

Snake Evolution - Design Decisions & Architecture Rationale v1.0

Document Header

Project: Snake Evolution Game

Document Type: Design Decisions & Architecture Rationale

Version: 1.0

Date: November 2025

Status: Reference Documentation

Prepared by: Senior Software Architect

Audience: Development Team, Architecture Review Board, Future Maintainers

Executive Summary

This document captures the **reasoning behind key architectural decisions** made during Snake Evolution v2.1 development. It explains not just *what* was built, but **why** specific technical choices were made, what alternatives were considered, and what trade-offs were accepted.

Purpose

- **Who should read:** Developers, architects, technical leads, future maintainers
- **What's covered:** Key architecture decisions, alternatives considered, trade-offs, design rationale
- **What's NOT covered:** Implementation details (see Technical Analysis) or requirements (see PRD)

Decision Categories

1. **Architecture Decisions** — System design and layering
2. **Technology Choices** — Framework, tooling, and platform selection
3. **Algorithm Choices** — Data structures and performance decisions
4. **Operational Decisions** — DevOps, monitoring, deployment strategies

1. Architecture Decisions

1.1 Layered Architecture (MVC-inspired)

Decision: Organize code into 5 distinct layers:

1. Presentation Layer (UI/scenes)
2. State Machine Layer (core logic)

3. Game Logic Layer (mechanics)
4. Data Persistence Layer (storage)
5. Infrastructure Layer (logging, profiling)

Rationale:

✓ **Separation of Concerns** — Each layer has single responsibility

- Presentation doesn't depend on persistence
- Game logic independent of UI
- Easy to test each layer independently
- Easy to swap implementations (e.g., different renderer)

✓ **Maintainability** — Clear structure for future developers

- New team members understand architecture quickly
- Adding features means knowing which layer to modify
- Reduced "spaghetti code" problems

✓ **Scalability** — Can extend without redesign

- Add new UI layer (mobile-specific)
- Add new persistence layer (cloud sync)
- Add new logic layer (new game modes)

Alternatives Considered:

✗ **Flat Architecture** (all code in one file)

- Rejected: Unmaintainable at scale, difficult to test

✗ **Event-Driven Only** (pure pub/sub, no layers)

- Rejected: No clear flow, race conditions likely

✗ **Game Engine Framework** (Unity, Godot)

- Rejected: Overkill for casual game, learning curve too high

Trade-offs:

⚖ **Some indirection** — More files/folders to navigate

- Mitigated by: Clear documentation and organization

⚖ **Communication overhead** — Data flows through layers

- Mitigated by: Event bus keeps layers loosely coupled

1.2 Finite State Machine with Atomic Locking

Decision: Implement state transitions as atomic operations with spin-lock to prevent race conditions.

Rationale:

✓ **Race Condition Prevention** — State never corrupted by concurrent transitions

- Previous attempts without locking had subtle bugs
- Concurrent event handling could cause invalid state combinations
- Atomic transactions ensure consistency

✓ **Predictable Behavior** — Clear state rules

- Valid transitions explicitly defined
- Queuing prevents input loss
- Recovery on error (automatic rollback)

✓ **Debuggability** — State history tracking

- Can replay exactly how state changed
- Easier to diagnose issues
- Better logging and monitoring

Alternatives Considered:

✗ Simple flag-based protection

- Rejected: Still vulnerable to re-entrancy issues

✗ Async/await pattern only (no locking)

- Rejected: JavaScript's single-threaded nature doesn't protect against event loop interleaving

✗ Redux-style immutable state

- Rejected: Too heavyweight for this game; overkill complexity

Trade-offs:

⚖ **Slight performance cost** — Spin-lock has minimal overhead

- Mitigated by: `Sleep(1ms)` reduces busy-waiting

⚖ **State queue requirement** — Queues queued transitions

- Benefit: Better than dropping inputs
- Alternative would be: Drop inputs (loses player commands)

1.3 Spatial Hashing for Collision Detection

Decision: Use spatial hash grid instead of brute-force $O(n)$ collision checks.

Rationale:

✓ **Performance** — $O(1)$ lookup vs $O(n)$ brute force

- At 200 cells, brute force = 200 checks per frame
- Spatial hash = 1-4 checks per frame (constant)
- At 60 FPS, this is significant

✓ **Scalability** — Accommodates future larger snakes

- Design supports unlimited snake size
- Won't degrade at 500+ cells
- Performance envelope remains consistent

✓ **Clear Algorithm** — Easy to understand and verify

- Visual grid structure in code matches concept
- Easy to add debugging visualization
- Easy to test with unit tests

Alternatives Considered:

✗ **Brute Force Collision** (check all segments)

- Rejected: $O(n)$ complexity; performance degrades with length

✗ **BSP Tree** (binary space partition)

- Rejected: Overkill complexity for 2D grid game

✗ **Quadtree** (hierarchical grid)

- Rejected: More complex than spatial hash without benefit for game scale

Trade-offs:

⚖ **Memory overhead** — Hash grid takes memory

- Mitigated by: Minimal (hash map of cell occupancy)
- Grid size 20x20: only 400 entries

⚖ **Grid resolution matters** — Must choose cell size carefully

- Cell too large: False positives (inefficient)
- Cell too small: More cells to check (defeats purpose)
- Decision: Cell size = 1 game unit (optimal)

2. Technology Choices

2.1 Vanilla JavaScript (No Framework)

Decision: Use ES6+ vanilla JavaScript instead of React, Vue, or other framework.

Rationale:

✓ **Bundle Size** — Zero framework overhead

- React: ~40KB minified
- Vue: ~30KB minified
- Vanilla JS: 0KB
- Target bundle: 512KB ✓ achieved

✓ **Performance** — No framework runtime penalty

- No virtual DOM overhead
- Direct Canvas manipulation
- Direct DOM manipulation when needed

✓ **Simplicity** — Canvas game doesn't need reactive framework

- No complex component state trees
- Imperative code clearer for game logic
- Easier for new developers to understand

✓ **Learning** — Good for teaching

- Shows fundamentals: events, state, rendering
- Not abstracted away by framework magic

Alternatives Considered:

✗ **React/Vue** — Popular frameworks

- Rejected: Adds 30-40KB; not designed for Canvas games
- Overkill for single-page game

✗ **Babylon.js/Three.js** — 3D engines

- Rejected: Overkill for 2D game; massive overhead

✗ **Phaser** (game framework)

- Rejected: Good option but adds 200KB+; want to learn from scratch

Trade-offs:

⚖ **No built-in reactivity** — Must manage state manually

- Mitigated by: Simple state machine handles this well

- Manual is actually clearer here

⚖ **More DOM boilerplate** — No JSX, no template syntax

- Mitigated by: Game is mostly Canvas, minimal DOM
- HTML is simple (main menu, HUD)

2.2 Canvas 2D API (Not WebGL)

Decision: Use Canvas 2D instead of WebGL for rendering.

Rationale:

✓ **Simplicity** — Much simpler API than WebGL

- Draw rectangles, circles, text directly
- No shader programming required
- Easier to debug visuals

✓ **Sufficiency** — Adequate for 2D pixel art style

- 2D sprites don't require 3D capabilities
- Performance: 60 FPS achievable easily
- Visual style suits Canvas 2D

✓ **Compatibility** — Wider browser support

- Canvas 2D supported everywhere
- WebGL more variable (especially mobile)

✓ **Developer Experience** — Lower learning curve

- New developers can contribute immediately
- Debugging tools more familiar
- Performance profiling straightforward

Alternatives Considered:

✗ **WebGL** — 3D graphics API

- Rejected: Overkill for 2D; requires shader expertise

✗ **SVG** — Vector graphics

- Rejected: Performance concerns for animated pixel grid

✗ **CSS Animations** — Style-based animation

- Rejected: Not suitable for real-time game loop; synchronization issues

Trade-offs:

🚧 **Performance ceiling** — Canvas 2D slower than WebGL

- Reality: Still does 60 FPS easily
- Actual limitation would appear at 500+ cells
- Decision: Worth it for simplicity until hitting actual ceiling

2.3 localStorage for Persistence (Not Database)

Decision: Use browser localStorage for saving leaderboard instead of cloud backend.

Rationale:

✓ **No Backend Required** — Eliminates server cost and complexity

- Netlify static hosting sufficient
- No database to manage
- No user authentication needed

✓ **Privacy** — User data stays on their device

- No telemetry collection
- GDPR compliance automatic
- No data breach risk

✓ **Offline Capability** — Works without internet

- Data persists even offline
- Leaderboard visible without server

✓ **Simplicity** — Client-side only solution

- Easier to develop
- Easier to test
- Easier to maintain

Alternatives Considered:

✗ **Cloud Database** (Firebase, AWS DynamoDB)

- Rejected: Adds cost, complexity, infrastructure
- Better for: Multiplayer (planned v3+)

✗ **IndexedDB** — Bigger local storage

- Rejected: localStorage sufficient for top-10 scores
- IndexedDB overkill for this use case

✗ **Service Worker Cache** — Offline-first

- Rejected: Not needed yet; planned for v2+

Trade-offs:

⚖️ **No cross-device sync** — Scores stuck on one device

- Mitigated by: Cloud sync planned for v2+
- Feature backlog item

⚖️ **Limited by storage quota** — 5-10MB typical limit

- Reality: Top-10 leaderboard only ~1KB
- Actual limit is not constraint for v1

2.4 GitHub Actions for CI/CD

Decision: Use GitHub Actions instead of Jenkins, GitLab CI, or other platform.

Rationale:

✓ **Integrated** — Part of GitHub (where code lives)

- No separate system to maintain
- Authentication automatic
- Workflows live with code (versioned)

✓ **Cost** — Free tier sufficient

- 2,000 minutes/month free
- Enough for continuous deployment
- No paid add-ons needed

✓ **Simplicity** — YAML-based configuration

- Human-readable
- Easy to learn
- Extensive documentation

✓ **Feature-Rich** — Has everything needed

- Build, test, lint, deploy
- Matrix testing (multiple environments)
- Artifact storage included

Alternatives Considered:

✗ **Jenkins** — Self-hosted CI

- Rejected: Requires server maintenance
- Complexity overkill for this project

✗ **GitLab CI** — Integrated with GitLab

- Rejected: Project already on GitHub

✗ **CircleCI/TravisCI** — Third-party services

- Rejected: Additional cost; GitHub Actions sufficient

Trade-offs:

⚖️ **Vendor lock-in** — Tied to GitHub ecosystem

- Mitigated by: GitHub is industry standard; exit cost low
- Could migrate to other CI if needed

3. Algorithm & Data Structure Decisions

3.1 Random Food Spawning Algorithm

Decision: Weighted random spawn (avoid occupied cells) vs. pure random retry loop.

Rationale:

✓ **Reliability** — Guaranteed to find free spot

- While loop until unoccupied cell found
- Works even at high snake density

✓ **Simplicity** — Easy to understand and debug

- No complex weight calculation
- No edge cases with weights

✓ **Performance** — Still $O(1)$ amortized

- Expected retries $\approx \text{gridSize} / \text{snakeLength}$
- At 200 cells on 400 grid: ~2 retries average

Alternative:

✗ **Weighted probability grid**

- Rejected: More complex; no real benefit
- Slightly better performance gains not worth complexity

Trade-offs:

⚖️ **Small chance of empty retry** — At 200+ cells, rare

- Reality: Acceptable tradeoff

3.2 Snake Movement: Queue-Based vs. Direct

Decision: Queue next movement, validate before execution.

Rationale:

✓ **Input Buffering** — Captures player intent

- Player can "pre-input" next direction
- Feels responsive even at 50ms input debounce
- Competitive gaming standard

✓ **Prevents Accidental Reversal** — No 180° turns

- Check direction validity before queuing
- Prevents death from rapid input changes

Alternative:

✗ **Direct movement (apply immediately)**

- Rejected: Feels less responsive
- Rapid inputs could cause accidental reversals
- Less forgiving gameplay

Trade-offs:

⚖ **Queue complexity** — Additional data structure

- Mitigated by: Simple queue; easy to understand

3.3 Performance Profiling: Custom vs. Framework

Decision: Build custom FPS counter and frame timer instead of using profiling framework.

Rationale:

✓ **Lightweight** — No additional dependencies

- Minimal overhead on performance
- Accurate measurements (no framework overhead)

✓ **Specific to Game** — Tailored metrics

- Measures what matters: frame time, collision time, render time
- Not generic metrics

✓ **Learning Tool** — Shows how profiling works

- Good teaching example
- Developers understand performance deeply

Alternative:

✗ Profiler.js or similar

- Rejected: Framework overhead defeats purpose of profiling

Trade-offs:

⚖️ **Manual implementation** — More code to write

- Mitigated by: ~100 lines of profiling code
- Worth it for accuracy and learning

4. Operational & Deployment Decisions

4.1 Netlify (Static Hosting) vs. Container Deployment

Decision: Use Netlify (static file hosting) instead of Docker/Kubernetes.

Rationale:

✓ **Simplicity** — Single deployment command

- No Docker setup
- No container orchestration
- No DevOps expertise required

✓ **Cost** — Free tier sufficient

- No monthly server costs
- Pay only if scale requires
- Perfect for MVP/launch

✓ **CDN Included** — Global distribution

- Automatic GZIP compression
- Edge caching
- Fast worldwide delivery

✓ **Automatic Deployment** — Git integration

- Push to main → automatic deploy
- No manual deployment steps
- Atomic deployments (blue/green)

Alternatives Considered:

✗ **Docker + Kubernetes** — Container orchestration

- Rejected: Massive overkill for static game

- Would increase complexity 10x

✗ **AWS S3 + CloudFront** — DIY setup

- Rejected: More configuration required
- Netlify gives same capability with less work

Trade-offs:

⚖️ **Vendor lock-in** — Tied to Netlify

- Mitigated by: Built on standard tech (just static HTML/CSS/JS)
- Migration cost low (just copy files to another host)

⚖️ **Limited scalability** — Not suitable for 100M+ users

- Reality: Fine for MVP; easy to migrate if needed
- Decision: Premature optimization avoided

4.2 Monitoring: Application-Level vs. Infrastructure

Decision: Focus on application-level metrics (FPS, error rate) not just infrastructure.

Rationale:

✓ **User Experience** — Metrics that matter to players

- FPS directly impacts enjoyment
- Error rate shows reliability
- Load time shows responsiveness

✓ **Actionability** — Can identify problems quickly

- "Low FPS on mobile" is actionable
- Generic CPU metrics less helpful

✓ **Cost** — Custom metrics cheaper

- Browser-side collection
- No expensive APM tool needed

Alternative:

✗ **Infrastructure-only monitoring** (CPU, memory, disk)

- Rejected: Doesn't show game quality
- Tells you system is up, not that game works well

Trade-offs:

⚖️ **More implementation work** — Build custom dashboards

- Mitigated by: Simple implementation; worth it

4.3 Testing Strategy: Unit + Integration + E2E

Decision: 85% target coverage (60% unit, 30% integration, 10% E2E) instead of 100% unit.

Rationale:

✓ **Practical Coverage** — Tests what matters most

- Critical paths: 100% coverage
- Edge cases: Good coverage
- UI details: Less testing needed

✓ **Maintenance** — Less test brittleness

- E2E tests catch real user issues
- Not testing implementation details
- Tests survive refactoring

✓ **Development Speed** — Faster test writing

- Don't need test for every function
- Focus on behavior, not coverage numbers

Alternatives Considered:

✗ 100% code coverage

- Rejected: False security; brittle tests
- Many tests would test implementation, not behavior

✗ Only E2E testing

- Rejected: Slow; hard to debug
- Integration tests faster and catch issues earlier

Trade-offs:

⚠ **Some code paths untested** — Edge cases may slip through

- Mitigated by: Good integration tests catch most issues
- Risk acceptable for MVP

5. Evolution & Future Decisions

5.1 Why Layered Architecture Supports Evolution

For v2.0 (Cloud Sync):

Existing decision to separate **Data Persistence Layer** means:

- Add new cloud persistence implementation
- Existing game logic unchanged
- Minimal risk of regressions

For v3.0 (Multiplayer):

Existing decision for **Event Bus communication** means:

- Add new multiplayer event handlers
- Existing single-player logic unchanged
- No refactoring needed

For v2.5 (Mobile App):

Existing decision for **Vanilla JS** means:

- Bundle same code for mobile
- No React Native necessary
- Just swap rendering layer

5.2 Known Limitations by Design

Why we didn't optimize further:

1. Canvas Rendering Bottleneck

- Could use WebGL, but: Not needed for MVP
- Plan: Switch if performance ceiling hit
- Current: 60 FPS easily achieved

2. No Cloud Persistence

- Could add backend, but: Cost/complexity unjustified for v1
- Plan: Add in v2 with cloud storage
- Current: localStorage sufficient

3. No Multiplayer

- Could add real-time sync, but: Enormous scope addition
- Plan: Add in v3 with proper architecture
- Current: Single-player works well

6. Decision Traceability

6.1 Decisions by Stakeholder

User Experience Decisions:

- Spatial hashing → Smooth 60 FPS gameplay
- Input buffering → Responsive feel
- Evolution system → Engagement progression

Business Decisions:

- Netlify hosting → Low cost (<\$10/month)
- No ads/tracking → Privacy-first positioning
- Open source → Community goodwill

Technical Decisions:

- Vanilla JS → Smaller bundle, faster load
- Canvas 2D → Simpler than WebGL
- GitHub Actions → Free, integrated CI/CD

Developer Experience Decisions:

- Layered architecture → Clear code organization
- FSM with locking → Predictable state
- Custom profiler → Learning opportunity

7. Document Information

Item	Details
Document Type	Design Decisions & Architecture Rationale
Version	1.0
Date Published	November 2025
Standard	ADR (Architecture Decision Record) inspired
Language	English
Audience	Development, Architecture, Maintenance

Document Version History

Version	Date	Changes
1.0	2025-11-06	Initial decision documentation for v2.1

Appendix: Decision Template

For future architecture decisions, use this template:

```
## Decision: [Title]

**Status:** PROPOSED / ACCEPTED / REJECTED

**Context:**
[What was the problem or requirement?]

**Rationale:**
[Why this decision?]

**Alternatives Considered:**
- [Alternative 1] – Reason rejected
- [Alternative 2] – Reason rejected

**Trade-offs:**
⚖ [Downside 1] – How mitigated
⚖ [Downside 2] – How mitigated

**Consequences:**
✔ Positive impact
✖ Negative impact

**Reversibility:**
[Easy to change? Difficult? Why?]

**Related Decisions:**
- Decision X (affected by)
- Decision Y (affects)
```

End of Design Decisions & Architecture Rationale

For implementation details, see Technical Analysis v2.1.

For requirements, see PRD v2.1.

For development procedures, see Implementation Guide v2.0.

[1] [2]

✱

1. PRD-v2-1-EN.pdf

2. Functional-Analysis-EN.pdf

