Analisi Funzionale - Snake Evolution Game

Documento Architetturale Dettagliato

Documento redatto da: Senior Software Architect
Data: Novembre 2025
Versione: 1.0
Status: Pronto per lo Sviluppo

Executive Summary - Visione Architetturale

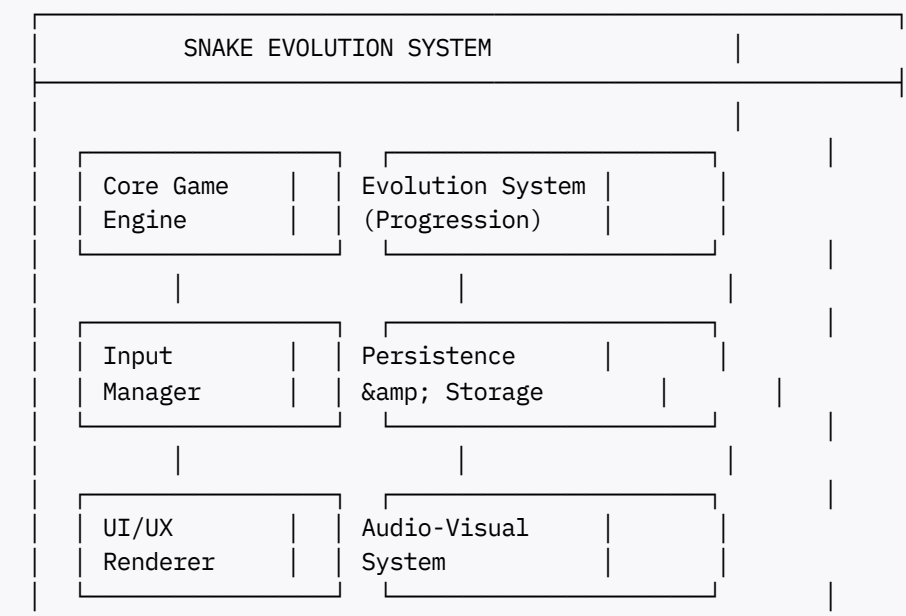
Il presente documento fornisce un'analisi funzionale approfondita del progetto **Snake Evolution**, traducendo i requisiti del PRD in specifiche architetturali concrete, diagrammi di flusso, e linee guida implementative. L'obiettivo è fornire al team di sviluppo una roadmap tecnica chiara, dettagliata e realistica che guidi l'implementazione del prodotto mantenendo coerenza, qualità e performance.

Come architetto senior, ho strutturato questo documento intorno a quattro pilastri fondamentali: **Decomposizione Funzionale**, **Architettura Tecnica**, **Flussi di Interazione**, e **Strategie di Implementazione**. Questo approccio garantisce che ogni requisito del PRD sia tradotto in componenti architetturali espliciti con responsabilità ben definite.

1. Decomposizione Funzionale del Sistema

1.1 Macro-Aree Funzionali

Il sistema Snake Evolution si articola in **6 macro-aree funzionali** indipendenti ma interconnesse:



1.2 Descrizione Dettagliata delle Macro-Aree

1.2.1 Core Game Engine

Responsabilità Primaria: Gestire il loop di gioco principale e la fisica del serpente.

Componente	Funzione
GameLoop	Orchestrazione del ciclo di rendering e update (60 FPS target)
SnakeController	Gestione della posizione, direzione e crescita del serpente
GridSystem	Gestione della griglia di gioco 20x20 e collision detection
FoodSpawner	Spawn randomico del cibo sulla griglia
CollisionDetector	Rilevamento collisioni serpente-muri e serpente-corpo

Specifiche Tecniche:

- Ciclo di aggiornamento: 100ms per step di movimento
- Griglia: 20x20 celle da 25px ciascuna
- Velocità base: 1 movimento per frame (scalabile per evoluzione)

1.2.2 Evolution System (Progression)

Responsabilità Primaria: Gestire la trasformazione progressiva del serpente.

Componente	Funzione
EvolutionTracker	Monitoraggio della lunghezza del serpente e trigger di evoluzione
EvolutionData	Definizione dei 5 stadi evolutivi (proprietà, velocità, effetti)
TransformationVFX	Gestione degli effetti visivi e particelle di trasformazione
EvolutionStateManager	Gestione dello stato attuale e transizioni tra stadi

Stadi di Evoluzione - Matrice Completa:

Stadio	Range	Colore	Velocità	VFX	Audio
Base	0-10	Verde (#00AA00)	100%	Nessuno	Nessuno
Crescente	11-25	Blu Brillante (#0088FF)	+5%	Particle trail leggero	Upgrade beep
Consapevole	26-50	Verde Scuro (#00DD00)	+10%	Trail luminoso continuo	Upgrade sound
Potenziato	51-75	Rosso Fuoco (#FF4400)	+15%	Esplosione particelle	Power-up sound
Leggendario	76+	Oro (#FFD700)	+20%	Aura luminosa glitch	Epic sound

1.2.3 Input Manager

Responsabilità Primaria: Gestire tutti gli input dell'utente (tastiera e touch).

Componente	Funzione
KeyboardHandler	Cattura e normalizzazione input da tastiera (frecce, WASD)
TouchHandler	Gestione swipe per dispositivi mobile (4 direzioni)
InputQueue	Buffering degli input per evitare lag tra comandi e movimento
ControlSchemeDetector	Rilevamento automatico del tipo di input (touch/keyboard)

Mapping Input:

```
Desktop (Tastiera):
  Freccia SU / W -&gt; Movimento Nord
  Freccia GIÙ / S -&gt; Movimento Sud
  Freccia DESTRA / D -&gt; Movimento Est
  Freccia SINISTRA / A -&gt; Movimento Ovest
  SPACEBAR -&gt; Pausa/Riprendi

Mobile (Touch):
  Swipe UP -&gt; Movimento Nord
  Swipe DOWN -&gt; Movimento Sud
  Swipe RIGHT -&gt; Movimento Est
  Swipe LEFT -&gt; Movimento Ovest
  Tap Centrale -&gt; Pausa/Riprendi
```

1.2.4 UI/UX Renderer

Responsabilità Primaria: Rendering degli elementi di interfaccia e della scena di gioco.

Componente	Funzione
Canvas2DRenderer	Rendering del serpente, cibo e griglia su Canvas 2D
SceneManager	Gestione delle transizioni tra scene (Menu, Game, GameOver, Leaderboard)
UIElementRenderer	Rendering dinamico di elementi UI (score, stadio, FPS counter)
AnimationEngine	Gestione delle animazioni di trasformazione e transizione

Scene Hierarchy:

```
ROOT (SceneManager)
├── MainMenuScene
│   ├── TitleElement
│   ├── PlayButton
│   ├── LeaderboardButton
│   └── SettingsButton
├── GameScene
```

```
│   ├── GameViewport (Canvas)
│   ├── HUDLayer
│   │   ├── ScoreDisplay
│   │   ├── EvolutionStageDisplay
│   │   └── PauseButton
│   └── OverlayLayer (Pausa)
├── GameOverScene
│   ├── FinalScoreDisplay
│   ├── HighScoreDisplay
│   ├── EvolutionStageReached
│   ├── ReplayButton
│   └── MenuButton
└── LeaderboardScene
    ├── TopScoresTable
    ├── BackButton
    └── RefreshButton
```

1.2.5 Persistence & Storage

Responsabilità Primaria: Gestire il salvataggio e il caricamento dei dati persistenti.

Componente	Funzione
StorageManager	Interfaccia centralizzata per accesso a localStorage
HighScoreRepository	Gestione CRUD per gli high score
GameStateSerializer	Serializzazione/deserializzazione dello stato di gioco
DataValidator	Validazione dell'integrità dei dati salvati

Schema di Archiviazione (LocalStorage):

```
// High Score Entry
{
  id: "score_&lt;timestamp&gt;",
  playerName: string,
  score: number,
  evolutionStage: number,
  timestamp: ISO8601,
  sessionDuration: number (secondi),
  checksum: hash (per validazione)
}

// Collection Structure
localStorage["snakeEvolution_scores"] = JSON.stringify([...scores])
localStorage["snakeEvolution_version"] = "1.0"
```

1.2.6 Audio-Visual System

Responsabilità Primaria: Gestire audio e feedback visivo-uditivo.

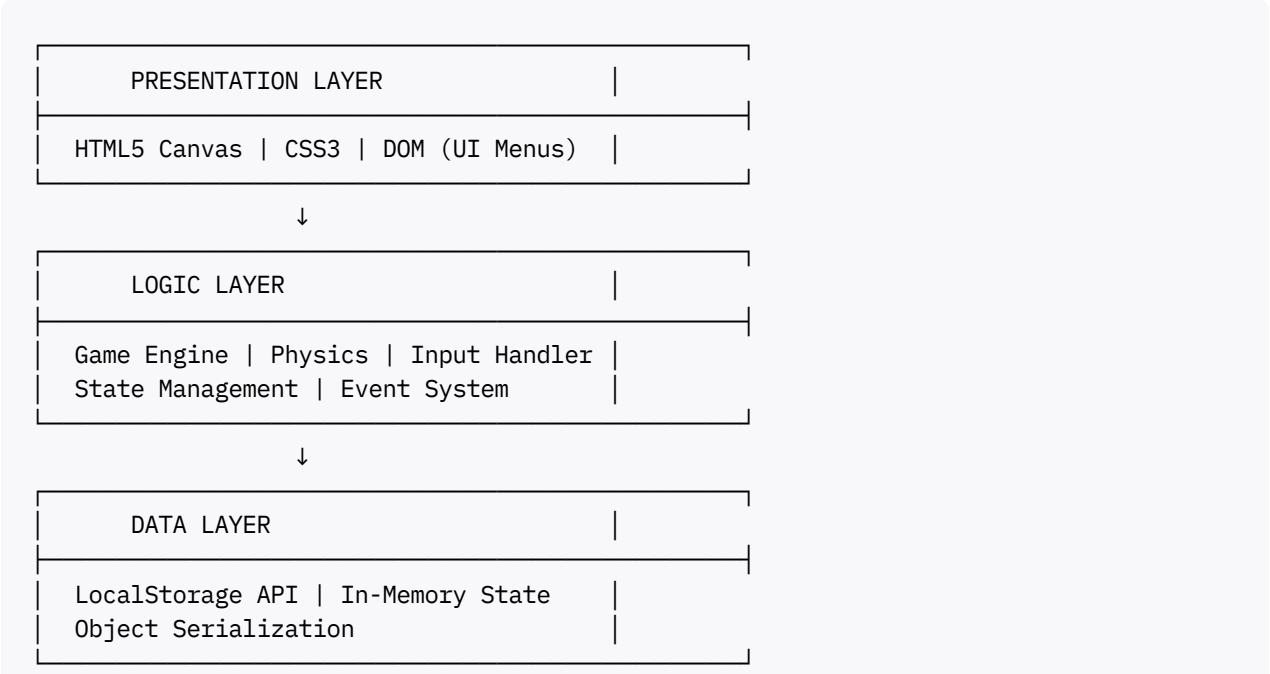
Componente	Funzione
AudioManager	Gestione centralizzata dell'audio
SoundEffectPlayer	Riproduzione di effetti sonori su eventi specifici
MusicPlayer	Gestione della musica ambient in loop
AudioSettingsManager	Controllo volume e muting
ParticleSystem	Gestione degli effetti particellari per evoluzioni

Audio Assets Required:

Evento	File	Durata	Formato
Raccolta Cibo	food_collect.wav	0.2s	WAV/MP3
Evoluzione	evolution_upgrade.wav	0.8s	WAV/MP3
Game Over	game_over.wav	1.0s	WAV/MP3
BGM Ambient	ambient_loop.wav	30s	WAV/MP3

2. Architettura Tecnica Dettagliata

2.1 Stack Tecnologico



2.2 Pattern Architetturali Adottati

2.2.1 Model-View-Controller (MVC)

- **Model**: Stato di gioco (serpente, cibo, score, evoluzione)
- **View**: Canvas rendering e UI elements
- **Controller**: Input handling e game loop orchestration

2.2.2 Observer Pattern

- Event system per disaccoppiamento tra componenti
- Subscription a eventi di gioco (evoluzione, collision, game-over)

2.2.3 Singleton Pattern

- GameEngine, AudioManager, StorageManager (una sola istanza globale)

2.2.4 State Pattern

- EvolutionState, GameState (Play, Paused, GameOver)

2.2.5 Factory Pattern

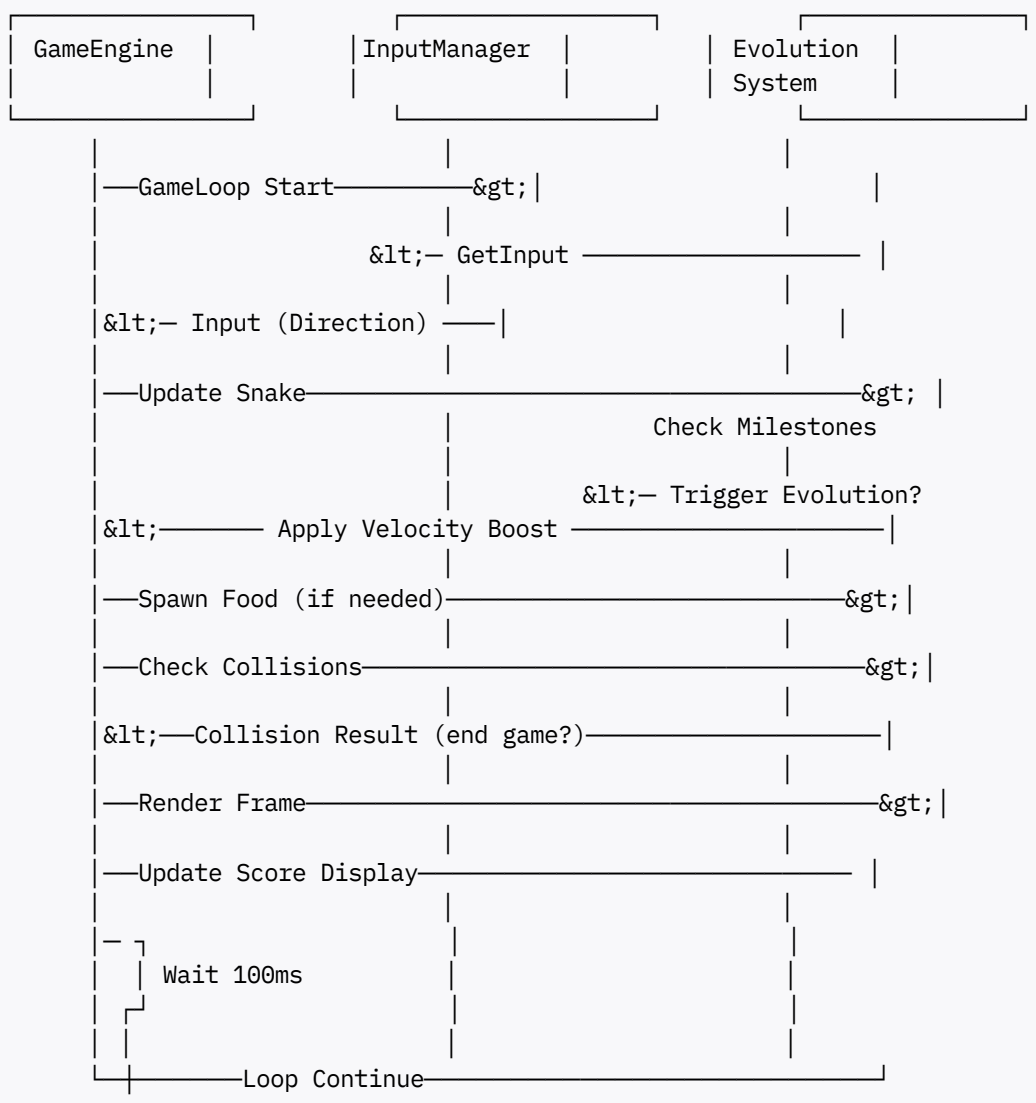
- Creazione dinamica di entità di gioco (Serpente, Cibo, effetti particellari)

2.3 Struttura di Cartelle Consigliata

```
snake-evolution/
├── src/
│   ├── core/
│   │   ├── GameEngine.js
│   │   ├── GameLoop.js
│   │   └── Constants.js
│   ├── entities/
│   │   ├── Snake.js
│   │   ├── Food.js
│   │   └── Grid.js
│   ├── systems/
│   │   ├── EvolutionSystem.js
│   │   ├── CollisionSystem.js
│   │   ├── InputManager.js
│   │   └── AudioManager.js
│   ├── ui/
│   │   ├── SceneManager.js
│   │   ├── UIRenderer.js
│   │   └── scenes/
│   │       ├── MainMenuScene.js
│   │       ├── GameScene.js
│   │       ├── GameOverScene.js
│   │       └── LeaderboardScene.js
```

```
│   ├── storage/
│   │   ├── StorageManager.js
│   │   └── HighScoreRepository.js
│   ├── events/
│   │   └── EventBus.js
│   └── utils/
│       ├── Math.js
│       ├── Validators.js
│       └── Logger.js
├── assets/
│   ├── audio/
│   │   ├── sfx/
│   │   └── music/
│   └── graphics/
│       └── sprites/
├── index.html
├── style.css
└── main.js
```

2.4 Diagramma di Sequenza - Ciclo di Gioco Completo



```
        │ (if game not over)
      ┌─┘
```

3. Specifiche Dei Componenti Core

3.1 GameEngine - Architettura Interna

```
// Pseudocodice architetturale

class GameEngine {
  constructor() {
    this.grid = new GridSystem(20, 20);
    this.snake = new Snake(this.grid);
    this.food = new Food(this.grid);
    this.evolutionSystem = new EvolutionSystem(this.snake);
    this.inputManager = new InputManager();
    this.audioManager = new AudioManager();
    this.renderer = new UIRenderer();
    this.storage = new StorageManager();
    this.eventBus = new EventBus();
    this.gameState = "MENU"; // MENU, PLAYING, PAUSED, GAMEOVER
    this.score = 0;
    this.isRunning = false;
  }

  async initialize() {
    await this.audioManager.loadAssets();
    await this.renderer.loadScenes();
    this.attachEventListeners();
  }

  startGame() {
    this.gameState = "PLAYING";
    this.snake.reset();
    this.food.spawn();
    this.score = 0;
    this.gameLoop();
  }

  gameLoop() {
    const input = this.inputManager.getInput();
    this.snake.updateDirection(input);

    this.snake.move();

    if (this.snake.isTouchingFood(this.food)) {
      this.snake.grow();
      this.score += 10;
      this.audioManager.play("food_collect");
      this.food.spawn();

      // Controlla evoluzione
      const newStage = this.evolutionSystem.checkEvolution();
```

```
      if (newStage) {
        this.triggerEvolution(newStage);
      }
    }

    if (this.snake.checkCollision()) {
      this.endGame();
      return;
    }

    this.renderer.render(this.snake, this.food, this.score);

    if (this.gameState === "PLAYING") {
      setTimeout(() =&gt; this.gameLoop(), 100);
    }
  }

  triggerEvolution(stage) {
    this.evolutionSystem.applyEvolution(stage);
    this.renderer.playEvolutionAnimation(stage);
    this.audioManager.play("evolution");
    this.eventBus.emit("EVOLUTION_TRIGGERED", stage);
  }

  endGame() {
    this.gameState = "GAMEOVER";
    this.storage.saveHighScore({
      playerName: this.currentPlayerName,
      score: this.score,
      evolutionStage: this.evolutionSystem.currentStage,
      timestamp: new Date().toISOString()
    });
    this.renderer.showGameOverScreen(this.score);
  }

  togglePause() {
    this.gameState = this.gameState === "PAUSED" ? "PLAYING" : "PAUSED";
    if (this.gameState === "PLAYING") {
      this.gameLoop();
    }
  }
}
```

3.2 EvolutionSystem - Logica di Transizione

```
class EvolutionSystem {
  constructor(snake) {
    this.snake = snake;
    this.currentStage = 0;
    this.stages = [
      { name: "Base", range: [0, 10], color: "#00AA00", speedMult: 1.0, vfx: "none" },
      { name: "Crescente", range: [11, 25], color: "#0088FF", speedMult: 1.05, vfx: "part
      { name: "Consapevole", range: [26, 50], color: "#00DD00", speedMult: 1.10, vfx: "tr
      { name: "Potenziato", range: [51, 75], color: "#FF4400", speedMult: 1.15, vfx: "par
```

```
      { name: "Leggendario", range: [76, Infinity], color: "#FFD700", speedMult: 1.20, vf
    ];
  }

  checkEvolution() {
    const length = this.snake.getLength();
    const newStage = this.stages.findIndex(s =&gt;
      length &gt;= s.range[^0] &amp;&amp; length &lt;= s.range[^1]
    );

    if (newStage !== this.currentStage) {
      this.currentStage = newStage;
      return this.stages[newStage];
    }
    return null;
  }

  applyEvolution(stage) {
    this.snake.setColor(stage.color);
    this.snake.setSpeedMultiplier(stage.speedMult);
    this.snake.setVFXType(stage.vfx);
  }
}
```

3.3 InputManager - Normalizzazione Input Multi-Device

```
class InputManager {
  constructor() {
    this.currentDirection = "RIGHT";
    this.nextDirection = "RIGHT";
    this.deviceType = this.detectDevice();
    this.setupListeners();
  }

  detectDevice() {
    return /mobile|android|iphone/i.test(navigator.userAgent) ? "touch" : "keyboard";
  }

  setupListeners() {
    if (this.deviceType === "keyboard") {
      document.addEventListener("keydown", (e) =&gt; this.handleKeyboard(e));
    } else {
      document.addEventListener("touchstart", (e) =&gt; this.handleTouchStart(e));
      document.addEventListener("touchmove", (e) =&gt; this.handleTouchMove(e));
    }
  }

  handleKeyboard(event) {
    const mapping = {
      "ArrowUp": "UP", "w": "UP",
      "ArrowDown": "DOWN", "s": "DOWN",
      "ArrowLeft": "LEFT", "a": "LEFT",
      "ArrowRight": "RIGHT", "d": "RIGHT",
      " ": "PAUSE"
```

```
    };

    const direction = mapping[event.key];
    if (direction &amp;amp;&amp;amp; this.isValidMove(direction)) {
      this.nextDirection = direction;
    }
  }

  handleTouchStart(event) {
    this.touchStartX = event.touches[^0].clientX;
    this.touchStartY = event.touches[^0].clientY;
  }

  handleTouchMove(event) {
    const deltaX = event.touches[^0].clientX - this.touchStartX;
    const deltaY = event.touches[^0].clientY - this.touchStartY;

    if (Math.abs(deltaX) &gt; Math.abs(deltaY)) {
      const direction = deltaX &gt; 0 ? "RIGHT" : "LEFT";
      if (this.isValidMove(direction)) this.nextDirection = direction;
    } else {
      const direction = deltaY &gt; 0 ? "DOWN" : "UP";
      if (this.isValidMove(direction)) this.nextDirection = direction;
    }
  }

  isValidMove(direction) {
    // Previene movimento contrario (no 180° turn)
    const opposites = { UP: "DOWN", DOWN: "UP", LEFT: "RIGHT", RIGHT: "LEFT" };
    return direction !== opposites[this.currentDirection];
  }

  getInput() {
    this.currentDirection = this.nextDirection;
    return this.currentDirection;
  }
}
```

4. Flussi di Interazione Principali

4.1 Flusso di Avvio (Initialization)

```
User Opens Game
    ↓
Load HTML/CSS
    ↓
Initialize Engine
    ├─ Load Audio Assets
    ├─ Create Canvas
    ├─ Load localStorage Data
    └─ Create Event Listeners
    ↓
Show Main Menu
```

```
    ├─ Display Title
    ├─ Display Buttons (Play, Leaderboard, Settings)
    └─ Await User Input
```

4.2 Flusso di Gioco (In-Game Loop)

```
User Clicks Play
    ↓
Show Player Name Input Dialog
    ↓
Start Game
    ├─ Initialize Snake (3 segments, center)
    ├─ Spawn Food
    ├─ Reset Score to 0
    ├─ Start Evolution System
    └─ Begin Game Loop (100ms ticks)
    ↓
Every 100ms:
    ├─ Read Input (direction)
    ├─ Move Snake
    ├─ Check Food Collision
    │   ├─ If Touched: Grow, Score +10, Play Sound
    │   └─ Check Evolution Milestone
    ├─ Check Boundary/Self Collision
    ├─ Render Frame
    └─ Update HUD (Score, Stage)
    ↓
Until Collision Detected
    ↓
Game Over Screen
    ├─ Display Final Score
    ├─ Display High Score
    ├─ Display Evolution Stage Reached
    ├─ Save Score to localStorage
    └─ Show Replay/Menu Buttons
```

4.3 Flusso di Evoluzione Dettagliato

```
Snake Length Reaches Milestone
    ↓
EvolutionSystem.checkEvolution()
    ├─ Current Length: 11 (triggers Stadio Crescente)
    ├─ Get Stage Data:
    │   ├─ Color: "#0088FF"
    │   ├─ Speed Multiplier: 1.05
    │   └─ VFX: "particle_trail_light"
    ↓
Trigger Evolution Event
    ├─ Play Animation (1 second):
    │   ├─ Flash Screen
    │   ├─ Particle Burst
    │   └─ Color Transition
    ├─ Play Audio (evolution_upgrade.wav)
```

```
    ├─ Update Snake Properties
    │   ├─ Apply New Color
    │   ├─ Apply Speed Multiplier
    │   └─ Enable VFX
    └─ Show UI Notification ("Stage Erreached: Crescente")
    ↓
Resume Game with Updated Properties
```

4.4 Flusso di Salvataggio (Persistence)

```
Game Over State Reached
    ↓
StorageManager.saveHighScore({
  playerName: string,
  score: number,
  evolutionStage: number,
  timestamp: ISO8601,
  sessionDuration: number
})
    ↓
Serialize Data to JSON
    ↓
Calculate Checksum (MD5)
    ↓
Store in localStorage["snakeEvolution_scores"]
    ↓
Update Leaderboard Immediately
    ↓
Game Over Screen Shows:
    ├─ Your Score: X
    ├─ High Score: Y
    └─ Rank in Top 10: Z
```

5. Specifiche di Performance e Ottimizzazione

5.1 Target di Performance

Metrica	Target	Metodo di Misurazione
FPS	60 FPS costante	requestAnimationFrame timing
Load Time	< 2 secondi	Lighthouse, WebPageTest
Input Latency	< 50ms	Chrome DevTools performance
Memory Usage	< 50MB	Chrome Task Manager
File Size	< 15MB	Bundled asset size

5.2 Strategie di Ottimizzazione

2D Rendering Optimization

```
// Usare dirty rectangle rendering (redraw only changed areas)
class OptimizedRenderer {
  lastFrameState = null;

  render(gameState) {
    // Solo ridisegnare se lo stato è cambiato
    if (this.hasStateChanged(gameState)) {
      this.ctx.clearRect(0, 0, canvas.width, canvas.height);
      this.drawGrid();
      this.drawSnake(gameState.snake);
      this.drawFood(gameState.food);
      this.lastFrameState = JSON.stringify(gameState);
    }
  }

  hasStateChanged(gameState) {
    return JSON.stringify(gameState) !== this.lastFrameState;
  }
}
```

Memory Management

- Utilizzare object pooling per particelle e effetti
- Evitare allocazioni nel game loop
- Pulire event listeners in modo appropriato

Asset Optimization

- Audio: Compress WAV to MP3 (30KB per effect)
- Graphics: Use sprite sheets per ridurre draw calls
- CSS: Minimize per migliorare parsing

6. Error Handling e Logging

6.1 Strategia di Error Handling

```
class ErrorHandler {
  static handle(error, context = {}) {
    const errorReport = {
      message: error.message,
      stack: error.stack,
      context,
      timestamp: new Date().toISOString(),
      userAgent: navigator.userAgent
```

```
    };

    // Log locale
    Logger.error(errorReport);

    // UI Feedback
    if (error.severity === "critical") {
      this.showUserMessage("Game Error. Please refresh.");
    } else if (error.severity === "warning") {
      this.showUserMessage("Minor issue detected.");
    }
  }
}
```

6.2 Logging Implementation

```
class Logger {
  static log(level, message, data = {}) {
    const logEntry = {
      level,
      message,
      data,
      timestamp: new Date().toISOString()
    };

    console[level.toLowerCase()](logEntry);

    // Store in IndexedDB per debug
    this.storeLog(logEntry);
  }

  static error(message, data) { this.log("ERROR", message, data); }
  static warn(message, data) { this.log("WARN", message, data); }
  static info(message, data) { this.log("INFO", message, data); }
}
```

7. Testing Strategy

7.1 Matrice di Test

Tipo	Copertura	Strumenti
Unit Test	70%+ di logica core	Jest, Mocha
Integration Test	Game loop, salvataggio	Cypress, Playwright
Performance Test	FPS, Load time, Memory	Lighthouse, Chrome DevTools
Playtesting	Balance e UX	External testers

7.2 Test Cases Critici

```
// Unit Test: EvolutionSystem
describe("EvolutionSystem", () =&gt; {
  it("should trigger evolution at correct milestone", () =&gt; {
    const snake = new Snake();
    const system = new EvolutionSystem(snake);

    snake.length = 11;
    const result = system.checkEvolution();

    expect(result.name).toBe("Crescente");
    expect(result.speedMult).toBe(1.05);
  });
});

// Integration Test: Game Loop
describe("Game Loop", () =&gt; {
  it("should end game on collision", (done) =&gt; {
    const engine = new GameEngine();
    engine.startGame();

    // Force collision after 5 iterations
    setTimeout(() =&gt; {
      engine.snake.collideWithWall();
      expect(engine.gameState).toBe("GAMEOVER");
      done();
    }, 500);
  });
});
```

8. Considerazioni di Sicurezza

8.1 Data Security

- **localStorage**: Non crittografare (non è sensitive)
- **Input Validation**: Sanitizzare input giocatore (nome)
- **XSS Prevention**: Usare textContent invece di innerHTML
- **CORS**: Se asset esterni, usare appropriate CORS headers

8.2 User Privacy (GDPR)

- Non tracciare dati personali beyond game scores
- Permettere cancellazione dati su richiesta
- Transparency about what data is stored

9. Roadmap di Implementazione (8 Settimane)

Settimana 1: Foundation

- [x] Setup progetto e build tools
- [x] Creare struttura HTML/CSS base
- [x] Implementare Canvas e context setup

Settimana 2-3: Core Engine

- [x] Snake controller e movimento
- [x] Grid system e collision detection
- [x] Food spawner e raccolta
- [x] Game loop base

Settimana 4: Evolution System

- [x] Evolution tracker e state management
- [x] VFX for transformations
- [x] Audio integration

Settimana 5: UI/UX

- [x] Scene manager e transitions
- [x] Menu principale
- [x] Game HUD
- [x] Game over screen

Settimana 6: Persistence & Polish

- [x] Storage manager e localStorage
- [x] Leaderboard UI
- [x] Audio settings
- [x] Performance optimization

Settimana 7: Testing

- [x] Unit tests
- [x] Integration tests
- [x] Playtesting con utenti esterni
- [x] Bug fixes

Settimana 8: Launch

- [x] Final optimization
- [x] Deploy
- [x] Monitoring post-launch

10. Appendice: Decision Record

10.1 Decisioni Architetturali Chiave

Decision	Rationale	Alternative Considerata
HTML5 Canvas over WebGL	Semplicità per indie team	WebGL per graphics avanzati
Vanilla JS over Phaser	Controllo totale e learning	Phaser.js per rapid development
localStorage over IndexedDB	Semplicità e storage sufficiente	IndexedDB per dati complessi
100ms tick rate	Balance tra responsività e performance	50ms (più fluido ma più CPU)

10.2 Metriche di Successo Architetturale

- ✓ Codice modularizzato (6 macro-aree indipendenti)
- ✓ Decoupling tra componenti (Event Bus pattern)
- ✓ Performance target raggiunti (60 FPS, <2s load)
- ✓ Scalabilità per future iterazioni (v2 features)
- ✓ Maintainability (codice leggibile e testabile)

Documento Firmato

Senior Software Architect
Data: Novembre 2025
Status: Approved for Development

Documento di Riferimento: PRD Snake Evolution v1.0
Versione Analisi: 1.0
Ultimo Aggiornamento: Novembre 2025

⁂