Vibe Coding: la nuova frontiera dello sviluppo software con l'AI

DI MASSIMO VAILATI
03/10/2025

VIBE CODING PROGRAMMAZIONE CON AI

Il **Vibe Coding** è un nuovo approccio allo **sviluppo software** che sfrutta i modelli linguistici per generare codice a partire da semplici prompt in linguaggio naturale. L'idea, lanciata da **Andrej Karpathy**, è quella di concentrarsi sull'intento e non sulla tecnica, delegando all'AI la scrittura e l'ottimizzazione del codice. Questo metodo può accelerare i progetti e aprire la programmazione anche a chi non conosce i linguaggi tradizionali, ma comporta rischi di manutenibilità e debito tecnico. In ambito scolastico, se usato con consapevolezza, offre spunti per attività pratiche, progetti interdisciplinari e sviluppo del pensiero critico.

Autori

MASSIMO VAILATI

Il **Vibe Coding** è una delle tendenze più affascinanti - e controverse - nel panorama attuale dello sviluppo software. Alimentato dalla rapida evoluzione dei **modelli linguistici di grandi dimensioni (LLM)**, come ChatGPT, Claude o Gemini, questo approccio rappresenta un **cambio di paradigma**: non si scrive più codice, si **parla con l'AI** per descrivere ciò che si vuole ottenere. E il codice lo scrive qualcun altro.

Il termine è stato introdotto nel febbraio 2025 da **Andrej Karpathy**, uno dei maggiori esperti mondiali nel campo dell'AI (ex OpenAI, Tesla, ora fondatore di Eureka Labs). Secondo lui, con il Vibe Coding si può arrivare a **"dimenticare che il codice esista"**, perché si lavora esclusivamente sull'intento, non sulla tecnica.

L'obiettivo è semplice: **ottenere un risultato** - un'applicazione, una funzione, un'interfaccia - descrivendolo in linguaggio naturale. È l'AI che interpreta le intenzioni, scrive il codice, lo modifica e lo ottimizza. L'utente si limita a "guidare" l'output attraverso i prompt.

Un altro esperto, **Simon Willison**, ha definito il Vibe Coding come "**costruire software senza rivedere il codice che l'LLM scrive**". Il concetto chiave è la **delegazione totale**: non c'è revisione, non c'è lettura del codice generato, non c'è analisi tecnica. Il focus è sull'**esperienza finale**, sul "vibe" dell'applicazione.

IL LINGUAGGIO NATURALE DIVENTA IL NUOVO CODICE

Secondo Karpathy, il "linguaggio di programmazione più potente oggi è l'**inglese**". O, più in generale, **la lingua che parliamo**. Questo significa che si può "programmare" con un prompt tipo:

"*Crea una web app che permetta di inserire una spesa, visualizzare un grafico mensile e salvare i dati in locale.*"

E l'AI produrrà una soluzione. Non sempre perfetta, ma funzionante. Se non ci convince, basta un altro prompt.

Tuttavia, l'approccio comporta rischi strutturali:

- Non si legge il codice generato.
- Non si conoscono le librerie usate.
- Non si capisce come funziona internamente il software.

E questo può diventare un problema serio.

Il Vibe Coding può generare **risultati impressionanti in pochi minuti**, ma crea anche una nuova forma di **debito tecnico**: si produce codice che nessuno conosce davvero, nemmeno chi l'ha "creato".

Nel gergo degli sviluppatori, questo codice è **"legacy"**: difficile da capire, modificare o estendere, come i vecchi sistemi aziendali privi di documentazione. E nel Vibe Coding questo rischio è moltiplicato, perché il codice nasce già opaco. Risultato? Il codice **funziona**, ma **non parla la lingua del progetto**.

Karpathy lo dice chiaramente: **l'AI è come uno stagista straordinario**. Lavora velocemente, propone soluzioni, ha fantasia. Ma non ha esperienza, né sensibilità progettuale. E soprattutto, **non capisce i vincoli reali di un team, un prodotto, una base di codice esistente**.

Ecco perché **il Vibe Coding funziona solo se supportato da competenze solide**:

- Serve **conoscere il dominio del problema**.
- Bisogna **leggere e correggere il codice generato**.
- È fondamentale **saper porre le giuste domande** all'AI e valutare le sue risposte con spirito critico.

VIBE CODING A SCUOLA: UNA RISORSA UTILE, SE USATA CON CONSAPEVOLEZZA

Pur con tutte le sue criticità, il Vibe Coding **può essere uno strumento prezioso anche in ambito didattico**, soprattutto nella scuola secondaria.
Utilizzando strumenti come ChatGPT o Claude, è possibile **progettare e sviluppare applicazioni più complesse**, anche senza conoscere perfettamente la sintassi di un linguaggio di programmazione. Questo permette di:

- **Coinvolgere studenti con background diversi** (artistico, umanistico, scientifico).
- Lavorare su **progetti interdisciplinari**.
- Concentrarsi su **competenze trasversali** come problem solving, comunicazione efficace e gestione dei requisiti.

Tuttavia, è fondamentale **educare gli studenti a un uso critico dell'AI**, spiegando:

- Che il codice va **letto e capito**.

Che un prompt va scritto con **precisione e contesto**.

Che **non basta ottenere un output funzionante**: serve anche qualità, coerenza e manutenibilità.

Il **Vibe Coding** è destinato a crescere, ma **non sostituirà mai il pensiero ingegneristico**. L'AI può essere un assistente, un acceleratore, un alleato straordinario. Ma come ogni tecnologia, **vale quanto la consapevolezza di chi la usa**.

Portarlo in classe è possibile - ed è auspicabile - **a patto che non si rinunci alla comprensione**. L'obiettivo della scuola non è ottenere un risultato perfetto al primo colpo, ma formare menti capaci di costruire, interpretare e migliorare. Anche - e soprattutto - in un mondo dove "scrivere codice" non significa più digitare ogni singola riga.

ATTIVITÀ PRATICA N.1

"Avoid the Blocks" - Gioco base generato via AI

Vogliamo realizzare un semplice gioco arcade dove il giocatore controlla un blocco (con le frecce sinistra/destra) e deve evitare ostacoli che cadono dall'alto. Il gioco diventa più veloce man mano che il tempo passa.
L'obiettivo: sopravvivere il più a lungo possibile.

Requisiti minimi del progetto

Interfaccia HTML semplice
Movimenti del giocatore con tastiera
Blocchi che cadono automaticamente
Collision detection
Punteggio crescente
Game over + restart

Utilizziamo ChatGPT e forniamo il seguente prompt iniziale:

> *"Crea un semplice gioco arcade in HTML e JavaScript. Il giocatore controlla un blocco alla base dello schermo con i tasti freccia sinistra/destra. Dall'alto cadono ostacoli. Se il blocco del giocatore tocca un ostacolo, il gioco termina. Mostra il punteggio e un pulsante per ricominciare."*

L'AI ci fornice la seguente struttura del progetto

```
/vibe-game/
├── index.html
├── style.css
└── script.js
```

e il codice dei tre file.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Avoid the Blocks</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
```

```
  <h1>Avoid the Blocks</h1>
  <div id="gameContainer">
    <div id="player"></div>
  </div>
  <p id="score">Score: 0</p>
  <button id="restartBtn" onclick="restartGame()">Restart</button>
  <script src="script.js"></script>
</body>
</html>
```

Analizziamo il codice prodotto. Esso comprende un `<div id="gameContainer">` che funge da area di gioco, una sorta di "canvas" HTML. All'interno, c'è un `<div id="player">` che rappresenta il blocco controllato dal giocatore.

Elementi di interfaccia:

- Un titolo `<h1>` con il nome del gioco.
- Un paragrafo `<p id="score">` che mostra il punteggio in tempo reale.
- Un pulsante `<button id="restartBtn">` che appare dopo la sconfitta e consente di ricominciare la partita.

Stilizzazione e logica:

- Il file esterno `style.css` definisce l'aspetto visivo del gioco (colori, dimensioni, layout).
- Il file `script.js` contiene tutta la **logica interattiva**: movimento del giocatore, generazione degli ostacoli, punteggio, rilevamento delle collisioni e restart.

Compatibilità e responsività:

- I metadati nel `<head>` (charset UTF-8, viewport) garantiscono che il gioco sia leggibile su tutti i dispositivi e ben visibile anche su mobile.

style.css

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
  background-color: #222;
  color: white;
}

#gameContainer {
  position: relative;
  width: 400px;
  height: 600px;
  margin: 20px auto;
  background-color: #333;
  overflow: hidden;
  border: 2px solid white;
}

#player {
  position: absolute;
  bottom: 10px;
  left: 180px;
```

```
  width: 40px;
  height: 40px;
  background-color: #00ffcc;
  border-radius: 5px;
}

.obstacle {
  position: absolute;
  width: 40px;
  height: 40px;
  background-color: red;
  top: 0;
  border-radius: 5px;
}

#restartBtn {
  padding: 10px 20px;
  font-size: 16px;
  display: none;
}
```

Gli stili scelti sono più comprensibili se si visualizza il gioco. Possiamo però notare i seguenti selettori:
`#gameContainer` che crea un contenitore per il gioco grande `400x600px`, centrato orizzontalmente con sfondo grigio scuro (`#333`) e bordo bianco. L'`overflow: hidden` impedisce che gli oggetti escano visivamente dal contenitore.
`#player` che costituisce il blocco del giocatore largo `40px` e alto `40px`, posizionato in basso al centro e di colore turchese brillante (`#00ffcc`) per evidenziarlo. L'attributo `border-radius: 5px` gli dà angoli leggermente arrotondati.
`.obstacle` che definisce gli ostacoli come blocchi rossi della stessa dimensione del giocatore. Appaiono in alto (`top: 0`) e cadono verso il basso via JavaScript e sono posizionati assolutamente per muoversi liberamente nel contenitore.
Riconoscere questi stili ci permette di personalizzare la grafica del gioco.

script.js

```
const player = document.getElementById("player");
const gameContainer = document.getElementById("gameContainer");
const scoreDisplay = document.getElementById("score");
const restartBtn = document.getElementById("restartBtn");

let gameInterval;
let obstacleInterval;
let score = 0;
let isGameOver = false;

function movePlayer(event) {
  if (isGameOver) return;
  const left = parseInt(window.getComputedStyle(player).getPropertyValue("left"));
  if (event.key === "ArrowLeft" && left > 0) {
    player.style.left = `${left - 40}px`;
  } else if (event.key === "ArrowRight" && left < 360) {
    player.style.left = `${left + 40}px`;
  }
}

function createObstacle() {
  const obstacle = document.createElement("div");
  obstacle.classList.add("obstacle");
```

```
  obstacle.style.left = `${Math.floor(Math.random() * 10) * 40}px`;
  gameContainer.appendChild(obstacle);

  let obstacleTop = 0;
  const fallSpeed = 5;

  const fall = setInterval(() => {
    if (isGameOver) {
      clearInterval(fall);
      return;
    }

    obstacleTop += fallSpeed;
    obstacle.style.top = `${obstacleTop}px`;

    const playerLeft = parseInt(player.style.left);
    const obstacleLeft = parseInt(obstacle.style.left);

    if (
      obstacleTop >= 550 &&
      obstacleLeft === playerLeft
    ) {
      endGame();
      clearInterval(fall);
    }

    if (obstacleTop > 600) {
      obstacle.remove();
      clearInterval(fall);
      score++;
      scoreDisplay.textContent = `Score: ${score}`;
    }
  }, 30);
}

function startGame() {
  document.addEventListener("keydown", movePlayer);
  score = 0;
  isGameOver = false;
  restartBtn.style.display = "none";
  scoreDisplay.textContent = "Score: 0";

  gameInterval = setInterval(() => {
    createObstacle();
  }, 800);
}

function endGame() {
  isGameOver = true;
  clearInterval(gameInterval);
  document.removeEventListener("keydown", movePlayer);
  restartBtn.style.display = "inline-block";
}

function restartGame() {
  // Remove old obstacles
  document.querySelectorAll(".obstacle").forEach(o => o.remove());
  player.style.left = "180px";
```

```
  startGame();
}

startGame();
```

Il codice JavaScript rappresenta la componente **più complessa e concettualmente rilevante**. A differenza di HTML e CSS, che definiscono rispettivamente la struttura e lo stile dell'interfaccia grafica, **JavaScript contiene il cuore logico del gioco**: è il motore che ne determina il comportamento dinamico.

È proprio qui che risiede la vera "intelligenza" del progetto, ed è in questa parte che l'intervento dell'IA, attraverso la metodologia del **vibe coding**, risulta più evidente e potente. Cerchiamo di capire il codice. Per questa fase possiamo ancora farci aiutare dall'IA chiedendo di commentare il funzionamento del codice che forniamo come prompt.

Inizializzazione degli elementi DOM
Le prime righe recuperano gli elementi chiave dalla pagina HTML: il blocco del giocatore, il contenitore del gioco, il contatore del punteggio e il pulsante di restart.

Variabili globali:
`gameInterval` e `obstacleInterval`: usati per gestire i timer.
`score`: memorizza il punteggio attuale.
`isGameOver`: flag per sapere se il gioco è terminato.

Funzioni:

`movePlayer(event)`
Gestisce il movimento del giocatore: si attiva premendo i tasti freccia sinistra/destra, sposta il giocatore di 40 pixel per volta (larghezza del blocco). Il movimento è vincolato ai limiti del contenitore (tra 0 e 360 px). Se il gioco è finito (`isGameOver === true`), ignora l'input.

`createObstacle()`
Creazione e caduta degli ostacoli: crea un nuovo blocco (`div.obstacle`) posizionato casualmente in una delle 10 colonne (10 × 40px = 400px di larghezza), fa cadere l'ostacolo incrementando il valore della sua proprietà top ogni 30 millisecondi. Se un ostacolo raggiunge il giocatore chiama `endGame()`. Se l'ostacolo supera il bordo inferiore (600px), viene rimosso e il punteggio aumenta.

`startGame()`
Avvia il gioco: azzera punteggio e stato di fine gioco, nasconde il pulsante di restart., assegna un timer (`setInterval`) per generare nuovi ostacoli ogni 800 ms, ascolta gli eventi da tastiera per controllare il giocatore.

`endGame()`
Gestisce la fine del gioco: imposta `isGameOver = true` per bloccare ulteriori azioni, ferma la generazione di nuovi ostacoli, disattiva il controllo da tastiera e mostra il pulsante per ricominciare.

`restartGame()`
Fa ripartire il gioco: rimuove tutti gli ostacoli attualmente visibili, riposiziona il giocatore al centro e richiama `startGame()` per far ripartire il ciclo di gioco da zero.

Possiamo ora eseguire il gioco.

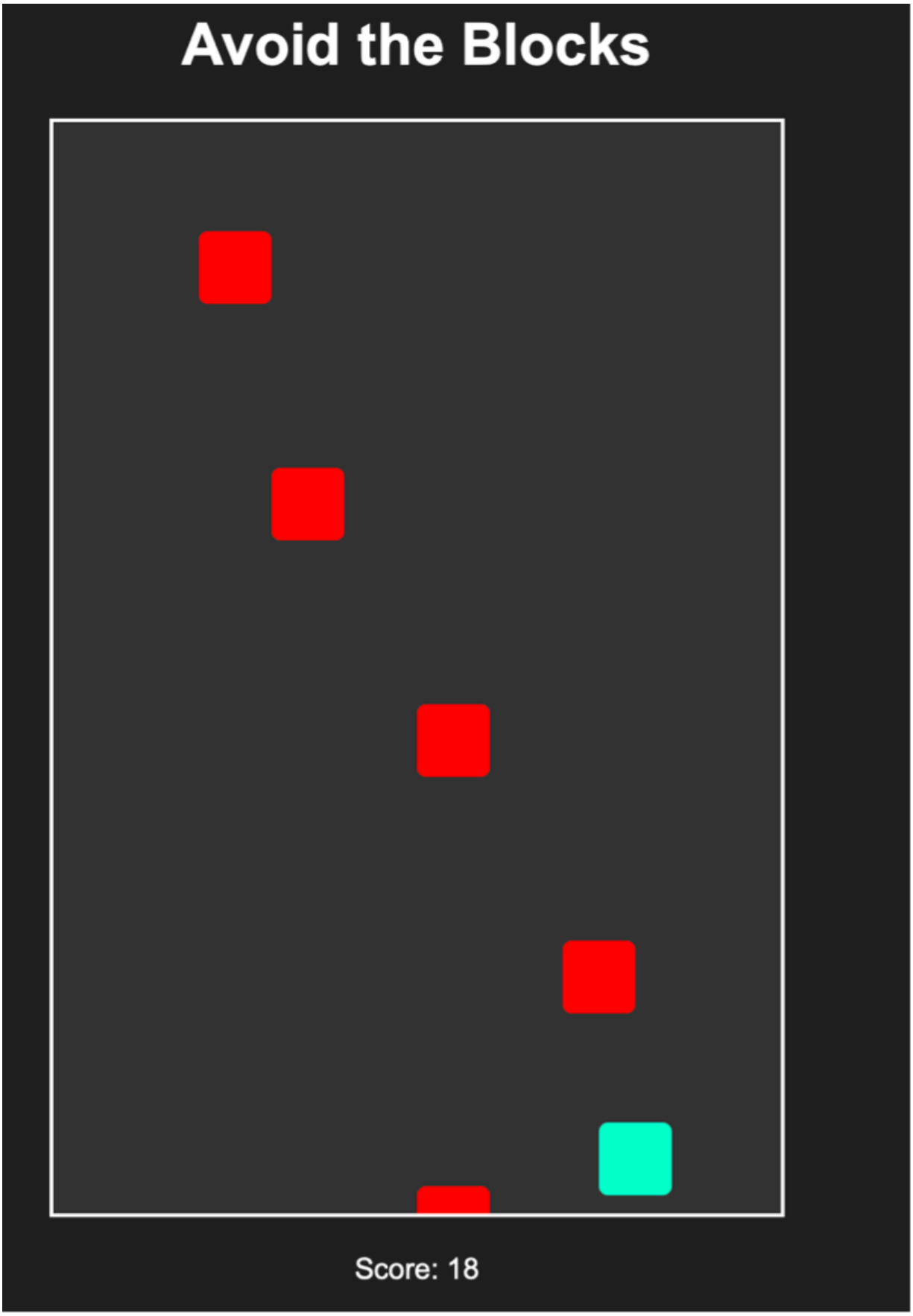

Fig.1

Il risultato è gradevole e sembra funzionare correttamente ma appena il giocatore viene colpito da un blocco il gioco non termina. Sembra che non vengano gestire correttamente le collisioni. Diamo un nuovo prompt a ChatGPT:

"non viene gestita la collisione e il gioco non termina quando un blocco colpisce il giocatore"

La risposta dell'IA è la seguente:

> *Hai perfettamente ragione. Il controllo delle collisioni nel codice attuale è troppo semplificato: confronta solo la* ***posizione orizzontale*** *del blocco (`left`) e un valore fisso verticale (`obstacleTop >= 550`), ma* ***non verifica realmente l'intersezione tra i rettangoli del giocatore e dell'ostacolo****.*
> *Ti propongo qui una versione* ***corretta*** *con un* ***vero controllo di collisione rettangolare*** *(bounding box), che termina il gioco quando un ostacolo tocca il giocatore.*
> *Sostituisci il blocco* `fall` *nel file* `script.js` *con questo codice migliorato:*

```
const fall = setInterval(() => {
 if (isGameOver) {
  clearInterval(fall);
  return;
 }

 obstacleTop += fallSpeed;
 obstacle.style.top = `${obstacleTop}px`;

 const playerRect = player.getBoundingClientRect();
 const obstacleRect = obstacle.getBoundingClientRect();

 // COLLISION DETECTION
 if (
  obstacleRect.bottom > playerRect.top &&
  obstacleRect.top < playerRect.bottom &&
  obstacleRect.left < playerRect.right &&
  obstacleRect.right > playerRect.left
 ) {
  endGame();
  clearInterval(fall);
  return;
 }

 if (obstacleTop > 600) {
  obstacle.remove();
  clearInterval(fall);
  score++;
  scoreDisplay.textContent = `Score: ${score}`;
 }
}, 30);
```

Questa versione utilizza le **bounding box reali** con `getBoundingClientRect()` che restituisce le coordinate effettive sullo schermo. La condizione di collisione controlla se i due rettangoli **si sovrappongono** e in tal caso richiama `endGame()`.
Ora il gioco funziona perfettamente. Continuando di questo passo possiamo aggiungere (e controllare) altre funzionalità con prompt successivi come:

> *"Aggiungi musica di sottofondo e suono quando il gioco finisce"*
> *"Rendi il gioco più difficile nel tempo aumentando la velocità dei blocchi"*
> *"Mostra una schermata Game Over animata"*
> *"Aggiungi leaderboard con punteggi migliori usando localStorage"*

ATTIVITÀ PRATICA N.2

Integrazione di nuove funzionalità con il Vibe Coding

Si propone agli studenti di **selezionare un progetto precedentemente realizzato** (oppure un esercizio complesso svolto in classe) e di utilizzarlo come base per introdurre **una nuova funzionalità**.
L'attività consiste nel:

1. **Formulare una richiesta chiara a un LLM (es. ChatGPT)**, descrivendo in linguaggio naturale la nuova funzionalità da aggiungere.
2. **Specificare, se necessario, vincoli tecnici** come:

   Costrutti da utilizzare (es. strutture condizionali, cicli, funzioni, classi...).
   Librerie preferenziali o già presenti nel progetto.
   Requisiti di compatibilità con l'architettura esistente.

3. **Analizzare il codice generato dall'AI** per valutarne:

   La correttezza funzionale.
   La coerenza stilistica e architetturale con il progetto originale.
   L'eventuale necessità di modifiche manuali o ottimizzazioni.

Competenze attivate:

   Scrittura di prompt efficaci e mirati.
   Lettura e comprensione del codice generato.
   Capacità di integrazione e refactoring.
   Valutazione critica dell'apporto dell'AI nello sviluppo software.

Suggerimento didattico:
Può essere utile **confrontare le soluzioni generate da diversi studenti** o gruppi per discutere collettivamente le scelte fatte, le differenze emerse e le strategie più efficaci per guidare l'AI verso output coerenti e sostenibili.