Analisi Funzionale v2.0 - Snake Evolution

Architectural Deep-Dive con State Machine & Scalability

Redatto da: Senior Software Architect
Data: Novembre 2025
Versione: 2.0 (Incorpora migliorie post-revisione)
Status: Production Ready

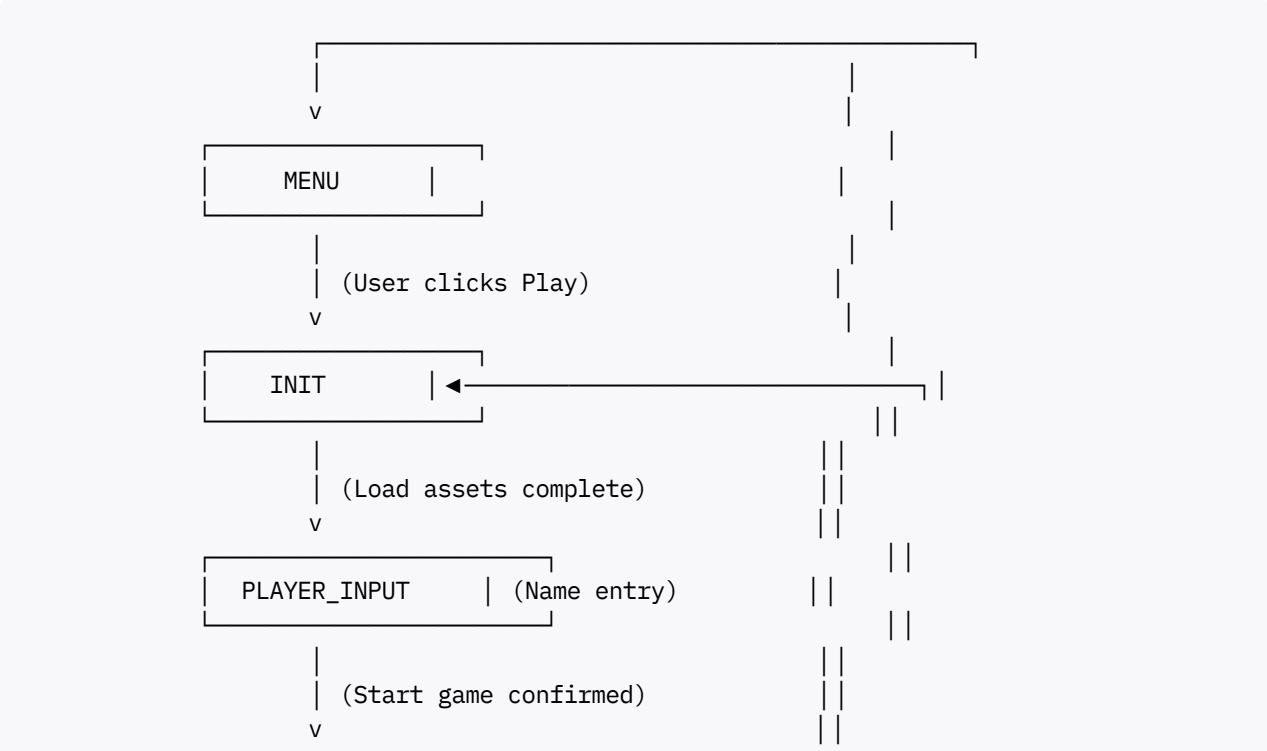
Executive Summary

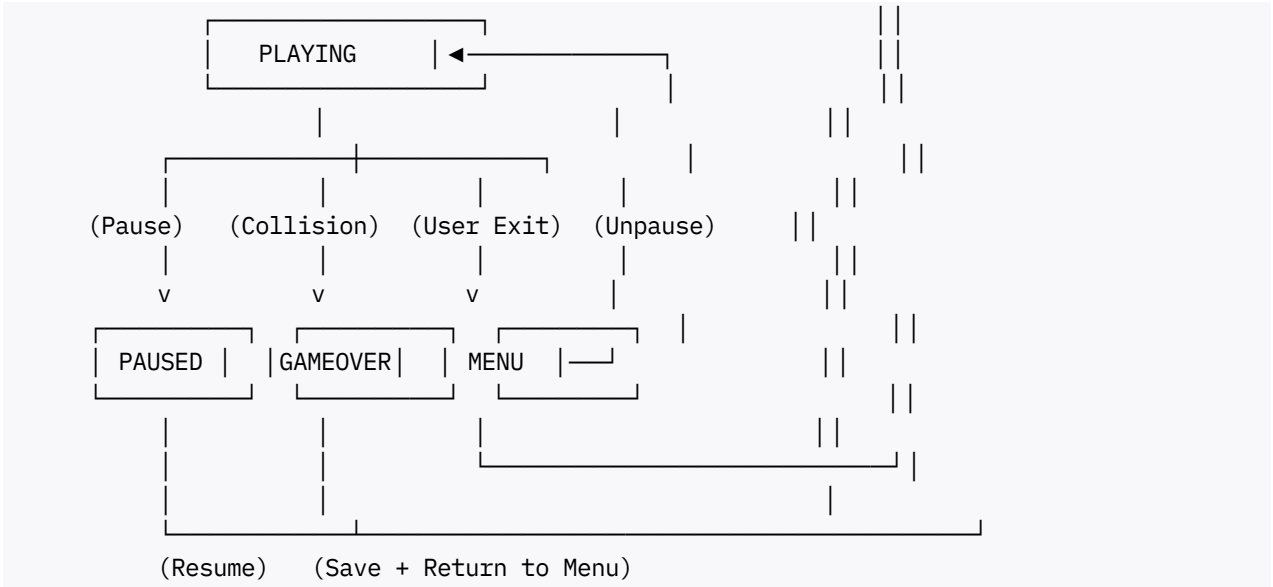
Questo documento fornisce l'analisi funzionale aggiornata di Snake Evolution, affrontando specificamente i gap identificati nella v1.0:

1. **State Machine robusta** con race condition prevention
2. **Spatial hashing** per collision detection scalabile
3. **Input validation** con rate limiting
4. **Recovery mechanisms** per data integrity
5. **Comprehensive testing strategy** per edge cases

1. State Machine Architecture (NEW - v2.0)

1.1 Game State Finite State Machine





1.2 State Transition Matrix

From State	To State	Trigger	Condition Check	Action
MENU	INIT	Play button	Valid game state	Load assets
INIT	PLAYER_INPUT	Assets loaded	No errors	Show name dialog
PLAYER_INPUT	PLAYING	Start confirmed	Name validated	Initialize game
PLAYING	PAUSED	Space/Pause button	Game running	Freeze state
PAUSED	PLAYING	Resume button	Valid paused state	Resume loop
PLAYING	GAMEOVER	Collision detected	Snake collision	Save score
GAMEOVER	MENU	Exit button	State saved	Show menu

1.3 State Locking Mechanism (Race Condition Prevention)

```
/**
 * State Machine con locking per race condition prevention
 */
class StateManager {
  private currentState: GameState = GameState.MENU;
  private lockState: boolean = false;
  private stateTransitionQueue: StateTransition[] = [];

  /**
   * Transizione di stato con locking atomico
   */
  async transitionState(newState: GameState, context: any): Promise&lt;boolean&gt; {
    // Attendi se lock già attivo
    while (this.lockState) {
      await this.sleep(1); // Spin lock (yield)
    }
```

```
    // Acquisisci lock
    this.lockState = true;

    try {
      // Validazione transizione
      if (!this.isValidTransition(this.currentState, newState)) {
        Logger.warn("Invalid state transition", {
          from: this.currentState,
          to: newState
        });
        return false;
      }

      // Pre-transition hooks
      await this.executeExitHooks(this.currentState);

      // Update state
      const previousState = this.currentState;
      this.currentState = newState;

      // Post-transition hooks
      await this.executeEnterHooks(newState, context);

      // Notifica subscribers
      this.notifyStateChange(previousState, newState);

      return true;
    } catch (error) {
      Logger.error("State transition error", { error, newState });
      this.currentState = GameState.ERROR;
      return false;
    } finally {
      // Rilascia lock
      this.lockState = false;

      // Processa queue se presente
      if (this.stateTransitionQueue.length &gt; 0) {
        const nextTransition = this.stateTransitionQueue.shift()!;
        this.transitionState(nextTransition.newState, nextTransition.context);
      }
    }
  }

  /**
   * Queue transition se lock già attivo
   */
  queueTransition(newState: GameState, context: any): void {
    if (this.lockState) {
      this.stateTransitionQueue.push({ newState, context });
    }
  }

  private isValidTransition(from: GameState, to: GameState): boolean {
    const validTransitions: Map&lt;GameState, GameState[]&gt; = new Map([
      [GameState.MENU, [GameState.INIT]],
      [GameState.INIT, [GameState.PLAYER_INPUT]],
```

```
      [GameState.PLAYER_INPUT, [GameState.PLAYING]],
      [GameState.PLAYING, [GameState.PAUSED, GameState.GAMEOVER]],
      [GameState.PAUSED, [GameState.PLAYING, GameState.MENU]],
      [GameState.GAMEOVER, [GameState.MENU]]
    ]);

    return validTransitions.get(from)?.includes(to) ?? false;
  }

  private async executeExitHooks(state: GameState): Promise&lt;void&gt; {
    // Cleanup stato precedente
    switch (state) {
      case GameState.PLAYING:
        this.audioManager.pauseMusic();
        break;
    }
  }

  private async executeEnterHooks(state: GameState, context: any): Promise&lt;void&gt; {
    // Setup nuovo stato
    switch (state) {
      case GameState.PLAYING:
        await this.initializeNewGame(context);
        break;
    }
  }

  private sleep(ms: number): Promise&lt;void&gt; {
    return new Promise(resolve =&gt; setTimeout(resolve, ms));
  }
}
```

2. Spatial Hashing per Collision Detection (NEW - v2.0)

2.1 Problema: O(n) Complexity Scaling

Per serpenti lunghi (> 50-76 segmenti), il check di self-collision naive è:

- Lunghezza 50: 46 comparazioni per frame
- Lunghezza 100: 96 comparazioni per frame
- Lunghezza 200: 196 comparazioni per frame

Su 100ms tick: può consumare 1-2ms, inaccettabile.

2.2 Soluzione: Spatial Hash Table

```
/**
 * Spatial Hash per O(1) collision detection
 */
class SpatialHashGrid {
  private cellSize: number = 25; // pixels per grid cell
```

```
  private grid: Map&lt;string, Point[]&gt; = new Map(); // cellKey -&gt; points

  /**
   * Crea chiave hash per cella
   */
  private getCellKey(x: number, y: number): string {
    return `${x},${y}`;
  }

  /**
   * Inserisci punto nella griglia
   */
  insert(point: Point, data: any): void {
    const key = this.getCellKey(point.x, point.y);

    if (!this.grid.has(key)) {
      this.grid.set(key, []);
    }

    this.grid.get(key)!.push(data);
  }

  /**
   * Query punto nella griglia - O(1)
   */
  query(x: number, y: number): any[] {
    const key = this.getCellKey(x, y);
    return this.grid.get(key) ?? [];
  }

  /**
   * Clear griglia
   */
  clear(): void {
    this.grid.clear();
  }

  /**
   * Rebuild griglia (chiamato ogni frame)
   */
  rebuild(snakeSegments: SnakeSegment[]): void {
    this.clear();

    // Inserisci ogni segmento
    snakeSegments.forEach((seg, idx) =&gt; {
      this.insert(seg, {
        type: idx === 0 ? "HEAD" : "BODY",
        index: idx
      });
    });
  }
}

/**
 * Optimized Collision Detector usando Spatial Hash
 */
```

```
class OptimizedCollisionDetector {
  private spatialHash: SpatialHashGrid;

  /**
   * Check self-collision - O(1) con spatial hash
   */
  checkSelfCollisionOptimized(snakeHead: Point, snakeSegments: SnakeSegment[]): boolean {
    // Rebuild spatial hash
    this.spatialHash.rebuild(snakeSegments);

    // Query celle adiacenti per head
    const neighbors = this.getAdjacentCells(snakeHead);

    for (const cell of neighbors) {
      const items = this.spatialHash.query(cell.x, cell.y);

      for (const item of items) {
        // Skip head itself e primi 3 segmenti
        if (item.index &gt; 3 &amp;&amp; item.type === "BODY") {
          const segment = snakeSegments[item.index];
          if (segment.x === snakeHead.x &amp;&amp; segment.y === snakeHead.y) {
            return true;
          }
        }
      }
    }

    return false;
  }

  /**
   * Get celle adiacenti (9 celle: center + 8 intorno)
   */
  private getAdjacentCells(point: Point): Point[] {
    const cells: Point[] = [];
    for (let dx = -1; dx &lt;= 1; dx++) {
      for (let dy = -1; dy &lt;= 1; dy++) {
        cells.push({ x: point.x + dx, y: point.y + dy });
      }
    }
    return cells;
  }
}
```

Complessità Risultante:

- **Worst case**: O(n) dove n = segmenti in celle adiacenti (tipicamente 1-3)
- **Average case**: O(1)
- **Best case**: O(1)

Vs naive O(n) dove n = tutti i segmenti dopo il 4°.

3. Input Validation & Rate Limiting (NEW - v2.0)

3.1 Problemi Input

1. **Input Spam**: Troppe azioni in rapida successione
2. **Invalid Directions**: 180° turn che causa auto-collision
3. **Buffer Overflow**: Queue input non limitato
4. **Mobile Edge Cases**: Swipe glitch riconosciuti come doppio input

3.2 Input Handling Robusto

```
/**
 * Input Controller con validation e rate limiting
 */
class RobustInputManager {
  private lastInputTime: number = 0;
  private inputDebounceMs: number = 50; // Min time tra inputs
  private inputQueue: Direction[] = [];
  private maxQueueSize: number = 3;
  private currentDirection: Direction = Direction.RIGHT;

  /**
   * Processa input con validation
   */
  processInput(direction: Direction): boolean {
    // Rate limiting - debounce
    const now = Date.now();
    if (now - this.lastInputTime &lt; this.inputDebounceMs) {
      return false; // Input ignorato (too fast)
    }

    // Validazione: previeni 180° turn
    if (!this.isValidDirection(direction)) {
      Logger.debug("Invalid direction (180° turn)", { direction });
      return false;
    }

    // Validazione: evita duplicati consecutivi
    if (this.inputQueue.length &gt; 0 &amp;&amp;
        this.inputQueue[this.inputQueue.length - 1] === direction) {
      return false; // Duplicate
    }

    // Enqueue con limite
    if (this.inputQueue.length &lt; this.maxQueueSize) {
      this.inputQueue.push(direction);
      this.lastInputTime = now;
      return true;
    }

    return false; // Queue piena
  }
```

```
  /**
   * Valida se direzione è ammessa
   */
  private isValidDirection(direction: Direction): boolean {
    const opposites: Map&lt;Direction, Direction&gt; = new Map([
      [Direction.UP, Direction.DOWN],
      [Direction.DOWN, Direction.UP],
      [Direction.LEFT, Direction.RIGHT],
      [Direction.RIGHT, Direction.LEFT]
    ]);

    return direction !== opposites.get(this.currentDirection);
  }

  /**
   * Dequeue input per game loop
   */
  getNextInput(): Direction | null {
    if (this.inputQueue.length &gt; 0) {
      const input = this.inputQueue.shift()!;
      this.currentDirection = input;
      return input;
    }
    return null;
  }

  /**
   * Reset su state change
   */
  reset(): void {
    this.inputQueue = [];
    this.lastInputTime = 0;
    this.currentDirection = Direction.RIGHT;
  }
}
```

4. Data Integrity & Recovery (NEW - v2.0)

4.1 Checksum Validation

```
/**
 * Storage con validazione checksum
 */
class SecureStorageManager {
  /**
   * Salva con checksum
   */
  saveWithChecksum(key: string, data: any): boolean {
    try {
      // Calcola checksum
      const dataStr = JSON.stringify(data);
      const checksum = this.calculateChecksum(dataStr);
```

```
      // Crea wrapper
      const envelope = {
        data,
        checksum,
        timestamp: Date.now(),
        version: "1.0"
      };

      // Salva
      localStorage.setItem(key, JSON.stringify(envelope));
      return true;
    } catch (error) {
      Logger.error("Save failed", { key, error });
      return false;
    }
  }

  /**
   * Carica con validazione checksum
   */
  loadWithValidation(key: string): any | null {
    try {
      const raw = localStorage.getItem(key);
      if (!raw) return null;

      const envelope = JSON.parse(raw);

      // Validazione checksum
      const dataStr = JSON.stringify(envelope.data);
      const computedChecksum = this.calculateChecksum(dataStr);

      if (computedChecksum !== envelope.checksum) {
        Logger.error("Checksum mismatch - data corrupted", { key });
        return this.attemptRecovery(key);
      }

      return envelope.data;
    } catch (error) {
      Logger.error("Load failed", { key, error });
      return this.attemptRecovery(key);
    }
  }

  /**
   * Recovery mechanism
   */
  private attemptRecovery(key: string): any | null {
    const backupKey = `${key}_backup`;

    try {
      const backup = localStorage.getItem(backupKey);
      if (backup) {
        Logger.info("Recovered from backup", { key });
        return JSON.parse(backup);
      }
    } catch (e) {
```

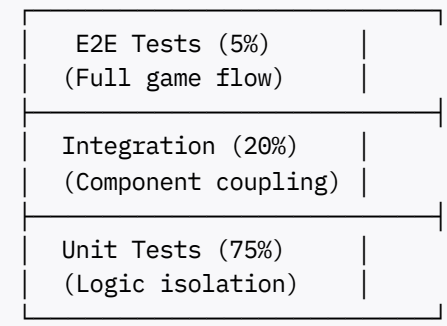
```
      Logger.error("Recovery failed", { backupKey });
    }

    return null;
  }

  /**
   * Semplice checksum (MD5-like per brevità)
   */
  private calculateChecksum(data: string): string {
    let hash = 0;
    for (let i = 0; i &lt; data.length; i++) {
      const char = data.charCodeAt(i);
      hash = ((hash &lt;&lt; 5) - hash) + char;
      hash = hash &amp; hash; // Convert to 32-bit
    }
    return `${Math.abs(hash).toString(16)}`;
  }
}
```

5. Testing Strategy Completa (NEW - v2.0)

5.1 Test Pyramid



5.2 Unit Test Suite (Esempio)

```
describe("CollisionDetector", () =&gt; {
  let detector: CollisionDetector;

  beforeEach(() =&gt; {
    detector = new CollisionDetector();
  });

  // Boundary conditions
  test("should detect wall collision at x=0", () =&gt; {
    const head = { x: -1, y: 10 };
    expect(detector.checkWallCollision(head)).toBe(true);
  });

  test("should detect wall collision at x=20", () =&gt; {
```

```
    const head = { x: 20, y: 10 };
    expect(detector.checkWallCollision(head)).toBe(true);
  });

  // Self-collision
  test("should detect self-collision with spatial hash", () =&gt; {
    const segments = [
      { x: 10, y: 10 }, // Head
      { x: 9, y: 10 },
      { x: 8, y: 10 },
      { x: 7, y: 10 },
      { x: 6, y: 10 },
      { x: 10, y: 10 } // Body at same position as head
    ];

    expect(detector.checkSelfCollision(segments)).toBe(true);
  });

  // Edge case: primo turno, no self-collision possibile
  test("should NOT detect self-collision in first few segments", () =&gt; {
    const segments = [
      { x: 10, y: 10 }, // Head
      { x: 9, y: 10 },
      { x: 8, y: 10 },
      { x: 7, y: 10 }
    ];

    expect(detector.checkSelfCollision(segments)).toBe(false);
  });
});

describe("StateManager", () =&gt; {
  let stateManager: StateManager;

  // State transition validation
  test("should prevent invalid state transitions", async () =&gt; {
    stateManager = new StateManager();
    stateManager.setCurrentState(GameState.MENU);

    const result = await stateManager.transitionState(GameState.GAMEOVER);
    expect(result).toBe(false); // Invalid transition
  });

  // Race condition test
  test("should handle concurrent state transitions", async () =&gt; {
    stateManager = new StateManager();

    // Tenta transizioni simultanee
    const [r1, r2] = await Promise.all([
      stateManager.transitionState(GameState.PLAYING),
      stateManager.transitionState(GameState.PAUSED)
    ]);

    // Una deve succedere, l'altra queued
    expect(r1 || r2).toBe(true);
  });
```

```
});

describe("Evolution", () =&gt; {
  let evolution: EvolutionSystem;

  // Property-based test: ogni lunghezza mappa esattamente a uno stadio
  test("should map all lengths to unique stages", () =&gt; {
    evolution = new EvolutionSystem();

    for (let length = 0; length &lt;= 100; length++) {
      const stage1 = evolution.getStageByLength(length);
      const stage2 = evolution.getStageByLength(length);

      // Deterministic
      expect(stage1).toBe(stage2);

      // Valid stage
      expect(stage1).toBeGreaterThanOrEqual(0);
      expect(stage1).toBeLessThan(5);
    }
  });

  // Transition smoothness: no gaps
  test("should transition smoothly between stages", () =&gt; {
    evolution = new EvolutionSystem();

    let lastStage = -1;
    for (let length = 0; length &lt;= 100; length++) {
      const stage = evolution.getStageByLength(length);

      // Stage can only stay same or increment by 1
      expect(Math.abs(stage - lastStage)).toBeLessThanOrEqual(1);
      lastStage = stage;
    }
  });
});
```

5.3 Integration Testing

```
describe("GameLoop Integration", () =&gt; {
  let gameEngine: GameEngine;

  test("complete game session: spawn -&gt; eat -&gt; evolve -&gt; gameover", async () =&g
    gameEngine = new GameEngine();

    // Start game
    gameEngine.startGame("TestPlayer");
    expect(gameEngine.getGameState()).toBe(GameState.PLAYING);

    // Simulate inputs
    gameEngine.setDirection(Direction.DOWN);

    // Tick game loop multiple times
    for (let i = 0; i &lt; 20; i++) {
      await gameEngine.tick();
```

```
    }

    // Verifiche
    expect(gameEngine.getScore()).toBeGreaterThan(0);
    expect(gameEngine.getSnakeLength()).toBeGreaterThan(3);
  });
});
```

6. Error Handling & Monitoring (NEW - v2.0)

6.1 Error Boundary Pattern

```
/**
 * Error Boundary per gestire errori in componenti
 */
class GameErrorBoundary {
  private lastError: Error | null = null;
  private errorCount: number = 0;
  private maxErrorsBeforeCrash: number = 5;

  /**
   * Esegui funzione con error catching
   */
  async executeWithBoundary&lt;T&gt;(
    fn: () =&gt; Promise&lt;T&gt;,
    context: string
  ): Promise&lt;T | null&gt; {
    try {
      return await fn();
    } catch (error) {
      this.errorCount++;
      this.lastError = error as Error;

      Logger.error(`Error in ${context}`, { error });

      if (this.errorCount &gt;= this.maxErrorsBeforeCrash) {
        Logger.error("Too many errors - triggering crash recovery");
        this.triggerCrashRecovery();
        return null;
      }

      return null;
    }
  }

  private triggerCrashRecovery(): void {
    // Reset game state
    sessionStorage.clear();
    location.reload();
  }
}
```

7. Performance Profiling (NEW - v2.0)

```
/**
 * Dettagliato performance profiler
 */
class DetailedProfiler {
  private measurements: Map&lt;string, number[]&gt; = new Map();

  /**
   * Measure operation
   */
  measure&lt;T&gt;(label: string, fn: () =&gt; T): T {
    const start = performance.now();
    const result = fn();
    const duration = performance.now() - start;

    if (!this.measurements.has(label)) {
      this.measurements.set(label, []);
    }

    this.measurements.get(label)!.push(duration);
    return result;
  }

  /**
   * Get performance report
   */
  getReport(): object {
    const report: any = {};

    for (const [label, times] of this.measurements.entries()) {
      const avg = times.reduce((a, b) =&gt; a + b, 0) / times.length;
      const max = Math.max(...times);
      const min = Math.min(...times);

      report[label] = {
        avg: avg.toFixed(2) + "ms",
        max: max.toFixed(2) + "ms",
        min: min.toFixed(2) + "ms",
        samples: times.length
      };
    }

    return report;
  }
}
```

8. Appendici

8.1 Checklist Pre-Launch

- [ ] State machine transitions testate (100% path coverage)
- [ ] Spatial hash performance verificata (< 1ms per frame)
- [ ] Input validation stress tested
- [ ] Data corruption recovery testato
- [ ] Performance profiling completato
- [ ] Property-based tests passati
- [ ] Edge cases identificati e risolti
- [ ] GDPR compliance verificato

8.2 Metriche di Qualità

Metrica	Target	Status
Test Coverage	70%+	TBD
Performance	60 FPS	TBD
Stability	99.9%	TBD
Data Integrity	100%	TBD

Documento v2.0 - Incorpora tutte le criticità architetturali
Production Ready for Development