.
Come suggerisce il nome, esso si basa sul comportamento che il rispettivo software dovrebbe mostrare. Grazie al cosiddetto linguaggio ubiquo, ovvero al "linguaggio comunemente usato", anche i meno esperti possono creare determinate descrizioni comportamentali. Il linguaggio ubiquo deriva dal domain-driven design (DDD), che, come il BDD, si concentra sul dominio dell'applicazione. Entrambi gli approcci tengono conto di tutte le aree coinvolte nella creazione del software e le combinano indipendentemen-

te da framework, linguaggi di programmazione o strumenti. L'uso di un linguaggio uniforme consente proprio questo.
Catch2 supporta ampiamente il BDD costruendo un insieme di apposite macro che vengono mappate su TEST_CASE e SECTION, con un supporto interno per renderli più agevoli con cui lavorare.

- SCENARIO(scenario_name,[tags])

Questa macro esegue il mapping su TEST_CASE e funziona nel medesimo modo (stessa infatti è anche la lista dei parametri), tranne ovviamente che per la firma della macro stessa.

- GIVEN(string)
- WHEN(string)
- THEN(string)

Queste macro mappano invece la clausola SECTION.Semanticamente, una clausola GIVEN può avere più clausole WHEN indipendenti al suo interno. Ciò consente a un test di avere, un set inizializzato di oggetti comuni a più sottotest che li utilizzano in vari modi in ciascuna delle clausole WHEN senza ripetere l'inizializzazione dalla clausola GIVEN. Quando sono presenti clausole dipendenti, come una seconda clausola WHEN che dovrebbe verificarsi solo dopo che la precedente clausola WHEN è stata eseguita e convalidata, sono presenti macro aggiuntive che iniziano con AND_:

- AND_GIVEN (string)
- AND_WHEN (string)
- AND_THEN(string)

Questi sono usati per concatenare GIVEN, WHEN e THEN insieme. La clausola AND_* è inserita all'interno della clausola da cui dipende. Possono esserci più clausole indipendenti che dipendono tutte da una singola clausola esterna.

Risultati

In tale sezione andremo ad analizzare i risultati ottenuti dall'esecuzione delle test suite scritte con Catch2.
Una volta generato quindi l'eseguibile del progetto di test (contenente le varie test suite implementate ed aggiunte ad esso attraverso CMake) è possibile quindi lanciare quest'ultimo ed osservare gli esiti dei test.
Catch funziona discretamente anche senza alcuna opzione aggiuntiva della riga di comando ma mette comunque a disposizione (in tutti quei casi in cui si desidera un maggiore controllo) molti comandi, tra cui :

- **-s, −−success** : Visualizzazione dei risultati per i test con successo
- **-r, −−reporter** : I reporter sono il modo in cui l'output di Catch2 (risultati di asserzioni, test, benchmark e così via) viene formattato e scritto. Il reporter predefinito è chiamato reporter "Console" e ha lo scopo di fornire un output relativamente dettagliato e di facile utilizzo.
  I tipi di reporter supportati più comunemente usati sono :
  - **Console**;
  - **Xml** : XML Reporter scrive in un formato XML specifico di Catch.
    Il vantaggio di questo formato è che corrisponde bene al modo in cui funziona Catch (soprattutto le funzionalità più insolite, come le sezioni nidificate) ed è un formato completamente in streaming, ovvero scrive l'output mentre va, senza dover memorizzare tutto il suo risultati prima che possa iniziare a scrivere.
    Lo svantaggio è che, essendo specifico di Catch, nessun server di build esistente comprende il formato in modo nativo. Può essere utilizzato come input per una trasformazione XSLT che potrebbe convertirlo, ad esempio, in HTML, anche se questo ovviamente perde il vantaggio dello streaming.
  - **JUnit** : JUnit Reporter scrive in un formato XML che imita lo schema JUnit ANT.
    Il vantaggio di questo formato è che lo schema JUnit Ant è ampiamente compreso dalla maggior parte dei server di build e quindi di solito può essere utilizzato senza lavoro aggiuntivo.

Lo svantaggio è che questo schema è stato progettato per corrispondere a come funziona JUnit e c'è una discrepanza significativa con il modo in cui funziona Catch. Inoltre, il formato non è riproducibile in streaming (perché gli elementi di apertura contengono i conteggi dei test non riusciti e superati come attributi), quindi l'intera esecuzione del test deve essere completata prima di poter essere scritta.

- **-v, −−verbosity {quiet,normal,high}** : La modifica della verbosità può cambiare la quantità di dettagli che i reporter di Catch2 producono;
- **-d, −−durations** : Se attivato, Catch riporterà la durata di ogni test case, in millisecondi.

Lanciando quindi l'eseguibile con alcuni dei comandi sopra elencati si ottiene tale risultato :

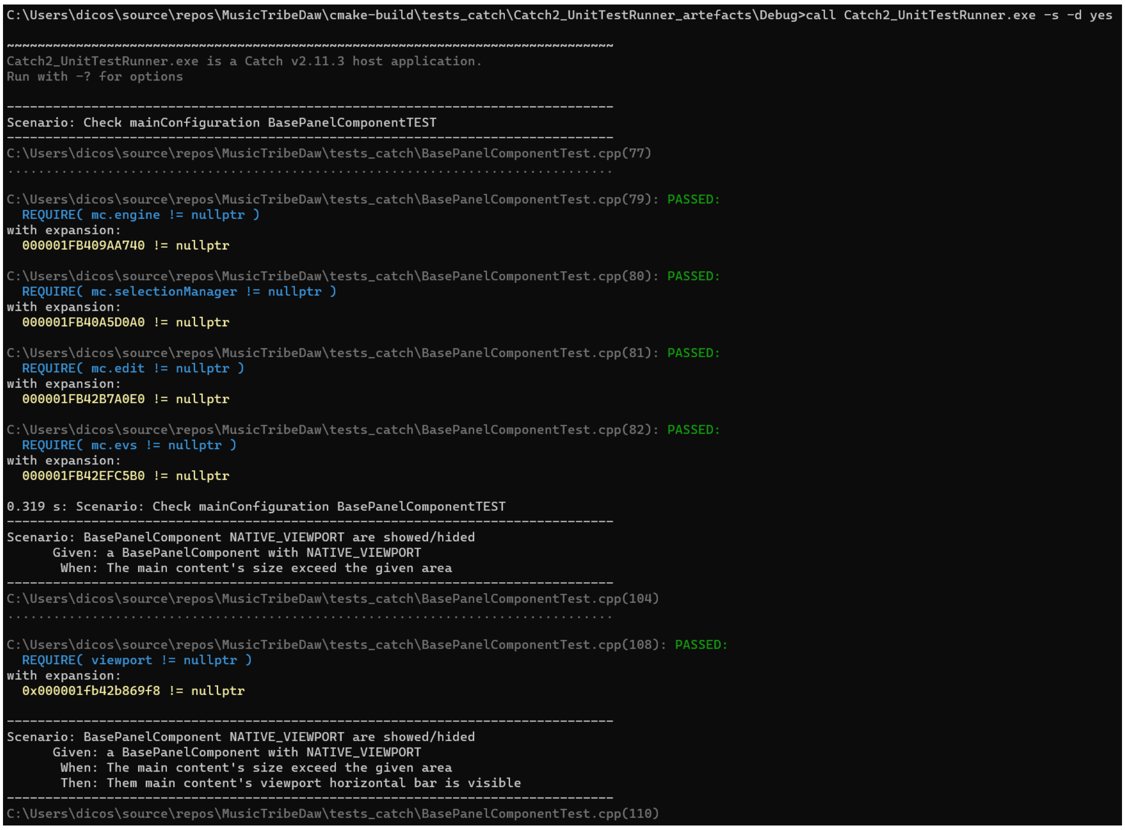

Figura 4.17: Catch2 test result

Ai fini dell'implementazione di una pipelina per la CI attraverso il portale Azure descritta nel capitolo successivo, viene anche generato attraverso un reporter JUnit un

file che può essere compreso dal TestResultPublisher di Azure per la tracciabilità dei risultati.

Appendice B : *Conforto tra unit test frameworks*

Confronto con GoogleTest e JUnit5

In tale appendice viene approfondita la motivazione che ha portato alla scelta di Catch2 andando a mostrare le principali differenze con altri unit test framework disponibili sul mercato : **Google Test** e **JUnit** (le differenze con quest'ultimo sono mostrate a scopo didattico poichè si basano su diverse tecnlogie incopatibili tra loro).

Integrazione con CMake

Considerando l'utilizzo di GoogleTest, una prima evidenza è che la sua importazione nel progetto passa attraverso Cmake.
E' opportuno quindi creare il file CMakeLists.txt e dichiarare la dipendenza da GoogleTest. Esistono molti modi per esprimere le dipendenze nell'ecosistema CMake; il più semplice è usare il modulo *FetchContent* offerto da CMake che effettua il download di dati (o pacchetti) come parte della configurazione della build.
Un esempio può essere il seguente :

```
cmake_minimum_required(VERSION 3.14)
project(my_project)

# GoogleTest requires at least C++11
set(CMAKE_CXX_STANDARD 11)

include(FetchContent)
FetchContent_Declare(
  googletest
```

```
  URL https://github.com/google/googletest/archive/609281088
    cfefc76f9d0ce82e1ff6c30cc3591e5.zip
)

set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
```

Dove :

- **FetchContent_Declare** : registra le opzioni che descrivono come popolare il contenuto specificato. Se tali dettagli sono già stati registrati in precedenza in questo progetto (indipendentemente da dove nella gerarchia del progetto), questa e tutte le successive chiamate per lo stesso contenuto ¡nome¿ vengono ignorate.
- **FetchContent_Declare** : Questo comando assicura che ciascuna delle dipendenze denominate venga popolata e potenzialmente aggiunta alla build prima del ritorno

Da qui una prima differenza con *Catch2* dove, per il suo utilizzo, non è necessario specificare alcuna dipendenza esterna nel *CMakeLists.txt*.
Trattandosi infatti di un *single header framework* basta aggiungere il file *catch.hpp* negli include del progetto ed è possibile utilizzarlo.

Asserzioni

Per quanto riguarda l'uso delle asserzioni con GoogleTest, esse sono rappresentate da macro che assomigliano a chiamate di funzione.
Quando un'asserzione ha esito negativo, il framework stampa il file sorgente dell'asserzione e la posizione del numero di riga, insieme a un messaggio di errore che può essere personalizzato.
Esse presentano principalmente due formati :

- **ASSERT_*** << "error message" : generano errori irreversibili quando falliscono e interrompono la funzione corrente;
- **EXPECT_*** << "error message" : generano errori non fatali, che non interrompono la funzione corrente.

Ognuna delle asserzioni sopra elencate viene specializzata in base al tipo di confronto che deve essere effettuato :

- **Binario** : Le seguenti asserzioni confrontano due valori.Quest'ultimi devono essere confrontabili dal relativo operatore di confronto dell'asserzione :
  - EXPECT(ASSERT)_EQ(val1,val2);
  - EXPECT(ASSERT)_NE(val1,val2);
  - EXPECT(ASSERT)_GE(val1,val2).
- **Booleano** : Le seguenti asserzioni verificano le condizioni booleane :
  - EXPECT(ASSERT)_TRUE(condition);
  - EXPECT(ASSERT)_FALSE(condition).
- **Stringhe** : Le seguenti asserzioni confrontano due stringhe C :
  - EXPECT(ASSERT)_STREQ(str1,str2);
  - EXPECT(ASSERT)_STRNE(str1,str2).
- **Floating-Point** : Le seguenti asserzioni confrontano due valori a virgola mobile. A causa di errori di round-off, è molto improbabile che due valori a virgola mobile corrispondano esattamente, quindi EXPECT_EQ non è adatto.
  GoogleTest fornisce anche asserzioni che utilizzano un limite di errore predefinito basato su Units in the Last Place (ULP).
  - EXPECT(ASSERT)_FLOAT_EQ(val1,val2) : Verifica che i due valori float val1 e val2 siano approssimativamente uguali, entro 4 ULP l'uno dall'altro;
  - EXPECT(ASSERT)_DOUBLE_EQ(val1,val2) : Verifica che i due valori double val1 e val2 siano approssimativamente uguali, entro 4 ULP l'uno dall'altro;

– EXPECT(ASSERT)_NEAR(val1,val2,abs_error) : Verifica che la differenza tra val1 e val2 non superi l'errore assoluto associato abs_error.

Catch2 tratta le asserzioni in modo completamente diverso.
Esso scompone le espressioni condizionali in un formato C-style naturale riducendole a una o due che verranno sempre utilizzate.
Questo implica che, a differenza di GoogleTest, quando si vuole richiamare un asserzione per effettuare un qualsiasi confronto non si deve tener conto di quale asserzione scegliere da catalogo (EXPECT(ASSERT)_EQ , EXPECT(ASSERT)_NE , ecc...) poichè Catch wrappa l'intero catalogo in macro quali REQUIRE (corrispondente a ASSERT_*) e CHECK (corrispondente a EXPECT_*) decomponendo l'espressione passata come argomento e risolvendola.
Focalizzandoci sui confronti in virgola mobile Catch risulta essere più flessibile poichè permette di stabilire un errore sia relativo che assoluto.

```
int a=1,b=2;

GoogleTest :
  EXPECT_NE(a,b);
  EXPECT_GT(b,a)

Catch2 :
  CHECK(a != b);
  CHECK(b > a)
```

Creazione dei test

I test vengono generati attraverso le apposite macro *TEST()* che rappresentano standard funzioni C++ che non restituiscono alcun un valore.
Nel corpo di tali funzioni, insieme a qualsiasi istruzione C++ valida che si desidera includere, vengono utilizzate le varie asserzioni messe a disposizione dal framework

per determinarne l'esito. Il risultato del test è quindi determinato dalle asserzioni in esso presenti; se una qualsiasi asserzione nel test fallisce (in modo fatale o non) o se il test si arresta in modo anomalo, l'intero test fallisce.

```
TEST(TestSuiteName, TestName) {
  ... test body ...
}
```

Il primo argomento è il nome della test suite e il secondo argomento è il nome del test all'interno della test suite. Entrambi i nomi devono essere identificatori C++ validi. Gli identificatori validi sono quelli che rispettano le seguenti regole di sintassi :

- L'identificatore deve essere composto da uno o più caratteri (C++ non pone limiti alla lunghezza di un identificatore);
- Solo i caratteri alfabetici, le cifre numeriche e il carattere di sottolineatura (_) sono validi in un identificatore;
- Il primo carattere di un identificatore deve essere alfabetico o un trattino basso (non può essere una cifra numerica).

Il nome completo di un test è costituito quindi dalla suite di test che lo contiene e dal suo nome individuale. I test di diverse suite di test possono avere lo stesso nome individuale.

> Ancora una volta troviamo in Catch un punto a favore poichè il nome del test viene descritto da una stringa arbitraria e non un identificatore valido C++ rendendo quindi la descrizione del test più flessibile e intuitiva.
> Sia per Catch che per GoogleTest l'intero descrittore del test deve essere univoco in tutto l'eseguibile.
> Inoltre Catch permette di arricchire un test associando ad esso uno o più tag, cosa che non è supportata dalla controparte.

Nell'evenienza in cui due o più test implementati eseguano operano su dati simili, è possibile utilizzare un test *FIXTURE* che consente di riutilizzare la stessa configurazione di oggetti per diversi test.
Per definire una *FIXTURE* occorre :

- Derivare dalla classe *::testing::Test*
- Dichiarare tutti gli oggetti che si intende utilizzare.
- Se necessario, scrivere un costruttore predefinito o una funzione SetUp() per preparare gli oggetti per ogni test.
- Se necessario, scrivere un distruttore o una funzione TearDown() per rilasciare le risorse allocate in SetUp() .
- Se necessario, definire le subroutine da condividere con i test.

```
class QueueTest : public ::testing::Test {
 QueueTest() {
     // You can do set-up work for each test here.
  }

 ~QueueTest() override {
     // You can do clean-up work that doesn't throw exceptions here.
  }
   // If the constructor and destructor are not enough for setting up
  // and cleaning up each test, you can define the following methods:

 protected:
  void SetUp() override {
    // Code here will be called immediately after the constructor (right
    // before each test).
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }

  void TearDown() override {
    // Code here will be called immediately after each test (right
    // before the destructor).
  }

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

Quando si utilizza una *FIXTURE*, deve essere utilizzata la macro *TEST_F()* invece di *TEST()* in quanto consente di accedere a oggetti e subroutine nel *test FIXTURE* :

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(q0_.size(), 0);
}

TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(n, nullptr);

  n = q1_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 1);
  EXPECT_EQ(q1_.size(), 0);
  delete n;

  n = q2_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 2);
  EXPECT_EQ(q2_.size(), 1);
  delete n;
}
```

Quando vengono eseguiti questi test, si verifica quanto segue:

1. *GoogleTest* costruisce un oggetto *QueueTest* (chiamiamolo t1);
2. viene invocato il metodo t1.SetUp() che inizializza t1;
3. Il primo test (IsEmptyInitially) viene eseguito su t1;
4. t1.TearDown() esegue la pulizia al termine del test;
5. t1 viene distrutto;
6. gli stessi passaggi vengono ripetuti su un altro oggetto QueueTest, questa volta eseguendo il test DequeueWorks.

E' possibile effettuare un'analogia anche con **JUnit5** che offre la possibilità di avere classi *Nested* in una test suite attraverso le quali è possibile ridurre la duplicazione di codice.
Questo poiché le istruzioni presenti nei metodi di setUp() delle Nested Class saranno eseguite seguendo la gerarchia di annidamento e dunque è possibile evitare di riscrivere il codice di tali metodi per ogni test da eseguire.

```
1  import org.junit.jupiter.api.*;
2
3  @DisplayName("JUnit 5 Nested Example")
4  class JUnit5NestedExampleTest {
5
6      @BeforeAll
7      static void beforeAll() {
8          System.out.println("Before all test methods");
9      }
10
11     @BeforeEach
12     void beforeEach() {
13         System.out.println("Before each test method");
14     }
15
16     @AfterEach
17     void afterEach() {
18         System.out.println("After each test method");
19     }
20
21     @AfterAll
22     static void afterAll() {
23         System.out.println("After all test methods");
24     }
25
26     @Nested
27     @DisplayName("Tests for the method A")
28     class A {
29
30         @BeforeEach
```

```
31          void beforeEach() {
32              System.out.println("Before each test method of the A class");
33          }
34
35          @AfterEach
36          void afterEach() {
37              System.out.println("After each test method of the A class");
38          }
39
40          @Test
41          @DisplayName("Example test for method A")
42          void sampleTestForMethodA() {
43              System.out.println("Example test for method A");
44          }
45
46          @Nested
47          @DisplayName("When X is true")
48          class WhenX {
49
50              @BeforeEach
51              void beforeEach() {
52                  System.out.println("Before each test method of the WhenX
    class");
53              }
54
55              @AfterEach
56              void afterEach() {
57                  System.out.println("After each test method of the WhenX
    class");
58              }
59
60              @Test
61              @DisplayName("Example test for method A when X is true")
62              void sampleTestForMethodAWhenX() {
63                  System.out.println("Example test for method A when X is
    true");
64              }
65          }
66      }
67  }
```

Tale struttura implementa di fatto un approccio BDD.

JUnit5 richiamerà quindi i metodi di setUp() e tearDown() seguendo la gerarchia di contesto del metodo di test in questione ottenendo il seguente risultato :

```
Before all test methods
Before each test method
  Before each test method of the A class
  Example test for method A
  After each test method of the A class
After each test method
Before each test method
  Before each test method of the A class
    Before each test method of the WhenX class
    Example test for method A when X is true
    After each test method of the WhenX class
  After each test method of the A class
After each test method
After all test methods
```

> In Catch2 c'è un approccio diverso e più naturale per il C++.
> Quest'ultimo è incorporato nel modo in cui il test runner esegue le sezioni di test.
> Non vi è la necessità di definire metodi quale setUp() e tearDown().
> Catch2 esegue ricorsivamente ogni test case, immergendosi in ogni sezione nidificata solo una volta, tornando quindi in cima al termine di quest'ultima e ricominciando.
> In questo modo, non ci sono setUp() e tearDown() poichè ogni sezione nidificata ha il proprio ambiente impostato, con impostazioni locali.

Volendo effettuare quindi un esempio con Catch2 consideriamo la presenza di una classe *Shooter* che si vuole testare.

Tale classe prevede un costruttore (alias del setUp) ed un distruttore (alias del tearDown). A valle dello *SCENARIO* viene definita una sezione *GIVEN* (seguendo un approccio BDD) e da qui l'oggetto *shooter* viene costruito invocando il suo costruttore. Viene quindi raggiunta la prima sezione *WHEN* e successivamente eseguita la prima

sezione *THEN* che contiene le asserzioni in merito ai metodi sull'oggetto che vogliamo testare.

Una volta terminata tale sezione il framework trascurerà quindi l'esecuzione delle altre sezioni , distruggendo l'oggetto allocato (con conseguente ricostruzione), rieseguendo la prima sezione *WHEN* ed eseguendo stavolta la seconda sezione di tipo *THEN* e così via... assicurando quindi di creare e ripristinare l'ambiente prima e dopo l'esecuzione di ogni test.

```
SCENARIO("Set Up and Tear Down show", "[setup-teardown]")
{
  GIVEN("A simple Shouter")
  {
    Shouter shouter("shouter");

    WHEN("After init")
    {
      THEN("Didn't shout much")
      {
        REQUIRE(1 == shouter.getShoutCnt());
        REQUIRE(1 == shouter.getCreactionCnt());
      }


      THEN("Created 2 times here")
      {
        (2 == shouter.getCreactionCnt());
      }
    }

    WHEN("Shouting")
    {
      shouter.shout("hello");

      THEN("It shouted once beside ctor")
      {
        REQUIRE(2 == shouter.getShoutCnt());
          REQUIRE(3 == shouter.getCreactionCnt());
      }
    }
```

```
  }
}
```

L'esecuzione dello snippet di codice sopra mostrato produrrà quindi il seguente risultato :

```
SCENARIO : Set Up and Tear Down show
    GIVEN : A simple Shouter
    [shouter] shouter setUp() !
        WHEN : After init
            THEN : Didn't shout much
    [shouter] shouter tearDown() !
SCENARIO : Set Up and Tear Down show
    GIVEN : A simple Shouter
    [shouter] shouter setUp() !
        WHEN : After init
            THEN : Created 2 times here
    [shouter] shouter tearDown() !
SCENARIO : Set Up and Tear Down show
    GIVEN : A simple Shouter
    [shouter] shouter setUp() !
        WHEN : Shouting
        [shouter] hello !
            THEN : It shouted once beside ctor
    [shouter] shouter tearDown() !
```

A valle quindi dei confronti con i framework di test discussi in precedenza, la scelta di Catch2 nel progetto in questione è stata motivata principalmente dai seguenti fattori:

- Estrema semplicità nell'installazione : essendo un *single header framework* il suo utilizzo è ridotto all'inclusione di quest'ultimo nel progetto attraverso CMake;
- Estrema semplcità di utilizzo : Le macro di asserzione sono ridotte al minimo attraverso la scomposizione in un formato C-style naturale;
- Supporto al BDD nativo;
- Possibilità di formattare l'output attraverso reporter il cui formato è ampiamente riconosciuto dalla maggior parte dei server di build.

Come tutti i framework di test presenta varie limitazioni e svantaggi che possono essere :

- **Thread safe assertions** : Le macro di asserzione di Catch2 non sono thread-safe.
  Ciò non significa che non è possibile utilizzare i thread all'interno del test di Catch, ma che solo un singolo thread può interagire con le asserzioni di Catch e altre macro.
  Ciò deve essere assicurato effettuando la *yeld()* prima della chiamata ad un asserzione.
- **Sezioni annidate in loop** : Se si stanno usando delle SECTION all'interno di loop, esse devono essere create con nomi diversi per ogni iterazione del loop.
- **Esecuzione di più test in parallelo** : Catch2 mantiene l'esecuzione del test in un processo rigorosamente seriale. Nel caso in cui una suite di test impieghi troppo tempo per essere eseguita e si vuole parallelizzare, devono eseguire più processi *side by side*.
  Ciò può essere realizzato in 2 modi:
  - E' possibile spacchettare la test suite in più binari ed eseguirli in parallelo;
  - è possibile eseguire lo stesso binario di test più volte, ma eseguire un diverso sottoinsieme dei test in ogni processo.

Appendice C : *Codice sorgente del CMakeLists.txt*

In tale appendice viene mostrata la gerarchizzazione dei tools/framework effettuata per il progetto in questione :

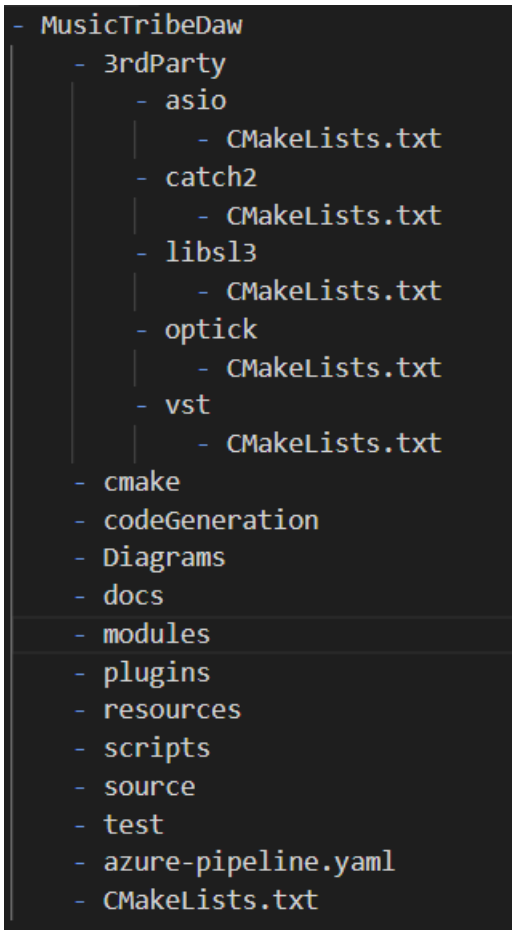

Figura 4.18: CMake hierarchy

- **3rdParty** : è la directory che contiene tutti i tool offerti da terze parti che vengono integrati nel progetto tramite l'apposito CMakeLists.txt che viene riconosciuto

dall'omonimo di top-level che lo include nel progetto e ne gestisce le dipendenze. Nello specifico :

- **asio** : I driver ASIO sono una tecnologia per l'audio digitale sviluppata da *Steinberg* che minimizza i tempi di latenza fra input e output (ossia il ritardo di risposta): ciò è particolarmente importante quando si lavora con la registrazione audio/MIDI.
- **Catch2** : Catch2 è principalmente un framework di unit test per C ++, ma fornisce anche funzionalità di micro-benchmarking bacische e semplici macro BDD (Behavior-driven development).
  Sebbene sia un *single header framwork* esso viene fornito con alcuni script *.make* che forniscono una serie di utilies come l'auto-registrazione dei test.
- **libsl3** : libsl3 è progettato per consentire una comunicazione comoda ed efficiente con il database SQLite 3.x basato sul suo linguaggio naturale, che è SQL.
- **Optick** : Optick è un profiler C++ super leggero progettato principalmente per i giochi ma ben adatto a tutte quelle applicazione che vedono una forte gestione gestione della grafica.
  Fornisce l'accesso a tutti gli strumenti necessari per un'analisi e un'ottimizzazione efficienti delle prestazioni quali :
  * strumentazione;
  * contesti di commutazione;
  * campionamento;
  * contatori GPU;
  * etc...
- **Vst** : Cartella composta essenzialmente da librerie a supporto per lo standard VST.
  VST è stato lanciato nel 1996 ed ha avuto fin dall'inizio un'ampia diffusione grazie ad una relativa semplicità di realizzazione di plugin e alla presenza di un valido supporto per il programmatore costituito da un kit di sviluppo software reso pubblico gratuitamente: un gran numero di sviluppatori ha iniziato a rilasciare plugin anche gratuiti, e di buon livello.

- **cmake** : cartella che contiene tutti gli scipt scritti in linguaggio makefile che vengono inclusi dal CMakeLists.txt di top-level attraverso la direttiva *include()*
- **codeGeneration** : cartella che contiene i tool usati per la codeGeneration , in particolare è stato utilizzato *FMPP* che consiste in è uno strumento di pre-elaborazione di file di testo generici utilizzando i modelli FreeMarker .
  Elabora intere directory in modo ricorsivo e può essere utilizzato per generare siti Web statici completi, codice sorgente, file di configurazione, ecc.
  Può inoltre caricare dati da fonti come CSV, XML e JSON nei file generati.
- **Diagrams** : cartella che contine i diagrammi quali Class Diagram , Activity Diagram ecc...
- **Modules** : L'intero progetto è suddiviso in moduli... ovvero pacchetti con minimo accoppiamento tra di loro ed alta coesione al fine di ridurre le dipendenze favorendo la loro portabilità in futuri progetti.
- **Plugins** : Trattandosi di una DAW è necessaria una forte gestione dei plugins , ovvero programmi esterni sviluppati da terze parti che sono quindi esterni alla DAW che interagiscono con essa modificando i flussi audio/MiDi da essa gestiti.
- **Resources** : Contiene tutte le varie risorse del progetto come immagini, icone , fonts , ecc...
- **Scripts** : Cartella di utility che contiene file .bat che eseguono procedure di build , generation e configuration attraverso la linea di comando CMake in base alle esigenze
- **Source** : Contiene i vari file sorgenti usati dall'applicativo
- **tests** : Contiene le test suite scritte per convalidare la bontà del lavoro effettuato
- **azure-pipeline** : è un file di configurazione in formato .yaml che costituisce una pipelin per l'implementazione della CI nel progetto
- **CMakeLists** : top-level file usato da CMake per la gestione di tutte le dipendenze del progetto e le inclusioni di tutti i moduli

Andiamo quindi ad osservare la struttura del CMakeLists.txt di top-level :

```
# DAW App CMakeLists.txt

cmake_minimum_required(VERSION 3.12)


include(cmake/Custom.cmake)
include(cmake/CopyIfDifferent.cmake)

set(PRJ_NAME "MusicTribeJam")

option(BUILD_DOCS "Build documentation" OFF)
option(BUILD_UNIT_TESTS "Build unit tests" OFF)
option(BUILD_VST "Build VST Support" ON)

project(${PRJ_NAME} VERSION 0.0.1)

option(JUCE_ENABLE_MODULE_SOURCE_GROUPS "Show all module sources in IDE
    projects" OFF)

add_subdirectory(modules/juce_608)

if("${ENABLE_OPTICK}" STREQUAL "")
set(USE_OPTICK_VAL 0)
else()
set(USE_OPTICK_VAL ${ENABLE_OPTICK})
endif()

set(USE_VST 0)
set(USE_AU 0)
set(VST2_PATH "3rdparty/vst/vstsdk366_27_06_2016_build_61" )

if(BUILD_VST)
set(USE_VST 1)
endif()


if(MSVC)
  add_compile_options($<$<CONFIG:Release>:/MT> # Runtime library: Multi-
    threaded
```

```
                    $<$<CONFIG:RelWithDebInfo>:/MT> # Runtime library: Multi-threaded
                    $<$<CONFIG:Debug>:/MTd> # Runtime library: Multi-threaded Debug
                $<$<CONFIG:Debug>:/MP> # Compile with Multiple Core Debug
                    )
endif()

set(JUCER_ENABLE_GPL_MODE 0)

if (BUILD_DOCS)
add_subdirectory(docs)
endif()

juce_set_vst2_sdk_path(${VST2_PATH})


juce_add_module(modules/tracktion/develop_040521/tracktion_engine)
juce_add_module(3rdparty/optick_components)
juce_add_module(modules/dawtribe_components)
juce_add_module(modules/dawtribe_api)
juce_add_module(modules/dawtribe_utils)
juce_add_module(modules/musictribe_dawlnf)

juce_add_module(modules/3rdparty_components)
juce_add_module(modules/ai_components)


juce_add_gui_app(${PRJ_NAME}
    # VERSION ...                       # Set this if the app version is different to the project version
    # ICON_BIG ...                      # ICON_* arguments specify a path to an image file to use as an icon
    # ICON_SMALL ...
    # DOCUMENT_EXTENSIONS ...           # Specify file extensions that should be associated with this app
    # COMPANY_NAME ...                  # Specify the name of the app's author
    PRODUCT_NAME "${PRJ_NAME}")     # The name of the final executable, which can differ from the target name
```

```
71
72
73 juce_generate_juce_header(${PRJ_NAME})
74
75
76 set_property(TARGET ${PRJ_NAME} PROPERTY CXX_STANDARD 17 )
77 set(THIRDPARTY_INC )
78 set(THIRDPARTY_SRC )
79 set(THIRDPARTY_LIB )
80
81
82 add_subdirectory(3rdparty/asio)
83 add_subdirectory(3rdparty/libsl3)
84 add_subdirectory(3rdparty/optick_components)
85
86 #---------------- Internal Plugins  --------------------------
87 set(INTERNAL_PLUGINS
88 P0XXXVintage_Juce
89 )
90
91
92 set(INTERNAL_PLUGINS_INCL_FOLDERS )
93
94 set(INTERNAL_PLUGINS_SRC )
95
96 foreach(P IN LISTS INTERNAL_PLUGINS)
97
98 include(plugins/${P}/PluginSrcFiles.cmake)
99
100   foreach(S IN LISTS ${P}_INCL_FOLDERS)
101   list(APPEND INTERNAL_PLUGINS_INCL_FOLDERS ${CMAKE_CURRENT_SOURCE_DIR}/
       plugins/${P}/${S})
102   endforeach()
103
104   foreach(S IN LISTS ${P}_SRC)
105   list(APPEND INTERNAL_PLUGINS_SRC ${CMAKE_CURRENT_SOURCE_DIR}/plugins/${P
       }/${S})
106   endforeach()
107
108 endforeach(P IN LISTS INTERNAL_PLUGINS)
```

```
109
110
111 #-------------------------------------------------------------------
112
113
114 target_link_libraries(dawtribe_components INTERFACE sl3)
115
116
117
118 #-----  Generated Stuff --------------------------------
119 set(MUSICTRIBEJAM_DB_VERSION 1)
120
121 set(MUSICTRIBEJAM_UPDATE_CLIENTID "2a57b499-7edb-45d3-b506-2506a535cbda")
122 set(MUSICTRIBEJAM_UPDATE_CLIENTSECRET "
    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX")
123 set(MUSICTRIBEJAM_UPDATE_URL "http://internal.cloud.midasconsoles.com/api/
    v2")
124
125
126 configure_file(${PROJECT_SOURCE_DIR}/codegeneration/MusicTribeJam.h.in ${
    CMAKE_CURRENT_BINARY_DIR}/MusicTribeJam.h @ONLY)
127 #-------------------------------------------------------------
128
129
130 set(DIALOG_SRC
131
132   Source/Dialogs/AddTracksDialog.cpp
133   Source/Dialogs/PreferenceComponent.cpp
134   Source/Dialogs/DawAudioDeviceSelectorComponent.cpp
135 )
136
137 set(API_SRC
138
139   Source/Api/Engine/EngineApi.cpp
140   Source/Api/Engine/DawApi.cpp
141 )
142
143 set(TRACKTION_SRC
144
145   Source/Tracktion/DawUIBehaviour.cpp
```

```
146   Source/Tracktion/DawEngineBehaviour.cpp
147   Source/Tracktion/PluginWindow.cpp
148
149 )
150
151
152 set(UPDATE_SRC
153
154   Source/Updater/AutoUpdater.cpp
155   Source/Updater/VersionInfo.cpp
156   source/Updater/BehringerAutoUpdate.cpp
157   source/Updater/BehringerUtils.cpp
158   source/Updater/BehringerVersionInfo.cpp
159
160 )
161
162 set(ENGINE_SRC
163
164   Source/Api/Engine/EngineApi.cpp
165   Source/Api/Engine/DawApi.cpp
166
167   Source/Utils/DebugCommands.cpp
168   Source/Utils/AppServiceLocator.cpp
169
170 )
171
172 set(TRIBEAPP_SRC
173
174   Source/Main.cpp
175     Source/MainComponent.cpp
176     Source/MenuCommands.cpp
177   Source/AppInit.cpp
178
179   Source/LookandFeel/AppLookAndFeel.cpp
180 )
181
182 target_sources(${PRJ_NAME} PRIVATE
183
184
185
```

```
186   ${API_SRC}
187
188   ${DIALOG_SRC}
189   ${ENGINE_SRC}
190   ${UPDATE_SRC}
191   ${TRACKTION_SRC}
192   ${TRIBEAPP_SRC}
193   ${THIRDPARTY_SRC}
194   ${INTERNAL_PLUGINS_SRC}
195 )
196
197
198
199 target_include_directories(${PRJ_NAME} PRIVATE
200   ${CMAKE_CURRENT_BINARY_DIR}
201   ${THIRDPARTY_INC}
202
203
204     Source/AboutComponent/include
205   Source/Dialogs/include
206   Source/LookandFeel/include
207   Source/Tracktion/include
208   Source/Updater/include
209   Source/Api/Engine/include
210   Source/Utils/include
211
212   Source
213
214   ${INTERNAL_PLUGINS_INCL_FOLDERS}
215
216 )
217
218 source_group("Api"                    FILES ${API_SRC})
219 source_group("Dialogs"                  FILES ${DIALOG_SRC})
220 source_group("Engine"                 FILES ${ENGINE_SRC})
221 source_group("Update"                 FILES ${UPDATE_SRC})
222 source_group("Tracktion"                FILES ${TRACKTION_SRC})
223 source_group("App"                    FILES ${TRIBEAPP_SRC})
224
225
```

```
226
227 include(CTest)
228
229 target_compile_definitions(${PRJ_NAME} PRIVATE
230     # JUCE_WEB_BROWSER and JUCE_USE_CURL would be on by default, but you
    might not need them.
231     JUCE_WEB_BROWSER=0  # If you remove this, add 'NEEDS_WEB_BROWSER TRUE'
    to the 'juce_add_gui_app' call
232     JUCE_USE_CURL=0     # If you remove this, add 'NEEDS_CURL TRUE' to the
    'juce_add_gui_app' call
233     JUCE_APPLICATION_NAME_STRING="$<TARGET_PROPERTY:${PRJ_NAME},
    JUCE_PROJECT_NAME>"
234     JUCE_APPLICATION_VERSION_STRING="$<TARGET_PROPERTY:${PRJ_NAME},
    JUCE_VERSION>"
235   JUCE_DISPLAY_SPLASH_SCREEN=0
236   JUCE_USE_MP3AUDIOFORMAT=1
237   JUCE_USE_LAME_AUDIO_FORMAT=1
238   JUCE_ASIO=1
239   JUCE_PLUGINHOST_VST=${USE_VST}
240   JUCE_PLUGINHOST_VST3=${USE_VST}
241   JUCE_PLUGINHOST_AU=${USE_AU}
242   JUCE_VST3_CAN_REPLACE_VST2=1
243   JUCE_CUSTOM_VST3_SDK=0
244   JUCE_ENABLE_REPAINT_DEBUGGING=0
245   TRACKTION_ENABLE_TIMESTRETCH_SOUNDTOUCH=1
246   THPAR_SQLITE=1
247   USE_OPTICK=${USE_OPTICK_VAL}
248
249   )
250
251
252
253 add_subdirectory(resources)
254
255
256 target_link_libraries(${PRJ_NAME} PRIVATE
257     ${PRJ_NAME}Data            # If we'd created a binary data target, we'd
     link to it here
258     juce::juce_gui_extra
259   juce::juce_cryptography
```

```
  dawtribe_components
  dawtribe_utils
  dawtribe_api
  3rdparty_components
  ai_components
  tracktion_engine
  optick_components
  musictribe_dawlnf
  ${THIRDPARTY_LIB}
  )

target_link_directories(${PRJ_NAME} PRIVATE
${THIRDPARTY_LIB_DIR}
)



if (BUILD_UNIT_TESTS)
  list (APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/3rdparty/
    catch2/Catch2/contrib")
    enable_testing()
    add_subdirectory(3rdparty/catch2/Catch2 EXCLUDE_FROM_ALL)
    include(CTest)
    include(Catch)
  add_subdirectory(tests_catch)
  list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")
endif()
```

Appendice D : *Il framework JUCE*

Figura 4.19: JUCE

JUCE è un framework applicativo C++ multipiattaforma ed open source per la creazione di applicazioni desktop e mobili di alta qualità, inclusi plug-in audio VST, VST3, AU, AUv3, RTAS e AAX.
Esso può essere facilmente integrato con progetti esistenti tramite CMake o può essere utilizzato come strumento di generazione di progetti tramite il *Projucer*, che supporta l'esportazione di progetti per Xcode (macOS e iOS), Visual Studio, Android Studio, Code::Blocks e Linux Makefile oltre a contenere un editor di codice sorgente.[13]
Il Projucer può essere utilizzato per creare nuovi progetti JUCE, visualizzare tutorial ed eseguire esempi. È anche possibile includere il codice sorgente dei moduli JUCE direttamente in un progetto esistente, oppure costruirli in una libreria statica o dinamica che può essere collegata ad un progetto. Per poter essere integrato con CMake è richiesta la sua versione 3.15.

Esso è organizzato in moduli (come mostrato nell'immagine **??**) ognuno dei quali, se aggiunto al progetto, abilita una serie di funzionalità a supporto :

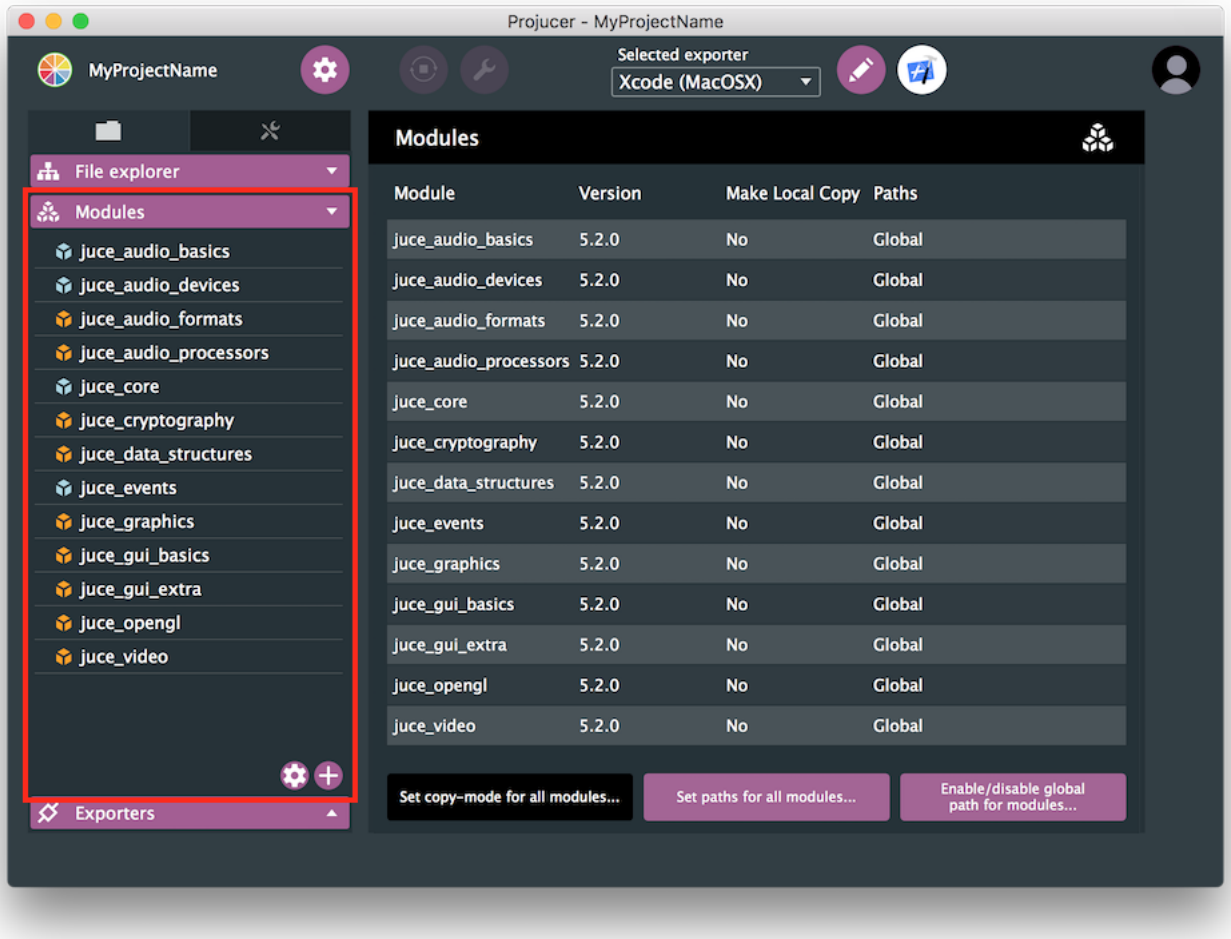

Figura 4.20: JUCE modules

Per esempio *juce_gui_basic* e *juce_gui_basic* sono i moduli che presentano le funzionalità rispettivamente di base ed avanzate per definire classi *Component* (ovvero che possono essere reinderizzate a video) ed altri costrutti utili per la creazione di una GUI.

Creare quindi un interfaccia attraverso gli oggetti *Component* è un'operazione molto veloce e flessibile; basterà estendere quindi dalla classe *Component* ed effettuare l'override di specifici metodi quale *paint()* e *resized()* utili rispettivamente a disegnare il componente e posizionarlo sullo schermo con determinate dimensioni.
Si innesta quindi un meccanismo di parentela tra i componenti che vengono man mano innestati dove il parent setta le dimensioni del figlio (dando come constraint il proprio bound) e così via...
Un esempio può essere il seguente codice :

```
1  //------------------ .h ------------------------
2  #pragma once
3
4  #include <JuceHeader.h>
5  using namespace juce;
6
7  class ChildComponent : public Component
8  {
9  public:
10   ChildComponent(const String& compName);
11   ~ChildComponent();
12
13     void paint(juce::Graphics&) override;
14     void resized() override;
15
16 private:
17
18     String componentName;
19 };
20
21
22
23 class MainComponent  : public juce::Component
24 {
25 public:
26     //
     ==============================================================================
27     MainComponent();
28     ~MainComponent() override;
29
30     //
     ==============================================================================
31     void paint (juce::Graphics&) override;
32     void resized() override;
33
34 private:
35
36     ChildComponent* childComponent;
```

```
37
38      JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
39  };
40
41  //----------------- .cpp ------------------------
42  #include "MainComponent.h"
43
44  ChildComponent::ChildComponent(const String& compName) : componentName(
        compName)
45  {
46  }
47
48  ChildComponent::~ChildComponent()
49  {
50  }
51
52  void ChildComponent::paint(juce::Graphics& g)
53  {
54      Colour backgroundColour = Colours::grey;
55      auto totalArea = getLocalBounds().toFloat();
56
57      g.setColour(backgroundColour);
58      g.fillRoundedRectangle(totalArea, 5);   //Just fill all component's
        area with a gray color.
59
60      g.setColour(backgroundColour.contrasting());
61      g.drawText(componentName,totalArea, Justification::centred);
62  }
63
64  void ChildComponent::resized()
65  {
66  }
67
68
69
70  //
        ==============================================================================
71  MainComponent::MainComponent()
72  {
```

```
73      childComponent = new ChildComponent("Child Component !");
74      addAndMakeVisible(childComponent);  //Now we add the "childComponent"
        as mainComponent child ( and also make it visible )
75      setSize (600, 400);     //This is the main component... so it has the
        responsability to set the main app size.
76  }
77
78  MainComponent::~MainComponent()
79  {
80      delete childComponent;
81  }
82
83
84  void MainComponent::paint (juce::Graphics& g)
85  {
86      g.setColour(Colours::red);  //We can assign a colour ( standard or not
        ) to the Graphic object that has the responsability to draw and fill
        shapes.
87      Rectangle<float> componentArea = getLocalBounds().toFloat();    //We
        can get the component area that was setted by it's parent in it's
        resized() method
88      g.fillRect(componentArea);   //Now we can just draw a rectangle that
        covers the "componentArea" bounds with red colour setted before
89  }
90
91  void MainComponent::resized()
92  {
93      childComponent->setBounds(getLocalBounds().reduced(20));    //In this
        way we can set a bound to all child
94  }
```

Il risultato sarà una window con l'aspetto mostrato in figura 4.21

Figura 4.21: JUCE demo output

La classe Component fornisce inoltre metodi per la gestioni di callback relative a :

- Lost e grab del focus;
- Mouse gestures quali il click sul componente, l'entrata e l'uscita del mouse ecc...
- Eventi di iterazione con il **ValueTree** , che rappresenta il data base locale a tutte le applicazioni juce;
- ecc...

Il modo più semplice per includere JUCE nel progetto è aggiungerlo come sottodirectory e includere la riga *add_subdirectory(JUCE)* nel CMakeLists.txt di top-level.
Altre API verso CMake possono essere :

- **juce_add_<target>** : alcuni esempi possono essere *juce_add_gui_app* e *juce_add_console_app* che configurano un target eseguibile denominato <target>.
  Inoltre *juce_add_plugin* aggiunge un target di libreria statica con nome <target>, insieme a target extra per ciascuno dei formati di plug-in specificati (VST, VST3, ecc...) .
- **juce_add_binary_data** : Crea una libreria statica che incorpora il contenuto dei file passati come argomenti a questa funzione.

Aggiunge quindi un target di libreria chiamato <name> che può essere collegato ad altri target usando *target_link_libraries*.

- **juce_generate_juce_header** : Analizza i moduli JUCE che sono stati collegati <target> e genera un JuceHeader.h che contiene #includeistruzioni per ciascuna delle intestazioni del modulo. Questa intestazione contiene anche un'istruzione facoltativa *using namespace juce* e un blocco *ProjectInfo* facoltativo , ciascuno dei quali può essere disabilitato impostando le definizioni di compilazione *DONT_SET_USING_JUCE_NAMESPACE* e *JUCE_DONT_DECLARE_PROJECTINFO* rispettivamente. L'intestazione risultante può essere inclusa con include <JuceHeader.h>.

- **juce_add_modules** : aggiunge un target di libreria per il modulo JUCE situato nel percorso fornito. <path> deve essere il percorso di una directory del modulo (es. /Users/me/JUCE/modules/juce_core). Questo aggiungerà una libreria di interfaccia con un nome che corrisponde al nome della directory del modulo. La libreria risultante può essere collegata ad altri obiettivi normalmente, utilizzando target_link_libraries.

Memory Leak Detector

Nella sezione di codice mostrata precedentemente 4.3 vi è presente la macro *JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (ClassName)* che elimina il costruttore di copia e l'overload dell'operazione di assegnazione dalla classe *ClassName* ma soprattutto consente di incorporare un leak-detector object all'interno di essa. Questo perchè JUCE supporta un meccanismo di rilevazione dei memory leak tramite instrumentazione del codice in modo completamente trasparente all'utente.
In tale sezione andremo quindi a descrivere il suo funzionamento e verrà effettuato un paragone con LeakCanary, ovvero un memory leak detector usato principalmente per Android. Sostanzialmente il Leak Detector di JUCE provvede a mantenere un conteggio statico interno del numero di istanze attive, in modo che quando l'app viene arrestata e vengono chiamati i distruttori statici, può verificare se sono presenti istanze rimanenti che potrebbero causare leak.

```
1 template <class OwnerClass>
2 class LeakedObjectDetector
3 {
```

```
4  public:
5      //
       ==============================================================================
6      LeakedObjectDetector() noexcept                                    { ++(
       getCounter().numObjects); }
7      LeakedObjectDetector (const LeakedObjectDetector&) noexcept        { ++(
       getCounter().numObjects); }
8
9      LeakedObjectDetector& operator= (const LeakedObjectDetector&) noexcept
       = default;
10
11     ~LeakedObjectDetector()
12     {
13         if (--(getCounter().numObjects) < 0)
14         {
15             DBG ("*** Dangling pointer deletion! Class: " <<
       getLeakedObjectClassName());
16
17             /** If you hit this, then you've managed to delete more
       instances of this class than you've
18                 created.. That indicates that you're deleting some dangling
        pointers.
19
20                 Note that although this assertion will have been triggered
       during a destructor, it might
21                 not be this particular deletion that's at fault - the
       incorrect one may have happened
22                 at an earlier point in the program, and simply not been
       detected until now.
23
24                 Most errors like this are caused by using old-fashioned,
       non-RAII techniques for
25                 your object management. Tut, tut. Always, always use std::
       unique_ptrs, OwnedArrays,
26                 ReferenceCountedObjects, etc, and avoid the 'delete'
       operator at all costs!
27             */
28             jassertfalse;
29         }
```

```
30      }
31
32  private:
33      //
    ==============================================================================
34      class LeakCounter
35      {
36      public:
37          LeakCounter() = default;
38
39          ~LeakCounter()
40          {
41              if (numObjects.value > 0)
42              {
43                  DBG ("*** Leaked objects detected: " << numObjects.value <<
     " instance(s) of class " << getLeakedObjectClassName());
44
45                  /** If you hit this, then you've leaked one or more objects
     of the type specified by
46                      the 'OwnerClass' template parameter - the name should
    have been printed by the line above.
47
48                      If you're leaking, it's probably because you're using
    old-fashioned, non-RAII techniques for
49                      your object management. Tut, tut. Always, always use
    std::unique_ptrs, OwnedArrays,
50                      ReferenceCountedObjects, etc, and avoid the 'delete'
    operator at all costs!
51                  */
52                  jassertfalse;
53              }
54          }
55
56          Atomic<int> numObjects;
57      };
58
59      static const char* getLeakedObjectClassName()
60      {
61          return OwnerClass::getLeakedObjectClassName();
```

```
62      }
63
64      static LeakCounter& getCounter() noexcept
65      {
66          static LeakCounter counter;
67          return counter;
68      }
69  };
```

Confronto con Leak Canary

Volendo quindi effettuare un confronto con un altro leak-detector viene preso in considerazione **LeakCanary**.

LeakCanary è uno strumento open-source basato sull'instrumentazione del codice sorgente e successivo monitoraggio dell'heap.

E' basato sull'idea di monitorare continuamente la memoria heap al fine di permettere allo sviluppatore di scoprire se oggetti dell'applicazione dotati di un ciclo di vita, come le activity, i fragment ecc... sono, dopo la loro distruzione, effettivamente deallocati dal garbage collector ed in caso contrario, fornire la catena di riferimenti che può aver dato origine al problema.

Una prima distinzione è quindi l'usabilità delle due librerie; mentre LeakCanary è capace di installarsi ed autoconfigurarsi una volta aggiunta la sua dipendeza nel build grandle della propria applicazione, il LeakedObjectDetector proposto da JUCE deve essere aggiunto tramite l'apposita macro in ogni classe.

Ciò è però necessario poichè il C++ non ha un meccanismo di gestione automatica della memoria come garbage collector (java) che fornisce un automatico e completo rilevamento e rimozione di oggetti che non sono più in uso (comportando un degrado delle prestazioni poichè necessità di CPU time computing per essere eseguito).

Quindi è necessario instrumentare il codice affinchè siano tenuti dei contatori alle referenze osservate (nelle classi che si registrano al leak detector) per poi controllare in fase di distruzioni se esse sono state eliminate (attravero l'apposito operatore di *delete*).

Una seconda differenza riguarda le performance : mentre leak canary effettua un continuo monitoraggio e dumping dell'area heap (rendendolo CPU time consuming) il LeakedObjectDetector di JUCE interviene esclusivamente nella fase di shutdown dell'applicazione quando vi è la distruzione di tutti gli oggetti istanziati non degradando

le prestazioni dell'applicazione.
Come ultima distinzione ,a differenza di Leak Canary, con il LeakedObjectDetector offerto da JUCE non permette di ottenere lo stack trace di possibili leak rilevati ma bensì solo di ottenere il numero di istanze per ogni tipo di ogni potenziale leak.
Il framework però mette a disposizione un ulteriore leak detector denominato *HeavyweightLeakedObjectDetector*.
Per usarlo basterà agganciarlo alla classe in questione tramite l'apposita macro *JUCE_HEAVYWEIGHT_LEAK_DETECTOR(ClassName)* e provvederà quindi alla creazione di uno stack trace per i potenziali leak rilevati.E' pensato quindi ad una valida alternativa (che appesantisce però le prestazioni) quando il classi leak detector non dovesse bastare.

Elenco delle figure

Bibliografia

[1] *(PDF) A Comparison Between Five Models Of Software Engineering*. https://www.researchgate.net/publication/258959806_A_Comparison_Between_Five_Models_Of_Software_Engineering. (Accessed on 07/15/2021).

[2] *Agile Project Management. Overview delle principali metodologie Agile quali ... - David Corbucci - Google Libri*. https://books.google.it/books?id=o4WXCgAAQBAJ&printsec=frontcover&hl=it&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false. (Accessed on 11/05/2021).

[3] *Azure Pipelines Guida per l'utente - Concetti chiave - Azure Pipelines — Microsoft Docs*. https://docs.microsoft.com/it-it/azure/devops/pipelines/get-started/key-pipelines-concepts?view=azure-devops. (Accessed on 11/28/2021).

[4] Len Bass, Ingo Weber e Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[5] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[6] *Behavior-driven development (BDD) - IONOS*. https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/behavior-driven-development/. (Accessed on 01/10/2022).

[7] *catchorg/Catch2: A modern, C++-native, test framework for unit-tests, TDD and BDD - using C++14, C++17 and later (C++11 support is in v2.x branch, and C++03 on the Catch1.x branch)*. https://github.com/catchorg/Catch2. (Accessed on 01/06/2022).

[8] *Ciclo di vita dello sviluppo software (SDLC) - Big water Consulting*. https://bigwater.consulting/2019/04/08/software-development-life-cycle-sdlc/. (Accessed on 07/15/2021).

[9] *cmake-it.pdf*. https://riptutorial.com/Download/cmake-it.pdf. (Accessed on 02/06/2022).

[10] *Continuous Integration: spiegazione, vantaggi e svantaggi - IONOS*. https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/continuous-integration/. (Accessed on 01/30/2022).

[11] Martin Fowler e Matthew Foemmel. *Continuous integration*. 2006.

[12] *Gradle vs. Maven - DZone Java*. https://dzone.com/articles/gradle-vs-maven. (Accessed on 02/06/2022).

[13] *juce-framework/JUCE: JUCE is an open-source cross-platform C++ application framework for desktop and mobile applications, including VST, VST3, AU, AUv3, RTAS and AAX audio plug-ins.* https://github.com/juce-framework/JUCE. (Accessed on 12/11/2021).

[14] *The Eight Phases of a DevOps Pipeline — by Jakob Pennington — Taptu — Medium*. https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba. (Accessed on 02/05/2022).

[15] *What is a DevOps Toolchain? – BMC Software — Blogs*. https://www.bmc.com/blogs/devops-toolchain/. (Accessed on 02/06/2022).

Ringraziamenti

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito alla realizzazione di questo elaborato.

Un ringraziamento particolare va al mio relatore Tramontana Porfirio che mi ha seguito, con la sua infinita disponibilità, in ogni step della realizzazione dell'elaborato, fin dalla scelta dell'argomento.
Grazie anche al mio correlatore Mirabile Carmelo per i suoi preziosi consigli e per avermi suggerito puntualmente le giuste modifiche da apportare alla mia tesi.

Ringrazio infinitamente la mia famiglia: senza i loro insegnamenti ed il loro supporto, questo lavoro di tesi non esisterebbe nemmeno.

Grazie a tutti i miei colleghi di corso, per avermi sempre incoraggiato fin dall'inizio del percorso universitario.

Ringrazio la mia ragazza Francesca per avermi trasmesso la sua immensa forza e il suo coraggio.
Grazie per tutto il prezioso tempo che mi hai dedicato, per aver ascoltato i miei sfoghi e per tutti i momenti di spensieratezza.
Grazie perché ci sei sempre stata... e sempre ci sarai.

Grazie a tutti i miei amici, fonti di distrazione e divertimento senza i quali non saprei neanche cosa farne di questa tesi.
Un ringraziamento particolare ad Esito e Bchele... siete veramente straordinari.