1

Strumenti di sviluppo basati su intelligenza artificiale: paradigmi e confronto

Questo lavoro di tesi riguarda un confronto tra strumenti di sviluppo allo stato dell'arte e conseguenti considerazioni; è quindi necessaria una fase di accurata ricerca e selezione.

1.1 Il paradigma LOW CODE con agenti AI

Nonostante il Low Code sia un paradigma non particolarmente recente e ben esplorato, sono ancora disponibili numerosi strumenti di qualità adatti a semplificare complessità e tempistiche in fase di sviluppo, iterazione, testing, valutazione e deployment.
Ki Reply [1], azienda leader nel settore, è specializzata nell'utilizzo dei più recenti Gen AI tools nelle fasi del SDLC e nell'adozione di soluzioni Low Code per lo sviluppo di applicazioni.
Essendo partner di Outsystems [2], Mendix [3] e Creatio [4], ovvero compagnie che hanno creato le relative piattaforme di sviluppo Low Code, il focus sarà su questi strumenti, in particolare su Outsystems e il suo nuovo strumento di sviluppo basato su AI Generativa: Mentor.

Outsystems Mentor [5] è un tool di sviluppo basato sull'AI generativa che permette di:

- **Generare** applicazioni per mezzo di un prompt ed eventuali documenti di requisiti in input
- **Iterare** su struttura e modelli dati dell'applicazione per mezzo di modifiche o aggiunta di contenuti.
- **Incorporare** Agenti AI nell'applicazione, permettendo di selezionare il LLM specifico
- **Validare** l'applicazione per mezzo di strumenti di analisi codice approfondita

...

1.2 IDE AI-assistiti per lo sviluppo tradizionale

È stato necessario effettuare una ricerca sullo stato dell'arte degli IDE pensati per lo sviluppo di applicazioni con l'uso dell'AI, e sono stati selezionati tra i più moderni framework. Per ognuno di essi è stata effettuata una ricerca sulla letteratura ed è stata sviluppata una piccola applicazione per verificarne funzionamento, qualità e difetti. I criteri con cui sono stati selezionati sono quindi due:

- La piattaforma è stata sviluppata o caratterizzata da importanti aggiornamenti nell'ultimo anno (quindi dal 2024 in poi);
- La piattaforma si occupa in quasi completa autonomia di sviluppare l'applicazione.

Dal secondo punto ne segue che piattaforme pensate per la generazione di parti di codice su prompt (ad esempio ChatGPT) siano quindi escluse, in quanto l'autonomia di esecuzione delle task è ritenuta cruciale.
Per un primo confronto delle funzionalità, ciascuno dei prodotti qui riportati è stato testato con un semplice prompt, in cui viene richiesto di creare una webapp caratterizzata da semplici operazioni CRUD per la gestione di note. Il modello impostato su tutti gli IDE è Claude 3.5 Sonnet [6].

1.2.1 Stato dell'arte

GitHub Copilot

GitHub Copilot è uno dei più noti assistenti alla codifica basati sull'intelligenza artificiale, sviluppato da GitHub in collaborazione con OpenAI. Alimentato inizialmente da Codex e ora da modelli di nuova generazione (come GPT-4), `Copilot` si integra in ambienti di sviluppo popolari come Visual Studio Code, JetBrains IDE e Neovim. La sua principale caratteristica è la capacità di generare automaticamente codice sorgente a partire da commenti in linguaggio naturale, snippet parziali o pattern osservati nel contesto del file corrente, riducendo drasticamente l'effort manuale dello sviluppatore.
La potenza di `Copilot` risiede nella sua abilità di automatizzare interi blocchi funzionali, come funzioni complete, test unitari, e strutture di scaffolding, rendendolo uno strumento utile non solo per il completamento del codice, ma anche per la prototipazione rapida, l'esplorazione di alternative sintattiche e il refactoring.
Nel 2024 `Copilot` ha superato i 77.000 clienti aziendali, con una crescita del 180% rispetto all'anno precedente, e ha contribuito a oltre il 40% della crescita dei ricavi di GitHub, portando il tasso di ricavi annuali a 2 miliardi di dollari [7].
Studi recenti hanno valutato l'efficacia di `Copilot` in contesti pratici. Vaithilingam et al. [8] hanno evidenziato come `Copilot` riduca significativamente il tempo necessario per completare compiti di programmazione, pur sollevando questioni sulla qualità e sicurezza del codice generato. Ziegler et al. [9] confermano questi risultati in scenari reali, mostrando un aumento nella produttività e una maggiore soddisfazione degli sviluppatori. Infine, Nguyen e Nadi [10] offrono un'analisi empirica sulle interazioni tra sviluppatori e `Copilot`, mostrando come l'adozione del sistema possa influenzare il processo decisionale durante lo sviluppo.
L'impatto di `Copilot` apre la strada a nuove forme di sviluppo semi-automatico, dove l'interazione tra umano e AI assume un ruolo centrale nella scrittura del software, potenzialmente ridefinendo l'intero ciclo di vita del codice.

Windsurf

`Windsurf` [11] è un ambiente di sviluppo integrato (IDE) avanzato sviluppato da Codeium, un'azienda valutata 1,25 miliardi di dollari. Costruito sulla base di Visual Studio Code, `Windsurf` integra funzionalità di assistenza alla codifica potenziate dall'intelligenza artificiale, come Cascade [12], un agente AI che fornisce suggerimenti contestuali in tempo reale, e Supercomplete [13], un sistema di completamento del codice avanzato. Supporta oltre 70 linguaggi di programmazione ed è disponibile per Windows, macOS e Linux.
L'IDE consente una collaborazione fluida tra sviluppatore e AI attraverso la pianificazione e l'esecuzione intelligente dei compiti. Questa integrazione permette una gestione ef-

ficiente di codebase complesse, migliorando la produttività degli sviluppatori. `Windsurf` è stato adottato da grandi organizzazioni come JPMorganChase e Dell per aumentare l'efficienza nei progetti su larga scala [14].
`Windsurf` e, più in generale, Codeium, sono stati oggetto di studio in varie ricerche accademiche. Ovi et al. [15] conducono un'analisi comparativa delle prestazioni di Codeium rispetto ad altri assistenti alla codifica basati su AI, evidenziandone la solidità nei task di completamento e debugging. Lakshman e Abhinav [16] ne discutono il ruolo emergente nel contesto dell'ingegneria del software moderna. Terragni et al. [17] esplorano la crescente sinergia tra sviluppatori e strumenti AI-driven come `Windsurf`, delineandone le implicazioni future. Infine, Haque et al. [18] analizzano rischi quali hallucinations e vulnerabilità legate all'uso di strumenti assistiti da LLM, menzionando `Windsurf` per il suo approccio agent-based pensato per mitigare tali criticità.

Qodo

Tra le soluzioni più moderne per lo sviluppo assistito da AI, `Qodo` [19] (precedentemente noto come Codium) si distingue per la sua capacità di automatizzare revisioni del codice e generare test unitari di alta qualità. La piattaforma viene citata nella letteratura recente come esempio concreto di applicazione pratica dell'intelligenza artificiale nel supporto quotidiano agli sviluppatori, sia nel contesto della code review automatica [20], sia nell'analisi critica delle scelte progettuali effettuate dai test generator basati su LLM [21]. `Qodo` si integra nei flussi DevOps esistenti, offrendo feedback contestuali che riducono il debito tecnico e migliorano la produttività.

Manus

`Manus` [22] è una delle nuove piattaforme AI focalizzate sull'elaborazione del linguaggio naturale (NLP), con l'obiettivo di competere con i principali attori globali nel campo, come ChatGPT, DeepSeek e Gemini. Secondo un'analisi comparativa [23], `Manus` si distingue per l'accuratezza nelle risposte contestuali e per una forte specializzazione nelle applicazioni professionali. È citata anche tra i protagonisti della cosiddetta rinascita dell'AI cinese, insieme ad altri modelli emergenti [24], a testimonianza del crescente interesse verso soluzioni linguistiche più verticali e localizzate.

Cursor

`Cursor` è un ambiente di sviluppo integrato (IDE) potenziato dall'intelligenza artificiale, progettato per ottimizzare l'efficienza degli sviluppatori attraverso funzionalità avanzate. Tra queste, spiccano l'autocompletamento contestuale del codice, la generazione automatica di funzioni a partire da descrizioni in linguaggio naturale e un sistema di debug assistito che fornisce feedback immediati sugli errori . Recentemente, `Cursor` ha

introdotto la funzionalità "Agenti", che consente all'editor di comprendere il contesto, utilizzare il terminale e completare compiti in modo autonomo basandosi su un singolo prompt, riducendo così l'intervento manuale dello sviluppatore . Queste innovazioni hanno attirato l'attenzione di oltre 30.000 utenti, tra cui ingegneri di OpenAI e Shopify, e hanno portato la startup a ottenere un finanziamento di 60 milioni di dollari, raggiungendo una valutazione di 400 milioni [25].
Attualmente, la letteratura accademica su `Cursor` è limitata. Tuttavia, un articolo recente [26]propone un framework dettagliato per lo sviluppo di interfacce utente frontend assistito dall'AI, delineando 76 passaggi atomici per garantire coerenza, scalabilità e iterazione post-deployment . Questo lavoro evidenzia il potenziale di strumenti come `Cursor` nel facilitare lo sviluppo di interfacce utente complesse.

Replit

`Replit` [27] è una piattaforma di sviluppo online che consente di scrivere, eseguire e condividere codice direttamente dal browser, senza bisogno di installare nulla. Fondata nel 2016, oggi conta oltre 30 milioni di utenti e supporta più di 50 linguaggi di programmazione. `Replit` è nota per la sua interfaccia collaborativa in tempo reale, simile a Google Docs per il codice. Nel 2023 ha raccolto 97 milioni di dollari in un'estensione del round di Serie B, raggiungendo una valutazione di 1,16 miliardi di dollari [28]. Tra le sue innovazioni più recenti ci sono Ghostwriter, un assistente AI per la scrittura del codice, e Agent, uno strumento che permette di generare intere applicazioni partendo da semplici descrizioni in linguaggio naturale [29].

1.2.2 Esperimento comparativo sugli ambienti AI-assistiti

Versioni valutate

- **Windsurf** 1.9.0 (Windsurf Extension 1.46.2)
- **Cursor** 0.50.5 (VSCode 1.96.2)
- **Qodo** 1.0.10 (VSCode 1.100.2)
- **Copilot** 1.322.0, **Copilot Chat** 0.27.1 (VSCode 1.100.2)
- **Replit** versione del 9 maggio 2025

Metodologia di valutazione

Per confrontare l'efficacia degli ambienti di sviluppo basati su agenti AI, è stato chiesto a ciascun IDE di generare un'applicazione CRUD (una semplice app per note). Ogni ambiente ha avuto accesso allo stesso modello linguistico sottostante (Claude 3.5 Sonnet [6]), in modo da isolare il più possibile l'effetto dell'interfaccia e del comportamento dell'agente.
In tutte le iterazioni del test è stato sottoposto il seguente prompt:

```
Build a full-stack Note Manager application. Use FastAPI for the backend
and plain HTML + JavaScript for the frontend.

The backend must expose a REST API at the base path /api/notes/ supporting
full CRUD operations: create, read (all and by id), update, and delete
notes. Each note must have the fields: id, title, content, createdAt,
updatedAt.

Store notes persistently using a file or database (not in-memory).

The frontend must allow users to: View the list of existing notes, Create
a new note, Edit an existing note, Delete a note and Filter notes by title
using a search bar. All operations must update the UI without errors or
page reloads.

Also, write at least four automated tests for the backend API using pytest,
covering all CRUD endpoints (create, read, update, and delete). The tests
must verify the actual behavior and responses of the API.

Organize the project into three top-level folders: backend/, frontend/,
and tests/.

Write clear, modular, and maintainable code.
```

Sono state raccolte le seguenti metriche:

- **Tempo di esecuzione dell'agente**: tempo, espresso in minuti, durante il quale l'agente ha effettivamente generato codice o risposte, escludendo i tempi di inattività o di attesa da parte dell'utente.
- **Tempo di verifica manuale**: tempo, espresso in minuti, impiegato dall'utente per analizzare il comportamento dell'applicazione, individuare eventuali difetti e redigere eventuali prompt correttivi. Corrisponde all'effort umano nel ciclo di sviluppo assistito.
- **Tempo di verifica automatica**: tempo, espresso in minuti, necessario all'esecuzione di strumenti automatici di validazione, come ad esempio una test suite. Si tratta di tempo macchina, non imputabile all'intervento diretto dell'utente.
- **Interazioni correttive**: numero totale di interventi espliciti dell'utente necessari per correggere malfunzionamenti evidenti dell'applicazione (ad esempio: API non funzionanti, test falliti, mancanza di persistenza dei dati).

Un'applicazione è considerata completa solo quando soddisfa tutti i criteri della checklist funzionale [Tabella 1.1] riportata nella pagina seguente.

ID	Requisito	Descrizione operativa della validazione	Criterio di superamento
1	CRUD API funzionante	Dall'interfaccia utente, si eseguono manualmente le seguenti operazioni: (a) creazione di una nota (b) visualizzazione (c) modifica (d) eliminazione Dopo ciascuna azione si verifica che l'interfaccia rifletta correttamente l'esito dell'operazione.	Tutte e quattro le operazioni sono eseguite correttamente, senza errori visibili dal lato utente.
2	Persistenza su file o database	Dopo aver creato e modificato almeno una nota, si riavvia l'applicazione (interrompendo e riavviando il server). Al riavvio si verifica la presenza dei dati modificati.	I dati precedenti al riavvio risultano ancora accessibili e modificabili.
3	Frontend completo (view, create, edit, delete)	Si accede all'interfaccia utente e si verifica manualmente la possibilità di: (a) visualizzare le note esistenti (b) crearne una nuova (c) modificarne una (d) eliminarne una Ogni azione deve riflettersi in tempo reale sull'interfaccia, senza errori o ricaricamenti forzati.	Tutte e quattro le operazioni sono disponibili dal frontend e funzionano correttamente, con aggiornamento immediato e coerente dell'interfaccia.
4	Test automatici presenti e funzionanti	Si esegue `pytest` (o comando equivalente). Si verifica che: (a) siano presenti quattro o più test (b) esista almeno un test per ognuna delle quattro operazioni CRUD via API (c) la suite venga eseguita senza errori (d) le asserzioni siano coerenti con i comportamenti attesi	Tutti e quattro i criteri elencati risultano soddisfatti.
5	Struttura del progetto	Si analizza la struttura dei file e delle directory, verificando la presenza di cartelle dedicate (es. `frontend/`, `backend/`, `tests/`), e il corretto raggruppamento dei file per responsabilità (es. logica, interfaccia, modelli).	Sono presenti almeno tre directory distinte (frontend, backend, tests), e i file sono allocati coerentemente al loro ruolo. Non vi è ambiguità tra logica frontend e backend.
6	Barra di ricerca funzionante	Si creano almeno tre note con titoli distinti. Si inserisce un termine nel campo di ricerca e si verifica che i risultati si filtrino correttamente. Una volta cancellata la ricerca, si verifica la ricomparsa di tutte le note.	Il filtro opera in tempo reale (o al submit) e restituisce risultati coerenti con il testo immesso.

Tabella 1.1: Checklist dei requisiti applicativi, con validazione e criteri di superamento

La verifica della corretta funzionalità della CRUD API (ID Requisito **1**) è stata automatizzata per mezzo del codice qui riportato [Listing 1.1], eseguito con il comando `pytest` durante l'esecuzione dell'applicazione. La verifica dei restanti requisiti funzionali è stata effettuata manualmente esplorando la directory del progetto ed eseguendo l'applicazione in locale.

Listing 1.1: Test automatico API

```python
import httpx

BASE_URL = "http://localhost:8000"

def test_crud_sequence():
    note = {
        "title": "Test Title",
        "content": "Test Content"
    }

    # CREATE
    response = httpx.post(f"{BASE_URL}/api/notes/", json=note)
    assert response.status_code in (200, 201)
    created = response.json()
    assert "id" in created
    note_id = created["id"]

    # READ
    response = httpx.get(f"{BASE_URL}/api/notes/")
    assert response.status_code == 200
    notes = response.json()
    assert any(n["id"] == note_id for n in notes)

    # UPDATE
    updated = {**note, "title": "Updated Title"}
    response = httpx.put(f"{BASE_URL}/api/notes/{note_id}", json=updated)
    assert response.status_code == 200
    assert response.json()["title"] == "Updated Title"

    # DELETE
    response = httpx.delete(f"{BASE_URL}/api/notes/{note_id}")
    assert response.status_code == 200

    # VERIFY DELETION
    response = httpx.get(f"{BASE_URL}/api/notes/{note_id}")
    assert response.status_code in (404, 400)
```

Qualora uno o più requisiti non fossero stati raggiunti, si rendeva necessaria un'interazione correttiva, ovvero un prompt inviato all'agente con lo scopo di completare l'applicazione entro l'iterazione successiva.
La struttura di questo prompt è formalizzata come segue:

```
The following requirements are not met:
- [Requirement 1]: [Observed deviation]. Expected: [Expected behavior].
- ...
- [Requirement N]: [Observed deviation]. Expected: [Expected behavior].
Please fix the above issues. Do not modify parts that are already working.
```

In presenza di errori in compilazione o in esecuzione, `[Observed deviation]` è stato associato all'errore, riportato integralmente o in forma abbreviata qualora lo stack trace risultasse eccessivamente verboso; `[Expected behavior]` è stato in questi casi sostituito con `"No error"`.
La presenza del termine `"Please"` all'inizio dell'ultima istruzione è deliberata: diversi studi dimostrano che i LLM sono in grado di riconoscere segnali di cortesia nei prompt [30], e che un tono moderatamente cortese può migliorare la qualità delle risposte generate [31].

Le interazioni che non comportano una correzione diretta (come conferme, review di file o prompt generati automaticamente dall'IDE) non sono conteggiate, al fine di mantenere un confronto equo tra strumenti che adottano stili di interazione differenti, pur considerando la presenza di una trascurabile influenza sul tempo totale.

1.3 Risultati

L'esperimento ha coinvolto cinque ambienti di sviluppo AI-assistiti: **Copilot**, **Cursor**, **Qodo**, **Replit** e **Windsurf**, ciascuno testato su tre iterazioni indipendenti. Ogni IDE ha generato un'applicazione web full-stack con funzionalità CRUD e requisiti frontend specifici, sulla base di un prompt uniforme. I risultati ottenuti sono stati aggregati per calcolare medie su tre metriche principali: **tempo medio dell'agente**, **tempo medio di interazione manuale** e **numero medio di correzioni**.

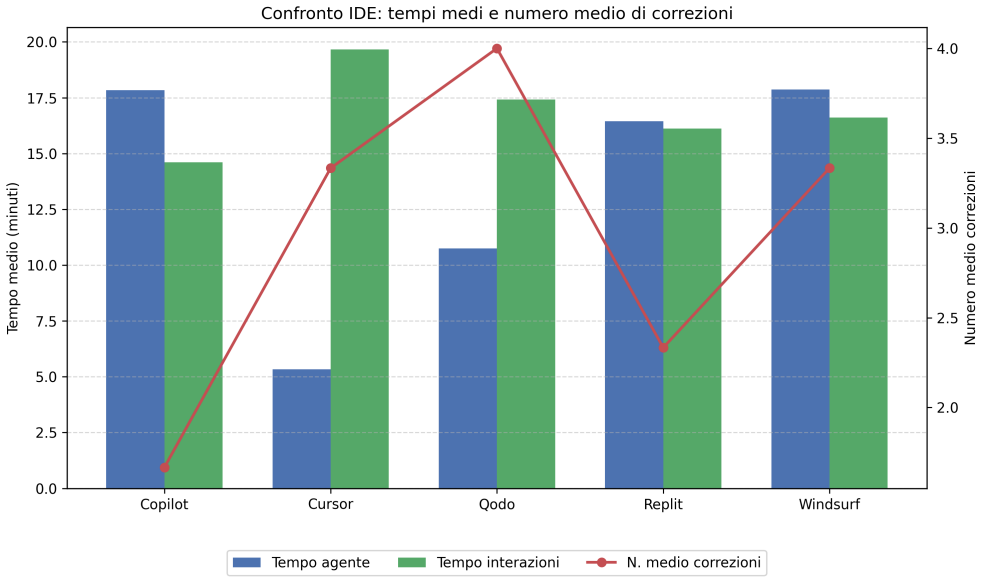

Figura 1.1: Confronto IDE: tempi medi e numero medio di correzioni

La Figura 1.1 mostra una sintesi visiva dei risultati. Le barre verticali rappresentano i tempi medi (agente e interazioni), mentre la linea rossa indica il numero medio di correzioni necessarie per completare ciascun progetto.

I valori aggregati sono i seguenti:

IDE	Tempo agente	Tempo interazioni	Correzioni
Copilot	17.83	14.08	1.67
Cursor	5.33	19.14	3.33
Qodo	10.75	16.53	4.00
Replit	16.44	16.03	2.33
Windsurf	17.86	15.81	3.33

Tabella 1.2: Tempi medi di esecuzione dell'agente e delle interazioni con esso (in minuti), e relativo numero medio di correzioni.

Tutti gli strumenti hanno portato al completamento del progetto, ma con prestazioni e comportamenti eterogenei in termini di effort richiesto all'utente.

1.3.1 Analisi qualitativa delle correzioni

Oltre al numero complessivo di correzioni per ciascun IDE, sono stati analizzati gli aspetti dell'applicazione che risultano più frequentemente mancanti durante la generazione automatica.
Sono stati considerati sei tipi di requisito, corrispondenti ai criteri funzionali della checklist funzionale in Tabella 1.1:

- **API**
- **Persistenza dati**
- **Frontend**
- **Tests**
- **Organizzazione progetto**
- **Searchbar**

L'analisi aggregata ha evidenziato le seguenti frequenze:

Requisito	Frequenza
Tests	20
API	16
Frontend	14
Searchbar	2
Organizzazione progetto	1
Persistenza dati	0

Tabella 1.3: Frequenza assoluta dei requisiti mancanti durante lo sviluppo automatico

Si osserva che i test automatici sono il requisito meno frequentemente implementato correttamente al primo tentativo, confermando la tendenza degli agenti a sottovalutare le fasi di validazione automatica. Errori nelle **API** e nella **UI** (Frontend) sono comuni nelle prime iterazioni, mentre **Searchbar** e **Organizzazione progetto** risultano più stabili o trascurabili. Nessuna implementazione ha mostrato problemi relativi alla persistenza dei dati nel database.

Distribuzione per IDE Ogni IDE mostra un pattern diverso nella tipologia di errori: ad esempio, alcuni tendono a dimenticare i test, altri a generare frontend incompleti o non correttamente funzionanti. La Figura 1.2 illustra la distribuzione dei requisiti mancanti per IDE in forma di barre sovrapposte.

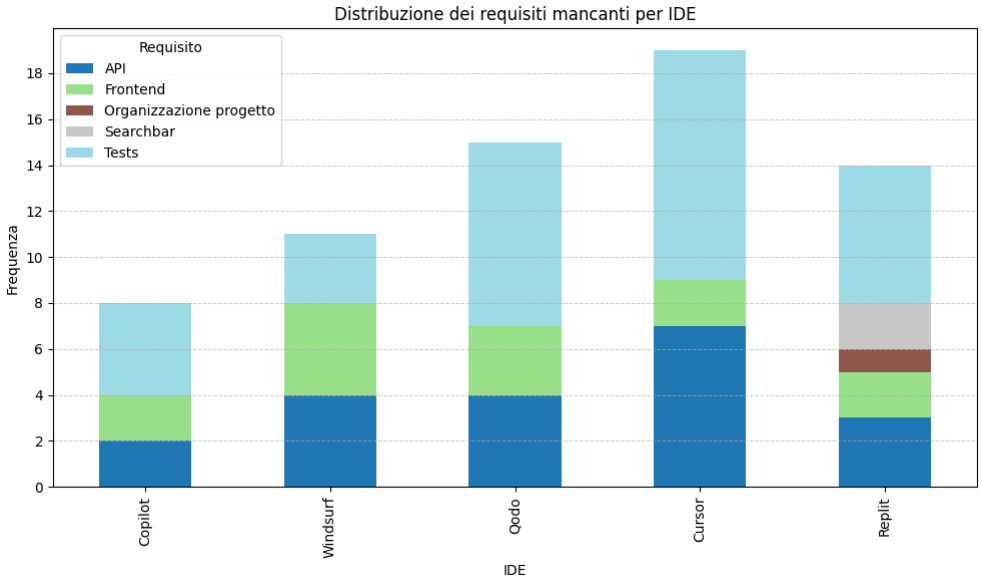

Figura 1.2: Frequenze di errori su tutti i test per ogni IDE

Questa analisi qualitativa permette di comprendere non solo quanto un agente sbaglia, ma anche dove commette errori ricorrenti, offrendo indicazioni utili per selezionare strumenti adeguati in base al contesto progettuale.

1.3.2 Analisi qualitativa della soluzione generata

Dopo aver valutato quantitativamente il comportamento degli strumenti AI-assistiti nella generazione di un'applicazione CRUD, è stata condotta un'analisi qualitativa sul codice prodotto. L'obiettivo è misurare la qualità, la sicurezza e la manutenibilità delle soluzioni generate, al fine di ottenere una visione più completa delle prestazioni dei vari ambienti.

Obiettivi dell'analisi

L'analisi si è focalizzata su tre dimensioni principali:

- **Manutenibilità e leggibilità del codice**, attraverso l'individuazione di code smell, funzioni complesse, strutture ridondanti e violazioni di buone pratiche.
- **Sicurezza applicativa**, tramite l'identificazione di vulnerabilità note o comportamenti a rischio.
- **Qualità dell'interfaccia grafica**, attraverso la valutazione automatica della soluzione.

Progetti analizzati

Sono stati analizzati i quindici progetti generati nella fase iniziale dell'esperimento, prodotti a gruppi di tre da un diverso IDE AI-assistito (Copilot, Cursor, Qodo, Replit, Windsurf). Tutti i progetti implementano una webapp per la gestione di note con architettura full-stack (FastAPI per il backend, HTML e JavaScript per il frontend), generata a partire dallo stesso prompt.

Strumenti utilizzati

Per effettuare un'analisi strutturata e comparabile, sono stati adottati i seguenti strumenti open source:

- **SonarQube [32]**: analizzatore statico per codice Python e JavaScript, utilizzato per individuare code smell, duplicazioni, bug potenziali e vulnerabilità. È stato utilizzato in versione Community con deploy locale via Docker.
- **Bandit [33]**: tool di analisi della sicurezza specifico per codice Python, con particolare attenzione a vulnerabilità note (es. uso di `eval`, gestione file non sicura, credenziali hardcoded).

- **Lighthouse [34]**: tool di valutazione automatica della qualità del frontend (accessibilità, performance, best practices), eseguito in locale tramite Chrome DevTools.

Metriche raccolte

Le metriche estratte da ciascun progetto sono state organizzate in due tabelle comparative. La prima tabella presenta le metriche raccolte tramite SonarQube, mentre la seconda tabella illustra le metriche ottenute da Lighthouse. Inoltre, per Lighthouse, sono stati considerati anche i messaggi specifici per ciascuna metrica, come "Render blocking requests" o "Network dependency tree".

SonarQube Le metriche estratte da SonarQube sono focalizzate sulla qualità e la sicurezza del codice. La tabella sottostante riporta i principali indicatori analizzati:

Metrica	**Descrizione**
Security	Indica la presenza di vulnerabilità di sicurezza nel codice, come XSS o SQL injection; indica il numero di problemi rilevati.
Reliability	Misura la stabilità del codice, rilevando possibili errori che potrebbero causare malfunzionamenti; indica il numero di problemi rilevati.
Maintainability	Valuta la facilità con cui il codice può essere mantenuto, modificato o esteso; indica il numero di problemi rilevati.
Coverage (%)	Percentuale di codice coperto da test automatici. Una copertura più alta indica una maggiore affidabilità del codice.
Duplications (%)	Percentuale di codice duplicato in più file, che può portare a difficoltà di manutenzione e aumentare il debito tecnico.
Security Hotspots	Aree del codice che potrebbero potenzialmente essere vulnerabili, anche se non necessariamente a rischio immediato; indica il numero di problemi rilevati.

Lighthouse Per quanto riguarda Lighthouse, sono stati considerati quattro principali indicatori di performance e qualità, che riportano un valore da 0 (peggiore) a 100 (migliore), con un focus sui messaggi di avviso specifici per ogni metrica. La tabella sottostante riporta le metriche analizzate:

Metrica	Descrizione
Performance	Misura la velocità e l'efficienza del sito web, con punteggi che indicano l'ottimizzazione delle risorse e dei tempi di caricamento.
Accessibility	Analizza l'accessibilità del sito per utenti con disabilità, segnalando eventuali violazioni relative a contrasto dei colori, etichettatura di pulsanti, ecc.
Best Practices	Valuta se il sito segue le migliori pratiche in termini di sviluppo web, come l'uso di HTTPS, errori di browser, o configurazioni errate.
SEO	Misura l'ottimizzazione del sito per i motori di ricerca, considerando la presenza di meta tag, descrizioni e la struttura del contenuto.

In aggiunta, per ogni metrica, sono stati presi in considerazione i messaggi specifici rilevati durante l'analisi, ad esempio:

- **Performance**: "Render blocking requests", "Network dependency tree", "Use efficient cache lifetimes".
- **Accessibility**: "Background and foreground colors do not have a sufficient contrast ratio".
- **Best Practices**: "Browser errors were logged to the console".
- **SEO**: "Document does not have a meta description".

Questi messaggi forniscono dettagli sui problemi specifici che potrebbero influire sulle performance e sull'usabilità del sito, consentendo di adottare interventi mirati per migliorare la qualità complessiva del progetto.

Analisi della qualità del codice con SonarQube

L'analisi del codice, effettuata tramite SonarQube, ha fornito metriche relative alla sicurezza, affidabilità, manutenibilità, copertura e duplicazione del codice. Un'osservazione importante riguarda la **copertura del codice**, che è uno degli indicatori più critici per valutare la qualità di un'applicazione.

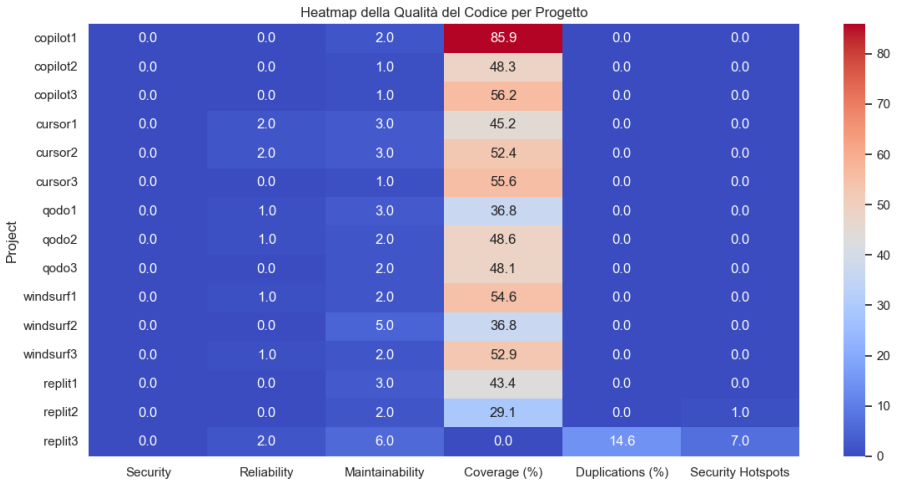

Figura 1.3: Heatmap dei risultati delle analisi di SonarQube sui progetti

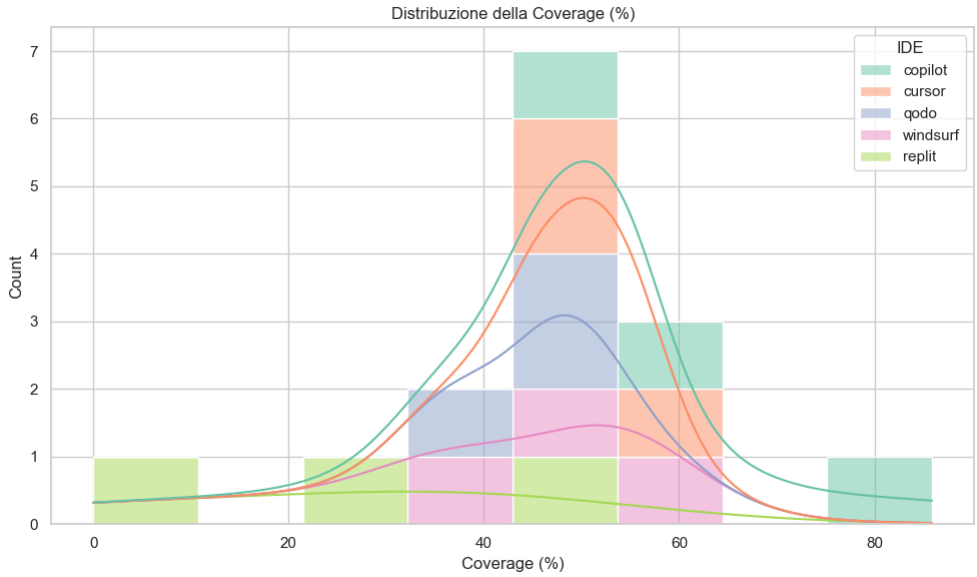

Figura 1.4: Distribuzione della coverage sui progetti analizzati

La distribuzione della copertura del codice è presentata nella Figura 1.4. I progetti sviluppati con `Copilot` e `Cursor` hanno mostrato una copertura media elevata (con picchi di 85.9% e 54.6% rispettivamente), seguiti da `cursor1` e `qodo1` con una copertura tra il 36% e il 45%. In generale, i progetti che utilizzano `Copilot` sembrano avere una copertura più consistente rispetto agli altri IDE, con un punteggio massimo che raggiunge quasi il 90% di copertura.

D'altra parte, i progetti generati con `Replit` mostrano una copertura del codice significativamente inferiore, con il progetto `replit3` che non raggiunge alcuna copertura (`0%`). Questo suggerisce che, sebbene `Replit` possa essere efficace per sviluppare rapidamente prototipi o MVP, non garantisce lo stesso livello di qualità del codice in termini di test automatizzati o copertura completa.

Per quanto riguarda la sicurezza e la manutenibilità, le metriche mostrano che nessun progetto, tranne quelli sviluppati con `Replit`, ha registrato criticità di sicurezza gravi o problemi di manutenibilità evidenti. La Figura 1.3 mostra che la maggior parte dei progetti presenta valori di affidabilità e manutenibilità superiori a 50%, con `copilot1` che eccelle con un punteggio di 85.9% in termini di sicurezza.

Analisi della sicurezza del codice con Bandit

L'analisi della sicurezza del codice tramite Bandit ha rivelato un aspetto già notato nei test effettuati con SonarQube.

Figura 1.5: Risultati dell'analisi di sicurezza effettuata con Bandit sui progetti

I progetti sviluppati con `Replit` sono stati gli unici a presentare vulnerabilità di sicurezza. In particolare, `replit3` ha mostrato un problema di alta gravità e due di media gravità, mentre `replit1` e `replit2` non hanno presentato problemi significativi. Questo implica che, pur avendo una buona performance generale, `Replit` potrebbe richiedere un'attenzione particolare per quanto riguarda la sicurezza.

Analisi del frontend con Lighthouse

Le performance del frontend sono state misurate utilizzando Lighthouse, una suite di strumenti che ha analizzato vari aspetti della webapp, come performance, accessibilità, best practices e SEO.

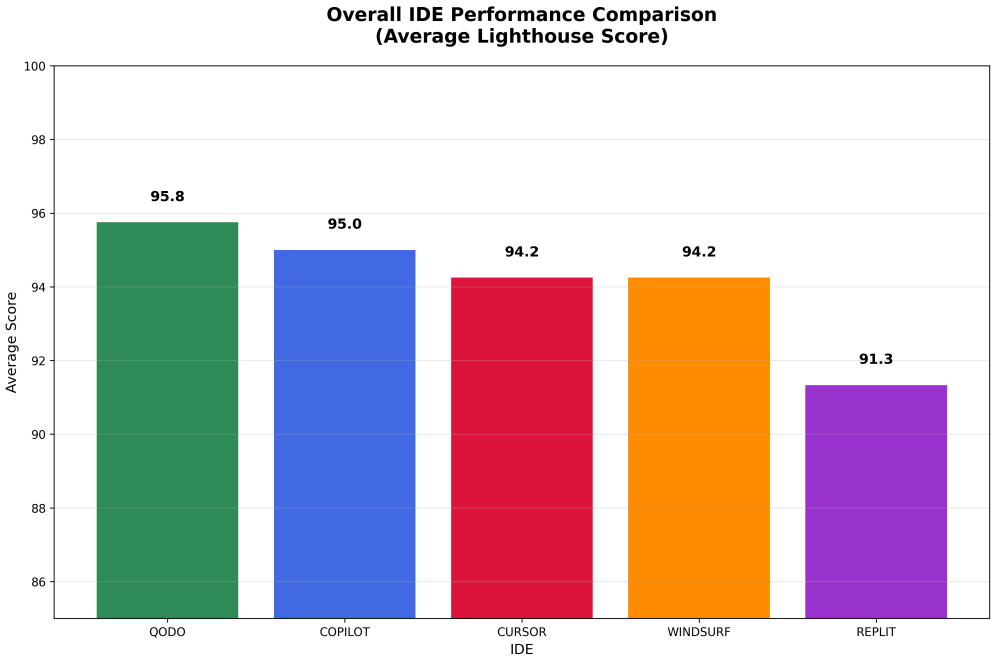

Figura 1.6: Risultati medi per IDE delle analisi effettuate con Lighthouse [34].

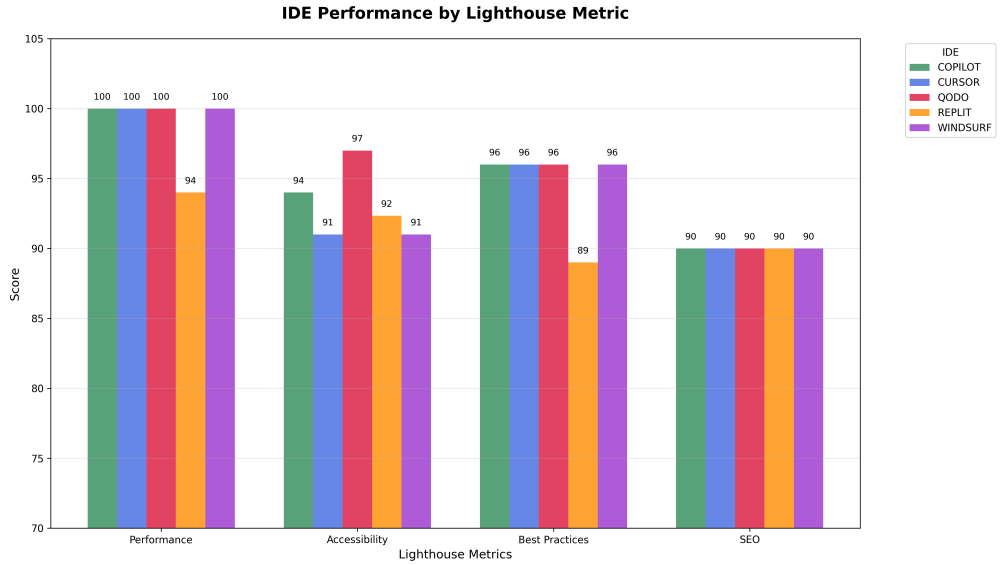

Figura 1.7: Risultati specifici per IDE relativi alle analisi effettuate con Lighthouse.

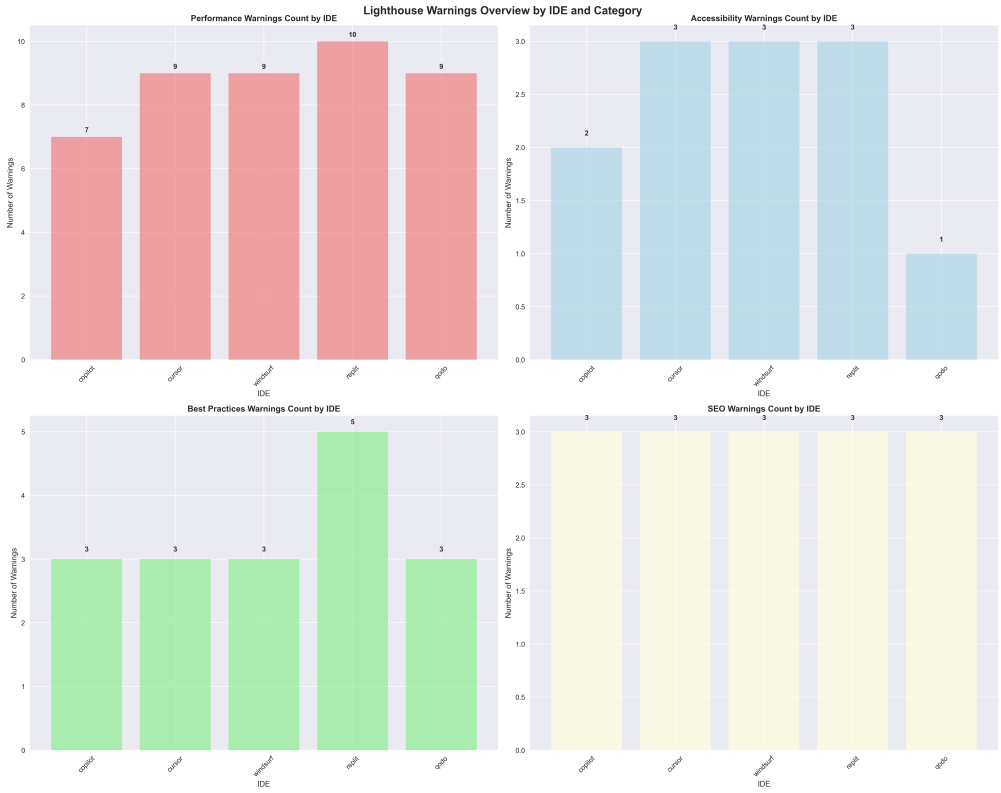

Figura 1.8: Problemi segnalati da Lighthouse per ogni metrica.

Nella Figura 1.6, possiamo osservare che i punteggi di performance di tutti gli IDE sono stati elevati, con la maggior parte dei progetti che ha raggiunto il punteggio massimo di 100 nella metrica di *Performance*. Questo suggerisce che, dal punto di vista delle performance, gli IDE non presentano grandi differenze. Tuttavia, alcune discrepanze sono evidenti nelle altre metriche.
Per esempio, `replit2` ha ottenuto un punteggio significativamente più basso per l'accessibilità (86%), principalmente a causa di problemi legati ai colori di sfondo e contrasto. In confronto, `copilot2`, `windsurf2` e `cursor2` hanno ottenuto punteggi molto più elevati (91% o superiori). Questo evidenzia una leggera debolezza di `Replit` nel garantire l'accessibilità, nonostante le ottime performance generali.
Per quanto riguarda le best practices, i punteggi sono stati generalmente elevati per tutti gli IDE, con solo `replit3` che ha mostrato un punteggio inferiore di 75% a causa della presenza di script legacy e di dipendenze di rete non ottimizzate.
Infine, la metrica SEO è risultata uniforme, con tutti gli IDE che hanno ottenuto 90% in SEO, eccetto `replit3`, che ha mostrato una leggera debolezza (75%) a causa dell'uso di cookie di terze parti e della mancanza di una descrizione meta.

Conclusioni quantitative sull'analisi dei risultati

Analizzando quantitativamente i risultati, emergono alcune tendenze chiare:
`Copilot` e `Windsurf` sono gli IDE con le performance migliori in termini di qualità del codice e copertura del test, con `copilot1` che ha mostrato una copertura eccellente del 85.9% e `windsurf1` che ha mantenuto valori elevati in tutte le metriche di qualità.
`Replit`, sebbene performante dal punto di vista della generazione rapida di codice, ha mostrato significativi limiti nella copertura del codice e nelle problematiche di sicurezza, con alcuni progetti (come `replit3`) che hanno ottenuto una copertura nulla e una vulnerabilità di alta gravità.
Le metriche di Lighthouse hanno mostrato punteggi molto simili tra gli IDE, ma `Replit` ha mostrato un peggioramento nei punteggi di accessibilità e SEO, particolarmente nei progetti più recenti.
Questi risultati confermano che non esiste un IDE ideale per tutti i tipi di progetto, ma che la scelta dello strumento deve dipendere dalle necessità specifiche del progetto, dal livello di supervisione umana richiesto e dalla criticità delle problematiche di sicurezza e di accessibilità.

1.4 Discussione

Dall'analisi dei dati emerge che ogni IDE AI-assistito mostra un diverso equilibrio tra autonomia dell'agente, intervento umano richiesto e numero di correzioni necessarie.

Qodo si distingue per il numero medio più elevato di correzioni (4.00), accompagnato da un effort manuale significativo (16.53 minuti) e un tempo agente intermedio (10.75 minuti). Questi valori indicano un comportamento molto dinamico ma instabile: l'agente produce codice con una certa velocità, ma frequentemente incompleto o impreciso, richiedendo continue revisioni da parte dell'utente. La velocità di generazione, quindi, non si traduce in risparmio di effort complessivo, considerando che si sono verificati frequenti fallimenti nella generazione di test e UI. `Qodo`, inoltre, non è al momento provvisto di funzionalità di esecuzione di comandi su terminale, in quanto si limita ad elencare le azioni necessarie all'avvio dell'applicazione; ciò ha inficiato sul tempo necessario alle interazioni. L'analisi qualitativa ha confermato questa tendenza, evidenziando come i test automatici siano stati il requisito più frequentemente trascurato, con un'alta frequenza di errori nelle operazioni di validazione e verifica, nonostante la rapidità di generazione del codice.

Cursor è l'IDE con il tempo agente più basso (5.33 minuti), ma anche quello con il tempo medio di interazione manuale più alto (19.14 minuti). Questo suggerisce che, pur generando risposte rapidamente, le sue soluzioni richiedono una notevole attività di verifica, debug e modifica da parte dello sviluppatore. Questo è anche dovuto al fatto che, come `Qodo`, l'IDE non è provvisto di una funzionalità di esecuzione autonoma di comandi su terminale (pur offrendo la possibilità di eseguire i comandi proposti al click di un tasto "run", l'agente non ha modo di ispezionare il terminale). Il numero di correzioni (3.33) è elevato, ma coerente con l'effort richiesto. L'elevata incidenza di errori legati ad API e Tests, anche nelle iterazioni finali, sembra indicare che `Cursor` rappresenti uno strumento reattivo più orientato alla prototipazione rapida che alla solidità architetturale e probabilmente maggiormente utile in fase di esplorazione. L'analisi qualitativa ha confermato che, nonostante l'efficace prototipazione, le aree più critiche di `Cursor` rimangono la gestione delle API e la qualità dei test automatici, che non sono sempre correttamente implementati, richiedendo un intervento umano costante.

Copilot si colloca tra le soluzioni più bilanciate: il tempo agente è alto (17.83 minuti), ma il numero di correzioni è contenuto (1.67) e l'interazione manuale si mantiene sotto la soglia dei 15 minuti. A differenza di `Qodo` e `Cursor`, `Copilot` dispone di un tool di esecuzione comandi da terminale, con relativa ispezione degli output stampati su di esso; ciò rende l'interazione molto più scorrevole. Tuttavia, si sono verificati casi di blocco del sistema, che possono influire negativamente sull'esperienza d'uso, soprattutto in scenari reali dove la fluidità operativa è essenziale. La stabilità relativa di `Copilot` si riflette anche nella distribuzione delle correzioni, che compaiono spesso nella prima iterazione e si esauriscono rapidamente. L'impressione è quindi quella di uno strumento ormai maturo ma non sempre stabile. L'analisi qualitativa ha messo in luce che, sebbene il codice generato da `Copilot` sia di buona qualità, con una riduzione significativa dei difetti e una buona copertura dei test, rimangono dei margini di miglioramento nella

gestione della sicurezza e delle vulnerabilità. Nonostante ciò, `Copilot` si distingue come uno degli IDE più completi ed efficaci in termini di coerenza e affidabilità.

Replit presenta un profilo simile a quello di `Copilot`, con tempi molto vicini per agente (16.44 minuti) e interazione (16.03 minuti). Anche il numero di correzioni (2.33) è moderato, suggerendo un buon bilanciamento tra autonomia e affidabilità. Il tool mostra maggiore varietà nei tipi di errore, inclusi Searchbar e Organizzazione del progetto, probabilmente riconducibili a specificità nella pipeline generativa adottata dall'agente. È interessante notare come `Replit` mantenga questo equilibrio pur operando interamente da browser, senza ambienti di sviluppo locali; questa forma di disaccoppiamento si mostra come punto di forza, nonostante il problema dell'adozione di un nuovo strumento di sviluppo su utenti esperti dia spazio a nuove difficoltà. L'ambiente si mostra fortemente orientato all'autonomia dell'agente, che è capace di effettuare screenshot dell'applicazione per verificarne il corretto funzionamento, eseguire comandi su terminale e verificarne gli output. La qualità del risultato finale, unita alla semplicità di accesso, lo rende una scelta interessante per team distribuiti o scenari educational. L'analisi qualitativa ha confermato che, pur mantenendo un buon equilibrio tra autonomia e controllo, `Replit` ha mostrato alcune lacune in ambito di ricerca (Searchbar) e nell'organizzazione generale del progetto. Questi difetti, sebbene minori, possono rallentare il processo di sviluppo e richiedere una maggiore supervisione.

Windsurf si comporta in modo costante, con valori mediamente alti in tutte le metriche: 17.86 minuti per l'agente, 15.81 per l'interazione e 3.33 correzioni. Questo suggerisce un agente che tenta di coprire ampie porzioni del ciclo di sviluppo, ma che spesso commette errori da correggere manualmente: la distribuzione delle correzioni mostra una prevalenza di problemi legati ad API, Tests e Frontend, spesso in contemporanea. Questo suggerisce un agente ambizioso ma ancora incline a errori sistemici. La presenza costante di difetti in più aree implica un ciclo iterativo più lungo e laborioso. Similarmente a `Copilot`, `Windsurf` è provvisto di un tool per l'esecuzione autonoma di comandi da terminale, con ispezione dei risultati. L'impressione complessiva è quella di uno strumento potente ma che, in contesti complessi, richiede ancora aggiustamenti per garantire coerenza e affidabilità. L'analisi qualitativa ha confermato che, nonostante un supporto solido per le fasi di sviluppo avanzato, `Windsurf` necessita di una continua rifinitura, soprattutto nei test automatici e nella generazione di interfacce utente coerenti.

1.4.1 Riflessioni pratiche

I risultati suggeriscono che non esiste, ad oggi, un IDE agentico "migliore" in senso assoluto, ma piuttosto strumenti con caratteristiche tecniche e interattive differenti, più o meno adatte a specifici contesti d'uso. La scelta dovrebbe dipendere dal tipo di progetto, dal livello di supervisione richiesto e dalla fase del ciclo di sviluppo.

In scenari dove la velocità di generazione è prioritaria (ad esempio nella fase di prototipazione o nella creazione di MVP), strumenti come `Cursor` o `Qodo`, entrambi integrati in Visual Studio Code, offrono risposte rapide e una buona capacità di iterazione. Tuttavia, richiedono una verifica accurata e un contributo umano costante per il completamento dei requisiti funzionali. I difetti frequenti nei test e nell'interfaccia, come riscontrato nell'analisi qualitativa, richiedono un continuo intervento dell'utente.
Per applicazioni in cui è importante ridurre il numero di interventi manuali e garantire un risultato coerente al primo tentativo, `Copilot` e `Replit` risultano scelte più affidabili. In particolare, `Replit` offre un'esperienza completa pur essendo interamente web-based, semplificando l'accesso in contesti didattici e collaborativi, e ha dimostrato una qualità del codice superiore, con minori difetti rispetto agli altri IDE.
`Windsurf`, invece, propone una ricca integrazione di funzionalità all'interno di un fork di Visual Studio Code, con assistenza intelligente al completamento e debugging. Tuttavia, il numero elevato di correzioni necessarie fa emergere la necessità di una rifinitura manuale non trascurabile, soprattutto in fasi avanzate dello sviluppo. La sua ambizione di coprire ampie porzioni del ciclo di sviluppo non è ancora del tutto supportata da una solida qualità del codice, come evidenziato nell'analisi qualitativa.

1.4.2 Conclusioni

Nel complesso, l'esperimento conferma quanto discusso in letteratura: l'intelligenza artificiale, anche nei contesti più sofisticati, non è ancora in grado di sostituire completamente il ruolo dello sviluppatore; può però amplificarne la produttività, a condizione che vi sia un buon bilanciamento tra capacità generativa dell'agente e supervisione umana. L'integrazione tra metrica quantitativa (tempi e numero di correzioni) e analisi qualitativa (tipologia di errori) ha permesso di evidenziare i limiti strutturali dei vari IDE: alcuni trascurano sistematicamente aspetti come i test, altri faticano a generare interfacce coerenti, altri ancora falliscono in aspetti architetturali trasversali.
La scelta dello strumento più adatto non dovrebbe tener conto esclusivamente delle metriche di performance, ma anche del tipo di progetto, del momento specifico in cui ci si trova all'interno del ciclo di sviluppo e del livello di controllo che si desidera esercitare sul codice prodotto. In contesti a bassa criticità o in fasi preliminari, può essere vantaggioso adottare strumenti più automatizzati e rapidi, anche a costo di maggiore effort successivo. Al contrario, in progetti complessi o in fasi avanzate, risulta spesso preferibile selezionare ambienti che offrano maggiore trasparenza, controllo e coerenza strutturale, anche se ciò comporta tempi più lunghi o interazioni più frequenti.

Bibliografia

[1] Ki Reply. URL: https://www.reply.com/ki-reply/en.

[2] Outsystems. URL: https://www.outsystems.com/.

[3] Mendix. URL: https://www.mendix.com/.

[4] Creatio. URL: https://www.creatio.com/.

[5] OS Mentor. URL: https://www.outsystems.com/low-code-platform/mentor-ai-app-generation/.

[6] Claude 3.5 Sonnet. URL: https://claude.ai/.

[7] IT BREW. *Microsoft earnings show role of Copilot tool in revenue growth.* URL: https://www.itbrew.com/stories/2024/08/13/github-copilot-stumbles-as-microsoft-earnings-report-remains-favorable?utm_source=chatgpt.com.

[8] Priyan Vaithilingam, Tianyi Zhang e Elena L. Glassman. «Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models». In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: 10.1145/3491101.3519665. URL: https://doi.org/10.1145/3491101.3519665.

[9] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin e Edward Aftandilian. *Productivity Assessment of Neural Code Completion*. 2022. arXiv: 2205.06537 [cs.SE]. URL: https://arxiv.org/abs/2205.06537.

[10] Nhan Nguyen e Sarah Nadi. «An Empirical Evaluation of GitHub Copilot's Code Suggestions». In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 1–5. DOI: 10.1145/3524842.3528470.

[11] Windsurf by Codeium. URL: https://windsurf.com/.

[12] Windsurf Cascade. URL: https://windsurf.com/cascade.

[13] Windsurf Supercomplete. URL: https://windsurf.com/windsurf-tab.

[14] AIPure. *Windsurf.* URL: https://aipure.ai/it/products/windsurfing/features.

[15] Md Sultanul Islam Ovi, Nafisa Anjum, Tasmina Haque Bithe, Md. Mahabubur Rahman e Mst. Shahnaj Akter Smrity. *Benchmarking ChatGPT, Codeium, and GitHub Copilot: A Comparative Study of AI-Driven Programming and Debugging Assistants*. 2024. arXiv: 2409.19922 [cs.SE]. URL: https://arxiv.org/abs/2409.19922.

[16] Bijee Lakshman e Abhinav S. «Impact of AI Tools in Software Engineering: Boon or a Bane». In: *Journal of Software Engineering Tools & Technology Trends* 11.01 (2024), pp. 14–23.

[17] Valerio Terragni, Annie Vella, Partha Roop e Kelly Blincoe. «The Future of AI-Driven Software Engineering». In: *ACM Trans. Softw. Eng. Methodol.* (gen. 2025). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3715003. URL: https://doi.org/10.1145/3715003.

[18] Ariful Haque, Sunzida Siddique, Md. Mahfuzur Rahman, Ahmed Rafi Hasan, Laxmi Rani Das, Marufa Kamal, Tasnim Masura e Kishor Datta Gupta. *SOK: Exploring Hallucinations and Security Risks in AI-Assisted Software Development with Insights for LLM Deployment*. 2025. arXiv: 2502.18468 [cs.SE]. URL: https://arxiv.org/abs/2502.18468.

[19] Qodo (formerly Codium). URL: https://www.qodo.ai/.

[20] Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar e Eray Tüzün. *Automated Code Review In Practice*. 2024. arXiv: 2412.18531 [cs.SE]. URL: https://arxiv.org/abs/2412.18531.

[21] Noble Saji Mathews e Meiyappan Nagappan. *Design choices made by LLM-based test generators prevent them from finding bugs*. 2024. arXiv: 2412.14137 [cs.SE]. URL: https://arxiv.org/abs/2412.14137.

[22] Manus. URL: https://manus.im/.

[23] D.R. Adinath e I.S. Smiju. «Advancements in AI-Powered NLP Models: A Critical Analysis of Manus AI, Gemini, Grok AI, DeepSeek, and ChatGPT». In: *SSRN Electronic Journal* (mar. 2025). Available at SSRN: https://ssrn.com/abstract=5185131 or http://dx.doi.org/10.2139/ssrn.5185131.

[24] Douglas Youvan. «DeepSeek and China's AI Renaissance: The Rise of the Four Dragons and Six Tigers». In: (mar. 2025). DOI: 10.13140/RG.2.2.17393.80483.

[25] Forbes. *Dentro la startup da 400 milioni di dollari che sviluppa il software usato dagli ingegneri di OpenAI*. URL: https://forbes.it/2024/08/26/cursor-startup-sviluppa-software-usato-openai/.

[26] Paul Pajo. «A Step-by-Step Framework for AI-Assisted Frontend UI Development: 76 Atomic Steps for Consistency, Scalability, and Post-Deployment Iteration». In: (apr. 2025). DOI: 10.13140/RG.2.2.30076.65928.

[27] Replit. URL: https://replit.com/~.

[28] Replit. *Raising $97.4M at $1.16B Valuation to Expand our Cloud Services and Lead in AI Development*. 2023. URL: https://blog.replit.com/b-extension.

[29] Replit. *Turn natural language into code - Replit AI*. 2024. URL: https://replit.com/ai.

[30] Marta Andersson e Dan McIntyre. «Can ChatGPT recognize impoliteness? An exploratory study of the pragmatic awareness of a large language model». In: *Journal of Pragmatics* 239 (2025), pp. 16–36. ISSN: 0378-2166. DOI: https://doi.org/10.1016/j.pragma.2025.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0378216625000323.

[31] Ziqi Yin, Hao Wang, Kaito Horio, Daisuke Kawahara e Satoshi Sekine. «Should We Respect LLMs? A Cross-Lingual Study on the Influence of Prompt Politeness on LLM Performance». In: *Proceedings of the Second Workshop on Social Influence in Conversations (SICon 2024)*. A cura di James Hale, Kushal Chawla e Muskan Garg. Miami, Florida, USA: Association for Computational Linguistics, nov. 2024, pp. 9–35. DOI: 10.18653/v1/2024.sicon-1.2. URL: https://aclanthology.org/2024.sicon-1.2/.

[32] SonarQube. URL: https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/.

[33] Bandit. URL: https://bandit.readthedocs.io/en/latest/.

[34] Lighthouse. URL: https://developer.chrome.com/docs/lighthouse/overview?hl=it.