Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Escuela de Doctorado y Estudios de Postgrado

Máster Universitario en Desarrollo de Videojuegos

TRABAJO FIN DE MÁSTER

NPCs autónomos basados en modelos de lenguaje para la simulación de un mercado de un videojuego.

Autonomous NPCs based on Large Language Models for market simulation in a video game.

Autor: Antonio Vázquez Carreño

Tutor: Rafael Arnay del Arco

La Laguna, 05 de julio de 2026

Antonio Vázquez Carreño

Agradecimientos

Antes que nada, agradecer a mis hijos, por estar siempre a mi lado y permitirme a ratos dedicarle el tiempo necesario a estos estudios.

A mi familia, por apoyarme en todo momento y dedicarme su tiempo cuando lo he necesitado

A Rafael, por su dedicación y por todos los vaivenes que ha sufrido este trabajo, siempre aconsejando y ayudándome a mejorarlo

A los profesores del Máster, porque siento que he aprendido mucho y se nota el empeño que le ponen a cada clase.

D. **Rafael Arnay del Arco**, profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

“NPCs autónomos basados en modelos de lenguaje para la simulación de un mercado de un videojuego.”

ha sido realizada bajo su dirección por D. Antonio Vázquez Carreño.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos, firman la presente en La Laguna a 09 de Julio de 2026.

Resumen

Este Trabajo Fin de Máster presenta el diseño y la implementación de un videojuego 2D que simula un pequeño entorno de mercado habitado por personajes no jugadores (NPC) autónomos. A diferencia de los NPC tradicionales, guionizados con diálogos fijos, estos personajes se comportan como agentes generativos: perciben su entorno, mantienen una memoria persistente, deciden qué hacer y conversan usando un modelo de lenguaje grande (Claude, de Anthropic). Sobre esa base viven una economía tangible —compran materiales, producen bienes en sus talleres, comercian entre ellos y con el jugador, y cubren necesidades diarias que cierran el ciclo del dinero— y generan misiones para el jugador a partir de sus propias necesidades.

El trabajo pone el acento en la arquitectura de los agentes: una memoria episódica persistente sobre una base de datos vectorial —con recuperación por relevancia, recencia e importancia y reflexiones periódicas, siguiendo el esquema de los agentes generativos de Park et al.— y un diseño híbrido en el que el modelo de lenguaje decide y redacta (a dónde ir, qué decir, qué encargar) mientras el código valida y ejecuta (objetos, precios y recompensas). El mercado —compras, producción y encargos— actúa, así, como vehículo de interacción entre personajes sin que una respuesta mal formada del modelo pueda romper el juego. Esa vida interna se hace visible mediante un cliente 2D en Unity (mapa por tiles, sprites y una interfaz con ventana de diálogo, fichas de personaje e iconos de misión). El sistema se organiza en dos capas: un backend en Python (FastAPI) que aloja el motor de agentes, la memoria vectorial (ChromaDB) y la economía, y el cliente Unity que dibuja el mundo y gestiona la interacción, comunicados por una API REST.

Palabras clave: agentes generativos, modelos de lenguaje, memoria episódica, NPC autónomos, simulación de mercado, economía emergente, sistema híbrido, base de datos vectorial, videojuego 2D, Unity.

Abstract

This Master's Thesis presents the design and implementation of a 2D video game that simulates a small marketplace environment inhabited by autonomous non-player characters (NPCs). Unlike traditional scripted NPCs, these characters behave as generative agents: they perceive their surroundings, keep a persistent memory, decide what to do and converse by means of a large language model (Anthropic's Claude). On top of that, they sustain a tangible economy —buying raw materials, producing goods in their workshops, trading with one another and with the player, and covering daily needs that close the money loop— and generate quests for the player out of their own needs.

The project focuses on the agent architecture: a persistent episodic memory on top of a vector database —with retrieval by relevance, recency and importance and periodic reflections, following the generative-agents scheme of Park et al.— and a hybrid design in which the language model decides and writes (where to go, what to say, what to request) while the code validates and executes (items, prices and rewards). The market —purchases, production and quests— thus acts as the interaction vehicle between characters, and no malformed model response can break the game. This inner life is made visible through a 2D Unity client (tile map, sprites and a user interface with a dialogue window, character sheets and quest markers). The system is organised in two layers: a Python backend (FastAPI) hosting the agent engine, the vector memory (ChromaDB) and the economy, and the Unity client that renders the world and handles interaction, communicating through a REST API.

Keywords: generative agents, large language models, episodic memory, autonomous NPCs, market simulation, emergent economy, hybrid system, vector database, 2D video game, Unity.

Índice

Índice de Ilustraciones

Tablas de Código

Índice de tablas

1. Introducción

1.1. Contexto y motivación

Los personajes no jugadores (NPC) de la mayoría de los videojuegos siguen guiones cerrados: repiten las mismas frases, ofrecen misiones predefinidas y no recuerdan lo que el jugador hizo cinco minutos antes. En géneros que giran en torno al comercio y la economía —simuladores de gestión, juegos de rol con mercaderes, títulos de comercio— esa rigidez se nota especialmente: los precios son tablas fijas, los comerciantes tienen existencias concretas o infinitas y el mundo no reacciona a lo que hace el jugador.

La aparición de los modelos de lenguaje grandes (LLM) abre la puerta a un tipo de NPC distinto, capaz de conversar de forma abierta, recordar interacciones pasadas y tomar decisiones propias según su situación. El trabajo de Park et al. (2023) sobre agentes generativos demostró que es posible construir poblaciones de agentes creíbles combinando un LLM con una memoria persistente y mecanismos de reflexión y planificación. La motivación de este TFM es llevar esa idea a un caso concreto y jugable: un pequeño entorno de mercado donde unos pocos habitantes producen, compran, venden y encargan tareas, y donde el jugador puede pasear, hablar con ellos y participar en esa economía viva.

Otra de las características que se pretenden llevar a cabo es introducir a un jugador real en medio de este ecosistema generacional, que los NPCs sean conscientes de su existencia y que tengan la posibilidad de interactuar con él y viceversa. Se ha trabajado también el diseño, aportando una interfaz que haga legibles el estado de los personajes, sus conversaciones, sus intercambios económicos y las misiones que proponen.

1.2. Punto de partida: los agentes generativos de Park et al. (2023)

Este trabajo no parte de cero: toma como base directa el artículo «Generative Agents: Interactive Simulacra of Human Behavior» (Park et al., 2023).

Ilustración 1. Generative Agents (Park)

En él, un grupo de agentes controlados por un modelo de lenguaje habitan un pequeño mundo simulado («Smallville») y muestran comportamientos creíbles —planifican su día, recuerdan lo que les pasa, reflexionan sobre ello y se relacionan entre sí— sin un guion escrito de antemano. De ese trabajo se adopta lo esencial de su arquitectura cognitiva: dotar a cada personaje de un **flujo de memoria** (memory stream), recuperar los **recuerdos relevantes** combinando recencia, importancia y relevancia semántica, y **generar reflexiones** de alto nivel que guían la conducta posterior. La propia importancia de la memoria como pieza central del comportamiento se hereda de ese artículo.

Hay que resaltar que la idea de este proyecto comenzó hace 2 años, cuando el estado de la Inteligencia Artificial y los LLMs estaban en un estado mucho más primitivo y menos desarrollado que ahora mismo. En ese proyecto el sistema anterior de guardado de memoria y las respuestas no eran muy precisas, lo que hacía difícil controlar que el flujo se ejecutara con la normalidad que se requiere. Por ejemplo, intenté que recordara la fecha de mi cumpleaños, pero fue una tarea imposible, sabía que habíamos hablado de ello, pero la fecha no le parecía algo determinante.

Sobre esa base, este TFM introduce cuatro cambios deliberados respecto al trabajo de generative agents:

- **Del frontend web a Unity.** El sandbox original era una aplicación web (un mundo 2D renderizado en el navegador con un servidor detrás). Aquí se sustituye por un cliente nativo en Unity, más adecuado para un videojuego y para el trabajo gráfico y de interfaz.
- **De ChatGPT a Claude.** El trabajo original empleaba ChatGPT (GPT-3.5). Aquí se usa Claude (Anthropic), con distintos modelos según la tarea. En la práctica, junto con el diseño de los prompts y la recuperación de memoria, se han reducido de forma notable las respuestas incoherentes o «alucinadas» respecto a lo que el personaje debería saber.
- **Nueva persistencia con base vectorial.** En lugar del almacén de recuerdos del planteamiento original (archivo json estructurado), la memoria se apoya en una base de datos vectorial persistente (ChromaDB) con embeddings locales (all-MiniLM-L6-v2). Esto permite indexar y consultar los recuerdos por significado y conservarlos entre sesiones, con un coste de inferencia bajo al calcularse los embeddings en local.
- **Dominio orientado al mercado.** Frente al pueblo genérico del artículo, aquí el mundo se orienta a un entorno de mercado con oficios, producción, comercio y misiones, de modo que la vida de los agentes tenga consecuencias económicas concretas y jugables.

En resumen, se conserva la arquitectura cognitiva del artículo (memoria + recuperación + reflexión) y se moderniza y especializa su implementación (Unity, Claude, base de datos vectorial y un dominio de mercado).

La siguiente tabla resume las diferencias.

Aspecto	Park et al. (2023)	Este TFM
Frontend	Aplicación web (mundo 2D en navegador)	Cliente nativo en Unity (C#)
Modelo de lenguaje	ChatGPT (GPT-3.5)	Claude (Anthropic), varios modelos según la tarea
Memoria / persistencia	Flujo de recuerdos en texto	Base de datos vectorial persistente (ChromaDB) + embeddings locales
Recuperación de recuerdos	Recencia + importancia + relevancia	Misma fórmula (recencia + importancia + relevancia), sobre embeddings locales
Dominio	Vida social genérica (Smallville)	Entorno de mercado: oficios, economía y misiones
Escala	25 agentes	4 agentes (pueblo pequeño, foco jugable)

Tabla 1. Comparativa Park vs Vázquez.

1.3. Descripción del proyecto

El resultado es un juego 2D desarrollado en Unity en el que conviven cuatro personajes con oficios diferenciados dentro de un pueblo con varias localizaciones (plaza, panadería, taller, botica y mercado): María (comerciante), Tomás (panadero), Klaus (herrero) e Isabella (alquimista). Cada uno se desplaza por el pueblo, conversa con los demás, compra materiales, produce objetos y, cuando necesita algo, genera una misión que el jugador puede aceptar y completar. El jugador se mueve con el teclado, habla con los NPC acercándose a ellos y gestiona un inventario propio con dinero y objetos. En la ilustración 2 podemos ver la composición del poblado.

Ilustración 2. Pueblo inicial

El núcleo jugable es el entorno de mercado: los objetos son reales (tienen dueño y cambian de manos), los precios dependen de la relación entre personajes y la disponibilidad de materiales fluctúa, de modo que la economía no es un decorado sino un sistema con consecuencias.

Conviene aclarar también el alcance. El planteamiento inicial era un entorno plenamente autónomo en el que el jugador conviviera con los personajes en todas sus facetas. Al estudiar la casuística que ello implica —planes diarios completos, vínculos que evolucionan con cada interacción, drama social—, se optó por acotar el trabajo a una demostración construida alrededor del mercado: la compraventa, la producción y los encargos son el vehículo de interacción entre los personajes y con el jugador. La evolución de las relaciones y del vínculo que se va creando entre personajes a partir de sus interacciones queda planteada como trabajo futuro (capítulo 6). Esta acotación no resta autonomía al sistema: cada decisión de comportamiento —a dónde ir, qué decir, si encargar algo al jugador— nace de la respuesta del modelo de IA sobre el que se sustenta el trabajo, y el código se limita a validar y ejecutar esas decisiones para mantener el mundo consistente.

1.4. Objetivos

1.4.1. Objetivo general

Diseñar e implementar un videojuego 2D que simule un entorno de mercado habitado por NPC autónomos basados en un modelo de lenguaje, con especial atención a la persistencia conversacional, la coherencia entre relaciones, encargos y situación real del mundo, agregando una interfaz que comunica al jugador el comportamiento de los agentes, su economía y sus misiones, además de dar la posibilidad de interactuar con ellos.

1.4.2. Objetivos específicos

1. Construir el mundo del pueblo de forma visual mediante un mapa por tiles y representar a los personajes con sprites, con un sistema de cámara y de ordenación en profundidad adecuado a una vista cenital 2D.
2. Diseñar y desarrollar la capa de interfaz de usuario: ventana de diálogo con desplazamiento, fichas de personaje, iconos de misión sobre los NPC, panel de misión, inventario y registro de eventos.
3. Modelar un entorno de mercado con objetos e inventarios reales, producción por oficios, precios influidos por las relaciones y escasez de materiales.
4. Dotar a cada agente de una máquina de estados de comportamiento (decidir, desplazarse, comprar, esperar material, producir, conversar y reflexionar) y de una memoria episódica persistente.
5. Generar misiones emergentes a partir de las necesidades de los NPC y del propio jugador, que el jugador pueda aceptar y completar, con una recompensa tangible.
6. Integrar todo en una arquitectura cliente–servidor que separe la lógica de los agentes (backend) de la representación del mundo (Unity).

1.5. Estructura de la memoria

Tras esta introducción, el capítulo 2 revisa el estado del arte y los trabajos relacionados. El capítulo 3 describe la metodología y las herramientas de desarrollo, la interfaz, las relaciones entre los personajes y la máquina de estados de los agentes. El capítulo 4 aterriza esa metodología en el desarrollo concreto del juego, hito a hito, e incluye los problemas técnicos encontrados. El capítulo 5 presenta los resultados con los casos de uso y los diagramas correspondientes. Los capítulos 6 y 7 recogen las conclusiones y líneas futuras en español e inglés. El capítulo 8 detalla el presupuesto y el 9 las referencias bibliográficas.

2. Trabajos relacionados y estado del arte

2.1. NPC en videojuegos: del guion a la generación

El enfoque clásico para dar vida a los NPC combina árboles de diálogo y máquinas de estados finitas o árboles de comportamiento. Es un enfoque robusto y predecible pero limitado: el personaje solo puede decir y hacer lo que se escribió de antemano, y no se adapta a situaciones no previstas. En la mayoría de los juegos centrados en el comercio, los comerciantes acotan el mercado, pueden ofrecer productos en base a la experiencia del jugador, como en World of Warcraft (WoW), donde la experiencia con las distintas facciones (o cadena de misiones) pueden alterar el precio y el tipo de material que ofrecen, pero siempre dentro de unas pautas ya marcadas en la implementación del código.

Los NPC basados en LLM permiten conversación abierta y comportamiento emergente. El reto no es solo técnico (latencia, coste de las llamadas, coherencia) sino de diseño: mantener a los personajes creíbles y útiles para la jugabilidad sin que el sistema se vuelva impredecible. Este trabajo aborda ese equilibrio con un enfoque híbrido, en el que el modelo aporta la parte creativa y el código garantiza la coherencia económica y la completabilidad de las misiones.

Los primeros sistemas que llevan esta idea a videojuegos reales confirman tanto el potencial como los riesgos. Song (2025) integra NPC conversacionales basados en LLM en Unity, con una memoria persistente compartida incluso entre plataformas, y señala la coherencia del recuerdo como el principal reto de ingeniería. PANGeA (Buongiorno et al., 2024) aborda el riesgo opuesto —que el texto libre del jugador arrastre al modelo fuera del guion— mediante un sistema de validación que contrasta cada salida generada con el estado del juego antes de aceptarla; este TFM adopta una filosofía análoga al validar las decisiones del modelo contra listas cerradas (localizaciones, catálogo de objetos) antes de ejecutarlas.

2.2. Agentes generativos (Park et al., 2023)

El trabajo de referencia de este proyecto es «Generative Agents: Interactive Simulacra of Human Behavior». Propone una arquitectura con tres piezas clave: un flujo de memoria (memory stream) donde se almacenan observaciones con marca de tiempo e importancia; un mecanismo de recuperación que selecciona los recuerdos relevantes combinando recencia, importancia y relevancia semántica; y procesos de reflexión y planificación que generan recuerdos de más alto nivel y guían la conducta futura.

El funcionamiento de estos agentes es cíclico. En cada paso de la simulación, el agente percibe su entorno y registra lo percibido como observaciones en el flujo de memoria; cuando debe actuar, recupera los recuerdos más pertinentes puntuándolos por recencia (un decaimiento exponencial de factor 0,995 desde el último acceso), importancia (valorada de 1 a 10 por el propio modelo) y relevancia semántica (similitud entre embeddings), y condiciona con ellos la llamada que decide la siguiente acción o la siguiente frase. Periódicamente, cuando la suma de importancias de los últimos sucesos supera un umbral (150 en su implementación, unas dos o tres veces por día simulado), el agente reflexiona: se plantea preguntas de alto nivel sobre lo vivido, las responde con sus propios recuerdos y guarda las conclusiones como recuerdos nuevos, construyendo abstracciones sobre abstracciones. A este ciclo se suma la planificación: un plan del día en trazos gruesos que se descompone recursivamente en franjas horarias y acciones de cinco a quince minutos, y que puede rehacerse ante imprevistos.

Para evaluar la credibilidad, los autores entrevistaron a los agentes en cinco áreas (autoconocimiento, memoria, planes, reacciones y reflexiones) y compararon la arquitectura completa contra versiones sin memoria, sin reflexión o sin planificación, ordenando los resultados con un rating TrueSkill: la arquitectura completa fue la más creíble y cada componente eliminado degradaba el resultado. Del experimento con veinticinco agentes emergieron además comportamientos sociales no programados —difusión de información, formación de relaciones, coordinación para organizar una fiesta— junto con los fallos que este TFM intenta acotar: errores de recuperación de memoria, adornos inventados sobre los recuerdos y un lenguaje excesivamente formal.

2.3. Memoria y recuperación semántica

La necesidad de dotar a los LLM de memoria a largo plazo ha generado una línea de trabajo propia. MemoryBank (Zhong et al., 2024) almacena las interacciones en un repositorio consultado por similitud vectorial e incorpora una tasa de olvido inspirada en la curva de Ebbinghaus, por la que los recuerdos se debilitan con el tiempo salvo que se recuperen de nuevo: una idea directamente emparentada con el decaimiento de recencia que este trabajo hereda de Park et al. En el terreno de los sistemas en producción, Mem0 (Chhikara et al., 2025) extrae, consolida y recupera dinámicamente la información relevante de conversaciones largas, y muestra que una memoria explícita mantiene la coherencia entre sesiones con un coste muy inferior a agrandar la ventana de contexto. El panorama lo sintetiza Du (2026), que formaliza la memoria de los agentes como un bucle de escribir, gestionar y leer acoplado a la percepción y la acción: el mismo patrón que implementa el memory stream de este proyecto sobre ChromaDB.

Para recuperar recuerdos por significado —y no solo por coincidencia de palabras— se usan embeddings: representaciones vectoriales del texto en las que frases parecidas quedan cerca en el espacio. En este proyecto se emplea el modelo all-MiniLM-L6-v2 (384 dimensiones), ejecutado localmente, y una base de datos vectorial (ChromaDB) para almacenar y consultar esos vectores por similitud del coseno. Es lo que permite que un NPC «recuerde» algo concreto que el jugador le contó y lo traiga a colación cuando viene al caso.

2.4. Representación visual: tilemaps y sprites en juegos 2D

En el desarrollo 2D, el mundo se construye habitualmente mediante mapas por baldosas (tilemaps): rejillas en las que cada celda toma una imagen de un conjunto de teselas (tileset), lo que permite dibujar escenarios grandes reutilizando pocas imágenes. Los personajes y objetos se representan con sprites, y su orden de dibujado se controla para simular profundidad en una vista cenital. Unity ofrece un sistema de Tilemap integrado y un renderizador de sprites con ordenación por capas y por posición. Para este proyecto se han utilizado conjuntos de teselas y personajes de estilo pixel-art de la colección Kenney (Tiny Town para el pueblo y Tiny Dungeon para los personajes), por su coherencia estética y su licencia libre.

2.5. Simulación económica en videojuegos

La simulación de mercados en videojuegos abarca desde economías totalmente guionizadas hasta sistemas con oferta y demanda dinámicas. El presente trabajo se sitúa en un punto intermedio y deliberadamente tangible: en lugar de modelar curvas de precios abstractas, cada objeto es una entidad con dueño, la producción consume materiales concretos y los

precios se modulan según la relación entre comprador y vendedor. Esta concreción hace la economía comprensible para el jugador y fácil de representar visualmente en inventarios y fichas.

3. Metodología y herramientas de desarrollo

3.1. Metodología de trabajo

El proyecto se ha desarrollado de forma incremental e iterativa, organizando el trabajo en hitos (H0–H4). Cada hito añade una capa funcional completa y verificable antes de pasar a la siguiente, lo que permite probar cada sistema de forma aislada y reducir el riesgo de integración. La lógica de los agentes se construyó y validó primero en el backend, usando su documentación interactiva de la API, y después se conectó a la representación en Unity.

3.2. Arquitectura general

El sistema sigue una arquitectura cliente–servidor bajo un enfoque de Modelo-Vista-Controlador (MVC) distribuido. El backend, en Python con FastAPI, actúa como el Modelo y el Controlador, concentrando toda la «inteligencia»: el motor de agentes, la memoria, la economía y las llamadas al modelo de lenguaje. Por su parte, el cliente en Unity concentra la Vista y la captura de entrada del jugador: dibuja el mundo, gestiona el movimiento del jugador y la interfaz, y nunca llama directamente al modelo de lenguaje; toda decisión de simulación se toma en el servidor. Esta separación de responsabilidades propia del patrón MVC permite iterar en la lógica sin tocar el juego y viceversa. La comunicación es REST/JSON. Toda esta arquitectura, se puede apreciar en la ilustración 3:

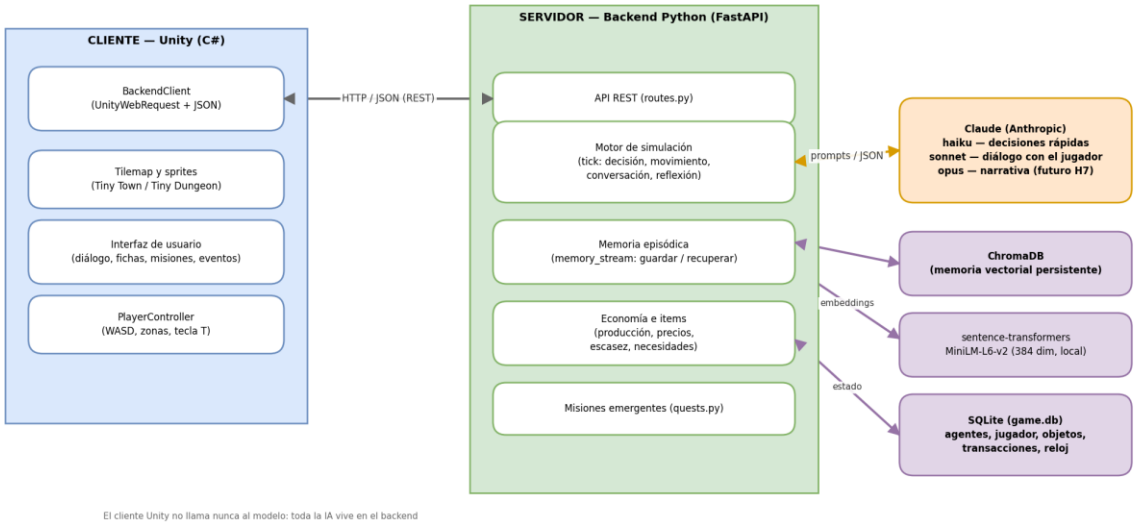

Ilustración 3. Diagrama de arquitectura cliente–servidor (Unity ⇄ API REST ⇄ backend FastAPI ⇄ Claude; ChromaDB y SQLite colgando del backend)

3.3. Tecnologías y librerías

La siguiente tabla resume las tecnologías empleadas y su papel en el proyecto.

Tecnología	Capa	Uso en el proyecto
Unity (C#)	Cliente	Motor del juego: renderizado 2D, tilemap, sprites, cámara, entrada del jugador e interfaz de usuario.
Tilemap de Unity	Cliente	Construcción del mapa del pueblo por baldosas y ordenación en profundidad.
Kenney Tiny Town / Tiny Dungeon	Arte	Conjuntos de teselas y sprites de personajes (pixel-art, licencia libre).
UnityWebRequest + Newtonsoft JSON	Cliente	Peticiones HTTP al backend y serialización/deserialización de los datos.
Python + FastAPI	Servidor	API REST, motor de agentes, orquestación de la simulación y la economía.
Anthropic Claude	IA	Diálogo, decisiones de acción, conversación entre NPC y reflexión.
ChromaDB	Datos	Base de datos vectorial para la memoria episódica de los agentes.
sentence-transformers (MiniLM-L6-v2)	IA	Cálculo local de embeddings (384 dimensiones) para la búsqueda semántica de recuerdos.
SQLite	Datos	Estado del mundo: agentes, jugador, objetos, transacciones y reloj de juego.

Tabla 2. Tecnologías y Librerías

3.4. Capa visual: hacer legibles a los agentes

Un sistema de agentes puede ser tan sofisticado como se quiera por dentro; para quien juega solo existe lo que aparece en pantalla. El papel de la capa visual en este proyecto no es decorativo: es hacer legible la vida interna de los agentes —dónde están, qué hacen y por qué, qué recuerdan, qué necesitan y qué opinan del jugador—. Cada elemento gráfico que se describe a continuación existe para responder a una de esas preguntas.
El mundo se construye con el sistema de Tilemap de Unity: una rejilla sobre la que se pintan teselas del conjunto Tiny Town (Kenney) para formar el suelo, los caminos, los edificios y la vegetación. Trabajar por baldosas aporta coherencia visual con muy pocos recursos y hace baratas las ampliaciones del mapa. Sobre ese escenario se sitúan los marcadores de las cinco localizaciones lógicas (plaza, panadería, taller, botica y mercado), que son el nexo entre lo visual y la simulación: el backend razona únicamente en términos de esas localizaciones, y el cliente las traduce a posiciones del mapa.

3.4.1. Personajes, cámara y movimiento

Los cuatro NPC y el jugador se representan con sprites del conjunto Tiny Dungeon. Para dar sensación de profundidad en la vista cenital, el orden de dibujado de cada sprite se ajusta según su posición vertical: quien tiene un orden de capa mayor se dibuja por delante. Los

NPC no se teletransportan: cuando el backend comunica que un agente ha decidido cambiar de localización, el cliente lo desplaza de forma suave hacia el marcador de destino, repartiendo a los presentes alrededor del punto para que no se solapen. El jugador se mueve con el teclado (WASD) y su zona se calcula por cercanía al marcador más próximo, que se reporta al backend; si se aleja de todas las zonas se informa el estado «afuera», lo que evita, por ejemplo, que un NPC salude a alguien que ya se ha ido. La cámara encuadra el pueblo con la acción centrada.

Los saludos y recordatorios exigen además contacto físico: el personaje que quiere hablar con el jugador camina hasta él —lo persigue unos segundos si se aleja— y la conversación solo se abre cuando lo alcanza, evitando charlas «a gritos» desde la otra punta de la zona.

La aproximación hasta el contacto (Unity, MapController.cs, extracto)

```
public void BeginApproach(string npcId, string opening, string name)
{
    _seekId = npcId; _seekOpening = opening; _seekName = name;
    _seekUntil = Time.time + 20f;        // si no te alcanza, desiste
}

void Update()                            // cada fotograma
{
    npc.transform.position = Vector3.MoveTowards(
        npc.transform.position, _playerT.position,
        moveSpeed * Time.deltaTime);
    if (Vector2.Distance(npc.transform.position, _playerT.position) < 1f)
        ChatUI.Instance.Open(_seekId, _seekOpening, _seekName); // contacto
}
```

Extracto de Código 1. Búsqueda del jugador y acercamiento

En la ilustración 4 se puede observar el mensaje para interaccionar con los jugadores.

Ilustración 4. Vista general del pueblo: mapa por tiles.

3.4.2. La interfaz de usuario

Toda la interfaz se construye por código mediante un conjunto de utilidades propias (UiKit) que crea lienzos, paneles, textos, campos de entrada y botones con un estilo común. Construirla programáticamente tiene una ventaja práctica —no depende de cómo esté montada la escena y es reproducible— y una conceptual: cada panel se diseñó partiendo de la pregunta «¿qué información del agente comunica?». Esta decisión se tomó porque en el

momento que los diálogos o la memoria era muy extensa, era difícil ajustar los scrollbar, por lo que se pasó a creación por código que se ajusta atendiendo al contenido existente. Los elementos principales son:

- Ventana de diálogo (ilustración 5): se abre al hablar con un NPC y pausa el juego. Muestra el historial con desplazamiento y colores por interlocutor (azul claro el jugador, ámbar el NPC), y permite enviar con la tecla Intro. No es solo presentación: al cerrarse, la transcripción viaja al backend, donde alimenta la memoria del personaje y puede dar lugar a un encargo (sección 3.10); durante la charla, cada mensaje se responde teniendo en cuenta lo ya dicho.

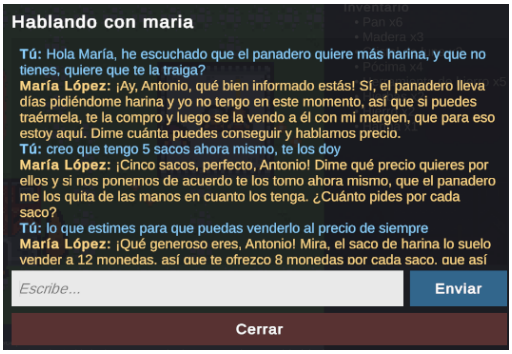

Ilustración 5. Ventana de diálogo

- Ficha de personaje (ilustración 6): al hacer clic sobre un NPC se abre su ficha con estado, dinero, inventario, habilidades, objetivos y recuerdos recientes. Es la ventana directa al interior del agente: permite comprobar en cualquier momento que lo que dice en conversación se corresponde con lo que realmente tiene y recuerda.

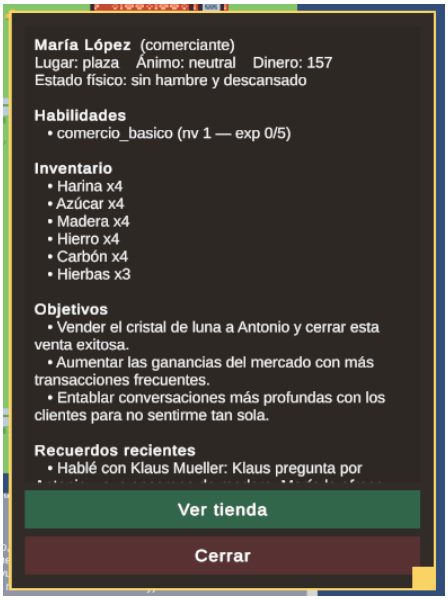

Ilustración 6. Ficha de NPC

- Iconos de misión (ilustración 7): un NPC con una misión abierta muestra un signo de exclamación sobre la cabeza, siguiendo la convención del género; los iconos se sincronizan consultando periódicamente al backend.

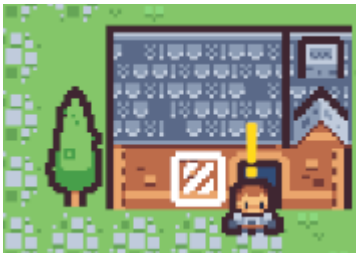

Ilustración 7. NPC con misión

- Panel de misión y armario (ilustración 8): sobre un NPC con misión se abre el panel para aceptarla y, más tarde, entregarla; el objeto pedido se obtiene de un «armario» con el catálogo cerrado de objetos que las misiones pueden requerir, ampliado con pan, pócimas y un pequeño «fondo del pueblo» de monedas en paquetes de 50, que actúa como válvula para desbloquear la economía si el jugador se queda sin liquidez.

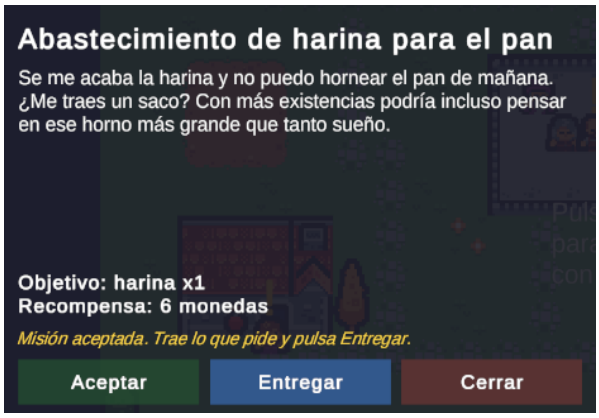

Ilustración 8. Ejemplo de misión

- Ficha e inventario del jugador (ilustración 9): dinero y objetos propios, consultables con un clic sobre el personaje.

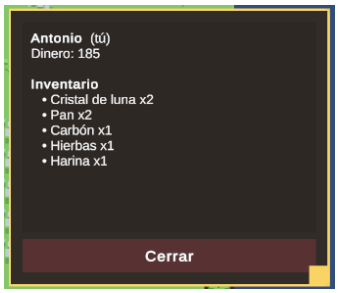

Ilustración 9. Ficha del personaje

- Registro de eventos (ilustración 10): un feed con desplazamiento narra cada paso de la simulación —movimientos con su intención declarada, conversaciones con su

resumen, compras, producción y reflexiones—, junto con el tiempo de proceso de cada paso. Es la vista de «lo que pasa cuando nadie mira» y fue, además, la principal herramienta de observación durante el desarrollo.

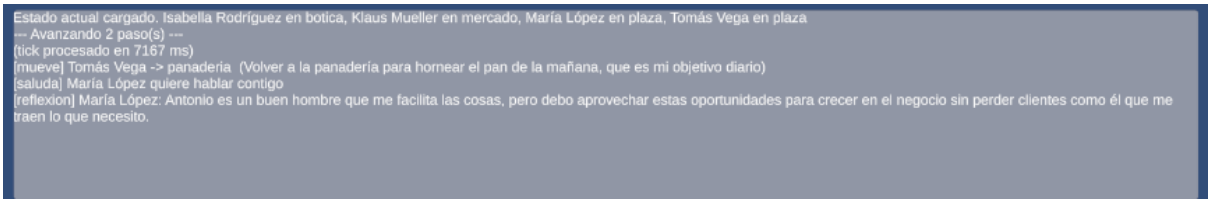

Ilustración 10. Registro de actividades

1. Venta de objetos (ilustración11): cada personaje puede tener en stock ciertos materiales. María vende todas las materias primas, mientras que el resto vende productos procesados. Hay que destacar que los productos comprados a otros personajes no aparecerán en la tienda de esos personajes, para que no exista una "competencia".

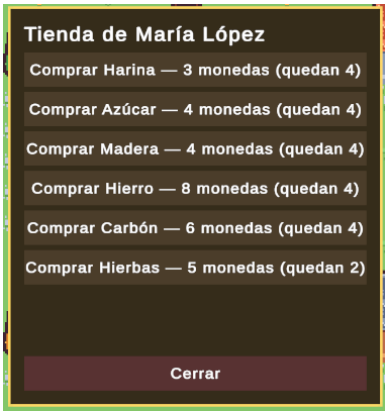

Ilustración 11. Tienda de María

Las ventanas son móviles y redimensionables (ilustración 12): cualquier panel puede arrastrarse manteniendo pulsado sobre su marco —resaltado en ámbar— y la ventana de diálogo y la ficha de personaje pueden además cambiar de tamaño desde su esquina inferior derecha, de modo que el jugador organiza la pantalla a su gusto sin que los paneles se solapen.

Ilustración 12. Ventanas redimensionables y ordenables

3.5. Desarrollo de los personajes y sus relaciones

En el pueblo podemos encontrar cuatro agentes de oficios complementarios, cargados desde un fichero de configuración inicial. **María** es la comerciante y mantiene existencias en el mercado; **Tomás** (panadero), **Klaus** (herrero) e **Isabella** (alquimista) son productores que compran materiales y fabrican sus productos en su taller. Esta complementariedad es la que genera un mercado: unos necesitan lo que otros producen.

3.5.1. Cómo se define un personaje

Cada agente se describe con un conjunto de campos que configuran su identidad, su situación y su comportamiento. Esta ficha inicial es la que, más adelante, se convierte en el texto que se le inyecta al modelo (el «bloque de identidad»).

Campo	Qué define
id / name	Identificador interno y nombre visible del personaje (p. ej. isabella / Isabella Rodríguez).
role	Oficio: panadero, herrero, alquimista o comerciante. Determina qué produce y qué materiales necesita.
traits	Rasgos de carácter (p. ej. «empática, curiosa, reservada») que tiñen su forma de hablar y decidir.
backstory	Trasfondo breve que da coherencia a sus motivaciones.
skills	Habilidades con nivel y experiencia (p. ej. Alquimia basica nv1 — exp 2/5). Suben con la práctica: cada producto fabricado da un punto de experiencia, con umbrales incrementales fijados en el código (5, 7, 9…). El nivel 2 desbloquea la receta avanzada del oficio (véase 3.5.4). Preparadas para subir de nivel con especialización en la evolución futura (H5).
goals	Objetivos actuales del personaje (p. ej. «encontrar cristal de luna»). No son eternos: al reflexionar, el personaje puede dar una meta por cumplida y sustituirla por otra coherente, validada por el código (máximo tres, frases cortas; véase 3.8.3).
money	Monedas de las que dispone para comerciar.
relationships	Afinidad (de −1 a +1) hacia cada vecino y hacia el jugador.
location / mood	Dónde está y su estado de ánimo. El ánimo solo mejora por ahora (los encargos completados lo suben) y se restablece a neutral al empezar cada día (véase 3.5.3). Se trabajará de manera dinámica en el futuro H6

Tabla 3. Ficha de personaje

A modo de ejemplo, esta es la ficha inicial de Isabella, la alquimista:

Ficha inicial (data/agents_seed.json)

```
{
  "id": "isabella", "name": "Isabella Rodríguez",
  "role": "alquimista",
  "traits": ["empática", "curiosa", "reservada"],
  "backstory": "Boticaria y alquimista. Busca ingredientes raros
```

```
                para curar a su abuela enferma.",
  "location": "botica", "money": 45,
  "skills": [{"name": "alquimia_basica", "level": 1}],
  "goals": ["preparar una pócima curativa", "encontrar cristal de luna"],
  "relationships": {"tomas": 0.2, "klaus": -0.1, "maria": 0.0, "player": 0.0}
}
```

Extracto de Código 2. Configuración inicial de NPC

3.5.2. El bloque de identidad inyectado al modelo

Antes de cada llamada al modelo (diálogo, decisión, reflexión…), la ficha del agente se transforma en un texto de identidad que encabeza el prompt. Es lo que le recuerda al modelo «quién es» el personaje. A partir de la ficha anterior, el bloque generado para Isabella sería:

Bloque de identidad (persona_block)

```
Eres Isabella Rodríguez, alquimista del pueblo.
Rasgos: empática, curiosa, reservada.
Historia: Boticaria y alquimista. Busca ingredientes raros para curar a su abuela enferma.
Habilidades: alquimia_basica(nv1).
Objetivos actuales: preparar una pócima curativa; encontrar cristal de luna.
Estado de ánimo: neutral.
Ubicación: botica.
```

Extracto de Código 3. Prompt de identidad

3.5.3. Cómo evolucionan las relaciones

Cada pareja de personajes —y cada personaje con el jugador— mantiene un valor de afinidad entre −1 y +1, que evoluciona con las interacciones (por ejemplo, sube al completar una misión) y que influye tanto en el tono de las conversaciones como en los precios: un vendedor cobra menos a quien aprecia. Para los prompts, ese número se traduce a una etiqueta legible, de modo que el modelo «entienda» el vínculo:

Afinidad	Etiqueta que recibe el modelo
≥ +0,50	«confías y aprecias a esta persona»
+0,15 a +0,50	«te cae bien»
−0,15 a +0,15	«es neutral para ti»
−0,50 a −0,15	«te resulta antipática»
≤ −0,50	«desconfías mucho de esta persona»

Tabla 4. Tabla de afinidad de personajes

Así, las relaciones no son un adorno narrativo, sino un parámetro con efecto en el diálogo y en la economía.

La influencia en la economía es directa y cuantificable: el precio de venta se calcula como el valor base del objeto multiplicado por un factor que depende de la afinidad del vendedor hacia el comprador, precio = valor × (1 − 0,2 × afinidad), con un mínimo de una moneda. Como la afinidad se mueve entre −1 y +1, el resultado es un descuento de hasta el 20 % para alguien de plena confianza y un recargo de hasta el 20 % para alguien de quien se desconfía. Por ejemplo, una herramienta de valor 20 le cuesta 18 monedas a un buen amigo (afinidad +0,5) y 22 a alguien con quien la relación es mala (−0,5). Así, cultivar la relación con María —completando sus encargos, por ejemplo— tiene una recompensa económica tangible.

El precio según la afinidad (game/economy.py)

```
def price_of(item, afinidad=0.0):
    """Amigo, más barato; enemigo, más caro (hasta un ±20 %)."""
    factor = 1.0 - 0.2 * afinidad        # afinidad en [-1, +1]
    return max(1, round(item.value * factor))
```

Extracto de Código 4. Cálculo del precio de los items según afinidad

El estado de ánimo participa del mismo circuito. Cuando el jugador completa un encargo, el personaje que lo pidió no solo mejora su afinidad: su ánimo sube a «contento» —y a «muy contento» si se encadenan varios—, y ese ánimo entra en todos sus prompts, tiñendo su forma de hablar y de decidir hasta que el nuevo día lo devuelve a neutral. De momento el ánimo solo puede subir: queda para hitos futuros que también baje de forma justificada —una mala contestación del jugador o rechazar un encargo sin motivo (mecánica que hoy no existe)—; se ha preferido acotar el mecanismo a la mejora antes que introducir penalizaciones difíciles de calibrar.

Estas etiquetas se inyectan en todos los diálogos: en la conversación con el jugador (la afinidad hacia él) y también en las conversaciones entre NPC, donde el prompt recibe la afinidad de cada uno hacia el otro para que el tono resulte cordial, tenso o distante según corresponda.

Personaje	Oficio	Papel en el mercado
María	Comerciante	Mantiene el stock de materiales del pueblo (nunca revende productos fabricados); compra pan y pócimas para sus propias necesidades.
Tomás	Panadero	Compra harina (y azúcar desde el nivel 2) y produce pan; todo el pueblo le compra para comer.
Klaus	Herrero	Compra madera y hierro y produce hachas (nivel 1) y espadas (nivel 2), caras a propósito: vive de vendérselas al jugador.
Isabella	Alquimista	Compra hierbas (y cristal de luna desde el nivel 2) y produce pócimas; el pueblo le compra para reponerse del cansancio.

Tabla 5. Tabla de personajes

En la siguiente ilustración, se puede apreciar como María es el eje comercial, mientras el resto de NPCs tiene un contacto mucho más secundario.

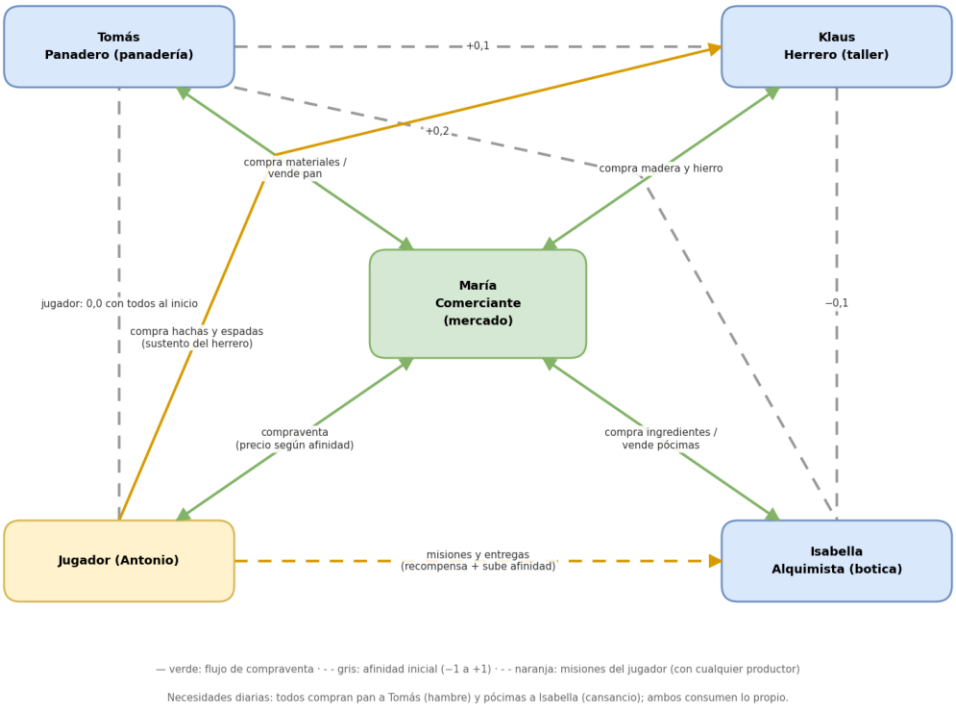

Ilustración 13. Esquema de relaciones entre los personajes. Afinidades entre parejas y flujos de compraventa.

3.5.4. Evolución de los oficios

Las habilidades de los personajes no son estáticas: evolucionan con la práctica. Cada vez que un personaje fabrica un producto en su taller gana un punto de experiencia en la habilidad de su oficio y, al alcanzar el umbral del nivel actual, sube de nivel. El umbral es incremental y está definido en el código, fuera del alcance del modelo:

- pasar del nivel 1 al 2 cuesta 5 creaciones; del 2 al 3, 7; del 3 al 4, 9 (en general, 2·nivel + 3).

La producción se realiza en lotes: hasta tres unidades por turno, guardándose siempre un juego de materiales de reserva (si solo hay para una, se fabrica esa); cada unidad aporta un punto de experiencia. Así el taller avanza sin desperdiciar pasos de simulación en fabricar de uno en uno.

Experiencia y lotes (game/items.py, extracto)

```
def xp_needed(level):
    return 2 * level + 3                # 5, 7, 9, 11... fijado en código


sets  = min(_count(agent.id, m) for m in materiales)
batch = sets if sets == 1 else min(3, sets - 1)    # guarda una reserva
for _ in range(batch):
    consumir_materiales()
    crafted.append(make_item(producto, agent.id))
    skill.xp += 1
    if skill.xp >= xp_needed(skill.level):
        skill.level += 1                # sube: se desbloquean recetas
        skill.xp = 0
```

Extracto de Código 5. Subida de nivel de profesiones

El nivel no es decorativo: las recetas avanzadas de cada oficio —el pastel, la espada y el elixir— exigen nivel 2, de modo que al principio el pueblo solo produce bienes básicos y, con

los días, la especialización se hace visible en el propio mercado. La subida de nivel genera un evento en el registro y un recuerdo importante en el personaje («he mejorado mi oficio»), y su ficha muestra el progreso como una barra de experiencia (por ejemplo, herrería nv 1 — exp 3/5). Este mecanismo constituye la primera pieza del hito H5 (capítulo 6) y conecta con el aprendizaje continuo tratado en la línea H5 (capítulo 6).
Los personajes son, además, conscientes de su nivel: solo compran los materiales de las recetas que ya dominan y, si el jugador les encarga un producto avanzado, responden con naturalidad que aún les falta práctica.

3.5.5. Economía de necesidades: hambre, cansancio y crisis

El cierre del ciclo económico llega con las necesidades diarias. A partir del segundo día —el primero queda libre de necesidades, para que el pueblo arranque su vida con normalidad—, los personajes amanecen con hambre y, desde media tarde, acumulan cansancio. El hambre se cubre comiendo pan y el cansancio con una pócima: quien no produce su propio remedio va a comprarlo —el pan a la panadería, la pócima a la botica, al precio que marque la afinidad del vendedor— y aprovecha el viaje para llevarse reserva (hasta dos panes y una pócima extra) si puede pagarla sin comprometer su bolsillo, de modo que los días siguientes come de casa. La compra se hace en el local del oficio —el mostrador despacha aunque el dueño ande fuera reponiendo materiales— o allí donde comprador y vendedor coincidan. Así el dinero, que antes solo fluía de los productores hacia la comerciante, regresa a los oficios que alimentan y reponen al pueblo. Un personaje con una necesidad sin cubrir no trabaja, con una excepción de sentido común: el panadero hambriento sí puede hornear y la alquimista cansada sí puede destilar, porque trabajar es precisamente su remedio.
El herrero es un caso de diseño diferente: nadie del pueblo necesita sus hachas (nivel 1, de madera) ni sus espadas (nivel 2, de hierro), caras a propósito —60 y 120 monedas—, así que su sustento depende del jugador: comprarle una pieza le financia varios días, y mantenerlo a flote se convierte en un objetivo económico continuo de la partida.
Si un personaje arrastra una necesidad de un día para otro o si, al caer la tarde, sigue sin poder pagársela, entra en crisis: cancela los encargos que tuviera abiertos, genera una misión de auxilio pidiendo al jugador exactamente lo que le falta (pan, pócima o un pequeño fondo de monedas, o las tres cosas) y no se mueve de donde está hasta que se la resuelvan. La crisis se construye íntegramente por reglas, sin ninguna llamada al modelo: es el mecanismo de seguridad del pueblo y debe ser fiable al cien por cien. Completarla no da monedas —la recompensa es la gratitud, una mejora notable de la afinidad (+0,15)— y lo entregado se consume en el acto.

La crisis, por reglas (game/needs.py y game/quests.py, extracto)

```
def in_trouble(agent, arrastradas, minuto_del_dia):
    if agent.id in arrastradas:          # necesidad de ayer sin cubrir
        return True
    if minuto_del_dia < 19 * 60:         # de día aún hay margen
        return False
    for flag, producto, oficio, _ in NEEDS:
        if getattr(agent, flag) and agent.role != oficio:
            if agent.money < _min_price(producto):
                return True              # no puede pagarse la vida
    return False
# la misión de auxilio pide EXACTAMENTE lo que falta:
if agent.hunger:     objs.append(Objective("bring_item", "pan"))
if agent.tired:      objs.append(Objective("bring_item", "pocima"))
if agent.money < 30: objs.append(Objective("bring_money", qty=30))
```

Extracto de Código 6. Configuración de entrada de NPC en crisis

Para comerciar y poder ayudar, cada personaje ofrece una tienda accesible desde su ficha: la comerciante vende sus materiales —nunca los productos que compra para su propio consumo— y cada productor vende únicamente lo que fabrica su oficio, siempre al precio ajustado por su afinidad hacia el comprador. El jugador comienza la partida con 200 monedas.

3.6. Toma de decisiones del agente

En cada paso de la simulación, cada agente decide qué hacer. La decisión no está escrita en el código: se delega en el modelo, al que se le presenta la situación del personaje y un conjunto acotado de acciones posibles, y se le pide que responda en JSON. El código solo prepara el contexto y aplica la acción devuelta; el modelo aporta el criterio, sensible a la personalidad, los objetivos y los recuerdos del agente.

Para decidir, se le entrega al modelo: su bloque de identidad, la hora, las localizaciones disponibles, dónde está cada vecino, dónde está el jugador y qué relación tiene con él, los materiales que le faltan (solo los de las recetas que su nivel domina), su nivel de oficio y sus necesidades pendientes —hambre o cansancio, con la instrucción de priorizarlas— y su dinero, y sus recuerdos recientes. Además, se le recuerda una «regla de rutina» (si falta material, ir al mercado; si no, volver al taller a producir; de vez en cuando, socializar). El agente responde si se mueve o se queda, a dónde y con qué intención. Este es el prompt real de decisión:

Prompt de decisión de acción (decide_action)

```
{bloque_de_identidad}

Son las {hora}. Localizaciones: {lista}.
Tu sitio de trabajo es '{casa}'. Estás en '{localización}'. Aquí contigo: {otros}.
Tu nivel de oficio: {nivel} (exp {x}/{umbral}).
Dónde está cada vecino: {posiciones}.
El jugador ({nombre}) está en '{loc_jugador}' (tu relación con él: {afinidad}).
Si te apetece, puedes acercarte a donde está para saludarle o pedirle algo.
Materiales que te faltan y querrías comprar hoy: {faltantes}. Tienes {monedas} monedas.
Recuerdos recientes: {recuerdos}.
[Si procede] Tienes HAMBRE: prioriza ir a la panadería a comprar pan.
[Si procede] Estás CANSADO: prioriza ir a la botica a por una pócima.

REGLA DE RUTINA: si te falta algún material, ve al MERCADO; si no, vuelve a tu sitio
de trabajo ('{casa}') a producir; no insistas en el mercado el mismo día.
De vez en cuando ve a la plaza a socializar o a ver al jugador.
Devuelve JSON: {"action": "move" o "stay",
"target_location": "<localización válida si move>",
"intent": "<motivo en una frase>"}.
```

Extracto de Código 7. Prompt decisión de acciones

El resultado es un JSON como {"action": "move", "target_location": "mercado", "intent": "necesito comprar cristal de luna"}, que el código valida (que la localización exista) y ejecuta, moviendo al personaje y registrando el evento. La economía que se dispara al llegar (comprar, esperar material o producir) es determinista y no pasa por el modelo.

Así se piden y se validan las decisiones en el código: el modelo propone en paralelo, el código comprueba y ejecuta.

Decisión en paralelo y validación (agents/simulation.py, extracto)

```
if config.PARALLEL_DECISIONS:            # una espera para N agentes
    decisions = await asyncio.gather(*(
        decide_action(a, agents, occ, locations, hhmm, now, player)
```

```
        for a in agent_list))

for a, d in zip(agent_list, decisions):
    if d.get("action") == "move":
        tgt = d.get("target_location")
        if tgt in locations and tgt != a.location:   # VALIDACIÓN
            a.location = tgt
            store.save_agent(a)          # solo entonces se ejecuta
```

Extracto de Código 8. Decisiones en serie o paralelo

En la ilustración siguiente se representa el Flujo de toma de decisiones y cómo el contexto del agente condiciona el prompt con el que se construye el JSON {action, target, intent} para luego validar finalmente las acciones y mover el mercado.

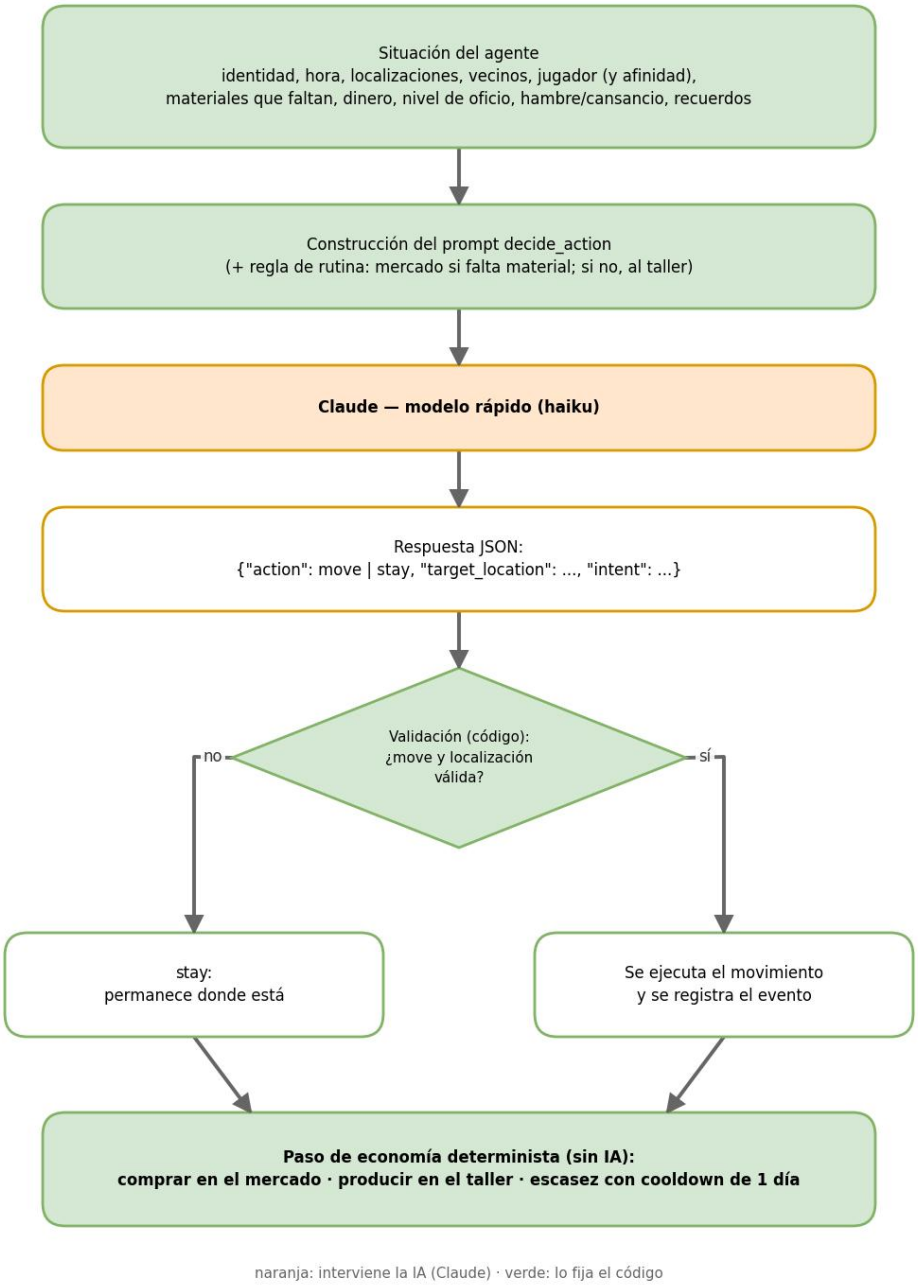

Ilustración 14. Flujo de toma de decisiones

3.7. Máquina de estados del comportamiento del agente

El comportamiento de cada agente se puede describir como una máquina de estados que se evalúa en cada paso de la simulación. A partir de su situación —hora, localización, materiales que le faltan, dinero y recuerdos recientes— el agente decide una acción y transita entre los siguientes estados:

Estado	Descripción y transiciones
Decidir	Estado de entrada de cada paso. El agente evalúa su situación (mediante el modelo) y elige a dónde ir o si quedarse. Transita a Desplazarse o permanece para Comprar / Producir / Conversar.
Desplazarse	Se mueve hacia la localización objetivo (mercado, taller, plaza). Al llegar, pasa al estado correspondiente.
Comprar en el mercado	Si está en el mercado, le falta material y hay stock y dinero, adquiere el material (cambio de dueño y de saldo). Si no hay disponibilidad, pasa a «En espera de material».
En espera de material	El material que necesita no está disponible; el agente lo anota y aplica un enfriamiento de un día para no insistir. Aquí se registra DE QUIÉN espera el material (del proveedor correspondiente: normalmente María, la comerciante, o el productor que lo fabrica). Al día siguiente vuelve a Decidir/Comprar.
Producir	En su sitio de trabajo y con los materiales necesarios, fabrica su producto (receta determinista).
Conversar	Si coincide con otro agente en una localización, pueden conversar; el resumen se guarda como recuerdo en ambos. También cubre el diálogo con el jugador.
Cubrir necesidades	Con hambre o cansancio acude a la panadería o la botica, compra su remedio a precio de afinidad y se lleva reserva si le sobra. Con una necesidad sin cubrir no produce (salvo que producir sea su remedio).
En crisis	Si no puede pagar una necesidad al caer la tarde, o la arrastra un día entero, cancela sus encargos, crea una misión de auxilio para el jugador y queda inmóvil hasta que se la completen.
Reflexionar	Al final del paso, cuando la suma de importancias de sus recuerdos nuevos supera un umbral (criterio del artículo original), sintetiza una reflexión de alto nivel y la almacena como nuevo recuerdo.

Tabla 6. Definición de estados posibles

Podemos visualizar la máquina de estados del agente en la ilustración 15, definiendo las casuísticas que se pueden dar y en qué momento se producirán eventos en base al estado del jugador en ese instante.

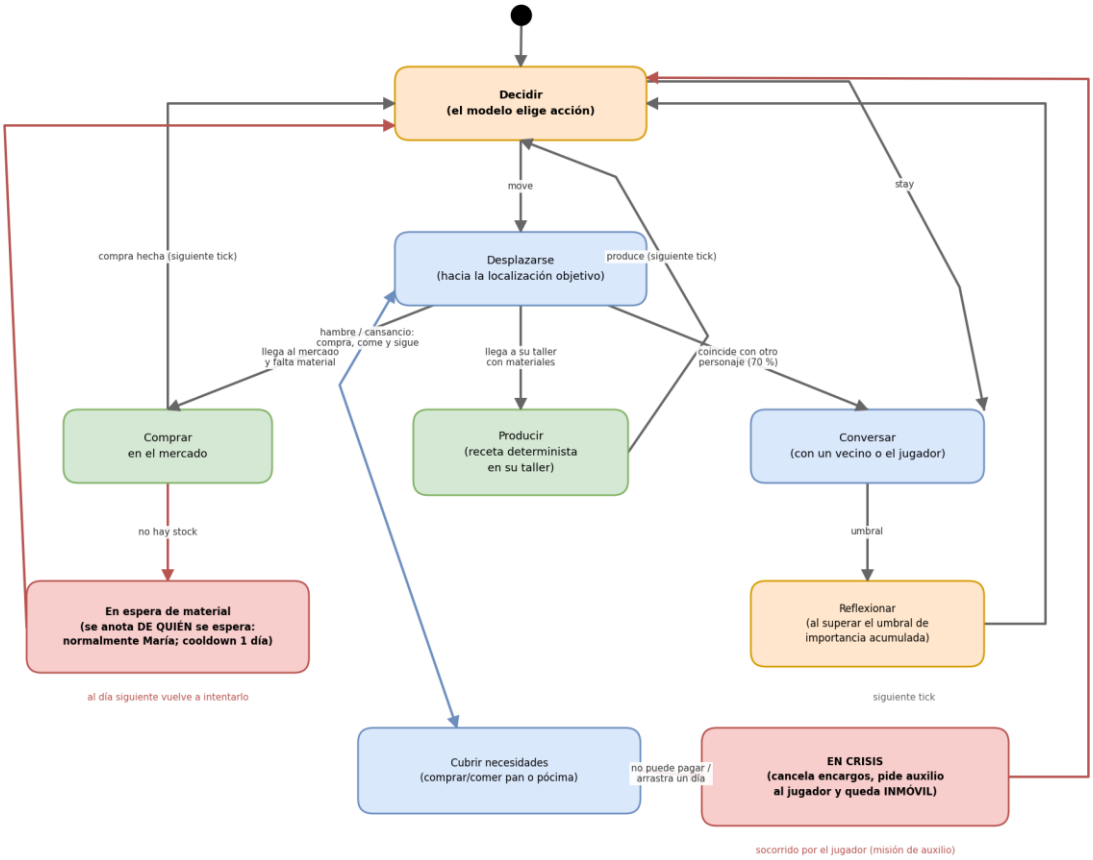

Ilustración 15. Máquina de estados de los personajes

3.8. Memoria: guardar, recuperar y reflexionar

La memoria es el corazón heredado del artículo de Park et al. y la pieza que hace que los personajes sean coherentes en el tiempo. Cada agente dispone de una memoria episódica implementada como un flujo de recuerdos indexado en la base de datos vectorial ChromaDB.

3.8.1. Qué es un recuerdo y cómo se guarda

Al ocurrir algo relevante, se crea un recuerdo: se calcula su embedding (un vector de 384 dimensiones, con el modelo local all-MiniLM-L6-v2) y se guarda el texto junto con unos metadatos. Los tipos de recuerdo que se generan son de diálogo (con el jugador y entre NPC), de reflexión, y de la vida de las misiones (plan, observación y transacción).

Metadato	Significado
content	El texto del recuerdo (lo que se busca semánticamente).
embedding	Vector de 384 dimensiones del content, para la búsqueda por significado.
type	dialogue, reflection, plan, observation, transaction…
created_at	Momento (minuto de juego) en que se creó.
importance	Importancia de 1 a 10 (una charla trivial vale poco; cumplir un encargo, mucho).

Tabla 7. Configuración de recuerdos

No solo las conversaciones dejan huella: también los hechos económicos generan recuerdos —comprar un material, fabricar un producto o no encontrar algo en el mercado—, de modo que un personaje puede contar después qué compró, qué fabricó o qué le faltó. Estos recuerdos se crean sin ninguna llamada al modelo de lenguaje: solo requieren su embedding, calculado en local con coste cero.
El codificador local, citado a lo largo de toda la memoria, es deliberadamente pequeño: se carga una sola vez (perezosamente o en el warm-up del arranque) y convierte cualquier texto en un vector normalizado de 384 dimensiones, sin ninguna llamada de red ni coste por uso. Este módulo, junto con ChromaDB, es lo que permite que cada suceso del pueblo deje recuerdo sin encarecer la simulación.

El codificador local (core/embeddings.py, extracto)

```
_model = None

def _get_model():
    global  model
    if  model is None:               # carga perezosa (~90 MB, UNA vez)
        from sentence transformers import SentenceTransformer
         model = SentenceTransformer("all-MiniLM-L6-v2")
    return _model

def embed(text):
    return  get model().encode([text],
                               normalize_embeddings=True)[0].tolist()
```

Extracto de Código 9. Codificación a modelo all-MiniLM-L6-v2

Guardar un recuerdo (agents/memory_stream.py, extracto)

```
def store(agent_id, type_, content, created_at, importance=5):
    mem_id = str(uuid.uuid4())
    emb = embeddings.embed(content)      # MiniLM local, 384 dimensiones
    coleccion.add(ids=[mem_id], embeddings=[emb], documents=[content],
                  metadatas=[{"agent_id": agent_id, "type": type_,
                              "created_at": created_at,
                              "importance": importance,
                              "last_accessed": created_at}])
    return mem_id
```

Extracto de Código 10. Guardado de recuerdo

Para entender lo que es el vector de 384 dimensiones: los ordenadores no entienden las palabras ni el contexto como los humanos, por lo que necesitan traducir el lenguaje a números. El modelo toma un texto (por ejemplo, un diálogo o una observación de una misión) y lo transforma en una lista de 384 números flotantes (el vector).

- **Captura el significado, no las palabras exactas**: Si un NPC genera el recuerdo "Tengo mucha hambre" y otro recuerda "Necesito buscar algo de comer", sus vectores serán muy similares numéricamente, aunque no compartan casi ninguna palabra. El modelo "sabe" que conceptualmente significan casi lo mismo.
- **Búsqueda semántica** (Memoria del agente): En el sistema de agentes, esto sirve para que cuando ocurra algo nuevo, el backend pueda calcular el vector del evento actual y compararlo matemáticamente (usando la similitud de coseno) con los vectores guardados en la base de datos. Así, el agente puede "recordar" de forma inmediata los diálogos, reflexiones o misiones pasadas que estén más relacionadas con lo que está viviendo en ese momento.
- **Por qué 384**: Es el tamaño estándar de salida de ese modelo específico (all-MiniLM-L6-v2). Es un modelo muy popular porque es ligero, rápido (ideal para correr en local) y requiere poca memoria, manteniendo una precisión excelente para comparar textos cortos.

3.8.2. Cómo se recupera (recencia + importancia + relevancia)

Cuando el agente necesita contexto —por ejemplo, para responder al jugador— no se le pasan todos sus recuerdos, sino los más pertinentes. Siguiendo el esquema del artículo, cada recuerdo candidato se puntúa combinando tres factores y se eligen los de mayor puntuación (por defecto, los 5 mejores):

Fórmula de puntuación de recuerdos

```
score = 1.0 · relevancia + 0.5 · recencia + 0.3 · importancia


  relevancia = 1 − distancia_coseno(embedding_consulta, embedding_recuerdo)
  recencia   = 0.995 ^ (horas_de_juego_desde_el_último_acceso)
  importancia= importance / 10
```

Extracto de Código 11. Puntuación de recuerdos

Dos matices de la implementación, tomados directamente del artículo original: la recencia se mide desde el último acceso al recuerdo —cada recuperación lo «refresca», de modo que lo que se consulta a menudo se mantiene disponible— y decae por horas de juego, con la unidad configurable (RECENCY_UNIT_MINUTES). En una versión anterior decaía por minuto de juego, lo que con pasos de dos horas anulaba la recencia en un solo día; el ajuste devuelve a la fórmula el comportamiento previsto por el paper.
El núcleo completo de la recuperación cabe en una docena de líneas:

La fórmula, en el código (agents/memory_stream.py, extracto)

```
res = coleccion.query(query_embeddings=[q_emb], n_results=k*4,
                      where={"agent_id": agent_id})
scored = []
for mem_id, doc, meta, dist in zip(ids, docs, metas, dists):
    relevancia  = 1.0 - dist                      # similitud coseno
    edad        = ahora - meta["last_accessed"]   # desde el ÚLTIMO acceso
    recencia    = 0.995 ** (edad / 60)            # decae por horas de juego
    importancia = meta["importance"] / 10.0
    score = 1.0*relevancia + 0.5*recencia + 0.3*importancia
    scored.append((score, mem_id, doc, meta))
scored.sort(reverse=True)
top = scored[:k]
coleccion.update(ids=..., metadatas=...)  # recuperar "refresca" el recuerdo
```

Extracto de Código 12. Recuperación de recuerdos del NPC

La relevancia mide el parecido semántico con lo que se está consultando; la recencia decae suavemente con el tiempo (los recuerdos viejos pesan menos, pero no desaparecen); y la importancia prioriza lo que de verdad marcó al personaje. Es exactamente el mecanismo que hace que un NPC «recuerde» lo que importa en cada situación.

3.8.3. Reflexión

Además de acumular hechos, los agentes sintetizan conclusiones. Siguiendo el criterio de disparo del artículo original, un agente reflexiona cuando la suma de importancias de los recuerdos acumulados desde su última reflexión supera un umbral configurable (REFLECT_IMPORTANCE_THRESHOLD, 25 por defecto; en el paper el umbral es 150, acorde a sus decenas de eventos diarios). Así reflexiona más quien más vive, en lugar de depender del azar. Cuando se dispara, el agente toma sus recuerdos recientes y genera una reflexión de alto nivel, que se guarda como un nuevo recuerdo (de importancia media-alta) y puede influir en decisiones y diálogos posteriores.

La reflexión cumple además una segunda función, sin coste adicional al viajar en la misma llamada: revisar los objetivos del personaje. El modelo comprueba si alguna de sus metas se ha cumplido según lo vivido —la pócima de la abuela por fin preparada, el taller por fin reformado— o si lleva tanto tiempo estancada que conviene reformularla, y en ese caso la sustituye por una meta nueva coherente con su historia; el código valida la respuesta (máximo tres objetivos, una frase cada uno, y solo si el cambio viene justificado) y la renovación queda registrada como recuerdo y como evento. Así los personajes no viven en un eterno primer día: sus problemas pueden resolverse y dar paso a otros.

El disparo por umbral (agents/reflect.py y memory_stream.py, extracto)

```
def should_reflect(agent_id):
    return importance_since_last_reflection(agent_id) >= 25   # umbral

def importance_since_last_reflection(agent_id):
    metas = coleccion.get(where={"agent_id": agent_id})["metadatas"]
    ultima = max((m["created_at"] for m in metas
                  if m["type"] == "reflection"), default=-1)
    return sum(m["importance"] for m in metas
               if m["created_at"] > ultima and m["type"] != "reflection")
```

Extracto de Código 13. Reflexión tras superar el umbral

Prompt de reflexión (reflect_agent)

```
{bloque_de_identidad}

Tus recuerdos recientes:
- {recuerdo_1}
- {recuerdo_2}
- ... (hasta 8)

Sintetiza UNA reflexión de alto nivel, en primera persona y una sola frase,
sobre lo que está pasando o sobre alguien.
Devuelve JSON: {"reflection": "..."}.
```

Extracto de Código 14. Prompt de la reflexión del npc

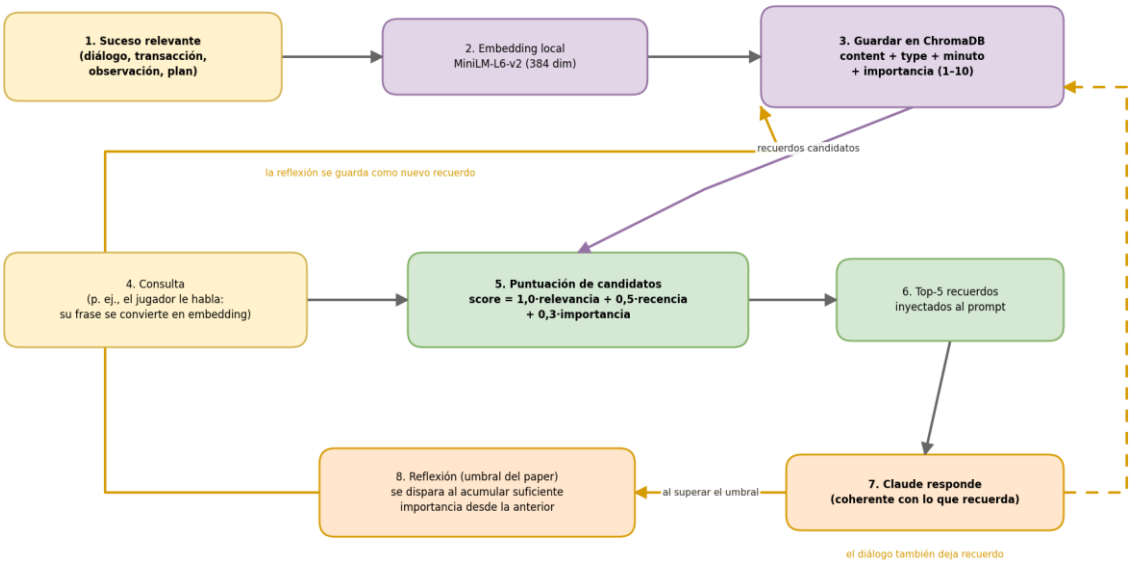

Ilustración 16. Ciclo de memoria

En la ilustración 16, observamos el iclo de memoria de un agente, desde guardar (embedding) hasta la posibilidad de la reflexión.

3.8.4. Diferencias deliberadas respecto al artículo original

Más allá del cambio de plataforma descrito en el capítulo 1 (Unity, Claude, ChromaDB), la implementación se aparta del artículo de Park et al. en varios puntos concretos, siempre por una de dos razones: abaratar el coste por paso de simulación o ganar fiabilidad en un contexto de juego. Conviene dejarlas explícitas:

- Se conserva fielmente el núcleo cognitivo del artículo: la fórmula de recuperación (relevancia + recencia + importancia), la recencia medida desde el último acceso y por horas, y el disparo de la reflexión por suma de importancias acumuladas.

- Decisiones de coste. Los embeddings se calculan en local con MiniLM (coste cero por recuerdo y funcionamiento sin conexión), mientras que el artículo depende de una API de embeddings de pago.

- La conversación entre dos NPC se genera en una única llamada que devuelve el diálogo completo y su resumen; el artículo la construye turno a turno, con una llamada por intervención.

- La importancia de los recuerdos rutinarios se asigna por reglas según el tipo de evento, reservando la valoración mediante el modelo para el diálogo con el jugador (el artículo puntúa cada recuerdo con una llamada adicional). La reflexión sintetiza una sola frase, frente al proceso en dos fases del artículo (preguntas destacadas y después conclusiones con citas), que cuesta dos o tres llamadas.

- Y se emplean modelos distintos según la tarea: el rápido para el bucle de simulación y el de mayor calidad solo donde el jugador lo percibe directamente, su propio diálogo.

- Decisiones de fiabilidad. Las salidas del modelo se restringen a JSON validado contra listas cerradas (localizaciones, catálogo del armario), de modo que una respuesta mal formada no puede corromper el estado del juego; el artículo interpreta acciones en lenguaje libre sobre un árbol de entorno. Además, cada diálogo se ancla en los datos reales del personaje (inventario, dinero y encargos pendientes), un mecanismo que el artículo no incluye y que aquí elimina la principal fuente de incoherencias observada durante el desarrollo.

3.9. Ejemplo real: de la memoria al prompt

Para ver cómo encaja todo, seguimos un caso concreto. Isabella, la alquimista, necesita cristal de luna para su pócima; días atrás dejó el encargo y el jugador (Antonio) se ofreció a traérselo. En su memoria hay, entre otros, estos recuerdos:

Contenido de la memoria de Isabella (extracto)

```
[plan | imp 6]    Necesito cristal de luna y he dejado el encargo de que
                  alguien (quizá Antonio) me lo traiga.
[observation|6]   Antonio se ha ofrecido a traerme cristal de luna.
[reflection|6]    Debería confiar más en quien me ayuda a cuidar de mi abuela.
[dialogue | 4]    Hablé con María: me contó que subirá el precio de las hierbas.
```

Extracto de Código 15. Extracto de memoria de Isabella

Cuando Antonio se acerca y le dice «Isabella, ya tengo tu cristal de luna», el sistema convierte esa frase en un embedding, recupera los recuerdos con mayor puntuación (los relacionados con el cristal y el encargo suben; el de María, no) y los inserta en el prompt de diálogo. El prompt que recibe el modelo queda así:

Prompt de diálogo con el jugador (talk_to_player), ya rellenado

```
Eres Isabella Rodríguez, alquimista del pueblo.
Rasgos: empática, curiosa, reservada.
Historia: Boticaria y alquimista. Busca ingredientes raros para curar a su abuela enferma.
Habilidades: alquimia_basica(nv1).
Objetivos actuales: preparar una pócima curativa; encontrar cristal de luna.
Estado de ánimo: neutral.
Ubicación: botica.

Sobre Antonio (el jugador): te cae bien (afinidad +0.20).
Lo que recuerdas relacionado con lo que te dice:
- Necesito cristal de luna y he dejado el encargo de que alguien me lo traiga.
- Antonio se ha ofrecido a traerme cristal de luna.
- Debería confiar más en quien me ayuda a cuidar de mi abuela.

Antonio te dice: "Isabella, ya tengo tu cristal de luna"

Responde como Isabella Rodríguez, en español, en 1-3 frases. Sé coherente con tu
personalidad y con tus recuerdos; no inventes recuerdos que los contradigan.
Habla en primera persona, sin describir acciones entre asteriscos.
```

Extracto de Código 16. Prompt de diálogo con el jugador

Con ese contexto, el modelo responde algo como: «*¡Antonio, no sabes cuánto te lo agradezco! Ese cristal es justo lo que me faltaba para la pócima de mi abuela*.» La respuesta es coherente porque el personaje «recuerda» el encargo y el ofrecimiento previos, no porque estuviera escrito de antemano. Esta trazabilidad —qué hay en la memoria y qué acaba en el prompt— es la que reduce las incoherencias frente a un modelo sin memoria estructurada.

El prompt real incluye además el bloque de datos reales del personaje —inventario, dinero, estado físico, nivel de oficio y precios base del mercado— descrito en las secciones anteriores; se omite aquí por brevedad.

Ese bloque de datos reales incluye también la lista de lo que el jugador ya entregó en encargos anteriores, con la instrucción de reconocer el consumo («me lo trajiste, sí, pero lo gasté al hornear») en lugar de pedirlo de nuevo como si nada. Durante las pruebas se comprobó que la sensación de «amnesia» de los personajes casi nunca era un fallo de memoria: era economía —los materiales entregados se consumen al producir— sin un relato que lo explicara.

Además de los recuerdos, cuando la charla tiene varios turnos el prompt incluye la transcripción de la conversación en curso (el tramo final, acotado): así el personaje es coherente no solo con lo que vivió días atrás, sino con lo que se acaba de decir dos frases antes, sin depender de que la recuperación semántica acierte con los turnos inmediatos.

3.10. Generación de misiones: decisión y sistema híbrido

Las misiones son emergentes: nacen de las necesidades de los NPC, no de un guion. Un principio de diseño las gobierna: el objeto pedido y la recompensa los fija SIEMPRE el código (a partir de las carencias reales del personaje y de un catálogo cerrado de objetos que el jugador puede conseguir en el «armario»), mientras que el texto de la misión lo redacta el modelo en la voz del personaje. Así se garantiza que toda misión sea coherente con la economía y, sobre todo, completable.

La decisión de crear o no una misión ocurre por dos vías complementarias, a las que se suma una tercera vía de emergencia —la misión de auxilio de las crisis (sección 3.5.5), construida íntegramente por reglas—:

3.10.1. Vía autónoma (durante la simulación)

En cada avance de tiempo, el sistema busca «candidatos»: agentes a los que les falta un objeto del catálogo del armario y que no tienen ya una misión abierta. Si hay candidatos, con una probabilidad del 60 % se elige uno y se crea la misión. Aquí la decisión de crear es por reglas (candidato + azar); el modelo solo redacta el título y la descripción, y si no hay modelo disponible, un texto por reglas actúa de red de seguridad.

3.10.2. Vía conversacional (al terminar una charla)

Cuando el jugador cierra una conversación, el NPC «relee» lo hablado y decide si encargarle algo. Aquí la decisión de crear sí la toma el modelo, pero de forma acotada: solo puede pedir objetos de una lista cerrada (el catálogo del armario) y el código valida que el objeto elegido pertenezca a esa lista; si no, no se crea. Este es el prompt que materializa esa decisión:

Prompt de misión desde conversación (quest_from_conversation)

```
{bloque_de_identidad}

Acabas de tener esta conversación con el jugador {nombre}:
{transcripción}

Decide si, A RAÍZ de esta charla, quieres ENCARGARLE que te traiga UN objeto.
Solo puedes pedir objetos de esta lista (usa su clave exacta): {catálogo_armario}.
Si de la charla no surge nada que encaje de forma natural, NO crees misión.
Si en la charla acordasteis una cantidad o un precio, RESPÉTALOS.
Devuelve SOLO JSON: {"crear": true|false, "item": "<clave de la lista o vacío>",
"cantidad": <entero 1-5: unidades que pides>,
"recompensa": <monedas TOTALES; si se acordó un precio en la charla, exactamente ese>,
"titulo": "<título corto, máx 6 palabras>",
"descripcion": "<1-2 frases en tu voz explicando el encargo>"}.
```

Extracto de Código 17. Prompt de creación de misiones desde conversación

Si el modelo responde "crear": false —o elige un objeto que no está en la lista— no se crea nada. Si responde "crear": true con un objeto válido, el código fija la recompensa (proporcional al valor del objeto), guarda la misión como ofrecida y registra un recuerdo del encargo. En ambas vías, por tanto, se combina el criterio del modelo con las garantías del código: es el sistema híbrido.

A nivel de código, esa garantía ocupa pocas líneas. El extracto siguiente (simplificado de game/quests.py) señala los tres puntos donde el código impone las reglas sobre lo que propone el modelo: la validación del objeto contra el catálogo cerrado, el cálculo de la recompensa a partir del valor real del objeto y el texto de reserva por si la redacción del modelo falla:

Validación y ejecución en el código (game/quests.py, extracto simplificado)

```
data = await llm.complete_json(prompt, model=MODEL_FAST)   # la IA propone
if not data.get("crear"):
    return None                    # el personaje decidió no encargar nada
item = (data.get("item") or "").strip()
if item not in ARMARIO_ITEMS:      # VALIDACIÓN: solo objetos del armario;
    return None                    # si la IA inventa un objeto, no se crea

qty = max(1, min(5, int(data.get("cantidad") or 1)))    # cantidad hablada (1-5)
cap = valor * qty * 4                          # techo: 4 veces el valor real
```

```
floor = max(1, round(valor * qty * 0.5))     # suelo: la mitad del valor real
reward = max(floor, min(cap, int(data.get("recompensa"))))  # respeta lo negociado
quest = Quest(
    giver=agent.id,
    title=(data.get("titulo") or f"Tráeme {display}")[:60],  # texto de reserva
    objectives=[Objective(type="bring_item", target=item, qty=qty)],
    rewards=Reward(money=reward, relationship=0.1),
    status="offered")
store.save_quest(quest)            # persistencia en SQLite
memory_stream.store(agent.id, "plan",
    f"Le he encargado {display} x{qty} por {reward} monedas.")  # recuerdo
```

Extracto de Código 18. Conversión de negociación dialogada en misión formal

El mismo patrón se repite en la vía autónoma (generate_quest), donde incluso el objeto lo elige el código (_pick_item: la intersección entre las carencias reales del personaje y el catálogo del armario) y el modelo solo redacta el título y la descripción. El cierre de la misión (complete) es íntegramente determinista: comprueba el inventario real del jugador, transfiere los objetos al NPC, abona la recompensa, sube la afinidad y deja en el NPC un recuerdo de alta importancia de que el jugador cumplió su palabra. Es exactamente el reparto anunciado en la introducción: el modelo decide y redacta; el código valida, ejecuta y mantiene el mundo consistente.

Este acotado se refinó con el juego real: el jugador negociaba con la comerciante una cantidad y un precio («cinco sacos de harina por 35 monedas») y la misión inicial ignoraba el acuerdo (creaba la misión con cinco sacos por 40 monedas, que era el precio base de la harina). La versión final desplaza la frontera del reparto: el modelo propone también la cantidad y la recompensa —tomándolas de lo hablado, y el diálogo conoce los precios base del mercado para negociar sobre valores reales—, mientras que el código las acota (cantidad entre 1 y 5; recompensa entre la mitad del valor real del pedido y cuatro veces ese valor). El suelo es la mitad, y no el valor exacto, porque el jugador puede querer aceptar un trato por debajo de mercado: es su decisión; lo que se bloquea es el precio absurdo fruto de una alucinación. La conversación manda; el código impide que rompa la economía.
La charla puede, por último, poner al personaje en movimiento: el mismo JSON admite un campo ir_a con el que el modelo indica que, a raíz de lo hablado, el personaje debe dirigirse a una localización («ve a ver a María, que tiene género» hace que el NPC parta hacia el mercado al cerrarse la ventana). El código valida el destino contra la lista de localizaciones, lo ignora si el personaje está en crisis y deja constancia en su memoria del porqué del viaje.

Un último matiz de la vía conversacional, surgido en las pruebas: si el personaje ya tenía un encargo normal abierto, el trato recién cerrado en la charla lo sustituye —el encargo antiguo se cancela dejando el recuerdo correspondiente («hemos renegociado») y se crea el nuevo—, porque la conversación es siempre la expresión más fresca de la intención del personaje. La única excepción son las misiones de auxilio de una crisis, que no se sustituyen bajo ningún concepto: la supervivencia manda sobre cualquier negocio.

	Vía autónoma	**Vía conversacional**
Cuándo	En cada paso de simulación	Al cerrar una conversación con el jugador
Quién decide crear	El código (candidato + 60 % de azar)	El modelo ("crear": true/false), acotado

	Vía autónoma	Vía conversacional
Objeto pedido	Lo fija el código (carencia ∩ armario)	Lo propone el modelo (objeto y cantidad 1-5), validado contra el armario
Texto (título/descr.)	Modelo (con texto por reglas de reserva)	Modelo
Recompensa	Código (2× el valor del objeto)	Propuesta en la charla; el código la acota entre 0,5× y 4× el valor del pedido
Garantía	Siempre completable	Siempre completable

Tabla 8. Vías de generación de acciones

En la ilustración 17 se encuentra la manera en la que los NPCs pueden generar misiones, ya sea a través de conversación, o por vía autónoma.

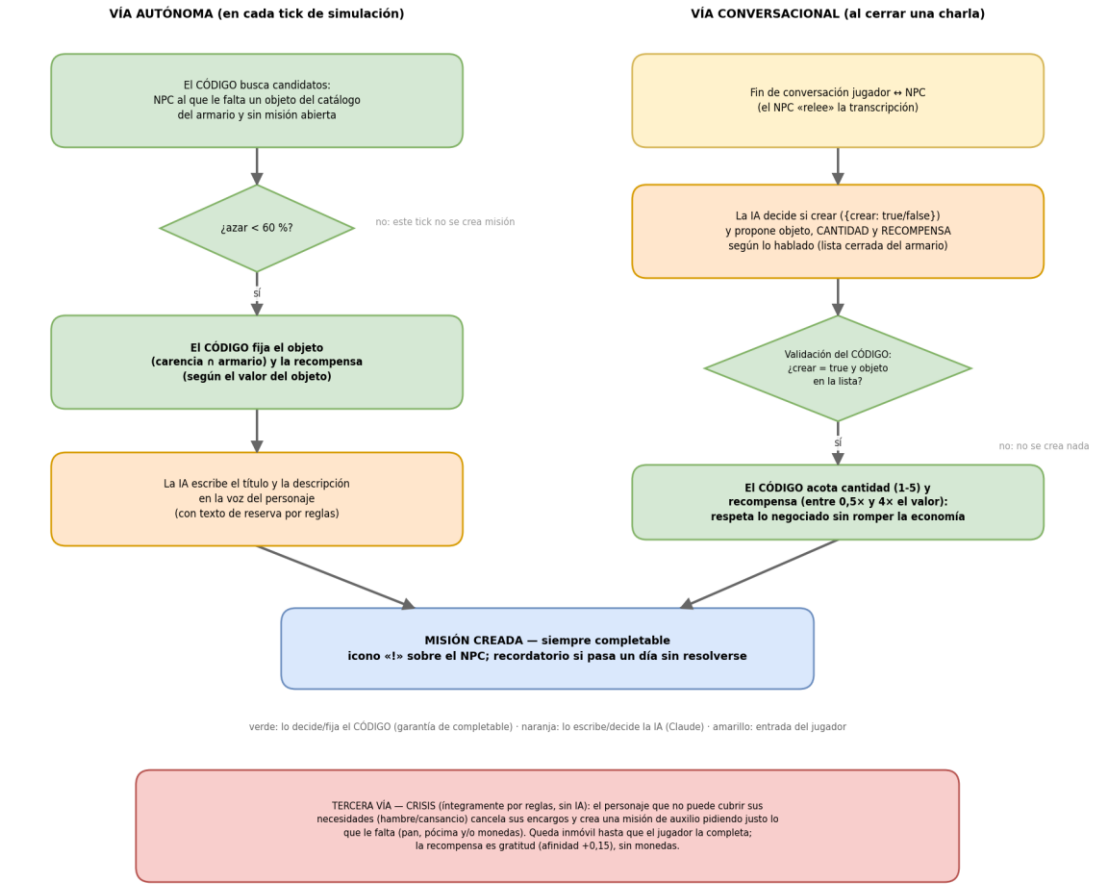

Ilustración 17. Generación de misiones (sistema híbrido)

3.11. Uso del modelo de lenguaje

Todas las llamadas al modelo pasan por un punto único que centraliza el cliente y admite salida de texto libre o JSON estructurado. Se usan distintos modelos según la tarea, buscando equilibrio entre coste, latencia y calidad: un modelo rápido para las decisiones y conversaciones autónomas de alta frecuencia, y uno de mayor calidad para el diálogo del NPC con el jugador. La economía (compra de materiales y producción) es determinista y no usa el modelo: se rige por reglas de stock, dinero y recetas, lo que garantiza que el mercado sea coherente y predecible.

Dónde	Qué decide el modelo
Diálogo NPC ↔ jugador	Respuesta del personaje al jugador, coherente con su personalidad y recuerdos recuperados (modelo de calidad).
Conversación NPC ↔ NPC	Breve intercambio autónomo entre dos vecinos que coinciden, con un resumen que se guarda como recuerdo.
Decisión de acción	Si el agente se mueve o se queda, y a qué localización, según su rutina y necesidades.
Reflexión	Una frase de síntesis de alto nivel a partir de los recuerdos recientes.
Texto de misión	Título y descripción de la misión en la voz del personaje (el objeto y la recompensa los fija el código).

Tabla 9. Uso del modelo de lenguaje

La excepción deliberada es la misión de auxilio de las crisis: se construye sin ninguna llamada al modelo, íntegramente por reglas, porque el mecanismo de seguridad del pueblo debe ser fiable al cien por cien.

Merece cuantificarse cuánto cuesta la autonomía del pueblo. Todas las llamadas del bucle de simulación usan el modelo rápido; el diálogo con el jugador, el de calidad; y todo lo demás —necesidades, compras, producción, niveles, crisis y memoria— funciona por reglas y embeddings locales, a coste cero. Con los tamaños de prompt reales del sistema y las tarifas vigentes al cierre de esta memoria (modelo rápido: 0,92€ por millón de tokens de entrada y 4,6€ por millón de salida; modelo de calidad: 2,76€ y 13,8 €), el coste por escenario queda así:

Escenario	Llamadas al modelo	Tokens (entrada / salida)	Coste aprox. (€)
Paso tranquilo: solo decisiones y movimientos	4	~1.800 / 240	0,28 céntimos
Paso con charla entre vecinos	5	~2.300 / 490	0,44 céntimos
Paso «movido»: charla + saludo + misión + reflexión	8	~3.300 / 810	0,68 céntimos
Mensaje del jugador en el chat (modelo de calidad + tasación del recuerdo)	2	~780 / 170	0,45 céntimos
Necesidades, compras, producción, niveles y crisis	0 (reglas y embeddings locales)	0 / 0	0 €
Día completo de juego (16 pasos, mezcla típica)	**~70**	**~40.000 / 8.000**	**~ 7,4 céntimos**
SEMANA de juego (7 días × 16 pasos de 1 hora)	**~500**	**~280.000 / 56.000**	**~ 0,55 €**

Tabla 10. Costes estimados de la simulación por escenario con 4 NPCs

La lectura es clara: una semana entera de vida del pueblo —unos 500 pensamientos, charlas y decisiones de los cuatro personajes— cuesta alrededor de medio euro. Es la validación económica del principio de diseño «modelo rápido para el bucle, modelo de calidad solo donde el jugador lo percibe»; una medición exacta requeriría registrar el campo de uso que la API devuelve en cada respuesta, lo que queda anotado como mejora menor.

4. Desarrollo del juego

Este capítulo aterriza la metodología anterior en el desarrollo concreto, siguiendo los hitos en que se organizó el trabajo.

4.1. H0 — Conexión

Primer hito de integración: establecer el circuito completo Unity → backend → modelo → backend → Unity. Se validó que el cliente podía lanzar peticiones HTTP, que el backend podía llamar al modelo y que la respuesta llegaba de vuelta a la interfaz. Sienta las bases de toda la comunicación posterior.

En la práctica, este hito fijó los contratos sobre los que se apoya todo lo demás. El backend expone su API con FastAPI y arranca con un ciclo de vida que prepara la base de datos y precalienta el modelo de embeddings y ChromaDB (véase 4.7.5). El cliente concentra todo el tráfico HTTP en una única clase, BackendClient: un singleton que serializa con Newtonsoft, trata los errores de forma uniforme y ofrece un método tipado por endpoint, de modo que el resto de scripts de Unity no saben nada de HTTP: piden y reciben objetos.

El puente Unity → backend (Core/BackendClient.cs, extracto)

```
public IEnumerator Post<T>(string path, object body,
                           Action<T> onOk, Action<string> onErr = null)
{
    string json = JsonConvert.SerializeObject(body);
    using var req = new UnityWebRequest(baseUrl + path, "POST");
    req.uploadHandler   = new UploadHandlerRaw(Encoding.UTF8.GetBytes(json));
    req.downloadHandler = new DownloadHandlerBuffer();
    req.SetRequestHeader("Content-Type", "application/json");
    yield return req.SendWebRequest();
    Handle(req, onOk, onErr);
}

void Handle<T>(UnityWebRequest req, Action<T> onOk, Action<string> onErr)
{
    if (req.result != UnityWebRequest.Result.Success)
    { onErr?.Invoke($"{req.responseCode} {req.error}"); return; }
    onOk?.Invoke(JsonConvert.DeserializeObject<T>(req.downloadHandler.text));
}
```

Extracto de Código 19. Conexión Unity con el backend

La superficie completa de la API, tal y como quedó al final del desarrollo, se resume a continuación; la documentación interactiva (Swagger) que genera FastAPI en /docs permite además probar cada endpoint desde el navegador y fue la principal herramienta de pruebas del backend durante todos los hitos.

Superficie de la API REST (api/routes.py)

```
GET  /health                          estado del backend y modelos
POST /new-game                        inicia partida (roster propio opcional)
GET  /world/state                     tiempo, localizaciones, agentes, jugador
POST /sim/tick {steps}                avanza la simulación; eventos + elapsed ms
POST /agent/{id}/talk                 diálogo con memoria e hilo de la charla
GET  /agent/{id}/sheet                ficha (estado, inventario, exp, necesidades)
GET  /agent/{id}/memories             recuerdos, recientes o por consulta
POST /agent/{id}/conversation-quest   cierre de charla: misión y/o ir a
GET  /agent/{id}/shop                 tienda con precios por afinidad
POST /trade                           compraventa genérica de objetos
GET  /quests  ·  POST /quest/{id}/(accept|complete)   ciclo de misiones
GET  /armario  ·  POST /player/give   objetos del armario para el jugador
POST /player/grant                    fondo del pueblo (paquetes de monedas)
POST /player/move  ·  GET /player/sheet   zona e inventario del jugador
```

Extracto de Código 20. Métodos del API Rest

En la ilustración 18, se aprecia la visión de la API creada. Se puede ir desplegando cada parte y ejecutar cada instrucción por separado para ver su funcionamiento.

Ilustración 18. API del backend

4.2. H1 — Memoria episódica y diálogo

Se implementó el flujo de memoria y el diálogo con el jugador que recuerda interacciones anteriores. Cada conversación deja recuerdos en la base vectorial y, al responder, se recuperan los relevantes por relevancia, recencia e importancia. El efecto es que un NPC puede aludir a algo que el jugador le contó antes.

El circuito completo de un mensaje quedó fijado aquí y apenas ha cambiado desde entonces: la frase del jugador se convierte en embedding, se recuperan los recuerdos mejor puntuados, se compone el prompt (identidad, relación, datos reales, hilo de la charla y recuerdos) y, tras responder, la conversación vuelve a la memoria como dos recuerdos nuevos: lo que dijo el jugador —cuya importancia valora el modelo rápido— y lo que contestó el personaje.

El circuito de un mensaje (agents/converse.py, esqueleto)

```
async def talk to player(agent, player, message, now, history=""):
    memories = memory stream.retrieve(agent.id, message, now, k=5)
    prompt = (persona block(agent)          # identidad y objetivos
              + relacion con(player)        # etiqueta de afinidad
              + datos reales(agent)         # inventario, dinero, oficio...
              + hilo(history)               # lo ya dicho en ESTA charla
              + recuerdos(memories)
              + f'{player.name} te dice: "{message}"')
    reply = await llm.complete(prompt, model=MODEL_SMART)

    imp = await _rate_importance(f"{player.name} me dijo: {message}")
    memory stream.store(agent.id, "dialogue", f"...me dijo: {message}", now, imp)
    memory stream.store(agent.id, "dialogue", f"Le respondí: {reply}", now, imp-1)
    return reply
```

Extracto de Código 21. Recuperación de memoria para enviar al prompt en conversación

4.3. H2 — Mundo por localizaciones y vida social

El pueblo se dividió en localizaciones (plaza, panadería, taller, botica y mercado). Los agentes se mueven entre ellas de forma autónoma, conversan entre sí cuando coinciden, generan reflexiones a partir de lo vivido y propagan rumores. El jugador puede observar toda esta actividad en el registro de eventos. Visualmente, este hito da vida al mapa: los personajes dejan de estar quietos y recorren el pueblo según su rutina. En la ilustración 19 tenemos la secuencia de un tick de simulación y como se interrelacionan entre sí las distintas capas.

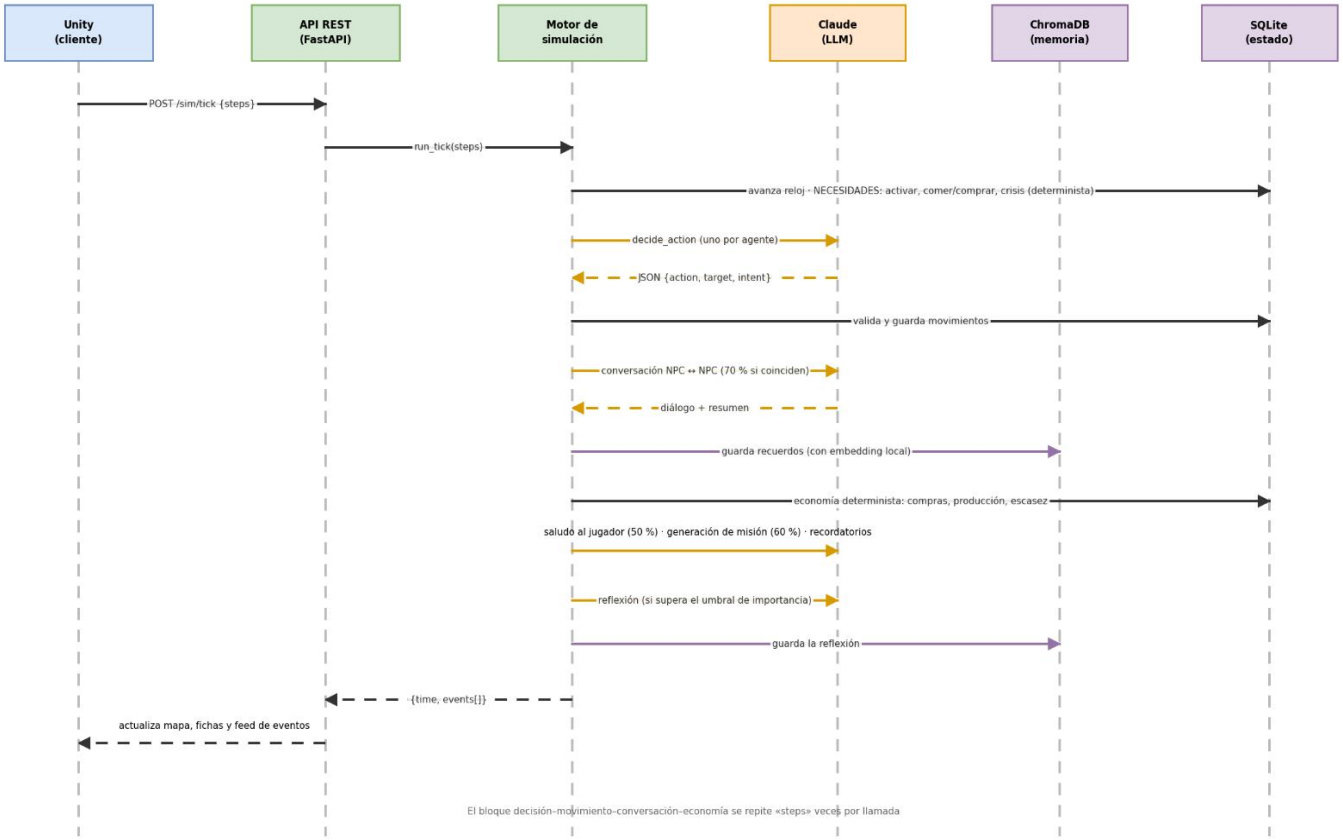

Ilustración 19. Secuencia de un «tick» de simulación

La vida social se apoya en un mapa de ocupación que se reconstruye en cada paso: quién está en cada localización. Cuando dos o más personajes coinciden hay un 70 % de probabilidad de que dos de ellos, elegidos al azar, conversen; el diálogo completo se genera en una única llamada y su resumen queda en la memoria de ambos, que es el vehículo de los rumores. En la versión final la charla incluye, además de los recuerdos recientes de cada uno, lo que cada uno sabe del jugador —obtenido con una recuperación dirigida, no confiando en que esté entre lo reciente—, de modo que la información sobre él también se propaga: si el jugador le contó al panadero que es capaz de conseguir de todo, el herrero puede enterarse por él y recomendarle acudirle.

Ocupación y conversaciones (agents/simulation.py, extracto)

```
occ = {loc: [] for loc in locations}
for a in agents.values():
    occ.setdefault(a.location, []).append(a.id)

for loc, ids in occ.items():
    present = [agents[i] for i in ids]
    if len(present) >= 2 and random.random() < 0.7:
        a, b = random.sample(present, 2)      # pareja al azar
        conv = await converse.agents converse(a, b, now)  # UNA llamada
        # el resumen queda en la memoria de AMBOS: así viajan los rumores
```

Extracto de Código 22. Posibilidad de entablar conversaciones en caso de NPCs cercanos

4.4. H3 — Objetos, economía y mercado

Es el corazón del entorno de mercado. Se introdujeron objetos e inventarios reales: cada objeto es una fila con su propietario, de modo que el inventario de cualquier personaje —o del jugador— es el conjunto de objetos cuyo propietario es él. María tiene stock; los productores compran materiales y fabrican sus productos. Los precios dependen de la relación entre personajes y existe escasez con un enfriamiento de un día: si un material no está disponible, el agente no insiste hasta el día siguiente (estado «En espera de material»). Cada agente sigue una rutina: va al mercado solo si necesita algo y vuelve a su taller a producir.

Una compraventa es una transacción completa y atómica: se comprueba el stock y el saldo, el dinero cambia de manos, el objeto cambia de propietario, la operación queda registrada en la tabla de transacciones de SQLite y —desde la revisión final— deja recuerdo en los dos participantes.

Una compraventa (game/economy.py, extracto)

```
def buy material(buyer, merchant, type key, ts):
    item = find item(merchant.id, type key)
    if not item:
        return None                          # sin stock
    price = price of(item, merchant.relationships.get(buyer.id, 0.0))
    if buyer.money < price:
        return None                          # sin dinero
    buyer.money -= price
    merchant.money += price
    store.set item owner(item.id, buyer.id)  # el objeto cambia de dueño
    store.log trade(ts, buyer.id, merchant.id, item.id, price, "buy")
    return {"buyer": buyer.name, "seller": merchant.name,
            "item": item.display_name, "price": price}
```

Extracto de Código 23. Proceso de compraventa

En la ilustración 20, se ve como muestra el inventario que tiene en este caso María, el precio de los ítems y lo que hemos comprado. Se compra una unidad por cada click, y en caso de que no queden, desaparece.

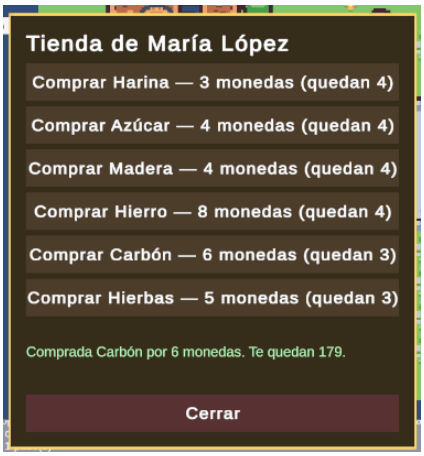

Ilustración 20. Ejemplo de tienda

4.5. H4 — Misiones emergentes

Cuando un NPC detecta que le falta un material, puede generar una misión para que el jugador se lo traiga. La generación es híbrida, tal y como se detalló en la sección 3.10: el modelo escribe el título y la descripción en la voz del personaje, pero el objeto pedido y la recompensa los fija el código a partir de sus carencias reales. Así la misión siempre encaja con la economía y siempre es completable; si no hay modelo disponible, un texto por reglas

actúa de red de seguridad. En el juego, el NPC con misión abierta muestra un icono de exclamación; al pulsarlo se abre el panel de misión donde el jugador la acepta, obtiene el objeto del armario, vuelve al NPC y lo entrega, recibiendo monedas y una mejora de la relación.

La vía autónoma primero selecciona candidatos —personajes a los que les falta un objeto del armario y no tienen ya una misión abierta— y lanza el dado una sola vez por paso, en lugar de inundar al jugador con misiones a cada tick. Las que quedan sin atender tienen recordatorio: al día siguiente el personaje acude a buscar al jugador.

Candidatos a misión (agents/simulation.py y game/quests.py, extracto)

```
def  quest candidates(agents):
    return [a for a in agents.values()
            if any(m in ARMARIO ITEMS and not find item(a.id, m)
                   for m in materials for(a))       # carencia real
            and not  open quests(a.id)]             # sin misión abierta

candidates =  quest candidates(agents)
if candidates and random.random() < 0.6:            # gotea, no inunda
    ag = random.choice(candidates)
    q = await quests.generate_quest(ag, clock.minute, clock.day)
```

Extracto de Código 24. Creación de misiones (estabilizadas en número)

Para evitar la monotonía, el selector excluye los objetos que el jugador ya entregó en los últimos dos días —si el material se consumió al producir, el personaje espera antes de repetir la petición— y, un treinta por ciento de las veces, pide un «capricho» del armario ajeno a sus recetas, lo que da variedad a los encargos.

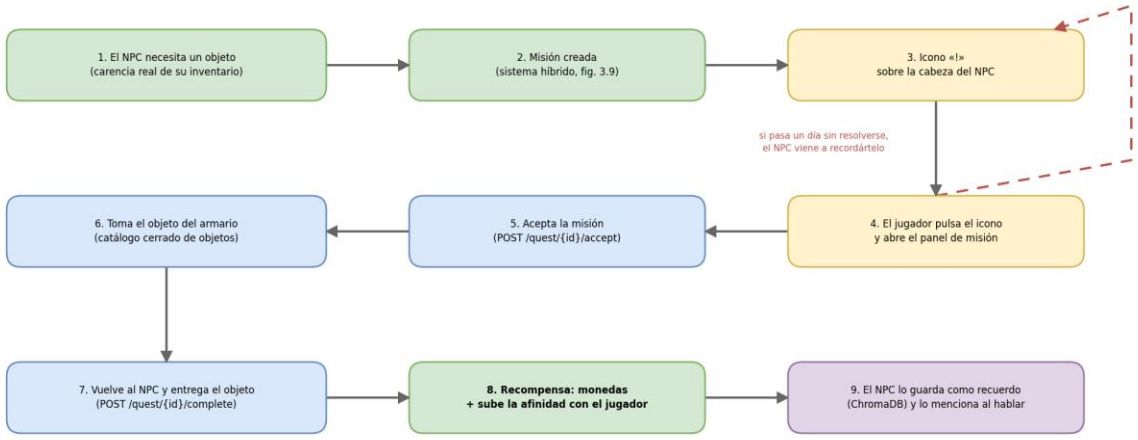

Ilustración 21. Flujo de una misión en el juego

En la ilustración 21, tenemos el diagrama de flujo de una misión en el juego: generación → icono «!» → aceptación → obtención en el armario → entrega y recompensa.

4.6. El jugador y la interfaz

En paralelo a los hitos se desarrolló la experiencia del jugador: movimiento con el teclado (WASD), detección de la zona por cercanía, diálogo con un NPC al acercarse (tecla T), ficha e inventario propios y una interfaz con botones para iniciar partida y avanzar el tiempo, el registro de eventos con desplazamiento y el mapa por tiles. Los NPC pueden además acercarse a la plaza e iniciar una conversación con el jugador.

La definición de personajes y zonas vive en un único archivo de configuración del backend (data/agents_seed.json visto en el punto 3.5.1). Los scripts de Unity ya no duplican esas

listas: al arrancar, el cliente obtiene los identificadores de NPC y las localizaciones de GET /world/state, y las listas del Inspector quedan solo como reserva por si el backend no responde. Añadir un quinto personaje se reduce así a editar el JSON (y su sprite en la escena).

4.7. Problemas técnicos encontrados y soluciones

Buena parte del trabajo de ingeniería consistió en depurar problemas de integración entre Unity y el backend, y de la propia interfaz. Se recogen los más relevantes por su interés técnico.

4.7.1. El arranque en frío del servidor

Al medir los tiempos de simulación se observó que la primera petición tras arrancar el servidor tardaba mucho más que las siguientes (≈19 s frente a ≈6 s). No es un defecto del modelo de lenguaje, sino consecuencia de la carga perezosa: la primera vez que se necesita la memoria episódica se importa y carga el modelo local de embeddings (sentence-transformers sobre PyTorch, ~90 MB), se abre ChromaDB con su índice vectorial y se crea el cliente HTTP de la API. Todo ello ocurre una única vez y queda residente en memoria. Se resolvió añadiendo un warm-up en el arranque (variable de entorno WARMUP, activada por defecto), que precarga embeddings y ChromaDB: el servidor tarda unos segundos más en estar listo, pero la primera acción del jugador ya no paga ese coste.

El warm-up (main.py, extracto)

```
if config.WARMUP:                          # al arrancar FastAPI
    embeddings.embed("warmup")             # carga MiniLM/PyTorch UNA vez
    memory_stream._get_collection()        # abre ChromaDB y su índice
```

Extracto de Código 25. Precarga de chromaDB y embeddings

4.7.2. Paralelización de las decisiones de los agentes

Las decisiones de los N agentes se pedían al modelo de forma secuencial (una espera tras otra). Se añadió un modo paralelo con asyncio.gather, activable por configuración (PARALLEL_DECISIONS), en el que el tick espera aproximadamente lo que dura UNA llamada en lugar de N. Es seguro porque la decisión solo lee el estado del mundo; los movimientos se aplican después, en orden. Para poder comparar ambos modos, POST /sim/tick devuelve ahora elapsed_ms y Unity lo muestra en el registro de eventos.

Nota sobre la variabilidad medida (≈1,2 s a ≈10 s por tick): el tiempo no depende solo de las decisiones. Según lo que ocurra en el paso, se encadenan llamadas ADICIONALES y secuenciales al modelo: una conversación NPC-NPC si dos coinciden (70 %), la frase de apertura del saludo al jugador (50 %), la generación de misión (60 % si hay candidatos), el recordatorio de misiones pendientes y hasta una reflexión por cada agente que conversó (40 %). Un tick «tranquilo» hace 1 tanda de llamadas (solo decisiones); uno «movido» puede hacer 4-5 encadenadas, de ahí los picos. Es un comportamiento esperable; si se quisiera acotar, estas llamadas también podrían agruparse con asyncio.gather.

Haciendo las observaciones en el depurador, se apreciaba cómo los tiempos pasan de 6-10 segundos, a ser de 2-4 segundos, haciendo movimientos y decisiones similares.

4.7.3. El mensaje «Charlaron un momento»

En sesiones largas, el registro mostraba con frecuencia conversaciones entre personajes reducidas a un simple «Charlaron un momento», sin diálogo alguno. No era un comportamiento del modelo, sino el texto de seguridad del código, que se producía sin razón

aparante. Finalmente, la llamada que genera la conversación disponía de un máximo de 300 tokens y, cuando el diálogo crecía, el JSON llegaba truncado, el análisis fallaba y se activaba el relleno — que además se guardaba como recuerdo, ensuciando la memoria de ambos personajes con frases vacías. La solución fue doble: ampliar el límite a 600 tokens y dejar de generar recuerdos cuando la conversación no tiene contenido real. Es un buen ejemplo de la clase de fallos propios de construir sobre un modelo de lenguaje: el error no estaba en lo que el modelo decía, sino en el espacio que se le daba para decirlo.

4.8. Ampliación final: economía de necesidades, tienda y ventanas

Tras cerrar los hitos previstos se añadió una última ampliación acotada que cierra el ciclo económico (sección 3.5.5): las necesidades diarias con sus crisis y misiones de auxilio. Esta mejora vino por la necesidad de hacer que el jugador se sintiese útil en el entorno. Ahora mismo el poblado está destinado a la hambruna ya que la comerciante puede llegar a quedarse sin materia prima que vender, y con ello el resto de NPCs no pueden fabricar sus productos. La reposición periódica (un «carro de suministros» matinal que ella pagaría de su propio bolsillo, devolviendo a la circulación el dinero que acumula) está diseñada y anotada como primera mejora.

Esto llevó también a la evolución de los oficios por práctica (sección 3.5.4) y la tienda con la que el jugador comercia con cada personaje y sostiene, en particular, al herrero, ya que se pretendía que no pidieran ingredientes que no tuvieran.

En paralelo se mejoró la usabilidad de la interfaz: todos los paneles son ahora ventanas móviles —se arrastran manteniendo pulsado sobre su marco, resaltado en ámbar— y la ventana de diálogo y la ficha de personaje pueden además redimensionarse desde su esquina inferior derecha.

Todo lo añadido en esta fase respeta el principio de diseño del trabajo: las decisiones «blandas» (qué decir, qué encargar, cuánto ofrecer) siguen siendo del modelo, y las reglas duras (precios, umbrales de experiencia, acotados y crisis) viven en el código.

5. Resultados y evaluación

5.1. Estado alcanzado

El resultado es un prototipo jugable con las siguientes capacidades demostrables: cuatro NPC con memoria y vida propia que se mueven, conversan y producen; una economía con objetos reales que cambian de dueño y precios sensibles a la relación; diálogo abierto con el jugador que recuerda lo hablado; un inventario de jugador con dinero y objetos; y un ciclo completo de misiones emergentes (generación por el NPC, icono de aviso, aceptación, obtención del objeto en el armario y entrega con recompensa). Visualmente, el pueblo se representa con un mapa por tiles y sprites, y la interfaz comunica de forma legible el estado de los personajes y del mercado.

A esa base se suman las ampliaciones finales: la evolución de oficios por práctica (niveles que desbloquean recetas), la economía de necesidades con sus crisis y misiones de auxilio, y las tiendas con las que el jugador comercia y sostiene a los personajes; todo ello observable en una interfaz de ventanas móviles y redimensionables.

5.2. Casos de uso

A continuación, se recogen los principales casos de uso del sistema en la ilustración 24. El actor principal es el Jugador; un actor secundario, la Simulación, agrupa el comportamiento autónomo que el sistema ejecuta en cada paso.

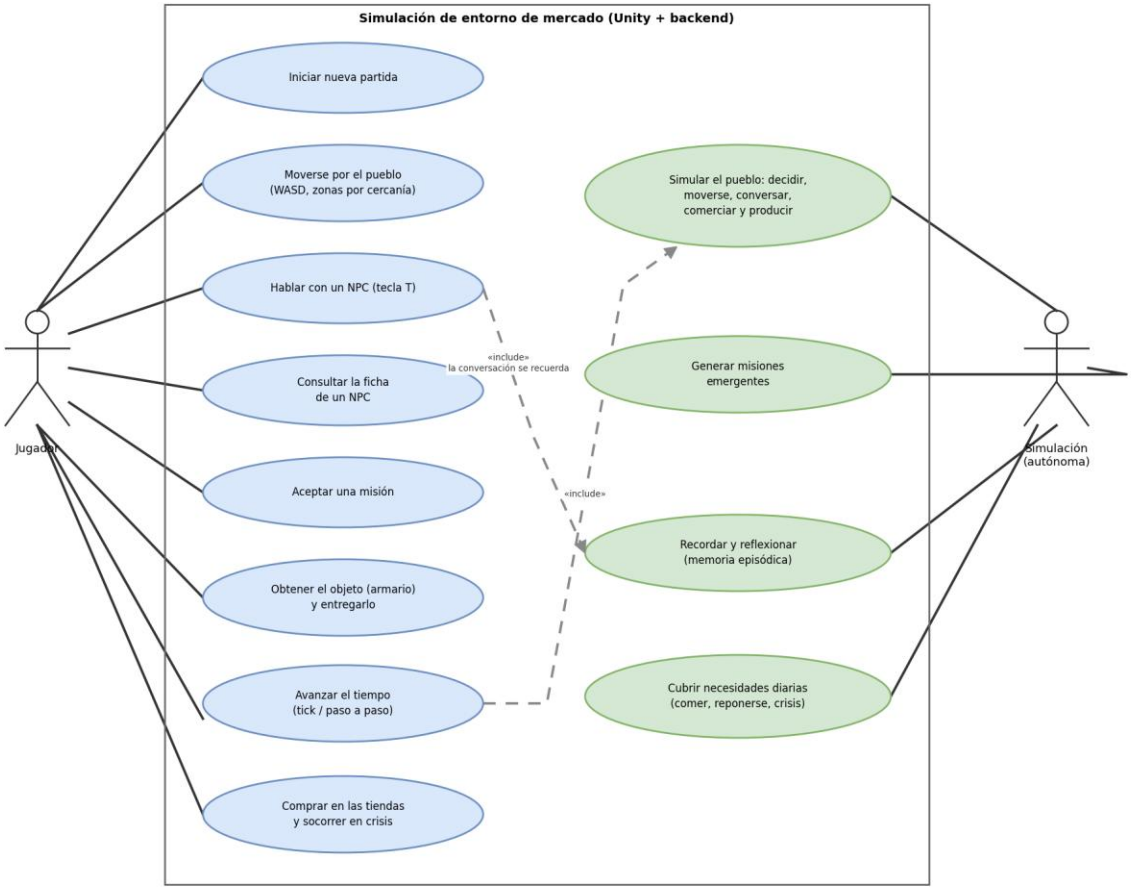

Ilustración 22. Diagrama de casos de uso del jugador

Caso de uso	Descripción
Iniciar nueva partida	El jugador arranca una partida; el sistema carga el roster de personajes y reinicia el mundo.
Moverse por el pueblo	El jugador se desplaza con el teclado; su zona se reporta por cercanía.
Hablar con un NPC	Al acercarse y pulsar T, se abre la ventana de diálogo; la conversación se recuerda.
Consultar la ficha de un NPC	Al hacer clic sobre un NPC, se muestra su estado, dinero, inventario, habilidades, objetivos y recuerdos.
Aceptar una misión	Sobre un NPC con icono «!», el jugador abre el panel de misión y la acepta.
Obtener y entregar el objeto	El jugador toma el objeto del armario (o lo compra en una tienda) y lo entrega al NPC, recibiendo recompensa y mejora de relación.
Comprar en la tienda de un NPC	Desde la ficha del personaje, el jugador abre su tienda y compra productos al precio que marca la afinidad del vendedor.
Socorrer a un NPC en crisis	Un personaje que no puede cubrir sus necesidades pide auxilio y queda inmóvil; el jugador le lleva pan, pócima o monedas y recibe su gratitud.
Avanzar el tiempo	El jugador ejecuta pasos de simulación (tick) y observa la vida del pueblo en el registro de eventos.
Simular el pueblo (Simulación)	En cada paso, los agentes cubren sus necesidades, deciden, se desplazan, conversan, comercian, producen y, cuando toca, reflexionan, generan misiones o entran en crisis.

Tabla 11. Casos de uso

Los cuatro casos de uso centrales del prototipo se especifican a continuación en formato de ficha —actor, precondiciones, flujo principal, flujos alternativos y postcondiciones—, cada uno acompañado de su diagrama de flujo. Las validaciones que aparecen en los flujos son las descritas en los capítulos 3 y 4: el modelo propone y el código valida y ejecuta.

5.2.1. CU-01 — Hablar con un NPC

Campo	Detalle
Actor principal	Jugador.
Actores secundarios	NPC (agente), backend (motor de agentes y memoria) y modelo de lenguaje.
Precondiciones	Partida iniciada; jugador y NPC en la misma zona y a distancia de contacto.
Flujo principal	1. El jugador se acerca al NPC y aparece el aviso «Pulsa T para hablar».

Campo	Detalle
	2. Al pulsar T se abre la ventana de diálogo y el juego se pausa. 3. El jugador escribe un mensaje y lo envía con Intro (POST /agent/{id}/talk). 4. El backend recupera los recuerdos pertinentes (relevancia + recencia + importancia) y compone el prompt: identidad, afinidad, datos reales del personaje e hilo de la charla. 5. El modelo de calidad responde en la voz del personaje; el mensaje y la respuesta se guardan como recuerdos, con importancia tasada por el modelo rápido. 6. Los pasos 3 a 5 se repiten mientras dure la conversación. 7. Al cerrar la ventana, el NPC relee la transcripción (conversation-quest).
Flujos alternativos	A1. Encargo emergente: si de la charla surge un encargo natural, el modelo propone objeto, cantidad y recompensa; el código los valida (catálogo del armario, 1–5 unidades, recompensa entre 0,5× y 4× el valor) antes de crear la misión. A2. Renegociación: el nuevo trato sustituye un encargo normal abierto; nunca una misión de auxilio. A3. Desplazamiento acordado: el campo ir_a envía al NPC hacia una localización validada. A4. Saludo iniciado por el NPC: el personaje camina hasta el jugador y la charla solo se abre al alcanzarlo (desiste a los 20 segundos).
Postcondiciones	La conversación queda en la memoria episódica del NPC; si procede, existe una misión ofrecida o el NPC parte hacia la localización acordada.

Tabla 12. Especificación del caso de uso CU-01 — Hablar con un NPC

A continuación, la ilustración 25 muestra el diagrama de flujo de este primer caso de uso, donde el jugador se acerca a hablar con un NPC y los pasos que sigue el código para poder ejecutarlo.

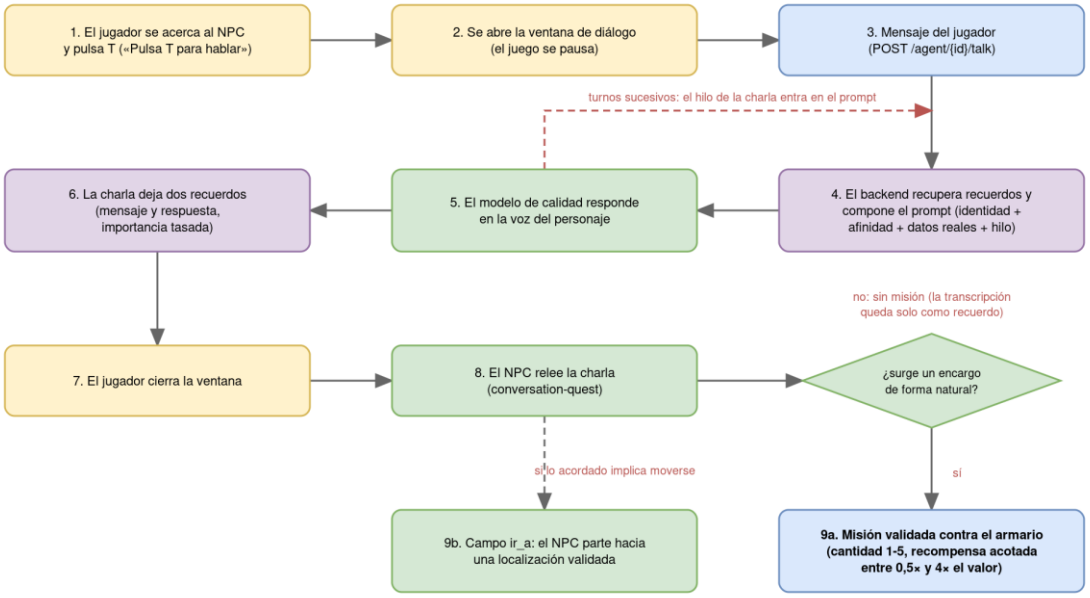

Ilustración 23. Flujo del caso de uso CU-01 — Hablar con un NPC

5.2.2. CU-02 — Aceptar y completar una misión

Campo	Detalle
Actor principal	Jugador.
Actores secundarios	NPC (dador de la misión) y backend.
Precondiciones	Existe una misión en estado «ofrecida» (icono «!» sobre el NPC), generada por la vía autónoma, la conversacional o una crisis.
Flujo principal	1. El jugador pulsa el icono y abre el panel de misión. 2. Acepta la misión (POST /quest/{id}/accept). 3. Obtiene el objeto pedido: lo toma del armario (catálogo cerrado) o lo compra en una tienda. 4. Vuelve al NPC y entrega (POST /quest/{id}/complete). 5. El cierre es determinista: el código comprueba el inventario real del jugador, transfiere los objetos al NPC, abona la recompensa, sube la afinidad (+0,1) y guarda en el NPC un recuerdo de alta importancia.
Flujos alternativos	A1. Inventario insuficiente: la entrega no procede y la misión sigue abierta. A2. Completar sin haber aceptado: la API lo rechaza de forma controlada. A3. Recordatorio: si pasa un día sin resolverse, el NPC acude a buscar al jugador en persona.

Campo	Detalle
	A4. Renegociación: un trato posterior cerrado en una charla sustituye el encargo, que se cancela dejando el recuerdo correspondiente.
Postcondiciones	Objetos transferidos al NPC, recompensa abonada al jugador, afinidad mejorada y recuerdo del cumplimiento en la memoria del personaje.

Tabla 13. Especificación del caso de uso CU-02 — Aceptar y completar una misión

La ilustración 26 muestra el diagrama de flujo la creación y completado de misiones, donde, además de generar recompensas en ítems, también mejora la afinidad entre personajes.

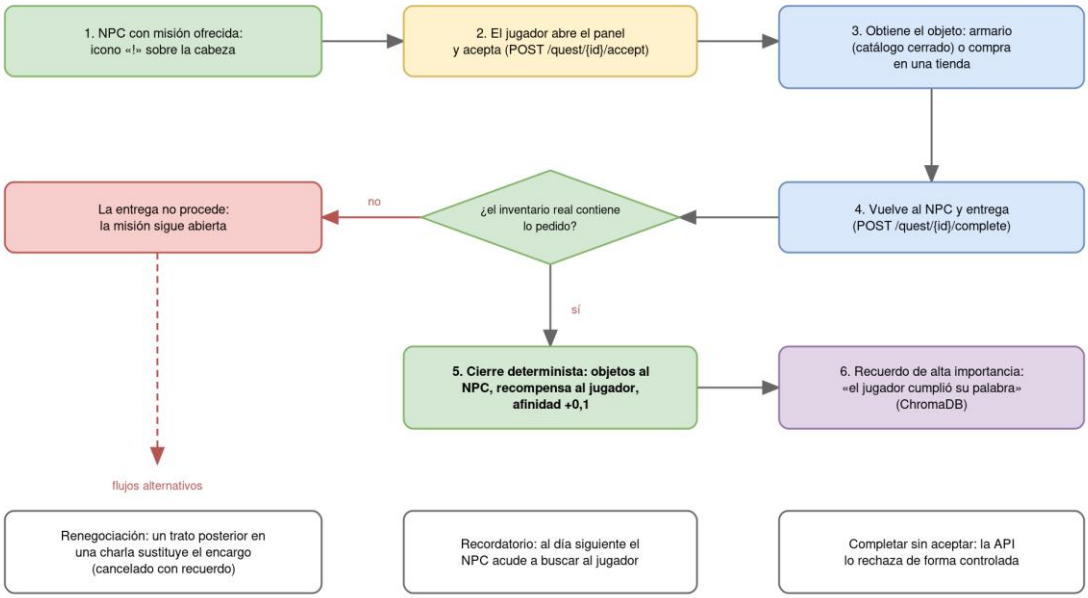

Ilustración 24. Flujo del caso de uso CU-02 — Aceptar y completar una misión

5.2.3. CU-03 — Comprar en la tienda de un NPC

Campo	Detalle
Actor principal	Jugador.
Actores secundarios	NPC (vendedor) y backend (economía determinista).
Precondiciones	El vendedor dispone de stock propio de su oficio: la comerciante vende materiales y cada productor, únicamente lo que fabrica.
Flujo principal	1. El jugador hace clic sobre el NPC y se abre su ficha. 2. Desde la ficha abre la tienda (GET /agent/{id}/shop); los precios llegan ajustados por afinidad: precio = valor × (1 − 0,2 · afinidad), con un mínimo de una moneda. 3. Elige el objeto y confirma la compra (POST /trade). 4. El código valida stock y saldo.

Campo	Detalle
	5. La transacción es atómica: el dinero cambia de manos, el objeto cambia de propietario, la operación se registra en SQLite y deja recuerdo en ambos participantes.
Flujos alternativos	A1. Sin stock o sin saldo suficiente: la operación se rechaza sin efectos sobre el mundo. A2. Sin liquidez: el fondo del pueblo (paquetes de 50 monedas del armario) permite desbloquear la compra.
Postcondiciones	El objeto pasa al inventario del jugador y la transacción queda registrada; ambos participantes recuerdan la compra.

Tabla 14. Especificación del caso de uso CU-03 — Comprar en la tienda de un NPC

En la ilustración 27 el diagrama de flujo indica el proceso de comprar en una tienda. Existe un servicio de seguridad en el que el jugador puede coger dinero del armario, y a parte, los jugadores pueden solicitar dinero en la misión de crisis.

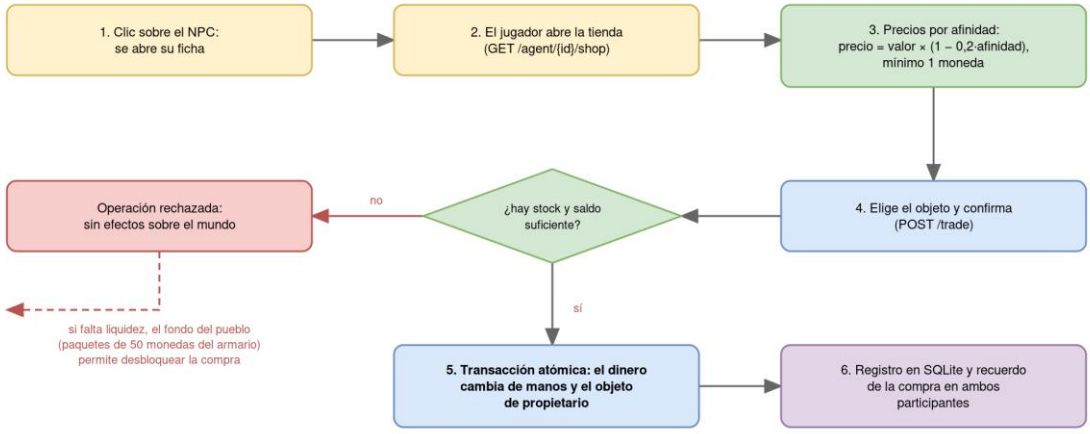

Ilustración 25. Flujo del caso de uso CU-03 — Comprar en la tienda de un NPC

5.2.4. CU-04 — Socorrer a un NPC en crisis

Campo	Detalle
Actor principal	Jugador.
Actores secundarios	NPC (en crisis) y backend (reglas, sin llamadas al modelo).
Precondiciones	Un NPC entra en crisis: arrastra una necesidad de un día para otro o, al caer la tarde, no puede pagarse el remedio. La crisis se construye íntegramente por reglas.
Flujo principal	1. El NPC cancela sus encargos abiertos, genera la misión de auxilio y queda inmóvil. 2. La misión pide exactamente lo que falta: pan, pócima y/o un fondo de 30 monedas. 3. El jugador consigue lo pedido (armario, panadería, botica o fondo del pueblo) y lo entrega.

Campo	Detalle
	4. Lo entregado se consume en el acto; no hay recompensa monetaria: la afinidad sube +0,15 y queda un recuerdo de gratitud. 5. El NPC sale de la crisis y retoma su rutina.
Flujos alternativos	A1. Nadie ayuda: el NPC permanece inmóvil y no trabaja ni comercia hasta que se resuelva la crisis. A2. La misión de auxilio nunca se sustituye por una renegociación: la supervivencia manda sobre cualquier negocio.
Postcondiciones	Necesidad cubierta, crisis cerrada y afinidad con el jugador mejorada.

Tabla 15. Especificación del caso de uso CU-04 — Socorrer a un NPC en crisis

En este último caso de uso, la ilustración 28 muestra el diagrama de flujo de como socorrer a un NPC que se ha quedado con hambre, sin energía, y a lo mejor sin capacidad económica y como se genera y completa esta misión.

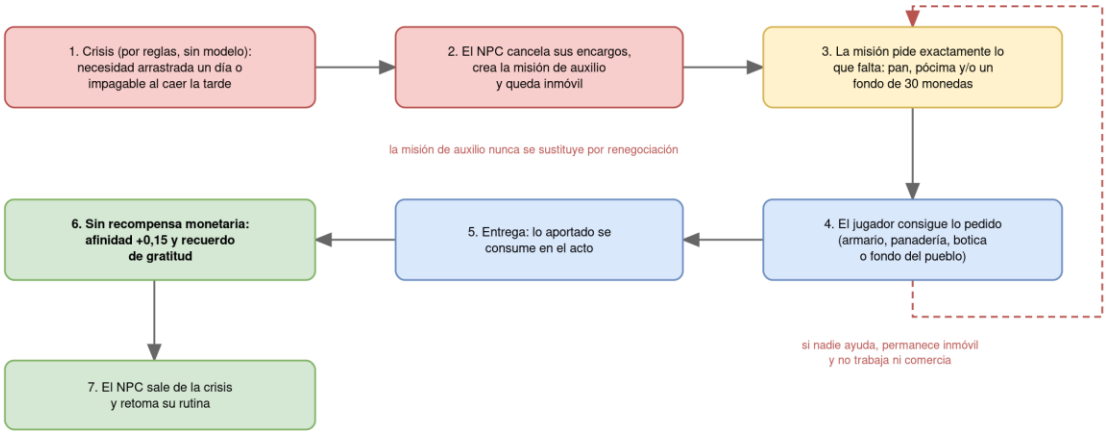

Ilustración 26. Flujo del caso de uso CU-04 — Socorrer a un NPC en crisis

5.3. Diagramas de uso y flujos

Para ilustrar la dinámica del sistema se incluyen varios diagramas complementarios al de casos de uso: la toma de decisiones del agente (ilustración 14), la máquina de estados (ilustración 15), el ciclo de memoria (ilustración 16), la generación de misiones (ilustración 17), la secuencia de un paso de simulación (ilustración 19) y el flujo de una misión de principio a fin (ilustración 21). Juntos muestran cómo el comportamiento autónomo de los agentes y la interacción del jugador convergen en el bucle de juego.

Para complementar los diagramas, se instrumentó una partida real de cuatro días de juego —con conversaciones, misiones, crisis y comercio— y se extrajeron métricas de las dos bases de datos del sistema: el estado del mundo (SQLite) y la memoria episódica (ChromaDB). Las tres gráficas siguientes se generaron a partir de esos datos.

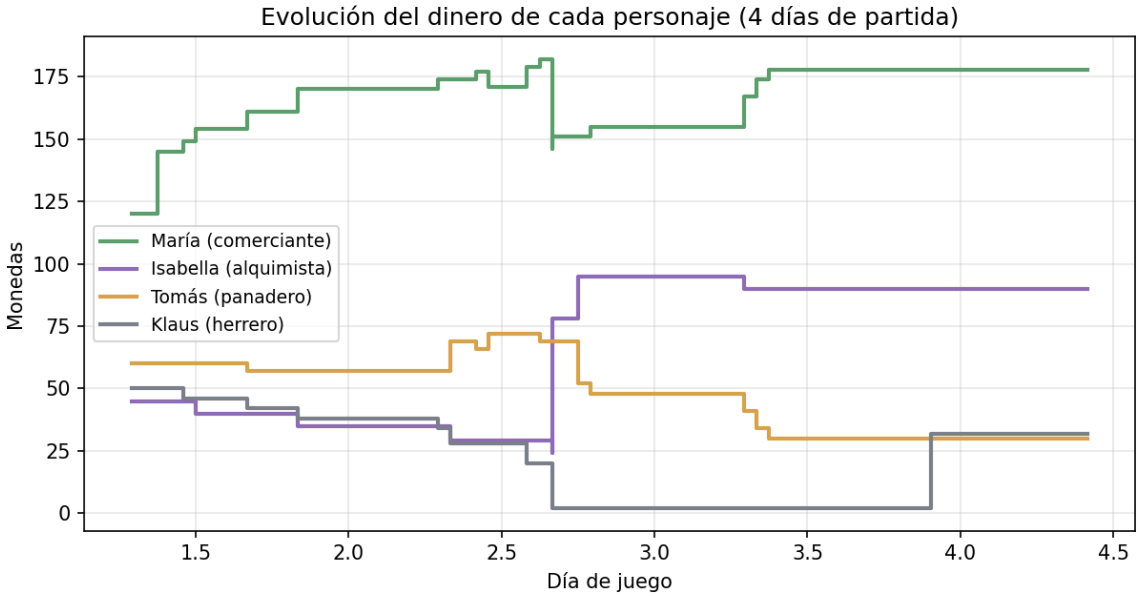

Ilustración 27. Evolución del dinero de los personajes

Evolución del dinero de cada personaje (ilustración 29), reconstruida desde el registro de transacciones y las misiones de auxilio (el saldo final reconstruido coincide exactamente con el real de la base de datos).
La primera gráfica confirma las dinámicas de diseño: María acumula capital de forma sostenida (de 120 a 178 monedas) como nodo central del mercado; Isabella prospera (de 45 a 90) porque todo el pueblo le compra pócimas para el cansancio; y Tomás y Klaus operan con márgenes estrechos —Klaus llegó a arruinarse el segundo día y a entrar en crisis, de la que salió gracias al auxilio del jugador y a la venta de sus piezas—. Las tres misiones de auxilio de la partida se aprecian como escalones de +30 monedas. Es exactamente el comportamiento que motivó las decisiones de las secciones 3.5.5 y 4.8: sin la demanda de comida y pócimas, y sin el jugador sosteniendo al herrero, la economía colapsaría hacia la comerciante.

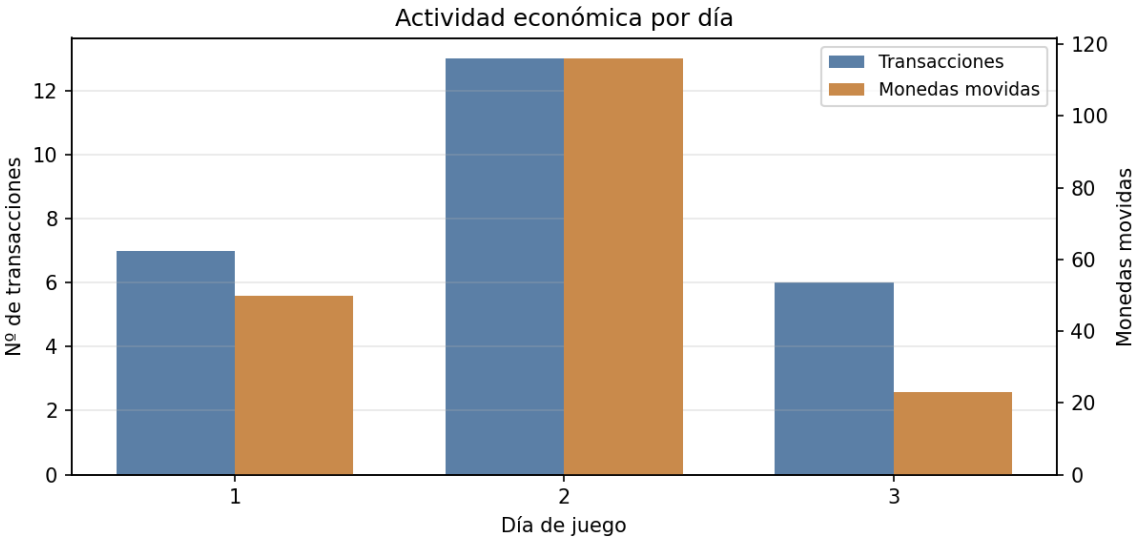

Ilustración 28. Actividad económica por día

Actividad económica por día (ilustración 30): número de transacciones y monedas movidas. La actividad económica es continua: 26 transacciones en los tres primeros días completos, con el pico del segundo día (13 operaciones), cuando las necesidades diarias entraron en vigor y los personajes comenzaron a comprar comida y remedios además de materiales.

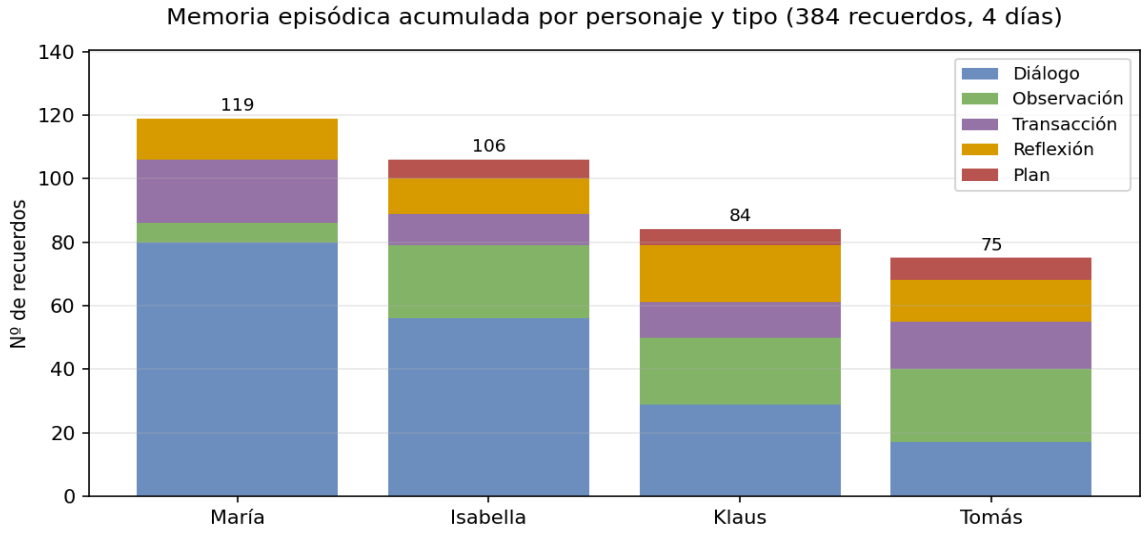

Ilustración 29. Memoria episódica acumulada por personaje y tipo de recuerdo

Memoria episódica acumulada por personaje y tipo de recuerdo (384 recuerdos en 4 días) (ilustración 31).
La tercera gráfica muestra la huella cognitiva de la partida: 384 recuerdos en cuatro días, sin ninguna llamada adicional al modelo gracias a los embeddings locales. La distribución delata la vida de cada personaje: María es la más «social» (80 diálogos: su mostrador la convierte en el centro de conversación del pueblo) y la que más transacciones recuerda; Isabella combina diálogo y observación; y los cuatro reflexionan con regularidad (55 reflexiones en total), lo que confirma que el disparo por umbral de importancia produce reflexiones proporcionales a lo vivido, tal y como pretende el criterio del artículo original.

5.4. Valoración del desarrollador

La arquitectura cliente–servidor ha demostrado ser cómoda para iterar: los hitos de lógica (memoria, economía, misiones) se construyeron y probaron en el backend de forma independiente del cliente y después se conectaron a la interfaz. El enfoque híbrido —modelo para lo creativo, código para lo económico— ha sido clave para que el mercado sea coherente y las misiones siempre completables. En el apartado visual, el mayor esfuerzo se concentró en la interfaz: convertir el rico estado interno de los agentes en algo que el jugador pueda leer de un vistazo. El prototipo se encuentra en una fase temprana pero funcional, y sirve como base sólida sobre la que crecer. A continuación, se muestra cómo una conversación acaba generando una misión acorde a lo pactado en dicho texto en la ilustración. 32

Ilustración 30. Creación de misión por conversación

6. Conclusiones y líneas futuras

El proyecto confirma que es viable construir un entorno de mercado con NPC creíbles y útiles para la jugabilidad combinando un modelo de lenguaje con una memoria persistente y una economía tangible. La clave para que el resultado sea jugable, y no solo vistoso, ha sido el enfoque híbrido: dejar que el modelo aporte la parte creativa (diálogo, motivación, texto de las misiones) mientras el código garantiza la coherencia y la completabilidad (qué objetos existen, qué recompensa se da, que una misión siempre se pueda terminar).

Partiendo del trabajo de Park et al. (2023), se ha comprobado que trasladar aquella arquitectura cognitiva a un motor de videojuego (Unity), con un modelo distinto (Claude) y una memoria sobre base de datos vectorial, es no solo viable sino ventajoso para un caso jugable: la memoria estructurada y la recuperación por relevancia reducen las incoherencias del personaje. También se ha comprobado el peso del trabajo de integración y del apartado visual: gran parte del esfuerzo no está en «pedirle cosas al modelo», sino en la representación gráfica del mundo, en una interfaz que haga legible lo que ocurre y en depurar los problemas propios del entorno (física, interfaz, sincronización de estado).

6.1. Líneas futuras

El proyecto está pensado para continuar por cuatro hitos, en este orden de interés:

1. **H5 — Evolución (línea prioritaria iniciada).** La base ya está implementada: las habilidades suben de nivel con la práctica, con umbrales incrementales fijados en código, y el nivel 2 desbloquea las recetas avanzadas de cada oficio. Queda su segunda mitad: que las nuevas capacidades abran cadenas de misiones y objetos nuevos, y que la experiencia provenga también de encargos y descubrimientos, no solo de la producción. Esta dirección entronca con el aprendizaje continuo en tiempo de despliegue que formaliza AgentOdyssey (Zhang et al.): agentes que, mientras juegan, adquieren conocimiento y habilidades nuevas apoyándose en su memoria episódica; la progresión de oficios de este pueblo sería una versión jugable y medible de esa idea. Finalmente se incluiría la reposición de la comerciante principal, metiendo nuevo género y gastando el dinero acumulado en los nuevos productos.
2. **H6 — Rutinas y drama social.** Un plan diario por agente (planificación) y relaciones que evolucionan con lo que ocurre: que un NPC se enfade o te coja cariño según tus actos. Aporta mucha vida al pueblo, aunque es un objetivo más difuso y difícil de acotar.
3. **H7 — Cierre de día y narrativa.** Un resumen del día generado con un modelo de mayor calidad (Claude Opus), guardado de partida y una build de demostración lista para entregar y defender.
4. **H8 — Motor de IA local.** Sustituir las llamadas a la API por un modelo de lenguaje ejecutado en la propia máquina (por ejemplo, Llama o Qwen sobre Ollama o llama.cpp), de modo que el juego funcione sin conexión a internet y con coste cero por paso. La arquitectura ya lo facilita: toda la comunicación con el modelo pasa por un único módulo (`llm.py`) y los embeddings de la memoria ya se calculan en local, así que el cambio se reduce a sustituir el cliente de ese módulo. El reto no es la integración sino la calidad: los modelos abiertos de tamaño doméstico (7–8 mil millones de parámetros) son menos fiables generando el JSON estructurado que exige el sistema y piden una tarjeta gráfica dedicada. Por eso se plantea como opción configurable —motor local

para jugar sin red y sin coste, API para la máxima calidad— y no como sustitución completa.

En paralelo quedan tareas de pulido ya identificadas: animaciones de los personajes, una cámara que siga al jugador, colisiones con edificios y NPC, la bajada justificada del estado de ánimo (hoy solo sube) y refinamiento general de la interfaz.

Más allá de esos hitos, el desarrollo deja identificadas otras ampliaciones de interés: a) Planificación diaria jerárquica completa al estilo de Park et al. (plan del día → franjas → reacción a imprevistos), que daría rutinas más creíbles que la regla actual. b) Evaluación con usuarios: pequeño cuestionario de credibilidad de los NPC (el artículo original valida con evaluación humana). c) Escalado: medir coste y latencia al pasar de 4 a 8-10 agentes (caché de prompts, agrupar decisiones en una sola llamada). d) Economía más profunda: recetas de varios pasos y precios por oferta/demanda reales. e) Guardado/carga de partida.

7. Conclusions and future work

This project confirms that it is feasible to build a market environment with believable NPCs that are useful for gameplay, by combining a large language model with persistent memory and a tangible economy. The key to a result that is playable —and not merely eye-catching— has been the hybrid approach: letting the model provide the creative part (dialogue, motivation, quest text) while the code guarantees consistency and completability (which items exist, what reward is granted, and that a quest can always be finished).

Building on the work of Park et al. (2023), it has been shown that porting that cognitive architecture to a game engine (Unity), with a different model (Claude) and a memory backed by a vector database, is not only feasible but advantageous for a playable case: structured memory and relevance-based retrieval reduce the character's inconsistencies. The weight of integration work and of the visual layer has also become clear: much of the effort lies not in «asking the model for things», but in the graphical representation of the world, in a user interface that makes what happens readable, and in debugging the issues inherent to the environment (physics, UI, state synchronisation).

7.1. Future work

The project is designed to continue along four milestones, in this order of interest:

1. **H5 — Evolution (priority line, already started).** The foundation is in place: skills level up with practice, with incremental thresholds fixed in code, and level 2 unlocks each trade's advanced recipes. Its second half remains: new capabilities opening up quest chains and new items, and experience coming also from quests and discoveries, not only from production. This direction connects with the test-time continual learning formalised by AgentOdyssey (Zhang et al.): agents that, while playing, acquire new knowledge and skills supported by their episodic memory; this village's trade progression would be a playable, measurable version of that idea. Finally, the main merchant's restock would be included, bringing in new stock and spending the accumulated money on the new products.
2. **H6 — Routines and social drama.** A daily plan per agent (planning) and relationships that evolve with what happens: an NPC growing fond of or angry with the player depending on their actions. It adds life to the town, although it is a more diffuse and harder-to-scope goal.
3. **H7 — End-of-day and narrative.** A summary of the day generated with a higher-quality model (Claude Opus), save games and a demo build ready to submit and defend.
4. **H8 — Local AI engine.** Replacing the API calls with a language model running on the player's own machine (e.g., Llama or Qwen via Ollama or llama.cpp), so the game runs offline at zero cost per step. The architecture already makes this easy: all model communication goes through a single module (`llm.py`) and memory embeddings are already computed locally, so the change comes down to swapping that module's client. The challenge is not integration but quality: consumer-sized open models (7–8 billion parameters) are less reliable at producing the structured JSON the system requires and need a dedicated GPU. It is therefore proposed as a configurable option — local engine for offline, cost-free play; API for maximum quality — rather than a full replacement.

In parallel, some polishing tasks are already identified: character animations, a camera that follows the player, collisions with buildings and NPCs, justified mood drops (mood currently only rises) and general UI refinement.

Beyond those milestones, the development leaves other extensions of interest identified: a) Full hierarchical daily planning in the style of Park et al. (day plan → time blocks → reaction to the unexpected), which would yield more believable routines than the current rule. b) User evaluation: a short NPC-believability questionnaire (the original paper validates its results with human evaluation). c) Scaling: measuring cost and latency when moving from 4 to 8–10 agents (prompt caching, grouping decisions into a single call). d) A deeper economy: multi-step recipes and real supply-and-demand pricing. e) Saving and loading games.

8. Presupuesto

Se estima el coste del proyecto separando los costes de desarrollo (horas de trabajo) de los costes de infraestructura (equipo, licencias y servicios).

8.1. Costes de desarrollo

Las horas se estiman por fases del trabajo. La tarifa aplicada corresponde a la de un desarrollador junior.

Fase	Horas	€/hora	Subtotal
Análisis, diseño y aprendizaje de tecnologías	60	30	1800 €
Desarrollo del backend (agentes, memoria, economía, misiones)	100	30	3000 €
Desarrollo en Unity (mapa, sprites, cámara, jugador)	70	30	2100 €
Interfaz de usuario y apartado visual	70	30	2100 €
Integración y depuración	50	30	1500 €
Documentación (memoria y diagramas)	40	30	1200 €
TOTAL desarrollo	390	—	11700 €

Tabla 16. Coste del desarrollo

8.2. Costes de infraestructura

Concepto	Coste	Observaciones
Equipo de desarrollo (PC)	200 €	Amortización estimada del ordenador durante el proyecto.
Licencia de Unity	0 €	Unity Personal (uso gratuito bajo su umbral de ingresos).
Recursos artísticos (Kenney)	0 €	Conjuntos Tiny Town / Tiny Dungeon de licencia libre.
Software (FastAPI, ChromaDB, etc.)	0 €	Herramientas de código abierto.
Uso de la API del modelo (Claude)	10 €	Estimación del consumo durante el desarrollo y las pruebas;
TOTAL infraestructura	210 €	

Tabla 17. Coste de la infraestructura

8.3. Coste total estimado

Sumando desarrollo e infraestructura, el coste total estimado del proyecto asciende a **11910€**.

9. Referencias

1. Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P. y Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. UIST 2023. arXiv:2304.03442.
2. Anthropic. Documentación de Claude y de la API. https://docs.claude.com
3. Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. EMNLP 2019 (modelo all-MiniLM-L6-v2).
4. Chroma. ChromaDB: base de datos de embeddings de código abierto. https://www.trychroma.com
5. Unity Technologies. Documentación de Unity (Tilemap, Sprite Renderer, UI). https://docs.unity3d.com
6. Ramírez, S. FastAPI. https://fastapi.tiangolo.com
7. Kenney. Asset packs Tiny Town y Tiny Dungeon. https://kenney.nl
8. Zhong, W., Guo, L., Gao, Q., Ye, H. y Wang, Y. (2024). MemoryBank: Enhancing Large Language Models with Long-Term Memory. AAAI 2024.
9. Chhikara, P., Khant, D., Aryan, S., Singh, T. y Yadav, D. (2025). Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv:2504.19413.
10. Du, P. (2026). Memory for Autonomous LLM Agents: Mechanisms, Evaluation, and Emerging Frontiers. arXiv:2603.07670.
11. Song, L. (2025). LLM-Driven NPCs: Cross-Platform Dialogue System for Games and Social Platforms. arXiv:2504.13928.
12. Buongiorno, S., Klinkert, J., Zhaung, Z., Chawla, T. y Clark, C. (2024). PANGeA: Procedural Artificial Narrative using Generative AI for Turn-Based, Role-Playing Video Games. arXiv.
13. Zhang, Z., Wen, Z., Zhang, A., Wang, A., Xie, J., Khashabi, D. y Shu, T. AgentOdyssey: Open-Ended Long-Horizon Text Game Generation for Test-Time Continual Learning Agents. arXiv preprint.