Sentinel UI-Local: Arquitectura de Agente Cognitivo

Este documento define la capa superior de **Sentinel**, transformando el modelo de lenguaje local (**Gemma**) en un agente autónomo capaz de navegar y testear flujos de usuario de "A a B" mediante razonamiento recursivo y validación multimodal.

1. El Núcleo Agentico (Cognitive Engine)

Para lograr un comportamiento de agente real, factorizamos la lógica en un ciclo **ReAct (Reasoning + Acting)**. El agente no recibe una lista de comandos; recibe un **Objetivo Final** y genera su propio plan dinámico.

1.1. Ciclo de Vida del Agente (The Loop)

1. **Percepción Multimodal:** Captura del VisualTreeAsset (JSON) y del render de cámara (Screenshot).
2. **Razonamiento (CoT):** Descomposición del objetivo en una "Sub-tarea inmediata".
3. **Selección de Herramienta:** Elección de la herramienta especializada más eficiente.
4. **Ejecución y Observación:** Realización de la acción y análisis del cambio resultante.
5. **Evaluación de Progreso:** Decisión de si se ha alcanzado el objetivo o si se requiere una corrección.

2. Especialización de Herramientas Agenticas

Para reforzar la autonomía, el agente dispone de herramientas que le permiten gestionar su propio proceso de navegación.

Herramienta	Propósito Agentico	Comportamiento del Agente
analyze_navigation_map	Crea un mapa mental de la jerarquía.	El agente identifica "nodos ciegos" o paneles ocultos.
execute_atomic_action	Realiza una interacción física (Click/Type).	El agente espera un "Reflow" de la UI antes de seguir.
verify_visual_state	Compara la captura de cámara con la intención.	El agente confirma que el mundo 3D reaccionó a la UI.

report_blockage	Documenta por qué no puede seguir el flujo.	El agente autodiagnostica errores (ej: un modal bloquea el click).

3. Implementación de la Capa de Razonamiento (C#)

Siguiendo principios de tipado fuerte y SOLID, el **AgentController** actúa como el puente de pensamiento del Editor.

3.1. Interfaz del Controlador del Agente

```
namespace Sentinel.Editor.Interfaces
{
   public interface IAgentController
   {
      System.Threading.Tasks.Task<bool> ProcessGoalAsync(string goalDescription);
      void UpdateAgentMemory(string observation);
   }
}
```

3.2. Implementación del Agente (Editor Persistent)

```
using UnityEngine;
using UnityEngine.UIElements;
using System.Collections.Generic;
using System.Threading.Tasks;
using Sentinel.Editor.Interfaces;

namespace Sentinel.Editor.Core
{
   public sealed class AgentCognitiveController : IAgentController
   {
      [SerializeField] private List<string> _actionHistory = new List<string>();
      private string _currentGoal;

      public async Task<bool> ProcessGoalAsync(string goalDescription)
      {
         _currentGoal = goalDescription;
         bool goalReached = false;
         int maxAttempts = 10;

         while (!goalReached && maxAttempts > 0)
```

```
        {
            // 1. Obtener estado actual (Percepción)
            string uiState = SentinelEditorProvider.Inspector.GetSerializedHierarchy();
            string visualCapture = SentinelEditorProvider.Vision.GetCaptureAsBase64();

            // 2. Consultar a Gemma (Razonamiento)
            // El prompt incluye la memoria (_actionHistory) y el estado actual.
            AgentDecision decision = await GetGemmaDecision(uiState, visualCapture);

            // 3. Ejecutar acción especializada
            string result = await ExecuteDecisionAsync(decision);

            // 4. Actualizar memoria
            _actionHistory.Add($"Acción: {decision.ToolName} | Resultado: {result}");

            if (result == "GOAL_REACHED") goalReached = true;
            maxAttempts--;
        }

        return goalReached;
    }

    private async Task<string> ExecuteDecisionAsync(AgentDecision decision)
    {
        // El agente elige la herramienta especializada (Atomic Action)
        return await SentinelEditorProvider.Interaction.DispatchAction(decision);
    }
  }
}
```

4. Estrategia de "Pensamiento en Cadena" (Chain of Thought)

Para que Gemma se comporte como un agente, el sistema le obliga a seguir un protocolo de razonamiento antes de emitir un JSON de herramienta.

Protocolo de Razonamiento del Agente:

1. **MEMORIA:** ¿Qué hice justo antes? (Evitar bucles infinitos).
2. **OBSTÁCULO:** ¿Hay algo que me impida ver el objetivo? (Un dropdown abierto, un fade-out).
3. **INTENCIÓN:** ¿Cuál es el paso más pequeño que me acerca al panel de "Sonido"?

4. **VERIFICACIÓN:** Si hago click aquí, ¿qué espero ver en la cámara o en el árbol visual?

5. Gestión de Fallos y Autocorrección

Un agente real sabe cuando ha fallado. Hemos reforzado el sistema con **Estrategias de Recuperación**:

- **Retry con Inspección Profunda:** Si un click no produce un cambio visual, el agente usa get_element_properties para ver si el elemento tiene un pickingMode incorrecto.
- **Backtrack:** Si el agente llega a un "callejón sin salida" (ej: entra en una sección equivocada), es capaz de identificar el botón de "Atrás" analizando el texto de la UI mediante find_element_by_text.
- **Validación Cruzada:** El agente no cree a la UI ciegamente. Si el VisualElement dice "Abierto" pero capture_game_view no muestra cambios, el agente reporta una **Inconsistencia de Estado**.

6. Conclusión de la Arquitectura

Al factorizar la inteligencia en este **AgentController**, convertimos a Unity en un entorno donde el LLM no es un invitado, sino un operador con criterio. La combinación de las herramientas del **Unite 2025 Framework** con la capacidad de visión de cámara permite que el agente valide flujos de juego completos, asegurándose de que el volumen realmente cambió porque "vio" el slider moverse y "leyó" el estado interno del motor.