DreamUp: Browser Game QA Pipeline

Version: 1.2
Date: November 3, 2025
Timeline: 3-5 days core implementation + optional stretch features

Overview

DreamUp is a general AI game generator that began at Gauntlet's first 24-hour hackathon.If successful, your project would be used to help make our game-building-agent capable of automated self-improvement.

Objective

Build an AI agent that autonomously tests browser games by simulating user interactions, capturing visual evidence, and evaluating playability metrics. The system must work with any web-hosted game URL.

Business Value

- Solution may be used to automate game QA for DreamUp's generated games
- Enables feedback loops for game-building agent improvement
- Demonstrates production-ready AI agent architecture

Success Criteria

- Successfully tests 3+ diverse browser games end-to-end
- Generates structured reports with 80%+ accuracy on playability assessment
- Handles common failure modes gracefully (crashes, slow loads, rendering issues)
- Clean, documented, modular codebase

Scope

In Scope (Core Features - Days 1-5)

1. **Browser Automation Agent**

    - Load game from URL
    - Detect and handle common UI patterns (start buttons, menus, game over screens)
    - Walk through the game based on the controls it finds
    - Implement timeouts and retry logic

2. **Evidence Capture**

    - Take 3-5 timestamped screenshots per test session
    - Save artifacts to structured output directory
    - Include console logs and error messages

3. **AI Evaluation**

    - Use LLM to analyze screenshots and logs
    - Assess: successful load, responsive controls, stability
    - Output structured JSON with pass/fail, confidence scores, and issue descriptions

4. **Execution Interface**

    - Our game dev agent runs in a lambda function and we would like to invoke the QA from this environment
    - Typescript file is preferred (i.e. executed with `bun run qa.ts`, `npx tsx qa.ts`, etc.)
    - Running with a CLI command is acceptable too: `qa-agent <game-url>`
    - Structured output: `{status, playability_score, issues[], screenshots[], timestamp}`

Out of Scope

- Multiplayer or network-dependent games
- Mobile browser emulation
- Security/performance testing
- Integration with DreamUp production systems (prototype only)

Optional Stretch Features (If Time Permits)

- **GIF Recording:** Capture gameplay as animated GIF for reports
- **Batch Testing:** Sequential testing of multiple URLs with aggregated reporting
- **Advanced Metrics:** FPS monitoring, load time analysis, accessibility checks
- **Web Dashboard:** Simple UI for viewing test results and history

Game Engine Context

Scene Stack

The game engine has four scene types: Canvas2D & Canvas3D scenes provide full ECS runtimes with physics, rendering, and game logic; UI scenes are pure DOM elements; and

Composite scenes layer multiple child scenes together. The game is represented as a stack of these scenes with push/pop operations.

One common pattern is a composite scene containing a 2D/3D game scene with a UI overlay for the HUD. UI scenes can optionally suspend scenes beneath them (pausing game logic while showing pause menus), while composite scenes automatically manage suspension state and propagate lifecycle calls (mount, unmount, update, draw) to all children.

Input System

The input system uses a two-layer architecture: low-level hardware capture (keys, mouse, pointer) and high-level gameplay abstractions. Actions map multiple inputs to named gameplay events, tracking pressed, down, released, and hold duration states.

Axes provide continuous values: 1D axes return [-1, 1] (with smoothing and opposite-direction cancellation), while 2D axes return vectors {x, y} with diagonal normalization. These axes can be bound to WASD, arrow keys, virtual joysticks, and D-pads.

Actions and axes decouple game logic from hardware—multiple input sources trigger the same action or axis, allowing keyboard, touch, and virtual controls to work interchangeably. Game code queries these abstractions through the scene's InputManager, keeping implementation details hidden.

Our game dev agent picks the input schema as it plans the game; you can assume that the QA agent will be provided the control layout as an input prompt, such as:

```
// Actions - discrete button events
gameBuilder.createAction('Jump')
  .bindKey(' ')
  .bindKey('w')
  .bindVirtualButton('#btn-jump')

// In system: scene.inputManager.isActionPressed('Jump')

// 1D Axis - horizontal movement (-1 to 1)
gameBuilder.createAxis('MoveHorizontal')
  .bindKeys('a', 'd')
  .bindKeys('ArrowLeft', 'ArrowRight')
  .bindButtons('#dpad .dpad-left', '#dpad .dpad-right')
  .setSmoothing(0.15)

// In system: const x = scene.inputManager.getAxis('MoveHorizontal')

// 2D Axis - combined directional input (normalized)
gameBuilder.createAxis2D('Move')
  .bindWASD()
  .bindArrowKeys()
  .bindJoystick('#joystick')
  .bindVirtualDPad('#dpad')
  .setSmoothing(0.2)

// In system: const {x, y} = scene.inputManager.getAxis2D('Move')
```

An example input schema in game code.

Technical Architecture

Stack

- **Browser Use:** Browserbase w/ Stagehand (recommended) or alternative
- **Language:** Typescript preferred, though any language is allowed
- **LLM Framework:** Vercel's AI SDK preferred

Agent Design

1. Initialize → Load game URL in headless browser
2. Observe → Wait for initial render, capture baseline screenshot
3. Interact → Execute action sequence:
    a. Find and click start/play buttons
    b. Simulate basic gameplay (arrow keys, spacebar, mouse clicks)
    c. Navigate 2-3 screens/levels if applicable
4. Monitor → Detect crashes, freezes, or errors via:

    a. Console error logs
    b. Page state changes
    c. Screenshot comparison
5. Evaluate → Submit evidence to LLM with structured prompt:
    a. "Does the game load successfully?"
    b. "Are controls responsive?"
    c. "Did the game complete without crashes?"
6. Report → Generate JSON output with findings

Error Handling

- Max execution time: 5 minutes per game
- Retry failed loads up to 3 times
- Graceful degradation if screenshots fail

Deliverables

Required

1. **Source Code:** GitHub repository with clear structure
2. **Documentation:** README with setup, usage, and architecture overview
3. **Test Results:** QA reports for 3-5 sample games (include screenshots)
4. **Demo:** 2-5 minute video showing end-to-end execution

Optional

- Unit and integration tests
- Docker containerization for easy setup
- Comparison of results across different LLM models

Test Cases

Validate against diverse game types:

1. **Simple Puzzle:** Basic click interactions (e.g., tic-tac-toe)
2. **Platformer:** Keyboard controls, physics (e.g., simple Mario clone)
3. **Idle/Clicker:** Minimal interaction, persistent state
4. **Broken Game:** Intentionally buggy game to test failure detection
5. **Complex Game:** Multiple levels/screens (e.g., RPG demo)

For the purposes of this project, you can work with browser games from any source you want. Feel free to make your own test cases or use these repositories of free HTML games:

- itch.io/games/html5
- kongregate.com
- html5games.com

Evaluation Criteria

- **Functionality:** Core features work as specified
- **Code Quality:** Clean architecture, proper error handling, documentation
- **AI Integration:** Effective prompts, accurate evaluations
- **Robustness:** Handles edge cases, recovers from failures

Risks & Mitigations

Risk	Impact	Mitigation
Agent loops infinitely	High	Max action count, total timeout
LLM gives inconsistent results	Medium	Structured prompts, confidence thresholds, fallback heuristics
Games don't load in headless mode	High	Test with headed mode, screenshot comparison
Scope creep	High	Strict adherence to 3-5 day core; no stretches until complete
API costs exceed budget	Low	Cache responses, use cheaper models for iteration

Getting Started

Recommended approach:

1. Set up Browserbase (free tier has 1 browser-hour included)
2. Create minimal agent that loads a game and takes a screenshot
3. Add one interaction type at a time
4. Integrate LLM evaluation last (can mock initially)
5. Iterate on real games early and often

Contact:

Matt Smith (slack or email: matt.smith@superbuilders.school)

Also feel free to ask questions in the `#superbuilders-projects` slack channel (if it becomes too busy, we can switch to the `#dreamup-projects` channel that we set up earlier).