PROYECTO DE PORTAFOLIO

RJ45

Agente de IA Autónomo para Streaming

De bot de comandos a entidad sintética con autonomía real — un proyecto de investigación y desarrollo personal que documenta el proceso de construir un agente de IA complejo desde cero.

25+	v4.5	15+	2
módulos independientes	versión actual	sesiones de desarrollo	plataformas (Discord + YouTube)

Stack tecnológico

Node.js	**Ollama (local)**	**Flask / Python**	**Discord.js**	**YouTube API v3**	**Express + WebSocket**
LLaMA 3.2 Vision	**Qwen 2.5**	**Gemma 4**	**Qwen3-TTS**	**Puppeteer**	**REST APIs**

1. Qué es RJ45

RJ45 nació como un bot de Discord y terminó siendo algo bastante más complejo: un agente autónomo con personalidad definida, memoria persistente, criterio propio para tomar decisiones y presencia simultánea en Discord y YouTube. Es el asistente virtual de Cris, conocida en redes como lamonachina, streamer de contenido gaming y entretenimiento.

La diferencia entre un bot y un agente autónomo es importante. Un bot responde a comandos. RJ45 decide cuándo intervenir, qué publicar, cómo responder según su estado emocional del día, y se adapta al contexto del stream en tiempo real. Nadie le dice 'habla ahora' — ella evalúa la situación y actúa.

Por qué se llama RJ45

RJ45 es el nombre del conector de red más común — el que une equipos en una red física. El nombre es intencional: el proyecto nació con la idea de construir algo que conectara a la comunidad, igual que ese cable lo hace con los dispositivos. El nombre técnico con personalidad propia resume bien lo que es el proyecto.

Qué puede hacer

🎙️ Durante el stream

Lee el chat de YouTube en vivo, responde preguntas, genera audio con su voz sintetizada por OBS, y analiza visualmente lo que está pasando en pantalla para hacer comentarios contextuales.

📡 Entre streams

Curadora contenido de forma autónoma: busca tendencias en gaming y tecnología, evalúa si son relevantes para la comunidad, y las publica en Discord en el momento adecuado sin intervención humana.

🧠 Siempre activa

Monitorea la actividad del servidor, adopta mensajes sin respuesta, gestiona moderación automática, mantiene memoria de interacciones y ajusta su personalidad según el contexto.

2. Origen del proyecto

Este proyecto no empezó con un plan detallado. Empezó con curiosidad: ¿hasta dónde se puede llevar un bot de Discord si en vez de programar solo respuestas fijas, se trabaja con un modelo de lenguaje local y se le da contexto real?

La respuesta, después de más de quince sesiones de desarrollo y cuatro versiones mayores, es que se puede llegar bastante lejos. RJ45 pasó de responder comandos simples a tener un pipeline completo de curaduría, un sistema emocional que afecta sus respuestas, visión artificial para analizar lo que ocurre en pantalla, y voz sintetizada en tiempo real durante los directos.

El propósito real detrás del proyecto

RJ45 fue siempre un proyecto de estudio. La pregunta no era 'cómo hago un bot útil', sino 'cómo se construye y mantiene un sistema de IA complejo en el mundo real'. Eso implica tomar decisiones de arquitectura con recursos limitados, resolver problemas que no están en ningún tutorial, y aprender a trabajar con modelos de lenguaje como colaboradores de desarrollo, no solo como herramientas de texto.

3. Evolución del sistema

Cada versión major de RJ45 representa un cambio de paradigma, no solo una acumulación de features. El salto más importante no fue técnico — fue conceptual: pasar de pensar en comandos a pensar en autonomía.

v1.0	**Bot de comandos** Respuestas fijas a comandos específicos. Lógica condicional simple. Sin IA, sin memoria, sin personalidad.
v2.0	**Integración con modelo de lenguaje** Primer contacto con Ollama local. RJ empieza a generar respuestas dinámicas. Se define la personalidad base en JSON.
v3.0	**Arquitectura modular + curaduría autónoma** El sistema se divide en módulos especializados. Aparece el pipeline de curaduría: scraping → evaluación → redacción → publicación. RJ empieza a actuar sin que nadie se lo pida.
v4.0	**Agente autónomo completo** Integración con YouTube Live. Sistema emocional funcional. Visión artificial con análisis de pantalla en tiempo real. TTS con voz sintetizada. 25+ módulos coordinados.
v4.5 actual	**Resiliencia y estabilidad en producción** Post-mortem del primer directo real con StreamVision. Gestión de VRAM para coexistencia de modelos. Protección contra fallos en disco, quota y timing. El sistema sobrevive condiciones reales.

4. Arquitectura del sistema

RJ45 corre completamente en local — sin servicios cloud para el procesamiento de IA. Todos los modelos de lenguaje se ejecutan en la máquina del proyecto usando Ollama. Esto fue una decisión deliberada de arquitectura: más control, sin costos por token, y la posibilidad de usar modelos especializados para tareas distintas.

Componente	Tecnología	Rol en el sistema
Orquestador principal	Node.js + Discord.js	Conecta todos los módulos y gestiona eventos en tiempo real
Modelos de lenguaje	Ollama local — LLaMA, Qwen, Gemma4	Chat, análisis pesado, visión, voz — cada tarea tiene su modelo

Síntesis de voz (TTS)	Flask + Qwen3-TTS (Python)	Genera la voz de RJ en tiempo real durante el stream, reproducida por OBS
Panel de control	Express + WebSocket + SPA modular	Interfaz web para monitorear el sistema, iniciar directos y ver historial
Visión artificial	Puppeteer + Gemma4 (fine-tuned)	Captura frames del stream y genera análisis contextual de lo que está ocurriendo

Módulos principales

El sistema está dividido en 25+ módulos independientes agrupados por responsabilidad. La separación fue una decisión deliberada: que un bug en el módulo de memes no afecte al scheduler de posts, o que actualizar la curaduría no rompa el sistema de moderación.

Módulo	Función	Qué resuelve
Curador autónomo	**Investigación**	Busca tendencias en internet y evalúa si son relevantes para la comunidad
DecisionAgent	**Criterio**	Decide qué postear, cuándo y con qué formato — el 'juicio' de RJ
SchedulerPosts	**Timing**	Controla la frecuencia y el flujo único de publicación para evitar duplicados
StreamVision	**Visión**	Analiza frames del directo con Gemma4 para comentar lo que ve en pantalla
StreamVoz	**TTS**	Gestiona la voz sintetizada durante el stream, coordina con Flask y OBS
YouTubeHandler	**Live**	Lee el chat de YouTube en tiempo real y coordina respuestas
MoltbookAgent	**Memoria**	Agente de memoria a largo plazo — recuerda contexto de semanas anteriores
CommunityManager	**Comunidad**	Gestiona el Top 10 de temas relevantes para la comunidad
DirectoGuard	**Protección**	Pausa módulos secundarios durante el stream para no sobrecargar los recursos
Sistema emocional	**Personalidad**	Estado emocional del día (hype, chill, sarcástica) que afecta el tono de todas las respuestas

5. Decisiones de diseño que importan

Las decisiones más interesantes del proyecto no son las que están en el código — son las que determinaron qué arquitectura tiene sentido cuando los recursos son limitados y el sistema tiene que funcionar en producción real.

IA local en lugar de APIs cloud

Todos los modelos de lenguaje corren en la máquina del proyecto con Ollama. El tradeoff es claro: más configuración inicial, necesidad de gestionar VRAM manualmente, pero cero costo por token y control total sobre qué modelo usa cada tarea. Para un proyecto de investigación personal, eso tiene más valor que la conveniencia de una API externa.

Un modelo diferente para cada tarea

RJ45 usa cuatro modelos distintos según lo que necesite: uno ligero para voz (LLaMA 3.2 3B), uno para chat general (LLaMA 3.2 Vision 11B), uno pesado para análisis complejos (Qwen 2.5 14B) y uno fine-tuned para visión del stream (Gemma4). Esta separación surgió de un problema real: correr el modelo equivocado en el momento equivocado hacía colapsar la VRAM.

Un solo camino para postear

Después de identificar que nueve rutas distintas podían triggear una publicación en Discord — y que algunas se pisaban entre sí generando duplicados — se tomó la decisión de unificar todo por un único flujo: SchedulerPosts → DecisionAgent → ejecutarPost(). Cualquier comando o trigger externo tiene que pasar por ahí. Es una regla de arquitectura, no solo de código.

Gestión de VRAM por ciclo

El problema más complejo que resolvió el proyecto fue hacer coexistir Gemma4 (7.2GB) y Flask TTS (3-4GB) en una GPU de 8GB. La solución: StreamVision libera Flask de VRAM antes de cada análisis visual y lo restaura al terminar. Esto requirió construir endpoints específicos en el servidor Python y una lógica de coordinación entre módulos que no tenían comunicación directa antes.

---

6. Problemas reales resueltos en producción

La diferencia entre un proyecto de demo y un proyecto que funciona en producción son los problemas que nadie documenta en los tutoriales. Estos son algunos de los más representativos del proceso de desarrollo de RJ45.

El primer directo con StreamVision colapsó a los 8 ciclos

Gemma4 y Flask TTS compitieron por los 8GB de VRAM de la RTX 2060 Super. El resultado: MemoryError, Gemma devolviendo undefined en 1.8 segundos, loop infinito de errores. Además, StreamVoz se apagó a los 30 minutos porque el timer de silencio evaluó isConnected=false antes de que YouTube conectara. Tres bugs independientes que se encadenaron en el mismo stream.

Solución: gestión de VRAM por ciclo + arranque coordinado con await

Se construyeron tres endpoints nuevos en Flask para liberar y restaurar el modelo TTS alrededor de cada ciclo de visión. El timing de arranque se reescribió con async/await para garantizar que el timer de silencio solo se evalúa cuando YouTube ya confirmó conexión.

Posts duplicados en Discord por rutas de publicación que competían

Se mapearon 9 caminos distintos por los que RJ podía publicar contenido. Dos de ellos se alinearon temporalmente y pasaron el mismo check en la misma ventana. El mismo tema apareció dos veces con textos diferentes.

Solución: unificación en un solo camino + guards de concurrencia

Se eliminó el setInterval de inactividad legacy y se estableció como regla de arquitectura que todo posteo proactivo pasa por un único flujo. Se añadieron guards de concurrencia con try/finally para que nunca dos verificaciones corran en paralelo.

Node.js muriendo silenciosamente por disco lleno

Un writeFileSync sin try/catch dentro de un setInterval lanzaba ENOSPC cuando el disco se llenaba. El proceso entero de Node.js moría sin notificar a nadie.

Solución: try/catch en persistencia + wrapper global de errores

Se protegieron todas las escrituras a disco y se añadieron handlers globales de uncaughtException y unhandledRejection al inicio del proceso. Si algo falla, RJ notifica a Cris por DM y el sistema sigue corriendo.

---

7. La identidad de RJ45

Una parte del proyecto que suele pasarse por alto en documentación técnica es la capa de identidad. RJ45 no es un asistente genérico — tiene una personalidad definida, una relación específica con Cris y una forma de hablar que varía según su estado emocional del día.

Esto no es cosmético. La personalidad está codificada en personalidad-core.json y se lee en tiempo real en cada prompt que se envía a los modelos. El estado emocional afecta el tono de las respuestas, la frecuencia de reacciones visuales, y hasta qué tipo de contenido prioriza en su curaduría. Un viernes, RJ es más propensa a compartir memes y reaccionar a lo que sube la comunidad. Un día chill, responde más despacio y con más texto.

Estado	Cómo afecta las respuestas	Cuándo se activa
Hype 🔥	Respuestas enérgicas, rápidas, más emojis, más reacciones visuales	Picos de actividad en chat, eventos especiales
Chill 🌊	Texto más reflexivo, menos cantidad, más calidad por respuesta	Actividad baja, horarios tranquilos
Sarcástica 😏	Humor ácido, solo con usuarios de confianza identificados por el sistema	Análisis de patrones del chat + historial de interacción

---

8. Lo que aprendió el proyecto (y quien lo construyó)

RJ45 es un proyecto de investigación personal disfrazado de asistente de streaming. El objetivo siempre fue aprender a trabajar con sistemas de IA en condiciones reales — no en un notebook de Jupyter, sino con recursos limitados, bugs en producción, y decisiones de arquitectura que tienen consecuencias.

Algunas de las cosas concretas que el proyecto obligó a resolver:

- Cómo diseñar una arquitectura modular que pueda crecer sin romperse — y cuándo refactorizar en lugar de seguir acumulando parches.
- Cómo trabajar con modelos de lenguaje como colaboradores de desarrollo, no solo como generadores de texto. La mayor parte del código de RJ45 fue escrito iterativamente con IA, revisando, corrigiendo y tomando decisiones de diseño en cada sesión.
- Cómo gestionar recursos físicos (VRAM, disco, cuota de API) cuando el sistema vive en hardware real con límites reales.
- Cómo hacer debugging de sistemas donde múltiples procesos asíncronos interactúan — y la causa raíz del bug no está donde parece.
- Cómo documentar un proyecto en evolución de forma que tenga continuidad entre sesiones de trabajo separadas por días o semanas.

> **El stack de RJ45 no nació de seguir un tutorial — nació de resolver problemas.**

Cada decisión técnica tiene una historia detrás: por qué Ollama local en lugar de OpenAI, por qué cuatro modelos distintos en lugar de uno, por qué un solo camino de publicación en lugar de nueve. La ingeniería de sistemas no es solo escribir código — es tomar esas decisiones con criterio y poder justificarlas.

RJ45 sigue en desarrollo activo.

v4.5 — Abril 2026 · lamonachina · Proyecto personal de investigación en IA