Politecnico di Torino

Politecnico di Torino

Corso di Laurea in Ingegneria Gestionale
Classe LM-31
A.a. 2025/2026
Sessione di Laurea Marzo 2026

Tesi di Laurea Magistrale

Analisi empirica e ottimizzazione del Project Management nelle startup IT: il caso Nano i-Tech

Relatore:
Prof. Giulio Mangano

Candidato:
Mattia Ghigo 329989

Sommario

Lista figure

Lista tabelle

Lista formule

Capitolo 1: Introduzione

L'industria del software è stata storicamente caratterizzata dalla ricerca costante di metodologie e strumenti volti a ottimizzare il ciclo di vita dello sviluppo, *Software Development Lifecycle* (*SDLC*). Negli ultimi anni, tale ricerca ha subito un'accelerazione senza precedenti con l'introduzione dei *Large Language Models* (*LLM)* applicati alla scrittura del codice.

Strumenti come *GitHub Copilot*, *Gemini* e *ChatGPT* hanno fatto il loro ingresso nei team di sviluppo promettendo drastici aumenti di efficienza. Studi recenti condotti da enti autorevoli, come la ricerca di McKinsey & Company (2023), stimano che l'uso dell'*AI* generativa possa incrementare la produttività degli sviluppatori tra il 20% e il 45% a seconda della complessità del task [1], mentre esperimenti controllati da *GitHub* riportano un aumento della velocità di completamento dei task fino al 55% [2].

Tuttavia, queste metriche si concentrano prevalentemente sulla velocità di scrittura, *Coding Speed*. Dal punto di vista del *Project Management*, la velocità di scrittura risulta essere solamente una delle tante variabili da attenzionare. La vera sfida per le aziende moderne non è solo scrivere codice rapidamente, ma garantire la prevedibilità dei rilasci e la qualità del prodotto immesso sul mercato. In questo contesto, l'introduzione di nuove tecnologie agisce come un fattore di perturbazione: se da un lato accelera l'esecuzione, dall'altro introduce nuove variabili di rischio che possono compromettere la pianificazione se non adeguatamente governate.

1.1 Il contesto operativo: la sfida della gestione in una startup

Lo scenario in cui si inserisce questa ricerca è l'ecosistema ad alta incertezza di una *startup* innovativa. Le *startup* operano strutturalmente in condizioni di risorse limitate e pressione temporale, *Time to Market*. In tale contesto, il rispetto delle scadenze non è un semplice KPI operativo, ma una condizione necessaria per la sostenibilità del business e la fiducia degli investitori.

Qui emerge il ruolo strategico del *Project Management*: passare da una gestione a braccio, tipica delle fasi embrionali, a un processo strutturato e misurabile, come nelle grandi aziende consolidate. Il compito del *Project Manager* è identificare e mitigare le cause dei ritardi, che in un ambiente giovane e dinamico, come quello delle *startup*, possono essere molteplici: dalla poca chiarezza dei requisiti, alla mancanza di esperienza del team, fino all'adozione di tecnologie sperimentali. L'obiettivo di questa Tesi è analizzare come queste variabili interagiscano tra loro, rendendo difficile la stima dei tempi, *Estimation Accuracy*.

1.2.1 Il caso di studio: l'ecosistema Insurtech di Nano i-Tech

Lo studio empirico alla base di questa Tesi è stato condotto all'interno di *Nano i-Tech S.r.l.*, una *startup* innovativa fondata nel 2021 a Saluzzo (CN), che rappresenta oggi un'eccellenza emergente nel panorama Insurtech italiano. Nata dall'intuizione di Sebastiano Andreis (CEO & Co-founder) e Simone Alberto (CTO & Co-founder), l'azienda si è posta fin da subito l'obiettivo di colmare il divario tecnologico del settore assicurativo, applicando logiche di sviluppo avanzato a un mercato tradizionalmente statico.

Nonostante la giovane età, *Nano i-Tech* è riuscita a dimostrare una maturità industriale che l'ha portata a stringere una partnership strategica con *eVISO S.p.A.*, società tecnologica quotata sul mercato *Euronext Growth Milan*, leader nell'intelligenza artificiale applicata alle *commodities*. L'ingresso di un *player* quotato nel capitale sociale ha segnato un punto di svolta: il team di sviluppo non ha operato in un contesto di pura sperimentazione, bensì sotto l'osservazione di *stakeholder* industriali esigenti, dove il rispetto delle scadenze e la qualità del software sono vincolati a precisi obiettivi di business.

L'attività di R&D dell'azienda si focalizza su due progetti principali, entrambi caratterizzati da un elevato profilo di rischio ed innovazione.

- **IMBA - IlMioBrokerAssicurativo (B2C):** applicazione *web-responsive* rivolta all'utente finale. Funge da comparatore di polizze e da portafoglio digitale, permettendo ai clienti di archiviare le proprie coperture assicurative e di richiedere consulenze in tempo reale.
- **Navisio (B2B SaaS):** piattaforma gestionale *cloud-based* progettata per intermediari assicurativi e broker. Il software permette la gestione completa del portafoglio clienti, la preventivazione automatizzata e la gestione dei sinistri, integrandosi via API con le principali compagnie assicurative. Rappresenta il *core business* tecnologico dell'azienda.

L'ambizione di competere con *player* internazionali, mantenendo la struttura agile di una *startup,* ma il rigore di una società partecipata, ha reso necessario l'utilizzo massiccio di strumenti di codifica generativa, *AI-Augmented Coding*. È in questo dualismo tra velocità d'esecuzione, garantita dall'*AI*, e affidabilità industriale, richiesta dagli investitori, che si inserisce il problema di ricerca di questa Tesi.

1.2 Definizione del problema: le cause invisibili dell'errore di stima

Il cuore del problema affrontato in questo lavoro è la discrepanza sistematica tra ciò che viene pianificato e ciò che viene effettivamente consegnato, *Planning Variance*. Nelle metodologie di

Project Management tradizionali, si assume che l'errore di stima derivi principalmente da requisiti poco chiari o imprevisti tecnici. Tuttavia, l'osservazione sul campo ha suggerito l'esistenza di fattori più sottili, legati alle dinamiche di lavoro quotidiane.

L'introduzione dell'*AI* nel flusso di sviluppo è stata identificata come un amplificatore per quanto riguarda queste problematiche. Sebbene l'*AI* venga adottata per ridurre i tempi, paradossalmente si è osservato che in assenza di processi di controllo rigorosi, può diventare essa stessa una causa di ritardo. Il problema, dunque, non è l'*AI* in sé, ma la mancanza di un modello di gestione capace di adattarsi ai nuovi flussi di lavoro. Come può un *Project Manager* garantire la consegna puntuale, *On-Time Delivery*, quando il team utilizza strumenti che aumentano il volume di codice ma rischiano di abbassarne la comprensione e la qualità? Capire le cause di questo fenomeno, che siano esse tecniche, umane o organizzative, è il primo passo per risolverlo.

1.3 Obiettivi della Tesi e domanda di ricerca

L'obiettivo primario di questo elaborato è fornire una diagnosi analitica delle cause di inefficienza e dei ritardi nello sviluppo software, focalizzandosi sull'interazione complessa tra sviluppatori *Junior*, strumenti di Intelligenza Artificiale Generativa e complessità del progetto.

Attraverso l'analisi di dati empirici raccolti in un arco temporale di 8 mesi, la Tesi si propone di rispondere ai seguenti quesiti di ricerca.

- **Analisi delle determinanti:** quali sono i fattori critici che determinano scostamenti significativi tra l'*effort*, stimato in fase di pianificazione, e i valori rilevati a consuntivo all'interno di un team operante in una *startup*?
- **Impatto dell'automazione:** in che misura l'adozione dell'*AI* incide sulla varianza di pianificazione? L'utilizzo di tali strumenti agisce come un acceleratore della produttività, *Velocity*, o, al contrario, come un generatore di instabilità che incrementa il tasso di rilavorazione, *Rework*?
- **Strategie di controllo:** quali metriche e processi di *Project Management* possono essere implementate per recuperare prevedibilità e stabilità, trasformando l'innovazione tecnologica in un reale vantaggio competitivo?

Il risultato finale intende offrire un contributo pratico ai *Project Manager* che operano in contesti ad alta innovazione, proponendo un *framework* metodologico per interpretare i dati di sviluppo e correggere le stime in modo proattivo. Più nello specifico, l'elaborato mira a colmare il divario tra le metriche tecniche di basso livello, come il *Code Churn* o la densità di codice generato, e le decisioni strategiche di alto livello.

L'obiettivo ultimo è dotare la governance di progetto di uno strumento analitico, definibile come *AI-Safe Framework*, capace di trasformare l'incertezza introdotta dall'Intelligenza Artificiale da un rischio imprevedibile a un parametro calcolabile. Questo approccio permetterà ai gestori dei progetti non solo di monitorare lo scostamento tra l'avanzamento effettivo e quello preventivato, ma anche di applicare coefficienti correttivi alle *roadmap* in tempo reale, facilitando un allineamento trasparente e *data-driven* con gli *stakeholder* e gli investitori.

Capitolo 2: Analisi della letteratura

L'integrazione dell'Intelligenza Artificiale Generativa nei processi di sviluppo software non rappresenta un semplice aggiornamento tecnologico, ma una forza trasformativa che impone di ripensare le fondamenta stesse della disciplina. Questo capitolo si propone di costruire il quadro teorico indispensabile per interpretare le dinamiche osservate nel caso studio, partendo da un confronto critico sulle metodologie di sviluppo per comprendere come i modelli tradizionali e agili vengano alterati dall'accelerazione imposta dagli strumenti automatici.

Attraverso un'analisi puntuale dei paper scientifici recenti, verrà esplorato il delicato equilibrio tra l'aumento della velocità produttiva, la cosiddetta *Velocity*, e il rischio concreto di un degrado della sicurezza e nella stabilità del codice, un fenomeno spesso occultato da un eccesso di fiducia verso i suggerimenti della macchina. Infine, per superare una valutazione puramente qualitativa, sarà definito un rigoroso insieme di metriche, dal debito tecnico al *Code Churn*, fino all'analisi dei fattori umani come il *Bus Factor*. Questi strumenti concettuali ci permetteranno di misurare con precisione i costi nascosti di uno sviluppo non supervisionato, fornendo la chiave di lettura per comprendere le reali cause dei ritardi e dei fallimenti nei progetti di *Nano i-Tech*.

2.1 Il Project Management nello sviluppo software: dal controllo all'adattabilità

L'evoluzione delle metodologie di gestione dei progetti software riflette il passaggio storico da un'economia industriale, basata sulla prevedibilità, ad un'economia della conoscenza, caratterizzata da alta volatilità ed incertezza. Per comprendere le dinamiche osservate nel caso studio relativo a *Nano i-Tech*, è necessario analizzare la dicotomia tra approcci predittivi, *Waterfall*, e adattivi, *Agile*, per poi inserire tali modelli nel contesto specifico delle *startup Insurtech*.

L'approccio predittivo e i limiti del Waterfall

Storicamente, l'ingegneria del software ha adottato il modello a cascata, noto come *Waterfall Model*: un approccio lineare e sequenziale derivato dai processi manifatturieri. Come descritto da McCormick (Figura 1), questo modello suddivide il ciclo di vita del software in fasi distinte e non sovrapponibili: Requirements, Analysis, Design, Coding, Testing e Acceptance [3]. La caratteristica fondamentale di questo approccio è la necessità di completare interamente una fase prima di procedere alla successiva, con un'enfasi massiccia sulla documentazione preventiva e sulla stabilità dei requisiti iniziali.

Tuttavia, i paper evidenziano come questo rigore formale si scontri spesso con la realtà operativa. Nello studio comparativo condotto da Mishra e Alzoubi, emerge che i progetti gestiti con metodologia *Waterfall* presentano un tasso di fallimento del 30%, contro il 10% dei progetti *Agile* [4]. La rigidità del *Waterfall*, sebbene offra un'illusione di controllo, si rivela inadeguata nei contesti in cui i requisiti non sono chiari fin dall'inizio o cambiano in corso d'opera. Nonostante questo, Boehm e Turner introducono il concetto di *Home Grounds*, sostenendo che gli approcci predittivi rimangono superiori in contesti specifici caratterizzati da bassa volatilità dei requisiti e alta criticità, come per software *safety-critical*, dove la stabilità è prioritaria rispetto alla velocità [5].

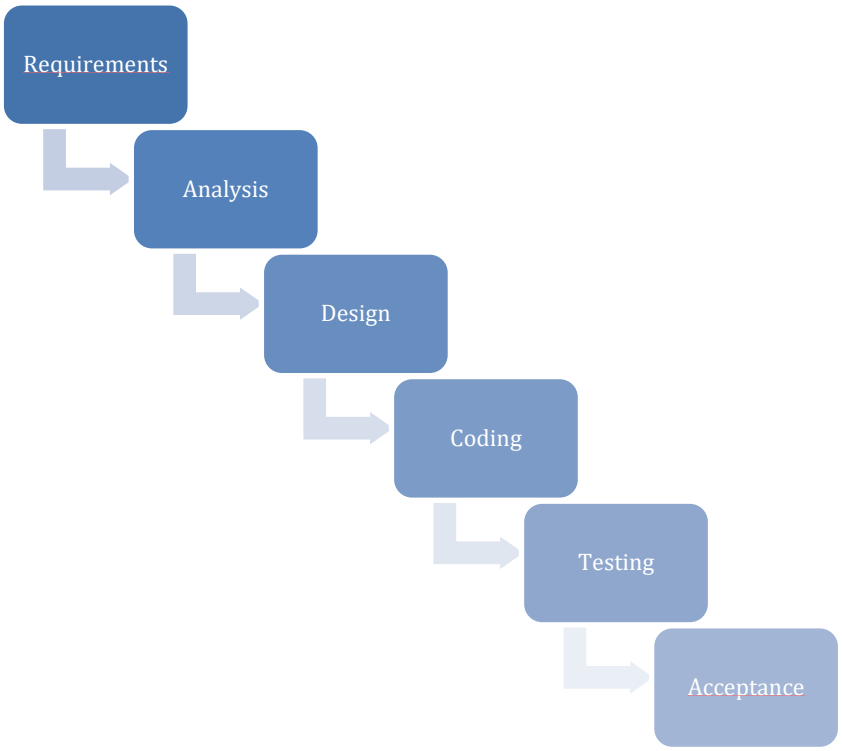

Figura 1: modello Waterfall.

La rivoluzione Agile e la gestione dell'incertezza

In risposta alle inefficienze dei modelli predittivi, si è affermato il paradigma *Agile* (Figura 2). Secondo la revisione sistematica della letteratura condotta da Dybå e Dingsøyr, l'*Agile* non va inteso come una semplice tecnica di *Project Management*, ma come una filosofia che sposta il focus dai processi alle persone e dalla negoziazione contrattuale alla collaborazione continua con il cliente finale [6]. I dati empirici supportano questa transizione: Mishra riporta che le iniziative *Agile* presentano un tasso di successo del 40%, significativamente superiore al 15% del *Waterfall*,

grazie alla capacità di gestire il cambiamento attraverso iterazioni brevi, definite *Sprint*, e feedback continui [4].

Tuttavia, l'*Agile* non è esente da rischi. Wishnie sottolinea come l'assenza di disciplina ingegneristica possa trasformare l'agilità in caos. Il successo dell'*Agile* richiede un bilanciamento tra flessibilità e rigore; senza questo equilibrio, i team rischiano di cadere in pratiche di *hacking* destrutturato piuttosto che di sviluppo iterativo [5]. È proprio in questa zona grigia tra agilità e disciplina che operano le *startup*, il cui obiettivo primario spesso non è la qualità del codice, ma la sopravvivenza sul mercato.

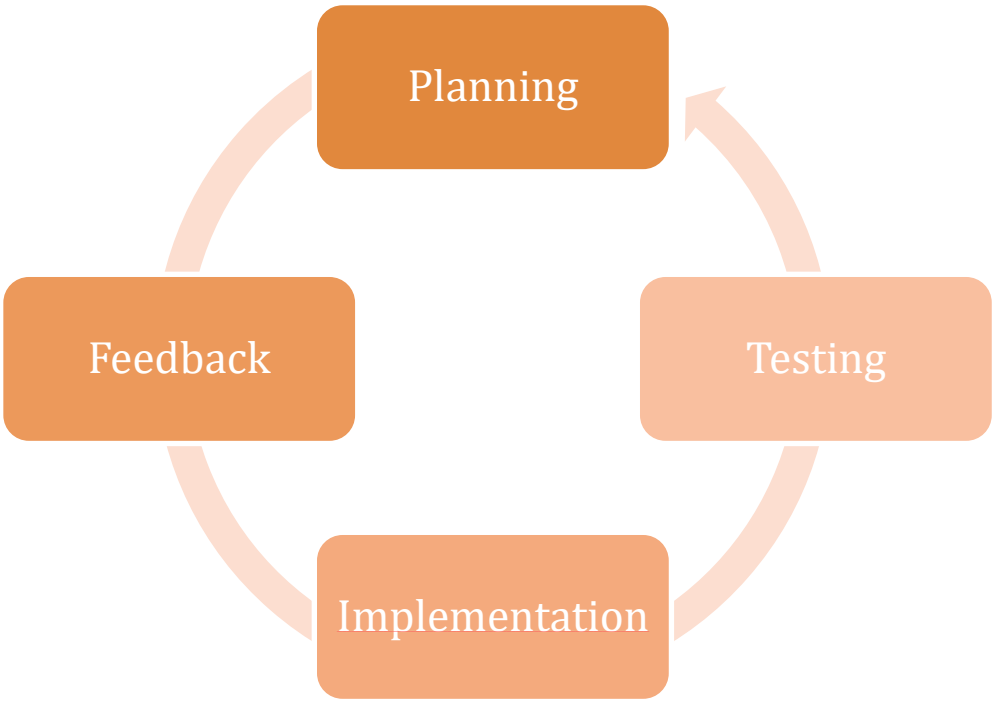

Figura 2: ciclo iterativo dello sviluppo Agile.

Il Framework Scrum e l'adattamento nelle startup

Tra le implementazioni dei principi *Agile*, il framework *Scrum* (Figura 3) è quello che ha trovato maggiore diffusione empirica. Secondo la classificazione di Dybå e Dingsøyr, lo *Scrum* si distingue per il suo approccio iterativo-incrementale strutturato in cicli temporali fissi detti *Sprint*, tipicamente di 2 o 4 settimane, gestiti attraverso tre artefatti principali: il *Product Backlog*, ovvero la lista dei requisiti, lo *Sprint Backlog*, comprendente i task del ciclo corrente, e il *Burndown Chart*, con il quale si indica il monitoraggio del progresso [6]. Il *framework* prescrive inoltre rituali specifici come il *Daily Stand-up meeting* per la sincronizzazione del team e la *Sprint Retrospective* per il miglioramento continuo.

Tuttavia, l'applicazione dello *Scrum* nel contesto delle *startup* subisce spesso notevoli deviazioni rispetto alla teoria. Paternoster e colleghi, rilevano che, sebbene molte *startup* dichiarino di adottare lo *Scrum*, l'implementazione è spesso parziale o adattata per far fronte alla carenza di risorse. Pratiche fondamentali come il testing automatizzato o le retrospettive formali vengono frequentemente sacrificate per mantenere l'alta velocità di rilascio richiesta dal mercato [7]. Questa modalità operativa, definita talvolta come *Scrum-but*, al fine di indicare che: *si fa Scrum, ma senza...;* descrive fedelmente il contesto iniziale di *Nano i-Tech*, dove la struttura dei rituali, come *Sprint* e *Daily*, era presente, ma mancava il rigore ingegneristico necessario a prevenire il debito tecnico.

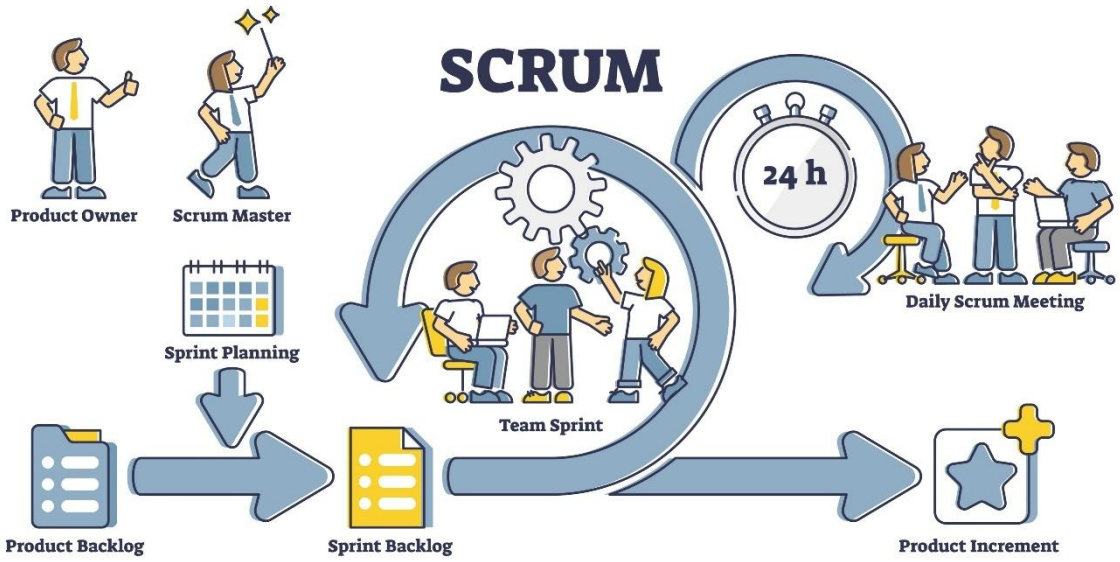

Figura 3: framework Scrum e flusso di lavoro dello Sprint.

La gestione del flusso: il metodo Kanban

Mentre lo *Scrum* impone una cadenza temporale ritmica, gli *Sprint*, molte *startup* integrano o sostituiscono questo approccio con il *Kanban* (Figura 4), una metodologia derivata dal *Lean Manufacturing*, *Toyota Production System*. Nella rassegna di Dybå e Dingsøyr, il *Kanban* viene inquadrato nelle pratiche di *Lean Development*, focalizzate sulla riduzione degli sprechi, *Muda*, e sull'ottimizzazione del flusso di valore [6].

I principi cardine del *Kanban*, fondamentali per comprendere le metriche di flusso analizzate in questa Tesi, come il *Cycle Time*, sono due.

- **Visualizzazione del lavoro:** l'uso di una *Kanban Board*, fisica o digitale come *Jira*, permette di rendere esplicito lo stato di ogni *task*, aumentando la trasparenza richiesta nei team piccoli descritti da Paternoster [7].

- **Limitazione del Work In Progress - WIP:** a differenza dello *Scrum*, che limita il lavoro per intervalli di tempo, *Kanban* limita la quantità di *task* paralleli. Questo meccanismo serve a prevenire i colli di bottiglia e il *Context Switching* eccessivo.

Nel contesto *Greenfield* descritto da Giardino, dove la pressione per il rilascio è massima, il *Kanban* viene spesso adottato per la sua flessibilità: permette di inserire task urgenti in qualsiasi momento, senza aspettare il prossimo *Sprint Planning*, a patto che la capacità del team lo consenta [8]. Tuttavia, come si vedrà nei dati di *Nano i-Tech*, questa flessibilità estrema, se supportata dalla velocità di generazione dell'*AI*, rischia di saturare il flusso, trasformando la limitazione del *WIP* in un accumulo incontrollato di codice da revisionare.

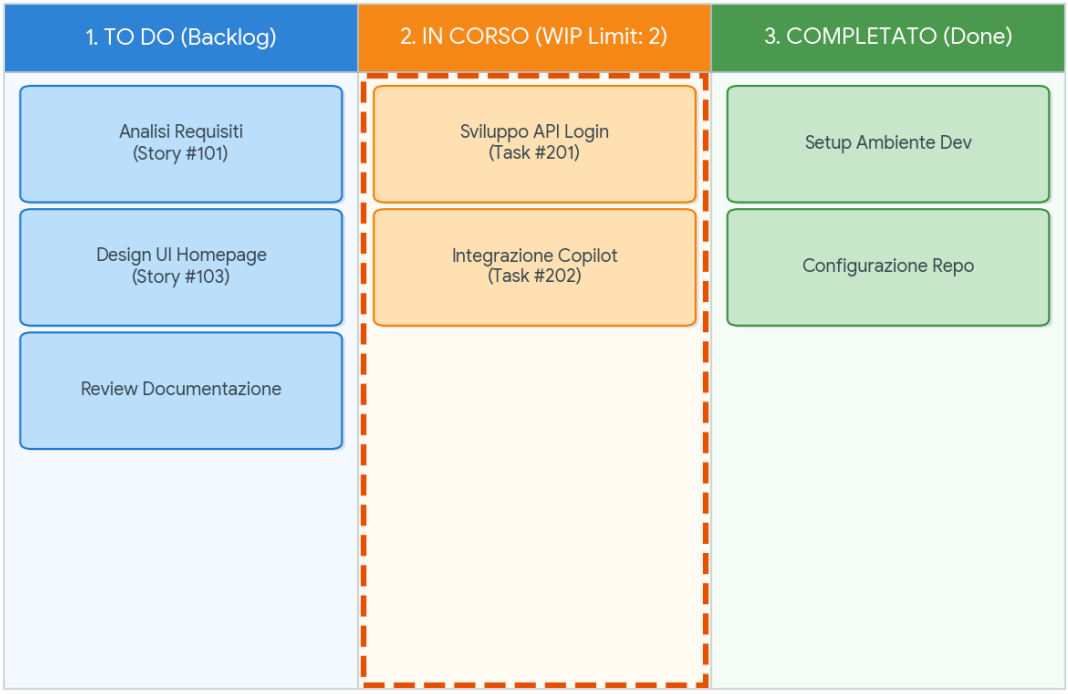

Figura 4: gestione del flusso tramite Kanban Board e limiti WIP.

Il Greenfield Startup Model: la velocità come imperativo di sopravvivenza

Il contesto operativo delle *startup software* differisce radicalmente da quello delle grandi aziende consolidate. Paternoster e colleghi definiscono le *startup* come organizzazioni che operano in condizioni di estrema incertezza, forti pressioni temporali, *time-pressure*, e grave carenza di risorse, sia umane sia economiche [7]. In questo scenario, le pratiche di ingegneria del software vengono spesso adattate o ignorate per favorire la velocità.

L'analisi più pertinente per il caso studio oggetto di Tesi è fornita dal *Greenfield Startup Model* (*GSM*), teorizzato da Giardino e colleghi. Secondo questo modello, le *startup* nelle fasi iniziali,

denominate *Early-stage*, prioritizzano una strategia di *Speed-up development* per rilasciare il prodotto il prima possibile e validare il mercato. Questa scelta strategica comporta una conseguenza inevitabile e consapevole: l'accumulo di debito tecnico [8]. Giardino spiega che nelle *startup*:

- si riducono al minimo le attività di *testing* e di documentazione;
- si accetta una bassa precisione ingegneristica per abbreviare il *Time-to-Market*;
- il debito tecnico non è visto come un errore, ma come uno strumento di investimento per guadagnare tempo.

Questa ricerca fornisce la chiave di lettura per i comportamenti osservati in *Nano i-Tech*: l'uso dell'*AI* da parte dei profili *Junior* per generare grandi quantità di codice senza *refactoring* non è un'anomalia, ma la perfetta applicazione, forse estremizzata, del modello *Greenfield* descritto da Giardino.

Il paradosso Insurtech: scontro tra agilità e compliance

L'ultimo livello di complessità è dato dal dominio applicativo: l'applicazione di tecnologie digitali al settore assicurativo, *Insurtech*. Caleb descrive l'*Insurtech* come un settore che tenta di applicare l'innovazione rapida, come *AI*, *IoT* e *Big Data*, ad un'industria tradizionalmente conservatrice e iper-regolamentata [9]. Questo crea un modello operativo bimodale:

- da un lato, la necessità di agilità per sviluppare interfacce cliente moderne e *pricing* personalizzati;
- dall'altro, si contrappone la rigida necessità di solidità strutturale e conformità normativa, indispensabili per trattare asset critici come dati sensibili e contratti legali in accordo con vincoli quali GDPR e indici di solvibilità.

Questa tensione crea attrito. Palfreyman e Morton evidenziano come la trasformazione digitale in questi settori richieda non solo nuovi strumenti, ma un cambiamento culturale profondo per allineare i team di sviluppo, che vogliono correre, con i dipartimenti di rischio e *compliance*, che devono frenare [10]. Nel caso studio di questa Tesi, si vedrà come questo disallineamento sia esploso quando il codice generato velocemente dall'*AI*, seguendo il modello *Greenfield*, si è scontrato con i requisiti rigidi del *backend* assicurativo, causando i picchi di errore di stima analizzati nel Capitolo 4.

2.2 L'impatto dell'AI generativa nel ciclo di sviluppo software

L'introduzione dei *Large Language Models* (*LLM)* nel flusso di lavoro degli sviluppatori rappresenta un cambiamento di paradigma che va oltre la semplice automazione. Se le metodologie *Agile* hanno cercato di ottimizzare il processo, l'*AI* Generativa interviene

direttamente sulla capacità produttiva del singolo individuo, alterando le metriche tradizionali di *Velocity* e introducendo nuove categorie di rischio. Per analizzare questo fenomeno, ci affidiamo a tre studi fondamentali condotti nel 2023 da Harvard Business School, Microsoft Research e Stanford University, che delineano il quadro teorico della *Frontiera Tecnologica Frastagliata*.

L'accelerazione della produttività e l'effetto Leveling

Il primo impatto misurabile dell'*AI* è un drastico aumento della velocità nella scrittura del codice. Nello studio controllato condotto da Peng e colleghi su *GitHub Copilot*, è stato chiesto a un campione di sviluppatori di completare un *task* standardizzato, ovvero la scrittura di un server *HTTP* in *JavaScript*. I risultati sono inequivocabili: il gruppo assistito dall'*AI* ha completato il compito il 55.8% più velocemente rispetto al gruppo di controllo, 71.17 minuti contro 160.89 minuti (Figura 5) [12].

Tuttavia, il dato più rilevante per il contesto di una *startup* con profili *Junior* è quello che Peng definisce effetto eterogeneo o di livellamento. L'*AI* avvantaggia in misura sproporzionata gli sviluppatori con meno esperienza, permettendo loro di colmare il gap produttivo con i profili *Senior*. Mentre i programmatori esperti vedono incrementi marginali, i profili meno esperti ottengono un *boost* di produttività massiccio, riducendo la varianza delle prestazioni all'interno del team [12]. Questo fenomeno spiega l'illusione di competenza osservata inizialmente in *Nano i-Tech*: l'*AI* ha permesso ai *Junior* di generare codice a ritmi da *Senior*, nascondendo però la mancanza di comprensione profonda delle architetture sottostanti.

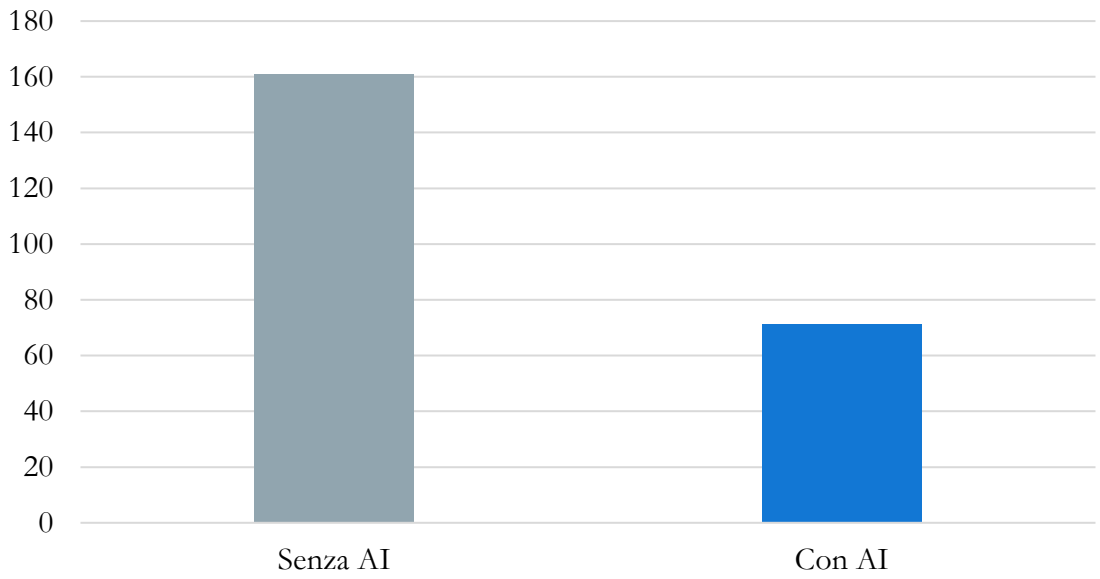

Figura 5: confronto dei tempi di completamento per un task di programmazione standardizzato.

La Jagged Technological Frontier - frontiera frastagliata

Se la velocità aumenta, la qualità segue una traiettoria non lineare. Dell'Acqua e colleghi, Harvard Business School, introducono il concetto di *Jagged Technological Frontier* per descrivere le capacità dell'*AI*. A differenza delle tecnologie precedenti, che offrivano prestazioni uniformi, essa mostra un confine irregolare (Figura 6): eccelle in alcuni compiti complessi, ma fallisce inaspettatamente in altri apparentemente semplici [11].

Lo studio di Dell'Acqua dimostra che:

- **all'interno della frontiera**, per i *task* che rientrano nelle capacità del modello, l'uso dell'*AI* porta a un aumento del 12.2% nel completamento dei *task*, del 25.1% nella velocità e del 40% nella qualità del risultato;
- **fuori dalla frontiera**, per i task che richiedono una comprensione contestuale sottile o dati non presenti nel training set, l'uso dell'*AI* è dannoso. La correttezza delle soluzioni crolla di 19 punti percentuali rispetto a chi non la usa [11].

Il rischio, per un team di sviluppo, è l'invisibilità di questa frontiera. L'*AI* risponde con la stessa sicurezza, o allucinazione convincente, sia che il *task* sia dentro sia che sia fuori la frontiera, rendendo difficile per un programmatore non esperto distinguere una soluzione geniale da una catastrofica.

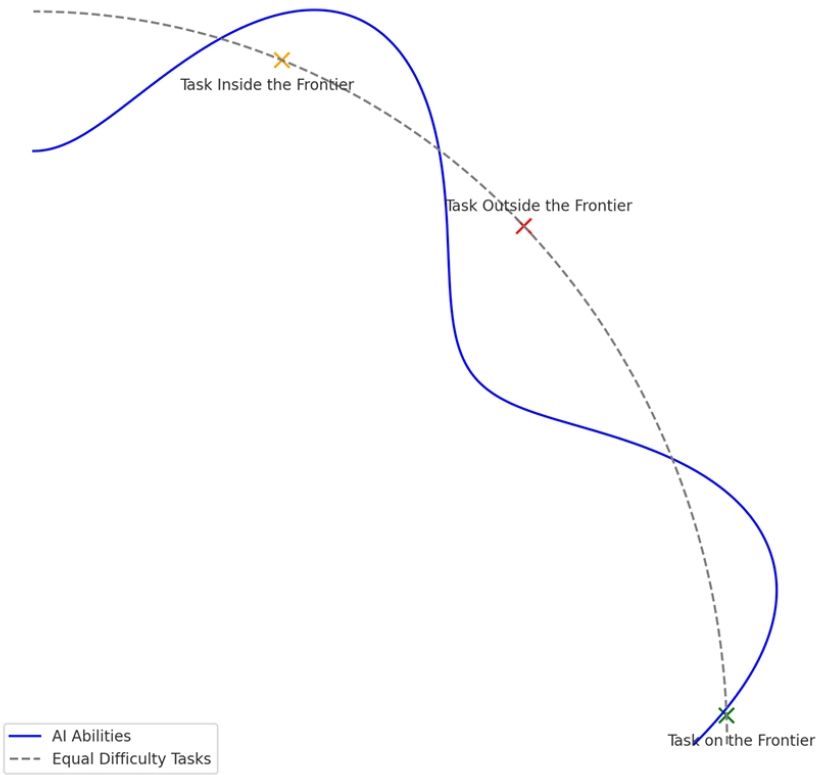

Figura 6: frontiera tecnologica frastagliata delle capacità dell'IA.

Sicurezza del codice e Automation Bias

Il prezzo di questa velocità indiscriminata è spesso pagato in termini di sicurezza informatica. Lo studio di Perry e colleghi, Stanford, indaga specificatamente se gli assistenti di scrittura codice portino a scrivere del codice meno sicuro. I risultati sono allarmanti: i partecipanti con accesso all'*AI* hanno prodotto codice contenente vulnerabilità critiche, come *SQL Injection* o uso di cifrari deboli, in misura significativamente maggiore rispetto a chi scriveva manualmente. Nello specifico *task* di *coding*, solo il 3% degli utenti che hanno utilizzato l'*AI* ha scritto codice sicuro, contro il 21% del gruppo di controllo (Figura 7) [13].

Ciò che rende questo dato critico è il fattore psicologico noto come *Automation Bias*. Perry rileva che, nonostante il loro codice fosse oggettivamente peggiore, gli sviluppatori assistiti dall'*AI* avevano una fiducia significativamente maggiore nella correttezza del proprio lavoro rispetto a chi non ne faceva uso [13]. Questo eccesso di fiducia, *Overconfidence*, disattiva i meccanismi di controllo critico: lo sviluppatore *Junior* tende ad accettare l'output della macchina come se fosse una verità validata, trasferendo il codice in produzione, o in *Code Review*, senza un'adeguata verifica. Questo comportamento è la causa scatenante del collo di bottiglia nelle revisioni che analizzeremo nel Capitolo 4.

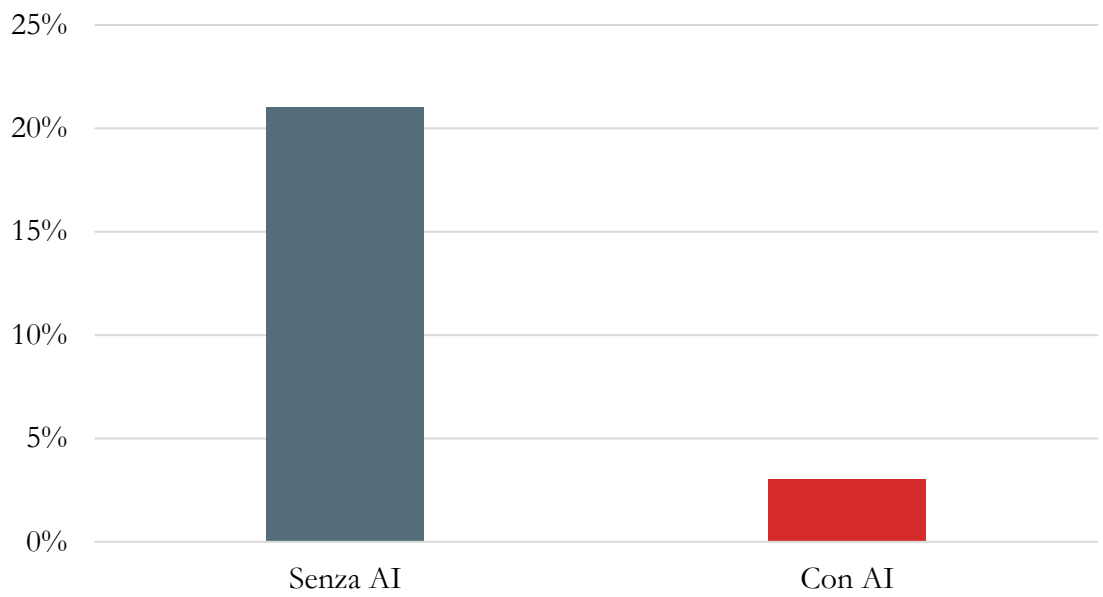

Figura 7: confronto sulla sicurezza del codice: sviluppo manuale vs assistito da AI.

Modelli di integrazione uomo-macchina: centauri vs cyborg

Infine, la letteratura suggerisce che il successo o il fallimento nell'adozione dell'*AI* dipenda dalla modalità di interazione. Dell'Acqua e colleghi identificano due archetipi comportamentali.

- **Centaurs:** utenti che mantengono una chiara divisione dei compiti, delegando all'*AI* solo le attività dentro la frontiera e riprendendo il controllo manuale per quelle complesse. È un approccio strategico che richiede alta conoscenza e competenza;
- **Cyborgs:** utenti che integrano completamente il flusso di lavoro con la macchina, interagendo continuamente a livello di *micro-task* e delegando gran parte del processo cognitivo [11].

L'analisi dei paper suggerisce che i profili *Junior* tendono naturalmente verso il modello *Cyborg*, ovvero all'accettazione passiva e continua del codice scritto dall'*AI*, massimizzando la velocità ma esponendosi ai rischi della frontiera frastagliata. Al contrario, la mitigazione del rischio richiederebbe un approccio da *Centauro*, tipico dei *Senior*, capace di discernere quando l'*AI* sta incappando in errori. La mancata transizione da *Cyborg* a *Centauro* è quindi la principale causa del debito tecnico involontario.

2.3 Metriche di qualità del software e fattori umani

Per valutare oggettivamente l'impatto dell'*AI* Generativa sui processi di sviluppo descritti nelle sezioni precedenti, è necessario definire un set di metriche sia di tipo quantitativo sia di tipo qualitativo. La letteratura scientifica ci fornisce strumenti precisi per misurare non solo la qualità statica del codice, ma anche la sua evoluzione dinamica nel tempo e l'impatto cognitivo sul team di sviluppo.

In questa sezione, verranno analizzati il debito tecnico, il *Code Churn*, la complessità ciclomatica e i fattori psicologici, come *Bias* e carico cognitivo, che determinano la sostenibilità di un progetto software.

Tassonomia del debito tecnico: Reckless vs Prudent

Il concetto di *Technical Debt*, introdotto originariamente da Ward Cunningham, descrive il costo implicito del lavoro aggiuntivo causato dalla scelta di una soluzione più facile e rapida invece di un approccio migliore che richiederebbe, ovviamente, più tempo.

Tuttavia, non tutto il debito è uguale. Trumler & Paulisch [14], riprendendo il modello del *Technical Debt Quadrant* di Martin Fowler (Figura 8), distinguono quattro tipologie di debito basate su due assi: l'intenzionalità, *Deliberate* vs *Inadvertent*, e la competenza, *Prudent* vs *Reckless*.

- **Deliberate / Reckless:** non abbiamo tempo per il design;

- **Deliberate / Prudent:** dobbiamo rilasciare ora, affronteremo le conseguenze dopo, tipico degli MVP consapevoli;
- **Inadvertent / Reckless:** che cos'è il Layering?;
- **Inadvertent / Prudent:** ora sappiamo come avremmo dovuto farlo.

Nel contesto dello sviluppo software assistito dall'*AI*, l'ipotesi principale è che l'uso irrazionale di strumenti come *Copilot* da parte di profili *Junior* spinga l'accumulo di debito nel quadrante Inadvertent/Reckless. Gli sviluppatori, fidandosi della correttezza sintattica del codice generato, introducono difetti strutturali senza esserne realmente consapevoli. Trumler nota come l'assenza di pratiche rigorose come il *Test Driven Development* (*TDD*) trasformi rapidamente questo debito invisibile in un ostacolo insormontabile per la manutenzione futura, aumentando esponenzialmente la densità dei difetti [14].

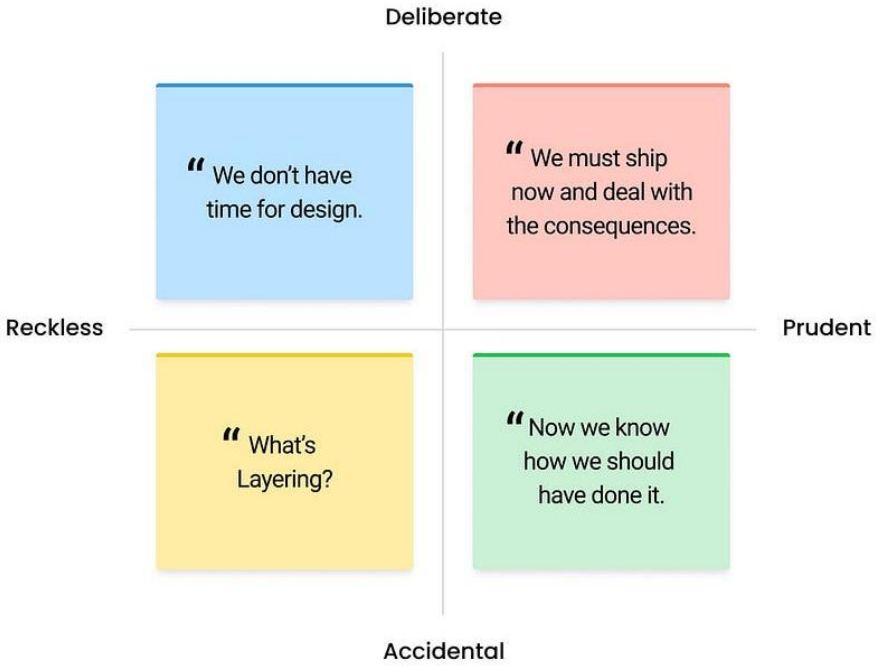

Figura 8: quadranti del debito tecnico di Fowler.

Code Churn: instabilità e predizione dei difetti

Mentre il debito tecnico è una metafora, il *Code Churn* è una misura fisica dell'instabilità del sistema. Nagappan & Ball [15] definiscono il *Relative Code Churn* come la quantità di codice aggiunto, modificato o cancellato in uno script rispetto alla sua dimensione totale in un determinato arco temporale.

Lo studio di Nagappan dimostra che il *Code Churn* è un predittore di difettosità del software significativamente più accurato rispetto alle metriche statiche tradizionali, come le linee di codice (*LOC*). Esiste una forte correlazione positiva tra un alto tasso di *Churn* e la densità di *bug* post-rilascio.

Questa metrica è particolarmente rilevante nell'era dell'*AI* Generativa. Il flusso di lavoro tipico dell'interazione con un *LLM*

$$Prompt \rightarrow Codice\ Errato \rightarrow Refining\ del\ Prompt \rightarrow Riscrittura\ del\ codice$$

genera un fenomeno di scuotimento, *Thrashing*: ampie porzioni di codice vengono generate e rigenerate in tempi brevissimi. Secondo i modelli di Nagappan, questo comportamento non è indice di alta produttività, ma un segnale di allarme precoce, *Early Warning*, di instabilità sistemica. Il codice che subisce troppe variazioni in poco tempo è statisticamente destinato a fallire in produzione [15].

La densità di complessità ciclomatica

La complessità del flusso di controllo è tradizionalmente misurata tramite la complessità ciclomatica di McCabe $v(G)$, che conta il numero di cammini linearmente indipendenti attraverso il codice sorgente, determinato dal numero di ramificazioni condizionali come *if*, *while* e *for*.

Tuttavia, Gill & Kemerer [16] argomentano che la complessità assoluta è insufficiente per valutare la manutenibilità. Introducono quindi il concetto di *Cyclomatic Complexity Density*, calcolata come il rapporto tra la complessità ciclomatica e la dimensione del modulo, *LOC*.

$$Complexity\ Density = \frac{v(G)}{LOC} \quad [1]$$

Questa distinzione è cruciale per analizzare il codice generato dall'*AI*. Gli *LLM* tendono spesso a essere verbosi o a produrre logiche ridondanti se non ottimizzati. Un'alta densità di complessità indica che la logica decisionale è compressa in poche righe, rendendo il codice estremamente difficile da comprendere, *Cognitive Dense*, per un revisore umano. Gill & Kemerer dimostrano che l'alta densità è il singolo fattore più impattante sulla riduzione della produttività di manutenzione: più il codice è denso di logica, anche se sintatticamente corretto, più tempo richiederà per essere modificato o corretto in futuro [16].

Il fattore umano: Bus Factor e carico cognitivo

La sostenibilità del software non dipende solo dal codice, ma dalla distribuzione della conoscenza all'interno del team.

Jabrayilzade e colleghi [17] analizzano il *Bus Factor*: una metrica che indica il numero minimo di membri del team che devono abbandonare il progetto, o *essere investiti da un autobus*, affinché il progetto si blocchi per mancanza di conoscenza critica.
In un team dove i profili *Junior* delegano la comprensione del problema all'*AI*, si verifica un paradosso: il codice viene generato, ma la conoscenza di come funziona non viene internalizzata dallo sviluppatore. Di fatto, la conoscenza risiede nel modello *AI*, non accessibile, o nell'unico *Senior* in grado di revisionare il tutto. Questo riduce il *Bus Factor* reale a livelli pericolosamente bassi, spesso pari a 1, rendendo il progetto estremamente fragile al *turnover* del personale.
Questo problema è aggravato dal carico cognitivo, *Cognitive Load*, durante le revisioni. Gonçalves e colleghi [18] evidenziano come l'efficacia delle *Code Review* diminuisca drasticamente quando il carico mentale supera una certa soglia. L'*AI* Generativa, aumentando il volume di codice prodotto, *Velocity*, inonda i revisori di una quantità di righe superiore alla loro capacità di elaborazione. In assenza di strategie di revisione esplicite, come checklist guidate, i revisori entrano in uno stato di sovraccarico che porta a revisioni superficiali, *rubber-stamping*, permettendo a *bug* critici e a violazioni di sicurezza di raggiungere l'ambiente di produzione.

Bias cognitivi nel Software Engineering

Infine, per comprendere perché i team ignorino spesso questi segnali di degrado qualitativo, è necessario fare riferimento alla psicologia cognitiva applicata. Mohanani e colleghi [19] forniscono una mappatura sistematica dei *bias* cognitivi nel *software engineering* che influenzano le decisioni tecniche.
Due *bias* emergono come particolarmente rilevanti nell'interazione con l'*AI*.

- **Automation Bias / Optimism Bias:** la tendenza a sovrastimare la correttezza dei sistemi automatizzati e a sottostimare i rischi, portando a testare meno rigorosamente il codice generato;
- **Confirmation Bias:** la tendenza a cercare solo prove che confermino il funzionamento del codice, testando l'*happy path*, ignorando i casi limite, *edge cases*, dove l'*AI* tipicamente fallisce.

La combinazione di questi *bias* crea una cecità collettiva verso il debito tecnico accumulato, che diventa visibile solo quando il sistema collassa sotto il peso della propria instabilità strutturale.

Capitolo 3: Metodologia della ricerca

La presente indagine empirica si focalizza sui processi di sviluppo software della *startup* innovativa *Nano i-Tech*, attiva nel settore *Insurtech*. Lo studio prende in esame un orizzonte temporale di otto mesi, daAprile 2025 a Novembre 2025, analizzando i flussi di lavoro relativi ai tre principali progetti dell'azienda, alcuni già citati precedentemente.

- **IMBA - IlMioBrokerAssicurativo (B2C):** applicazione *web-responsive* rivolta all'utente finale. Funge da comparatore di polizze e portafoglio digitale, permettendo ai clienti di archiviare le proprie coperture assicurative e richiedere consulenze in tempo reale.
- **Nano (Corporate Identity):** portale istituzionale che funge da *touchpoint* per investitori e partner. A differenza dei precedenti, non eroga servizi operativi ma comunica la *brand identity*, la *roadmap* tecnologica e le posizioni aperte, *recruiting*.
- **Navisio (B2B SaaS):** piattaforma gestionale *cloud-based* progettata per intermediari assicurativi e broker. Il software permette la gestione completa del portafoglio clienti, la preventivazione automatizzata e la gestione dei sinistri, integrandosi via API con le principali compagnie assicurative. Rappresenta il *core business* tecnologico dell'azienda.

Il team di sviluppo oggetto di osservazione presenta una struttura ibrida, composta da risorse interne e da consulenti esterni, operanti in regime misto, sia in presenza sia da remoto.

3.1 Strumenti e processo di generazione dei dati

La base dati utilizzata nello studio non è frutto di una rilevazione a campione, ma rappresenta la digitalizzazione completa dell'attività operativa dell'azienda. I dati derivano dall'ecosistema di strumenti di *Project Management* e *Version Control* adottati da *Nano i-Tech* per la gestione del ciclo di vita del software.

3.1.1 Fonte gestionale: ecosistema Jira e plugin Tempo

La fonte primaria per l'analisi di processo è la suite *Atlassian Jira Software*, la cui attività di tracciamento costituisce la base del dataset analizzato (*Raccolta_Dati_Sperimentali.xlsx*, foglio *Jira* [20]).

L'organizzazione del lavoro in *Nano i-Tech* segue una struttura gerarchica derivata dalle metodologie *Agile* (Schwaber & Beedle, 2001) [21], che scompone i requisiti di progetto in unità di granularità decrescente. Questa tassonomia si riflette direttamente nel dataset analizzato.

- **Epics:** rappresentano i contenitori macroscopici del lavoro. Secondo la definizione di Leffingwell (2010) [22], un'epica costituisce un'iniziativa di dimensioni significative che

richiede analisi, definizione e sviluppo estesi, la cui durata eccede tipicamente il singolo ciclo iterativo, *Sprint*. Nel contesto aziendale osservato, le epiche coincidono con le macro-fasi di rilascio mensile, fungendo da raggruppamento logico per tutte le attività svolte in un determinato mese solare.

- **User Stories e Task:** rappresentano l'unità atomica di pianificazione e consegna. Come teorizzato da Cohn (2004) [23], la *User Story* descrive una funzionalità dal punto di vista dell'utente finale, enfatizzando il valore di business piuttosto che la specifica tecnica. La struttura standard adottata segue il modello *Role-Feature-Benefit* proposto da Beck (2000) [24]: *Come [ruolo], voglio [funzionalità], affinché [beneficio]*. Nel dataset, queste entità costituiscono i singoli record, righe, oggetto di misurazione temporale.

Processo di popolamento del dato

La generazione del dato segue un flusso strutturato che coinvolge attori diversi.

- **Pianificazione - input del PM:** all'inizio di ogni ciclo, il *Project Manager* definisce le date di *Planned Start* e *Planned Finish* per ogni *ticket*. Questo definisce l'aspettativa manageriale, la *baseline*;
- **esecuzione - input del Team:** gli sviluppatori aggiornano lo stato del *ticket*, passando, ad esempio, da *To do* a *Done*, generando automaticamente i *Timestamp* di *Actual Start* e *Actual Finish*;
- **consuntivazione - input su Tempo:** poiché *Jira*, di default, non traccia la durata in ore con precisione sufficiente, la *startup* adotta il plugin *Tempo Timesheets*. Ogni risorsa è tenuta a inserire manualmente i *Worklogs*, definiti anche diari di lavoro, sui *ticket*. È proprio da questo plugin che deriva la variabile fondamentale *Time Spent*, che rappresenta lo sforzo reale.

Il file XLSX analizzato è dunque il risultato dell'esportazione di questi registri, normalizzati per associare a ogni riga, rappresentate un singolo *task*, le proprietà del suo contenitore, ovvero l'*epic*, e del progetto di appartenenza, definito *Stream*.

Il dataset estratto presenta un elevato livello di granularità. Oltre alle metriche temporali necessarie per il calcolo dei KPI, sono state mantenute variabili relative alle interazioni, i *Comments*, e alla gerarchia, *Parent* e *Subtask*, per permettere eventuali analisi qualitative sulle cause dei ritardi. Nelle seguenti tabelle, dalla Tabella 1 alla Tabella 5, viene riportato il dizionario completo delle variabili disponibili.

Tabella 1 - dizionario delle variabili del dataset Gestionale (Jira) – Identificativi.

Categoria	Nome Variabile	Descrizione	Tipo di Dato
Identificativi	Issue Type	Tipologia del ticket (Epic, Story, Task, Subtask). Definisce il livello gerarchico.	Categoriale
Identificativi	Issue Key	Codice alfanumerico univoco del ticket (es. SPNFP-58).	Stringa
Identificativi	Issue Stream	Progetto o flusso di lavoro di appartenenza (IMBA, Nano, Navisio).	Categoriale
Identificativi	Issue Summary	Titolo descrittivo del ticket.	Testo
Identificativi	Parent Key	Codice univoco del ticket genitore (link alla Epic di riferimento).	Stringa
Identificativi	Parent Summary	Titolo del ticket genitore.	Testo

Tabella 2 - dizionario delle variabili del dataset Gestionale (Jira) – Temporali.

Categoria	Nome Variabile	Descrizione	Tipo di Dato
Temporali	Creation Date	Data di inserimento del ticket a sistema. Base per il calcolo del Lead Time.	Data
Temporali	Scheduling Date	Data in cui il ticket è stato inserito nello scheduling (Backlog refinement).	Data
Temporali	Planned Start	Data di inizio lavori prevista dal Project Manager.	Data
Temporali	Planned Finish	Data di fine lavori, Deadline, prevista dal Project Manager.	Data
Temporali	Planned Duration (days)	Durata stimata in giorni lavorativi (calcolata o inserita).	Intero
Temporali	Actual Start	Data del primo log di attività o transizione in In Progress.	Data
Temporali	Actual Finish	Data di completamento effettivo (corretta manualmente per gli Zombie Tickets).	Data
Temporali	Actual Duration (days)	Giorni lavorativi intercorsi tra Actual Start e Actual Finish.	Intero
Temporali	Actual Elapsed Time (days)	Giorni solari totali (inclusi weekend) intercorsi durante l'esecuzione.	Intero

Tabella 3 - dizionario delle variabili del dataset Gestionale (Jira) – Sforzo.

Categoria	Nome Variabile	Descrizione	Tipo di Dato
Sforzo	Actual Effort (h)	Ore totali consuntivate sul ticket tramite plugin Tempo.	Decimale (Float)

Tabella 4 - dizionario delle variabili del dataset Gestionale (Jira) – Risorse.

Categoria	Nome Variabile	Descrizione	Tipo di Dato
Risorse	Task Owner	Ruolo della risorsa principale assegnataria del ticket (Assignee).	Categoriale
Risorse	Contributors	Elenco di altre risorse che hanno registrato tempo sul ticket.	Lista
Risorse	Reporter	Ruolo della persona che ha creato il ticket (spesso il PM o un Tester).	Categoriale

Tabella 5 - dizionario delle variabili del dataset Gestionale (Jira) – Qualitativi.

Categoria	Nome Variabile	Descrizione	Tipo di Dato
Qualitativi	Priority	Livello di priorità assegnato (come Highest, High, Medium, Low).	Ordinale
Qualitativi	Labels	Etichette, tag, per categorizzazioni trasversali.	Lista
Qualitativi	Task Comments	Numero di commenti registrati sul ticket principale. Indicatore di complessità comunicativa.	Intero
Qualitativi	Subtask Comments	Numero di commenti registrati sui Task figli.	Intero
Qualitativi	Total Comments	Somma totale delle interazioni discorsive sul ticket.	Intero

Per evidenziare la struttura del dato grezzo e la distinzione tra pianificazione ed esecuzione, si riporta in Tabella 6 un estratto del file analizzato.

Tabella 6 - campione del dataset gestionale.

Issue Type	Issue key	Issue Stream	Issue Summary	Planned Start	Planned Finish	Planned Duration	Actual Start	Actual Finish	Actual Duration
Epic	SPNFP-40	Sprint	Sprint Aprile 2025	01/04/2025	30/04/2025	22	01/04/2025	07/05/2025	27
Story	SPNFP-58	NAVISIO	NAVISIO - MVP	01/04/2025	09/05/2025	29	01/04/2025	07/05/2025	27
Story	SPNFP-51	IMBA	IMBA - Fatturazione integrata - Analisi feature ARUBA	01/04/2025	08/04/2025	6	01/04/2025	06/05/2025	26
Story	SPNFP-41	IMBA	IMBA - Consensi GDPR a norma	01/04/2025	02/04/2025	2	01/04/2025	15/04/2025	11
Story	SPNFP-53	IMBA	IMBA - Scontistica	02/04/2025	10/04/2025	7	02/04/2025	15/04/2025	10
Story	SPNFP-52	IMBA	IMBA - Firma digitale/generazione PDF	02/04/2025	11/04/2025	8	02/04/2025	11/04/2025	8
Story	SPNFP-7	IMBA	IMBA - Testing payment service	09/04/2025	15/04/2025	5	09/04/2025	15/04/2025	5
Story	SPNFP-57	IMBA	IMBA - App mobile - Analisi versione Wezard + analisi porting in Expo	14/04/2025	16/05/2025	25	14/04/2025	07/05/2025	18
Story	SPNFP-54	IMBA	IMBA - Emissione certificato	14/04/2025	17/04/2025	4	14/04/2025	17/04/2025	4
Story	SPNFP-55	IMBA	IMBA - Generazione report convenzioni	17/04/2025	23/04/2025	5	17/04/2025	23/04/2025	5
Story	SPNFP-56	IMBA	IMBA - Invio report convenzioni mensile	24/04/2025	29/04/2025	4	24/04/2025	29/04/2025	4

Si noti la compresenza di date pianificate, indicate con *Planned*, e date effettive, indicate con *Actual*, per la medesima *Issue*.

3.1.2 Fonte tecnica: repository GitHub

Per analizzare l'output produttivo reale, lo studio si avvale dei dati estratti da *GitHub*, la piattaforma di *hosting* per il controllo della versione del codice. A differenza dei dati gestionali, soggetti all'errore umano di inserimento manuale, questo dataset (*Raccolta_Dati_Sperimentali.xlsx, foglio GitHub* [20]) rappresenta una traccia oggettiva e immutabile delle attività.

Workflow di generazione

Ogni volta che uno sviluppatore completa una porzione di codice, effettua un'operazione di *Commit* e *Push* sul server centrale. Il sistema registra automaticamente:

- **autore**, identificando univocamente chi ha svolto il lavoro, successivamente anonimizzato in fase di analisi;
- **volumetria**, calcolando il differenziale di righe aggiunte, *Additions,* e rimosse, *Deletions*.

I dati grezzi sono stati inizialmente estratti dal repository mantenendo la granularità originale, a livello di singolo *commit*. Successivamente, in fase di costruzione del dataset, si è proceduto a una aggregazione per raggruppare i volumi di codice per mese e per risorsa. Questa operazione di sintesi ha permesso di trasformare il log tecnico disaggregato in una serie storica confrontabile con le metriche gestionali di *Jira*.

A differenza del dataset gestionale, strutturato come elenco di record singoli, il dataset tecnico è strutturato come matrice multidimensionale. La Tabella 7 descrive le dimensioni logiche utilizzate per organizzare i volumi di produzione del codice.

Tabella 7 - struttura logica del dataset Tecnico (GitHub).

Ruolo nel Dataset	**Variabile Logica**	**Rappresentazione nel XLSX**	**Descrizione**
Dimensione temporale	Mese	Righe	Mese di riferimento dell'aggregazione.
Dimensione risorsa	Contributor ID	Intestazioni di colonna	Ruolo della risorsa che ha generato il codice (es. Backend Developer 1, CTO). Ogni colonna rappresenta un contributor.
Metrica (KPI)	Activity Type	Raggruppamento colonne	Tipologia di attività misurata. Si divide in tre categorie: Commits, Additions e Deletions.
Valore	Volume	Celle interne	Il valore numerico assoluto, rappresentato da un intero, corrispondente all'incrocio tra Mese, Risorsa e Attività.

Poiché il dataset tecnico è presentato in forma matriciale, aggregata per facilitare il confronto mensile, le variabili sono descritte in termini di dimensioni, righe e colonne, e metriche.
In Tabella 8 è mostrato un estratto del file focalizzato sulla frequenza di invio codice, *Commits*. La medesima struttura a matrice si ripete nel dataset per le metriche volumetriche, *Additions* e *Deletions*.

Tabella 8: campione del dataset tecnico - vista Commits.

	Commits				
Mese	**Backend Developer 1**	**Backend Developer 2**	**Consulente IT 1**	**CTO**	**Frontend Developer**
Aprile	27	25	0	119	3
Maggio	17	16	0	72	0
Giugno	1	15	0	57	0
Luglio	35	83	4	180	8
Agosto	3	32	2	196	24
Settembre	35	23	6	313	19
Ottobre	28	14	17	154	71
Novembre	29	2	20	265	11

I dati sono organizzati in una matrice che incrocia il mese di competenza, righe, con il ruolo tecnico, colonne.

3.2 Preprocessing e normalizzazione dei dati

Prima di procedere al calcolo delle metriche, il dataset grezzo è stato sottoposto a una rigorosa fase di pulizia e arricchimento, *preprocessing* dei dati, per correggere distorsioni e garantire la privacy.

Anonimizzazione

In conformità con le normative sulla privacy e gli accordi di riservatezza aziendali, si è proceduto all'anonimizzazione dei dati sensibili:

- **risorse**, i nomi dei dipendenti e dei consulenti sono stati sostituiti con etichette di ruolo funzionale (*Backend Developer 1, Backend Developer 2, CEO, Consulente IT, CTO, Frontend Developer, CIO/PM, UI/UX Designer*);
- **task**, i campi *Summary* contenenti riferimenti espliciti a clienti terzi o dati sensibili sono stati generalizzati o omessi dall'analisi testuale.

Calcolo della Planned Duration - inferenza

Poiché il *Project Manager* impostava su *Jira* esclusivamente le date di inizio, *Planned Start*, e di fine, *Planned Finish*, senza inserire esplicitamente la stima in giorni, la variabile *Planned Duration* è stata calcolata in fase di *preprocessing* secondo la formula:

$$Planned\ Duration\ =\ NetworkDays(Planned\ Start, Planned\ Finish) \quad [2]$$

Questo passaggio è stato necessario per ottenere un denominatore valido per il calcolo delle varianze di pianificazione.

Esempio di inferenza della durata.

Per meglio comprendere la procedura, si consideri il ticket reale SPNFP-58 (Navisio MVP) presente nel dataset.

- **Planned Start:** 01/04/2025 – Martedì;
- **Planned Finish:** 09/05/2025 - Venerdì;
- **Durata solare - lorda:** 39 giorni.

Applicando il calcolo dei giorni lavorativi, escludendo sabati, domeniche e festività, il valore inferito risulta essere di 29 giorni. Tale valore coincide con la capacità produttiva effettiva attesa e viene utilizzato come denominatore per il calcolo della *Planning Variance*.

Correzione della Actual Finish Date

Durante l'analisi preliminare è emerso un disallineamento sistematico tra la conclusione reale dei lavori e la chiusura amministrativa dei *ticket* su *Jira*. Spesso i *task* venivano completati tecnicamente, ma lasciati in stato *In Progress* o *Review* per giorni prima di essere spostati in *Done*, fenomeno degli *Zombie Tickets*.

Per evitare di sovrastimare i ritardi, la *Actual Finish Date* è stata corretta manualmente incrociando l'ultimo *Worklog* temporale registrato su *Tempo* o l'ultima transizione di stato significativa, scartando così i tempi morti puramente burocratici.

3.3 Definizione delle metriche analitiche

A valle del processo di normalizzazione, sono stati calcolati gli indicatori di performance (KPI) necessari per rispondere agli obiettivi di ricerca. L'elaborazione operativa di tali indicatori, inclusa l'applicazione puntuale delle formule ai dati grezzi, è riportata integralmente nelle sezioni dedicate ai modelli di calcolo del dataset (*foglio Calcoli Jira* e *foglio Calcoli Github* [20]).

Le metriche sono suddivise in tre macroaree: efficienza di flusso, affidabilità della pianificazione e qualità tecnica.

Metriche di flusso - Time Metrics

Questi indicatori misurano la velocità di attraversamento del *Workflow* di sviluppo, dal momento della richiesta al rilascio.

- **Lead Time**

  Il *Lead Time* rappresenta il tempo totale intercorso tra la creazione del *ticket* a sistema, *Creation Date*, e la sua chiusura definitiva, *Actual Finish*. Questa metrica riflette il tempo di attesa complessivo percepito dagli *stakeholder* e, se confrontata con i tempi di esecuzione tecnica, permette di identificare eventuali colli di bottiglia nelle fasi preliminari di approvazione e analisi dei requisiti.

$$Lead\ Time\ =\ Actual\ Time\ -\ Creation\ Date \quad [3]$$

- **Waiting Time**

  Il *Waiting Time* quantifica il periodo che un *ticket* trascorre in stato di giacenza nel *Backlog* prima di essere effettivamente preso in carico dal team di sviluppo. Un valore elevato in questa metrica è indice di inefficienza nella pianificazione, *Backlog Management*, e suggerisce uno squilibrio tra la capacità produttiva disponibile e la mole di richieste in ingresso.

$$Waiting\ Time\ =\ Actual\ Start\ Date\ -\ Creation\ Date \quad [4]$$

- **Cycle Time**

  Il *Cycle Time* misura il tempo effettivo dedicato alla lavorazione tecnica del *ticket*, calcolato a partire dal primo *log* di attività, *Actual Start*, fino al rilascio, *Actual Finish*. Rappresentando la velocità pura del team di sviluppo, un suo incremento ingiustificato è spesso sintomatico di complessità tecniche impreviste o di continue interruzioni operative.

$$Cycle\ Time\ =\ Actual\ Finish\ Date\ -\ Actual\ Start\ Date \quad [5]$$

Metriche di pianificazione ed efficienza - Process Metrics

Questi indicatori valutano la capacità del *Management* di stimare correttamente i lavori e l'efficienza nell'eseguirli.

- **Delivery Delay**

  Il *Delivery Delay* calcola lo scostamento in giorni solari tra la data di consegna effettiva e la scadenza preventivata, *Deadline*. Valori positivi evidenziano un ritardo visibile al cliente, mentre valori negativi indicano una consegna anticipata rispetto alle attese manageriali.

$$Delivery\ Delay\ =\ Actual\ Finish\ Date\ -\ Planned\ Finish\ Date \qquad [6]$$

- **Duration Overrun**

  La *Duration Overrun* misura la dilatazione della durata dei lavori rispetto alla finestra temporale originariamente preventivata, indipendentemente dalla data di scadenza finale. Questa metrica è fondamentale per isolare l'errore di stima dello sforzo, *Effort*, da eventuali slittamenti dovuti a ritardi nell'avvio delle attività.

$$Duration\ Overrun\ =\ Actual\ Duration\ -\ Planned\ Duration \qquad [7]$$

- **Planning Variance**

  La *Planning Variance* esprime l'errore percentuale della stima temporale, indicando quanto l'esecuzione reale si sia discostata dalla previsione. Valori positivi segnalano un *Optimism Bias*, sottostima della complessità, mentre valori negativi indicano una sovrastima cautelativa delle tempistiche.

$$Planning\ Variance\ (\%)\ =\ \frac{Actual\ Duration\ -\ Planned\ Duration}{Planned\ Duration}\ x\ 100 \qquad [8]$$

- **Execution Efficiency**

  L'*Execution Efficiency* valuta la densità del lavoro svolto, mettendo in relazione le ore effettivamente lavorate, *Time Spent*, con la durata calendariale del *task*. Valori bassi (< 20%) denotano una bassa efficienza di flusso dovuta a tempi morti o multitasking, mentre valori elevati (> 80%) indicano un'attività svolta in regime di *Swarming*, ovvero con focus esclusivo e senza interruzioni.

$$Execution\ Efficiency\ (\%)\ =\ \frac{Total\ Time\ Spent\ (hours)}{Actual\ Duration\ (days)\ x\ Standard\ Work\ Day\ (8h)}\ x\ 100 \qquad [9]$$

Metriche di qualità e sviluppo - Code Metrics

Indicatori derivati dal repository *GitHub* per qualificare la natura dello sforzo tecnico e i rischi strutturali.

- ***Refactoring Ratio***

  Il *Refactoring Ratio* esprime la percentuale di codice rimosso rispetto a quello aggiunto, fungendo da indicatore per distinguere lo sviluppo di nuove funzionalità, *Greenfield,* dalle attività di manutenzione. Picchi improvvisi o valori superiori al 50% indicano fasi di pesante riscrittura, *Rework*, spesso causate da requisiti poco chiari o dalla necessità di sanare debito tecnico pregresso.

$$Refactoring\ Ratio\ = \frac{Deletions}{Additions} \qquad [10]$$

- **Churn Density**

Il *Churn Density* normalizza il volume totale di codice movimentato, aggiunto e rimosso, rispetto al numero di *ticket* chiusi, rappresentando il costo tecnico medio per il completamento di una funzionalità. Un aumento di questa metrica suggerisce un'instabilità dei requisiti, *Requirements Volatility*, evidenziando che per chiudere lo stesso numero di *task* è stato necessario uno sforzo di codifica sproporzionato.

$$Churn\ Density\ = \frac{\Sigma(Additions\ +\ Deletions)}{Total\ Closed\ Issues} \qquad [11]$$

- **Hero Factor**

  L'*Hero Factor* quantifica il rischio organizzativo derivante dalla dipendenza da una risorsa chiave, analizzando la distribuzione del contributo tecnico. Valori superiori al 50% riferiti a una singola figura, nel caso in esame il CTO, evidenziano una *Hero Culture*, dove il successo del progetto è vincolato alla capacità produttiva di un unico individuo, creando un collo di bottiglia strutturale, *Bus Factor*.

$$Hero\ Factor\ = \frac{Total\ Volume\ CTO\ (Additions\ +\ Deletions)}{Total\ Volume\ Team\ (Additions\ +\ Deletions)}\ x\ 100 \qquad [12]$$

Capitolo 4: Analisi dei risultati e discussione

In questo capitolo vengono esposti i risultati dell'indagine empirica applicando il *framework* di metriche definito nel Capitolo 3. L'esposizione segue una logica sequenziale: si analizzeranno dapprima i flussi di processo, derivanti da *Jira*, per quantificare le inefficienze temporali, per poi approfondire le dinamiche tecniche, derivanti da *GitHub*, al fine di indagarne le cause strutturali.

4.1 Analisi dei flussi di lavoro - Jira Software

La prima fase dell'indagine è volta a verificare l'effettiva capacità di *delivery* del team in relazione ai volumi di lavoro gestiti. I dati grezzi sono stati analizzati per quantificare il carico operativo e identificare le aree di maggiore assorbimento delle risorse.

Distribuzione del carico di lavoro

Sebbene il perimetro di analisi comprenda tre iniziative distinte, la quantificazione dei *ticket* gestiti, come mostrato in Figura 4.1, rivela un forte sbilanciamento operativo.

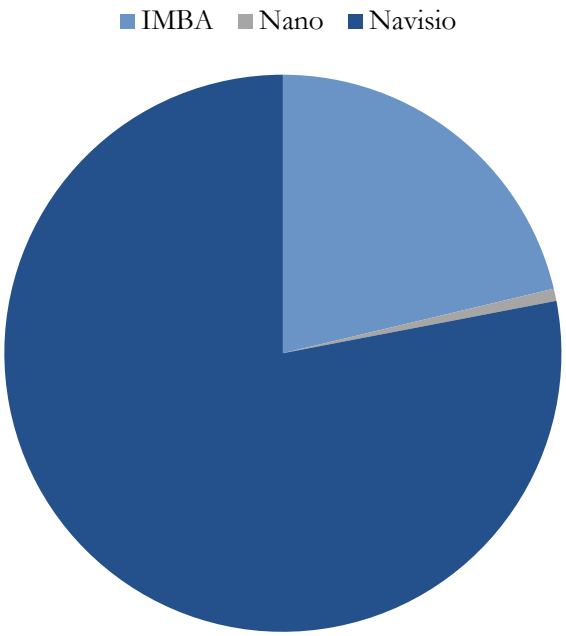

Figura 9: ripartizione percentuale del carico di lavoro effettivo sui tre progetti.

I dati confermano che Navisio assorbe il 78% della capacità produttiva totale del team, consolidandosi come il driver principale delle attività di sviluppo. Al contrario, il progetto Nano,

i cui *ticket* afferiscono esclusivamente agli interventi di manutenzione e correzione del sito web aziendale, ha generato un volume di attività trascurabile, appena dello 0.7%. Questa evidenza permette di focalizzare la successiva analisi delle inefficienze prevalentemente sul flusso di lavoro di Navisio, essendo questo il vero *core business* dell'azienda.

Analisi dei tempi: Lead Time vs Cycle Time

Per valutare la fluidità del processo di sviluppo, sono state confrontate le metriche di attraversamento, *Lead Time*, con i tempi di lavorazione attiva, *Cycle Time*. La Figura 4.2 mostra l'andamento mensile di queste due grandezze.

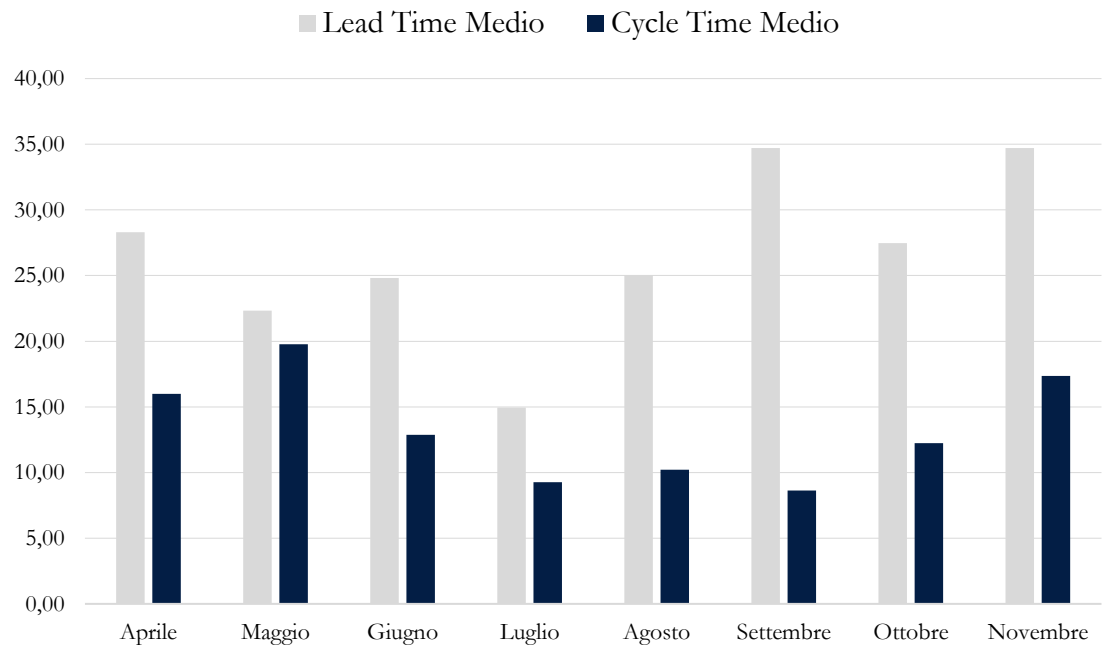

Figura 10: confronto tra Lead Time medio, tempo totale, e Cycle Time, tempo di lavorazione attiva per mese.

A supporto dell'analisi visiva, è stato calcolato l'indice di *Execution Efficiency*.

Tabella 9: dettaglio mensile dell'Execution Efficiency e della Duration Overrun.

Mese	Execution Efficiency	Duration Overrun
Aprile	82.61%	2.30
Maggio	46.05%	3.11
Giugno	53.38%	1.50
Luglio	52.51%	1.73
Agosto	50.41%	1.86
Settembre	53.47%	0.16
Ottobre	53.51%	5.00
Novembre	49.31%	5.76

L'incrocio tra i dati quantitativi e il contesto organizzativo permette di distinguere tre fasi evolutive.

- **Fase di Onboarding e assestamento - Aprile:** come dettagliato in Tabella 4.1, il primo mese registra un valore di efficienza anomalo, pari all'82.61%. Tale dato fuori scala non riflette una reale capacità produttiva a regime, ma è imputabile alla discontinuità gestionale avvenuta proprio ad Aprile, con l'inserimento della nuova figura di *Project Management*. La fase conoscitiva del nuovo *PM* e il passaggio da una gestione interna destrutturata a una pianificazione formale hanno generato un effetto di pulizia iniziale e riorganizzazione, portando a una saturazione temporanea delle risorse, *Crunch Mode,* per allineare lo stato dei lavori ai nuovi standard;
- **consolidamento dei processi e sostenibilità – da Maggio ad Agosto:** superata la fase di riassetto iniziale, il quadrimestre centrale segna il raggiungimento di un equilibrio operativo strutturale. L'*Execution Efficiency* si stabilizza in una forbice compresa tra il 50% e il 53%: tale valore, lontano dall'essere un indice di scarsa produttività, rappresenta invece il *benchmark* di sostenibilità nel lungo periodo. Esso indica che la giornata lavorativa del *ticket* è ripartita equamente tra sviluppo attivo, circa 4 ore effettive, e attività fisiologiche di supporto, come allineamenti col *PM*, *Code Review* o gestione di imprevisti. In questa fase, inoltre, si registra un trend virtuoso: il *Cycle Time* scende progressivamente, da 19 giorni a circa 10 giorni, e l'errore di stima, *Overrun*, si riduce, dimostrando che le nuove metodologie introdotte dal *Project Manager* sono state assimilate, portando il team alla sua reale velocità di crociera;

- **paradosso di Settembre - Backlog Aging:** nel sesto mese, a fronte di una buona performance tecnica, *Cycle Time* minimo di 8.64 giorni ed efficienza stabile al 53%, il *Lead Time* esplode a 34.72 giorni. Questo divario, di oltre 26 giorni di attesa, conferma che, una volta stabilizzato il processo esecutivo, il collo di bottiglia si è spostato sulla pianificazione. I *ticket* vengono inseriti nel *Backlog* con largo anticipo dal *PM*, ma rimangono in attesa per settimane prima di entrare nello *Sprint*. I ritardi percepiti, dunque, non dipendono dalla velocità degli sviluppatori, ma dalle tempistiche di messa in coda tipiche di una gestione strutturata a *Backlog*.

Puntualità e affidabilità della pianificazione

L'impatto delle dinamiche temporali sulla puntualità di consegna è analizzato nella Figura 4.3, che mappa la distribuzione dei ritardi, *Delivery Delay*, rispetto alla data di pianificazione.

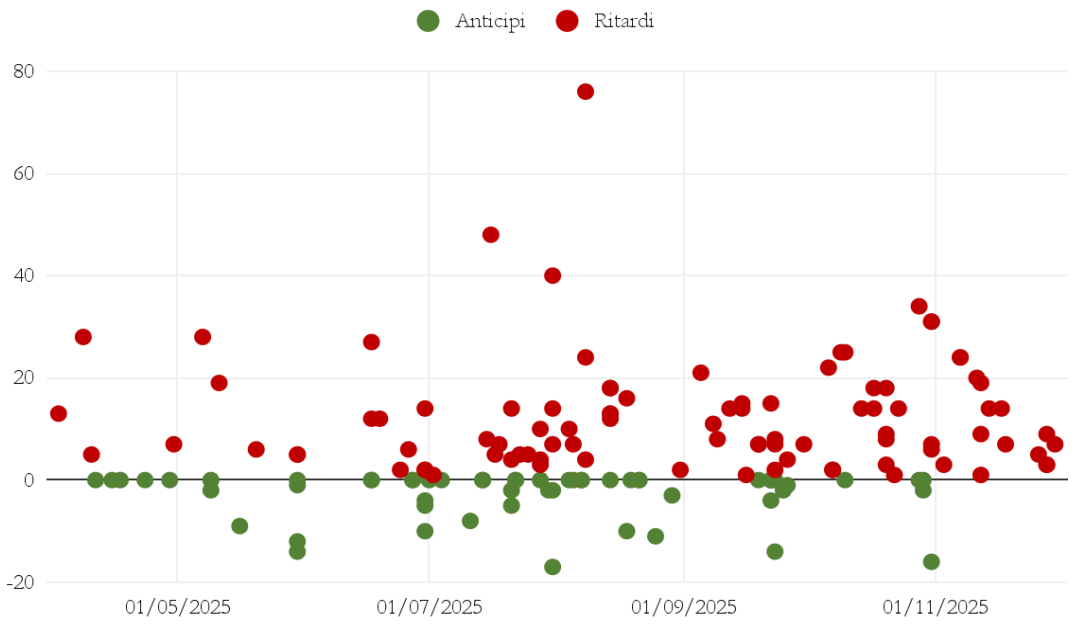

Figura 11: distribuzione temporale del Delivery Delay.

I punti situati al di sopra della linea dello zero, di colore rosso, indicano le consegne avvenute oltre la data di scadenza. Per comprendere le cause di tali slittamenti, è fondamentale incrociare il dato visivo con la *Duration Overrun*, scostamento tra la durata stimata e quella reale, il cui andamento mensile rivela tre fasi distinte.

- **Learning Curve gestionale – da Aprile a Settembre:** nel primo semestre si osserva un netto miglioramento della capacità predittiva. L'errore di stima medio, *Overrun*,

scende dai 2 o 3 giorni iniziali fino al minimo storico di 0.16 giorni a Settembre. Questo dato è di importanza critica per l'intera analisi: nel mese di Settembre, lo scostamento tra stima e realtà è stato praticamente nullo. Poiché nello stesso mese il *Lead Time* era molto alto, si può affermare con certezza che i ritardi di consegna di quel periodo non sono imputabili all'esecuzione tecnica, ma esclusivamente ai tempi di attesa nel *Backlog*;

- **degrado prestazionale di fine periodo – da Ottobre a Novembre:** nell'ultimo bimestre si registra un'inversione di tendenza, con l'*Overrun* che sale bruscamente a 5.00 giorni e successivamente a 5.76 giorni. A differenza dei mesi estivi, in autunno il ritardo assume una natura esecutiva: i *task* hanno richiesto significativamente più tempo del previsto per essere completati. Tale peggioramento improvviso suggerisce che il carico di lavoro accumulato, nel *Backlog*, e la complessità tecnica abbiano iniziato a saturare le risorse, rendendo meno affidabili le previsioni iniziali;
- **impatto degli Outlier:** infine, si nota come la media complessiva del *Delivery Delay* sia fortemente influenzata da alcuni casi isolati, *outlier*. Depurando il dataset dai valori estremi, il ritardo medio *Filtered* risulta significativamente inferiore, confermando che il processo mantiene una stabilità di fondo, interrotta solo da specifiche eccezioni o picchi di complessità non prevedibili.

L'analisi dimostra che il team ha raggiunto un'ottima maturità nella stima dei tempi, con il picco di precisione a Settembre, ma l'eccessivo accumulo di lavoro nella parte finale dell'anno ha reintrodotto incertezza nel processo.

Analisi integrata degli scostamenti temporali

A conclusione dell'indagine sui flussi *Jira*, la Tabella 4.2 riepiloga i valori mensili delle metriche chiave, permettendo di confrontare direttamente l'efficienza di attraversamento, *Waiting Time*, con l'accuratezza della pianificazione, *Duration Overrun* e *Planning Variance*.

Tabella 10: quadro riepilogativo delle inefficienze mensili.

Mese	Waiting Time	Duration Overrun	Planning Variance
Aprile	12.30	2.30	79.13%
Maggio	3.38	3.11	42.82%
Giugno	11.94	1.50	13.57%
Luglio	7.24	1.73	69.02%
Agosto	15.67	1.86	50.04%
Settembre	26.08	0.16	16.17%
Ottobre	16.05	5.00	163.49%
Novembre	18.50	5.76	140.54%

L'analisi incrociata dei dati evidenzia tre scenari distinti.

- **Efficienza tecnica vs collo di bottiglia - Settembre:** il sesto mese presenta una *Planning Variance* minima pari al 16.17%, e un *Overrun* quasi nullo, +0.16 gg, certificando una stima dei tempi pressoché perfetta. Di conseguenza, l'elevato *Waiting Time*, di 26.08 gg, registrato nello stesso periodo non è imputabile al team di sviluppo, ma identifica inequivocabilmente un blocco a livello di gestione del *Backlog*, *Aging*;
- **collasso delle stime – da Ottobre a Novembre:** l'ultimo bimestre segna una rottura strutturale. Una varianza superiore al 140-160% indica che i *task* hanno richiesto più del doppio del tempo previsto. In questa fase, il ritardo non è più dovuto all'attesa, ma a una sottostima sistematica della complessità, sintomo di una pianificazione frettolosa o di requisiti poco chiari in chiusura dell'anno;
- **instabilità sui task brevi - Luglio:** anche in presenza di un *Overrun* assoluto contenuto, +1.73 gg, la varianza percentuale può risultare elevata, 69%. Questo suggerisce che, sui *task* di breve durata, il margine di errore relativo rimane alto, evidenziando una difficoltà nel prevedere con precisione le microattività rispetto a quelle più strutturate.

I dati confermano il fatto che le criticità del processo non sono omogenee, ma oscillano tra problemi puramente organizzativi, come a Settembre, e carenze di stima tecnica, come a Novembre.

4.2 Analisi tecnica e qualità del codice - GitHub

Mentre l'analisi dei dati relativi a *Jira* ha fornito una panoramica sui tempi di attraversamento, l'analisi dei dati estratti dal repository *GitHub* permette di entrare nel merito della materia prima: il codice sorgente. In questa sezione si indagano le dinamiche di collaborazione e la stabilità del software per comprendere le cause tecniche delle anomalie autunnali.

Accentramento della conoscenza: analisi dell'Hero Factor

La prima dimensione indagata è la distribuzione del carico tecnico, misurata attraverso l'*Hero Factor*, ovvero la percentuale del volume totale gestita dal *Lead Developer*, nel caso in analisi questa figura corrisponde al *CTO*. L'andamento è visualizzato nella Figura 4.4.

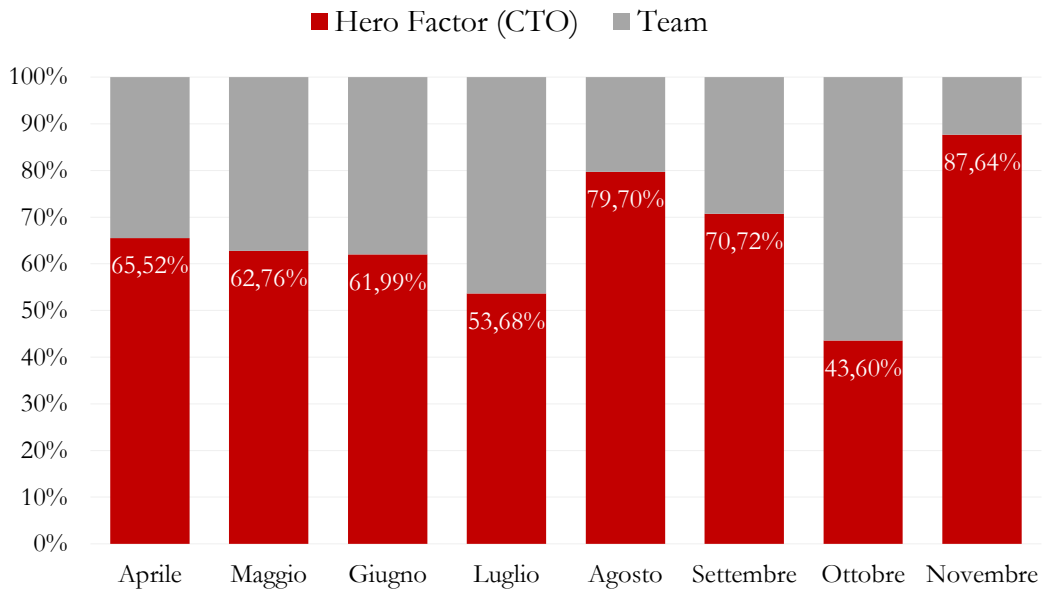

Figura 12: distribuzione mensile del volume di codice movimentato.

L'analisi rivela un pattern comportamentale chiaro.

- **Fase di controllo – da Aprile a Settembre:** il *CTO* gestisce stabilmente tra il 60% e l'80% del volume. È un assetto centralizzato ma comunque stabile;
- **anomalia di Ottobre (43.60%):** la quota del *CTO* crolla. Il team prende il sopravvento volumetrico, generando enormi quantità di codice con scarsa supervisione del *Lead*;

- **presa in carico di Novembre (87.64%):** di fronte alla crisi, il *CTO* torna a gestire la quasi totalità del codice, segno che il sistema richiedeva un intervento esperto non delegabile.

Dinamiche volumetriche e Refactoring Ratio

Per comprendere la qualità del lavoro svolto, la Figura 4.5 propone una visualizzazione combinata.

- **Additions - barre verdi:** indicano le nuove righe introdotte;
- **Deletions - barre rosse:** indicano le righe eliminate;
- **Refactoring Ratio - linea grigia:** misura quanto codice viene rimosso per ogni riga di codice aggiunta. Un valore basso indica accumulo puro, un valore alto indica sostituzione o pulizia.

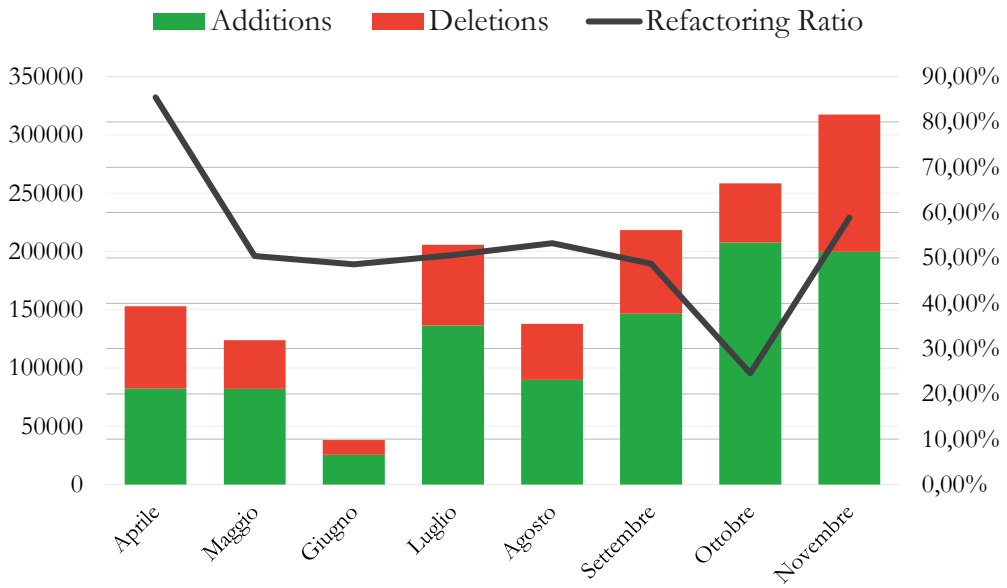

Figura 13: volumi di codice, Additions e Deletions, e andamento del Refactoring Ratio.

L'interpretazione della linea grigia svela la dinamica del fallimento.

- **Equilibrio - Giugno (48.57%):** con un *Refactoring Ratio* vicino al 50%, per ogni due righe scritte ne veniva cancellata una. È un indice di sviluppo sano: il codice evolve sostituendo il vecchio, senza ingigantire eccessivamente la *codebase*;
- **accumulo incontrollato - Ottobre (24.54%):** a Ottobre il *Refactoring Ratio* crolla. A fronte di 207602 righe aggiunte, le righe cancellate sono minime. Il team ha usato l'*AI*

per scrivere codice nuovo senza preoccuparsi di rimuovere il debito tecnico o ottimizzare l'esistente. È la fase di *Code Bloat*, ovvero di gonfiamento del codice;

- **sostituzione forzata - Novembre (58.90%):** il sistema saturo ha richiesto un intervento drastico. Il *ratio* risale quasi al 60%: il *CTO* ha dovuto cancellare 117669 righe per far posto alle correzioni. Non è stato sviluppo incrementale, ma una ristrutturazione pesante.

Sintesi tecnica: la densità del lavoro - Churn Density

A conclusione dell'indagine sui dati prelevati da *GitHub*, l'analisi si focalizza sul *Churn Density*. Come definito nel quadro metodologico, questa metrica normalizza i volumi di codice rispetto al numero di *ticket* gestiti, fornendo un indicatore preciso della pesantezza media e della complessità di ogni singola *issue* affrontata dal team.
La Tabella 4.3 riepiloga l'evoluzione mensile di questo indicatore, permettendo di valutare se la granularità del lavoro sia rimasta costante o se abbia subito alterazioni nel corso del tempo.

Tabella 11: volume medio di codice movimentato per singola issue.

Mese	Churn Density
Aprile	15286.00
Maggio	13748.11
Giugno	2383.88
Luglio	9344.45
Agosto	6264.64
Settembre	8732.40
Ottobre	12312.05
Novembre	18673.12

L'analisi della serie storica evidenzia un andamento a U, caratterizzato da tre fasi distinte.

- **Fase di bonifica iniziale – da Aprile a Maggio:** il periodo di osservazione si apre con valori elevati, superiori alle 13000 righe per singola *issue*. Questo dato è fisiologico in una fase di *setup* o di ristrutturazione iniziale, coincidente con l'introduzione dei nuovi processi di *Project Management*. Il team ha lavorato su *task* consistenti, probabilmente

legati alla pulizia del *codice legacy* o alla riformulazione di moduli core per allinearsi ai nuovi standard;

- **fase di stabilità - Giugno:** a Giugno si registra il minimo storico: il *Churn Density* crolla a 2383 righe. Questo rappresenta il punto di equilibrio ideale, ovvero *benchmark*. Terminata la bonifica iniziale, il team ha iniziato a lavorare su *task* più specifici, granulari e ben definiti. In questo mese, il flusso di lavoro ha raggiunto la massima efficienza tecnica;
- **fase di degrado e inflazione – da Luglio a Novembre:** con l'adozione degli strumenti di *AI Coding* a Luglio, il trend si inverte drasticamente. La densità risale progressivamente, passando dalle 9000 righe estive fino all'esplosione di Novembre. Il valore finale di 18673 righe per *issue* indica che il sistema è tornato a una complessità ingestibile, ma per motivi opposti a quelli iniziali: non più una bonifica pianificata, bensì un accumulo di codice inflazionato e debito tecnico non gestito, che ha trasformato ogni *ticket* in un intervento massivo di riscrittura.

La parabola del *Churn Density* racconta la storia di un processo che, dopo essere stato risanato a Giugno, ha perso nuovamente la sua granularità a causa dell'introduzione non governata dell'*AI*, arrivando a fine anno a livelli di pesantezza operativa superiori persino a quelli della fase di avvio.

4.3 Analisi della capacità produttiva, Velocity, e resilienza del Team

Prima di procedere alla definizione dei modelli predittivi, è necessario analizzare l'andamento storico della velocità operativa, *Velocity*, sull'intero periodo di osservazione, da Aprile a Novembre. L'analisi dei dati a consuntivo, *Story Points*, permette di valutare la resilienza dell'organizzazione di fronte agli imprevisti.

Il contesto gerarchico è determinante: il team presenta una struttura fortemente sbilanciata, con un unico profilo *Senior*, il *CTO*, e un gruppo operativo di sviluppatori di livello medio, *Mid-level*. L'uscita del *Backend Developer 2* a inizio Novembre ha agito da stress-test per questa configurazione.

Team Velocity: l'illusione dell'accelerazione

La Tabella 4.4 riporta la serie storica della produttività complessiva. I dati permettono di isolare le tre fasi evolutive del progetto.

Tabella 12: Team Velocity Chart mensile.

Sprint	Story Points (Totale Team)
Aprile	99.46
Maggio	110.70
Giugno	102.36
Luglio	124.35
Agosto	110.45
Settembre	121.13
Ottobre	116.34
Novembre	86.71

L'andamento visivo è rappresentato nella Figura 4.6, che evidenzia graficamente la variazione dei volumi di lavoro completati.

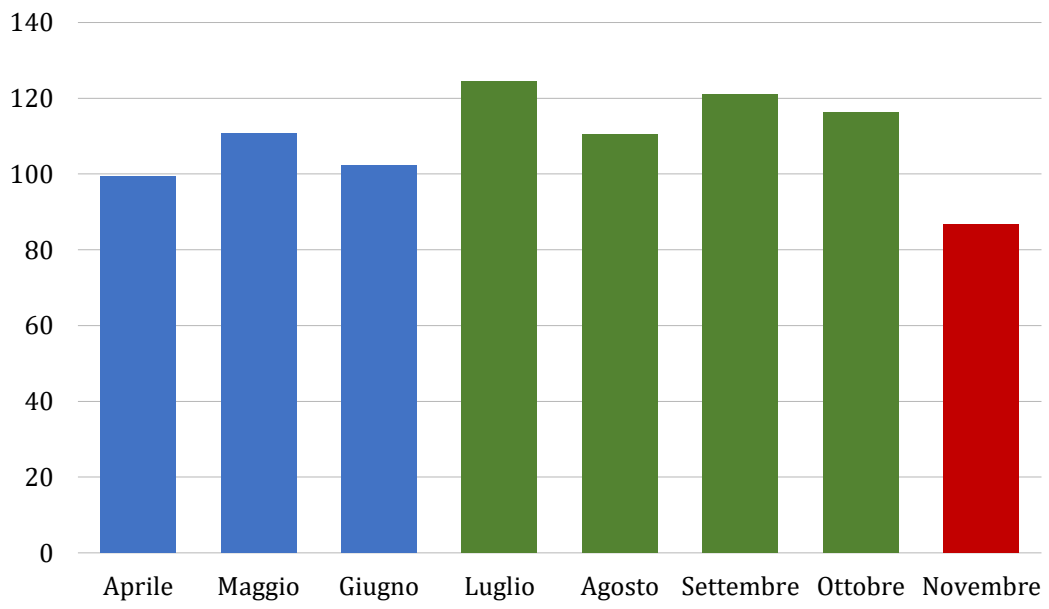

Figura 14: Team Velocity Chart mensile.

L'analisi del trend mostra come l'uso dell'*AI* abbia mascherato i limiti strutturali del team.

- **Sostenibilità iniziale – da Aprile a Giugno:** inizialmente, il team, presentava una media sostenibile, circa 100 punti;

- **boost artificiale – da Luglio a Ottobre:** con l'adozione di *Copilot*, la *Velocity* è salita del 20%, picco pari a 124.35. L'*AI* ha permesso ad un team di soli *Mid-level* di produrre volumi di codice tipici di un team più esperto, creando un'illusione di *upskilling* immediato;
- **crollo - Novembre:** il calo a 86.71 punti, -25.5%, riporta la produttività sotto i livelli di partenza. Questo dimostra che la maggiore velocità dei mesi precedenti non era consolidata, ma dipendeva totalmente dalla presenza dell'organico al completo per gestire il *rework*.

Individual Velocity: analisi del Breakdown

La Tabella 4.5 disaggrega i dati per evidenziare il comportamento delle singole risorse.
Per chiarezza di analisi sul codice, il focus è sui programmatori, ma sono riportati tutti i dati per completezza.

Tabella 13: ripartizione della Velocity per singola risorsa.

	Story Points Consuntivo				
Sprint	**Backend Developer 1**	**Backend Developer 2**	**CTO**	**Frontend Developer**	**UI/UX Designer**
Aprile	22.80	23.04	20.42	13.36	19.84
Maggio	24.24	24.24	26.87	15.49	19.86
Giugno	21.84	21.84	24.39	15.91	18.38
Luglio	26.40	26.40	28.67	19.88	23.00
Agosto	23.04	23.04	25.62	18.75	20.00
Settembre	25.44	25.44	28.25	20.00	22.00
Ottobre	26.40	26.40	28.64	20.89	14.00
Novembre	23.04	11.52	25.34	17.31	9.50

La Figura 4.7 mostra le traiettorie individuali e la reazione del sistema alla perdita di una risorsa.

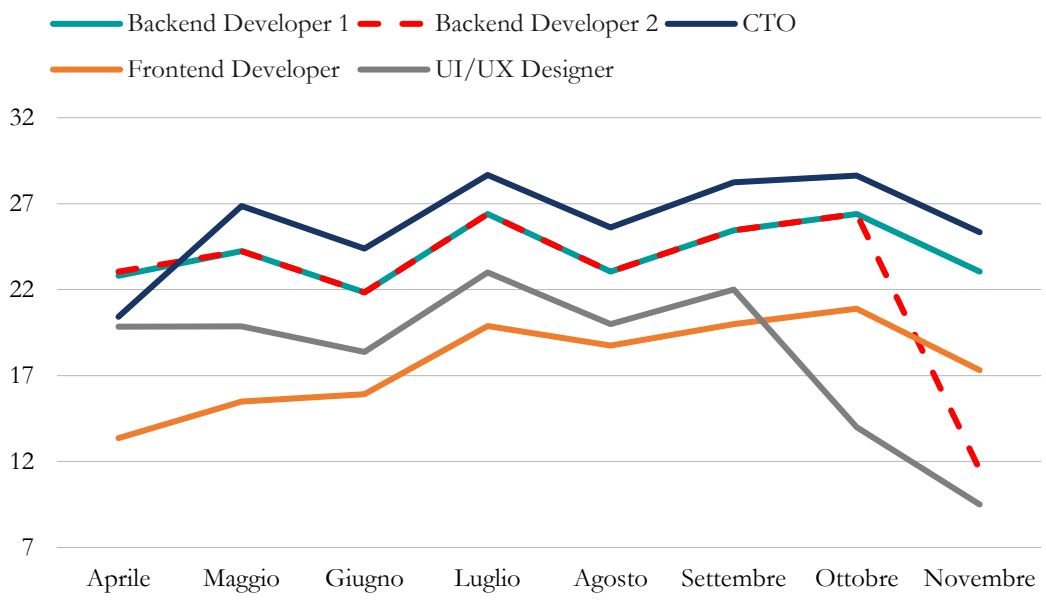

Figura 15: Velocity Chart disaggregato per risorsa.

L'analisi dei dati disaggregati svela la dinamica interna del crollo.

- **Evento scatenante - Novembre:** il *Backend Developer 2* vede crollare il proprio consuntivo nel mese di Novembre, passando da una media stabile di 26.00 punti a 11.52. Questo dato certifica l'interruzione dell'attività lavorativa avvenuta nella prima metà del mese;
- **mancata resilienza dei Junior:** confrontando Ottobre e Novembre, si nota che il *Backend Developer 1*, passando da 26.40 a 23.04, e il *Frontend Developer*, passando da 20.89 a 17.31, riducono la loro produttività. Anche il *UI/UX Designer*, passando da 14.00 a 9.50, rallenta drasticamente: se gli sviluppatori sono bloccati nello riscrivere il codice vecchio, non possono implementare le nuove interfacce grafiche, bloccando a catena anche il lavoro di design;
- **conferma dell'Hero Factor:** il *CTO* registra 25.34 punti a Novembre. Sebbene in calo rispetto al picco, è il valore più alto di tutto il team, più del doppio del *Backend Developer 2* uscente e superiore a tutti gli altri. Questo conferma che la continuità operativa del progetto, nel momento di crisi, è dipesa interamente dalla sua capacità di lavoro individuale.

Sintesi: un sistema a singolo punto di fallimento

I dati confermano che l'organizzazione soffriva di un *Bus Factor* pari a 1. L'*AI* ha permesso ai profili *Mid-level* di performare sopra la media per mesi, ma ha generato un debito tecnico che solo un *Senior* poteva gestire. Quando l'organico si è ridotto, l'unico *Senior*, il *CTO*, è diventato il collo di bottiglia del sistema. Il crollo della *Velocity* non è stato causato solo dalla mancanza di una persona, ma dall'incapacità strutturale del team residuo di gestire la complessità tecnica in assenza di una guida disponibile a tempo pieno.

4.4 Analisi integrata e modellazione matematica del rischio

L'analisi isolata delle metriche di processo, derivanti da *Jira*, e di codice, derivanti da *GitHub*, ha evidenziato anomalie puntuali, ma non è sufficiente a spiegare le cause profonde del fallimento previsionale registrato in autunno. Questa sezione incrocia i flussi di dati per ricostruire la catena causale. L'obiettivo è superare la semplice osservazione empirica per formalizzare un modello statistico in grado di quantificare come la qualità del codice e la supervisione tecnica influenzino l'affidabilità delle stime in un team *AI-augmented*.

Matrice riepilogativa - DataSet integrato

La Tabella 4.6 aggrega le variabili chiave monitorate durante l'intero ciclo di vita del progetto da Aprile a Novembre. La tabella funge da scatola nera del sistema, mettendo in relazione gli input tecnici, come *Codice* e *Refactoring*, con gli output gestionali, come *Varianza* e *Velocity*.

Tabella 14: quadro riepilogativo delle metriche incrociate.

Mese	Churn Density	Refactoring Ratio	Hero Factor (CTO)	Story Points (Totale Team)	Planning Variance
Aprile	15286.00	85.43%	65.52%	99.46	79.13%
Maggio	13748.11	50.44%	62.76%	110.70	42.82%
Giugno	2383.88	48.57%	61.99%	102.36	13.57%
Luglio	9344.45	50.61%	53.68%	124.35	69.02%
Agosto	6264.64	53.28%	79.70%	110.45	50.04%
Settembre	8732.40	48.72%	70.72%	121.13	16.17%
Ottobre	12312.05	24.54%	43.60%	116.34	163.49%
Novembre	18673.12	58.90%	87.64%	86.71	140.54%

Un'analisi preliminare dei dati evidenzia il cortocircuito di Ottobre: la coincidenza di un *Refactoring Ratio* al minimo storico, 24.54%, e di un *Hero Factor* basso, 43.60%, ha generato il picco massimo di errore, 163.49%. Per validare statisticamente questa osservazione, si è proceduto alla modellazione matematica.

Modellazione dell'errore - Regressione Lineare Multipla

Per isolare il peso specifico di ciascuna variabile sull'errore di stima, *Planning Variance*, è stata effettuata una *Model Selection* comparando diverse configurazioni di variabili indipendenti, inclusi modelli basati su *Cycle Time* e numero di risorse.

Il modello risultato statisticamente più affidabile, con $R^2 = 0.655$, pone in relazione l'errore con tre driver fondamentali.

- **Complessità volumetrica (X_1):** *Churn Density*;
- **qualità del codice (X_2):** *Refactoring Ratio*;
- **supervisione tecnica (X_3):** *Hero Factor*.

Modello di predizione della varianza

L'analisi di regressione restituisce la seguente equazione empirica:

$$Planning\ Variance = 0.79 + (Churn\ Density \cdot 8.6 \cdot 10^{-5}) - (Refactoring\ Ratio \cdot 1.38) - (Hero\ Factor \cdot 0.43) \quad [13]$$

Il coefficiente di determinazione $R^2 = 0.655$ indica che il modello spiega il 65.50% della varianza osservata. Ciò conferma che il fallimento delle stime non è stato casuale, ma determinato per due terzi da fattori tecnici e organizzativi misurabili.

Interpretazione manageriale dei coefficienti β

L'analisi dei coefficienti risultanti dal modello di regressione non si limita a fornire un'equazione predittiva, ma offre una chiave di lettura profonda sulle dinamiche interne del team. Ogni coefficiente rappresenta una leva specifica che ha influenzato, positivamente o negativamente, la stabilità del progetto.

- ***Refactoring Ratio*** **come leva primaria - $\beta_2 = -1.38$**

  Dall'analisi emerge un dato inequivocabile: il *Refactoring Ratio* rappresenta il fattore determinante dell'intero modello. Essendo il coefficiente più alto in valore assoluto e di segno negativo, esso agisce matematicamente come il principale freno all'incertezza, evidenziando una correlazione inversa diretta tra la pulizia del codice e l'errore di pianificazione.

In termini operativi, questo conferma che l'attività di revisione umana non è un costo accessorio, ma il pilastro della stabilità: la validazione empirica si riscontra nel mese di Ottobre, quando il crollo dell'indice di pulizia al 24.54% ha rimosso l'unica barriera efficace contro l'instabilità dell'*AI*. Secondo il modello, ogni riduzione del 10% nel *Refactoring* si traduce teoricamente in un incremento dell'errore di stima di circa il 14%, rendendo di fatto ingovernabile la pianificazione;

- **Hero Factor come stabilizzatore - $\boldsymbol{\beta_3 = -0.43}$**

  Anche il coefficiente relativo all'*Hero Factor* presenta un segno negativo, confermando che la presenza operativa del *Senior*, il *CTO*, svolge una funzione cruciale di mitigazione del rischio.

  A differenza del modello basato sul mero numero di risorse, che mostrava correlazioni spurie dovute ai costi di coordinamento, questa variabile isola il valore della competenza. Il dato suggerisce che la capacità previsionale del team non è democratica: quando la supervisione scende sotto una soglia di guardia, come il 43% registrato a Ottobre, il gruppo di lavoro *Junior/Mid* perde i riferimenti necessari per valutare correttamente la complessità generata dall'*AI*, portando a una sottostima sistematica dei tempi di *delivery*;

- **Churn Density come costante di instabilità - $\boldsymbol{\beta_1 \approx +8.6 \cdot 10^{-5}}$**

  Il coefficiente positivo associato alla densità del codice identifica la vera fonte di instabilità del sistema. Sebbene il valore unitario possa apparire trascurabile, il suo impatto reale è determinato dall'effetto moltiplicatore sui grandi volumi.
  In un progetto *AI-driven*, dove la produzione di codice è massiva, raggiungendo picchi di 18000 righe/task a Novembre, questo coefficiente agisce come un generatore costante di rumore di fondo. Esso dimostra che l'adozione dell'*AI* introduce una volatilità strutturale: la facilità di scrittura del codice si paga con un aumento proporzionale della complessità, che tende naturalmente a far divergere le stime se non attivamente contrastata dalle due leve di controllo precedenti, ovvero *Refactoring* e supervisione.

Validazione del modello: lo stress test di Ottobre

Il mese di Ottobre costituisce il caso di studio critico, Case Study, che valida la robustezza del modello matematico proposto.

Mentre i mesi di Giugno e Settembre rappresentano il funzionamento fisiologico del team, con supervisione attiva e *Refactoring* costante, Ottobre definisce un esperimento naturale involontario in cui i meccanismi di controllo sono venuti meno simultaneamente.

L'analisi dei dati di questo mese conferma la capacità predittiva dell'equazione del rischio.

- **Input:** la combinazione di alto volume di codice *AI* (*Churn Density* > 12k) e assenza di freni qualitativi (*Refactoring Ratio* < 25%, Hero Factor < 45%) ha generato un punteggio di rischio teorico elevatissimo;
- **output:** il sistema ha reagito esattamente come previsto dal modello, producendo una *Planning Variance* del 163%, il massimo della serie storica.

Conclusione

Questo stress test dimostra empiricamente che l'*AI* non riduce la necessità di competenze umane, ma ne sposta il focus. La produttività apparente, *Team Velocity* elevata a 116.34 punti, in assenza di *Refactoring* e supervisione si traduce immediatamente in debito tecnico e inaffidabilità previsionale. Ottobre è la prova che la sicurezza del processo *AI-driven* dipende interamente dai fattori di mitigazione umani.

4.5 Definizione del framework operativo AI-Safe

L'equazione del rischio elaborata nel capitolo precedente

$$Planning\ Variance \approx Churn\ Density - Refactoring\ Ratio - Hero\ Factor$$

ha dimostrato che l'introduzione dell'*AI* nel ciclo di sviluppo non è neutrale: essa agisce come un moltiplicatore di velocità che, se non bilanciato da specifici meccanismi di controllo, si trasforma in un moltiplicatore di instabilità.

Alla luce dei coefficienti β emersi dalla regressione, appare evidente che le metodologie *Agile* tradizionali, basate su stime a sensazione o *Story Points*, non sono sufficienti per gestire i flussi di lavoro *AI-augmented*. È necessario adottare un approccio ingegneristico, basato su soglie quantitative rigide.

Questo paragrafo definisce il *framework AI-Safe*, un protocollo costituito da tre barriere di sicurezza progettate per neutralizzare le cause di varianza identificate.

4.5.1 Modelli operativi di Governance e mitigazione del rischio

Modello A: Traffic Light Control – Object Based Sizing

Uno dei limiti operativi riscontrati nel progetto è la difficoltà dei profili *Junior* nel fornire stime astratte, come gli *Story Points*, affidabili, specialmente quando l'*AI* maschera la complessità implementativa sottostante. Per ovviare a questo problema e prevedere il volume di codice,

Churn, prima dell'avvio dei lavori, il *framework* abbandona la stima per sforzo in favore di una stima per entità, definita *Object-Based Sizing*.
Poiché l'*AI* tende a generare codice verboso per ogni singola entità funzionale richiesta, si adotta la seguente matrice di conversione per determinare il colore del semaforo.

Tabella 15: matrice di rischio basata sulle entità.

Complessità Funzionale (Cosa devi fare?)	**Esempi Pratici (Backend / Frontend)**	**Churn Previsto (AI)**	**Zona Semaforo**
Singola Entità	· 1 API Endpoint · 1 Componente UI · Modifica a 1 Tabella	< 2000 righe	🟢 Verde *(Procedere)*
Flusso Multiplo	· 2-3 API Endpoints collegate · Intera Pagina con logica	2000 - 8000 righe	🟡 Giallo *(Checkpoint al 50%)*
Sistema Complesso	· Nuovo Modulo intero · Modifiche architetturali · > 3 Endpoints/Tabelle	> 8000 righe	🔴 Rosso *(Obbligo di Slicing)*

La regola del 3

Per i team *Junior* supportati da *AI*, si stabilisce una regola tassativa: un singolo *Task* non può mai coinvolgere più di 3 entità logiche, come ad esempio 3 API o 3 tabelle.
Un programmatore *Senior* potrebbe gestire mentalmente le interazioni tra 5-6 entità generate dall'*AI*. Un *Junior*, affidandosi al codice macchina, perde la visione d'insieme. Se il *task* tocca più di 3 entità, il rischio di *Code Bloat* ingestibile e di errore di stima sale esponenzialmente, Zona Rossa. In tal caso, il *task* deve essere spezzato in due *ticket* distinti prima di essere assegnato.

Modello B: Dynamic Quality Gate - soglia di Refactoring

Se il Modello A agisce sul volume, input, il Modello B agisce sulla qualità del codice prodotto, output.
L'analisi di regressione ha identificato nel *Refactoring Ratio*, β = -1.38, la leva più potente per abbattere l'errore di stima. I dati mostrano chiaramente che, quando il team accetta passivamente l'output dell'*AI* (*Refactoring* < 25%, come ad Ottobre), la pianificazione salta.

Per un team composto prevalentemente da profili *Junior/Mid*, il rischio principale non è la mancanza di competenza tecnica, ma l'*over-reliance* cognitiva: lo sviluppatore tende a fidarsi della soluzione fornita dal *LLM*, senza analizzarla in profondità.
Per mitigare questo rischio, si introduce la regola del *40% Golden Rule*:
nessuna Pull Request (PR) contenente codice generato da AI può essere approvata se il tasso di modifica umana (Refactoring Ratio) è inferiore al 40%.

Implementazione nel ciclo di Review

Questa soglia non è puramente burocratica, ma serve a forzare un comportamento attivo. In fase di *Code Review*:

- **analisi differenziale**, se il codice committato è quasi identico al copia e incolla standard dell'*AI*, la *PR* viene respinta d'ufficio con causale *Insufficient Human Oversight*;
- **dimostrazione di comprensione**, per superare il gate del 40%, lo sviluppatore è costretto a rinominare le variabili per adattarle al contesto specifico, *Domain Language*, estrarre metodi o semplificare la logica, *Clean Code*, o ad aggiungere commenti esplicativi o documentazione;
- **effetto freno**, questa operazione rallenta deliberatamente la fase di scrittura, che l'*AI* renderebbe istantanea, costringendo l'umano a riacquisire la proprietà intellettuale del codice. Questo tempo perso nel *Refactoring* viene recuperato, secondo il modello matematico, evitando il *debug* successivo causato da codice non compreso.

Modello C: protocollo di supervisione e integrazione

Mentre i modelli A e B agiscono sul singolo sviluppatore, il Modello C disciplina l'interazione tra i membri del team. L'analisi di regressione ha evidenziato due dinamiche critiche.

- La supervisione tecnica riduce drasticamente l'errore, β = -0.43;
- l'aumento delle risorse collaborative, come *Backend* + *Frontend*, tende lievemente ad aumentarlo, β = +0.14, a causa dei costi di allineamento sui contratti generati dall'*AI*.

Per ottimizzare questi fattori, si introducono due regole organizzative:

1. Hero Check - supervisione selettiva

Non potendo il *Senior*, *CTO*, supervisionare il 100% del codice, la sua presenza va allocata strategicamente dove il rischio è massimo.
La regola che ne deriva è: se un *task* ricade nella Zona Gialla o Rossa, definite precedentemente, è obbligatoria una *Architecture Review* preventiva di 15 minuti con il *Senior* prima di scrivere la prima riga di codice.

L'obiettivo è evitare che il *Junior* utilizzi l'*AI* per implementare un'architettura errata che richiederà giorni per essere corretta. L'investimento di 15 minuti del *Senior* previene l'effetto deriva;

2. Contract First Policy - gestione dell'integrazione

Il coefficiente positivo sulle risorse conferma che l'*AI* genera spesso disallineamenti silenziosi tra *Backend*, API, e *Frontend*, UI, specialmente in team *Junior*.

La regola da seguire è: nei *task* che coinvolgono più di una risorsa, *Backend* + *Frontend*, è vietato utilizzare l'*AI* per definire le interfacce di comunicazione.

Prima di avviare *Copilot* o *GPT*, i due sviluppatori devono scrivere manualmente il contratto dell'interfaccia, ad esempio file Swagger/*OpenAPI* o definizioni *TypeScript*. L'*AI* potrà essere usata solo successivamente per implementare il codice interno che rispetta quel contratto rigido. Questo elimina la frizione di integrazione, causa del coefficiente +0.14, trasformando l'*AI* da generatore di confusione a semplice esecutore di logica.

4.5.2 Framework di stima corretta - AI Adjusted Estimation

A conclusione del capitolo, si fornisce uno strumento pratico per convertire le stime ottimistiche dei programmatori *Junior* in pianificazioni realistiche per il *Management*.

Poiché la *Planning Variance* media osservata varia dal 13.57% di Giugno al 163.49% di Ottobre, in base alle condizioni al contorno, non è possibile applicare un *buffer* unico. Si propone quindi la formula della stima dinamica, che applica un moltiplicatore di rischio basato sui semafori definiti in precedenza.

Formula per il Project Manager

$$Tempo\ reale\ =\ (Stima\ Developer)\cdot\ (Moltiplicatore\ di\ Rischio) \qquad [14]$$

Dove il moltiplicatore di rischio è assegnato secondo la seguente tabella, calibrata sui dati storici del progetto.

Tabella 16: matrice dei moltiplicatori di rischio (K) per la stima dinamica.

Zona Semaforo	**Caratteristiche *Task***	**Moltiplicatore (K)**	**Giustificazione Empirica**
🟢 Verde	< 3 Entità	1.2x (+20%)	Corrisponde alla varianza fisiologica di Giugno (13%)
🟡 Giallo	3+ Entità o Integrazione BE/FE	1.5x (+50%)	Corrisponde alla varianza media di Agosto/Luglio
🔴 Rosso	*Task* complessi non spezzati	2.5x (+150%)	Corrisponde al picco di Ottobre/Novembre (140-160%)

Capitolo 5: Conclusioni e sviluppi futuri

Il presente lavoro di Tesi ha condotto un'indagine empirica sull'impatto dell'Intelligenza Artificiale Generativa nelle dinamiche di gestione dei progetti software all'interno di una *startup* operante nel settore *Insurtech*. Attraverso l'analisi incrociata di dati di processo, derivanti da *Jira*, e metriche tecniche, ottenute per mezzo di *GitHub*, su un arco temporale di otto mesi, è stato possibile mettere in discussione l'assunto di linearità tra automazione e velocità promesso dai *Large Language Models*, portando alla luce i costi nascosti in termini di stabilità e prevedibilità.
Di seguito saranno discussi i risultati finali in risposta alle domande di ricerca, le implicazioni teoriche e manageriali derivanti dallo studio, nonché i benefici e i limiti del lavoro svolto.

5.1 Sintesi dei risultati e risposta alle domande di ricerca

L'obiettivo primario della ricerca era identificare i fattori determinanti dello scostamento tra l'*effort* stimato e i valori a consuntivo. I risultati ottenuti permettono di fornire risposte puntuali ai quesiti formulati nel Capitolo 1.

Quali sono i fattori critici che determinano la varianza di pianificazione? La modellazione matematica, regressione lineare multipla con $R^2 = 0.655$, ha dimostrato che l'errore di stima non è casuale, ma è governato da tre variabili fondamentali.

- **Refactoring Ratio ($\boldsymbol{\beta = -1.38}$):** è emerso come il fattore di stabilità più potente. La mancanza di rilavorazione umana del codice *AI*, crollata al 24.54% nel mese critico di Ottobre , è la causa primaria dell'esplosione dell'errore di stima (+163%).
- **Hero Factor ($\boldsymbol{\beta = -0.43}$):** la supervisione del *Senior*, il *CTO*, agisce come unica barriera efficace contro l'instabilità. Quando la supervisione è scesa sotto la soglia critica del 45%, il team *Junior* ha perso la capacità di valutare la complessità reale, portando al fallimento delle stime.
- **Churn Density ($\boldsymbol{\beta_1 \approx +8.6 \cdot 10^{-5}}$):** l'alto volume di codice generato per *task* agisce come una costante di instabilità, introducendo un rumore di fondo che rende il sistema intrinsecamente più difficile da prevedere.

In che misura l'AI agisce come acceleratore o generatore di instabilità? I dati confermano l'esistenza di un paradosso della produttività. Sebbene l'adozione di *Copilot* abbia inizialmente incrementato la *Team Velocity* del 20%, tale accelerazione si è rivelata un'illusione temporanea. L'assenza di pratiche di consolidamento ha trasformato la velocità in debito tecnico involontario, culminato nel crollo produttivo di Novembre, -25.5%. L'*AI* agisce quindi come

un acceleratore non lineare: aumenta la velocità nel breve periodo, ma incrementa esponenzialmente il rischio di *Rework* nel medio periodo se non governata.

Quali metriche possono recuperare la prevedibilità? Lo studio ha dimostrato che le metriche *Agile* tradizionali, come gli *Story Points*, perdono affidabilità in contesti *AI-Augmented*, poiché misurano l'output percepito e non la complessità strutturale. Il *framework AI-Safe*, basato su soglie rigide, come la *Golden Rule* del 40% di *Refactoring*, si è dimostrato l'unico approccio teorico in grado di neutralizzare la varianza, trasformando l'incertezza in un parametro gestibile attraverso coefficienti di rischio.

5.2 Implicazioni teoriche

Dal punto di vista accademico, il lavoro contribuisce alla letteratura sull'Ingegneria del Software e del *Project Management* estendendo due teorie fondamentali.

- **Validazione della Jagged Technological Frontier:** i risultati confermano empiricamente la teoria di Dell'Acqua et al. (2023) in un contesto operativo reale. Il team ha performato eccellentemente nei *task* standard, situati all'interno della frontiera, ma ha evidentemente fallito nei *task* di integrazione complessa, fuori dalla frontiera, dimostrando che l'*AI* non appiattisce la curva di competenza, ma ne rende i confini più pericolosi perché invisibili ai.
- **Ridefinizione del debito tecnico:** la Tesi propone una nuova lettura del modello di Fowler. Nell'era dell'*AI*, il debito non è più solo una scelta strategica, *Deliberate* vs *Inadvertent*, ma diventa un sottoprodotto automatico e involontario della generazione massiva di codice. Questo impone un aggiornamento delle teorie di gestione: il controllo qualità non può più essere *ex-post*, ma deve diventare un vincolo *ex-ante*, come nel modello *AI-Safe*.

5.3 Implicazioni pratiche e manageriali

Per i professionisti del settore e i *Project Manager* operanti in *startup*, lo studio offre indicazioni operative concrete.

- **Abbandono della stima per effort:** in un team potenziato dall'*AI*, la stima basata sul tempo o sulla fatica percepita è fallace. Si raccomanda il passaggio a una stima *Object-Based*, basata sul numero di entità/API toccate, applicando i moltiplicatori di rischio identificati.

- **Governance basata sui dati:** non ci si può fidare della sensazione di avanzamento. È necessario monitorare settimanalmente il *Refactoring Ratio* e il *Churn Density*. Un crollo del *Refactoring* sotto il 30% deve essere trattato come un segnale di allarme rosso, *Early Warning*, innescando un blocco immediato dello sviluppo di nuove *feature* in favore della stabilizzazione.
- **Gestione del Bus Factor:** l'uso dell'*AI* maschera le lacune di competenza. I manager devono imporre sessioni di *Architecture Review* obbligatorie per i *task* complessi, impedendo che i profili *Junior* deleghino interamente all'*AI* la definizione delle interfacce critiche.

5.4 Benefici del Framework AI-Safe

L'adozione del modello di controllo proposto offre benefici tangibili.

- **Stabilità previsionale:** applicando i coefficienti correttivi, la varianza di pianificazione può essere ridotta da valori critici, >100%, a margini fisiologici, <20%, ripristinando la credibilità della *roadmap*.
- **Sostenibilità del codice:** l'imposizione della soglia di Refactoring costringe il team a internalizzare la conoscenza del codice generato, riducendo il rischio di creare sistemi Black Box non manutenibili.
- **Trasparenza verso gli stakeholder:** il modello fornisce metriche oggettive per giustificare agli investitori eventuali rallentamenti, inquadrandoli non come ritardi, ma come investimenti necessari per la prevenzione del debito tecnico.

5.5 Limiti della ricerca e sviluppi futuri

Nonostante la significatività dei risultati, lo studio presenta alcune limitazioni intrinseche che aprono la strada a future indagini.

- **Campione limitato:** l'analisi si basa su un singolo caso studio, relativo a una specifica *startup* Insurtech. Sebbene paradigmatico, i risultati potrebbero non essere immediatamente generalizzabili a domini differenti.
- **Specificità tecnologica:** i dati riflettono l'uso di strumenti specifici, come *GitHub Copilot* e *Jira*, in un momento storico preciso. La rapida evoluzione dei modelli *LLM* potrebbe alterare i parametri di produttività nel breve termine.

- **Fattore umano:** l'*Hero Factor* è risultato determinante a causa della specifica composizione del team, un solo *Senior*. In team più bilanciati, l'impatto della supervisione potrebbe seguire dinamiche diverse.

In conclusione, questa Tesi dimostra che l'*AI* Generativa è uno strumento potente ma allo stesso tempo può causare molti danni, se non gestita in modo adeguato. La vera sfida per il *Project Management* del futuro non sarà come spingere il team ad andare più veloce, ma come costruire le barriere necessarie per impedire che la velocità si trasformi in una corsa verso il fallimento strutturale.

Riferimenti bibliografici

[1] CHUI, Michael, et al. The economic potential of generative AI. 2023.

[2] KALLIAMVAKOU, Eirini. Research: quantifying GitHub Copilot's impact on developer productivity and happiness. *The GitHub Blog*, 2022.

[3] MCCORMICK, Mike. Waterfall vs. Agile methodology. *MPCS, N/A*, 2012, 3: 18-19.

[4] MISHRA, Alok; ALZOUBI, Yehia Ibrahim. Structured software development versus agile software development: a comparative analysis. *International Journal of System Assurance Engineering and Management*, 2023, 14.4: 1504-1522.

[5] BOEHM, Barry; TURNER, Richard N. *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional, 2003.

[6] DYBÅ, Tore; DINGSØYR, Torgeir. Empirical studies of agile software development: A systematic review. *Information and software technology*, 2008, 50.9-10: 833-859.

[7] PATERNOSTER, Nicolò, et al. Software development in startup companies: A systematic mapping study. *Information and software technology*, 2014, 56.10: 1200-1218.

[8] GIARDINO, Carmine, et al. Software development in startup companies: the greenfield startup model. *IEEE Transactions on Software Engineering*, 2015, 42.6: 585-604.

[9] CALEB, Tiffany. InsurTech Innovations and the Future of Insurance: Opportunities and Challenges of Digitalization. *Available at SSRN 6068168*, 2026.

[10] PALFREYMAN, John; MORTON, Josh. The benefits of agile digital transformation to innovation processes. *Journal of Strategic Contracting and Negotiation*, 2022, 6.1: 26-36.

[11] DELL'ACQUA, Fabrizio, et al. Navigating the jagged technological frontier: Field experimental evidence of the effects of AI on knowledge worker productivity and quality. *Harvard Business School Technology & Operations Mgt. Unit Working Paper*, 2023, 24-013.

[12] PENG, Sida, et al. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

[13] PERRY, Neil, et al. Do users write more insecure code with ai assistants?. In: *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*. 2023. p. 2785-2799.

[14] TRUMLER, Wolfgang; PAULISCH, Frances. How "specification by example" and test-driven development help to avoid technial debt. In: *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2016. p. 1-8.

[15] NAGAPPAN, Nachiappan; BALL, Thomas. Use of relative code churn measures to predict system defect density. In: *Proceedings of the 27th international conference on Software engineering*. 2005. p. 284-292.

[16] GILL, Geoffrey K.; KEMERER, Chris F. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 1991, 17.12: 1284-1288.

[17] JABRAYILZADE, Elgun, et al. Bus factor in practice. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 2022. p. 97-106.

[18] GONÇALVES, Pavlína Wurzel, et al. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. *Empirical Software Engineering*, 2022, 27.4: 99.

[19] MOHANANI, Rahul, et al. Cognitive biases in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering*, 2018, 46.12: 1318-1339.

[20] Ghigo Mattia (2025). *Raccolta_Dati_Sperimentali.xlsx* [Dataset]. Dati di progetto Nano i-Tech (Aprile-Novembre 2025). Materiale allegato alla Tesi di laurea.

[21] SCHWABER, Ken; BEEDLE, Mike. *Agile software development with Scrum*. Prentice Hall PTR, 2001.

[22] LEFFINGWELL, Dean. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional, 2010.

[23] COHN, Mike. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

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