

UNIVERSIDAD MARIANO GÁLVEZ DE GUATEMALA



Ingeniería en sistemas de información y ciencias de la computación.

ARQUITECTURA DE COMPUTADORAS II

Bryan Steve Wielman Sandoval 0900-19-10273
Diego Alejandro Cabrera Guerra 0900-14-13435
Jacqueline Paola Velasquez Tambriz 0900-16-9735
Merlin Yessenia Lopez Ramirez 090-21-11430

GUATEMALA, GUATEMALA

Resumen ejecutivo

El proyecto parte de un problema bastante sencillo. Hoy por hoy, cuando una empresa quiere utilizar inteligencia artificial, normalmente lo que hace es consumir un servicio que se encuentra en la nube. El usuario hace una consulta, esa consulta sale de la infraestructura de la empresa, llega al proveedor, el proveedor la procesa y finalmente devuelve una respuesta. Esto funciona, pero nos genera dependencia de Internet, posibles costos por consumo, límites en la cantidad de solicitudes y, más importante todavía, tenemos que considerar qué información estamos enviando hacia un tercero.

Entonces, la idea del proyecto fue comprobar qué sucede si parte de ese procesamiento lo acercamos al usuario y lo ejecutamos localmente. Ahí es donde utilizamos el concepto de Edge Computing.

La solución está formada por tres mini PC Dell OptiPlex 3060 Micro conectadas por medio de una red Ethernet Gigabit. Tenemos un equipo principal, que vamos a llamar Master, y dos equipos de procesamiento, Worker1 y Worker2. En el Master se encuentra el portal web desarrollado en Python con Flask y también el balanceador de solicitudes. Los dos Workers ejecutan el modelo de inteligencia artificial mediante llama.cpp.

El modelo seleccionado es Qwen 2.5 de 1.5 mil millones de parámetros en formato GGUF y cuantización Q4_K_M. ¿Por qué utilizar un modelo de este tipo? Porque tenemos hardware limitado y no tenemos una GPU dedicada. Entonces necesitamos un modelo que pueda cargarse en memoria y ejecutarse directamente sobre CPU sin que el requerimiento de recursos sea demasiado alto. Informe_Tecnico_Edge_Computing...

El usuario no necesita conocer toda esta arquitectura. Él simplemente abre una página web, escribe una pregunta y la envía. El Master recibe esa consulta y el balanceador decide cuál de los dos Workers la va a procesar. Para esta distribución se utiliza round-robin, es decir, una solicitud se envía a Worker1, la siguiente a Worker2 y así sucesivamente.

Finalmente, el Worker procesa la consulta utilizando el modelo local, devuelve la respuesta al Master y el portal se la presenta nuevamente al usuario.

Además, tenemos una parte de monitoreo que nos permite observar cuántas solicitudes se han realizado, cuál Worker atendió la última solicitud, cuántas solicitudes ha procesado cada Worker y cuánto tiempo tomó generar una respuesta.

El alcance actual del proyecto es demostrar consultas de texto ejecutadas de manera local. No estamos diciendo que ya tenemos un asistente empresarial completo. No tenemos reconocimiento de voz, calendarios, recordatorios, acceso automático a manuales empresariales ni autenticación empresarial. Esas características forman parte de una posible evolución del proyecto.

Palabras clave

Edge Computing, inteligencia artificial local, modelo de lenguaje, Qwen 2.5, llama.cpp, Linux, clúster, balanceador, Flask, inferencia, CPU.

Índice

Resumen ejecutivo	2
Palabras clave.....	3
Índice.....	4
Introducción	7
Planteamiento del problema.....	8
Justificación	9
Objetivos	10
Antecedentes y marco conceptual.....	10
Computación en la nube.....	10
Edge Computing	11
Modelos de lenguaje grandes y pequeños.....	11
Qwen 2.5 1.5B	12
Cuantización	12
llama.cpp y GGUF	13
Balanceo round-robin.....	13
Linux y servicios de red.....	14
Monitoreo básico	14
Requerimientos y alcance	15
Requerimientos funcionales.....	15

Requerimientos no funcionales	16
Alcance incluido	16
Elementos fuera del alcance.....	16
Supuestos y restricciones	17
Diseño de la solución	17
Vista general de arquitectura.....	17
Diagrama lógico	18
Diseño de red	19
Decisión de usar conexión cableada	19
Diseño de monitoreo	20
Punto único de fallo	20
. Implementación técnica	21
Hardware.....	21
Software	21
Configuración de los Workers.....	22
Balanceador.....	22
Portal web	23
Inicio de los componentes.....	23
Consideraciones de almacenamiento y memoria.....	23
Uso de CPU.....	24

Funcionamiento desde el punto de vista del usuario	24
Acceso al portal.....	24
Envío de una consulta	24
Generación de respuesta	25
Visualización del estado.....	25
Visualización de estadísticas.....	25
Ejemplo de caso de uso	26
Capturas recomendadas	26
Plan de pruebas y resultados	29
Matriz de pruebas funcionales	29
Evidencia de respuesta del modelo	30
Estadísticas de ejemplo observadas	30
Procedimiento sugerido para repetir pruebas.....	30
Tabla para resultados finales	31
Analisis de resultados	32
Seguridad, privacidad y operación.....	34
Limitaciones del prototipo	36
Alcance real frente a visión futura	36
Conclusiones	37

Introducción

La inteligencia artificial generativa hoy por hoy se utiliza para una cantidad bastante grande de actividades. Podemos utilizarla para consultar información, redactar documentos, apoyar procesos, generar contenido, resolver dudas, etcétera, etcétera, etcétera.

El problema es que la mayoría de estos servicios funcionan en infraestructura de terceros. Básicamente nosotros enviamos una solicitud por Internet, alguien más ejecuta el modelo y nos devuelve el resultado.

Esto es práctico porque nosotros no necesitamos preocuparnos por el hardware que está detrás. Sin embargo, tenemos algunos elementos que tenemos que considerar: necesitamos conectividad, podemos tener límites de consumo, podemos tener costos variables y también existe el tema de qué información estamos enviando fuera de nuestra infraestructura.

Entonces aparece Edge Computing.

¿Qué es Edge Computing para este proyecto? Básicamente es acercar el procesamiento al lugar donde se está utilizando la información. En lugar de que absolutamente todas las consultas tengan que viajar hacia un centro de datos remoto, vamos a tener una infraestructura local que tenga capacidad de ejecutar esas consultas.

En nuestro caso, esa infraestructura está compuesta por tres mini PC con Ubuntu Server. Tenemos una red cableada, un modelo de lenguaje cuantizado y diferentes componentes de software que nos permiten separar la interfaz del usuario, la distribución de las solicitudes y el procesamiento del modelo.

El objetivo no es competir contra un modelo de nube de gran tamaño. Ese no es el alcance. Lo que queremos demostrar es que sobre hardware convencional podemos ejecutar un modelo local, distribuir solicitudes y presentar el servicio al usuario por medio de una interfaz sencilla.

Planteamiento del problema

El problema comienza cuando una organización depende completamente de un servicio remoto de inteligencia artificial.

¿Qué sucede si no tenemos acceso al servicio? No tenemos inteligencia artificial.

¿Qué sucede si se terminan los tokens, cambia la política de consumo o aumenta el costo? También tenemos una dependencia.

¿Qué sucede si la información que queremos consultar pertenece a la empresa? Tenemos que analizar si realmente queremos enviar esos datos hacia un tercero.

Entonces, el problema no es que la nube sea mala. La nube funciona perfectamente y probablemente nos permita utilizar modelos muchísimo más grandes. El problema es depender exclusivamente de ella para cualquier consulta que queramos realizar.

Otro escenario es el de una persona nueva que entra a una empresa. Esta persona seguramente va a tener preguntas acerca de procedimientos, herramientas, procesos y actividades. Normalmente va a preguntarle a otro empleado o a un supervisor. Eso genera interrupciones y también dependencia de otras personas.

La idea sería que algunas consultas iniciales puedan resolverse por medio de un asistente local.

En nuestro prototipo todavía no conectamos manuales empresariales. Entonces hay que tener claro eso. Lo que demostramos es la infraestructura necesaria para que, en una evolución futura, podamos incorporar ese tipo de información.

La pregunta principal sería entonces: ¿cómo podemos implementar un asistente de inteligencia artificial que procese consultas de texto localmente, utilizando hardware convencional, y que además pueda distribuir esas solicitudes entre diferentes nodos de procesamiento?

La solución que proponemos es tener un Master y dos Workers.

El Master recibe al usuario y coordina las solicitudes.

Los Workers son los que realmente ejecutan el modelo.

De esta manera no concentramos todo el procesamiento de inferencia en un único nodo y podemos demostrar una arquitectura distribuida.

Justificación

¿Por qué hacer este proyecto?

Primero, porque nos permite utilizar conceptos que hemos visto a lo largo del curso en un escenario real.

Tenemos CPU, memoria principal, almacenamiento secundario, sistema operativo Linux, red, servicios escuchando en puertos, comunicación cliente-servidor y distribución de procesamiento.

La cuantización del modelo también es importante. Si nosotros intentáramos ejecutar un modelo demasiado grande en estos equipos, probablemente no podríamos alojarlo correctamente en memoria o el rendimiento sería demasiado bajo.

Por eso utilizamos Qwen 2.5 de 1.5B cuantizado en Q4_K_M. Básicamente estamos reduciendo la precisión con la cual representamos los parámetros para poder reducir el tamaño del modelo y hacerlo más manejable sobre el hardware que tenemos.

También tenemos una justificación económica. Estamos utilizando hardware convencional que puede ser reutilizado. Eso no quiere decir que ejecutar inteligencia artificial localmente sea gratis. Tenemos electricidad, mantenimiento, almacenamiento, administración y soporte. Lo que eliminamos es el pago directo por cada consulta realizada a una API externa.

Finalmente está el tema de privacidad.

Si la consulta se procesa dentro de nuestra red, y nuestra aplicación no reenvía esa información hacia otro servicio, entonces podemos mantener esos datos dentro de nuestra infraestructura. Eso disminuye una vía de exposición, aunque obviamente no significa que automáticamente tengamos un sistema seguro. Todavía necesitamos controles de acceso, segmentación, registros y otras medidas si esto llegara a utilizarse en producción.

Objetivos

El objetivo general es diseñar e implementar un prototipo de asistente de inteligencia artificial local basado en Edge Computing, utilizando tres mini PC físicas y ejecutando un modelo de lenguaje directamente sobre CPU.

Para lograrlo necesitamos configurar los tres equipos con Ubuntu Server, establecer comunicación mediante Ethernet, ejecutar el mismo modelo en los dos Workers, crear un servicio de inferencia en cada uno, implementar un balanceador en el Master y finalmente proporcionar al usuario un portal web desde el cual pueda hacer las consultas.

También queremos agregar monitoreo básico. No tiene sentido tener varios equipos y no saber qué está sucediendo con ellos. Por eso el sistema registra solicitudes, distribución entre Workers y tiempos de respuesta.

Finalmente, necesitamos documentar qué funcionó, qué no funcionó y hasta dónde puede llegar la solución actual.

Antecedentes y marco conceptual

Computación en la nube

La computación en la nube, básicamente, es la capacidad que tenemos hoy por hoy de utilizar procesamiento, almacenamiento y diferentes servicios sin necesidad de tener físicamente toda esa infraestructura con nosotros. Entonces, en lugar de que la empresa compre todos los servidores, los configure y los administre localmente, lo que hace es consumir esos recursos a través de la red.

En el caso de inteligencia artificial funciona de una manera bastante sencilla. El usuario envía una solicitud, esa solicitud llega a la infraestructura del proveedor, ahí se ejecuta el modelo y finalmente se devuelve una respuesta. ¿Cuál es la ventaja? Que nosotros podemos utilizar modelos bastante grandes sin tener que comprar el hardware necesario para ejecutarlos localmente.

Ahora, obviamente esto tiene sus ventajas y sus consideraciones. La ventaja principal es que podemos escalar recursos, podemos acceder al servicio desde diferentes lugares y no tenemos que preocuparnos directamente por mantener toda la infraestructura física. Pero también

dependemos de la conectividad, dependemos de las políticas y costos del proveedor, tenemos que gestionar credenciales y, más importante todavía, tenemos que tener claro qué información estamos enviando hacia esa infraestructura externa.

Edge Computing

Básicamente, Edge Computing lo que busca es acercar el procesamiento al lugar donde nosotros tenemos los datos o donde necesitamos obtener el resultado. En lugar de mandar absolutamente todas las solicitudes hacia un centro de datos que puede estar geográficamente lejos, lo que hacemos es colocar parte de ese procesamiento mucho más cerca del usuario.

En nuestro proyecto esto se da utilizando las mini PC. El modelo está ejecutándose directamente sobre estos equipos y la comunicación entre ellos se realiza por medio de Ethernet. Entonces, el usuario entra al portal local, realiza una consulta y esa consulta se procesa dentro de la infraestructura que nosotros configuramos.

¿Cuál es la ventaja? Que disminuimos la dependencia de un servicio remoto para las consultas que nuestro modelo pueda resolver. Ahora, eso no quiere decir que automáticamente vamos a tener cero latencia. El tiempo de respuesta todavía va a depender del modelo que estamos utilizando, del procesador, de la memoria disponible, de la red y también de cuánto texto tenga que generar como respuesta.

Modelos de lenguaje grandes y pequeños

Un modelo de lenguaje, básicamente, lo que hace es utilizar una cantidad bastante grande de parámetros y relaciones matemáticas para poder predecir cuál es el siguiente token que debería de generar dependiendo de la entrada que recibió.

Entonces, nosotros escribimos una consulta, el modelo la procesa y por medio de esos cálculos comienza a generar token por token hasta construir una respuesta.

Ahora, ¿por qué existen modelos grandes y modelos pequeños? Porque obviamente entre más grande sea el modelo, normalmente vamos a necesitar más memoria y más capacidad de procesamiento para poder ejecutarlo.

Un modelo pequeño o compacto disminuye esos requerimientos, aunque también puede perder capacidad frente a un modelo mucho más grande. Para nuestro proyecto eso es importante porque estamos trabajando sobre CPU y sobre hardware limitado. Entonces no tendría mucho sentido tratar de cargar un modelo enorme que probablemente ni siquiera podamos alojar correctamente en memoria.

Qwen 2.5 1.5B

Para nuestro prototipo seleccionamos Qwen 2.5 de 1.5 mil millones de parámetros.

¿Por qué este modelo? Porque necesitamos algo que sea suficientemente pequeño para poder ejecutarlo sobre el hardware que tenemos, pero que al mismo tiempo sea capaz de generar respuestas de texto para demostrar el funcionamiento del proyecto.

El modelo está almacenado en formato GGUF y utiliza cuantización Q4_K_M. Esto nos permite ejecutarlo utilizando llama.cpp.

Ahora, el hecho de que un modelo tenga 1.5 mil millones de parámetros no quiere decir automáticamente que sea rápido o que sea mejor que otro. El rendimiento también depende del procesador que tengamos, de las instrucciones disponibles en la CPU, de la cantidad de memoria, de la configuración con la que levantemos el servidor, del contexto que le enviamos y de la cantidad de tokens que tiene que generar.

Entonces, el número de parámetros nos sirve como referencia del tamaño, pero no es el único elemento que determina el rendimiento.

Cuantización

La cuantización, básicamente, es una forma de reducir la precisión con la que nosotros almacenamos los valores numéricos del modelo.

Un modelo tiene una cantidad enorme de parámetros y cada parámetro necesita ocupar espacio en memoria y almacenamiento. Entonces, si nosotros logramos representar esos valores utilizando menos bits, reducimos el tamaño del modelo.

¿Cuál es la ventaja? Necesitamos menos memoria y menos almacenamiento para poder ejecutarlo.

¿Cuál es el costo? Que estamos perdiendo algo de precisión en esos valores y eventualmente eso puede afectar la calidad de algunas respuestas.

En nuestro caso utilizamos Q4_K_M, que es una cuantización aproximadamente de cuatro bits utilizada dentro del formato GGUF. Y precisamente esta reducción es una de las razones por las cuales podemos ejecutar el modelo sobre los equipos que tenemos disponibles.

llama.cpp y GGUF

llama.cpp es básicamente el software que nos permite ejecutar el modelo sobre nuestro hardware.

La ventaja es que podemos utilizar modelos de lenguaje en diferentes plataformas y configuraciones sin depender necesariamente de una GPU dedicada.

Ahora, el modelo tiene que estar almacenado de alguna manera, y ahí es donde entra GGUF.

GGUF es el formato en el cual tenemos guardado el modelo y también parte de la información necesaria para poder cargarlo correctamente.

Entonces, en nuestro proyecto tenemos el archivo GGUF de Qwen 2.5 y utilizamos llama.cpp para cargar ese archivo y ejecutar la inferencia.

Además, llama.cpp nos permite levantar llama-server. ¿Para qué nos sirve eso? Para que cada Worker pueda exponer el modelo como un servicio HTTP.

De esa manera el Master no necesita ejecutar directamente el modelo. Simplemente envía una solicitud al Worker, el Worker procesa la consulta y devuelve la respuesta.

Balanceo round-robin

Round-robin es probablemente una de las formas más sencillas de distribuir solicitudes entre varios nodos.

¿Qué hace? Simplemente va alternando.

Si tenemos Worker1 y Worker2, una solicitud se manda a Worker1, la siguiente a Worker2, después nuevamente Worker1 y así sucesivamente.

La ventaja es que es muy fácil de implementar y nos permite demostrar que ambos equipos están participando en el procesamiento.

Ahora, tiene una limitación bastante clara.

Round-robin no está revisando cuánto CPU está utilizando cada Worker, cuánto tiempo lleva procesando una consulta o cuántas tareas tiene pendientes.

Entonces puede suceder que Worker1 todavía esté trabajando y el algoritmo simplemente vuelva a enviarle otra solicitud porque le corresponde dentro de la secuencia.

Para nuestro prototipo funciona porque queremos demostrar distribución. Para un ambiente de producción seguramente tendríamos que utilizar un mecanismo que tome en cuenta disponibilidad o carga real.

Linux y servicios de red

En los tres equipos utilizamos Ubuntu Server.

¿Por qué Server? Porque no necesitamos toda una interfaz gráfica para ejecutar los servicios que requiere el proyecto.

Lo que necesitamos es un sistema operativo Linux que nos permita levantar nuestros procesos, configurar la red y mantener los servicios ejecutándose.

Cada servicio utiliza un puerto determinado.

Por ejemplo, una dirección IP nos dice en qué equipo de la red está el servicio, pero el puerto nos dice exactamente a qué aplicación dentro de ese equipo queremos conectarnos.

Entonces podemos tener un Worker con una única dirección IP y dentro de ese mismo equipo tener diferentes servicios escuchando en diferentes puertos.

Eso es importante porque el hecho de que un equipo responda a ping no quiere decir que el servicio esté disponible.

El equipo puede estar perfectamente conectado a la red, pero si llama-server no está ejecutándose en el puerto correspondiente, nosotros no vamos a poder hacer una inferencia.

Monitoreo básico

El monitoreo nos permite saber qué está sucediendo con nuestra infraestructura.

Si tenemos varios servicios y varios equipos, no podemos simplemente asumir que todo está funcionando porque vemos las computadoras encendidas.

Necesitamos alguna manera de verificar si los servicios realmente responden.

Por eso el prototipo incorpora comprobaciones de salud y estadísticas.

Podemos observar si el balanceador está disponible, si Worker1 responde, si Worker2 responde, cuántas solicitudes se han procesado, cuál fue el último Worker utilizado y cuánto tiempo tomó determinada solicitud.

Esto nos da una visión básica del comportamiento de la solución.

Ahora, tampoco vamos a decir que esto es una plataforma completa de observabilidad. No estamos almacenando métricas históricas avanzadas ni haciendo análisis profundo de CPU, memoria, errores, etcétera, etcétera, etcétera.

Lo que tenemos es suficiente para la demostración porque nos permite comprobar que los componentes están funcionando y observar cómo se están distribuyendo las solicitudes.

Requerimientos y alcance

Requerimientos funcionales

Código	Requerimiento	Estado
RF-01	El usuario puede abrir el portal desde un navegador en la red local.	Implementado
RF-02	El usuario puede enviar una consulta de texto desde el portal.	Implementado
RF-03	El portal envía la solicitud al balanceador del Master.	Implementado
RF-04	El balanceador alterna entre Worker1 y Worker2 mediante round-robin.	Implementado
RF-05	Cada Worker ejecuta un servidor de inferencia con el modelo local.	Implementado; depende de iniciar el servicio
RF-06	El portal muestra el estado del balanceador y de ambos Workers.	Implementado
RF-07	El sistema presenta estadísticas de solicitudes y tiempos de respuesta.	Implementado
RF-08	El asistente reconoce voz y convierte audio a texto.	Fuera del alcance actual
RF-09	El asistente consulta automáticamente manuales privados de la empresa.	Trabajo futuro

RF-10	El sistema administra calendarios y crea recordatorios.	Fuera del alcance actual
-------	---	--------------------------

Requerimientos no funcionales

Código	Requerimiento	Criterio
RNF-01	Procesamiento local	La inferencia de texto debe realizarse en los Workers configurados.
RNF-02	Disponibilidad observable	El portal debe mostrar el estado de los servicios consultados.
RNF-03	Mantenibilidad	Los componentes deben poder iniciarse y verificarse por separado.
RNF-04	Conectividad	Los tres equipos deben comunicarse por la red Ethernet del proyecto.
RNF-05	Uso de recursos	El modelo debe caber en la memoria disponible de los nodos.
RNF-06	Seguridad	La versión empresarial debe incorporar controles de acceso y protección de red.

Alcance incluido

- Infraestructura de tres mini PC físicos conectados por un switch Gigabit Ethernet.
- Ubuntu Server en los equipos.
- Modelo Qwen 2.5 1.5B GGUF Q4_K_M ejecutado mediante llama.cpp en Worker1 y Worker2.
- Portal web con Flask en el Master.
- Balanceador HTTP con distribución round-robin.
- Endpoint de salud para verificar la disponibilidad de los componentes.
- Panel de estadísticas básico.
- Pruebas manuales de conectividad y consultas.

Elementos fuera del alcance

- Reconocimiento de voz o interacción por comandos de voz.
- Integración con calendarios, recordatorios, correo o sistemas ERP.
- Recuperación aumentada por generación (RAG) con manuales privados.
- Autenticación empresarial, roles, auditoría de usuarios y gestión avanzada de secretos.
- Alta disponibilidad del Master o recuperación automática ante fallos.
- Escalado automático según CPU, memoria o longitud de la cola.
- Pruebas de rendimiento científicas con una muestra grande y condiciones controladas.
- Despliegue productivo con acuerdos de nivel de servicio.

Supuestos y restricciones

Para que el sistema funcione correctamente necesitamos que los dos Workers tengan activo su servicio de inferencia y que exista comunicación TCP entre el Master y los puertos que configuramos en cada nodo. Hay que tomar en cuenta que estamos trabajando con equipos de recursos limitados y que todo el procesamiento del modelo se está realizando directamente sobre CPU, así que el rendimiento también va a depender de la capacidad de cada equipo. Además, tanto la interfaz como el balanceador tienen que permanecer activos para que el usuario pueda enviar consultas y recibir una respuesta. En la arquitectura actual, el Master es un punto único de fallo, porque ahí tenemos concentrado el portal y el balanceador; entonces, si el Master deja de funcionar, aunque Worker1 y Worker2 continúen activos, el usuario ya no tendría acceso al sistema.

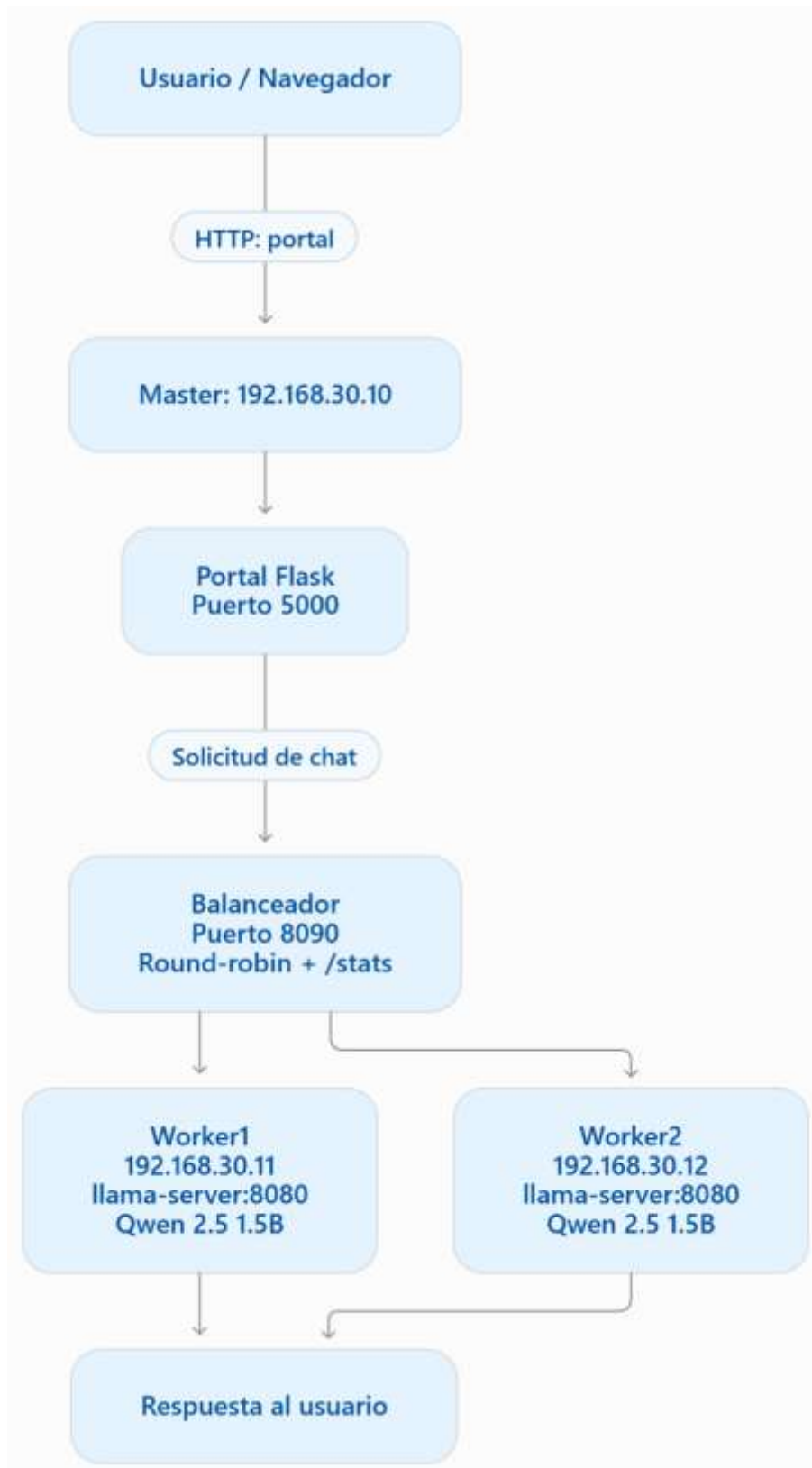
Diseño de la solución

Vista general de arquitectura

La arquitectura la dividimos en cuatro capas para tener claro qué función cumple cada parte del sistema. La primera es la capa de presentación, que es el portal web con el que interactúa el usuario. Luego tenemos la capa de coordinación, que corresponde al balanceador y es la que decide a cuál Worker se va a enviar cada solicitud. Después está la capa de inferencia, que es donde realmente se ejecuta el modelo en Worker1 y Worker2. Y finalmente tenemos la red física, que es la que permite que todos estos equipos puedan comunicarse entre sí por medio de Ethernet.

Componente	Equipo	Responsabilidad	Puerto / dirección
Portal Flask	Master	Interfaz web y rutas de API	TCP 5000; 192.168.30.10
Balanceador	Master	Distribuir solicitudes y registrar estadísticas	TCP 8090
Servidor de inferencia	Worker1	Ejecutar Qwen 2.5 mediante llama.cpp	TCP 8080; 192.168.30.11
Servidor de inferencia	Worker2	Ejecutar Qwen 2.5 mediante llama.cpp	TCP 8080; 192.168.30.12
Switch	Infraestructura física	Conectar los equipos por Ethernet	Gigabit Ethernet

Diagrama lógico



Diseño de red

Equipo	Dirección IPv4	Máscara	Función
Master	192.168.30.10	255.255.255.0 (/24)	Portal y balanceador
Worker1	192.168.30.11	255.255.255.0 (/24)	Inferencia de IA
Worker2	192.168.30.12	255.255.255.0 (/24)	Inferencia de IA
PC de administración (prueba)	192.168.30.20 o .21	255.255.255.0 (/24)	Acceso al portal

Todos los equipos que forman parte del proyecto trabajan dentro de la red privada 192.168.30.0/24. La dirección del equipo que se utilice para administrar o hacer la demostración puede cambiar dependiendo de qué computadora se conecte en ese momento, pero siempre debe estar dentro del mismo segmento de red para poder comunicarse con los demás dispositivos. Además, la red del proyecto se mantiene separada de la red Wi-Fi institucional, porque para acceder al portal no necesitamos hacer ningún puente entre ambas redes; mientras el equipo cliente esté conectado directamente a la LAN del proyecto, va a poder comunicarse con el Master y acceder al sistema sin ningún inconveniente.

Decisión de usar conexión cableada

Decidimos utilizar Ethernet Gigabit porque necesitamos una conexión estable entre los equipos y un ancho de banda suficiente para poder mover las solicitudes y las respuestas sin agregar problemas innecesarios durante las pruebas. Aunque la inferencia realmente se ejecuta en los Workers, el Master tiene que estar comunicándose constantemente con ellos para enviar el prompt y recibir la respuesta generada. Entonces, al utilizar una conexión cableada reducimos problemas de latencia, interferencia o variaciones de conectividad que sí podríamos tener en una red inalámbrica.

Diseño del flujo de datos

1. El usuario abre la dirección local del portal desde el navegador.
2. El navegador envía una consulta de texto al endpoint de chat del portal.
3. El portal reenvía la consulta al balanceador del Master.
4. El balanceador selecciona Worker1 o Worker2 siguiendo round-robin.
5. El servidor del Worker utiliza el modelo local para generar la respuesta.

6. El Worker devuelve la respuesta al balanceador.
7. El balanceador registra las métricas correspondientes a la solicitud exitosa y devuelve el resultado.
8. El portal presenta la respuesta en la interfaz y el panel de estadísticas actualiza sus valores periódicamente.

Diseño de monitoreo

El monitoreo básicamente nos permite saber si los componentes realmente están disponibles y cómo se está comportando el sistema durante las pruebas. El portal consulta una ruta de salud para verificar si el balanceador y los dos Workers están respondiendo. Además, consulta el endpoint `/api/stats`, que obtiene la información directamente del endpoint `/stats` del balanceador. Con esto podemos observar cuántas solicitudes se han procesado correctamente, cuántas atendió cada Worker, cuál fue el último Worker utilizado, cuánto tardó la última solicitud y cuál es el tiempo promedio registrado.

Ahora, hay que tener claro que este monitoreo todavía es bastante básico. El promedio únicamente se calcula con las solicitudes exitosas que se han registrado desde que se inició el balanceador. Entonces, ese valor no necesariamente representa todo el tiempo que el usuario percibe desde el navegador y tampoco incluye solicitudes que hayan fallado si esas solicitudes no se están contabilizando dentro de las métricas.

Punto único de fallo

En la arquitectura actual tenemos un punto único de fallo y ese punto es el Master. ¿Por qué? Porque en el Master tenemos tanto el portal como el balanceador. Entonces, si ese equipo deja de funcionar, el usuario ya no puede acceder al sistema aunque Worker1 y Worker2 continúen encendidos y ejecutando el modelo.

De la misma manera, si los dos Workers dejan de responder, el Master puede seguir funcionando, el portal puede seguir activo y el balanceador también, pero no vamos a tener ningún nodo disponible para generar una respuesta. Para una demostración académica esto es suficiente, pero si estuviéramos hablando de un ambiente de producción tendríamos que agregar mecanismos de redundancia y alta disponibilidad para evitar depender de un único equipo o servicio.

. Implementación técnica

Hardware

Elemento	Especificación conocida	Uso en el proyecto
Mini PC 1	Dell OptiPlex 3060 Micro; Intel Core i5 8. ^a generación; RAM 8 GB; SSD 256 GB	Master
Mini PC 2	Dell OptiPlex 3060 Micro; Intel Core i5 8. ^a generación; RAM 8 GB; SSD 256 GB	Worker1
Mini PC 3	Dell OptiPlex 3060 Micro; Intel Core i5 8. ^a generación; RAM 8 GB; SSD 256 GB	Worker2
Switch	TP-Link TL-SG1210MPE Gigabit Easy Smart PoE+	Interconexión Ethernet
Equipo cliente	PC con navegador y acceso a la red del proyecto	Acceder al portal

Software

Software / componente	Uso	Ubicación
Ubuntu Server	Sistema operativo Linux de los nodos	Master, Worker1 y Worker2
Python	Lenguaje usado para el portal y balanceador	Master
Flask	Framework web para el portal y endpoints API	Master
llama.cpp	Ejecución del modelo y servidor HTTP de inferencia	Workers; también existen binarios en Master
Qwen 2.5 1.5B GGUF Q4_K_M	Modelo de lenguaje para generar texto	Master y Workers según la instalación descrita; inferencia activa en Workers
Navegador web	Interfaz para el usuario	Equipo cliente

Configuración de los Workers

En Worker1 y Worker2 se encuentra el ejecutable llama-server y el archivo GGUF del modelo. El servidor debe iniciarse escuchando en 0.0.0.0, puerto 8080, para que pueda recibir solicitudes desde la red local. La ruta de las bibliotecas compartidas se incorpora a LD_LIBRARY_PATH para que el ejecutable pueda cargar los archivos .so incluidos en el paquete.

Ejemplo de configuración de entorno en cada Worker:

```
export LD_LIBRARY_PATH="$HOME/paquete-worker/bin:$LD_LIBRARY_PATH"
```

Ejemplo del comando de inicio utilizado:

```
~/paquete-worker/bin/llama-server -m ~/modelos/qwen2.5-1.5b-instruct-q4_k_m.gguf --  
host 0.0.0.0 --port 8080
```

***Nota operativa:** estos comandos deben ejecutarse en la terminal del Worker correspondiente. En el estado actual, el servicio permanece activo mientras el proceso siga ejecutándose; para una instalación permanente se recomienda configurar un servicio systemd.*

Balanceador

El balanceador se encuentra en el Master y se ejecuta por medio de Python utilizando el puerto 8090. ¿Qué hace este balanceador? Recibe las solicitudes que vienen desde el portal y decide a cuál de los dos Workers se va a enviar cada consulta. Para eso tenemos dos endpoints de inferencia, uno correspondiente a Worker1 en 192.168.30.11:8080 y otro correspondiente a Worker2 en 192.168.30.12:8080.

La estrategia que utilizamos es round-robin, entonces las solicitudes se van alternando entre los dos equipos. Una puede ir a Worker1, la siguiente a Worker2 y así sucesivamente. Además, el balanceador tiene una ruta de salud y una ruta de estadísticas que nos permiten revisar cuántas solicitudes se procesaron correctamente, cuántas atendió cada Worker, cuál fue el último nodo utilizado y cuánto tiempo tardaron las solicitudes.

Ahora, tampoco vamos a decir que este balanceador hace una gestión avanzada de carga. No está revisando en tiempo real cuánto CPU tiene disponible cada Worker ni está haciendo una tolerancia completa a fallos. Para el alcance actual del proyecto, lo que necesitamos es demostrar la distribución de solicitudes entre los dos nodos.

Portal web

El portal web está desarrollado utilizando Flask y se ejecuta directamente en el Master por medio del puerto 5000. Esta es básicamente la interfaz que va a utilizar el usuario para interactuar con todo el sistema. Desde ahí puede escribir una consulta, enviarla y visualizar la respuesta que genere el modelo.

Además de la parte de chat, el portal contiene indicadores de estado y un panel de estadísticas. La ruta `/api/health` se utiliza para verificar si el balanceador y los Workers están disponibles, `/api/chat` permite enviar la consulta hacia el balanceador y `/api/stats` recupera las estadísticas para mostrarlas en la interfaz.

El panel se actualiza cada cinco segundos. La idea es que el usuario pueda observar los cambios en las métricas después de realizar una consulta sin necesidad de estar recargando manualmente la página.

Inicio de los componentes

1. Encender los tres mini PC y verificar que la red local esté conectada.
2. Iniciar el servidor llama-server en Worker1.
3. Iniciar el servidor llama-server en Worker2.
4. En el Master, iniciar el balanceador desde la carpeta del proyecto.
5. En otra terminal del Master, iniciar la aplicación Flask del portal.
6. Desde un equipo cliente conectado a la red del proyecto, abrir `http://192.168.30.10:5000`.
7. Comprobar el estado de los dos Workers y enviar una consulta de prueba.

Consideraciones de almacenamiento y memoria

El modelo se encuentra guardado primero en almacenamiento secundario, en este caso en el SSD del equipo. Cuando nosotros iniciamos el servidor de inferencia, ese modelo tiene que cargarse en memoria para que pueda comenzar a utilizarse.

Ahora, como estamos utilizando un archivo GGUF cuantizado, logramos reducir el espacio que ocupa el modelo en comparación con una versión que utilice mayor precisión. Pero eso no quiere decir que si el archivo pesa determinada cantidad, exactamente esa misma cantidad va a ser el consumo total de RAM.

Durante la ejecución también tenemos contexto, búferes y otras estructuras que necesita llama.cpp para trabajar. Entonces, una cosa es el tamaño del archivo almacenado y otra cosa es la cantidad real de memoria principal que necesita el proceso durante la inferencia.

Uso de CPU

En este proyecto la inferencia se ejecuta directamente sobre CPU y no estamos utilizando una GPU dedicada. Esto es importante porque uno de los objetivos es demostrar que un modelo relativamente compacto puede ejecutarse utilizando hardware convencional.

Para poder demostrarlo técnicamente, durante una consulta podemos utilizar herramientas como top o htop en cada Worker y observar el comportamiento del proceso llama-server mientras el modelo está generando una respuesta.

La evidencia debería tomarse precisamente durante una inferencia real, porque lo que queremos ver es el uso del procesador mientras se están haciendo los cálculos. Y obviamente no tendría sentido colocar un porcentaje de CPU en el informe si ese valor no fue medido realmente durante una prueba reproducible.

Funcionamiento desde el punto de vista del usuario

Acceso al portal

Desde el punto de vista del usuario el sistema tiene que ser sencillo. El usuario se conecta a la red local y abre en el navegador la dirección <http://192.168.30.10:5000>, que corresponde al equipo Master y al servicio de Flask. A partir de ahí ya puede utilizar el asistente sin necesidad de abrir una terminal, conocer las direcciones IP de los Workers o entender cómo están configurados internamente los servicios.

Envío de una consulta

Una vez dentro del portal, el usuario simplemente escribe una pregunta en lenguaje natural. Por ejemplo, puede preguntar qué es Edge Computing o solicitar alguna explicación relacionada con el tema. Cuando envía el mensaje, el portal recibe esa consulta y comienza la comunicación

con el balanceador. Entonces, para el usuario es solamente escribir y enviar, aunque detrás de esa acción ya tenemos varios componentes comunicándose.

Generación de respuesta

Cuando la consulta llega al balanceador, este selecciona uno de los dos Workers y le envía la solicitud. En ese Worker tenemos el servidor de inferencia y el modelo cargado, así que ahí es donde realmente se procesa la entrada y se genera la respuesta. Después, el Worker devuelve el resultado al balanceador y finalmente el portal lo muestra nuevamente en el navegador. Entonces, el usuario únicamente ve la pregunta y la respuesta, aunque internamente la solicitud pasó por el Master, el balanceador y uno de los nodos de inferencia.

Visualización del estado

El portal también nos permite observar si los diferentes componentes están disponibles. Podemos verificar el balanceador, Worker1 y Worker2. Si alguno de los servicios no está ejecutándose, el sistema puede mostrar ese componente como no disponible.

Aquí hay algo importante: durante las pruebas comprobamos que un equipo podía responder perfectamente a un ping y aun así el servicio de inferencia no funcionar. ¿Por qué razón? Porque el ping únicamente nos confirma conectividad de red; no nos garantiza que llama-server esté ejecutándose o que el puerto HTTP esté disponible. Entonces, para decir que un Worker realmente está operativo, tenemos que comprobar también el servicio y no solamente la conectividad.

Visualización de estadísticas

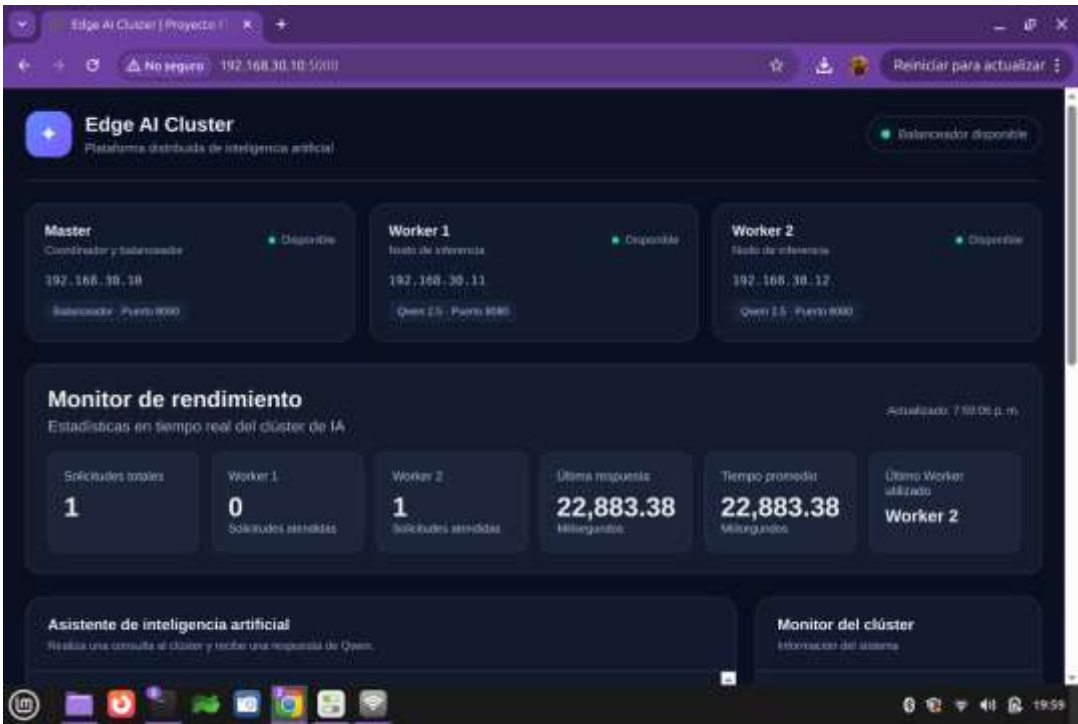
Finalmente tenemos el panel de estadísticas, que básicamente nos sirve para observar qué está pasando con las solicitudes. Ahí podemos ver cuántas consultas se han procesado correctamente, cómo se están distribuyendo entre Worker1 y Worker2, cuál fue el último Worker utilizado, cuánto tardó la última solicitud y cuál es el tiempo promedio registrado.

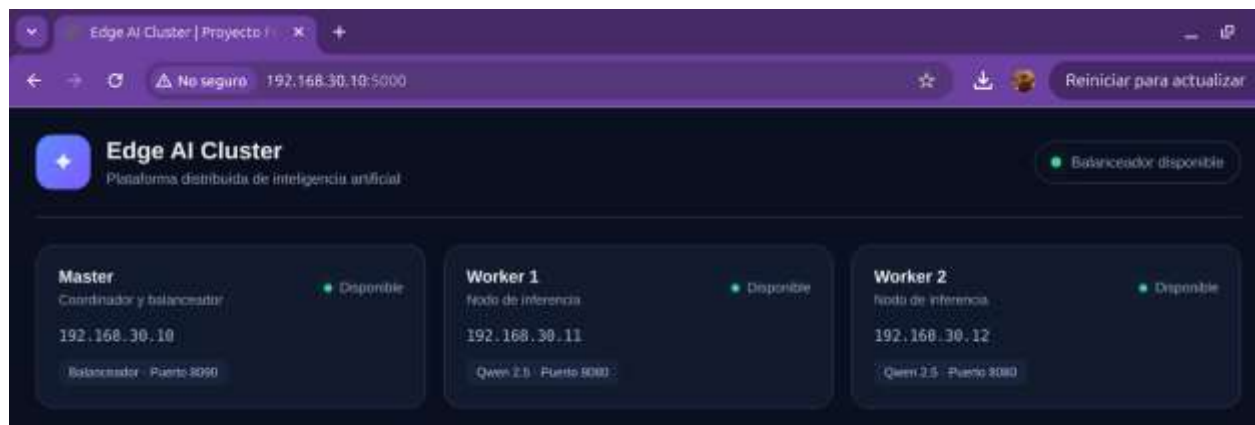
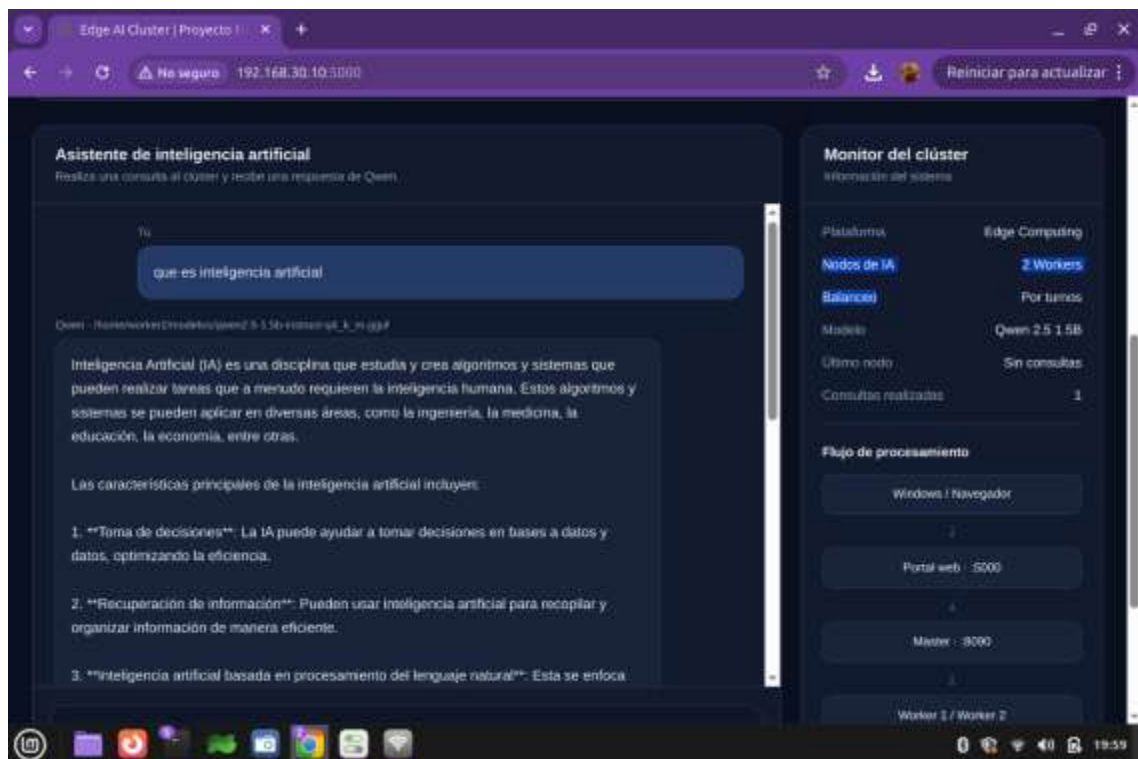
Estas cifras se van actualizando conforme el usuario realiza nuevas consultas. La idea es que no solamente podamos decir que el sistema está funcionando, sino que también podamos observar de alguna manera cómo se están utilizando los dos nodos y qué comportamiento está teniendo la solución durante las pruebas.

Ejemplo de caso de uso

Paso	Acción del usuario o sistema	Resultado esperado
1	El usuario abre el portal.	Se carga la interfaz web.
2	El usuario escribe una pregunta.	El texto queda listo para enviar.
3	El usuario envía la consulta.	El portal remite la solicitud al balanceador.
4	El balanceador selecciona un Worker.	La solicitud se dirige a uno de los servidores.
5	El modelo procesa la entrada.	Se genera una respuesta textual.
6	La respuesta vuelve al portal.	El usuario visualiza la respuesta.
7	El panel consulta estadísticas.	Se muestran los contadores y tiempos registrados.

Capturas recomendadas





```

worker1@worker1: ~
worker1@worker1:~$ ~/paquete-worker/bin/llama-server -m ~/modelos/qwen2.5-1.5b-instruct-q4_k_m
--host 0.0.0.0 --port 8080
llama_server: version 0.3.6 (alb2c3d)
llama_server: build with cc (Ubuntu 12.3.0-1ubuntu1) 12.3.0 for x86_64
llama_server: system info: n_threads = 4 | n_threads_batch = 4 | n_gga = 1 | f16_kv = 1 |
llama_server: loading model from /home/worker1/modelos/qwen2.5-1.5b-instruct-q4_k_m.gguf
llama_model_loader: loaded meta data with 34 key-value pairs and 291 tensors
llama_model_loader: Dumping model architecture:
  general.name           = qwen2.5-1.5b-instruct
  general.architecture    = qwen2
  general.parameters      = 1.52 B
  general.file_type       = 15 (Q4_K_M)
  general.quantization_version = 2
llama_model_loader: vocab size   = 151936
llama_model_loader: context size = 32768
llama_model_loader: n_embd      = 2048
llama_model_loader: n_layer     = 28
llama_model_loader: n_head      = 16
llama_model_loader: n_head_kv   = 4
llama_model_loader: successfully loaded model in 12.53 seconds
llama_server: HTTP server listening on 0.0.0.0:8080
llama_server: available endpoints:
  - POST /v1/chat/completions
  - POST /v1/completions
  - GET  /health
  - GET  /metrics
llama_server: server is ready to receive requests

```

```

worker1@worker1: ~
top - 14:23:17 up 1:12, 1 user, load average: 3.42, 2.91, 2.35
Tasks: 155 total, 2 running, 153 sleeping, 0 stopped, 0 zombie
%Cpu(s): 78.5 us, 19.8 sy, 0.0 ni, 1.3 id, 0.0 wa, 0.0 hi, 0.4 si, 0.0 st
MiB Mem : 7759.7 total, 612.1 free, 6187.3 used, 960.3 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 1238.6 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 2847 worker1   20   0 7242160 342112 84280 R 362.4   4.3   1:25.38 llama-server
 1012 worker1   20   0 225540 12380  4988 S   2.3   0.2   0:03.21 python3
 2981 worker1   20   0   9068  4640  3368 R   1.7   0.1   0:00.18 top
   732 root       20   0 217432 10284  7488 S   0.7   0.1   0:02.14 systemd
     1 root       20   0 168012 11848  8476 S   0.0   0.1   0:01.23 init
   345 root       19  -1  97892  5940  4320 S   0.0   0.1   0:00.49 systemd-journal
   412 root       20   0  28060  5200  3792 S   0.0   0.1   0:00.31 systemd-udevd
   678 root       20   0  45732  5760  4096 S   0.0   0.1   0:00.27 cron
   702 message+   20   0  11476  6608  3968 S   0.0   0.1   0:00.15 dbus-daemon
   876 root       20   0 21980  7480  4640 S   0.0   0.1   0:00.22 NetworkManager
   915 root       20   0 15728  6844  4096 S   0.0   0.1   0:00.18 sshd
 1024 worker1   20   0  44912 10244  5804 S   0.0   0.1   0:00.36 bash
 1102 worker1   20   0  37224  8832  5740 S   0.0   0.1   0:00.12 sshd
 1103 worker1   20   0 19924  7060  4332 S   0.0   0.1   0:00.08 sftp-server
 1201 root       20   0   8908  4516  3320 S   0.0   0.1   0:00.03 agetty
 1500 worker1   20   0 21832  7160  4284 S   0.0   0.1   0:00.07 systemd
 1604 root       20   0 30104  8920  5120 S   0.0   0.1   0:00.11 unattended-upgr

```

Plan de pruebas y resultados

Las pruebas funcionales nos sirven para comprobar que cada uno de los componentes realmente se está comunicando como esperamos y que el sistema completo funciona de extremo a extremo. No se trata solamente de decir que el portal abre o que los equipos responden, sino de verificar cada parte: red, servicios, balanceador, Workers, respuesta del modelo y estadísticas. En este informe se documentan las pruebas que ya se observaron durante el desarrollo, pero antes de entregar lo ideal es repetirlas y dejar evidencia por medio de capturas o salidas de terminal para demostrar cada resultado.

Matriz de pruebas funcionales

ID	Prueba	Procedimiento	Resultado observado / esperado	Estado
P-01	Conectividad IP	Ejecutar ping desde Master hacia Worker1 y Worker2.	Se observó respuesta ICMP; esto valida conectividad IP básica.	Realizada
P-02	Disponibilidad HTTP Worker1	Consultar <code>http://192.168.30.11:8080/health</code> .	Al estar activo llama-server, se espera respuesta HTTP correcta.	Realizada; repetir para evidencia
P-03	Disponibilidad HTTP Worker2	Consultar <code>http://192.168.30.12:8080/health</code> .	Al estar activo llama-server, se espera respuesta HTTP correcta.	Realizada; repetir para evidencia
P-04	Respuesta del modelo	Enviar una consulta de texto a través del balanceador.	Se obtuvo una respuesta con contenido generado por el modelo.	Realizada
P-05	Portal y API	Consultar <code>/api/health</code> y <code>/api/stats</code> .	El portal devuelve el estado de servicios y estadísticas.	Realizada
P-06	Distribución round-robin	Enviar varias solicitudes y revisar conteos por Worker.	Los conteos deben mostrar solicitudes dirigidas a ambos nodos.	Repetir y documentar
P-07	Actualización del panel	Enviar consultas y esperar la actualización periódica.	Las métricas se actualizan en el portal.	Realizada; insertar captura
P-08	Worker detenido	Detener controladamente un servidor y consultar su estado.	El portal debe indicar indisponibilidad del endpoint.	Pendiente de repetir/documentar
P-09	Uso de CPU	Observar <code>top</code> / <code>htop</code> durante una inferencia.	Se observa actividad del proceso llama-server.	Pendiente de capturar
P-10	Prueba de carga	Enviar múltiples solicitudes controladas y medir tiempos.	Obtener mediciones reproducibles de rendimiento.	No realizada formalmente

Evidencia de respuesta del modelo

Durante el desarrollo se hizo una prueba enviando una consulta a través del balanceador y se obtuvo una respuesta que incluía “prueba correcta”. ¿Qué nos demuestra esto? Que en ese momento el balanceador sí logró comunicarse con uno de los servidores de inferencia, enviar la solicitud y recibir correctamente la respuesta del modelo.

También tuvimos otro escenario bastante importante durante la configuración. Los Workers respondían a ping, entonces a nivel de red los equipos sí estaban disponibles, pero las solicitudes HTTP al puerto 8080 fallaban. ¿Por qué razón? Porque llama-server no estaba ejecutándose correctamente.

Cuando se inició el proceso utilizando la ruta correcta de las bibliotecas compartidas, los Workers volvieron a aparecer como disponibles. Esto nos deja bastante claro que una cosa es tener conectividad IP y otra cosa diferente es tener disponible la aplicación o el servicio que nosotros queremos utilizar.

Estadísticas de ejemplo observadas

En una de las pruebas funcionales se obtuvo información directamente desde el endpoint /stats. En ese momento se había procesado una solicitud, Worker1 había atendido esa consulta y se registró un tiempo de respuesta de 353.87 milisegundos.

Ese resultado nos sirve para demostrar que el balanceador sí está registrando las métricas y que podemos consultar información acerca de las solicitudes procesadas, el Worker utilizado y los tiempos de respuesta.

Ahora, hay que tener cuidado con cómo presentamos este valor. Una sola ejecución no nos permite decir que el tiempo promedio del sistema es 353.87 milisegundos ni tampoco que ese va a ser siempre su rendimiento. Es simplemente una medición obtenida en una prueba puntual. Para tener un dato que realmente podamos utilizar como referencia tenemos que repetir varias consultas bajo condiciones similares y calcular las métricas a partir de ese conjunto de pruebas.

Procedimiento sugerido para repetir pruebas

1. Confirmar que ambos Workers están ejecutando llama-server.
2. Desde el Master, comprobar los endpoints /health de ambos Workers.

3. Reiniciar el balanceador solamente si se desea comenzar un conjunto de estadísticas desde cero.
4. Enviar al menos diez consultas de texto, anotando la hora y el resultado.
5. Capturar el endpoint /stats después de cada conjunto de consultas.
6. Registrar si ambos Workers recibieron solicitudes y si hubo errores.
7. Medir por separado el tiempo de respuesta observado por el balanceador y, si es posible, el tiempo total percibido por el navegador.
8. Guardar capturas y anotar la configuración usada para que el resultado pueda repetirse.

Tabla para resultados finales

Métrica	Valor medido por el equipo	Cómo medirla
Número de consultas de prueba	[Completar]	Contar las consultas enviadas durante la prueba.
Consultas atendidas por Worker1	[Completar]	Endpoint /stats del balanceador.
Consultas atendidas por Worker2	[Completar]	Endpoint /stats del balanceador.
Tiempo promedio del balanceador	[Completar] ms	Endpoint /stats después de las pruebas.
Tiempo mínimo observado	[Completar] ms	Registrar varias solicitudes individualmente.
Tiempo máximo observado	[Completar] ms	Registrar varias solicitudes individualmente.
Uso de CPU en Worker1	[Completar] %	Capturar top/htop durante inferencia.
Uso de CPU en Worker2	[Completar] %	Capturar top/htop durante inferencia.
Errores observados	[Completar]	Revisar consola del portal, balanceador y Workers.

Análisis de resultados

Básicamente, el proyecto logró cubrir los objetivos principales que se habían planteado desde el inicio. Se consiguió construir una infraestructura local, ejecutar un modelo de lenguaje sobre los equipos disponibles, distribuir solicitudes entre dos nodos de procesamiento y, finalmente, presentar todo esto al usuario por medio de una interfaz web. Las pruebas que se realizaron demuestran que, mientras los servicios estén correctamente iniciados, existe comunicación entre el portal, el balanceador y los servidores de inferencia. También se comprobó que la disponibilidad de los Workers no depende únicamente de que el equipo esté encendido o responda en la red, sino de que llama-server esté ejecutándose correctamente y pueda cargar las bibliotecas compartidas necesarias. Por eso fue importante configurar `LD_LIBRARY_PATH`, ya que sin esa configuración el servidor no podía iniciar de manera adecuada. Esto también deja una consideración importante para el despliegue: si otra persona quiere reproducir el proyecto, no basta con copiar los archivos, también tiene que conocer estas configuraciones y dependencias.

Entre las ventajas observadas, una de las más importantes es que el modelo puede ejecutarse sobre equipos físicos convencionales sin necesidad de utilizar una GPU dedicada. Esto permite demostrar que un modelo relativamente compacto sí puede funcionar sobre CPU y memoria disponibles en mini PC comunes. Además, las consultas de la demostración se procesan dentro de la infraestructura local que configuramos, lo cual permite observar directamente qué equipo atiende cada solicitud. La separación de funciones también facilita mucho la identificación de problemas, porque tenemos claramente dividido qué hace el portal, qué hace el balanceador y qué hacen los Workers. Si algo falla, podemos ir revisando cada componente por separado. El balanceador, por ejemplo, se encarga de distribuir solicitudes completas entre los dos nodos, mientras que el portal facilita la interacción del usuario y al mismo tiempo muestra el estado de los componentes. Las estadísticas básicas también son útiles porque nos permiten observar cómo se comporta el sistema durante las pruebas y relacionar los conceptos de arquitectura, red y software con algo que realmente podemos medir y ver.

Ahora, ejecutar el modelo localmente tampoco significa que eliminamos todos los costos o toda la complejidad. Básicamente, lo que hacemos es cambiar parte de esa dependencia externa por responsabilidad interna. Tenemos que adquirir o reutilizar hardware, mantener los equipos, consumir energía, instalar actualizaciones y encargarnos del soporte. También hay que tomar en cuenta que el modelo que utilizamos es más pequeño que algunos modelos disponibles en la nube,

por lo tanto su capacidad puede ser menor y en algunos casos puede tardar más en generar una respuesta. Además, al tener varios nodos aumentamos la cantidad de componentes que tenemos que iniciar, supervisar y mantener. Entonces, la solución local tiene ventajas, pero no quiere decir que automáticamente sea más simple de operar.

Con respecto al balanceo, round-robin sí nos permite demostrar que las solicitudes se pueden distribuir entre Worker1 y Worker2, pero hay que tener claro qué significa esto. No estamos haciendo un balanceo inteligente basado en la carga real de cada equipo. El algoritmo simplemente alterna entre los dos nodos. Entonces puede suceder que un Worker esté todavía ocupado con una consulta y, aun así, vuelva a recibir otra porque le corresponde dentro de la secuencia. Para una demostración académica esto está bien porque nos permite comprobar que ambos equipos participan. Sin embargo, para un sistema de producción probablemente necesitaríamos algo más completo, por ejemplo verificaciones de salud, tiempos de espera, reintentos y algún mecanismo que seleccione el nodo dependiendo de su disponibilidad o carga real.

Los tiempos de respuesta también tienen que interpretarse con cuidado. Una prueba puntual puede mostrar un valor determinado, pero ese tiempo puede cambiar dependiendo de la longitud de la pregunta, la cantidad de tokens que genere el modelo, el contexto utilizado, el estado de la memoria caché y la carga que tenga la CPU en ese momento. Entonces, no sería correcto tomar una sola medición y concluir que el sistema es más rápido o más lento que una API en la nube. Para hacer una comparación real tendríamos que repetir las mismas consultas varias veces, mantener condiciones similares y medir no solamente la latencia, sino también la calidad de las respuestas obtenidas.

Finalmente, el valor principal del prototipo no está únicamente en que el modelo pueda responder una pregunta. La parte importante es haber logrado integrar una aplicación web, un balanceador y dos nodos de inferencia dentro de una misma arquitectura funcional. Eso permite demostrar cómo interactúan los componentes, qué configuraciones fueron necesarias, cómo se puede observar el comportamiento del sistema y cuáles son las limitaciones que todavía existen. Entonces, académicamente, el proyecto no se queda solamente en ejecutar un modelo de inteligencia artificial, sino en poder explicar qué recursos utiliza, cómo se comunican los equipos, cómo se distribuyen las solicitudes, qué problemas aparecieron durante la implementación y hasta dónde puede llegar realmente la solución actual.

Seguridad, privacidad y operación

Una de las ventajas principales de ejecutar el modelo de forma local es que las consultas pueden mantenerse dentro de la infraestructura que nosotros controlamos. Es decir, mientras la aplicación no reenvíe esa información hacia un servicio externo, los datos permanecen dentro de la red del proyecto. Ahora, si esto se llevara a una empresa y se comenzara a trabajar con información confidencial, no bastaría con asumir que por ser local ya es seguro. Tendríamos que revisar el código, las dependencias que utiliza la aplicación, los registros que se están generando y las conexiones salientes para confirmar realmente qué información se queda dentro de la infraestructura y qué información podría estar saliendo.

En el tema de red sucede algo similar. En este prototipo los servicios están escuchando en determinadas interfaces y puertos porque necesitamos que el Master y los Workers puedan comunicarse dentro de la LAN. Para una demostración académica esto funciona, pero en un ambiente empresarial no podemos dejar todos esos servicios disponibles para cualquier equipo. Lo correcto sería limitar el acceso únicamente a las direcciones y puertos que realmente se necesitan, utilizando segmentación de red y reglas de firewall. Y obviamente no tendría sentido exponer directamente hacia Internet el puerto de llama-server o del servidor de inferencia, porque estaríamos ampliando innecesariamente la superficie de ataque.

La autenticación también es una limitación de la versión actual. Hoy por hoy el proyecto está diseñado para funcionar dentro de una red local de prueba, así que no estamos diciendo que ya tenemos una plataforma empresarial con gestión de identidad. Si esto evolucionara a un sistema real, necesitaríamos autenticación para saber quién está entrando, autorización para definir qué puede hacer cada usuario y probablemente roles para limitar el acceso dependiendo del tipo de información que se quiera consultar. La idea sería evitar que cualquier persona conectada a la red pueda utilizar el asistente sin ningún tipo de control.

Otro punto importante es qué hacemos con las consultas. Si vamos a guardar conversaciones, tenemos que definir cuánto tiempo se van a conservar, dónde se van a almacenar y quién va a tener acceso a ellas. Si el objetivo principal de la solución es privacidad, probablemente lo mejor sea no almacenar más información de la necesaria. Podemos conservar métricas operativas, por ejemplo tiempos de respuesta o cantidad de solicitudes, sin necesidad de guardar siempre el contenido completo de cada consulta. Si una organización decide almacenar

conversaciones, entonces tendría que existir una política clara que autorice y controle ese almacenamiento.

También hay que tener claro que el modelo no es una fuente infalible. El modelo puede responder algo incorrecto, incompleto o incluso generar información que no corresponde con la realidad. Entonces, si este asistente se utilizara para apoyar a empleados, las respuestas importantes todavía tendrían que verificarse. No sería recomendable tomar decisiones críticas únicamente con base en una salida del modelo. Y si en algún momento se integra documentación privada de la empresa, tendríamos que controlar qué documentos puede consultar el sistema y qué usuarios tienen permiso para acceder a esa información.

En cuanto a continuidad operativa, el sistema depende de que el Master y los dos Workers estén encendidos y de que todos los procesos necesarios sigan ejecutándose. Entonces, si uno de esos servicios se detiene, tenemos que tener una forma clara de detectarlo y recuperarlo. Una mejora bastante lógica sería configurar servicios con systemd, de modo que llama-server, el balanceador y el portal puedan iniciar automáticamente y, si es necesario, reiniciarse de manera controlada. También sería importante registrar errores y tener un procedimiento sencillo para comprobar la salud de cada componente antes de empezar una demostración o después de una falla.

Terminamos con el tema físico, porque aunque las mini PC sean pequeñas, siguen consumiendo energía y generando calor. Si el sistema se utilizara durante periodos prolongados, tendríamos que mantener una ventilación adecuada, limpiar periódicamente los equipos, revisar el estado de los SSD y planificar las actualizaciones del sistema operativo y de los componentes de software. Si quisiéramos evaluar realmente el costo de operar esta solución local, también tendríamos que medir el consumo eléctrico de los tres equipos y compararlo con el costo que tendría utilizar un servicio externo.

Limitaciones del prototipo

- 1. El modelo es compacto y puede ofrecer respuestas menos completas que modelos de mayor tamaño.
- 2. La inferencia depende de la CPU y de la memoria disponible; no se utiliza una GPU dedicada.
- 3. El prototipo demuestra consultas de texto, pero no implementa entrada por voz.
- 4. No se ha integrado una base documental empresarial ni un mecanismo RAG.
- 5. No existen funciones implementadas de calendario, recordatorios ni ejecución de tareas externas.
- 6. El balanceador utiliza round-robin y no selecciona nodos según carga de CPU o longitud de la cola.
- 7. El Master es un punto único de fallo para el portal y el balanceador.
- 8. Las estadísticas se limitan a contadores y tiempos registrados por el balanceador; no constituyen una plataforma completa de observabilidad.
- 9. No se ha documentado una prueba de carga formal con suficientes repeticiones para comparar rendimiento.
- 10. El portal de demostración no debe asumirse como protegido por autenticación empresarial.
- 11. La disponibilidad depende de iniciar los procesos necesarios y mantenerlos activos.
- 12. La respuesta del modelo puede contener errores y requiere validación en usos relevantes.

Alcance real frente a visión futura

Capacidad	Estado actual	Posible evolución
Consulta de texto	Disponible en portal	Mejorar interfaz y manejo de errores
Modelo local	Qwen 2.5 1.5B Q4_K_M	Evaluar otros modelos compactos
Dos nodos de inferencia	Configurados en Workers	Agregar nodos y gestión automática
Balanceo	Round-robin	Health checks, timeout, reintentos y balanceo por carga

Conclusiones

e logró implementar un prototipo de asistente de inteligencia artificial local utilizando tres mini PC conectadas por Ethernet. La idea fue separar las funciones, teniendo un equipo principal y dos nodos encargados de la inferencia, de manera que cada componente tuviera una responsabilidad clara dentro de la arquitectura.

El modelo Qwen 2.5 de 1.5 mil millones de parámetros, utilizando formato GGUF y cuantización Q4_K_M, nos permitió ejecutar pruebas de generación de texto directamente sobre el hardware disponible. Esto demuestra que, con un modelo suficientemente compacto y utilizando llama.cpp, es posible realizar inferencia sobre CPU sin depender necesariamente de una GPU dedicada.

El portal desarrollado con Flask, el balanceador round-robin y los servidores de inferencia forman básicamente un flujo cliente-servidor completo. El usuario envía una pregunta desde el navegador, la solicitud pasa por el balanceador, se procesa en uno de los Workers y finalmente la respuesta regresa al portal para mostrarse al usuario.

El endpoint de estadísticas y los indicadores del portal nos dan una visibilidad básica de lo que está sucediendo dentro del sistema. Podemos observar solicitudes exitosas, cómo se están distribuyendo entre los Workers y los tiempos registrados, lo cual nos ayuda a comprobar que los distintos componentes realmente están participando en el procesamiento.

Una de las cosas importantes que mostraron las pruebas es que tener conectividad de red no significa automáticamente que el servicio esté disponible. Un equipo puede responder a ping, pero si llama-server no está ejecutándose o si sus bibliotecas no se cargaron correctamente, entonces el servicio de inferencia no va a responder. Entonces, conectividad y disponibilidad de aplicación son dos cosas diferentes.

El prototipo demuestra que sí es viable tener una solución local para consultas de texto, pero también hay que tener claro el alcance. Hoy por hoy no tenemos voz, calendarios, recordatorios, búsqueda documental empresarial ni controles de seguridad propios de una instalación de producción. Todo eso podría agregarse después, pero no forma parte de la versión actual.

Finalmente, el proyecto permitió integrar varios conceptos que hemos trabajado durante el curso: arquitectura de computadoras, Linux, redes, programación web y ejecución de modelos de lenguaje. Más allá de que el modelo responda una consulta, lo importante es poder explicar cómo se comunica cada componente, qué recursos utiliza, qué limitaciones tiene y por qué es necesario documentar tanto los resultados como los problemas encontrados durante la implementación.